5th International Workshop on Real-Time and Cyber-Physical Cloud (RT-Cloud 2026)

RT-Cloud 2026, July 7, 2026, Lund, Sweden

Edited by

Luigi De Simone
Kevin Schmidt

Proceedings

RT-Cloud 2026 **https://www.ecrts.org/rt-cloud-2026**

Editors

Luigi De Simone
Università degli Studi di Napoli Federico II, Italy
luigi.desimone@unina.it

Kevin Schmidt
Robert Bosch GmbH, Germany
kevin.schmidt4@de.bosch.com

ACM Classification 2012

Computer systems organization → Embedded and cyber-physical systems; Computer systems organization → Real-time systems; Software and its engineering → Real-time systems software.

Publication information

These proceedings are published online as open-access informal proceedings of RT-Cloud 2026. The proceedings are compiled by the organizing committee and are not part of a formal publisher series.
Online available at https://www.ecrts.org/rt-cloud-2026.

Publication date

July, 2026

License

Contents

Accepted Papers

5th International Workshop on Real-Time and Cyber-Physical Cloud (RT-Cloud 2026).
Editors: Luigi De Simone and Kevin Schmidt
Open Proceedings
Published online by the RT-Cloud 2026 Organizing Committee

Message from the Chairs

Welcome to RT-Cloud 2026, the *5th International Workshop on Real-Time and Cyber-Physical Cloud*. It is our pleasure to present these proceedings, which bring together contributions at the intersection of real-time systems, cloud and edge computing, cyber-physical systems, control engineering, networking, and machine learning.

As cyber-physical and real-time applications increasingly rely on cloud, edge, and networked platforms, new challenges arise in combining scalability and flexibility with predictable timing, robustness, safety, and efficient resource use. RT-Cloud 2026 provides a forum for discussing these challenges and for exploring abstractions, architectures, and methods that connect real-time guarantees with modern distributed computing platforms.

The program opens with a keynote by Kevin Schmidt from Bosch, titled "Cyber-Physical Cloud: Use Cases and Challenges Driven by Physics." The keynote discusses cloud-based cyber-physical systems, including vehicle control over 5G, edge, and cloud infrastructures, and highlights challenges related to network uncertainty, stability, fallback mechanisms, and the coupling of control performance with real-time system metrics.

The technical program includes 7 accepted papers organized around two themes. The first session, "Real-Time Cloud Resource Management and Control Offloading," focuses on synchronization, scheduling, resource allocation, control offloading, criticality-aware provisioning, and cost-optimal cloud capacity management. The second session, "Real-Time Edge Platforms, AI Inference, and Security," covers real-time containers in TSN-based distributed embedded systems, machine-learning inference on edge routers, and adaptive intrusion detection in multi-tenant cloud environments.

We thank all authors for their contributions to RT-Cloud 2026. We also thank the keynote speaker, the program committee, the reviewers, the participants, and everyone involved in organizing the workshop.

The RT-Cloud 2026 Organizing Committee
Luigi De Simone, Università degli Studi di Napoli Federico II
Kevin Schmidt, Robert Bosch GmbH, Germany

5th International Workshop on Real-Time and Cyber-Physical Cloud (RT-Cloud 2026).
Editors: Luigi De Simone and Kevin Schmidt
Open Proceedings
Published online by the RT-Cloud 2026 Organizing Committee

Organizers

Workshop Chairs

Luigi De Simone (Università degli Studi di Napoli Federico II, Italy)
Kevin Schmidt (Robert Bosch Automotive Steering GmbH, Germany)

Technical Program Committee

Allan Valle (TU Kaiserslautern, Germany)
David Kozhaya (ABB Corporate Research, Switzerland)
Harald Gustafsson (Ericsson Research, Sweden)
Remo Andreoli (Sant'Anna School of Advanced Studies, Italy)
Schahram Dustdar (Vienna University of Technology, Austria)
Johan Eker (Ericsson Research, Sweden)
Sanjoy Baruah (Washington University in St. Louis, USA)
Mauro Marinoni (Sant'Anna School of Advanced Studies, Italy)
Gerhard Fohler (TU Kaiserslautern, Germany)
Tilmann Unte (University of Augsburg, Germany)
Isser Kadusale (TU Kaiserslautern, Germany)
Paul Pop (Technical University of Denmark, Denmark)
Dario Faggioli (SUSE, Italy)
Luca Abeni (Sant'Anna School of Advanced Studies, Italy)
Luiz Maia (TU Kaiserslautern, Germany)
Ahmed Al Bayati (Lund University, Sweden)
Armando Migliaccio (DigitalOcean, USA)
Pranshu Jain (VMware by Broadcom, USA)

5th International Workshop on Real-Time and Cyber-Physical Cloud (RT-Cloud 2026).
Editors: Luigi De Simone and Kevin Schmidt
Open Proceedings
Published online by the RT-Cloud 2026 Organizing Committee

List of Authors

Luca Abeni [1]

Scuola Superiore Sant'Anna, Pisa, Italy

Luis Almeida [6]

Instituto de Telecomunicações, Porto, Portugal; Faculdade de Engenharia, Universidade do Porto, Portugal

Mohammad Ashjaei [2]

Mälardalen University, Västerås, Sweden

Barikisu A. Asulba [6]

CISTER Research Centre, Porto, Portugal; Faculdade de Engenharia, Universidade do Porto, Portugal

Tommaso Cucinotta [1]

Scuola Superiore Sant'Anna, Pisa, Italy

Johan Eker [3,4]

Department of Automatic Control, Lund University, Lund, Sweden

Gerhard Fohler [3]

Chair of Real-time Systems, RPTU University Kaiserslautern-Landau, Kaiserslautern, Germany

Johan Furunäs Åkesson [2]

Westermo Network Technologies AB, Västerås, Sweden

Sathish Gopalakrishnan [7]

Department of Electrical and Computer Engineering, The University of British Columbia, Canada

Harald Gustafsson [1]

Ericsson Research, Lund, Sweden

K. P. N. Jayasena [5]

Technische Universität Dresden, Germany

Isser Kadusale [3]

Chair of Real-time Systems, RPTU University Kaiserslautern-Landau, Kaiserslautern, Germany

Saad Mubeen [2]

Mälardalen University, Västerås, Sweden

Mohammad Shahrad [7]

Department of Electrical and Computer Engineering, The University of British Columbia, Canada

Pedro F. Souto [6]

CISTER Research Centre, Porto, Portugal; Faculdade de Engenharia, Universidade do Porto, Portugal

Per Strandberg [2]

Westermo Network Technologies AB, Västerås, Sweden

Emil Sundström [3,4]

Department of Automatic Control, Lund University, Lund, Sweden

Fredrik Svensson [1]

Ericsson Research, Lund, Sweden

Muhammad Waqas [2]

Mälardalen University, Västerås, Sweden

D. S. C. Wijesinghe [5]

Department of Computing and Information Systems, Faculty of Computing, Sabaragamuwa University of Sri Lanka, Sri Lanka

5th International Workshop on Real-Time and Cyber-Physical Cloud (RT-Cloud 2026).
Editors: Luigi De Simone and Kevin Schmidt
Open Proceedings
Published online by the RT-Cloud 2026 Organizing Committee

State Synchronization for Fault-Tolerant Cloud RAN Scheduling: A Design Space Exploration

Harald Gustafsson[1], Fredrik Svensson[1], Luca Abeni[2], Tommaso Cucinotta[2]
[1]Ericsson Research, Lund, Sweden
{harald.gustafsson, fredrik.a.svensson}@ericsson.com
[2]Scuola Superiore Sant'Anna, Pisa, Italy
{luca.abeni, tommaso.cucinotta}@santannapisa.it

***Abstract*—Cloud Radio Access Network (RAN) deployments move baseband processing onto commercial off-the-shelf servers managed by cloud orchestration stacks. The MAC scheduler — the most latency-sensitive component in the baseband — is a stateful service updating its internal state thousands of times per second. Since it is required to recover from faults within tens of milliseconds, it must be able to easily synchronize its internal state with one or more replicas. This paper explores the problem of state synchronization in a Radio Access Network (RAN) scheduler considering three different aspects: *change detection* (so that only the fractions of the state that actually changed can be sent to other nodes), *state transfer*, and *redundancy model*. Several techniques for performing these three operations have been tested and compared, with different combinations implemented in a scheduler mock-up that faithfully reproduces the state layout and workload of a 5G NR or a potential 6G RAN scheduler, allowing evaluation of per-TTI synchronization overhead, dirty-data volume, and end-to-end failover latency. The experimental results show that the granularity of change detection and the latency of state transfer dominate the overhead of state synchronization. On the other hand, the choice of redundancy model primarily affects failover time rather than steady-state cost.**

***Index Terms*—Cloud RAN, fault tolerance, state synchronization, state externalization, 5G NR, 6G, RAN scheduler**

I. Introduction

Cloud and Edge computing infrastructures are increasingly being adopted for Cyber-Physical Systems (CPS) that interact with the physical environment under strict timing constraints [1], [2]. A prominent example is Cloud Radio Access Network (Cloud RAN), where baseband processing for cellular networks is deployed on commercial off-the-shelf (COTS) servers managed by Kubernetes [3]. This transition promises operational agility—decoupled hardware and software life cycles, elastic scaling, and workload consolidation—but also exposes the baseband processing stack to the failure modes of general-purpose computing infrastructures: software crashes, kernel panics, and hardware faults, that add up to the standard network-related faults due to the distributed nature of the system.

Providing fault tolerance in this kind of environment becomes particularly challenging when dealing with stateful services: while stateless services can be easily migrated, load-balanced, or re-routed across multiple instances to tolerate the failure of a single container, migrating stateful services is more complex [4]. In particular, such migrations require state from previous processing cycles; hence, a failing instance leaves its successor without the context needed to continue seamlessly.

The 5G NR [5] (New Radio) MAC scheduler is a typical example of such a stateful service, as it operates on per-user contexts used to take scheduling decisions. The MAC scheduler executes every Transmission Time Interval (TTI) of 1 ms (with sub-slot periodicity down to 125 µs), maintaining per-User Equipment (UE) state that includes HARQ process contexts, priority queue state, link adaptation parameters, and resource allocation bitmaps. A scheduler serving 100 UEs carries approximately 2 MB of mutable state that is updated every TTI (1000 times per second), or even more often. If the scheduler fails, connected UEs should recover connectivity within 50 ms to 2 seconds depending on the service class [5], making fast failover essential. These combined requirements make it challenging to implement a fault-tolerant MAC scheduler in a Cloud RAN environment.

State-of-the-art approaches to fault tolerance in cloud-native systems — such as checkpointing to distributed key-value stores [6], [7], in-memory replication through consensus protocols [8]–[10], or state management middleware [11] — introduce latencies measured in milliseconds per operation, orders of magnitude too slow to be applied at every TTI. The fundamental challenge is therefore: *how to continuously synchronize the rapidly-changing scheduler state to a standby instance with overhead low enough to fit within the 1 ms TTI, while enabling failover within the required time bounds?*

In this paper, we explore the problem of state synchronization, considering three orthogonal design dimensions:

1) **Change detection**: How to identify which parts of the state were modified during the current TTI. Different options, ranging from hardware-assisted page-level dirty bits to fine-grained software instrumentation and user-space memory diffing at cache-line granularity, are considered.
2) **State transfer**: How to transmit the modified parts of the state to the standby instance, using shared-memory ring buffers (same-host) or UDP with zero-copy (cross-host).
3) **Redundancy model**: How to organize the replicated instances. The classical primary/standby model where one instance processes all UEs is compared with a

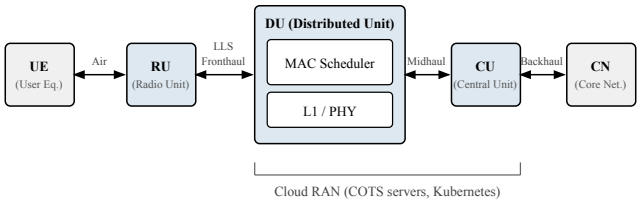


Fig. 1. 5G RAN architecture from UE to Core Network. The DU, containing the MAC scheduler, and the CU are realized as Cloud RAN on COTS servers.

sharding model where UEs are partitioned across active instances.

We implemented various combinations of these approaches in a scheduler mock-up that faithfully reproduces the state layout, access patterns, and timing of a 5G NR scheduler. The mock-up has been instrumented to measure various statistics such as the per-TTI synchronization overhead, the amount of exchanged data, and the complete failover and rejoin latency (broken down into detection, state readiness, and TTI re-synchronization).

Our contributions are:

- A structured decomposition of the design space for the fault-tolerant synchronization of the services' state.
- An implementation covering six change-detection mechanisms, three transport options, and two redundancy models with failover and rejoin capabilities.
- A comparative evaluation that quantifies the trade-offs between tracking granularity, transport latency, and redundancy-model overhead.

II. Background

A. Cloud RAN Architecture

Here we describe a 5G cellular network due to that 6G is not yet standardised, but on a high level there will be many similarities. A 5G cellular network connects User Equipment (UE) through Radio Units (RU) to baseband processing units, split into the Distributed Unit (DU) for user-plane processing and the Central Unit (CU) for control-plane functions (Fig. 1). Cloud Radio Access Network (RAN) realizes the DU and CU as containerized workloads on Commercial Off-The-Shelf (COTS) servers, managed by a cloud orchestration platform such as Kubernetes [3], [12].

Deployments range from single-server Distributed RAN (DRAN) sites co-located with antennas to multi-rack Centralized RAN (CRAN) sites serving many antenna installations. The fronthaul network between RU and DU has strict latency requirements in the order of 75–150 µs, limiting the geographic separation and making even CRAN installations small by data-center standards (typically under 10 racks). The DU is commonly deployed with direct hardware access (Single Root I/O Virtualization (SR-IOV) or userspace I/O) to Network Interface Cards (NICs) and accelerators to meet the throughput requirements of 10–100 Gbit/s.

B. A 5G New Radio (NR) or 6G MAC Scheduler

The Medium Access Control (MAC) scheduler is the central decision-making component in the DU. Every Transmission Time Interval (TTI) (1 ms), it performs a pipeline of operations:

1) *Physical-layer feedback*: Process Hybrid Automatic Repeat Request (HARQ) ACK/NACK, channel quality (Demodulation Reference Signal (DMRS)/Sounding Reference Signal (SRS)), and buffer status reports from UEs.
2) *Control-plane events*: Handle UE setup, release, and bearer reconfiguration.
3) *Resource allocation*: Run priority-based scheduling to assign Physical Resource Blocks (PRBs) to UEs for both Downlink (DL) and Uplink (UL).
4) *Output*: Emit Downlink Control Information (DCI) messages to the physical layer.

The scheduler maintains per-UE state contexts of approximately 16 KB each (cache-line aligned), comprising HARQ process state (16 processes per direction), DL and UL priority queues, link-adaptation parameters (Modulation and Coding Scheme (MCS), rank, Channel Quality Indicator (CQI)), and configuration (Radio Network Temporary Identifier (RNTI), bandwidth allocation, DMRS settings). For 100 active UEs, the total mutable state reaches approximately 1.77 MB. This state is modified extensively but sparsely every TTI: HARQ processes are updated, priority queues reordered, and link-adaptation parameters adjusted based on fresh channel feedback.

C. Fault Tolerance Requirements

When the scheduler fails, the connected UEs experience service disruption. It has been suggested for 6G to be able to reach recovery times of 50 ms for mission-critical services' traffic without requiring replication over separate networks. A replacement scheduler instance must acquire the failed instance's state—or sufficiently recent state—and resume scheduling without requiring UEs to perform a full re-establishment procedure.

The combination of (i) large state, (ii) high update rate, and (iii) sub-second recovery deadlines places this problem outside the regime of traditional cloud fault-tolerance mechanisms. Typical log-based replication (e.g., Raft [9]) and periodic checkpointing to external stores both impose latencies that exceed the 1 ms TTI budget, although some solutions does exist for smaller state sizes [13]. Application-specific state synchronization, tightly integrated with the scheduler's execution model, is therefore required.

D. Related Work

1) State management in distributed systems: Common approaches to state management in cloud applications range from centralized stores (relational databases, key-value stores such as BerkeleyDB [14]) to distributed NoSQL solutions using consensus protocols such as Paxos [15] or Raft [9]

for replication across fault-independent zones (e.g., MongoDB [16], DynamoDB [6], BigTable [7]). These solutions require at least one disk synchronization in the critical path, limiting throughput [17]. In-memory stores such as Redis [18], building on Viewstamped Replication [10], [19], avoid the disk bottleneck but still impose consensus-round latencies measured in hundreds of microseconds to milliseconds — too slow for the per-TTI budget. Solutions such as FASTER [8] and the state management layer of Szalay et al. [11] reduce state-access latency by co-locating state with compute, but target request–response workloads rather than periodic bulk synchronization. Liu et al. [20] address state management for Actor-model systems, but again at per-message granularity.

2) Fault Tolerance based on Virtual Machines: Projects like Remus [21] and COLO [22] provide fault tolerance to software running inside a VM by taking advantage of the VM migration and checkpointing features. Since these approaches operate at the granularity of entire VMs, they impose an overhead measured in milliseconds per checkpoint — incompatible with a 1 ms TTI.

3) Network function fault tolerance: FTMB [23] provides rollback-recovery for stateful middleboxes by logging packet-level state changes, but assumes per-packet state granularity and does not address the periodic batch-processing model of a MAC scheduler. Pico Replication [24] similarly targets per-packet processing. StateSafe [25] externalizes the network function state to a shared store, but targets microsecond-scale operations per state access, not bulk synchronization of megabytes per TTI.

4) Fault tolerance for time-critical cloud services: Abeni et al. [26] introduced a model for fault tolerance in real-time cloud computing, analyzing the trade-offs between replication overhead and recovery latency for time-critical services. In prior work [4], we extended this model to account for stateful microservices and investigated some policies for state handling (centralized stores vs. peer-to-peer replication), evaluating the performance impact through simulation. The present paper builds on these insights by moving from simulation to a concrete implementation of the state synchronization mechanisms for the specific case of a RAN scheduler, and by systematically exploring different alternatives for change detection, transport, and redundancy model.

To our knowledge, no prior work has systematically explored the design space for state synchronization of a real-time MAC scheduler at the granularity presented here.

III. Design Space

We decompose the state synchronization problem into three orthogonal dimensions, each admitting multiple design points with distinct trade-offs. Fig. 2 illustrates the three dimensions and their interactions. This paper has focus on the change detection and redundancy models, there are several other state transfer mechanisms that could be explored. Further, we could have also explored other redundancy models such as active/active or other full replication strategies; or primary/cold-standby with a reduced state replication. These where deemed less interesting for the RAN scheduler, due to higher cost or longer recovery times.

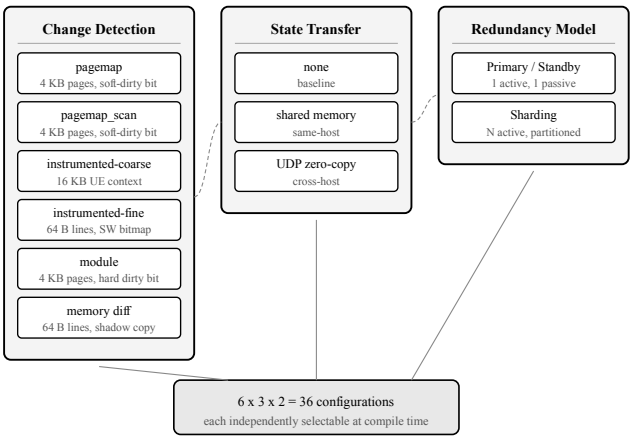


Fig. 2. The three-dimensional design space for scheduler state synchronization. Each dimension can be chosen independently, yielding 36 potential configurations.

A. Change Detection

The scheduler state is large but only a fraction changes during any single TTI. Efficient synchronization requires identifying the dirty subset. We consider six mechanisms spanning the spectrum from zero-instrumentation hardware tracking to fine-grained software tracking. There are other methods that could have been explored based on e.g., copy-on-write or disaggregated memory, but they have not been explored due to deemed less interesting.

1) Page-level hardware tracking (pagemap): Linux exposes per-page soft-dirty bits through `/proc/self/pagemap`. After clearing the bits, any write to a page sets its soft-dirty flag, which can be read at the end of the TTI. *Granularity*: 4 KB pages. *Overhead*: A page fault is triggered for a write, which sets the soft-dirty bit. One system call per TTI to read dirty bits and one to clear. Moreover, when the dirty bits are cleared the MMU's Translation Lookaside Buffer (TLB) has to be invalidated (this operation is known as *TLB flush* and can slow down the future accesses to the state buffer). *Trade-off*: The coarse granularity means that a single byte written to a 4 KB page marks the entire page as dirty, potentially over-reporting by orders of magnitude.

2) Instrumented tracking (instrumented): A bitmap is maintained with one bit per 64-byte cache line in the state region. Every write to scheduler state is instrumented with a marking macro that sets the corresponding bits. The tracker supports two granularity modes, selectable at compile time:

Coarse-grained: A single macro marks the entire UE context (≈16 KB) on any write. This requires minimal instrumentation, one mark per scheduling phase per UE, but reports substantially more data than actually changed.

Fine-grained: Specialized macros target individual cache-line-aligned sub-structures within the UE context. This

achieves the minimum practical dirty volume at the cost of instrumenting every write site in the scheduler code.

Overhead: One bitmap-set operation per mark; bitmap scan at TTI boundary. *Trade-off*: Fine-grained mode reduces dirty bytes by an order of magnitude compared to coarse, but requires extensive source-level modifications.

3) Page-scan hardware tracking (pagemap_scan): Uses the `PAGEMAP_SCAN` ioctl (Linux 6.7+) to batch-walk page tables and report dirty pages. *Granularity*: 4 KB pages (same as pagemap). *Overhead*: A page fault is triggered for a write, which sets the soft-dirty bit. Single ioctl per TTI instead of read and clear as for pagemap, but a TLB flush is still needed. *Trade-off*: Requires a recent kernel, same granularity limitations as pagemap method.

4) Kernel-module tracking (module): A custom kernel module exposes a character device (`/dev/dirty_track`) with an ioctl interface. Unlike pagemap and pagemap_scan, which rely on the kernel's soft-dirty bit, the module directly inspects hardware page-table dirty bits. Since hardware tracking, this method does not introduce write page faults. *Granularity*: 4 KB pages (same as pagemap). *Overhead*: The ioctl system call and the TLB flush needed when resetting the hardware dirty bits. *Trade-off*: Requires deploying a custom kernel module, which may conflict with cloud platform policies that restrict kernel modifications.

5) User-space memory diff (memory diff): A shadow copy of the entire state region is maintained in user space. At TTI end, the tracker compares the live state against the shadow at cache-line (64-byte) granularity to identify the dirty lines. The shadow is then updated to match the live state. *Granularity*: 64 bytes (effective), detected via byte-level comparison. *Overhead*: One full memory comparison per TTI. *Trade-off*: Achieves cache-line output granularity without any source-code modifications, but pays the cost of comparing the entire state region every TTI.

B. State Transfer

Once dirty regions are identified, they must be transmitted to the standby instance. We consider three transports covering same-host and cross-host deployments.

1) No synchronization (none): Dirty bytes are counted but not transmitted. This serves as the baseline to isolate tracking overhead from transport overhead.

2) Shared memory (shm): A POSIX shared-memory ring buffer connects instances on the same host. The sender writes a header (TTI number, region count, total bytes, timestamp) followed by region descriptors and data.

3) UDP with zero-copy (udp): Dirty regions are sent as UDP datagrams cross-hosts using `SO_ZEROCOPY` scatter-gather. Large payloads are chunked with continuation headers.

C. Redundancy Model

The redundancy model determines how UE ownership is distributed across instances and how failover is orchestrated.

1) Primary/Standby: One instance (the primary) processes all UEs and synchronizes its full dirty state every TTI. A standby instance receives state updates and monitors the primary via heartbeats. On primary failure, the standby detects the timeout, takes ownership of all UEs, and resumes scheduling.

Advantages: Simple to reason about; single writer for all state. *Disadvantages*: Standby resources are idle during normal operation; full state must be synchronized every TTI.

2) Sharding: UEs are partitioned across N active instances, each responsible for scheduling a subset. Each shard synchronizes only its owned UEs' state to peers. On failure, a leader (lowest surviving instance ID) redistributes the failed shard's UEs among survivors.

Advantages: All instances are productive; less data synchronized per instance; potentially faster failover since survivors already have their own state hot. *Disadvantages*: Requires distributed ownership management; redistribution adds complexity.

D. Fault Detection and UE Ownership

Orthogonal to the three main dimensions, fault detection and ownership management are supporting mechanisms required by both redundancy models.

Fault detection uses periodic heartbeats—either through shared memory (same-host) or UDP multicast (cross-host)—with configurable timeout (default 10 ms). The detecting instance triggers failover and self-fencing prevents a partitioned-but-alive instance from resuming.

UE ownership is tracked per-UE using a seqlock protocol in shared memory (same-host) or leader-broadcast over UDP multicast (cross-host). The seqlock design ensures that the TTI thread is never blocked: if a concurrent ownership update is in progress (odd sequence number), the scheduler simply skips that UE for the current TTI. A UE ownership update is only performed during failover redistribution or setup and release of a UE.

E. Rejoin and Recovery

A failed instance that restarts—or a new instance that is added—must be reintegrated into the system without disrupting ongoing scheduling. The rejoin mechanism is the inverse of failover. During a rejoin the sharding leader intentionally delays redistribution of UEs to allow state synchronisation to complete on the rejoining peer. UEs continue to be scheduled by their current owners throughout, so the rebalance does not constitute a scheduling gap beyond one TTI in worst case.

In primary/standby mode, a rejoining instance resumes its standby role and begins receiving the primary's dirty-state stream after the initial paced forced state synchronisation. In sharding mode, the leader calls a redistribution that moves a subset of UEs from the surviving shards to the rejoining instance, restoring the original partitioning. The same ownership protocol (seqlock or leader broadcast) is used, ensuring the TTI thread on active instances is never blocked during the transition.

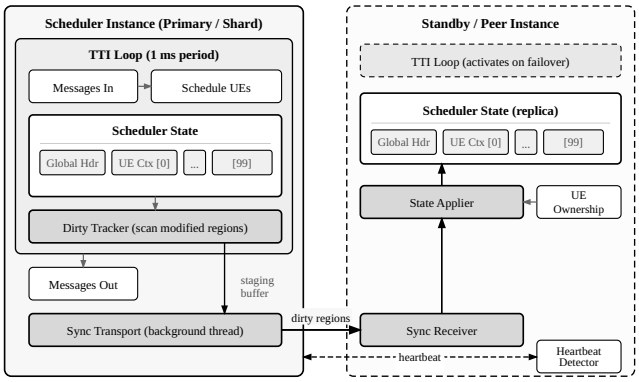


Fig. 3. Scheduler mock-up architecture. The primary instance runs the 1 ms TTI loop and synchronizes dirty state regions to the standby via a pluggable transport. The state applier on the standby checks UE ownership before merging updates.

Rejoin is important for system resilience: without it, each successive failure permanently reduces the number of available instances. Combined with failover, the system can tolerate repeated failure–recovery cycles while maintaining continuous scheduling. To make it likely not losing state, the number of shards needs to be adapted to the reliability of them and the time it takes for full recovery of the state on a rejoining shard.

IV. Implementation

We implemented the design space in a scheduler mock-up written in C++ that reproduces the structure and workload of a real RAN scheduler; Fig. 3 shows its overall architecture. Although this mock-up does not implement the exact algorithms used by a real 5G or 6G scheduler, it has been verified that its memory footprint, activation pattern, and CPU usage match those of the real application.

A. Scheduler Mock-up

The mock-up implements the four-phase TTI pipeline (Section II) with a faithful reproduction of the state layout. The `SchedulerState` structure (1.77 MB, cache-line aligned) contains a global header (576 bytes), 100 UE contexts of 16 KB each, and 10 random-access (MsgB) contexts of 16 KB each. Each UE context contains HARQ process arrays, priority queues, link-adaptation parameters, and configuration—matching the field layout and access patterns of a real scheduler.

A synthetic message generator produces realistic traffic patterns: HARQ ACK/NACKs, DMRS/SRS reports, buffer status reports, and control-plane events (UE setup/release/reconfiguration). The relative frequencies and per-UE arrival rates match realistic environments.

B. Asynchronous Synchronization Pipeline

To avoid adding synchronization latency to the TTI critical path, we decouple state capture from network transmission using an asynchronous pipeline:

1) At TTI end, the dirty tracker scans for modified regions and produces a list of (offset, size) pairs.
2) The TTI thread copies dirty data into a lock-free four-slot staging buffer in a single pass.
3) A background drain thread picks up staged datagrams and calls the transport's send function.

The wire format consists of a `SyncMessageHeader` (TTI number, region count, total bytes, timestamp), followed by `RegionDescriptor` entries and raw data. Payloads exceeding 65 KB are split into continuation chunks.

C. Failover and Rejoin Handling

When the fault detector reports a peer failure, the mock-up executes a failover sequence: (1) leader election (lowest alive instance ID), (2) the leader redistributes ownership of the failed instance's UEs among survivors and broadcasts the new ownership table, (3) each survivor merges the received state for its newly acquired UEs and rebuilds its active UE list, (4) scheduling continuous and the new UEs are incorporated on the next TTI boundary.

For rejoin, the heartbeat detector identifies a fresh heartbeat from a previously-failed peer. When an instance rejoins it needs to (1) get a full state and not only the changes, so other instances triggers a forced paced state sync over at most 50 ms when receiving heartbeat from a new instance. This needs to be completed before the instance is ready, and (2) signals that in heartbeats. Otherwise, the rejoining instance would have ownership of UEs it miss state for and would prevent applying incoming latest state updates. Similarly, in sharding mode a (3) rejoining instance will not take leadership, i.e. responsibility for deciding which UE is scheduled by which instance, before the other instances has provided it with the current mapping. (4) The leader redistributes UEs to include the rejoining instance and broadcasts the updated ownership. (5) Non-leader instances apply the ownership broadcast and release UEs that have been reassigned, ensuring no scheduling gap or duplication during the transition.

D. Change Detection Implementation

The five change-detection mechanisms are implemented as subclasses of a common `DirtyTrackerBase` interface, selected at compile time.

Pagemap and pagemap_scan both rely on the kernel's soft-dirty bit infrastructure. Pagemap reads `/proc/self/pagemap` and clears via a write to `/proc/self/clear_refs`; pagemap_scan uses the `PAGEMAP_SCAN` ioctl which combines scanning and clearing in a single system call.

Instrumented tracking uses a compile-time granularity flag (`GRANULARITY=coarse` or `fine`). In coarse mode, a single `MARK_DIRTY_UE` macro marks the full 16 KB context on any write. In fine mode, 12 specialized macros target cache-line-aligned sub-structures. A global pointer to the tracker is used to avoid virtual dispatch in the hot path.

Module tracking uses hardware page-table dirty bits (PTE dirty flag) instead of soft-dirty bits. The kernel mod-

ule walks page tables via `pte_dirty()` and clears with `pte_mkclean()`, avoiding the expensive `clear_refs` and the page faults during processing writes.

Memory diff maintains a page-aligned shadow copy of the full state region. At each TTI boundary, it performs a cache-line-level comparison (using SIMD-accelerated comparison) to report only the 64-byte regions that differ. The shadow is updated in place during scanning.

E. Build-time Configuration

Each design dimension is selected at compile time through Makefile variables (`TRACK`, `SYNC`, `FAULT`, `OWNERSHIP`, `MODE`), with an additional `GRANULARITY` variable for the instrumented tracker. This enables the compiler to eliminate virtual dispatch from the hot path and making sure that each benchmark configuration is independent.

V. Evaluation

A. Experimental Setup

Experiments are conducted on a dual-socket AMD EPYC 9654 (2×96 cores), 768 GB RAM, equipped with Mellanox ConnectX-7 NICs and a 100 Gb/s switch connecting them. The system runs Ubuntu 24.04 with Linux kernel 6.8 and the `realtime` tuned profile. Low-latency optimizations include: isolated CPUs with adaptive-tick scheduling on 188 of 192 cores, IRQs and kernel threads pinned to the 4 housekeeping cores, NMI watchdog disabled, transparent huge pages off, processor C-states limited to C1, PCIe ASPM disabled, and 64 GB of 1 GB huge pages pre-allocated. We validated the real-time isolation with `cyclictest` which reported 2 µs average and 21 µs maximum wake-up latency, confirming that any overhead exceeding this baseline is attributable to the mechanisms themselves rather than system jitter.

B. Change Detection

The scheduler mock-up is configured with two workload sizes: a *small* configuration with 50 active UEs out of 100 potential UEs (1.77 MB state region), and a *large* configuration with 400 active UEs out of 500 potential UEs (8.4 MB state region). The scheduler mock-up is pinned to isolated cores and uses `SCHED_FIFO` for the TTI-loop thread. Each experiment runs for 59 seconds (59 000 TTIs) after a warm-up period of 1 000 TTIs.

Bottom row group in Table I and II compare the dirty-data volume reported by each mechanism at two workload sizes.

At 50 UEs, the page-level methods (pagemap, pagemap_scan, module) report 363–410 KB/TTI on average, reflecting the 4 KB granularity: a single byte write marks an entire page. Instrumented-coarse reports even more (656 KB), because each scheduling phase marks the full 16 KB UE context regardless of how many fields changed. In contrast, instrumented-fine reports only 24 KB/TTI and memory diff 9 KB/TTI—reductions of 94% and 98% respectively over page-level tracking.

At 400 UEs, the distinction sharpens dramatically. Instrumented-coarse reaches 5.1 MB/TTI because every scheduled UE's full 16 KB context is marked. Pagemap reports 4.8 MB/TTI, but this is inflated because its overhead exceeds the TTI budget (it only reaches TTI 11 000 out of 59 000): messages back up between TTIs, causing more state to be modified per processed TTI. Pagemap_scan (reaching TTI 52 000) similarly over-reports at 2.7 MB/TTI due to partial TTI overruns. Module, which completes the full run, reports 2.6 MB/TTI—the true page-level dirty volume at this workload. The fine-grained methods scale much better: instrumented-fine at 180 KB/TTI and memory diff at 67 KB/TTI, representing reductions of 93% and 97% respectively over the module baseline.

Tables I and II break down the per-TTI overhead into four phases: processing (TTI loop execution including page fault overhead for kernel-based methods), scan (identifying dirty regions), send (staging data for transport), and clear (resetting dirty bits for the next TTI).

The dominant observation is that pagemap is impractical at scale. At 50 UEs, its clear phase alone costs 1074 µs—already exceeding the 1 ms TTI budget—and its total overhead reaches 1459 µs. At 400 UEs the situation worsens dramatically: pagemap's total overhead reaches 5264 µs (more than 5× the TTI budget).

The root cause of the pagemap latency spikes is a TLB shootdown: `clear_refs` write-protects pages so that future writes re-trigger the soft-dirty fault, which requires invalidating cached page table entries on *all* CPUs via inter-processor interrupts (IPIs). Monitoring the `tlb:tlb_flush` tracepoint confirms this—pagemap causes ≈3 850 TLB flushes per second versus ≈34 (background noise) for instrumented-fine, a 113× increase.

Furthermore, `clear_refs` holds the kernel's `mmap_lock` in write mode for the duration of the page table walk and TLB invalidation. Tracing with BPF confirms that any concurrent page fault on the same process—including faults triggered during the send phase's `memcpy`—blocks on this lock until `clear_refs` completes. This manifests as sporadic send-phase spikes that are mutually exclusive with the clear-phase spikes: per-TTI instrumentation shows that when send spikes, clear is normal, and vice versa, confirming that both are victims of the same lock being held for the full page-table walk duration.

Pagemap_scan eliminates the clear cost by combining scan and clear into a single ioctl. At 50 UEs it achieves 155 µs total, but at 400 UEs it still reaches 1107 µs due to elevated processing (885 µs from page faults) and send (171 µs) phases.

The instrumented methods avoid kernel interaction entirely. Fine-grained tracking totals only 16 µs (50 UEs) and 124 µs (400 UEs), with scan cost growing moderately as the bitmap is larger. Coarse-grained instrumented tracking is cheap per scan (59 µs at 50 UEs) but its send phase grows to 268 µs at 400 UEs due to the large volume of data staged for transport, giving a total of 401 µs.

The module tracker achieves 41 µs (50 UEs) and 302 µs

TABLE I

PER-TTI OVERHEAD BROKEN DOWN BY PHASE AND DIRTY BYTES (50 ACTIVE UES) OVER 60 S. PAGEMAP'S CLEAR PHASE EXCEEDS THE 1 MS TTI BUDGET ALONE AND THE METHOD OVERRUNS THE TTI PERIOD. INSTRUMENTED-FINE AND MEMORY DIFF REPORT ORDERS OF MAGNITUDE LESS DATA THAN PAGE-LEVEL METHODS.

Category		pagemap	instrumented (coarse)	instrumented (fine)	pagemap_scan	module	memory diff
Processing	max	243.0 µs	12.0 µs	13.0 µs	211.0 µs	17.0 µs	10.0 µs
	avg	**138.0 µs**	**4.00 µs**	**5.00 µs**	**121.0 µs**	**9.00 µs**	**4.00 µs**
	min	6.00 µs	3.00 µs	2.00 µs	20.0 µs	3.00 µs	2.00 µs
Scan	max	44.0 µs	24.0 µs	18.0 µs	107.0 µs	16.0 µs	72.0 µs
	avg	**19.0 µs**	**14.0 µs**	**10.0 µs**	**11.0 µs**	**9.00 µs**	**58.0 µs**
	min	18.0 µs	9.00 µs	1.00 µs	6.00 µs	6.00 µs	56.0 µs
Send	max	535.0 µs	196.0 µs	84.0 µs	240.0 µs	248.0 µs	153.0 µs
	avg	**228.0 µs**	**41.0 µs**	**1.00 µs**	**23.0 µs**	**23.0 µs**	**0.00 µs**
	min	6.00 µs	17.0 µs	0.00 µs	2.00 µs	3.00 µs	0.00 µs
Clear	max	1113 µs	0.00 µs	0.00 µs	0.00 µs	0.00 µs	0.00 µs
	avg	**1074 µs**	**0.00 µs**	**0.00 µs**	**0.00 µs**	**0.00 µs**	**0.00 µs**
	min	1068 µs	0.00 µs	0.00 µs	0.00 µs	0.00 µs	0.00 µs
Total	**avg**	**1459 µs**	**59.0 µs**	**16.0 µs**	**155.0 µs**	**41.0 µs**	**62.0 µs**
Dirty bytes	max	712.7 kB	1064.6 kB	77.6 kB	647.2 kB	651.3 kB	19.9 kB
	avg	**410.1 kB**	**656.8 kB**	**23.8 kB**	**363.3 kB**	**363.2 kB**	**9.4 kB**
	min	4.1 kB	387.2 kB	2.2 kB	53.2 kB	53.2 kB	1.3 kB

TABLE II

PER-TTI OVERHEAD BROKEN DOWN BY PHASE AND DIRTY BYTES (400 ACTIVE UES) OVER 60 S. PAGEMAP AND PAGEMAP SCAN TOTAL OVERHEAD EXCEEDS 1 MS, AND FOR SOME TTIS ALSO INSTRUMENTED (COARSE) OVERRUNS THE TTI PERIOD. INSTRUMENTED-FINE REMAINS THE LOWEST OVERHEAD IN TOTAL.

Category		pagemap	instrumented (coarse)	instrumented (fine)	pagemap_scan	module	memory diff
Processing	max	1857 µs	47.0 µs	93.0 µs	1556 µs	116.0 µs	42.0 µs
	avg	**1556 µs**	**27.0 µs**	**41.0 µs**	**884.5 µs**	**76.0 µs**	**23.0 µs**
	min	1265 µs	21.0 µs	27.0 µs	19.0 µs	23.0 µs	18.0 µs
Scan	max	105.0 µs	137.0 µs	121.0 µs	313.0 µs	56.0 µs	300.0 µs
	avg	**78.0 µs**	**106.0 µs**	**78.0 µs**	**51.0 µs**	**44.0 µs**	**286.0 µs**
	min	74.0 µs	75.0 µs	43.0 µs	10.0 µs	26.0 µs	281.0 µs
Send	max	4495 µs	1050 µs	104.0 µs	1859 µs	1643 µs	70.0 µs
	avg	**2486 µs**	**268.0 µs**	**5.00 µs**	**171.0 µs**	**182.0 µs**	**2.00 µs**
	min	878.0 µs	134.0 µs	1.00 µs	6.00 µs	43.0 µs	1.00 µs
Clear	max	1283 µs	1.00 µs	1.00 µs	6.00 µs	0.00 µs	0.00 µs
	avg	**1144 µs**	**0.00 µs**	**0.00 µs**	**0.00 µs**	**0.00 µs**	**0.00 µs**
	min	1139 µs	0.00 µs	0.00 µs	0.00 µs	0.00 µs	0.00 µs
Total	**avg**	**5264 µs**	**401.0 µs**	**124.0 µs**	**1106 µs**	**302.0 µs**	**311.0 µs**
Dirty bytes	max	5709.8 kB	6790.0 kB	344.8 kB	4747.3 kB	4436.0 kB	135.9 kB
	avg	**4751.4 kB**	**5053.1 kB**	**179.3 kB**	**2670.4 kB**	**2613.9 kB**	**67.2 kB**
	min	3813.4 kB	3274.1 kB	94.0 kB	90.1 kB	548.9 kB	39.9 kB

(400 UEs), benefiting from the no-op clear since the hard dirty bit is handled in-kernel during the scan, though ioctl calls overhead in the processing phase (76 µs) and send costs (182 µs) grow with UE count. In particular, the large send times are due to the TLB invalidation, that requires the MMU to reload many page table entries at each TTI (this same effect is visible for pagemap_scan too).

Memory diff shows a clear scaling trade-off: its scan phase grows from 58 µs (50 UEs) to 286 µs (400 UEs) as the shadow comparison must cover the larger state region. However, its send phase remains negligible (2 µs) because only the few actually-changed cache lines are transmitted, giving a total of 311 µs at 400 UEs.

C. Redundancy Model and Failover

To benchmark the scheduler mock-ups failover latency, a 2 instance setup on the same host is created. The instances are pinned to individual isolated cores. Table III breaks down failover latency by phase for both redundancy models across deployment profiles. As can be seen the majority of the time is due to the set heartbeat detection timeout of 10 ms. This time could be reduced based on the expected worst case communication latency. It can be seen a slight increase for using UDP networking instead of shared memory, although this difference would likely be larger when using redundancy over multiple hosts. The interesting number is the state ready values measuring the time from failover is detected until the

TABLE III
FAILOVER LATENCY FOR 50 UEs (MEAN ± 95% CI, μS, n=30). DETECT: HEARTBEAT TIMEOUT. STATE READY: OWNERSHIP TRANSFER AND STATE MERGE. TTI SYNC: WAIT FOR NEXT MS-ALIGNED BOUNDARY.

Model	Transport	Detect	State ready	TTI sync
PS	L	10085 ± 11	83 ± 3	902 ± 11
S	L	10077 ± 14	27 ± 2	884 ± 14
PS	N	10116 ± 15	108 ± 5	639 ± 21
S	N	10106 ± 14	28 ± 1	710 ± 159

PS = primary/standby, S = sharding, L = shared mem, N = UDP

TABLE IV
ROBUSTNESS TEST OF REPEATED FAILURE OF INSTANCES AND THE DURATION A UE IS UNSCHEDULED.

Mode	n	Mean (TTIs)	95% CI	Max (TTIs)
Primary/Standby	34	11.2	±0.07	12.1
Sharding	73	11.2	±0.24	20.0

ownership of the migrating UEs and their latest state is merged into the surviving instances state. As can be seen all these numbers are much smaller than one TTI of 1 ms. Sharding redundancy model has a slight advantage due to only one third of the UEs need to migrate and the surviving instance is already running the scheduling TTI loop. The state is then available when the next synchronized TTI will start.

D. Robustness Under Repeated Failure–Rejoin Cycles

To evaluate the performance over longer runs with repeated failure and rejoins of instances we run the test for 5 minutes with the schedulers in containers using the UDP networking[1]. Table IV summarises the scheduling gap observed during failover events across both redundancy modes. The gap is defined as the number of consecutive TTIs during which affected UEs are not scheduled, measured from the last heartbeat of the failed instance to the first TTI processed by the takeover instance.

In both modes the mean failover gap is 11.2 TTIs (ms), dominated by the 10 ms heartbeat timeout used for failure detection. This confirms the finding in previous section. Primary/standby exhibits a tighter confidence interval (±0.07) because all UEs fail over atomically to a single standby that already holds a complete state replica. In sharding mode the gap variance is higher (±0.24, max 20.0 TTIs) since each surviving instance independently detects the failure and takes over a subset of UEs; slight differences in heartbeat phase across survivors widen the distribution.

Figures 4 and 5 illustrate the observed per-instance view of UE ownership and state availability across multiple kill/restart cycles. In primary/standby mode (Fig. 4), the standby receives a continuous state replica (shown as half-height dashed bars) within 50 ms of startup, enabling sub-millisecond takeover once failure is detected. In sharding mode (Fig. 5), each

[1]We did not consider more advanced networking technologies such as TSN because we plan to move to different mechanisms based on RDMA.

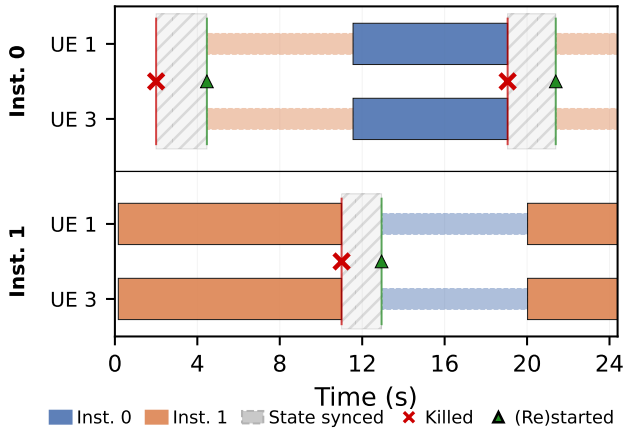


Fig. 4. UE ownership timeline in primary/standby mode across three kill/restart cycles. Solid bars indicate ownership (the instance schedules the UE); half-height dashed bars indicate synced state (replica held for failover readiness).

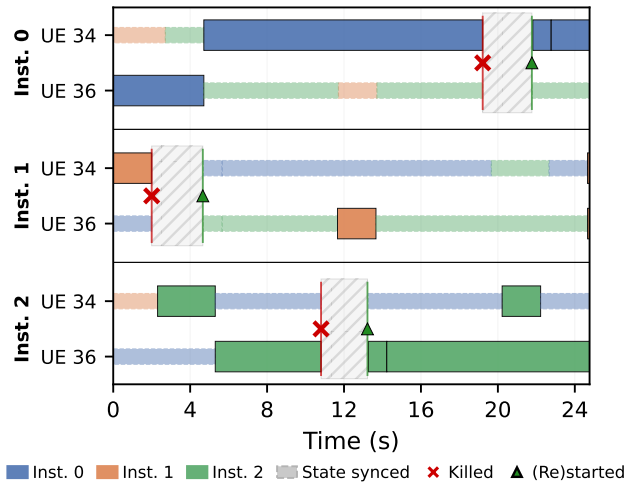


Fig. 5. UE ownership timeline in sharding mode (3 instances) across three kill/restart cycles. Each instance owns a partition of UEs (solid bars) while holding synced replicas of peers' UEs (half-height dashed bars).

instance holds synced state for UEs owned by peers. After a kill event, the surviving instances immediately assume ownership of the orphaned UEs, and a rejoining instance receives state before the leader redistributes ownership back. Notice that the simple replication and sharding mechanisms implemented in this mock-up only focus on state availability and correctness, and can cause some redundant ownership transfer (for example, look at what happens in Fig. 5 after instance 1 fails: the ownership of UE34 is transferred first to instance 2, and then to instance 0 - although these transfer look strange, they do not affect the service availability). This issue will be addressed as future work, as it has no impact on the current evaluation.

VI. Discussion

Pagemap's reliance on `clear_refs`—which acquires `mmap_lock`, walks the full page table, and triggers cross-CPU TLB shootdowns—makes it exceed the TTI budget even at 50 UEs. This is not merely a performance issue but a correctness one: the lock blocks concurrent page faults, causing mutual interference between the clear and send phases. Pagemap_scan mitigates the clear cost but still suffers from page-fault overhead at scale. The module tracker's use of hard dirty bits avoids page faults entirely, demonstrating that kernel-level tracking is viable if the soft-dirty infrastructure is bypassed. Unfortunately, both pagemap_scan and the custom module need to resort to a TLB flush, whose effects are visible in the "send" times.

Fine-grained methods (instrumented-fine, memory diff) report 93–98% less data than page-level methods and, critically, their dirty volume grows with *activity* rather than *state size*. This means the synchronization cost is largely independent of UE count—a property essential for scaling to denser deployments without proportionally increasing transport bandwidth. Coarse instrumentation, despite being software-based, can produce *more* dirty data than page-level tracking when contexts are large relative to actual modifications. The dirty tracking and staging any changes are overhead in the TTI loop and will reduce the time available for scheduling. It would be beneficial to be able to preempt such work before the next TTI start during periods when the scheduling has a short margin to the next TTI period.

State readiness completes in 27–108 µs—under 11% of a single TTI—confirming that continuous synchronization keeps replicas fresh enough for immediate takeover. The 10 ms heartbeat timeout dominates end-to-end failover, suggesting that investment in faster failure detection (hardware watchdogs, in-band heartbeats) would yield larger improvements than optimizing the state-transfer path further.

VII. Conclusion

We presented a systematic design space exploration for fault-tolerant state synchronization of a Cloud RAN MAC scheduler. By decomposing the problem into change detection, state transfer, and redundancy model, we implemented and compared 36 configurations in a scheduler mock-up faithful to a real RAN scheduler state layout and workload.

The key findings are: (1) the soft-dirty pagemap mechanism is fundamentally unsuitable for sub-millisecond TTIs due to page faults, TLB shootdowns and `mmap_lock` contention—it exceeds the 1 ms budget even at 50 UEs; (2) fine-grained change detection (instrumented-fine, memory diff) reduces dirty data by 93–98% over page-level methods, with volume scaling with activity rather than state size; (3) a custom kernel module using hard dirty bits avoids the soft-dirty page fault overhead while maintaining transparent tracking, at the cost of an increased send time (due to TLB invalidation); (4) failover latency is dominated by heartbeat detection time (10 ms), with state readiness completing in under 108 µs across all configurations.

Future work includes prioritisation and preemption of the state externalisation pipeline to avoid consuming TTI budget margin during high-load periods, scaling to larger deployments with thousands of UEs, and evaluating kernel-bypass transports such as RDMA and AF_XDP for cross-host synchronization.

References

[1] R. Andreoli, R. Mini, P. Skarin, H. Gustafsson, J. Harmatos, L. Abeni, and T. Cucinotta, "A multi-domain survey on time-criticality in cloud computing," *IEEE Transactions on Services Computing*, p. 1–19, 2025. [Online]. Available: http://dx.doi.org/10.1109/TSC.2025.3539197

[2] R. Buyya, S. N. Srirama, G. Casale, R. Calheiros, Y. Simmhan, B. Varghese, E. Gelenbe, B. Javadi, L. M. Vaquero, M. A. S. Netto, A. N. Toosi, M. A. Rodriguez, I. M. Llorente, S. D. C. D. Vimercati, P. Samarati, D. Milojicic, C. Varela, R. Bahsoon, M. D. D. Assuncao, O. Rana, W. Zhou, H. Jin, W. Gentzsch, A. Y. Zomaya, and H. Shen, "A manifesto for future generation cloud computing: Research directions for the next decade," *ACM Comput. Surv.*, vol. 51, no. 5, Nov. 2018. [Online]. Available: https://doi.org/10.1145/3241737

[3] Ericsson, "Cloud ran verified on cloud native solution," Ericsson Blog, 2023. [Online]. Available: https://www.ericsson.com/en/blog/2023/11/one-hand-shake-cloudran

[4] L. Abeni, H. Gustafsson, F. Svensson, and T. Cucinotta, "State management in real-time cloud and nfv services," in *2025 IEEE Conference on Network Function Virtualization and Software-Defined Networking (NFV-SDN)*, 2025, pp. 1–7.

[5] 3GPP, "Nr; nr and ng-ran overall description; stage 2," 3GPP TS 38.300.

[6] S. Perianayagam, A. Vig, D. Terry, S. Sivasubramanian, J. C. S. III, A. Mritunjai, J. Idziorek, N. Gallagher, M. Elhemali, N. Gordon, R. Krog, C. Lazier, E. Mo, T. Rath, and S. Sosothikul, "Amazon DynamoDB: A Scalable, Predictably Performant, and Fully Managed NoSQL Database Service," in *USENIX Annual Technical Conference*, Carlsbad, CA, 2022, pp. 1037–1048. [Online]. Available: https://www.usenix.org/conference/atc22/presentation/vig

[7] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable: A distributed storage system for structured data," *ACM Trans. Comput. Syst.*, vol. 26, no. 2, Jun. 2008. [Online]. Available: https://doi.org/10.1145/1365815.1365816

[8] B. Chandramouli, G. Prasaad, D. Kossmann, J. Levandoski, J. Hunter, and M. Barnett, "Faster: A concurrent key-value store with in-place updates," in *Proceedings of the 2018 International Conference on Management of Data*, ser. SIGMOD '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 275–290. [Online]. Available: https://doi.org/10.1145/3183713.3196898

[9] D. Ongaro and J. Ousterhout, "In search of an understandable consensus algorithm," in *2014 USENIX annual technical conference (USENIX ATC 14)*, 2014, pp. 305–319.

[10] B. M. Oki and B. H. Liskov, "Viewstamped replication: A new primary copy method to support highly-available distributed systems," in *Proceedings of the Seventh Annual ACM Symposium on Principles of Distributed Computing*, ser. PODC '88. New York, NY, USA: Association for Computing Machinery, 1988, p. 8–17. [Online]. Available: https://doi.org/10.1145/62546.62549

[11] M. Szalay, P. Mátray, and L. Toka, "State management for cloud-native applications," *Electronics*, vol. 10, no. 4, 2021. [Online]. Available: https://www.mdpi.com/20A79-9292/10/4/423

[12] L. Zhang and Y. Dong, "5g cloud ran on intel architecture," in *Proceedings of the 7th International Conference on Computer Science and Application Engineering*, ser. CSAE '23. New York, NY, USA: Association for Computing Machinery, 2023. [Online]. Available: https://doi.org/10.1145/3627915.3628022

[13] M. Poke and T. Hoefler, "Dare: High-performance state machine replication on rdma networks," in *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*, 2015, pp. 107–118.

[14] M. A. Olson, K. Bostic, and M. I. Seltzer, "Berkeley db," in *USENIX Annual Technical Conference, FREENIX Track*, 1999, pp. 183–191.

[15] L. Lamport, "The part-time parliament," *ACM Trans. Comput. Syst.*, vol. 16, no. 2, p. 133–169, May 1998. [Online]. Available: https://doi.org/10.1145/279227.279229

[16] W. Schultz, T. Avitabile, and A. Cabral, "Tunable consistency in mongodb," *Proc. VLDB Endow.*, vol. 12, no. 12, p. 2071–2081, Aug. 2019. [Online]. Available: https://doi.org/10.14778/3352063.3352125

[17] M. Barborak, A. Dahbura, and M. Malek, "The consensus problem in fault-tolerant computing," *ACM Comput. Surv.*, vol. 25, no. 2, p. 171–220, Jun. 1993. [Online]. Available: https://doi.org/10.1145/152610.152612

[18] S. Sanfilippo, "Redis in-memory data structure server (redis)," 2009. [Online]. Available: https://redis.io

[19] B. Liskov and J. Cowling, "Viewstamped replication revisited," MIT, Tech. Rep., 2012. [Online]. Available: https://dspace.mit.edu/handle/1721.1/71763

[20] Y. Liu, R. Laigner, and Y. Zhou, "Rethinking state management in actor systems for cloud-native applications," in *Proceedings of the 2024 ACM Symposium on Cloud Computing*, ser. SoCC '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 898–914. [Online]. Available: https://doi.org/10.1145/3698038.3698540

[21] B. Cully, G. Lefebvre, D. Meyer, M. Feeley, N. Hutchinson, and A. Warfield, "Remus: High availability via asynchronous virtual machine replication," in *Proceedings of the 5th USENIX symposium on networked systems design and implementation*. San Francisco, 2008, pp. 161–174.

[22] Y. Dong, W. Ye, Y. Jiang, I. Pratt, S. Ma, J. Li, and H. Guan, "Colo: Coarse-grained lock-stepping virtual machines for non-stop service," in *Proceedings of the 4th Annual Symposium on Cloud Computing*, ser. SOCC '13. New York, NY, USA: Association for Computing Machinery, 2013. [Online]. Available: https://doi.org/10.1145/2523616.2523630

[23] J. Sherry, P. X. Gao, S. Basu, A. Panda, A. Krishnamurthy, C. Maciocco, M. Manesh, J. a. Martins, S. Ratnasamy, L. Rizzo, and S. Shenker, "Rollback-recovery for middleboxes," in *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, ser. SIGCOMM '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 227–240. [Online]. Available: https://doi.org/10.1145/2785956.2787501

[24] J. Liu, M. Chu, J. Liu, J. Reich, and F. Zhao, "State-centric programming for sensor-actuator network systems," *IEEE Pervasive Computing*, vol. 2, no. 4, pp. 50–62, 2003.

[25] M. Kablan, A. Alsudais, E. Keller, and F. Le, "Stateless network functions: breaking the tight coupling of state and processing," in *Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'17. USA: USENIX Association, 2017, p. 97–112.

[26] L. Abeni, R. Andreoli, H. Gustafsson, R. Mini, and T. Cucinotta, "Fault tolerance in real-time cloud computing," in *2023 IEEE 26th International Symposium on Real-Time Distributed Computing (ISORC)*. IEEE, May 2023.

Deployment of Real-Time Containers in TSN-based Distributed Embedded Systems

Muhammad Waqas*, Per Strandberg†, Johan Furunäs Åkesson†, Saad Mubeen*, Mohammad Ashjaei*
*Mälardalen University, Västerås, Sweden
{muhammad.waqas, saad.mubeen, mohammad.ashjaei}@mdu.se
†Westermo Network Technologies AB, Västerås, Sweden
{per.strandberg, johan.furunasakesson}@westermo.com

***Abstract*—Container-based deployment is increasingly adopted in distributed industrial systems to support modular and flexible software integration; however, ensuring timing correctness in such systems depends on both container execution and predictable network communication. While real-time container platforms provide processor isolation and Time-Sensitive Networking (TSN) enables predictable and deterministic communication, the end-to-end timing behavior of distributed real-time applications remains strongly influenced by deployment decisions such as container placement and inter-container communication. This paper addresses this challenge by proposing a timing-aware deployment approach for distributed real-time containers over TSN-enabled infrastructure. The deployment problem is formulated as a mixed-integer linear program that jointly determines container placement and communication under processor capacity, placement admissibility, and network constraints. The candidate deployments are subsequently analyzed using CONAN-TSN to verify timing feasibility and deadline satisfaction.**

***Index Terms*—Real-time, Containers, Orchestration, TSN.**

I. Introduction

Industrial systems are transitioning to modular and distributed architectures that facilitate dynamic reconfiguration and flexible operation [1]. Consequently, such systems must enable rapid adaptation to changing production requirements and support the integration of components across distributed architectures. To address these requirements, industrial system architectures are shifting from rigid, hardware-centric solutions toward software-centric solutions, where system functionality is realized through software components deployed across distributed nodes [2]. While this transition enables scalability, reusability, and flexibility, it also introduces new challenges related to the structuring, deployment, and management of these systems.

To address these challenges, software containers have emerged as a key enabling technology, providing a lightweight and portable mechanism for deploying and managing software components independently of the underlying hardware [3]. Compared to traditional virtual machines, the reduced overhead of containerization makes this technology particularly suitable for industrial deployments, supporting use cases such as Programmable Logic Controller (PLC) virtualization and using these virtual PLCs for distributed real-time control [4], [5]. As industrial systems continue to evolve toward distributed architectures, the number of containerized applications communicating across networked nodes is expected to grow correspondingly.

In such containerized and distributed settings, satisfying end-to-end timing requirements depends not only on the execution of tasks within individual containers, but also on the communication between distributed nodes that host the containers. In this work, end-to-end refers to the timing behavior of both task execution in containers and message transmission in the network where deadlines must be satisfied at each level. At the same time, industrial networks typically carry mixed traffic, where time-critical communication must coexist with non-critical data flows on a shared network infrastructure [6]. This combination of stringent timing requirements and traffic heterogeneity necessitates communication technologies that provide timing predictable behavior while supporting traffic coexistence. Time-Sensitive Networking (TSN) [7] addresses these requirements by extending Ethernet with mechanisms such as traffic shaping and scheduling, enabling real-time and best-effort traffic to coexist on the same network while providing bounded latency for time-critical traffic.

Despite these advances, the combination of software containers and Time-Sensitive Networking (TSN) is insufficient to ensure the timing correctness of distributed real-time applications. While containerization provides a flexible execution environment and TSN enables predictable and deterministic communication, the end-to-end timing behavior of such applications remains highly dependent on how software components are deployed across computational nodes and how their communication flows are routed through the network. These decisions directly influence processor utilization and communication interference, and ultimately determine whether the specified timing requirements can be satisfied. Consequently, the core challenge in this regard is to determine container deployments that preserve timing requirements while jointly accounting for computational and communication constraints. This challenge is not explicitly addressed by mainstream deployment platforms, such as Kubernetes [8], which are primarily designed for general-purpose and best-effort workloads.

To address this gap, this paper proposes a timing-aware deployment approach for distributed real-time containers over TSN-enabled infrastructure. The paper provides the following main contributions.

- A timing-aware deployment workflow for distributed

real-time containerized applications over TSN-enabled industrial networks.
- An optimization-based formulation of container placement and inter-container communication, where deployments are constrained by processor capacity and network bandwidth, and subsequently analyzed for network schedulability.

II. Background and Related work

This section provides background on TSN networks and related work on real-time container deployment.

A. Background

TSN is a set of standards that provide mechanisms for predictable and deterministic communication over Ethernet [9]. TSN supports different traffic types, namely Scheduled Traffic (ST), Audio Video Bridging (AVB), and Best-effort (BE). ST is scheduled offline and transmitted with Time-aware Shaper (TAS) that controls the opening and closing of queue gates according to a predefined schedule. Thus, ST experiences a limited interference from other types of traffic. AVB traffic is shaped using the Credit-Based Shaper (CBS), which assigns credit to each AVB queue. Without the shaper, lower priority queues may experience long delays because the transmission is mainly decided by priority. CBS addresses this by controlling the bandwidth usage of each AVB queue through a credit mechanism. The credit increases when an AVB message is waiting for transmission or when the credit is negative, and decreases when the messages from that queue are transmitted. The rate at which the credit is replenished is called idleSlope, whereas the rate at which it is consumed during transmission is called sendSlope. An AVB queue is eligible for transmission when its credit is zero or positive [10]. In this work, we focus on AVB traffic and restrict it to the two stream reservation classes, Class A and Class B, because the deployment model targets class-based bandwidth-aware placement and routing. In this setting, each message is assigned to an AVB class, and feasibility depends on whether the selected routes have sufficient class-specific CBS bandwidth reservation.

B. Related work

There are several works that have focused on container-based virtualization for real-time systems. Queiroz et al. [11] provide a systematic review of container-based virtualization for industrial real-time systems covering latency, container platforms, orchestration, and open challenges. Existing work on real-time containers has mainly focused on enabling predictable execution on individual nodes. A common direction is to combine container-based deployment with real-time Linux, such as PREEMPT_RT, to reduce kernel latency for real-time workloads [12], [13]. Hierarchical Constant Bandwidth Server (HCBS) further supports temporal isolation by enforcing processor reservations at the cgroup level, allowing real-time containers a guaranteed execution time [14].

Struhar et al. [15] focus on computational orchestration, using HCBS-managed reservations to support processor allocation and runtime adaptation. Similarly, Fiori et al. [16] and Samimi et al. [17] extend container orchestration support for real-time workloads by enabling reservation-based execution of real-time containers. These works mainly focus on predictable container execution, CPU isolation, and computational resource management on nodes, while network constraints are not treated as part of the container deployment decision.

A complementary line of work by Garbugli et al. [18] addresses deterministic communication for containerized applications by providing a deterministic Kubernetes overlay network based on TSN and kernel bypassing techniques. However, they focus on the communication data path rather than deciding container placement or flow routing. Furthermore, Walker et al. [19] integrate container deployment with TSN configuration and allow custom schedulers to be incorporated. However, their focus is architectural integration, their scheduler is illustrative rather than a general placement and routing method, and their TSN configuration is centered only around time-triggered traffic.

Another body of work has studied the routing and schedule synthesis for TSN traffic, focused on scheduled traffic. Schweissguth et al. [20] and Hellmanns et al. [21] propose Integer Linear Programming-based formulations for routing and scheduling of TSN streams. Berisa et al. [22] study AVB-aware scheduling, where scheduled-traffic schedules are constructed while preserving the schedulability of AVB traffic. These studies mainly configure the network with the assumption of fixed end points. Chahed and Kassler [23] study a closely related problem by jointly optimizing TSN task placement, routing, and TAS-based schedule synthesis in virtualized edge-compute platforms. Their work focuses on scheduled traffic and network schedule generation, with node-level resources handled abstractly through candidate servers rather than explicit processor-capacity constraints.

In contrast, the work presented in this paper focuses on the deployment decision problem for distributed real-time containers. We formulate joint container placement and communication routing as a resource-constrained optimization problem that accounts for node-level processor capacity, communication bandwidth, and various timing requirements. The generated deployments are then evaluated through a separate timing feasibility analysis stage.

III. System Model

The system, denoted by $\mathcal{S}$, consists of two or more computing nodes and a TSN network. The system can be formally represented as follows.

$$\mathcal{S} := \langle \mathcal{N}, \mathcal{L}, \Pi, \mathcal{M} \rangle$$

where $\mathcal{N}$ is the set of computing nodes, and $\mathcal{L}$ is the set of links in the TSN network. The term Π represents the set of software containers hosted on the nodes, and $\mathcal{M}$ is the set of messages exchanged between containers.

A. Computing Node Model

The set of nodes, $\mathcal{N}$, represents a set of uniprocessor computing platforms that can host containers and is defined as follows.

$$\mathcal{N} := \{ \langle n_a, \mathrm{Cap}_a \rangle \mid a = 1, \ldots, |\mathcal{N}| \}$$

where each tuple $\langle n_a, \mathrm{Cap}_a \rangle$ represents a computing node n_a and its associated processing capacity Cap_a. The capacity Cap_a represents the amount of processor resource that can be allocated to container reservations on node n_a. During deployment, the sum of the processor demands of the containers assigned to a node must not exceed this capacity. Nodes may have different processing capacities, which allows the model to represent heterogeneous deployment platforms.

B. Container Model

The set of containers that execute tasks in the system is defined as follows.

$$\Pi := \{ \langle \pi_i, Q_i, P_i, A_i \rangle \mid i = 1, \ldots, |\Pi| \}$$

For a container $\pi_i \in \Pi$, Q_i denotes its reserved execution budget, P_i denotes the budget replenishment period, and A_i denotes the admissible node set of π_i. The admissible node set specifies the nodes on which the container is allowed to execute and is defined as follows.

$$A_i := \{ n_a \mid \langle n_a, \mathrm{Cap}_a \rangle \in \mathcal{N} \wedge \pi_i \text{ can be deployed on } n_a \}$$

The admissible node set captures the deployment restrictions. For example, a container that collects data from a physical sensor may need to be deployed on a node connected to that sensor, and therefore cannot be placed on every node in the system.

Containers may host either real-time or best-effort tasks. For real-time containers, the pair (Q_i, P_i) represents the reservation interface provided by the HCBS scheduling model [14]. In hierarchical scheduling, deriving these parameters is known as server design problem [15] and is outside the scope of this work. We assume that these values are given for every container. The internal workload of each real-time container is assumed to be designed and schedulable under its assigned reservation interface, such as in [15]. Therefore, the task model is not required for this paper.

C. Network Model

The network is modeled through a set of directed links $\mathcal{L}$. Each link $l \in \mathcal{L}$ represents a unidirectional communication link in the network topology. We assume that the network features TSN standards, hence we define the links as TSN links in the model. Following the TSN link model, a physical full-duplex connection is represented by two directed links, one in each direction. Thus, transmission and reception over the same physical connection are treated independently. A link may connect a node to a TSN switch, two TSN switches, or a TSN switch to a node. In this model, two nodes may have multiple paths to connect them in the network. A set of possible paths between nodes n_a and n_b is denoted by the set $\mathcal{R}_{a,b} = \{\rho_k\}$, where ρ_k is a path of links between nodes n_a and n_b.

In addition to the network topology, the network contains messages according to TSN standards. In this model, we assume that the containers transmit and receive only AVB messages, hence ST traffic class is outside the scope of this work. Let $\mathcal{X}$ denote the set of AVB classes considered in the system. For each link $l \in \mathcal{L}$ and AVB class $X \in \mathcal{X}$, the CBS queue is associated with an idleSlope $\alpha^{+}_{X,l}$ and a sendSlope $\alpha^{-}_{X,l}$. The idleSlope $\alpha^{+}_{X,l}$ denotes the credit replenishment rate of class X on link l. It specifies the bandwidth budget assigned to AVB class X on link l. Since $\alpha^{+}_{X,l}$ is defined per link and per AVB class, different AVB classes can be assigned different bandwidth on the same link.

The messages exchanged between containers are modeled by $\mathcal{M}$. Each message represents a communication between source and destination container. The set is defined as:

$$\mathcal{M} := \{ \langle m_j, C_j, \mathrm{Src}_j, \mathrm{Dst}_j, T_j, D_j, X_j \rangle \mid j = 1, \ldots, |\mathcal{M}| \}.$$

For a message m_j, C_j denotes the worst-case transmission time of the message. Moreover, Src_j and Dst_j denote the source and destination containers of message m_j, respectively. The source and destination nodes corresponding to m_j are not fixed in the message model, they are determined by the placement of its source and destination containers. Furthermore, D_j denotes the relative deadline, T_j denotes the message period, and $X_j \in \mathcal{X}$ denotes the AVB class assigned to message m_j. The message parameters are provided as a system input. Note that, when two containers are deployed within the same node, the messages are internal that are not modeled in this paper.

IV. Container deployment method

This section formulates the problem of deploying real-time containers on a TSN-based distributed embedded systems, followed by a design-time method to address it.

A. Problem formulation

The problem addressed in this work is the offline deployment of containerized real-time applications over a TSN network. The formulation considers a static deployment inputs, where the set of containers, admissible nodes, and message properties are known at design time. Dynamic changes to containers and message set changes are outside the scope of this work.

When communicating containers are deployed on different nodes, the deployment must determine both container placement and the routing of inter-container communication over the TSN network. We focus on real-time containers with explicit timing requirements and reserved processing resources. Best-effort (BE) containers are not considered, which can be deployed subsequently using the remaining processor capacity on each node, without any timing guarantees.

The placement part of the problem determines the mapping of containers to computational nodes. This mapping is constrained by the admissible placement set defined for each container, as well as by the available processing capacity of each node. In particular, a container can only be deployed on nodes within its admissible set, and the aggregate processor demand assigned to any node must not exceed its capacity. Moreover the deployment must ensure that the timing requirements of all containers are satisfied.

The routing part of the problem considers messages exchanged between containers that are deployed on different nodes. For a message m_j, let Src_j and Dst_j denote its source and destination containers, respectively. If these containers are assigned to nodes n_a and n_b respectively, the message must be routed along one of the candidate paths $\rho_k \in \mathcal{R}_{a,b}$ connecting these nodes. In contrast, if both containers are deployed on the same node, no network routing is required for the corresponding message.

The deployment must also satisfy the network resource constraints associated with routed messages. For each network link and AVB traffic class, the aggregate bandwidth demand of all messages traversing that link must not exceed the bandwidth allocated to that class. However, satisfying bandwidth constraints alone does not guarantee that messages meet their deadlines, as message latency is also influenced by TSN mechanisms, traffic interference, and the impact of multi-hop transmission. Therefore, bandwidth constraints are used to ensure network resource feasibility, while separate timing analysis is required to verify that the messages meet their corresponding deadlines.

The offline deployment problem is thus formulated as a joint container placement and message routing problem. Its objective is to produce a deployment that assigns all real-time containers to admissible nodes and selects routing paths between communicating containers such that the timing requirements of both containers and messages are satisfied.

B. Deployment method

In order to address this problem, we propose a multi-stage deployment method. The first stage takes the system model as input. Second, the resource-allocation part of the problem is formulated as a mixed-integer linear program (MILP). The MILP finds candidate deployments that satisfy the processor capacity, placement admissibility, routing, and AVB bandwidth constraints. Third, these candidate deployments are subsequently evaluated using the CONAN-TSN toolchain [24] to verify whether message deadlines are met. The toolchain supports several functions, including traffic mapping and network analysis. In this work, however, we focus exclusively on the timing analysis for AVB traffic to assess whether the selected routing paths guarantee that the messages' deadlines are met. Finally, a deployment is accepted only if it satisfies both the MILP constraints and the subsequent timing feasibility check. If no candidate deployment satisfies the timing feasibility check, the deployment is considered infeasible. The overall workflow is shown in Fig. 1.

At the container level, (Q_i, P_i) is assumed to provide a valid reservation for the internal workload of a container. This means that the reservation parameters are assumed to be correctly specified such that all tasks executing within the container meet their timing requirements. The formulation therefore does not check task-level schedulability, but only ensures that the total reserved processor capacity does not exceed the capacity of each node. This capacity check is necessary because the node shall not commit more execution time than it can serve.

The placement and routing problem is combinatorial because each real-time container must be assigned to a node and each inter-node message must be assigned to one candidate path. Even if routing is ignored, the problem reduces to assigning containers with processor demands to nodes with limited capacity, which resembles a bin packing problem and is NP-hard [25]. Therefore, the MILP is used to solve the resource allocation part of the problem, while detailed timing analysis is handled outside the optimization model. The optimization objective is to obtain a feasible candidate solution and avoid concentrating computation or communication load on a small set of nodes and links. For compactness, in the formulation below, n_a, π_i, and m_j refer to the node, container, and message components of their corresponding tuples in $\mathcal{N}$, Π, and $\mathcal{M}$, respectively.

For each real-time container $\pi_i \in \Pi$, let d_i denote its processor demand. This demand is derived from the reservation interface (Q_i, P_i), i.e., $d_i := \frac{Q_i}{P_i}$. The model uses this demand during placement decisions.

For each message m_j, let r_j denote its bandwidth demand. r_j is calculated from the message size divided by its period. It can also be derived from the worst-case transmission time of the message and the link speed.

The MILP ensures that, for each directed link l and AVB class X, the total bandwidth demand of messages of class X routed over link l does not exceed the assigned idleSlope $\alpha^{+}_{X,l}$.

We denote by Ω the set of node pairs for which at least one candidate path exists, defined as follows.

$$\Omega := \{(n_a, n_b) \mid \langle n_a, \mathrm{Cap}_a \rangle \in \mathcal{N},\ \langle n_b, \mathrm{Cap}_b \rangle \in \mathcal{N},\ n_a \neq n_b,\ \mathcal{R}_{a,b} \neq \emptyset\}. \tag{1}$$

Similarly, $\bar{\Omega}$ denotes the set of node pairs for which no candidate path exists.

$$\bar{\Omega} := \{(n_a, n_b) \mid \langle n_a, \mathrm{Cap}_a \rangle \in \mathcal{N},\ \langle n_b, \mathrm{Cap}_b \rangle \in \mathcal{N},\ n_a \neq n_b,\ \mathcal{R}_{a,b} = \emptyset\}. \tag{2}$$

The model uses two binary decision variables. The placement variable $x_{i,a}$ is equal to one if container π_i is assigned to node n_a, and zero otherwise. For nodes $n_a \notin A_i$, the placement variable is fixed to zero. Similarly, the routing variable $z_{j,a,b,k}$ is equal to one if message m_j is routed from node n_a to node n_b over candidate path $\rho_k \in \mathcal{R}_{a,b}$, and zero otherwise. Both variables are binary:

$$x_{i,a},\ z_{j,a,b,k} \in \{0, 1\}. \tag{3}$$

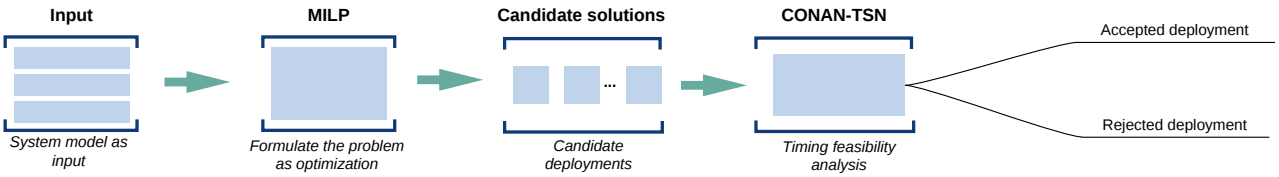


Fig. 1. Offline placement workflow.

Placement feasibility: Each container must be assigned to exactly one admissible node:

$$\sum_{n_a \in A_i} x_{i,a} = 1, \qquad \forall \pi_i \in \Pi. \tag{4}$$

The total processor demand of the containers placed on a node must not exceed its capacity:

$$\sum_{\pi_i \in \Pi} d_i x_{i,a} \leq \mathrm{Cap}_a, \qquad \forall \langle n_a, \mathrm{Cap}_a \rangle \in \mathcal{N}. \tag{5}$$

Placement-routing consistency: Routing is required only for messages whose source and destination containers are placed on different nodes. For every node pair $(n_a, n_b) \in \Omega$, the following constraints ensure that a candidate path is selected only when the source container of message m_j is placed on node n_a and its destination container is placed on node n_b:

$$\sum_{\rho_k \in \mathcal{R}_{a,b}} z_{j,a,b,k} \leq x_{\mathrm{Src}_j,a}, \qquad \forall m_j \in \mathcal{M},\ \forall (n_a, n_b) \in \Omega. \tag{6}$$

$$\sum_{\rho_k \in \mathcal{R}_{a,b}} z_{j,a,b,k} \leq x_{\mathrm{Dst}_j,b}, \qquad \forall m_j \in \mathcal{M},\ \forall (n_a, n_b) \in \Omega. \tag{7}$$

$$\sum_{\rho_k \in \mathcal{R}_{a,b}} z_{j,a,b,k} \geq x_{\mathrm{Src}_j,a} + x_{\mathrm{Dst}_j,b} - 1, \qquad \forall m_j \in \mathcal{M},\ \forall (n_a, n_b) \in \Omega. \tag{8}$$

Together, Constraints (6)-(8) force the candidate path selection sum to be one if the source and destination containers of m_j are placed on n_a and n_b, and zero otherwise. If no candidate path exists between two nodes, the model forbids placing the source and destination containers of the message on that node pair:

$$x_{\mathrm{Src}_j,a} + x_{\mathrm{Dst}_j,b} \leq 1, \qquad \forall m_j \in \mathcal{M},\ \forall (n_a, n_b) \in \bar{\Omega}. \tag{9}$$

Communication capacity feasibility: For each link and AVB class, the aggregate bandwidth of all messages routed over that link must not exceed the available bandwidth of that class:

$$\sum_{\substack{m_j \in \mathcal{M} \\ X_j = X}} \sum_{(n_a,n_b) \in \Omega} \sum_{\substack{\rho_k \in \mathcal{R}_{a,b} \\ l \in \rho_k}} r_j z_{j,a,b,k} \leq \alpha^{+}_{X,l}, \quad \forall l \in \mathcal{L},\ \forall X \in \mathcal{X}. \tag{10}$$

The offline optimization problem is therefore formulated as:

$$\min\ \lambda_u \delta_u + \lambda_b \delta_b + \lambda_h H, \tag{11}$$

subject to the placement, placement-routing consistency, and communication constraints above.

In Eq. (11), δ_u denotes the normalized processor-load imbalance across nodes. It is computed from the absolute deviation between the processor utilization of each node and the average processor utilization of the system. Similarly, δ_b denotes the normalized bandwidth-load imbalance across network links. It is computed from the absolute deviation between the bandwidth utilization of each link and the average bandwidth utilization of the system. These terms help balance the processor load across nodes and the bandwidth load across network links. For each candidate path $\rho_k \in \mathcal{R}_{a,b}$, h_{ρ_k} denotes its hop count, derived from the number of links in that path. It is computed by summing the hop count h_{ρ_k} of each selected candidate path, followed by normalization. Limiting the hop count is important because placing communicating containers closer together can reduce communication delay. The weights λ_u, λ_b, and λ_h control the relative importance of processor-load balancing, bandwidth-load balancing, and short-path routing, respectively.

The absolute-deviation terms used in δ_u and δ_b are implemented with auxiliary variables, while H is linear by construction. Therefore, the optimization model remains linear.

V. Implementation

The implementation of the offline placement method is currently in progress. The MILP model is implemented using Gurobi [26] with API `gurobipy` version 13.0.1 in Python. The prototype takes the system model as input, including the nodes, messages between containers, admissible node sets, and candidate paths, and constructs the placement and routing variables together with the constraints described in Section IV-B.

After solving the model, the prototype extracts the selected container placements and routes from the decision variables. These candidate deployments are then intended to be passed to CONAN-TSN for network-timing feasibility analysis. However, this integration is still under development. At this stage, the implementation is used to validate that the proposed formulation can be instantiated and solved for concrete system descriptions.

VI. Conclusion and Future Work

This paper presented an offline placement and routing selection workflow for distributed real-time containers deployed over TSN. The proposed approach formulates the deployment

problem as an MILP with accounting for processor availability, network timing requirements, and placement admissibility. The generated candidate deployments are then intended to be evaluated using CONAN-TSN to ensure schedulability.

The offline placement of containers is part of a broader timing-aware container orchestration framework. We plan to complete the implementation of the proposed workflow. As future work, we aim to extend the proposed workflow with an online deployment manager that can reorganize existing deployments when new containers are introduced to the system, ensuring that container deadlines and communication between containers are respected.

REFERENCES

[1] J. Morgan, M. Halton, Y. Qiao, and J. G. Breslin, "Industry 4.0 smart reconfigurable manufacturing machines," *Journal of Manufacturing Systems*, vol. 59, pp. 481–506, 2021.

[2] R. Neumann, M. Walker, M. Neubauer, and A. Verl, "Towards uniform and consistent data modelling of resources in distributed industrial control systems," *Procedia CIRP*, vol. 138, pp. 304–309, 2026.

[3] V. Struhár, M. Behnam, M. Ashjaei, and A. Papadopoulos, "Real-time containers: A survey," in *Workshop on Fog Computing and the Internet of Things*, 2020.

[4] T. Goldschmidt, S. Hauck-Stattelmann, S. Malakuti, and S. Grüner, "Container-based architecture for flexible industrial control applications," *Journal of Systems Architecture*, vol. 84, pp. 28–36, 2018.

[5] M. Sollfrank, F. Loch, S. Denteneer, and B. Vogel-Heuser, "Evaluating docker for lightweight virtualization of distributed and time-sensitive applications in industrial automation," *IEEE Transactions on Industrial Informatics*, vol. 17, no. 5, pp. 3566–3576, 2021.

[6] T. Zhang, G. Wang, C. Xue, J. Wang, M. Nixon, and S. Han, "Time-sensitive networking (tsn) for industrial automation: Current advances and future directions," *ACM Comput. Surv.*, vol. 57, no. 2, 2024.

[7] "IEEE TSN task group." http://www.ieee802.org/1/pages/tsn.html.

[8] Kubernetes, "Kubernetes," https://kubernetes.io/, accessed: 2026-04-20.

[9] M. Ashjaei, L. L. Bello, M. Daneshtalab, G. Patti, S. Saponara, and S. Mubeen, "Time-sensitive networking in automotive embedded systems: State of the art and research opportunities," *Journal of Systems Architecture, 2021*, vol. 121, pp. 1–47, 2021.

[10] D. Bujosa, J. Proenza, A. V. Papadopoulos, T. Nolte, and M. Ashjaei, "An improved worst-case response time analysis for avb traffic in time-sensitive networks," in *IEEE Real-Time Systems Symposium (RTSS)*, 2024, pp. 135–147.

[11] R. Queiroz, T. Cruz, J. Mendes, P. Sousa, and P. Simões, "Container-based virtualization for real-time industrial systems—a systematic review," *ACM Comput. Surv.*, vol. 56, no. 3, 2023.

[12] N. Ben Kebaier, B. Geib, E. Lyczkowski, R. Venanzi, and M. Barth, "Real-time performance evaluation of containerized virtual plcs: A comparative study of docker and podman for industrial automation," in *IEEE CAMAD*, 2025.

[13] Linux Foundation, "PREEMPT_RT Versions," https://wiki.linuxfoundation.org/realtime/preempt_rt_versions, accessed: 2026-02-18.

[14] L. Abeni, A. Balsini, and T. Cucinotta, "Container-based real-time scheduling in the linux kernel," *SIGBED Rev.*, vol. 16, no. 3, p. 33–38, 2019.

[15] V. Struhár, S. Craciunas, M. Ashjaei, M. Behnam, and A. Papadopoulos, "Hierarchical resource orchestration framework for real-time containers," *ACM Transactions on Embedded Computing Systems*, vol. 23, no. 1, pp. 1–24, January 2024.

[16] S. Fiori, L. Abeni, and T. Cucinotta, "Rt-kubernetes: containerized real-time cloud computing," in *ACM/SIGAPP Symposium on Applied Computing*, ser. SAC '22. Association for Computing Machinery, 2022, p. 36–39.

[17] N. Samimi, L. Abeni, D. Casini, M. Marinoni, T. Basten, M. Nasri, M. Geilen, and A. Biondi, "Enabling Containerisation of Distributed Applications with Real-Time Constraints," in *37th Euromicro Conference on Real-Time Systems (ECRTS)*, 2025.

[18] A. Garbugli, L. Rosa, A. Bujari, and L. Foschini, "Kubernetsn: a deterministic overlay network for time-sensitive containerized environments," in *IEEE International Conference on Communications (ICC)*, 2023.

[19] M. Walker, N. Imhoff, R. Neumann, M. Neubauer, A. Lechler, and A. Verl, "Methodology for the combined orchestration of real-time containers and time-sensitive network," *Procedia CIRP*, vol. 138, pp. 292–297, 2026.

[20] E. Schweissguth, P. Danielis, D. Timmermann, H. Parzyjegla, and G. Mühl, "ILP-based joint routing and scheduling for time-triggered networks," in *Proceedings of the 25th International Conference on Real-Time Networks and Systems*. New York, NY, USA: Association for Computing Machinery, 2017.

[21] D. Hellmanns, L. Haug, M. Hildebrand, F. Dürr, S. Kehrer, and R. Hummen, "How to Optimize Joint Routing and Scheduling Models for TSN Using Integer Linear Programming," in *Proceedings of the 29th International Conference on Real-Time Networks and Systems*, 2021.

[22] A. Berisa, L. Zhao, S. S. Craciunas, M. Ashjaei, S. Mubeen, M. Daneshtalab, and M. Sjödin, "AVB-aware Routing and Scheduling for Critical Traffic in Time-sensitive Networks with Preemption," in *Proceedings of the 30th International Conference on Real-Time Networks and Systems*. Association for Computing Machinery, 2022.

[23] H. Chahed and A. Kassler, "Optimizing tsn routing, scheduling, and task placement in virtualized edge-compute platforms," *2024 27th Conference on Innovation in Clouds, Internet and Networks (ICIN)*, 2024.

[24] D. B. Mateu, M. Ashjaei, and S. Mubeen, "CONAN-TSN: An Integrated Toolchain for CONfiguration and ANalysis of TSN Networks," in *30th IEEE International Conference on Emerging Technologies and Factory Automation*, 2025.

[25] M. Delorme, M. Iori, and S. Martello, "Bin packing and cutting stock problems: Mathematical models and exact algorithms," *European Journal of Operational Research*, vol. 255, no. 1, pp. 1–20, 2016.

[26] Gurobi Optimization, LLC, "Gurobi Optimizer Reference Manual," 2026. [Online]. Available: https://www.gurobi.com

Criticality-Aware Offloading: Robust Control with Priority-Based Provisioning

Isser Kadusale[1], Emil Sundström[2], Johan Eker[2], and Gerhard Fohler[1]

[1]*Chair of Real-time Systems*, *RPTU University Kaiserslautern-Landau*, Kaiserslautern, Germany
[2]*Department of Automatic Control*, *Lund University*, Lund, Sweden

***Abstract*—Offloading control computations to the edge/cloud continuum enables the use of computation-intensive controllers even for resource constrained devices. In safety-critical control systems, it can also provide robustness guarantees. Then, the controlled processes are subject to bounded disturbances and model uncertainty, while the remote control computations are executed on a multiprocessor server with uncertain execution times. The goal is to reduce overprovisioning of compute resources while still providing robustness guarantees, without requiring the devices or scheduler to explicitly know about the safety requirements. We propose a unified framework that combines robust control with a modified mixed-criticality scheduling model. The framework defines a minimal interface between the devices, scheduler, and remote processors, and derives schedulability guarantees that enable reduced resource usage. We implement and evaluate the approach in a simulated scenario with multiple PID-controlled cars offloading control requests to a remote robust Tube MPC controller, deployed on Kubernetes with a custom auto-scaling mechanism.**

***Index Terms*—Offloading, Mixed-Criticality, Real-Time Systems, Robust Control**

I. Introduction

Hard real-time scheduling is often conservative for control applications. However, reducing this conservatism is difficult in practice due to that timing uncertainty interacts with process uncertainty. That is, jobs may take longer than they are scheduled for at the same time as the physical process is affected by disturbances and modeling errors. Therefore, in this work, we seek to investigate how a combination of robust control (RC) and a modified mixed-criticality (MC) framework can avoid unnecessary overprovisioning of computational resources, while still providing safety guarantees of the process under both timing and process uncertainty.

MC scheduling was introduced by Vestal [1] to reduce time allocation conservatism while still protecting critical tasks. In the standard MC model, each task has a criticality level and several worst-case execution time (WCET) estimates, typically one for low-criticality mode and one for high-criticality mode. If a task exceeds its low-criticality budget, the system switches to a high-criticality mode. Low-criticality tasks are then dropped so that high-criticality tasks can still meet their deadlines. An extensive survey of this framework is given in [2].

However, this model is not a natural fit for control applications. As pointed out in [3], [4], the importance of a control task depends on the state of the controlled process, and is therefore not fixed. Moreover, dropping low-criticality control tasks is often unnecessarily conservative and may reduce closed-loop performance. For this work, we modify the MC framework so that the task criticality is state-dependent. Furthermore, rather than dropping low-criticality control tasks, the scheduler prioritizes high-criticality tasks while continuing to serve all tasks whenever possible. To avoid confusion in relation to classical MC systems, we use the terms best-effort and guaranteed instead of low-critical and high-critical in the remainder of the paper.

We focus on resource-constrained similar devices with simple local control structures that can obtain performance and robustness guarantees by offloading control requests to a remote multiprocessor server. A central design goal is to avoid adding safety-requirement-specific logic to the devices themselves. Instead, the devices and scheduler should remain agnostic to the particular safety constraints. This separation improves scalability. If the safety requirements are tightened, only the remote controller needs to be updated.

This objective is naturally connected to RC. RC addresses controller development and formal guarantees under bounded disturbances and model uncertainties. In this work, we use RC to encode the relevant performance and robustness requirements in the remote controller, and how critical scedulability of it is. For a broader introduction to RC, see e.g. [5].

The need to align real-time scheduling with the state-dependent timing requirements of control has recently been posed as an open challenge for the community by the Physics-Driven Real-Time CPS Challenge [6], in a cloud-offloaded control setting closely related to ours. To the best of our knowledge, however, the intersection of RC and MC scheduling in an offloading setting has not been thoroughly explored in the literature. The works [3], [7] consider MC scheduling from a control-performance perspective, but they do not study switching between local and offloaded controller, nor do they formulate the problem within a RC framework. Conversely, local-remote offloading architectures have been studied from a control perspective [8], [9], but not through a MC framework. This motivates the approach taken in this paper. Our main

This work was partially supported by the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation. The authors Sundström and Eker are members of the ELLIIT Strategic Research Area at Lund University.

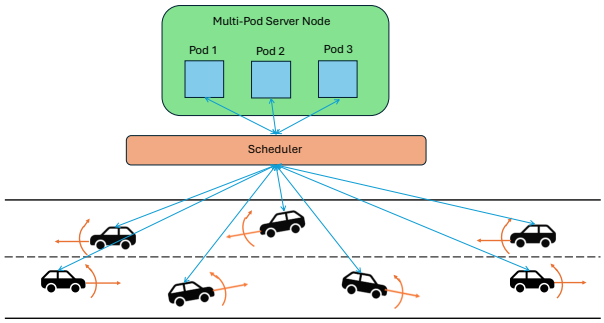


Fig. 1. System setup sketch. Many similar devices try to offload their control computations to the same multi-pod server.

contributions with this work-in-progress paper are summarized as follows:

- We introduce a framework that combines RC with a modified MC model for safety-critical control offloading applications.
- We define a minimal interface between device, scheduler, and server that enables robust control guarantees without requiring the devices or scheduler to explicitly know about these guarantees.
- We present numerical results on how control performance, resource usage, and execution times relate in our setup. Based on this, we propose new research directions regarding high control performance with minimal resource usage for our RC-MC framework.

II. Problem Setup

The setup consists of many replicas of similar devices (imagine a drone swarm or a fleet of cars of the same model) and a remote multi-processor server. The overall goal is to keep all agents within a safe region without over-provisioning of resources. This section describes the system, assumptions and controllers in more detail. Figure 1 illustrates the problem setup.

A. Agent

Consider an agent, with dynamics modeled with the discrete-time linear model

$$x^P_{t+1} = A^P x^P_t + B^P u_t + w^P_t, \quad w^P_t \in \mathbb{W}^P \tag{1}$$

where $x^P \in \mathbb{R}^n$ is the state of the agent, $u \in \mathcal{U} \subseteq \mathbb{R}^m$ is the control signal, $w^P \in \mathbb{W}^P \subset \mathbb{R}^n$ is an external bounded disturbance, and the matrices $A^P \in \mathbb{R}^{n\times n}$ and $B^P \in \mathbb{R}^{n\times m}$ are referred to as the *system matrices*. For now, we assume that the state can be measured directly. Safety requirements is expressed through that the state must be within a safe set, $x^P_t \in \chi^P_{\text{safe}} \subset \mathbb{R}^n$. The disturbance w accounts for physical disturbances and model errors, including nonlinearities. In a RC fashion, we do not assume any particular distribution of the disturbance, more than that it is bounded and that we know upper and lower bounds of each component.

The agent tries to follow a reference trajectory x^r_t, and we denote the tracking error as $x^e_t := x^r_t - x^P_t$. The agent tracks the reference with a local simple feedback controller, $\mathcal{K}_{\text{loc}} : \mathbb{R}^n \to \mathcal{U}$, taking x^e_t as argument. For example PID control can be written on this form by minor adjustments. The control procedure of the agent is the following:

1) Read sensor values (state) and send to a remote server.
2) If no control signal arrives from the remote controller within the sampling period of the agent, actuate its local controller. Else, actuate the remote control signal.

This naturally imposes a one-step delay of the dynamics. To account for that, we re-formulate (1) and augment the state vector to apply the delayed control signal through

$$\underbrace{\begin{bmatrix} x^P_{t+1} \\ \hat{u}_{t+1} \end{bmatrix}}_{x_{t+1}} = \underbrace{\begin{bmatrix} A^P & B^P \\ 0 & 0 \end{bmatrix}}_{A} \underbrace{\begin{bmatrix} x^P_t \\ \hat{u}_t \end{bmatrix}}_{x_t} + \underbrace{\begin{bmatrix} 0 \\ I \end{bmatrix}}_{B} u_t + \underbrace{\begin{bmatrix} w^P_t \\ 0 \end{bmatrix}}_{w_t} \tag{2}$$

such that the agent dynamics can be written as $x_{t+1} = Ax_t + Bu_t + w_t$ where the disturbance is now an element of the set $w_t \in \mathbb{W} \subset \mathbb{R}^{n+m}$. The safe set is also updated to the augmented setting according to

$$\chi_{\text{safe}} := \left\{ x_t \mid x^P_t \in \chi^P_{\text{safe}},\ \hat{u}_t \in \mathcal{U} \right\}. \tag{3}$$

We assume the following:

Assumption 1: There exists a bounded set χ_{loc} such that χ_{loc} is robustly invariant under the local controller, i.e.,

$$Ax_t + B\mathcal{K}_{\text{loc}}(x_t) + w_t \in \chi_{\text{loc}}, \quad \forall x_t \in \chi_{\text{loc}},\ \forall w_t \in \mathbb{W}. \tag{4}$$

In the nominal case $w_t = 0$, the local closed-loop is asymptotically stabilizing on χ_{loc}.

1) Remote Robust Controller: The remote controller $\mathcal{K}_{\text{off}} : \mathbb{R}^n \to \mathcal{U}$ is due to its access to extra compute resources assumed to robustly keep the agent state within a narrower set $\chi_{\text{off}} \subseteq \chi_{\text{safe}} \subseteq \chi_{\text{loc}}$. This is formulated as follows:

Assumption 2: There exists a set $\chi_{\text{off}} \subseteq \chi_{\text{safe}}$ and a remote robust control law $\mathcal{K}_{\text{off}}$ such that χ_{off} is robustly positively invariant for the delayed augmented system (2) under continued remote control, i.e.,

$$Ax_t + B\mathcal{K}_{\text{off}}(x_t) + w_t \in \chi_{\text{off}}, \quad \forall x_t \in \chi_{\text{off}},\ \forall w_t \in \mathbb{W}. \tag{5}$$

The set χ_{off} contains the augmented states for which the stored input $\hat{u}$ was generated by the remote controller.

Because of the one-step input delay, the first transition after assigning the remote controller may still depend on a control signal not generated by $\mathcal{K}_{\text{off}}$. This makes the robust switching guarantees non-trivial, and is handled below.

2) Reachable Sets and Safe Switching: To be able to guarantee safe switching conditions between offloading and local control, it is necessary to online calculate reachable sets under respective controller. If we know that the state of an agent is $x \in X$, we know that under the control law $\mathcal{K}$, i.e. $\mathcal{K}_{\text{loc}}$ or $\mathcal{K}_{\text{off}}$, the state at the next sampling time is in

$$\mathcal{F}_{\mathcal{K}}(X) := \{ Ax + B\mathcal{K}(x) + w \mid x \in X,\ w \in W \}. \tag{6}$$

Through (6) and from a current state x_t, we calculate the *disturbance reachable set* in k steps through

$$\begin{aligned} X_0(\mathcal{K}, x_t) &:= \{x_t\}, \\ X_{k+1}(\mathcal{K}, x_t) &:= \mathcal{F}_{\mathcal{K}}\big(X_k(\mathcal{K}, x_t)\big). \end{aligned} \tag{7}$$

Under Assumption 1 and bounded disturbances, the local closed-loop system has a bounded disturbance reachable set. However, this set need not be contained in χ_{safe}, which motivates offloading to the robust controller. Given a state x_t, we can define the *safe-time*, for which the local controller is guaranteed to keep the agent within χ_{safe}:

Definition 1: Given a state $x_t \in \chi_{\text{safe}}$ and a control law $\mathcal{K}$, define the safe horizon

$$\begin{aligned} &N_{\text{safe}}(\mathcal{K}, x_t) := \\ &\sup\left\{N \in \mathbb{Z}_{\geq 0} \,|\, X_k(\mathcal{K}, x_t) \subseteq \chi_{\text{safe}},\ \forall\, k = 0, \ldots, N\right\}. \end{aligned} \tag{8}$$

The corresponding safe-time is

$$\tau_{\text{safe}}(\mathcal{K}, x_t) := T_s N_{\text{safe}}(\mathcal{K}, x_t), \tag{9}$$

where T_s is the sampling time of the agent.

We can use the concept of safe-time to define the time until an agent MUST be provided with remote compute resources, to avoid the risk of leaving χ_{safe}:

Definition 2: Suppose the most recent safe-time calculation was performed at server time t_s, using measured state x_s. Define the latest time for which the remote controller must be applied as

$$t_{\text{latest}} := t_s + \tau_{\text{safe}}(\mathcal{K}_{\text{loc}}, x_s) - T_s, \tag{10}$$

where T_s accounts for the one-sample delay in (2). The offloading request from an agent is considered guaranteed at server time t if

$$t \geq t_{\text{latest}} - T_s \tag{11}$$

Note that T_s in (11) is needed due to that t_{latest} defines the time for which the remote controller must act before. So (11) defines the last sample before the remote controller cannot guarantee safety. We can also define the reverse direction, i.e. when an offloading request of an agent is assumed best-effort:

Definition 3: Let x_t be the latest state for which the remote controller is applied. Define the worst-case local safe-time after one remote-controlled step as

$$\tau_{\min}(x_t) := \min_{x^+ \in \mathcal{F}_{\mathcal{K}_{\text{off}}}(\{x_t\})} \tau_{\text{safe}}(\mathcal{K}_{\text{loc}}, x^+). \tag{12}$$

A guaranteed request with state x_t is declared best-effort if

$$\tau_{\min}(x_t) > 2T_s, \tag{13}$$

Similar to Definition 2, one T_s is for the one-step delay in (2) and the other T_s is for the delayed action time.

We can now state the following result.

Theorem 1: Consider an agent with initial state $x_0 \in \chi_{\text{off}}$ that initially uses the remote controller. Suppose that Assumption 2 holds. Assume that the switching policy satisfies the following two conditions.

1) While the local controller is used, the switch to the remote controller is made no later than t_{latest} in Definition 2.
2) Once the remote controller is used, the switch back to the local controller is made only after the request has become non-critical according to Definition 3.

Then, for every admissible disturbance sequence $w_t \in \mathbb{W}$, the closed-loop state satisfies

$$x_t \in \chi_{\text{safe}}, \qquad \forall t \geq 0. \tag{14}$$

Proof: We prove the claim by induction over sampling instants. By assumption, $x_0 \in \chi_{\text{off}} \subseteq \chi_{\text{safe}}$. Suppose now that $x_t \in \chi_{\text{safe}}$ at some sampling instant t. We show that $x_{t+1} \in \chi_{\text{safe}}$. There are two cases for this.

First, suppose the remote controller is applied. If $x_t \in \chi_{\text{off}}$, then Assumption 2 gives

$$Ax_t + B\mathcal{K}_{\text{off}}(x_t) + w_t \in \chi_{\text{off}} \subseteq \chi_{\text{safe}}, \quad \forall x_t \in \chi_{\text{off}},\ \forall w_t \in \mathbb{W}. \tag{15}$$

If this is the first step after switching from local to remote control, the previously stored input may have been generated by the local controller. By Definition 2, the switch is made before t_{latest}, accounting for the one-step delay. Hence, since $\hat{u}_{t+1}$ is then from $\mathcal{K}_{\text{off}}$ and $x^P_{t+1} \in \chi^P_{\text{safe}}$, the delayed transition x_{t+1} is in $\chi_{\text{off}} \subseteq \chi_{\text{safe}}$.

It remains to consider the case when the local controller is applied. If the controller has just switched from remote to local, then Definition 3 implies that every possible successor after the last remote-controlled step has local safe-time at least $2T_s$. From Definition 2, this corresponds to that the time when the agent becomes critical is strictly bounded by the first sampling time after the switch,

$$t \geq t_{\text{latest}} - T_s > t_s, \tag{16}$$

meaning that the first request after a switch from remote to local is non-critical, and $x_{t+1} \in \chi_{\text{safe}}$. If local control was also used in the previous step, then by Definition 2, the request has not reached the critical deadline. Since t_{latest} incorporates the one-sample delay, $x_{t+1} \in \chi_{\text{safe}}$.

Hence by induction, $x_{t+1} \in \chi_{\text{safe}}$ for all $t \geq 0$. ■

This means that as long as the switching between local and offloaded controllers satisfies the above switching conditions, the device is guaranteed to stay in the safe set.

III. Device-Scheduler-Server Interface

Our approach to the problem is to define a minimal interface between devices, scheduler and remote control workers. The purpose of the interface is to keep the robustness requirement inside the remote controller. That is, devices only send states and apply inputs, while the scheduler only schedules requests based on labels and timing bounds.

At the beginning of each control period, device i sends its current state x_t to the scheduler. The scheduler labels the

request as either guaranteed ($\mathcal{G}$) or best-effort ($\mathcal{B}$) and forwards the pair (x_t, p_t) to a control worker, where

$$p_t \in \{\mathcal{G}, \mathcal{B}\}. \tag{17}$$

The worker returns a control input u_t and timing information used by the scheduler for future scheduling decisions. The scheduler forwards u_t to the device. If the input does not arrive before the end of the control period, the device applies its local controller $\mathcal{K}_{\text{loc}}$.

A. Device

The device is responsible only for sending its state, x_t, and actuating either the received remote control signal or, if no response arrives in time, its local controller. It does not know the request label, the robustness requirements, or the performance bounds.

B. Control Worker

A control worker receives (x_t, p_t) from the scheduler and computes the remote robust control input together with scheduling-relevant timing bounds. If $p_t = \mathcal{B}$, the worker computes when future requests from the same device may need guaranteed service according to Definition 2. If instead $p_t = \mathcal{G}$, the worker computes a conservative time until requests from that device are not guaranteed anymore. For a state x_t, that is

$$T_s \ \min\{N \in \mathbb{Z}_{\geq 0} \mid x \text{ is best-effort } \forall\, x \in X_N(\mathcal{K}_{\text{off}}, x_t)\},$$

where the criteria for best-effort is from Definition 3. If the very next request is best-effort, the worker also returns $\tau_{\min}(x_t)$, which bounds how long the requests can remain best-effort.

C. Scheduler

The scheduler receives states from the devices, assigns request labels, schedules the requests, and determines the number of active workers. Its hard requirement is that all guaranteed requests are scheduled according to their WCET. Apart from that, the scheduler can freely choose the number of workers and how to schedule requests. The scheduler stores the timing bounds returned by the workers and uses them for future labeling, scheduling, and auto-scaling decisions. It does not need access to the plant model, disturbance set, or robust-control performance requirements.

IV. Schedulability Analysis

We model the offloading of the controller as a set of n similar independent periodic tasks $\Gamma = \{\tau_1, \tau_2, \ldots, \tau_n\}$, where each agent maps to a task. Each task is characterized by a tuple $\tau_i = (\varphi_i, \delta, T, D, C_B, C_G)$, where φ_i is the phase, δ is the maximum arrival jitter, T is the period, D is the deadline ($D \leq T$), C_G is the worst-case execution time (WCET), and C_B is the execution time budget for best-effort jobs, with $C_B < C_G$. A natural choice for C_B is the average execution time, but any value less than C_G is permitted. We assume that all tasks have the same period, deadline, execution time budget, and WCET. This stems from our assumption that our system consists of replicas of similar devices. For simplicity, we also assume that all tasks have the same maximum arrival jitter. We assume that execution times include all overheads associated with dispatching to a pod and scheduling within the pod's node. Each task gives rise to a sequence of jobs, where the k-th job of τ_i is characterized by a tuple $j_{i,k} = (A_{i,k}, D_{i,k}, s_{i,k}, C_B, C_G)$, where $A_{i,k}$ is the arrival time, $D_{i,k}$ is the absolute deadline, and $s_{i,k} \in \{B, G\}$ is the state of the job. The arrival time is given by $A_{i,k} = \varphi_i + kT + \eta_{i,k}$, where $\eta_{i,k} \in [0, \delta]$. The absolute deadline is $D_{i,k} = \varphi_i + kT + D$. A job in state G (guaranteed) must meet its deadline, while a job in state B (best-effort) may be dropped or allowed to miss its deadline. A job's state does not change and is known ultimately upon completion of the previous job. The state transitions between jobs of a task are governed by the safe-time framework in Section II. We denote the number of tasks whose jobs are currently best-effort as n_B, and the number of tasks whose jobs are currently guaranteed as n_G.

We provision computation resources for the task set based on the state of the tasks' jobs: guaranteed jobs are provisioned for C_G, while best-effort jobs are provisioned for C_B. Note that best-effort jobs can still execute for up to C_G. Our system must ensure that all guaranteed jobs meet their deadlines, while best-effort jobs are only provisioned for C_B and may be dropped.

We model pods as a set of identical processors $P = \{p_1, p_2, \ldots, p_m\}$, where the number of pods m can be dynamically changed at run time with a bounded delay Δ. This delay is the worst-case duration from the time our system decides to change the number of pods until the time the new pods are able to service jobs or the time pods are terminated. Pods execute jobs non-preemptively. This removes the need for adding logic to handle interrupting an ongoing computation, greatly simplifying the pod implementation.

As jobs arrive, they are placed into one of two queues depending on their state: the guaranteed queue, or the best-effort queue. Whenever a pod is available (i.e., not executing a job), the scheduler dispatches a job to it from the guaranteed queue if it is not empty or from the best-effort queue otherwise. Jobs within each queue are dispatched in FIFO order. Any job that has not been dispatched to a pod by time $D_{i,k} - C_B$ is removed from the queue and dropped.

A. Worst-Case Response Time for a Guaranteed Job

We determine the schedulability of guaranteed jobs by first deriving a bound on their response time, which we define as the elapsed time from its arrival until it finishes executing in a pod. When $m \geq n$, every job can immediately start executing in a pod upon arrival and the worst-case response time is simply C_G. We, therefore, consider the case where $m < n$.

Let $j_{i,k}$ denote an arbitrary guaranteed job, which arrives at time t, and let R denote the worst-case response time over all arrival patterns. Since the scheduler dispatches guaranteed jobs before any best-effort jobs, $j_{i,k}$ only waits when there aren't enough available pods to accommodate $j_{i,k}$ and any

guaranteed jobs ahead of $j_{i,k}$ in the guaranteed queue. The waiting time of $j_{i,k}$ is therefore determined by: the number of guaranteed jobs ahead of $j_{i,k}$ in the guaranteed queue, which we denote as n_{ahead}, and the number of jobs already executing in pods when $j_{i,k}$ arrives, which we denote as n_{exec}.

The bound for n_{ahead} is $n_{\text{ahead}} \leq n_G - 1$. Within a period, there is at most n_G jobs queued, including j, because $D \leq T$ and any job $j_{i,k}$ is dropped by time $D_{i,k} - C_B$ if it is not yet executing. The maximum wait is induced when all n_G jobs arrive simultaneously at t and $j_{i,k}$ is placed last in the queue due to a tie-break. If $j_{i,k}$ arrives later, then the wait is less, and earlier puts it ahead of the queue.

The bound for n_{exec} is $n_{\text{exec}} \leq \min(m, n_B)$. At most m pods can be occupied at any time, and at most n_B best-effort jobs exist in a period. The maximum wait is induced when $\min(m, n_B)$ best-effort jobs are dispatched immediately before t, so that each best-effort job can execute up to C_G. A job dispatched earlier than this contributes less to the wait, since it has consumed part of its execution time by t. This includes any job from an earlier period (i.e., jobs dispatched right before $D - C_B$). The jobs must also be best-effort jobs so that n_{ahead} can have the maximum value.

The response time of $j_{i,k}$ is the waiting time induced by these two quantities plus $j_{i,k}$'s own execution of C_G.

The worst-case response time R occurs for $j_{i,k}$ when $\min(m, n_B)$ best-effort jobs arrive immediately before t, and all n_G jobs including $j_{i,k}$ arrive simultaneously at t, where $j_{i,k}$ is placed last in the queue by a tie-break. R is the time required for m pods, to drain the $\min(m, n_B)$ best-effort jobs already executing and the n_G guaranteed jobs (including $j_{i,k}$), each executing for C_G. With m pods running in parallel and FIFO placing $j_{i,k}$ last in the guaranteed queue, $j_{i,k}$ completes at the end of the last batch, and its response time is given in Equation 18.

$$R = \left\lceil \frac{n_G + \min(m, n_B)}{m} \right\rceil \cdot C_G \tag{18}$$

The system is schedulable, in the sense that no guaranteed job is ever dropped and every guaranteed job meets its deadline, when $R \leq D - \delta$. When this holds, $j_{i,k}$ is dispatched at the latest at time $t+R-C_G$, which is at most $t+(D-\delta)-C_G$. Since $C_G > C_B$ this is no later than $t + (D_{i,k} - \delta) - C_B$, which is $j_{i,k}$'s drop time. Therefore, $j_{i,k}$ is not dropped.

B. Pod Resource Provisioning

We provision pods for the task set based on: the minimum pod count m such that the inequality from the previous subsection holds, and the pod count with enough aggregate capacity to serve best-effort jobs at C_B execution time and guaranteed jobs at C_G execution time. The former is a hard guarantee for guaranteed jobs, and the latter is a combined provisioning measure.

Substituting the worst-case response time from the previous subsection into $R \leq D - \delta$, dividing by C_G, and since the left side is an integer value, we get the schedulability inequality in Equation 19, where $Q = \left\lfloor \frac{D-\delta}{C_G} \right\rfloor$.

$$\left\lceil \frac{n_G + \min(m, n_B)}{m} \right\rceil \leq Q, \tag{19}$$

The inequality splits into two cases depending on whether the pod count exceeds the number of best-effort jobs. When $m \leq n_B$, we have $\min(m, n_B) = m$ and the ceiling reduces to $\lceil n_G/m \rceil + 1$, which yields $m \geq n_G/(Q-1)$. When $m > n_B$, we have $\min(m, n_B) = n_B$ and the ceiling reduces to $\lceil n/m \rceil$, which yields $m \geq n/Q$. Taking the smallest integer m in each case gives the piecewise minimum in Equation 20.

$$m_1^*(n_G) = \begin{cases} \lceil n_G/(Q-1) \rceil & \text{if } m_1^* \leq n_B, \\ \lceil n/Q \rceil & \text{if } m_1^* > n_B. \end{cases} \tag{20}$$

Both cases require $Q \geq 2$, that is $D - \delta \geq 2C_G$, so that a pod has time for at least two worst-case jobs before the deadline. If $Q = 1$, there's only enough time for one full C_G before the deadline, thereby requiring one pod per task to provide a hard guarantee to guaranteed jobs, since even one best-effort job blocking it just before it arrives can cause it to miss a deadline. If $Q = 0$, then the task parameters are infeasible.

Note that m_1^* is conservative. It is derived from the worst-case arrival pattern, in which guaranteed jobs arrive simultaneously and a full batch of best-effort jobs starts executing just before. When task phases φ_i are known, the actual burst seen at any instant may be smaller, and accounting for the task phases can yield a smaller pod count. We leave this to future work.

For the combined provisioning measure, we use a capacity heuristic: the per-period demand $n_G \cdot C_G + n_B \cdot C_B$ must fit within the pod-time $m \cdot D$ available before the deadline. Substituting $n_B = n - n_G$ and rearranging gives the inequality $nC_B + n_G(C_G - C_B) \leq mD$ whose smallest pod count is given in Equation 21.

$$m_2^*(n_G) = \left\lceil \frac{n\,C_B + n_G\,(C_G - C_B)}{D} \right\rceil \tag{21}$$

Both conditions must hold simultaneously, so we provision with the maximum of both.

$$m^*(n_G) = \max(m_1^*(n_G), m_2^*(n_G)) \tag{22}$$

Both m_1^* and m_2^* are non-decreasing in n_G, so m^* is non-decreasing as well. m^* stays constant across consecutive ranges of n_G and jumps to a new value only when n_G crosses certain thresholds, which can be computed directly from task parameters. We exploit this property with our autoscaler implementation so that the pod count doesn't have to be recomputed with every priority change.

V. Implementation

We implemented the system with three main components: a per-device client that sends control offloading requests; a server that contains the scheduler and autoscaler; and a worker that executes the offloaded controller, running as the sole process in a pod. Figure 2 shows the components and

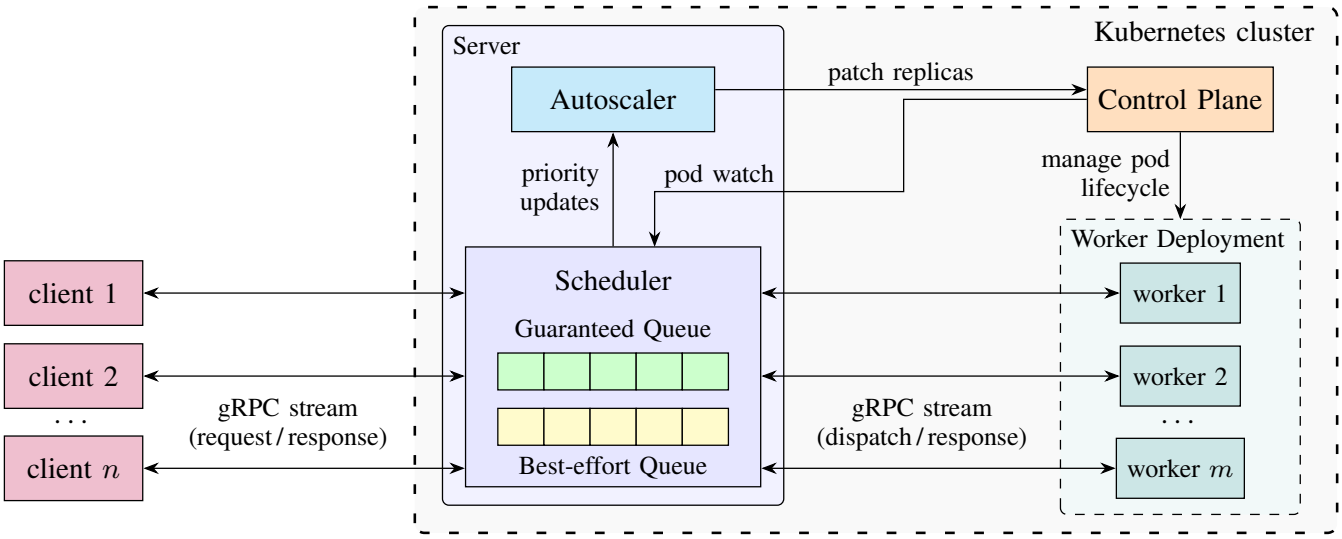


Fig. 2. Implementation overview.

their communication paths. The server process is deployed alongside the workers on a Kubernetes cluster, with the worker pods managed as a Kubernetes Deployment whose replica count the autoscaler adjusts at run time. Clients connect to the server from outside the cluster.

The mapping between the model in Section IV and the implementation is direct. Each device runs one client corresponding to one periodic task, and each control offloading request sent by the client corresponds to a job. Each pod runs one worker and executes at most one job at a time non-preemptively, corresponding to one processor. The scheduler maintains the two FIFO queues from the model and dispatches guaranteed jobs before best-effort jobs. The autoscaler tracks the pod count given by Equation 22 as a live replica count.

A. Physical Car Model

The physical simulation model of the client is a linearized car-model, as in the illustration from Figure 1. Each car is equipped with two internal controllers, one for speed control (output a_t) and one to control steering wheel angle (output δ_t). Denote the car's deviation from a straight reference path $e_{y,t}$, the orientation angle from going straight $e_{\psi,t}$ and the speed deviation from v_{ref} as $e_{v,t}$. The discrete dynamics of the car can then be modeled according to

$$\begin{aligned} e_{y,t+1} &= e_{y,t} + T\left(v_{\text{ref}} + e_{v,t}\right)\sin(e_{\psi,t}), \\ e_{\psi,t+1} &= e_{\psi,t} + T\left(\frac{v_{\text{ref}} + e_{v,t}}{\ell}\tan(\delta_t)\right), \\ e_{v,t+1} &= e_{v,t} + T\left(c_a a_t - c_v e_{v,t}\right), \end{aligned} \tag{23}$$

where T is the sampling time (period), l is the length of the wheelbase and c is a damping coefficient. Linearizing the model around $(e_{y,t}, e_{\psi,t}, e_{v,t}) = (0, 0, 0)$ and incorporating the one-step delay gives the model structure from (2) with

$$\begin{aligned} x_t^P &= \begin{bmatrix} e_{y,t} \\ e_{\psi,t} \\ e_{v,t} \end{bmatrix}, \quad A^P = \begin{bmatrix} 1 & Tv_{\text{ref}} & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 - Tc_v \end{bmatrix}, \\ B^P &= \begin{bmatrix} 0 & 0 \\ T\frac{v_{\text{ref}}}{\ell} & 0 \\ 0 & hc_a \end{bmatrix}, \quad u_t = \begin{bmatrix} \delta_t \\ a_t \end{bmatrix}, \quad w_t^P = \begin{bmatrix} w_{y,t} \\ w_{\psi,t} \\ w_{v,t} \end{bmatrix}, \end{aligned} \tag{24}$$

Even though a car is highly nonlinear, we use a linear simulation model with a relatively large disturbance set that accounts for smaller unmodeled nonlinearities and individual variations between devices. The motivation behind this is both to account for variations between devices and that the robust control literature on disturbed linear systems is huge, with many useful results. But most of all, advanced car modeling is outside the scope of this work.

B. Local Controllers

As described above, each car is equipped with local PID controllers for speed and steering control. The speed controller is a pure PI-controller that measures the speed deviation $e_{v,t}$ and actuates on the acceleration a_t. The steering wheel controller is a PID controller, where the P- and I-parts are calculated from $e_{y,t}$, but where the derivative is replaced with $e_{\psi,t}$ since it highly correlates with the derivative of $e_{y,t}$. The PID controllers are implemented to satisfy Assumption 1.

C. Remote Controller

As remote robust controller, we use Tube Model Predictive Controller (Tube MPC). At a high level, Tube MPC optimizes a nominal disturbance-free trajectory, like ordinary MPC, but surrounds this trajectory by a precomputed tube that contains all possible deviations caused by bounded disturbances. This fits the RC-MC framework well, since the remote controller

should both improve control performance and provide robustness guarantees according to Definitions 2 and 3. The nominal disturbance-free model used inside the MPC is

$$z_{t+1} = Az_t + Bv_t, \tag{25}$$

where $z_t \in \mathbb{R}^{n+m}$ is the nominal state and $v_k \in \mathbb{R}^m$ is the nominal input. At each sampling instant, the remote controller solves the following nominal MPC problem with tightened constraints:

$$\begin{aligned} \min_{v_0,\ldots,v_{N-1}} \quad & \sum_{i=0}^{N-1} \begin{bmatrix} z_k \\ v_k \end{bmatrix}^\top \begin{bmatrix} Q & S \\ S^\top & R \end{bmatrix} \begin{bmatrix} z_k \\ v_k \end{bmatrix} + z_N^\top P z_N \\ \text{Subject to: } & (25), \\ & z_i \in \chi_r \quad \forall i = 1, \ldots, N-1, \\ & v_i \in \mathcal{U}_r \quad \forall i = 0, 1, \ldots, N-1, \\ & z_N \in \chi_{f,r}. \end{aligned} \tag{26}$$

Here, χ_r, $\mathcal{U}_r$ are tightened versions of the original state and input constraints χ_{off} and $\mathcal{U}$. The tightened terminal set $\chi_{f,r}$ is assumed to be chosen such that the MPC problem is feasible on $z_k \in \chi_{\text{off}}$. The tightening of constraints accounts for all possible disturbances $w_t \in \mathbb{W}$. After solving (26), the applied input is

$$u_t = v_t + K(x_t - z_t), \tag{27}$$

where K is a precomputed feedback gain to keep the actual trajectory close to the original one. The matrices Q, S, R, and P correspond to the cost matrices of the nominal MPC problem. For a deeper explanation of how Tube MPC works and what is meant by the tightened constraints, we refer to textbooks within the field, e.g. [10].

We choose Q, S, R and P such that the unconstrained and undisturbed MPC solution exactly matches the PID controller according to the methodology in [11], to further emphasize that the remote controller provides extra robustness performance and guarantees only when needed. The value of K is

$$\begin{bmatrix} -0.068 & -0.53 & 0 & 0.01 & 0 & -0.19 & 0 \\ 0 & 0 & -2.35 & 0 & 1.28 & 0 & -1.0 \end{bmatrix},$$

where the extra two states are augmented integral states of $e_{y,t}$ and $e_{v,t}$ needed to match the PID controllers.

To see why Tube MPC gives robustness, define the deviation from the nominal trajectory as $e_k = x_k - z_k$. Then from (25) and (2), we have

$$e_{k+1} = (A - BK)\, e_k + w_k.$$

Since the disturbance w_k is assumed to be bounded, and $A - BK$ chosen to be stable, the deviation remains inside a bounded tube. We compute this tube offline by iterating

$$\mathcal{E}_{k+1} = (A - BK)\, \mathcal{E}_k \oplus W, \quad \mathcal{E}_0 = \{0\}, \tag{28}$$

until convergence. Denote the resulting invariant error set by $\mathcal{E}_\infty$. Then from an initial state x_0, all future states are guaranteed to lie in the tube with center from the solution of (26), and with tube intersection $\mathcal{E}_\infty$. This is illustrated in Figure 3, where the guaranteed reachable sets with Tube MPC is within a narrow tube, while the local controller does not have the same robustness guarantees.

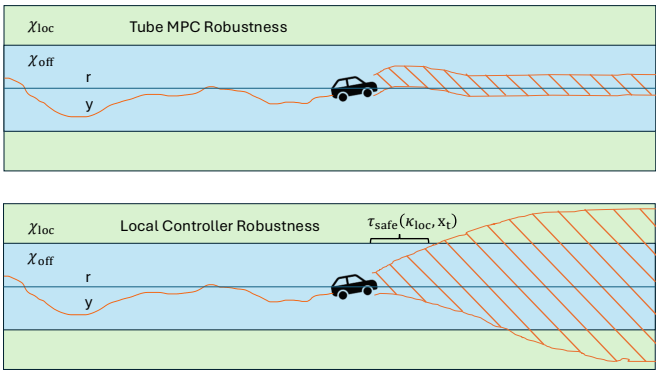


Fig. 3. Sketch of compared robustness margins between the local controller and the Tube MPC. In both scenarios, the same control strategy have led the agents to their current position. The shaded region correspond to respectively reachable state set under all possible disturbances from our disturbance set under respective control law.

The set χ_{off} is chosen as the robust feasible set of the Tube MPC. Under the standard Tube MPC conditions of a robust invariant error tube, tightened constraints, and an invariant terminal set, this feasible set is robustly positive invariant under the Tube MPC law, satisfying Assumption 2.

The remote control structure follows the procedure below when a request arrives:

1) Calculate the Tube MPC control signal.
2) If the request is guaranteed, calculate if the next request is best-effort according to Definition 3.
3) If the request is best-effort, calculate the time until the task becomes critical under worst-case disturbances according to Definition 2.

D. Request Path and Priority Scheduling

Each worker registers itself to the server when it is ready to receive requests. The scheduler considers a worker as available for dispatch after this registration completes, rather than waiting on Kubernetes pod readiness. The server establishes a persistent bidirectional gRPC stream to each worker after worker registration, and each client establishes a persistent bidirectional gRPC stream to the server. As clients send requests through the client streams, the scheduler places them on the appropriate queue, and eventually forwards them through the worker streams. Responses use the same streams in reverse.

The scheduler enforces the assumption made in Section IV that there are at most one queued job per task by two complementary rules: a new request from a client cancels any queued request from the same client; any queued request still in the queue past its drop time $D_{i,k} - C_B$ is removed at dequeue.

The client's priority classification is supplied by the worker. Each worker response carries the client's current priority and a prediction of when the priority changes. The worker is the authoritative source because it has the view of the client's

control-side state. This keeps the scheduler unaware of the control-side state and allows for updates on the scheduling priority as the controller produces new predictions.

The scheduler applies the priority classification on request arrival by placing the request on the appropriate FIFO queue. When a worker becomes available, the scheduler dispatches to it a request from the guaranteed queue if it is not empty and from the best-effort queue otherwise.

Priority scheduling can be enabled or disabled at compile time. When disabled, the scheduler still tracks the priority classification of requests but puts them into a single FIFO queue and dispatches them in arrival order. We use this configuration as a comparison in our evaluation.

E. Horizontal Pod Autoscaling

The autoscaler takes the stream of priority updates forwarded by the scheduler, and from it outputs the target replica count of the worker deployment using Equation 22.

From the priority updates, the autoscaler projects the future number of guaranteed clients n_G over two look-ahead windows rather than reacting to the current n_G. We set the length of these windows through environment variables passed to the server's Kubernetes Deployment, which the server reads at startup.

The autoscaler uses the first window to decide scale-up. When the maximum n_G within this window raises the target replica count, the autoscaler immediately increases the replica count to this target. The length of this window must be set to the maximum time that it takes to scale up the replica count with some buffer so that the new pods are available within the time that they are required.

The autoscaler uses the second window to decide scale-down. When the maximum n_G within this window results in a target replica count that is lower than the current replica count, the autoscaler reduces the replica count to that target. The purpose of this window is to suppress rapid scale-down immediately following a scale-up, avoiding oscillations when n_G fluctuates rapidly.

The target replica count can change on two kinds of events: a priority update that causes n_G to cross a threshold, or a client connecting or disconnecting. The autoscaler wakes only when n_G crosses a threshold within the two look-ahead windows or when a client connects or disconnects. When it wakes, it recomputes m^* and the n_G thresholds and then sets the new replica count based on the computed m^*.

The autoscaler controls the worker Deployment replica count through a patch on the Deployment's replica field. Kubernetes then handles the creation and termination of worker pods. The scheduler watches the pod lifecycle events through the Kubernetes API, so that it can track which workers are to be terminated. Workers terminated during scale-down enter a draining state in which they are not dispatched any new jobs but are allowed to finish their current job so that in-flight jobs are not interrupted when scaling down.

The autoscaler's target can be overridden at run time to a fixed replica count. We use this functionality in our evaluation.

VI. Evaluation

We evaluated the system on three identical machines, each equipped with an Intel Core i7-14700 processor, 16 GB DDR5 memory, and running Ubuntu 24.04. Two machines formed a two-node Kubernetes v1.30 cluster: one serving as the control plane and one as the worker node on which all worker pods were scheduled. The third machine ran the clients. All three machines were connected over a local area network. We disabled Intel SpeedShift and restricted C-states to C1 on all machines to make execution times more consistent.

Table I lists the scheduling, controller, and disturbance parameters that remain fixed across all experiments. Only the pod count m and the budget C_B vary between configurations.

TABLE I
FIXED EVALUATION PARAMETERS.

Group	Parameter	Value
Scheduling	Period T	500 ms
	Deadline D	500 ms
	WCET C_G	210 ms
	Arrival jitter δ	10 ms
	Scale-up window	3.5 s
	Scale-down window	6 s
Controller gains	Speed controller P-gain	0.45
	Speed controller I-gain	0.06
	Steering controller P-gain	0.006
	Steering controller I-gain	0.00004
	Steering controller D-gain	0.07
Model parameters	Reference speed v_{ref}	3 m/s
	Wheelbase ℓ	1.5 m
	Speed damping c_v	0.3 s^{-1}
	Lateral disturbance $w_{y,t}$	[-0.015, 0.015] m/s
	Steering disturbance	[-0.045, 0.045] rad/s
	Lateral disturbance	[-0.25, 0.25] m/s^2
Safety bounds	Lateral error $e_{y,t}$	[-3, 3] m
	Heading error $e_{\psi,t}$	unbounded
	Speed error $e_{v,t}$	[-2, 2] m/s
	Steering command δ_t	[-0.12, 0.12] rad
	Acceleration command a_t	[-0.5, 0.5] m/s^2

The workload consists of n clients that each issue requests at the fixed period T for the duration of the run, which is 250 seconds (500 steps). Disturbances are drawn from the same fixed seed in every iteration so that iteration-to-iteration variance reflects runtime nondeterminism (e.g., scheduling jitter, network, pod startup) rather than disturbance variability.

We characterized the request computation time by running the workload with 100 clients and 10 worker pods, under two conditions: with and without sustained interference. We induce sustained interference on the worker node using stress-ng with 15 CPU stressors. Table II shows the resulting computation time distribution.

TABLE II
PER-REQUEST COMPUTATION TIME (MS).

Condition	mean	p50	p95	p99	max
Without interference	32.57	28.9	52.65	55.13	205.09
With interference	63.24	55.64	103.74	157.46	170.16
All	45.82	37.61	89.47	137.97	205.09

A. Policy Comparison

The policy comparison experiment is a cross of five pod provisioning configurations against two dispatch modes, resulting in 10 test cases. The pod provisioning configurations consist of three static pod counts $m \in \{7, 9, 11\}$, and two autoscaler settings $C_B \in \{35, 45\}$ ms. $C_B = 35$ ms is slightly above the average execution time without interference (32.57 ms), while $C_B = 45$ ms matches the overall average execution time (45.82 ms).

We ran this experiment with an interference cycle of 30 s without stress-ng and 30 s with stress-ng 15 CPU stressors.

1) Procedure: We conducted the experiment using a script that executes the following procedure per iteration of a test case:

(a) **Warm-up.** Scale the worker deployment to a maximum replica count (20), start the interference application cycle, and start all n clients. Hold this state for a warm-up period of 5 seconds.
(b) **Policy switch.** For static cases, set the pod count to the target m. For autoscaler cases, release the override so the autoscaler controls the replica count from the initial maximum.
(c) **Measurement window start.** Flush and truncate the data files so that the measurement window does not include the warm-up period.
(d) **Measurement.** Let the clients run for the remainder of the configured run duration.
(e) **Measurement window end.** Issue a flush 5 seconds before the first client is expected to disconnect.
(f) **Teardown.** Wait for the clients to finish, stop the interference cycle, and then fetch the data files.

The warm-up period prevents missed requests at the start of the iteration when we are testing the autoscaler, since it would start with 1 pod otherwise. We flush before the first client disconnects, so that the measurement window ends before client disconnections begin. This leaves out scale-down measurements resulting from client disconnections from the collected data. Both measures also apply to the static test cases for consistency.

2) Results: Table III reports the aggregate results in all 10 configurations. We now discuss the contribution of priority dispatch and dynamic provisioning.

Without priority dispatch, all configurations starve guaranteed-state requests because the shared FIFO queue gives them equal treatment to best-effort requests. With priority dispatch enabled, the guaranteed success rate increases to 100% across all configurations. This identifies priority dispatch as the main mechanism for ensuring that all guaranteed requests are processed successfully. Even an under-provisioned static configuration delivers 100% success on guaranteed requests once priority dispatch is in place. Extremely low pod counts may still result in failed guaranteed requests, but this is not covered in our experiment.

Without priority dispatch, every static configuration leaves the guaranteed success rate below 8%, while both autoscaler settings deliver roughly twice that rate (12.9% and 13.9%) at a lower average pod count than static $m = 11$. We attribute this to the autoscaler provisioning extra capacity in anticipation of guaranteed-state clients rather than holding the pod count constant.

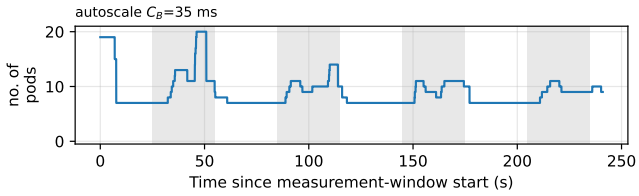


Fig. 4. Pod count over the measurement window for the autoscaler at $C_B = 35$ ms with priority dispatch. Shaded regions mark periods with interference. The autoscaler sits scales up during interference.

Figure 4 shows the autoscaler reacting to the interference cycle. The pod count sits at the autoscaler's floor during quiet phases and scales up during interference. A static configuration cannot adapt this way. A static deployment must commit to a pod count that either over-provisions for the quiet phase or under-provisions for interference. The autoscaler does not have this tradeoff.

B. Low C_B Behavior

Our system imposes $C_B < C_G$ but otherwise leaves a wide range of possible values. In the main evaluation experiment, C_B values below the overall average execution time still produced robust behavior with priority dispatch, which raises the question of whether the framework remains well-behaved at even lower C_B values. We examined this without interference to isolate the effect of C_B.

We ran the autoscaler with $C_B \in 15, 25, 35$ ms for 150 seconds (300 steps) each, with no interference applied to isolate the effect of C_B. Figure 5 shows the resulting pod count.

The bottom panel shows that the projected guaranteed count rises and falls repeatedly while the realized count stays at zero throughout. The autoscaler is therefore reacting to the projection rather than to actual proximity to the guaranteed state..

The projection drift is the result of a feedback loop. When C_B is set below the AET, the floor pod count m_2^* cannot serve all incoming requests, and the scheduler drops some best-effort requests. A dropped request produces no response from the remote controller, so the scheduler retains the most recent safe-time prediction for that client.

As time passes, this stale prediction makes the client appear to be approaching the guaranteed state, even though the client's actual state has not changed. The projection within the autoscaler's look-ahead window therefore rises, and the autoscaler scales up. The added pods allow the system to serve more requests, letting the controller produce fresh predictions that push the predicted transition to the guaranteed state further

TABLE III

POLICY COMPARISON ACROSS THE 10 CONFIGURATIONS. SUCCESS REFERS TO THAT REQUESTS ARE SERVED BEFORE THEIR DEADLINES. A CROSSING MEANS THAT THE DEVICE CROSSES THE BOUNDARY AND LEAVES χ_{SAFE}. SAMPLE VIOLATIONS IS THE NUMBER OF SAMPLES OUTSIDE χ_{SAFE}.

Type	Configuration	Priority	Overall Success %	Guaranteed Success %	Avg. pod count	Crossings	Sample Violations
Static	$m = 7$	on	72.2	100.0	7	0.0	0.0
		off	72.3	7.5	7	4.6	105.8
	$m = 9$	on	67.3	100.0	9	0.0	0.0
		off	67.3	7.1	9	2.2	28.6
	$m = 11$	on	89.2	100.0	11	0.0	0.0
		off	81.6	6.6	11	0.6	7.6
Autoscaler	$C_B = 35$ ms	on	82.4	100.0	9.77	0.0	0.0
		off	81.1	13.9	9.90	0.0	0.0
	$C_B = 45$ ms	on	88.4	100.0	10.03	0.0	0.0
		off	88.2	12.9	10.17	0.0	0.0

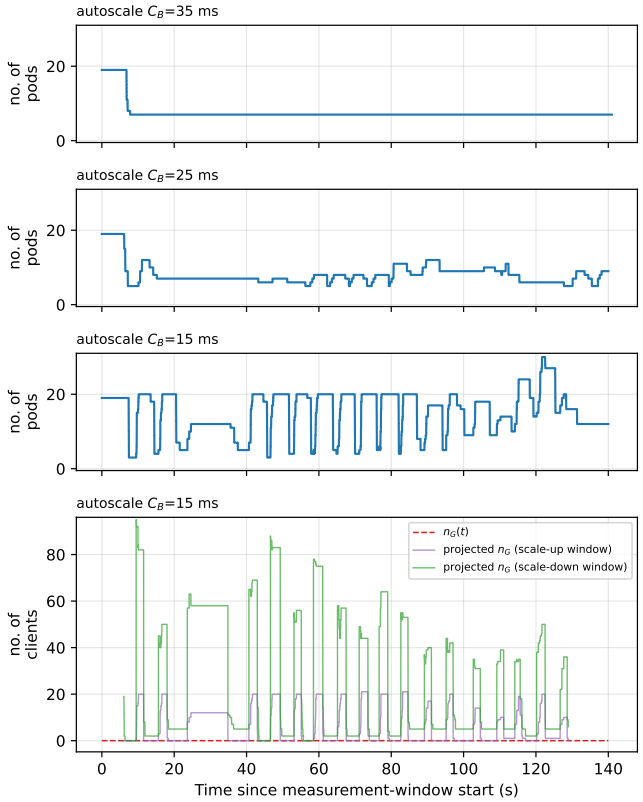


Fig. 5. Autoscaler behavior in isolation at $C_B \in 35, 25, 15$ ms. The pod count oscillates more aggressively as C_B decreases. The bottom panel shows that for $C_B = 15$ ms, the realized n_G stays near zero throughout while the projected n_G rises and falls, driving the autoscaler's response.

out, and the projection falls. The autoscaler scales back down to the floor and the cycle repeats.

The link between failed requests and guaranteed-state transitions is visible in per-request data from a separate experiment with 5 worker pods and a sweep of 60, 70, 80, 90, and 100 clients (Figure 6). At low C_B, this pattern drives the oscillation in Figure 5.

The same projection mechanism drives the autoscaler's productive response to interference in Section VI-A. There, failed requests cause scale-ups, and the added capacity stabilizes the

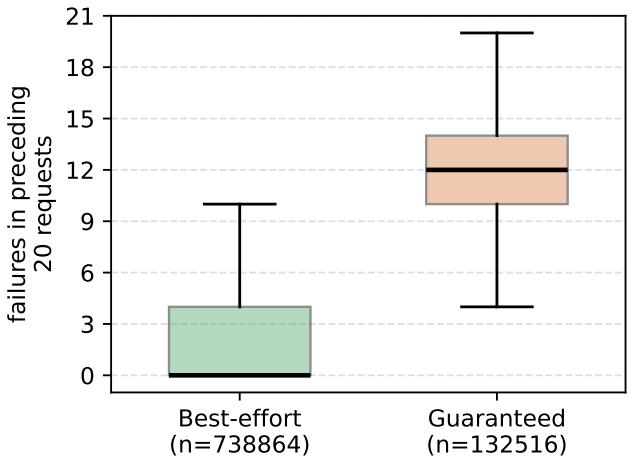


Fig. 6. Preceding-failure counts of successful requests, grouped by request state. Guaranteed-state requests are concentrated at higher preceding-failure counts compared to best-effort requests.

success rate against increased computation demand induced by the interference. The oscillating behavior is caused by the same mechanism but without an external load change to absorb. The scale-ups address an under-provisioning that returns once the autoscaler scales down, so the system cycles indefinitely.

VII. LIMITATIONS AND FUTURE WORK

The guarantee given by our system relies on the autoscaler being able to deliver new pods before predicted state transitions occur, which requires the scale-up window to be no shorter than the worst-case scale-up latency of the deployment. In our cluster, we observed a maximum latency of 2.5 s. The cluster has a single worker node that always runs at least one worker pod, so the worker image is permanently present in the node's container cache and additional pods start without an image pull. Production deployments can exhibit substantially longer scale-up latencies.

A question for future work is how to amortize this deployment-level latency without giving up the resource savings that motivate dynamic provisioning in the first place. One possible mechanism is to maintain a buffer pool of pods on top of the autoscaler's target, large enough to cover guaranteed

request demands that arrive while new pods are still starting. Given a bound on the worst-case scale-up latency, what is the minimum buffer pool size that preserves our system's guarantee?

The pod count m_1^* from Section IV is derived from the worst-case arrival pattern. A smaller pod count that preserves the guarantee is possible when accounting for task phases. Deriving a phase-aware bound is an extension worth exploring.

VIII. Conclusion

This work-in-progress investigates whether robust control guarantees can be preserved in a remote offloading setting without requiring the devices or scheduler to explicitly know the underlining robustness requirements. We propose a framework that combines robust control with a modified mixed-criticality model, define a minimal interface between devices, scheduler, and remote workers, and derive rules for when requests must receive guaranteed service and how many workers are required to preserve robust guarantees.

The implementation shows that the proposed framework can be realized in a simulated offloading scenario. The auto-scaler used in the evaluation provisions resources for guaranteed requests according to their WCET and for best-effort requests according to their AET. This conservative policy avoided all control requirement violations in the considered scenario. However, the resulting pod count was higher than expected, and a static configuration with fewer pods also preserved safety. Hence, the evaluation supports the feasibility of the framework, but does not establish that the implemented auto-scaler is resource-optimal.

Overall, the results suggest that the proposed RC-MC framework is a promising basis for reducing conservatism in safety-critical control offloading. At the same time, they show that the achievable resource savings depend strongly on the scheduling and auto-scaling policy. Future work will therefore focus on less conservative scheduling and auto-scaling strategies, and on evaluating their impact on both resource usage and closed-loop performance.

References

[1] S. Vestal, "Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance," in *28th IEEE international real-time systems symposium (RTSS 2007)*, pp. 239–243, IEEE, 2007.

[2] A. Burns and R. Davis, "Mixed criticality systems-a review," *Department of Computer Science, University of York, Tech. Rep*, pp. 1–69, 2013.

[3] L. Cheng, K. Huang, G. Chen, B. Hu, and A. Knoll, "Mixed-criticality control system with performance and robustness guarantees," in *2017 IEEE Trustcom/BigDataSE/ICESS*, pp. 767–775, IEEE, 2017.

[4] T. Rheinfels, M. Gaukler, and P. Ulbrich, "A new perspective on criticality: Efficient state abstraction and run-time monitoring of mixed-criticality real-time control systems," in *35th Euromicro Conference on Real-Time Systems (ECRTS 2023)*, pp. 11–1, Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2023.

[5] K. Zhou and J. C. Doyle, *Essentials of robust control*, vol. 104. Prentice hall Upper Saddle River, NJ, 1998.

[6] P. Pazzaglia, K. Schmidt, L. Beermann, D. Ziegenbein, and A. Hamann, "Invited paper: Physics-driven real-time CPS challenge," in *32nd IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS 2026, Saint Malo, France, May 11-14, 2026*, pp. 222–233, IEEE, 2026.

[7] R. Schneider, D. Goswami, A. Masrur, M. Becker, and S. Chakraborty, "Multi-layered scheduling of mixed-criticality cyber-physical systems," *Journal of Systems Architecture*, vol. 59, no. 10, pp. 1215–1230, 2013.

[8] P. Skarin, J. Eker, and K.-E. Årzén, "Cloud-based model predictive control with variable horizon," *IFAC-PapersOnLine*, vol. 53, no. 2, pp. 6993–7000, 2020.

[9] P. Skarin, J. Eker, and K.-E. Årzén, "A cloud-enabled rate-switching mpc architecture," in *2020 59th IEEE Conference on Decision and Control (CDC)*, pp. 3151–3158, IEEE, 2020.

[10] J. B. Rawlings, D. Q. Mayne, and M. M. Diehl, "Model predictive control: theory, computation, and design," *(No Title)*, 2020.

[11] M. Zanon and A. Bemporad, "Constrained controller and observer design by inverse optimality," *IEEE Transactions on Automatic Control*, vol. 67, no. 10, pp. 5432–5439, 2021.

Scalable Resource Allocation of Control Offloading for Next Generation Networks

Emil Sundström
Department of Automatic Control
Lund University
Lund, Sweden
emil.sundstrom@control.lth.se

Johan Eker
Department of Automatic Control
Lund University
Lund, Sweden
johan.eker@control.lth.se

***Abstract*—This article addresses the resource allocation problem for many mobile agents across multiple network cells when control loops are dynamically offloaded to remote servers. By focusing on scalability rather than optimality, we propose a decentralised allocation policy in which each device makes its own offloading decisions, without global system knowledge. Motivated by pre-granted scheduling and pre-emption mechanisms in emerging Ultra-Reliable Low-Latency Communications (URLLC) networks, we argue for a simplified network model that allows the upcoming offloading dynamics to be cast as a weighted potential game. Under this formulation, we prove that the total number of strategy updates grows linearly with the number of devices, and that this convergence rate is preserved even when incorporating practical control aspects such as fail-safe switching to local computation to manage weakly-hard deadline constraints.**

***Index Terms*—Offloading, URLLC, 5G, Resource Allocation, Game Theory**

I. Introduction

Advanced control algorithms such as non-linear model predictive control (NMPC) have high computational needs which makes them unsuitable for execution on resource constrained devices. To that end, wireless compute offloading offers an interesting opportunity and such services are likely to be available in the next generation of telecommunication systems.

The research on offloading has been an active field since the late 90s, from initially focusing on the feasibility of offloading, toward studies on infrastructure and decision-making mechanisms [1]. At the same time, cloud robotics has developed as a parallel line of research, demonstrating the potential of remote compute resources in control and robotics systems [2]. Much work has been done in the area of resource management, typically by solving highly complex optimization problems, and is still seen as an open research problem [3].

This paper also addresses resource management. However, we are limiting the use case to control loops and focusing on quickly finding a good-enough solution. Control loops differ from other offloading use cases in that the size of data sent over the air is typically small and constant, e.g., a state vector and control signal in NMPC, compared to a video processing service which may require a large bandwidth for uplink and vary from frame to frame.

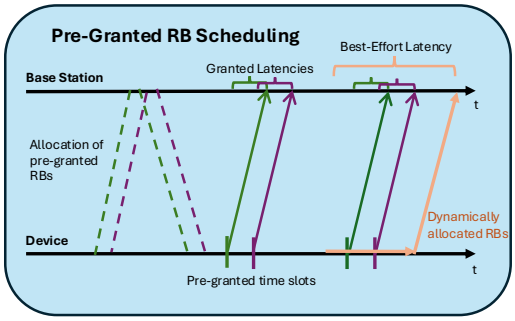
A key part of this article is the motivation for a simplified network model applicable through pre-granted resource block (RB) scheduling in the next generation telecommunication systems. Therefore, the core design objective of the proposed resource allocation policy, which is emphasized throughout the paper, is scalability rather than optimality. We consider highly dynamic network environments, where devices continuously enter and leave network cells, in which resource allocation must be responsive and fast. The driving use case throughout the paper is UAVs, often appearing in large clusters spread over multiple network cells.

Cascading best-response updates, where an updated offloading policy of one device leads to a long chain of decision updates from other agents, cannot be tolerated. Therefore, a good resource allocation policy must be resistant to these cascading responses. As is demonstrated in this article, relaxing the optimality requirement enables highly scalable and low-latency allocation, with a total number of policy updates that grows only linearly with the number of contending agents, without cascaded best-response updates.

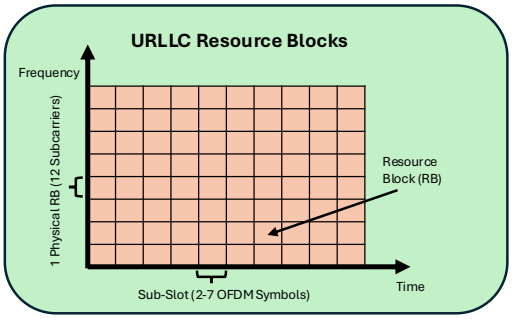
A practical aspect of decentralised large-scale offloading of control loops is how fail-safe offloading fallback decisions, i.e. to stop offloading after x missed deadlines, affect the dynamics of the system. We consider that issue in this paper, and show that local fallback decisions to prevent misses of weakly-hard deadline constraints [4] do not interfere with the linear scalability.

The problem of resource allocation for offloading has been extensively explored in the literature. For example, [5], and [6] propose different optimization methods to determine how both network and computing resources should be shared among agents. [7] studies how an approximation of an Integer Linear Programming (ILP) formulation can solve the resource allocation problem, while [8] proposes auction-based methods. Game theoretical approaches have also been applied to real-time-offloading previously [9]–[11]. However, many approaches for generic mobile agents employ network models more complex than necessary for control offloading.

This work was partially supported by the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation. The authors are members of the ELLIIT Strategic Research Area at Lund University.



(a) 5G pre-granted uplink scheduling and pre-emption of a job corresponding to non-critical dynamically allocated RBs. The orange non-critical RBs gets pre-empted by the green and purple pre-granted URLLC-scheduled RBs.



(b) How resources are divided in 5G, with emphasis of the RB sizes in URLLC. The resource split in both frequency and time components is a key part of how URLLC can reliably achieve the promised low latencies.

Fig. 1: Sketch of scheduling mechanisms of pre-granted resource blocks (RB) and pre-emptions in 5G URLLC, with RB sizes. The Figure is created with inspiration from [12] and [13].

A. *Contributions*

While there is extensive work on game-theoretic and distributed computation offloading methods, and some work on offloading for real-time and weakly-hard tasks, the literature typically targets generic applications and models network congestion rather than exploring the near-deterministic behaviour from pre-granted RB scheduling. It also rarely handles practical controller implementation aspects, such as how safe-guarding mechanisms against deadline-misses affect the scalability of the distributed resource allocation policy.

To the best of our knowledge, there is no prior decentralised offloading mechanism that is explicitly tailored to periodic control computations, motivates (with pre-allocated RB scheduling) the usage of a simplified network model to obtain a weighted potential game with a simple threshold-based best response, and integrates weakly-hard deadline constraints via a local fail-safe while still retaining linear scalability. This is the specific gap in the literature addressed in this work.

II. Network Model Simplification

In a game-theoretical setup, each agent acts according to a set of policies. In the offloading scenario, these policies are based on models of both the network and servers. A simpler model might make the convergence and scalability behavior of the game tractable, but with sacrificed optimality due to larger trade-off between model and reality.

The network model used throughout this paper relies on low-latency features found in 5G, such as Ultra-Reliable Low-Latency Communication (URLLC) [13], which allows for communication latency down to around 1 ms. 5G also offers pre-granted uplink and pre-emption, which minimises the negative effects of congestion and interference.

Figure 1 shows two important aspects of pre-granted resource scheduling in URLLC. Figure 1a shows a sketch of pre-granted scheduling and pre-emption, where each device is guaranteed a scheduled uplink slot. Dynamically allocated uplinks gets pre-empted and priority is given to the granted pre-scheduled devices. Figure 1b shows how resources are split up in frequency components and time components simultaneously in 5G.

The scheduling approach from Figure 1 is specifically applicable to real-time control offloading, where the timing demands are strict but the bandwidth requirements are small. Assuming that the radio interface (NR in 5G) is the bottleneck of the network, we can model the Round-Trip Time (RTT) for each agent as measurable but independent of actions from other agents in the game. Of course, internet is filled with all kinds of traffic and does therefore give a varying RTT for the agents. But the assumption that the agents within the game does not cause congestion for each other is a crucial assumption for our game theoretical approach.

III. System Model

Figure 2 illustrates a typical scenario for where the assumptions in this article are relevant. A number n of mobile devices are operating on a mission, and each agent can choose to compute its task locally or on a server. These agents are located in different network cells, where each cell corresponds to one antenna. In Figure 2, the network cells are denoted with m. The system also includes a number of servers. The servers can be of two types: edge or cloud. Both types of servers can host an unlimited number of devices, but edge servers have to share the computational power between all connected agents. In the figure, servers are denoted with a.

A. *Device Model*

Throughout this article, the terms device and agent are used interchangeably, both referring to a device defined in this section. A mobile device i is assumed to have a specific amount of computational capacity onboard and perform periodic control tasks with period T_i and deadline D_i. Let s_i be a positive device-specific size factor that scales the amount of required computations from a nominal (baseline) task. The purpose of s_i is to allow for different devices to have different complexity

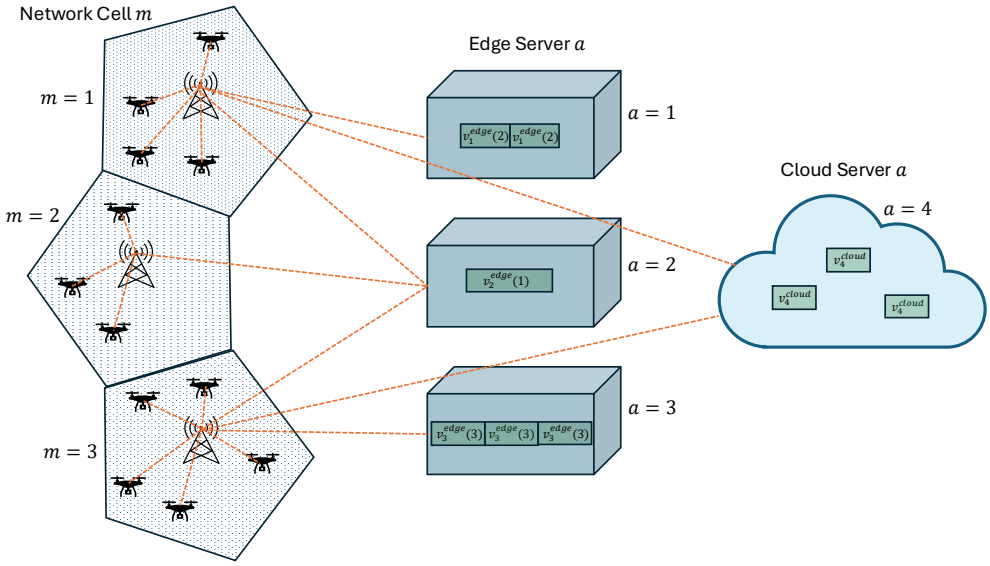


Fig. 2: A sketch of the system model. A number of agents, sketched as UAVs, located in different network cells are in a configuration where three of the agents offload to a cloud server, six offload to edge servers and the rest compute locally. The dashed orange lines represent connections to available servers from a network cell, $v_a^{edge}(n_a)$ denotes the time for edge server a to compute a nominal task if n_a other devices are offloading to it, and v_a^{cloud} is the time for cloud server a to compute a nominal task.

of their control algorithms. If agent i computes locally, the time to finish is

$$t_i = v_i^{local} s_i, \tag{1}$$

where v_i^{local} is the time for the local controller to finish a nominal control task. All devices are assumed to locally have positive slack, meaning local computational capacity enough for the task response time t_i to be shorter than the deadline D_i. Offloading is in this context motivated by performance improvement, rather than feature extension.

B. Edge Server Model

The resources in the edge servers are assumed to be shared evenly between the agents that are using it. For a nominal control task, the time to perform the computation on edge server a is $v_a^{edge}(n_a)$. We do not make any assumptions on $v_a^{edge}(n_a)$, more than that it does not depend on which devices that use it, but only on how many (n_a) that are offloading to it. If device i offloads its computations to edge server a, the expected time to finish a job is

$$t_i = \tau_{i,a} + v_a^{edge}(n_a) s_i, \tag{2}$$

where $\tau_{i,a}$ is the RTT from agent i to server a. The factor s_i is exactly the same as for the device model above, i.e. the device-specific size factor that scales the device's task size from a nominal task.

C. Cloud Server Model

When a device offloads its control computations to a cloud server, it is assumed to get a specific amount of resources independent of the other devices. The amount of resources a device allocates in cloud server a is denoted v_a^{cloud}, with the interpretation that it takes the cloud server the time v_a^{cloud} to compute a nominal control task. Hence, If device i offloads to cloud server a, the time to complete the task is then modelled as

$$t_i = \tau_{i,a} + v_a^{cloud} s_i, \tag{3}$$

where $\tau_{i,a}$ is the RTT from agent i to server a.

D. Network Model

To avoid that all devices have to keep communicating with all servers of interest all the time to get information of available capacity, it is assumed that some logic can be put close to the base station. This logic maintains up-to-date records of available capacity and RTT for a few close edge and cloud servers. Due to the type of traffic, we also make the assumption that we have negligible network congestion between agents in the game, motivated in Section II.

E. Decision Model

A feature in the model that enables scalability is that each device is only assumed to have knowledge about some servers located close to it, depending on what network cell it is in. Some applications may also have strict legal requirements for server locations. Therefore, the decision device i in network cell m has to make about where to put its computations is limited to the action space

$$A_i = \{C_i^{local}, ES_m^{q_e}, CS_m^{q_c}\}, \tag{4}$$

where C_i^{local} represents the local processor onboard, $ES_m^{q_e}$ is the set of the q_e closest relevant edge servers to network cell m, and $CS_m^{q_c}$ is the q_c closest relevant cloud servers. Hence, each element in A_i corresponds to local computing or a specific server.

IV. Game Formulation

A relevant question to consider in these types of distributed systems is whether the devices are able to agree on an allocation of resources among themselves or not. In other words, do they reach a configuration where each individual agent, with an individual value function, can not improve its value function by changing the place of its control task? In game theory, this preferable configuration is called a *Nash equilibrium*.

Another interesting question to investigate is the scalability of the system. If the number of devices increases significantly and are distributed over multiple network cells, how does that affect the convergence rate to a Nash equilibrium?

A. Defining a Utility Function

A utility function is a local reward function to each agent, which it maximizes by playing the most rewarding action a (local computing or a specific server), based on the strategies from the other agents $\mathbf{a}_{-i}$. For this game, the utility function is defined as the negative time to perform a control task. Therefore, (1), (2) and (3) give the utility function

$$u_i(a, \mathbf{a}_{-i}) = \begin{cases} -s_i v_i^{local} & \text{If } a \in C_i^{local} \\ -\tau_{i,a} - s_i v_a^{edge}(n_a) & \text{If } a \in ES_m^{q_e} \\ -\tau_{i,a} - s_i v_a^{cloud} & \text{If } a \in CS_m^{q_c}, \end{cases} \tag{5}$$

for agent i in network cell m.

B. Defining the Game

The formal definition of the game can be stated as follows.

Definition 1 (The Game): The offloading game of this paper is defined as $(V, \{A_i\}_{i\in V}, \{u_i\}_{i\in V})$, where V is the set of agents, A_i the set of actions defined in (4) and u_i the utility functions defined in (5).

V. Convergence Guarantees

Convergence guarantees for resource allocation policies are important. By finding a potential function such that the system can be classified as a *weighted potential game*, convergence to a Nash equilibrium is guaranteed.

A. Finding a Potential Function

Our approach to guarantee convergence for the system is to find a potential function of the system, such that the game can be defined as a *Potential Game*. The potential function is denoted $\Phi(\mathbf{a})$, with $\mathbf{a} = (a_1, a_2, \ldots, a_n)$ being a vector representing the strategies of all agents. In this class of games, we are always guaranteed to converge to a Nash equilibrium if all agents play improving moves (actions that increase the value of their utility functions).

To show that a game is a weighted potential game, the difference in utility function when agent i changes strategy multiplied by a weight factor ω_i must always be the same as the corresponding difference in the potential function,

$$\omega_i \Delta u_i = \Delta\Phi. \tag{6}$$

Here, $\Delta u_i = u_i(a_2, \mathbf{a}_{-i}) - u_i(a_1, \mathbf{a}_{-i})$ and $\Delta\Phi = \Phi(\mathbf{a}_2) - \Phi(\mathbf{a}_1)$, with the difference between $\mathbf{a}_1$ and $\mathbf{a}_2$ being that agent i plays a_1 in $\mathbf{a}_1$ and a_2 in $\mathbf{a}_2$.

The problem is normally to find such a potential function. [14] defined a class of games that is known as *congestion games*, a category in which our offloading game from Definition 1 belongs. By defining the non-congestive additive terms in (5) as

$$z_i(a) = \begin{cases} s_i v_i^{local} & \text{If } a \in C_i^{local} \\ \tau_{i,a} & \text{If } a \in ES_m^{q_e} \\ \tau_{i,a} + s_i v_a^{cloud} & \text{If } a \in CS_m^{q_c}, \end{cases} \tag{7}$$

such that

$$u_i(a, \mathbf{a}_{-i}) = -z_i(a) - s_i v_a(n_a) e_a, \tag{8}$$

where $e_a = 1$ if the agent offloads to an edge server ($a \in ES_m^{q_e}$) and $e_a = 0$ otherwise, we can define the candidate potential function

$$\Phi(\mathbf{a}) = -\left(\sum_{j\in ES}\sum_{k=1}^{n_j} v_j(k)\right) - \sum_{i=1}^{n} \frac{z_i(a_i)}{s_i}, \tag{9}$$

where ES is the set of all edge servers in the system and n_j is the number of devices offloading to server j. The first sum iterates over all edge servers and for each server (j), the value of $v_j(k)$ is added for all possible number of agents (k) that could use resource j up to the actual number of offloading devices to that server (n_j). This term is based on theory from congestion games [14], [15].

Theorem 1 (Weighted Potential Game): The non-cooperative game from Definition 1 is a weighted potential game with potential function as in (9) and weights $\omega_i = 1/s_i$.

Proof: To show that (9) is a weighted potential function to the system, (6) must hold for all strategy updates a_1 and a_2. This has to be proven for four scenarios for agent i in network cell m:

1) Agent i changes strategy between two edge servers in $ES_m^{q_e}$ ($a_1 \in ES_m^{q_e}$ and $a_2 \in ES_m^{q_e}$).
2) Agent i changes strategy between local computing C_i^{local} and offloading to an edge server in $ES_m^{q_e}$ ($a_1 \in C_i^{local}$ and $a_2 \in ES_m^{q_e}$).
3) Agent i changes strategy between offloading to an edge server in $ES_m^{q_e}$ and offloading to a cloud server in $CS_m^{q_c}$ ($a_1 \in ES_m^{q_e}$ and $a_2 \in CS_m^{q_c}$).
4) Agent i changes strategy between local computing C_i^{local} and offloading to a cloud server in $CS_m^{q_c}$ ($a_1 \in C_i^{local}$ and $a_2 \in CS_m^{q_c}$).

The four scenarios are only proven in one direction, since the reverse is trivial (multiply by minus one). The procedure is to verify that the only elements that change in the potential function fulfills (6).

Scenario 1):

$$\begin{aligned}\Delta\Phi =&\Phi(\mathbf{a}_2) - \Phi(\mathbf{a}_1) = \\ -v_{a_2}(n_{a_2}+1) + v_{a_1}(n_{a_1}) - \frac{z_i(a_2)}{s_i} + \frac{z_i(a_1)}{s_i} =& \\ \frac{u(a_2,\mathbf{a}_{-i})}{s_i} - \frac{u(a_1,\mathbf{a}_{-i})}{s_i} = \frac{1}{s_i}\Delta u_i,&\end{aligned} \tag{10}$$

Scenario 2):

$$\begin{aligned}\Delta\Phi = \Phi(\mathbf{a}_2) - \Phi(\mathbf{a}_1) = \\ -v_{a_2}(n_{a_2}+1) - \frac{z_i(a_2)}{s_i} + \frac{z_i(a_1)}{s_i} = \\ \frac{u(a_2,\mathbf{a}_{-i})}{s_i} - \frac{u(a_1,\mathbf{a}_{-i})}{s_i} = \frac{1}{s_i}\Delta u_i,\end{aligned} \tag{11}$$

Scenario 3):

$$\begin{aligned}\Delta\Phi = \Phi(\mathbf{a}_2) - \Phi(\mathbf{a}_1) = \\ v_{a_1}(n_{a_1}) - \frac{z_i(a_2)}{s_i} + \frac{z_i(a_1)}{s_i} = \\ \frac{u(a_2,\mathbf{a}_{-i})}{s_i} - \frac{u(a_1,\mathbf{a}_{-i})}{s_i} = \frac{1}{s_i}\Delta u_i,\end{aligned} \tag{12}$$

Scenario 4):

$$\begin{aligned}\Delta\Phi =&\Phi(\mathbf{a}_2) - \Phi(\mathbf{a}_1) = -\frac{z_i(a_2)}{s_i} + \frac{z_i(a_1)}{s_i} = \\ \frac{u(a_2,\mathbf{a}_{-i})}{s_i} - \frac{u(a_1,\mathbf{a}_{-i})}{s_i} =& \frac{1}{s_i}\Delta u_i.\end{aligned} \tag{13}$$

It follows from (10)-(13) that the game from Definition 1 is a weighted potential game with potential function as in (9) and weights $\omega_i = 1/s_i$. This completes the proof. ■

Proving that the problem can be described as a weighted potential game guarantees that systematically improving actions from the agents always makes the system converge to a Nash equilibrium [15]. To put it in other words, this decentralised way of managing the resource allocation problem of offloading control computations is guaranteed to converge to a solution.

VI. Scalability

Since one of the most important aspects of this article is scalability, it is crucial to make sure that the convergence rate scales favorably with the number of devices. By requiring a *Threshold-Based Improvement Rule* from the agents, good convergence behavior is guaranteed. However, not necessarily to an optimal solution, but to a satisfactory solution.

A. Threshold-Based Improvement Rule

In most games, each agent is assumed to play the most rewarding action every time. However, in a noisy environment where network conditions are varying, one typically does not want agents to update strategies every time a small improvement can be obtained. As a consequence of this, a Threshold-Based Improvement Rule is set for the agents in this game, meaning that a device only updates strategy if the relative increase in utility function is greater than a threshold.

Definition 2 (Threshold-Based Improvement Rule): Let $0 < \alpha < 1$ be a given threshold, and consider the game defined in Definition 1. Given a current strategy profile $\mathbf{a} = (a_1, a_2, \ldots, a_n)$, agent i considers switching from its current action a_i to a new action $a_i^* \in A_i$ that maximizes its utility function value, i.e.,

$$a_i^* \in \arg\max_{a \in A_i} u_i\left(a, \mathbf{a}_{-i}\right).$$

The agent updates to a_i^* if and only if

$$u_i\left(a_i^*, \mathbf{a}_{-i}\right) > \alpha u_i\left(a_i, \mathbf{a}_{-i}\right).$$

Otherwise, the agent retains its current action a_i.
Note that α appears on the right side of the inequality since all utilities are negative. The parameter α can therefore be interpreted as the fraction of the current response time the new strategy must achieve.

B. ε-Nash Equilibrium

An approximate equilibrium (or ε-equilibrium) is defined as a strategy profile, where no agent can increase its utility function more than ε by changing strategy. To prove that the offloading system approaches an ε-Nash equilibrium in linear time with respect to the number of agents, the first step is to bound the minimum utility function increment.

Lemma 1 (Minimum Utility Improvement): Consider the game in Definition 1 with the Threshold-Based Improvement Rule from Definition 2. Let Δu_i denote the increase in agent i's utility when it switches from its current action a_i to a new action a_i^*. Then, for every such update,

$$\Delta u_i > (1-\alpha) \min_{j,a} \tau_{j,a}, \tag{14}$$

where $\min_{j,a} \tau_{j,a}$ denotes the shortest RTT (round-trip time) among all possible pairs of agents j and servers a.

Proof: Recall from Definition 2 that a device i updates its action from a_i to a new action a_i^* only if

$$u_i(a_i^*, \mathbf{a}_{-i}) > \alpha u_i(a_i, \mathbf{a}_{-i}).$$

Thus, whenever such an update occurs, the resulting utility increment is

$$\begin{aligned}\Delta u_i = u_i(a_i^*, \mathbf{a}_{-i}) - u_i(a_i, \mathbf{a}_{-i}) > \\ \alpha\, u_i(a_i, \mathbf{a}_{-i}) - u_i(a_i, \mathbf{a}_{-i}) = (\alpha - 1)\, u_i(a_i, \mathbf{a}_{-i}).\end{aligned} \tag{15}$$

Next, since the RTT cannot approach zero, the largest (remember $u_i < 0$) possible value of the utility function is when a device with a fast processor (small v_i^{local}) computes a task of negligible size (small s_i) locally. This utility can in theory be infinitely close to zero. All other utilities for that agent are upper bounded by the negative of the smallest RTT in the entire system, $-\min_{j,a} \tau_{j,a}$. Since a_i^* corresponds to the largest utility, possibly zero, $u_i(a_i, \mathbf{a}_{-i})$ fulfills

$$u_i(a_i, \mathbf{a}_{-i}) < -\min_{j,a} \tau_{j,a}.$$

Inserting this into (15) results in (14). Finally, because $0 < \alpha < 1$ and $\min_{j,a} \tau_{j,a} > 0$, a positive lower bound on the utility increment Δu_i is guaranteed. This completes the proof. ■

C. Linear Convergence

Since the minimum change of the utility function is bounded and our system is a weighted potential game, the minimum step size of the potential function is also lower bounded. This lower bound gives an upper bound on the number of strategy updates, which shows that the convergence rate of the system scales linearly with the number of devices.

Theorem 2 (Bounded Convergence to ε-Nash equilibrium): Consider the offloading game from Definition 1, played by n devices under the Threshold-Based Improvement Rule of Definition 2. Then, starting from the strategy profile $\mathbf{a}^0$ where all agents compute locally, the game converges to an ε-Nash equilibrium in at most k strategy updates, where

$$k \leq n \frac{(\max_i s_i)\left(\max_i v_i^{\text{local}}\right)}{(1-\alpha)\min_{j,a} \tau_{j,a}}.$$

Hence, the worst-case number of strategy updates is bounded linearly by the number of devices n.

Proof: Recall that the game is a weighted potential game with weights $\omega_i = 1/s_i$. This and Lemma 1 allow us to bound the minimum potential function update to

$$\Delta\Phi = \frac{\Delta u_i}{s_i} \geq \frac{\min_i \Delta u_i}{\max_i s_i} > \frac{(1-\alpha)\min_{j,a} \tau_{j,a}}{\max_i s_i}.$$

Since the starting configuration in the theorem is local computing and $\Delta\Phi > 0$ due to the potential game and only improving strategy updates, we have from (7) and (9) that

$$\Phi(\mathbf{a}^0) > -n \max_i (v_i^{local}), \tag{16}$$

where n is the number of agents in the game. Since none of v, s and τ can ever be negative, the potential function is upper bounded by 0. Hence, we can write $\Phi(\mathbf{a}^0) + k\Delta\Phi < 0$. Substituting the bounds above, we get

$$-n \max_i \left(v_i^{\text{local}}\right) + k \frac{(1-\alpha)\min_{j,a} \tau_{j,a}}{\max_i s_i} < 0,$$

which simplifies into

$$k < n \frac{\max_i \left(v_i^{\text{local}}\right) \max_i (s_i)}{(1-\alpha)\min_{j,a} \tau_{j,a}}.$$

Hence, the total number of strategy updates k scales in worst case linearly in n. Once the process terminates, the resulting strategy profile is an ε-Nash equilibrium (by the properties of potential functions). This completes the proof. ■

Theorem 2 proves the linearity from a starting configuration when all devices compute locally. This is a reasonable assumption from a realistic point of view, but it is also easy to realize that other starting configurations also scales linearly in n. Since all devices always have the opportunity to compute locally, the possible strategy updates are at least as good as choosing local computing.

VII. Weakly-Hard Control Requirements

Executing control tasks remotely poses a risk of deadline misses, especially in these decentralized methods where the expected response time used for decision-making might be based on obsolete information. Suppose that every agent evaluates the update rule in Definition 2 at each control cycle. While a control job is executing in an edge server, additional devices might start offloading to the same remote resource, increasing response time for all agents. A device that decided to offload a slight moment earlier based that strategy on obsolete information, and might miss its deadline. To guard against such events and to be robust against packet losses, we refine the update rule in Definition 2 by incorporating fail-safe to local execution, which we assumed has a non-negative slack in Section III. Hence, constraints on packet losses for control algorithms, such as weakly-hard constraints [4], [16], can be tolerated by the decentralized allocation policy.

Definition 3 (Weakly-Hard Constrained Best Response): Let $\mathcal{C}_i$ be the weakly-hard deadline constraint for device i (e.g. (m_i, k_i), meaning that for m_i consecutive deadlines, the agent meets at least k_i of them). At every job release, device i applies the Threshold-Based Improvement Rule of Definition 2 only if $\mathcal{C}_i$ is satisfied. Otherwise, agent i switches to local execution.

Note that Definition 3 assumes that the strategy update is considered at every job release. This requirement can sometimes be relaxed depending on the weakly-hard constraint, such that the strategy update does not have to be considered every time. However, this is not further analysed here. With the Weakly-Hard Constrained Best Response in mind, we can now state the following.

Corollary 1: Consider the Offloading Game in Definition 1, played by n devices. Also assume that whenever the weakly-hard constraint $\mathcal{C}_i$ is violated, the response-time estimate for the next control cycle to the same server exceed the deadline D_i. Let k be the worst-case number of strategy updates under the Threshold-Based Improvement Rule of Definition 2 alone, as bounded in Theorem 2. Then, the worst-case number of strategy updates under the Weakly-Hard Constrained Best Response from Definition 3 is upper bounded by $2k + n$, and still grows linearly with the number of devices.

Proof: Under Definition 3, there are two kinds of strategy updates:

1) Updates according to Definition 2, for which Lemma 1 guarantees a per-update utility improvement of at least $(1-\alpha)\min_{j,a} \tau_{j,a}$, and Theorem 2 a fixed improvement of the potential function.
2) Fail-safe updates (server to local) only occur at a weakly-hard violation, which necessarily follows from a deadline miss while offloaded. Since our utility function can be directly translated as the negative response time of the current strategy, and the assumption on nominal response-time after violated constraints, this implies

$$-u_i(a_i, \mathbf{a}_{-i}) > D_i,$$

TABLE I: Simulation variable values. Some variables were randomized over an uniform distribution, with $\pm$ indicating the range.

Variable	Explanation	Value	$\pm$	Unit
s_i	Computational size of control tasks	1.5	0.5	-
v_c	Time to compute nominal control task in cloud	0.01	0	s
v_i^{local}	Time to compute nominal control task locally	0.1	0.05	s
v_{max}	From $v_j(n_j) = v_{max} n_j$ in edge servers	0.004	0	s
$\tau_{m,j}$	RTT between device and edge server	0.02	0.01	s
$\tau_{m,c}$	RTT between device and cloud server	0.1	0.02	s
α	Req. relative improvement to update strategy	0.95	-	-
$Size(ES_m^{q_e})$	Nbr of edge servers reachable from each cell	5	0	-
$Size(CS_m^{q_c})$	Nbr of cloud servers reachable from each cell	1	0	-

and as local computing is assumed to meet its deadline, $-u_i\left(a_i^{local}, \mathbf{a}_{-i}\right) < D_i$, which results in

$$\Delta u_i = u_i\left(a_i^{local}, \mathbf{a}_{-i}\right) - u_i\left(a_i, \mathbf{a}_{-i}\right) > 0.$$

The utility, and hence also the potential function, is strictly improving for fail-safe strategy updates.

The key here is that a fail-safe can only occur when currently on a server, and to perform another fail-safe later it must re-offload via a move of (1). Thus, the number of fail-safe moves is bounded by the number of moves by Definition 2, plus at most n updates if all agents start with offloading. Consequently, the total number of updates is at most $2k + n$, which preserves a linear bound on the total number of strategy updates. This concludes the proof. ■

So with the stated assumptions, the Weakly-Hard Constrained Best Response in Definition 3 preserves the linear convergence rate with respect to the number of agents, while simultaneously enforcing the weakly-hard constraints in consecutive deadline misses. This discussion also highlights the need for common analysis of control algorithms and resource allocation in the context of offloading, of which some relevant works on this topic from a control systems point of view can be found in works by [17], and [18].

VIII. Simulations

To verify and demonstrate the scalability of the decentralised allocation policy, the system was built and simulated. The simulation code used is found at [19]. The function describing nominal response time on edge servers, $v(n_a)$, was assumed to be $v(n_a) = v_{max} n_a$, meaning that all agents offloading there share the resources equally. Note that the theory is not constrained to this assumption on $v(n_a)$. All numerical parameters are presented in Table I, randomized on a uniform distribution with interval in the $\pm$-column. All simulation setups had in total two cloud servers.

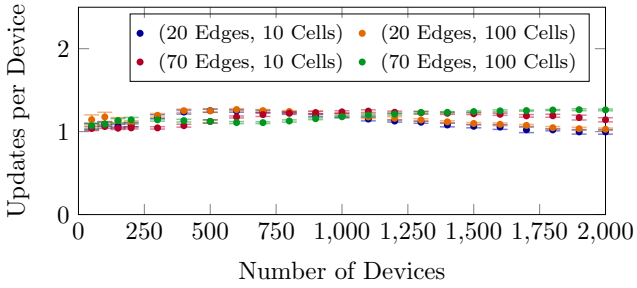


Fig. 3: Number of strategy updates per device to converge as the system scales, with standard deviations (error bars), with local computing as starting configuration.

Figure 3 shows the scalability of the proposed policy. Even though the number of agents increase, the average number of strategy updates each device needs to do remains similar.

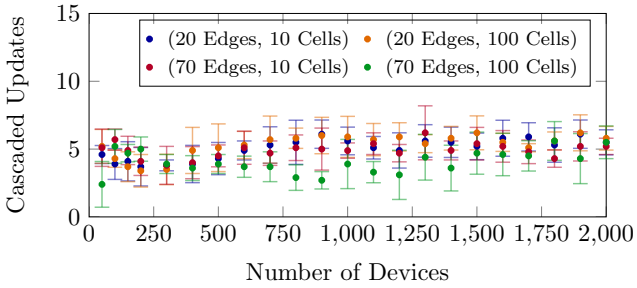


Fig. 4: Total number of cascaded strategy updates, with standard deviations (error bars), as five edge server offloading agents is removed.

The cascaded update responses, defined in Section I, is shown in Figure 4, where from a Nash Equilibrium, five edge server-offloading devices are removed, and the game is played to convergence. As one can see, the cascading response varies from simulation to simulation, but does not systematically increase as the system scales.

IX. Concluding Remarks

Motivated by network model simplifications from pre-granted RB scheduling in 5G URLLC, a simple game-based decentralised resource allocation policy for control computations is proposed. The policy scales linearly in the total number of updates with the number of agents, meaning that for each device, the expected number of strategy updates scales with $\mathcal{O}(1)$ as the number of agents increase. This is beneficial compared to central planners, like approximations of ILPs where the central planner's time complexity can grow cubic in number of devices [7], or auction-based approaches where the time complexity can often be $\mathcal{O}(n^2) - \mathcal{O}(n^4)$ depending on how the number of bids are defined [8]. Another important part, more than scalability and simplicity, toward real implementation is that control constraints such as weakly-hard constraints can be achieved with minor assumptions without interfering with the linear convergence.

ACKNOWLEDGMENT

The authors gratefully acknowledge reviewers' comments. We especially thank David Ohlin, Max Nyberg Carlsson, Ahmed Al Bayati, Yde Sinnema and Karl-Erik Årzén for valuable feedback on an earlier version of this manuscript.

REFERENCES

[1] K. Kumar, J. Liu, Y.-H. Lu, and B. Bhargava, "A survey of computation offloading for mobile systems," *Mobile Networks and Applications*, vol. 18, pp. 129–140, 2012.

[2] B. Kehoe, S. Patil, P. Abbeel, and K. Goldberg, "A survey of research on cloud robotics and automation," *IEEE Transactions on automation science and engineering*, vol. 12, no. 2, pp. 398–409, 2015.

[3] S. Dong, J. Tang, K. Abbas, R. Hou, J. Kamruzzaman, L. Rutkowski, and R. Buyya, "Task offloading strategies for mobile edge computing: A survey," *Computer Networks*, vol. 254, p. 110791, 2024.

[4] M. Maggio, A. Hamann, E. Mayer-John, and D. Ziegenbein, "Control-system stability under consecutive deadline misses constraints," in *32nd euromicro conference on real-time systems (ECRTS 2020)*, pp. 21–1, Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2020.

[5] M.-H. Chen, B. Liang, and M. Dong, "Multi-user multi-task offloading and resource allocation in mobile cloud systems," *IEEE Transactions on Wireless Communications*, vol. 17, no. 10, pp. 6790–6805, 2018.

[6] S. Jošilo and G. Dán, "Joint wireless and edge computing resource management with dynamic network slice selection," *IEEE/ACM Transactions on Networking*, vol. 30, no. 4, pp. 1865–1878, 2022.

[7] C. Gao and A. Easwaran, "Energy-efficient joint offloading and resource allocation for deadline-constrained tasks in multi-access edge computing," in *2025 IEEE 31st International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pp. 55–67, IEEE, 2025.

[8] X. Zheng, S. B. H. Shah, S. Usman, S. Mahfoudh, F. Shemim KS, and P. Kumar Shukla, "Resource allocation and network pricing based on double auction in mobile edge computing," *Journal of Cloud Computing*, vol. 12, no. 1, p. 56, 2023.

[9] M.-A. Messous, H. Sedjelmaci, N. Houari, and S.-M. Senouci, "Computation offloading game for an uav network in mobile edge computing," in *2017 IEEE International Conference on Communications (ICC)*, pp. 1–6, IEEE, 2017.

[10] Y. Ge, Y. Zhang, Q. Qiu, and Y.-H. Lu, "A game theoretic resource allocation for overall energy minimization in mobile cloud computing system," in *Proceedings of the 2012 ACM/IEEE International Symposium on Low Power Electronics and Design*, pp. 279–284, 2012.

[11] X. Chen, "Decentralized computation offloading game for mobile cloud computing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 26, no. 4, pp. 974–983, 2014.

[12] T. Dudda and A. Shapin, "Time-critical communication in 5g nr." Ericsson Blog, Feb. 2021. https://www.ericsson.com/en/blog/2021/2/time-\\critical-communication--5g-nr.

[13] E. Dahlman, S. Parkvall, and J. Skold, *5G/5G-advanced: the new generation wireless access technology*. Elsevier, 2023.

[14] R. W. Rosenthal, "A class of games possessing pure-strategy nash equilibria," *International Journal of Game Theory*, vol. 2, pp. 65–67, 1973.

[15] I. Milchtaich, "Congestion games with player-specific payoff functions," *Games and economic behavior*, vol. 13, no. 1, pp. 111–124, 1996.

[16] G. Bernat, A. Burns, and A. Liamosi, "Weakly hard real-time systems," *IEEE transactions on Computers*, vol. 50, no. 4, pp. 308–321, 2001.

[17] K.-E. Årzén, A. Cervin, J. Eker, and L. Sha, "An introduction to control and scheduling co-design," in *Proceedings of the 39th IEEE Conference on Decision and Control (Cat. No. 00CH37187)*, vol. 5, pp. 4865–4870, IEEE, 2000.

[18] D. Seto, B. Krogh, L. Sha, and A. Chutinan, "The simplex architecture for safe online control system upgrades," in *Proceedings of the 1998 American Control Conference. ACC (IEEE Cat. No. 98CH36207)*, vol. 6, pp. 3504–3508, IEEE, 1998.

[19] Sundström, E., "game-offload-article." https://github.com/emilcontrol/\\game-offload-article. Commit: 5a4572c, 2025.

Soft-Float-Aware One-Class SVM Inference on Edge Routers

Barikisu A. Asulba[1,3], **Pedro F. Souto**[1,3], **Luis Almeida**[2,3]

[1] CISTER Research Centre, Porto, Portugal
[2] Instituto de Telecomunicações, Porto, Portugal
[3] Faculdade de Engenharia, Universidade do Porto, Portugal

Email: {up202103270, pfs, lda}@fe.up.pt

***Abstract*—Network monitoring applications based on machine-learning (ML) are increasingly deployed on routers for real-time traffic analysis. However, ML inference on commodity routers is constrained by floating-point operations, which are especially expensive on devices without a hardware floating-point unit (FPU) because they are emulated through soft-float routines.**

We study radial-basis-function (RBF) One-Class SVM (OC-SVM) inference in this setting given its good detection performance and because it can be trained on benign data alone. Without an FPU, the computation of the RBF decision function is expensive and its use may lead to packet loss in the case of a high packet rate. To address this problem, we reduce the cost of the RBF OC-SVM by: 1) tuning the hyper-parameter ν, to reduce the number of support vectors in the model and consequently the number of arithmetic operations; 2) using quantization, to replace floating-point arithmetic operations with integer operations, and 3) using FastExp, an IEEE-754-based approximation of the exponential function.

On a Netgear R7000 router running OpenWRT, our approach reduces the mean model inference time to 0.080 ms, a 10.29 times improvement, over the inference time of the standard floating-point RBF OC-SVM baseline implementation. Under live replay of an attack-level load of approximately $4{,}747$ packets per second in an IoT network, our implementation eliminates packet loss against 74% loss for the baseline. These results show that a soft-float-aware implementation can substantially reduce RBF OC-SVM inference time and enable its use for router-level edge traffic monitoring.

***Index Terms*—Edge computing, router-level inference, One-Class SVM, fixed-point quantization, soft-float, anomaly detection, real-time systems.**

I. Introduction

Edge computing reduces latency by processing data close to where it is generated, rather than in a remote cloud [1]. This also applies to the detection of anomalous traffic, particularly attack-generated, that benefits from running anomaly-detection tasks in edge routers directly on live traffic streams [2], [3]. However, detecting anomalies in real-time on constrained router hardware poses a challenge: if inference rate is lower than packet arrival rate, buffers and queues fill up, increasing delay and eventually causing packet loss, compromising anomaly detection.

Recent studies on edge ML-based monitoring and real-time ML inference [4]–[6] assume hardware floating-point units (FPU) or target GPU- or hardware-accelerated neural-network inference. This does not apply to edge routers without FPU that emulate floating-point operations in software.

This paper studies inference with One-Class Support Vector Machine (OC-SVM), focusing on a Radial Base Function (RBF) kernel. OC-SVM is a lightweight model commonly used for anomaly detection because it can be trained using benign data only [7]–[11]. In terms of security, this feature enables detection of 0-day attacks. Nevertheless, the complexity of the exact RBF decision function scales with the number of support vectors and requires repeated floating-point operations, making it expensive on soft-float router hardware.

Existing SVM-acceleration techniques re-implement the kernel summation on designer-controlled FPGA and micro-controller datapaths [12], [13]. A linear kernel avoids the per-support-vector cost but cannot capture the nonlinear structure of the traffic; in our tests a linear kernel OC-SVM did not match the detection behavior of the RBF kernel, consistent with the known advantage of the RBF kernel for one-class problems [14]. The RBF kernel can instead be approximated by an explicit feature map — random Fourier features [15] or the Nyström method [16] — trained as a linear model in the approximate space, but this yields an approximate model rather than the exact RBF OC-SVM. Conversely, we retain the exact decision function, but reduce the cost of evaluating it. SunBlock [17] runs an OC-SVM detector on an OpenWRT router, but reports detection effectiveness and resource usage rather than the on-device inference cost or real-time inference behavior of the decision function.

We use the `dlib` [18] RBF OC-SVM as the floating-point baseline, which is based on LIBSVM [19]. For deployment on routers that lack FPU support, we address the inference bottleneck through the following contributions:

- Using support-vector count as deployment-aware model-selection criterion for RBF OC-SVM on soft-float routers;
- Implementing the RBF OC-SVM decision function using two approximation strategies: *partial quantization*, in which the distance computation is quantized while the exponential is approximated using FastExp [20]; and *full quantization*, in which all model parameters are additionally quantized.

We assess the improved scorer on a Netgear R7000 running OpenWRT under live capture, showing a reduction of up to 10.29 times in mean model inference time over the `dlib` RBF OC-SVM baseline.

II. Problem Statement

We consider a router-level anomaly detection task that scores the feature vector $x \in \mathbb{R}^d$ of each packet as the packets arrive. Let R_{in} be the packet arrival rate and R_{svm} be the sustained scoring rate of the OC-SVM inference engine. For online operation to remain stable, the scorer must keep up with the incoming stream:

$$R_{\text{svm}} \geq R_{\text{in}}. \tag{1}$$

Equivalently, if the per-packet time budget is $1/R_{\text{in}}$, the processing path must satisfy Eq. 2 where T_{feat} is the feature preparation time, T_{svm} is the OC-SVM inference time, and T_{queue} is the capture and queueing overhead.

$$T_{\text{queue}} + T_{\text{feat}} + T_{\text{svm}} \leq \frac{1}{R_{\text{in}}}, \tag{2}$$

If this condition is violated over time, packets accumulate in buffers and may eventually be dropped. This paper targets the inference term, so we isolate the model-side requirement:

$$T_{\text{svm}} \leq \frac{1}{R_{\text{in}}} - T_{\text{queue}} - T_{\text{feat}}. \tag{3}$$

The arrival rate R_{in} and the queueing overhead T_{queue} depend on the deployment and router load, so we do not assume fixed values for them. In our setting, feature preparation is lightweight, making T_{feat} small relative to OC-SVM inference. Reducing T_{svm} alone therefore does not guarantee that Equation (3) holds in all cases: if R_{in} is too high, the router may still fail to keep up. Our goal is to reduce the controllable model inference time T_{svm}, making the timing condition easier to satisfy for a wider range of deployments.

A. OC-SVM inference time

The standard RBF OC-SVM decision function, in Eq. (4):

$$\begin{aligned} f(x) &= \operatorname{sgn}\left(\sum_{i=1}^{N_{\text{sv}}} \alpha_i \, k(s_i, x) - \rho\right), \\ k(s_i, x) &= \exp(-\gamma \|s_i - x\|^2). \end{aligned} \tag{4}$$

where, $x \in \mathbb{R}^d$ is the input feature vector, d is the feature dimension, $s_i \in \mathbb{R}^d$ is the ith support vector, N_{sv} is the number of support vectors, α_i is the corresponding dual coefficient, ρ is the learned bias, and γ is the RBF kernel parameter.

For each input x, the decision function evaluates a sum of N_{sv} terms followed by one subtraction of the bias. Each term is the product of a constant coefficient, α_i, and a kernel value, $k(s_i, x)$, so the sum contributes N_{sv} multiplications and N_{sv} additions. Each kernel value in turn requires the computation of a squared Euclidean distance, $\|s_i - x\|^2$, in d dimensions – that is, d subtractions and d multiplications – followed by one multiplication of that squared distance by a constant, γ, and the computation of one exponential. In total, evaluating the decision function requires $O(N_{\text{sv}} \cdot d)$ multiplications and additions/subtractions, plus N_{sv} exponentials, the latter being individually far more costly than a single multiplication. The model inference time can therefore be approximated by Eq. (5):

$$T_{\text{svm}} \;\approx\; N_{\text{sv}} \left(T_{\text{dist}}(d) + T_{\text{exp}} + T_{\text{acc}}\right). \tag{5}$$

where $T_{\text{dist}}(d)$ is the time of computing the squared distance in d dimensions, T_{exp} is the time of evaluating the exponential term, and T_{acc} is the time of coefficient multiplication and accumulation.

Therefore, to reduce T_{svm} we target the computation of three terms using integer arithmetic, namely $T_{\text{dist}}(d)$ and T_{acc}, and the computation of T_{exp} using a fast approximation. Furthermore, we seek to choose an N_{sv} value that strikes a good balance between quality of the inference and the inference time.

TABLE I
ν-SENSITIVITY ANALYSIS FOR RBF OC-SVM SUPPORT-VECTOR COUNT, DETECTION PERFORMANCE, AND MEAN PER-SAMPLE EVALUATION TIME, MEASURED ON A DESKTOP DURING MODEL SELECTION.

ν	N_{sv}	DR	Avg inf_time (ms/sample)
10^{-5}	55	0.987	0.595
5×10^{-5}	254	0.988	2.723
2×10^{-4}	1,003	0.988	10.605

III. Fast RBF OC-SVM Inference

A. ν selection for reduced support-vector count

In OC-SVM, the hyperparameter ν affects both the learned boundary and the number of support vectors, N_{sv} retained by the trained model [7]. Therefore, we select ν so as to obtain a small N_{sv} while maintaining acceptable detection performance. Table I reports the sensitivity analysis performed on a x86-64 FPU-equipped desktop platform. Increasing ν from 10^{-5} to 2×10^{-4} raises the support-vector count from 55 to 1,003, slightly increases DR from 0.987 to 0.988, and increases the mean per-sample evaluation time from 0.595 ms to 10.605 ms, an approximately $17.82\times$ increase.

B. Quantization for RBF decision-function evaluation

Partial and Full Quantization share the same affine mapping for inputs and support vectors. Support vectors are quantized once after training, whereas each incoming feature vector is quantized at runtime according to Eq. 6:

$$q_x = M \cdot \frac{x - x_{\min}}{x_{\max} - x_{\min}} \;=\; a\,x + c, \tag{6}$$

where, $x_{\min}$ and $x_{\max}$ are the per-feature minimum and maximum values estimated from the training data, a and c are precomputed per-feature constants, and $M = 2^{10} = 1024$ is the quantization scale, corresponding to a left shift of 10 bits.

Given a quantized input q_x and support vector q_{s_i}, the squared distance is approximated as

$$\|s_i - x\|^2 \approx \frac{1}{M^2} \sum_{j=1}^{d} \left(q_{x,j} - q_{s_i,j}\right)^2. \tag{7}$$

Partial Quantization computes the distance sum using integer arithmetic, with $1/M^2$ folded into γ. Full Quantization further converts the coefficients, α_i, the bias, and the kernel value to integers.

C. FastExp for the RBF exponential

FastExp replaces the standard library call used to compute the RBF exponential, $\exp(-\gamma\|s_i - x\|^2)$, during each kernel evaluation. We use a bounded approximation based on Schraudolph [20] and its implementation in [21], which exploits the IEEE-754 single-precision layout to approximate $\exp(\cdot)$ with lower computation time. This approximation transforms the computation of an exponential into a single multiply–add followed by the cast to a `float` of the result reinterpreted as an integer. This trades numerical exactness for speed, but does not change the $O(N_{\text{sv}} \cdot d)$ complexity of inference; it reduces the constant cost of the exponential term in each kernel evaluation.

IV. Evaluation

We evaluate four OC-SVM decision-function implementations on live router traffic: the `dlib` RBF baseline, Partial Quantization, Partial Quantization + FastExp, and Full Quantization + FastExp. The model was trained offline on an FPU-equipped platform using only benign traffic from the Edge-IIoTset dataset [22]. We select $\nu = 5 \times 10^{-5}$ because it provides a good balance between inference time and decision performance. We evaluate on a benign trace and traces covering different attacks (e.g., DDoS, malware, and scanning), each replayed to the router with original inter-packet timing and captured on the bridge LAN interface.

A. Deployment target

The target platform is a Netgear R7000 router running OpenWRT 23.05.0. The router uses a Broadcom BCM5301X-family ARMv7-A processor. Hardware floating-point support is optional in this architecture: the ARMv7-A and ARMv7-R Architecture Reference Manual describes VFPv3 and VFPv4 as optional extensions to the ARMv7-A and ARMv7-R profiles [23]. In our OpenWRT deployment, the OC-SVM scorer executes through a soft-float path: `/proc/cpuinfo` reports no FPU support, and the OpenWRT SDK for the R7000 target produces ARM soft-float EABI binaries, as confirmed by `readelf`. Hardware packet-processing acceleration, if present, does not accelerate the user-space floating-point arithmetic used by the scorer. Broadcom's publicly available documentation on the BCM310X processor is silent about this.

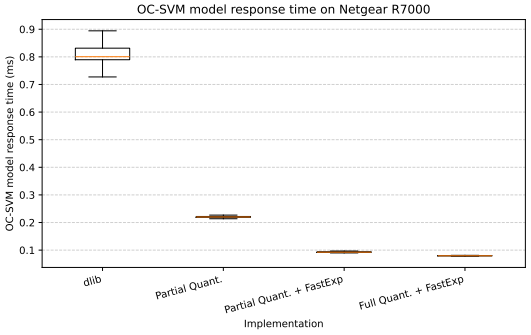


Fig. 1. Distribution of OC-SVM model inference time on the Netgear R7000 during live-router execution.

TABLE II
OC-SVM model inference time on the Netgear R7000 during live-router execution.

Implementation	Mean (ms)	Median (ms)
`dlib` RBF OC-SVM	0.820	0.801
Partial Quantization	0.220	0.220
Partial Quantization + FastExp	0.093	0.093
Full Quantization + FastExp	0.080	0.080

B. Model inference time

We measure *model inference time* using:

$$R_i^{model} = t_i^{score\_out} - t_i^{score\_in}.$$

where $t_i^{score\_in}$ and $t_i^{score\_out}$ are the values returned by calls to `std::chrono::steady_clock` placed immediately before and after, respectively, the call to OC-SVM decision function.

Fig. 1 shows the model inference times considering the `dlib` baseline, Partial Quantization, with and without FastExp, and Full Quantization with FastExp. The quantized implementations not only reduce the mean inference time but also its variability. Table II reports mean inference times, showing that Partial Quantization reduces from 0.820 ms to 0.220 ms. Adding FastExp reduces it to 0.093 ms, an $8.81\times$ improvement over the `dlib` baseline. Full Quantization + FastExp further reduces the mean inference time to 0.080 ms, a $10.29\times$ improvement over the baseline.

C. Sustainable throughput and packet loss

Table III reports the inference-bound rate derived from model inference time. The rate rises from $\approx 1{,}220$ packets per second (pps) for the `dlib` baseline to $\approx 10{,}742$ pps with Partial Quantization + FastExp and $\approx 12{,}546$ pps with Full Quantization + FastExp. The capture-and-extraction path alone sustains about $62{,}000$ pps, so the scorer (OCSVM decision function) is the limiting stage. Reducing T_{svm} therefore raises the arrival rate the monitoring path can sustain and widens the range of R_{in} for which the timing condition in Eq. 3 holds. The DDoS TCP live-replay generates approximately $4{,}747$ pps, i.e. the mean inter-arrival budget is 0.211 ms per packet. The

TABLE III
INFERENCE-BOUND PROCESSING RATE ON THE NETGEAR R7000, DERIVED FROM THE PER-PACKET INFERENCE TIME.

Configuration	Derived rate (pps)
`dlib` RBF OC-SVM	1,219
Partial Quantization	4,536
Partial Quantization + FastExp	10,742
Full Quantization + FastExp	12,546

TABLE IV
MEASURED ROUTER PACKET LOSS UNDER DDOS TCP LIVE REPLAY ($\approx$ 4,747 PPS).

Configuration	Load (pps)	Packet loss (%)
`dlib` RBF OC-SVM	$\approx$ 4,747	74.1
Partial Quantization + FastExp	$\approx$ 4,747	**0.0**

`dlib` baseline mean inference time (0.820 ms) exceeds this budget, whereas Partial Quantization + FastExp (0.093 ms) and Full Quantization + FastExp (0.080 ms) remain below it. This provides an empirical timing envelope for the tested workload.

To validate this envelope, we measured packet loss at the capture-to-scorer boundary during DDoS TCP live replay (Table IV). At $\approx$ 4,747 pps, the `dlib` pipeline experiences 74.1% packet loss, whereas Partial Quantization + FastExp completes the same replay with no measured packet loss. With normal IoT loads (1,000–1,300 pps), the `dlib` pipeline drops less than 2% of packets.

D. Decision fidelity and detection performance

In this section we evaluate how quantization and the approximation of the exponential function affect detection performance. We seek to evaluate mainly two aspects. First, how do the different techniques used to speed up inference time affect the output of the OC-SVM decision function. Second, what is the detection performance of the different implementations.

This evaluation was carried out offline on a PC, not during live execution on the router. For each dataset, the feature extractor read the PCAP file with the captured packets and passed the extracted feature vectors directly to each implementation of the OC-SVM decision function.

To evaluate decision fidelity we use as metric the decision mismatch ratio. That is, for each implementation, we compute the percentage of samples for which its output differs from that of the baseline implementation, i.e., when the sign of the decision score is different.

Table V shows that Partial Quantization preserves the reference decision for almost all samples (0.023% average mismatch). Adding FastExp increases the average mismatch to 0.417% for Partial Quantization and 0.396% for Full Quantization + FastExp. For all attack data sets, the mismatch ratio is below 1%, except for Vulnerability (to Web attacks) which has a mismatch ratio below 3%. We believe that even this higher value is a good tradeoff for the observed reduction in packet loss..

TABLE V
DECISION MISMATCH RATIO (%) RELATIVE TO THE FLOATING-POINT DLIB REFERENCE. PQ DENOTES PARTIAL QUANTIZATION, AND FQ DENOTES FULL QUANTIZATION.

Dataset	PQ	PQ+FastExp	FQ+FastExp
Backdoor	0.016	0.042	0.042
DDoS HTTP	0.063	0.388	0.343
DDoS TCP	0.001	0.006	0.005
OS Fingerprinting	0.005	0.009	0.008
Ransomware	0.000	0.004	0.005
SQL Injection	0.044	0.314	0.307
File Uploading	0.020	0.065	0.065
Vulnerability	0.055	2.919	2.780
XSS	0.001	0.005	0.005
Average	**0.023**	**0.417**	**0.396**

The mismatch ratio measures the quality of the approximation of each quantized implementation with respect to the reference implementation. It does not compare the output of each decision function implementation with the ground truth. A mismatch need not mean that the quantized model is wrong; it may be the case that it corrects an "error" of the reference implementation. Thus, the mismatch ratio is an upper-bound on the error introduced by a quantized model with respect to the reference implementation. To evaluate the detection performance, i.e. to evaluate the different implementations with respect to the ground truth, we consider two metrics: the detection rate (or true-positive rate) and the false-positive rate.

Table VI shows the values of these metrics for the different implementations, for the DDoS-TCP attack data set that we have used in our inference-time experiments. For all models the detection rate is higher than 98.7% and the false-positive rate is below 0.3%. Based on the decision fidelity values reported in Table V and the results we obtained in previous work [24] using a different methodology, we expect the results for the other attacks to still be competitive with respect to other ML algorithms, even if worse than those reported for the DDoS-TCP attack data set.

V. CONCLUSION

This paper studied the execution of OC-SVM-based anomaly detection in edge routers, particularly to detect attack-related traffic in IoT networks. We considered OC-SVM with the RBF kernel for its superior detection performance and found the inference step to be a bottleneck on soft-float edge routers, i.e., routers that lack hardware FPU. We then targeted the reduction of the dominant costs in the decision path using quantization and FastExp, a fast implementation of the exponential function. With Partial Quantization + FastExp we could reduce the mean model inference time from 0.820 ms to 0.093 ms on a Netgear R7000 OpenWrt router deployment, an 8.81$\times$ improvement over the `dlib` baseline that relies on software floating-point emulation. Full Quantization +

TABLE VI
PACKET-LEVEL DETECTION RATE AND FALSE-POSITIVE RATE (DR/FPR) FOR THE FLOATING-POINT `dlib` REFERENCE AND QUANTIZED IMPLEMENTATIONS.

Dataset	`dlib`	Partial Quant.	Partial Quant. + FastExp	Full Quant. + FastExp
DDoS TCP	0.9875/0.0029	0.9875/0.0029	0.9874/0.0029	0.9874/0.0029

FastExp further reduced the mean to 0.080 ms, a $10.29\times$ improvement over the baseline. From the same measurements, the inference-bound processing rate increased from $1{,}220$ pps for the baseline to $10{,}742$ pps with Partial Quantization + FastExp and $12{,}546$ pps with Full Quantization + FastExp. In live replay at an attack-level load of approximately $4{,}747$ pps, this reduction eliminated packet loss caused by the scorer throughput against 74% loss with the `dlib` baseline.

These results show that a soft-float-aware implementation can substantially reduce the OC-SVM inference bottleneck on router hardware, making it a viable solution in practical edge IoT networks. We are currently extending this work to the estimation of end-to-end timing bounds and a more complete evaluation of the detection performance.

REFERENCES

[1] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu, "Edge computing: Vision and challenges," *IEEE Internet of Things Journal*, vol. 3, no. 5, pp. 637–646, 2016, doi:10.1109/JIOT.2016.2579198.

[2] M. Eskandari, Z. H. Janjua, M. Vecchio, and F. Antonelli, "Passban IDS: An intelligent anomaly-based intrusion detection system for IoT edge devices," *IEEE Internet of Things Journal*, vol. 7, no. 8, pp. 6882–6897, 2020, doi:10.1109/JIOT.2020.2970501.

[3] X.-H. Nguyen, X.-D. Nguyen, H.-H. Huynh, and K.-H. Le, "Realguard: A lightweight network intrusion detection system for IoT gateways," *Sensors*, vol. 22, no. 2, p. 432, 2022, doi:10.3390/s22020432.

[4] S. Lee, W. Kang, M. Bertogna, H. S. Chwa, and J. Lee, "Timing guarantees for inference of AI models in embedded systems," *Real-Time Systems*, vol. 61, no. 2, pp. 259–267, 2025, doi:10.1007/s11241-025-09445-9.

[5] S. Kang, S. Lee, H. Koo, H. S. Chwa, and J. Lee, "RT-MDM: Real-time scheduling framework for multi-DNN on MCU using external memory," in *Proc. 61st ACM/IEEE Design Automation Conference (DAC)*, 2024, doi:10.1145/3649329.3655681.

[6] T. Abdelzaher, Y. Hu, D. Kara, T. Kimura, A. Misra, V. Ramani, O. Tardieu, T. Wang, M. Wigness, and A. Youssef, "The bottlenecks of AI: Challenges for embedded and real-time research in a data-centric age," *Real-Time Systems*, vol. 61, no. 2, pp. 185–236, 2025, doi:10.1007/s11241-025-09452-w.

[7] B. Schölkopf, J. C. Platt, J. Shawe-Taylor, A. J. Smola, and R. C. Williamson, "Estimating the support of a high-dimensional distribution," *Neural Computation*, vol. 13, no. 7, pp. 1443–1471, 2001, doi:10.1162/089976601750264965.

[8] D. M. J. Tax and R. P. W. Duin, "Support vector data description," *Machine Learning*, vol. 54, no. 1, pp. 45–66, 2004, doi:10.1023/B:MACH.0000008084.60811.49.

[9] E. Eskin, A. Arnold, M. Prerau, L. Portnoy, and S. Stolfo, "A geometric framework for unsupervised anomaly detection: Detecting intrusions in unlabeled data," in *Applications of Data Mining in Computer Security*. Kluwer/Springer, 2002, pp. 77–101, doi:10.1007/978-1-4615-0953-0_4.

[10] M. Ahmed, A. N. Mahmood, and J. Hu, "A survey of network anomaly detection techniques," *Journal of Network and Computer Applications*, vol. 60, pp. 19–31, 2016, doi:10.1016/j.jnca.2015.11.016.

[11] H. Hindy, D. Brosset, E. Bayne, A. K. Seeam, C. Tachtatzis, R. Atkinson, and X. Bellekens, "A taxonomy of network threats and the effect of current datasets on intrusion detection systems," *IEEE Access*, vol. 8, pp. 104 650–104 675, 2020, doi:10.1109/ACCESS.2020.3000179.

[12] C. J. C. Burges, "Simplified support vector decision rules," in *Proc. 13th International Conference on Machine Learning (ICML)*, 1996, pp. 71–77, author copy.

[13] D. Anguita, A. Ghio, S. Pischiutta, and S. Ridella, "A hardware-friendly support vector machine for embedded automotive applications," in *Proc. International Joint Conference on Neural Networks (IJCNN)*, 2007, pp. 1360–1364, doi:10.1109/IJCNN.2007.4371159.

[14] A. Bounsiar and M. G. Madden, "Kernels for one-class support vector machines," in *2014 International Conference on Information Science and Applications (ICISA)*. IEEE, 2014, pp. 1–4, doi:10.1109/ICISA.2014.6847419.

[15] A. Rahimi and B. Recht, "Random features for large-scale kernel machines," in *Advances in Neural Information Processing Systems (NeurIPS) 20*, 2007, pp. 1177–1184.

[16] C. K. I. Williams and M. Seeger, "Using the Nyström method to speed up kernel machines," in *Advances in Neural Information Processing Systems (NeurIPS) 13*, 2000, pp. 682–688.

[17] V. Safronov, A. M. Mandalari, D. J. Dubois, D. Choffnes, and H. Haddadi, "SunBlock: Cloudless protection for IoT systems," in *Passive and Active Measurement (PAM)*, ser. LNCS, vol. 14538. Springer, 2024, pp. 322–338, doi:10.1007/978-3-031-56252-5_15.

[18] D. E. King, "Dlib-ml: A machine learning toolkit," *Journal of Machine Learning Research*, vol. 10, pp. 1755–1758, 2009.

[19] C.-C. Chang and C.-J. Lin, "LIBSVM: A library for support vector machines," *ACM Transactions on Intelligent Systems and Technology*, vol. 2, no. 3, pp. 27:1–27:27, 2011.

[20] N. N. Schraudolph, "A fast, compact approximation of the exponential function," *Neural Computation*, vol. 11, no. 4, pp. 853–862, 1999, doi:10.1162/089976699300016467.

[21] J. Rade, "fastExp: Approximation of exp(x) (mit license)," https://gist.github.com/jrade/293a73f89dfef51da6522428c857802d, 2021, source code; distributed under the MIT license.

[22] M. A. Ferrag, O. Friha, D. Hamouda, L. Maglaras, and H. Janicke, "Edge-IIoTset: A new comprehensive realistic cyber security dataset of IoT and IIoT applications for centralized and federated learning," *IEEE Access*, vol. 10, pp. 40 281–40 306, 2022.

[23] *ARM Architecture Reference Manual: ARMv7-A and ARMv7-R Edition*, https://developer.arm.com/documentation/ddi0406/cd/, Arm Limited, 2018, accessed: 2026-05-22.

[24] B. A. Asulba, P. F. Souto, and L. Almeida, "Flow-based vs packet-level intrusion detection for iot networks: A comparative resource and performance analysis," in *2025 IEEE 30th International Conference on Emerging Technologies and Factory Automation (ETFA)*, 2025, pp. 1–4.

Cost-Optimal Cloud Capacity Provisioning with Soft Deadlines

Sathish Gopalakrishnan and Mohammad Shahrad
Department of Electrical and Computer Engineering
The University of British Columbia

***Abstract*—Cloud customers face a standing decision: pay the serverless (FaaS) premium for elastic, commitment-free capacity, or reserve virtual machines at a discount and bear the risk of paying for idle capacity. Recent work shows the boundary between the two can be crossed in both directions: idle VM capacity can absorb serverless work, while serverless can act as a safety valve when demand exceeds VM capacity. We ask whether one can jointly decide how many VMs to reserve and how to schedule heterogeneous jobs with deadlines across heterogeneous instance types, with provable guarantees. We model the joint provisioning-and-scheduling problem as an infinite-horizon average-cost Markov decision process (MDP) in which jobs of different types arrive as Poisson streams, carry soft deadlines paid for via weighted tardiness, and each instance type fits each job type to a different degree. The exact MDP is intractable, but a price-equalizing relaxation lower-bounds it by a sum of independent single-job dynamic programs whose minimizers double as an instance-assignment policy and as the per-type workload they induce. Provisioning each instance type by a square-root newsvendor rule on top of this assignment gives a policy whose excess cost over the lower bound is a sum of per-type newsvendor mismatch costs. Under load scaling by a factor θ, this excess grows as $\Theta(\sqrt{\theta})$ while the optimum grows as $\Theta(\theta)$, so the policy is asymptotically optimal with an $O(1/\sqrt{\theta})$ relative gap. On a pricing model spanning AWS EC2 instance families the realized gap is under $\sim 5\%$ and shrinks with scale. We connect the result to AWS Lambda Managed Instances, which productizes this FaaS–reserved boundary, and read off the model a quantitative rule for when a customer should FaaSen and when they should Un-FaaSen.**

I. Introduction

A cloud customer choosing between serverless (FaaS) and reserved virtual machines is using the same amount of compute under two pricing contracts. FaaS bills per invocation and asks for no commitment. In contrast, virtual machines (VMs) require a multi-period commitment and can sit idle, but cost as little as half as much [1]. On Amazon Web Services (AWS), for example, one has to choose from about 500 instance types across compute-, memory-, and accelerator-optimized families. Many workloads of interest have (soft) deadlines (inference serving, telemetry pipelines, event-driven analytics), so the placement decision has to meet timing requirements while managing cost.

The clear strategy is to reserve cheap capacity for the steady part of the workload and pay the serverless premium only on the spikes [2], [3]. Libra [4] proposed the notion of *cost indifference point*, a traffic volume at which serverless and VM are equally costly. Alternatively, reserved VMs that have already been paid for can absorb additional work at near-zero marginal cost. UnFaaSener [5] realizes this approach: it offloads functions off FaaS platforms onto non-serverless resources (idle VMs, on-premises servers, even personal machines), and reports savings up to 89.8% at a 90% offload cap. But UnFaaSener is *opportunistic*: it uses whatever spare capacity happens to exist. It never decides how much VM capacity to reserve up front, and offers no guarantee on the resulting cost relative to the best achievable.

We ask the planning-side counterpart to UnFaaSener:

> *Can we jointly decide how many VMs of each instance type to reserve and how to schedule heterogeneous jobs with deadlines across those instance types, so as to minimize long-run cost with a provable optimality guarantee?*

How many VMs to reserve depends on where jobs will be assigned; where to assign a job depends on what capacity has been reserved. We study the joint provisioning and scheduling question.

We model timing requirements directly: each job type carries a soft deadline d_i. A job is tardy if it remains incomplete past its deadline; a tardy job incurs a cost of w_i per slot until it completes. This soft real-time formulation lets the job-to-instance assignment trade compute price against the risk of tardiness, and reduces to a hard-deadline model when w_i is large relative to the VM premium. The deadline structure has a useful consequence: once a job is tardy, its optimal cost-to-go is a closed-form linear function of remaining steps (Lemma L2). The same dynamic program that prices tardiness tells the scheduler which instance to assign a job to and the provisioner how much capacity to reserve.

Contributions.

- A joint model of provisioning and scheduling on heterogeneous instance types for heterogeneous jobs with soft deadlines, formulated as an infinite-horizon average-cost MDP. Pure FaaS is the no-reservation extreme of this model (Section II).
- A price-equalizing relaxation that decouples the MDP into one single-job dynamic program (DP) per job type, and a closed-form lower bound on the optimum that sums per-type DP values weighted by arrival rates. The DP value has a closed form once a job is tardy (Section III).
- A deployable policy that reuses each per-type DP as its scheduling rule and provisions each instance type

by a square-root rule on top of the workload that DP induces; no separate workload-forecasting step is needed (Section IV).
- An asymptotic-optimality result: the policy's gap relative to the unknown optimum is inversely proportional to the square root of the load. The excess cost is identified exactly as a sum of per-instance newsvendor mismatch costs (Section V).
- An evaluation across 482 EC2 instance types showing a realized gap under $\sim 5\%$ that shrinks with scale, and a mapping of the result onto AWS Lambda Managed Instances (Section VI).

II. System Model and Problem

We model a single cloud customer running a job-processing platform over slotted time $t = 0, 1, 2, \ldots$. A slot is the provider's billing and autoscaling granularity (e.g., a minute). The customer chooses, once, how much reserved capacity to hold of each instance type, and then, every slot, how much on-demand capacity to add and how to assign active jobs to capacity.

A. Jobs and deadlines

There are I *job types*. Type-i jobs (e.g., model-training, batch-ETL, video-transcoding, web-crawling, or inference-serving jobs) arrive as an independent discrete-time Poisson stream: the number of type-i arrivals in a slot is $\mathrm{Poisson}(\lambda_i)$, independent across slots and types. A type-i job must complete $p_i \in \mathbb{N}$ *processing steps* and carries a *soft deadline* $d_i \in \mathbb{N}$ slots after arrival. A job is *tardy* if it remains incomplete past its deadline. A tardy job pays w_i per slot until it completes (the soft real-time / weighted-tardiness model [6]). The hard-deadline variant is the limit $w_i \to \infty$.

B. Instances and affinity

The customer may run jobs on J *VM instance types* $j \in \{1, \ldots, J\}$ (e.g., `c5.xlarge`, `m5.2xlarge`, `p3.8xlarge`), plus an *idle* option $j = 0$ (the job is not served this slot). When a type-i job is run for one slot on instance type j, one of its steps completes with probability $\alpha_{ij} \in [0, 1]$, the *job–instance affinity*; with the complementary probability the step does not complete. The idle option makes no progress, $\alpha_{i0} = 0$. Affinity captures how well an instance fits a job: a GPU instance has $\alpha \approx 0.99$ for model training and far less for a crawl. We require:

Assumption A1. *For every job type i,* $\max_{j \geq 1} \alpha_{ij} > 0$.

C. State of a job

At the start of a slot, an active type-i job is described by (e, k): its *elapsed time* e (slots since arrival) and the number $k \in \{0, \ldots, p_i\}$ of *stages completed*. A newly arrived job is in state $(0, 0)$. If assigned to instance type j for the slot, then in the next slot its state becomes $(e{+}1,\ k{+}1)$ with probability α_{ij} and $(e{+}1,\ k)$ otherwise. The job is complete when $k = p_i$. A job is tardy if $e \geq d_i$.

In scheduling terms, the release time r_i of a particular arrival is the slot at which it appears, so $e = t - r_i$. The job's *lateness* is $L_i = e - d_i$ and its *tardiness* is $T_i = (L_i)^+$. The cost in (2) below then has the familiar form $\sum_i w_i T_i$ paid one slot at a time, consistent with the weighted-tardiness objective $1 \,\|\, \sum w_i T_i$ of classical scheduling theory [6].

D. Decisions, capacity, and cost

The customer's decisions are:

- **Provisioning (once, at $t = 0$):** $N_j \in \mathbb{N}_0$ reserved VMs of each instance type j.
- **On-demand scaling (each slot t):** $n_{j,t} \in \mathbb{N}_0$ on-demand VMs of type j to spin up.
- **Assignment (each slot t):** a per-slot assignment of active jobs to instance types; we write $a_{ijt}(e, k)$ for the number of type-i jobs in state (e, k) assigned to type j.

Each VM processes one job for one slot, so the number of jobs assigned to type j in slot t cannot exceed the capacity available:

$$\sum_i \sum_{e,k} a_{ijt}(e, k) \;\leq\; N_j + n_{j,t}, \qquad j = 1, \ldots, J. \tag{1}$$

On-demand capacity is always purchasable ($n_{j,t}$ is unbounded), so (1) is never infeasible: unmet workload overflows onto on-demand VMs. VMs are *not* a waiting queue. A job that cannot get a (cheap) reserved VM is served immediately on a (more expensive) on-demand VM. The model is therefore an *overflow / loss* model on capacity, and the provisioning analysis below is a newsvendor problem rather than a queueing problem. In the classical newsvendor problem [7], a buyer commits to an order quantity before observing a random demand, paying c_o per unsold unit and c_u per unit of unmet demand; the cost-minimizing order is the $c_u/(c_o{+}c_u)$ quantile of the demand distribution. Here the order is the reserved-VM count N_j, c_o is the per-slot reservation rate paid on idle capacity, and c_u is the on-demand premium paid on overflow.

The cost incurred in slot t under a policy π is

$$C_t^\pi = \underbrace{\sum_{j=1}^{J} \left(c_j^R N_j^\pi + c_j^O \, n_{j,t}^\pi \right)}_{\text{VM cost}} + \underbrace{\sum_{i=1}^{I} \sum_{e \geq d_i} \sum_{k} w_i \, x_{i,t}^\pi(e, k)}_{\text{tardiness penalty}}, \tag{2}$$

where $x_{i,t}^\pi(e, k)$ is the number of type-i jobs in state (e, k) present in slot t. Reserved VMs cost c_j^R per slot whether or not they are used; on-demand VMs cost c_j^O per slot per VM spun up. Reserved capacity is never more expensive than on-demand:

Assumption A2. $0 \leq c_j^R \leq c_j^O$ *for every instance type j.*

E. Objective and the FaaS extreme

A policy π fixes the reserved capacities $\{N_j\}$ and, in each slot, the on-demand and assignment decisions as a function

of the current configuration of active jobs. We minimize the long-run average expected cost per slot,

$$g(\pi) = \limsup_{T\to\infty} \frac{1}{T}\,\mathbb{E}\left[\sum_{t=0}^{T-1} C_t^{\pi}\right], \quad \mathrm{OPT} = \min_{\pi} g(\pi). \tag{3}$$

Pure FaaS / serverless is exactly the extreme $N_j = 0$ for all j. Full reservation is the opposite extreme. The model spans the entire FaaS–reserved continuum, and (3) asks for the cost-optimal point on it.

F. Intractability of the exact problem

Problem (3) is an infinite-horizon average-cost MDP [8]. Its state must record the multiset of all active jobs' triples (i, e, k); its action must assign every active job to one of $J{+}1$ instance types subject to (1) and choose $\{n_{j,t}\}$. The state space grows without bound, the action space is combinatorial, and the single reserved set $\{N_j\}$ *couples all job types together*. The next section removes the coupling by a pricing argument and recovers a tractable lower bound.

a) Notation, in scheduling terms: Our primitives map cleanly onto classical scheduling: d_i is the relative deadline (due date), p_i is the processing length in discrete stages, r_i is the release time, and w_i is the weighted-tardiness weight. The provider-side quantities are cloud-specific: N_j is the up-front reserved-capacity commitment for instance type j, $n_{j,t}$ is the on-demand spillover in slot t, and $\alpha_{ij} \in [0, 1]$ generalizes a heterogeneous-machine speed by encoding both the (i,j)-fit and the reliability of completing a step in one slot. A reader more comfortable with $1 \,\|\, \sum w_i T_i$ can read the problem as multi-machine weighted tardiness on J machine types whose counts N_j we must purchase before observing the workload.

III. A Decomposition Lower Bound

The coupling in (3) comes entirely from the price gap between reserved and on-demand capacity: if there were no gap, there would be no reason to reserve, no shared set of VMs to contend for, and job types would not interact. We make this precise; the resulting decoupled problem yields both a computable lower bound on OPT and the instance-assignment rule used in Section IV.

A. The price-equalizing relaxation

Let OPT' denote the optimal long-run average cost of the *relaxed* problem obtained from (3) by setting on-demand prices equal to reserved prices, $c_j^O := c_j^R$ for all j, and leaving everything else unchanged.

Lemma L1. $\mathrm{OPT}' \le \mathrm{OPT}$.

Proof. Fix any policy π. Its per-slot cost (2) is nondecreasing in each c_j^O because $n_{j,t}^{\pi} \ge 0$. Lowering every c_j^O to c_j^R (legitimate by Assumption A2) therefore does not increase C_t^{π} in any slot, so $g'(\pi) \le g(\pi)$. Taking the minimum over policies gives $\mathrm{OPT}' \le \mathrm{OPT}$. □

B. The single-job dynamic program

In the relaxed problem, reserved and on-demand capacity cost the same c_j^R per slot, so reserving capacity that ever sits idle is wasteful while reserving capacity that is always used costs the same as buying it on demand. Reservations are, thus, not very useful: set $N_j = 0$ and serve every assigned job-step on on-demand capacity. Because on-demand capacity is unbounded, (1) never binds, and *the jobs no longer interact*. For each job, one would have to make similar per-slot decisions in isolation.

For a type-i job, define the per-slot tardiness rate $c_i(e) = w_i\,\mathbf{1}[e \ge d_i]$ and the optimal expected remaining cost $f_i(e, k)$ from state (e, k), with $f_i(\cdot, p_i) = 0$ and, for $k < p_i$,

$$\begin{aligned} f_i(e,k) = \min_{j\in\{0,\dots,J\}} \Big\{ & c_i(e) + c_j^R \\ & + \alpha_{ij}\, f_i(e{+}1, k{+}1) \\ & + (1-\alpha_{ij})\, f_i(e{+}1, k) \Big\}. \end{aligned} \tag{4}$$

The tardiness $c_i(e)$ is charged every slot the job is alive and tardy, independent of the instance chosen; the VM price c_j^R and the progress probability α_{ij} depend on the choice j.

Lemma L2 (Tardy-state closed form)**.** *For $e \ge d_i$, $f_i(e, k)$ is independent of e; writing $g_i(k) := f_i(e, k)$ for any $e \ge d_i$,*

$$g_i(k) = (p_i - k) \cdot \min_{j\ge 1} \frac{c_j^R + w_i}{\alpha_{ij}}. \tag{5}$$

Proof. For $e \ge d_i$, $c_i(e) = w_i$ and every successor state is also tardy, so (4) maps tardy states to tardy states with a constant per-slot penalty; hence the value depends on k only. Once tardy, completing one stage using a fixed instance $j \ge 1$ takes a $\mathrm{Geometric}(\alpha_{ij})$ number of slots, each costing $w_i{+}c_j^R$, for an expected per-stage cost $(w_i + c_j^R)/\alpha_{ij}$. The idle action $j = 0$ has $\alpha_{i0} = 0$, so it never completes the stage and accrues infinite expected cost; it is dominated by any $j \ge 1$ with $\alpha_{ij} > 0$ (which exists by Assumption A1) and is therefore excluded from the minimum. Stages are independent; the optimal stationary choice minimizes the ratio over $j \ge 1$. □

Lemma L2 bounds the tail, so (4) reduces to a finite backward recursion: initialize $f_i(d_i, k) = g_i(k)$ and sweep $e = d_i{-}1, \dots, 0$ over $k \in \{0, \dots, p_i\}$.

a) Complexity: The per-type state space is $(d_i{+}1){\times}(p_i{+}1)$ (elapsed time and stages completed) and the per-step action space is $J + 1$ (one of J instance types or the idle action), so the recursion runs in $O(d_i p_i J)$ time per job type and $O\big(J \sum_i d_i p_i\big)$ in total. In every configuration of our evaluation this completes in well under a second on a laptop. The main contrast is with the *joint* MDP induced by (3): its state space is the cross product of per-type occupancy vectors, exponential in I, so directly solving it is intractable. Decoupling reduces the problem to I independent recursions that scale only in each type's deadline window, processing length, and the number of instance types.

Its minimizer defines the *optimal assignment*

$$j^\star(i,e,k) \in \arg\min_j \Big\{ c_j^R + \alpha_{ij} f_i(e{+}1,k{+}1) + (1-\alpha_{ij}) f_i(e{+}1,k) \Big\}, \tag{6}$$

which we reuse as the scheduling rule of our policy.

C. Obtaining a lower bound

Theorem T1 (Decomposition lower bound)**.** *Under Assumptions A1 and A2,*

$$\mathrm{OPT} \;\geq\; \mathrm{LB} \;:=\; \sum_{i=1}^{I} \lambda_i \, f_i(0,0), \tag{7}$$

and $\mathrm{LB} = \mathrm{OPT}'$.

Proof. By Lemma L1 it suffices to show $\mathrm{OPT}' = \sum_i \lambda_i f_i(0,0)$. The relaxed problem decouples: the capacity constraint never binds and the per-slot cost (2) is additive over jobs, so an optimal relaxed policy serves each job by an optimal single-job policy independently. The per-job DP (4) solves a finite total-cost problem (not an average-cost-per-stage value function): the optimal expected *total* cost accrued by a single type-i job over its lifetime is, by definition, $f_i(0,0)$, which is finite by Assumption A1 and Lemma L2. Treating each type-i arrival as a renewal whose reward is this finite total cost, the renewal-reward theorem gives a long-run average type-i cost of $\lambda_i f_i(0,0)$. □

The gap between LB and the cost of any policy is, as the next section shows, the price of holding finite reserved capacity instead of the unbounded cheap capacity the relaxation pretends is available.

IV. Assignment, Workload, and Provisioning

The lower bound came from imagining unbounded cheap capacity. In practice we commit to a finite reserved capacity $\{N_j\}$ up front and pay the on-demand premium whenever the workload exceeds it. We now turn the single-job DP into a deployable policy by (i) reading off the per-instance-type workload the DP assignment induces, (ii) showing that workload is Poisson, and (iii) sizing each reserved set of VMs by a newsvendor rule. The excess cost of the resulting policy over LB turns out to be *exactly* a sum of newsvendor mismatch costs.

A. Poisson arrivals

Fix the optimal assignment $j^\star(i,e,k)$ from (6). Run the infinite-server (relaxed) system in steady state and let D_j be the number of type-j VMs requested in a slot. Let $q_i(e,k)$ be the steady-state probability that a type-i job is in state (e,k) (computed alongside the DP from $\{\alpha_{ij}\}$ and the assignment).

Theorem T2 (Poisson aggregate workload)**.** *In steady state,* $D_j \sim \mathrm{Poisson}(\mu_j)$ *with*

$$\mu_j = \sum_{i=1}^{I} \lambda_i \sum_{(e,k):\, j^\star(i,e,k)=j} q_i(e,k), \tag{8}$$

and the demands $(D_1,\ldots,D_J)$ *are mutually independent.*

Proof sketch. In the relaxed (infinite-server) system, jobs are served independently from release to completion, and Lemma L2 ensures each job's expected lifetime is finite under $j^\star$; the joint occupancy process therefore admits a stationary distribution by the standard discrete-time $M/G/\infty$ result [9], and the per-state probabilities $q_i(e,k)$ are well defined. A type-i job present in slot t with elapsed time e arrived in slot $t-e$; the number of such arrivals is $\mathrm{Poisson}(\lambda_i)$ and is independent across cohorts and types. By Poisson thinning (coloring), the count of type-i jobs in state (e,k) in slot t is $\mathrm{Poisson}(\lambda_i q_i(e,k))$, independent across (i,e,k). The workload for type-j VMs sums independent Poisson counts over the states assigned to j; the workloads D_j partition disjoint state sets and are mutually independent. Convergence of (8) follows from Lemma L2. □

The rates μ_j are produced *by the DP itself*: the same assignment that yields the lower bound yields the workload to be provisioned, with no separate workload-forecasting step. And because the system is an overflow (loss) system rather than a delay queue, the provisioning decision is governed by the marginal per-slot workload distribution $\mathrm{Poisson}(\mu_j)$, not by queue dynamics.

B. The proposed policy

We define the policy $\pi^\star$ by composing DP-based assignment with newsvendor provisioning.

Scheduling (every slot). Assign each type-i job in state (e,k) to instance type $j^\star(i,e,k)$ from (6). If a reserved VM of that type is free, use it (cost c_j^R); otherwise spin up an on-demand VM (cost c_j^O).

Provisioning (once, at $t=0$). Reserve

$$N_j^\star = \Big\lceil \mu_j + z_j \sqrt{\mu_j} \,\Big\rceil, \quad z_j = \Phi^{-1}\!\left(\frac{c_j^O - c_j^R}{c_j^O} \right), \tag{9}$$

where Φ is the standard normal CDF and z_j is the *cost-optimal quantile*.

The tuning quantity z_j depends only on the on-demand premium $(c_j^O - c_j^R)/c_j^O$: a large premium pushes $z_j > 0$ (reserve above the mean, aggressively); a small premium drives $z_j < 0$ (reserve below the mean, or not at all: go serverless).

C. Optimality gap

Theorem T3 (Exact gap)**.** *Let* μ_j *be as in* (8) *and* $D_j \sim \mathrm{Poisson}(\mu_j)$*. For any choice of reserved capacities* $\{N_j\}$*, the policy that uses DP-based assignment with these reservations has long-run average cost*

$$\mathrm{Cost}{-}\mathrm{LB} = \sum_{j=1}^{J} \mathbb{E}\Big[c_j^R \,(N_j - D_j)^+ + (c_j^O - c_j^R)\,(D_j - N_j)^+ \Big]. \tag{10}$$

The right-hand side is separable and convex in each N_j, and is minimized by the newsvendor critical fractile

$$\mathbb{P}(D_j \le N_j^\star) = \frac{c_j^O - c_j^R}{c_j^O}, \tag{11}$$

which the normal approximation $D_j \approx \mathcal{N}(\mu_j, \mu_j)$ solves as (9).

Proof. Tardiness costs and the assignment are identical under this policy and under the relaxed optimum: both assign by $j^\star$ and a job's progress and tardiness depend only on the assignment. The two costs differ only in the VM term. In a slot with type-j workload D_j, the policy pays $c_j^R N_j + c_j^O (D_j - N_j)^+$; the relaxed optimum pays $c_j^R D_j$. Using $N_j - D_j = (N_j - D_j)^+ - (D_j - N_j)^+$, the slot-wise difference is the integrand of (10). Taking expectations over $D_j \sim \text{Poisson}(\mu_j)$ gives (10).

Expression (10) is the classical newsvendor cost with overage $c_o = c_j^R$ (per idle reserved VM) and underage $c_u = c_j^O - c_j^R$ (per on-demand spillover VM). Its discrete first difference changes sign when $\mathbb{P}(D_j \le N_j) = c_u/(c_o + c_u)$, giving (11); the normal approximation yields (9). □

Equation (10) is the FaaSen/Un-FaaSen trade-off in dollars: reserve too little and you pay the on-demand premium on the overflow; reserve too much and you pay for idle commitments. The on-demand premium alone sets the optimal balance, and the resulting cost grows only as the *square root* of workload. Our next step is to clarify this rate.

a) Justifying the normal approximation: The exact newsvendor critical fractile in (11) can be evaluated against the Poisson CDF, but the answer is an integer quantile: a number for each μ_j rather than a formula, and one that obscures the structure we want to expose. Substituting the normal approximation $D_j \approx \mathcal{N}(\mu_j, \mu_j)$ (valid by the Poisson CLT once μ_j is moderate, and standard in square-root staffing [10], [11]) makes the headroom explicit as $z_j\sqrt{\mu_j}$, with the tuning quantile z_j depending only on the on-demand premium. The Poisson-to-normal substitution contributes an $O(1)$ rounding-plus-tail error per instance type, which we group into the $O(J)$ term in Proposition P1; Theorem T4 below shows that the *exact* Poisson gap shares the same $\Theta(\sqrt{\theta})$ scaling; the approximation affects only the constants.

V. Asymptotic Optimality

Theorem T3 reduced the cost of $\pi^\star$ relative to the lower bound to a sum of per-type newsvendor costs. We now evaluate that sum in closed form, show it grows as the square root of the workload, and conclude that $\pi^\star$ is asymptotically optimal as the workload scales.

A. Analyzing the gap

Lemma L3 (Newsvendor cost at the optimum)**.** *Let $D \sim \mathcal{N}(\mu, \mu)$, overage c_o, underage c_u, and $z = \Phi^{-1}(c_u/(c_o + c_u))$. At $N^\star = \mu + z\sqrt{\mu}$,*

$$c_o\,\mathbb{E}\,(N^\star - D)^+ + c_u\,\mathbb{E}\,(D - N^\star)^+ = (c_o + c_u)\,\sqrt{\mu}\,\varphi(z), \tag{12}$$

where φ is the standard normal density.

Proof sketch. With $\sigma = \sqrt{\mu}$ and standard normal loss $\mathcal{L}(z) = \varphi(z) - z(1 - \Phi(z))$, one has $\mathbb{E}\,(D - N)^+ = \sigma\mathcal{L}(z)$ for $z = (N - \mu)/\sigma$, and $\mathbb{E}\,(N - D)^+ = \sigma[\varphi(z) + z\Phi(z)]$. At $z = z$, $\Phi(z) = c_u/(c_o + c_u)$ and $1 - \Phi(z) = c_o/(c_o + c_u)$; substituting collapses the z-terms and leaves $(c_o + c_u)\sigma\varphi(z)$. □

Applying Lemma L3 per instance type with $c_o = c_j^R$, $c_u = c_j^O - c_j^R$ (so $c_o + c_u = c_j^O$), and combining with Theorem T3:

Proposition P1 (Closed-form gap)**.** *Under the normal approximation,*

$$\Delta^{\pi^\star} := \text{Cost}(\pi^\star) - \text{LB} = \sum_{j=1}^{J} c_j^O\,\sqrt{\mu_j}\,\varphi(z_j) + O(J), \tag{13}$$

where the $O(J)$ term accounts for integer rounding of $N_j^\star$ and the Poisson-to-normal approximation error.

B. Workload scaling

We scale the workload by a factor $\theta \ge 1$: every arrival rate becomes $\lambda_i^{(\theta)} = \theta\,\lambda_i$, with deadlines, affinities, prices, and penalties held fixed. Arrivals and the resulting capacity grow together. This is the many-server / QED regime, not the heavy-traffic regime where utilization climbs against a fixed set of reserved VMs.

Theorem T4 (Asymptotic optimality)**.** *Under Assumptions A1 and A2, as $\theta \to \infty$,*

$$\begin{aligned} &\text{LB}(\theta) = \Theta(\theta), \quad \Delta^{\pi^\star}(\theta) = \Theta(\sqrt{\theta}), \\ &\frac{\text{Cost}(\pi^\star) - \text{OPT}}{\text{OPT}} = O\!\left(\frac{1}{\sqrt{\theta}}\right) \to 0. \end{aligned} \tag{14}$$

Proof. The assignment $j^\star$, state probabilities $q_i(e, k)$, per-job value $f_i(0, 0)$, and per-type quantile z_j depend only on $\{\alpha_{ij}, c_j^R, w_i, d_i, p_i\}$, which are fixed under scaling. Hence $\text{LB}(\theta) = \theta\,\text{LB}(1) = \Theta(\theta)$. The workload means scale linearly, $\mu_j(\theta) = \theta\,\mu_j$. Under the normal approximation of Proposition P1, $\Delta^{\pi^\star}(\theta) = \sqrt{\theta}\sum_j c_j^O\sqrt{\mu_j}\varphi(z_j) + O(J) = \Theta(\sqrt{\theta})$. The same $\Theta(\sqrt{\theta})$ scaling holds for the exact Poisson cost in Theorem T3, since $\mathbb{E}\,|D_j - \mu_j| = \Theta(\sqrt{\mu_j})$ for $D_j \sim \text{Poisson}(\mu_j)$ as $\mu_j \to \infty$; only the constants depend on the normal approximation. Since $\text{OPT} \ge \text{LB}$ and $\text{Cost}(\pi^\star) - \text{OPT} \le \Delta^{\pi^\star}$, the relative gap is at most $\Theta(\sqrt{\theta})/\Theta(\theta) = \Theta(1/\sqrt{\theta})$. □

a) Interpretation: This is the many-server / quality-and-efficiency-driven (QED) regime of large-scale service systems [10], [11]: the reserved capacity $N_j^\star = \mu_j + z_j\sqrt{\mu_j}$ tracks the mean workload plus a headroom that scales as the standard deviation $\sqrt{\mu_j}$, i.e. *sublinearly*. Larger deployments need proportionally less buffer, so cost control gets tighter with scale. Deadlines enter through the per-job value f_i and the assignment $j^\star$: tighter deadlines or larger tardiness penalties w_i push the DP toward higher-affinity (and often more expensive) instances. This shifts the workload mix μ_j but leaves the $O(1/\sqrt{\theta})$ rate unchanged.

TABLE I
AVERAGE RELATIVE OPTIMALITY GAP $\Delta^{\pi^\star}/\mathrm{LB}$ OVER 482 EC2 INSTANCE TYPES (1-YEAR NO-UPFRONT RESERVED VS. ON-DEMAND), 100 RANDOM INSTANCES $\times$ 15 SCENARIOS.

Job arrival rate	Avg. gap (I=6)	Avg. gap (I=30)
$\lambda \sim \mathcal{U}[1,4]$	4.89%	3.17%
$\lambda \sim \mathcal{U}[2,8]$	3.52%	2.66%
$\lambda \sim \mathcal{U}[3,10]$	3.15%	2.19%

VI. NUMERICAL EVALUATION AND DISCUSSION

Our analysis predicts a small relative gap at realistic scale that shrinks as $O(1/\sqrt{\theta})$. We evaluate this on AWS EC2 instance families. Because both the lower bound (Theorem T1) and the policy gap (Proposition P1) are available in closed form, the evaluation is a direct computation rather than a simulation: for each instance we solve the per-job DPs, obtain the workload rates μ_j and quantiles z_j, and identify the ratio $\Delta^{\pi^\star}/\mathrm{LB}$.

a) Setup: We use 482 EC2 instance types spanning five families (C, M, R, P, G), with their on-demand prices c_j^O and the corresponding reserved (1-year, no-upfront) prices c_j^R. We instantiate $I \in \{6, 30\}$ job types with affinities α_{ij} drawn to reflect realistic job–instance fit (e.g., $\alpha \approx 0.99$ for accelerator-suited jobs on P/G instances, ≈ 0.25 for the same jobs on general-purpose instances), and deadlines d_i and step counts p_i in realistic ranges. Tardiness weights are set type-by-type as $w_i = \kappa \max_j c_j^O$ with $\kappa = 1$, so that one tardy slot costs roughly one slot of the most expensive on-demand VM. This value is large enough that tardy states are visibly penalized in the DP, yet small enough that no single weight dominates the cost. For each configuration we draw arrival rates λ_i uniformly from one of three ranges spanning low to high throughput, generate 100 random instances per scenario across 15 scenarios, and report the average relative gap.

b) Sensitivity to the tardiness weight: The weight w_i enters (4) only through the per-slot rate $c_i(e) = w_i \mathbf{1}[e \geq d_i]$ and through the tardy-state ratio $(c_j^R + w_i)/\alpha_{ij}$ in (5). Both are monotone and continuous in w_i: small weights leave the DP close to a cost-only choice (cheap, slow instances), and large weights push assignment toward the highest-affinity instance regardless of price. The induced workload mix μ_j therefore shifts smoothly with w_i, but the asymptotic gap rate $O(1/\sqrt{\theta})$ from Theorem T4 is unchanged: tighter deadlines or heavier weights move the workload, not the scaling.

c) Results: Table I reports the average relative gap $\Delta^{\pi^\star}/\mathrm{LB}$. The gap is small: under $\sim 5\%$ in the worst (lowest-throughput) case and as low as 2.2% at higher throughput. Within each column, raising the arrival rate shrinks the gap, consistent with the $O(1/\sqrt{\theta})$ prediction. More job types ($I = 30$ versus 6) lower the gap further. A richer job mix spreads the workload across more instance types, so the per-type means μ_j are larger and the square-root effect has more to work with.

d) When to FAASEN *and when to* UN-FAASEN*:* The cost-optimal quantile $z_j = \Phi^{-1}((c_j^O - c_j^R)/c_j^O)$ gives a sharp rule. Reserve capacity (UN-FAASEN) when the on-demand premium is large (so $z_j > 0$, reserve above the mean), when the workload is steady (so the headroom $z_j\sqrt{\mu_j}$ is small relative to the reserved capacity), and when throughput is high (so the $O(1/\sqrt{\theta})$ regime makes reservation pay off with a tight guarantee). Stay serverless (FAASEN) when the premium is small ($z_j \leq 0$), when the workload is bursty or low-volume, or when tight SLAs require absorbing emergency spikes that idle reserved capacity cannot justify. This is the regime where a reactive harvester like UNFAASENER [5] adds value, and the two approaches compose: provision the steady core proactively by the square-root rule, and let a reactive layer harvest the spiky tail.

e) AWS Lambda Managed Instances: AWS Lambda Managed Instances (announced at re:Invent 2025) [12] makes the FaaS–reserved boundary a deliberate product decision: it lets a customer run Lambda functions on EC2 instances of their own choosing, behind the serverless developer experience, with Reserved Instance or Savings Plan pricing applied to the underlying set of VMs. The provider has productized the decision we study. In our terms, standard Lambda is the pure-FaaS approach ($N_j = 0$ at price c_j^O); attaching a function to a managed reserved set is the un-FaaSened regime ($N_j > 0$ at price c_j^R). The quantile z_j answers *when* to attach; the square-root rule tells us *how much* capacity we should reserve. AWS's own guidance (reserve-priced managed instances for "high-volume, predictable" workloads) matches what Theorem T4 predicts qualitatively.

VII. RELATED WORK

Serverless cost and the FaaS/reserved boundary. Our work is motivated by UNFAASENER [5]: it offloads functions onto idle non-serverless resources to save cost, but it is reactive and offers no guarantee on cost relative to the best achievable. Several prior systems combine serverless with VMs to cut cost while keeping latency low [2], [3], [4], but they are either not deadline-aware or do not use deadlines fully. Kaffes et al. [13] give principled scheduling for serverless functions, but for a single function abstraction and without a reserved-vs-on-demand provisioning decision. We are proactive where these are reactive or descriptive, and provide an optimality guarantee none of them target.

Scheduling heterogeneous cloud workloads. The multiserver-job line [14] fixes server capacity and schedules within it; we also choose the capacity, and our heterogeneity is in the job–instance affinity α_{ij} rather than in server counts per job. Online competitive provisioning yields worst-case ratios; we target average-case asymptotic optimality in the QED regime that large deployments inhabit.

Many-server dimensioning. The square-root provisioning rule (square-root staffing in the queueing-theory literature) goes back to [10], [11], [9]. The novelty here is not the rule but that (i) the workload rates μ_j to be provisioned are produced by a job-assignment DP over job–instance affinities, and (ii) the safety coefficient z_j is set by the on-demand premium $(c_j^O - c_j^R)/c_j^O$ alone, not by waiting-cost trade-offs.

Real-time scheduling. Weighted-tardiness objectives are standard in soft real-time and scheduling theory [6]; what is new here is folding tardiness into a provisioning MDP whose decomposition leaves the per-job DP incorporating the deadline structure (Lemma L2).

VIII. Conclusion

Job types interact based on the price gap between reserved and on-demand capacity. If this gap is closed then the Markov Decision Process is separable: each type gets its own single-job dynamic program, solvable in $O(d_i p_i J)$ time. From there the assignment rule, the per-type Poisson workload, the lower bound, and the square-root provisioning size can all be derived using the same recursive formulation.

The tardiness penalty is included in the per-job dynamic program, which shifts which instance types accumulate workload. As a consequence, deadlines do not make provisioning harder. The provisioning layer (Poisson per-slot workload, the square-root provisioning rule) is the same whether jobs have tight deadlines or none. Tighter deadlines push the assignment toward higher-affinity (and usually more expensive) instances.

With new systems support, we can use a simple pricing knob to determine whether a job executes on a reserved VM instance or using a FaaS invocation. AWS Lambda Managed Instances takes what used to be a per-invocation product and bills it under EC2 per-instance pricing, so a customer picks either extreme of our model through one API. z_j determines whether we use a reserved instance or an FaaS invocation; the square-root rule determines the size of the reserved allocation.

Several open directions remain:

- Spot instances as a third pricing tier;
- Online learning of arrival rates and affinities when they are unknown;
- Supporting hard deadlines as the $w_i \to \infty$ limit;
- Handling non-Poisson arrivals.

References

[1] C. Lin, Y. Ma, and M. Shahrad, "Demystifying serverless costs on public platforms: Bridging billing, architecture, and os scheduling," in *Proceedings of the 21st European Conference on Computer Systems*, ser. EUROSYS '26. ACM, 2026, p. 1964–1981.

[2] J. R. Gunasekaran, P. Thinakaran, M. T. Kandemir, B. Urgaonkar, G. Kesidis, and C. Das, "Spock: Exploiting serverless functions for slo and cost aware resource procurement in public cloud," in *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*, 2019, pp. 199–208.

[3] C. Zhang, M. Yu, W. Wang, and F. Yan, "MArk: exploiting cloud services for cost-effective, SLO-aware machine learning inference serving," in *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference*, ser. USENIX ATC '19. USA: USENIX Association, 2019, p. 1049–1062.

[4] A. Raza, Z. Zhang, N. Akhtar, V. Isahagian, and I. Matta, "LIBRA: An economical hybrid approach for cloud applications with strict SLAs," in *2021 IEEE International Conference on Cloud Engineering (IC2E)*, 2021, pp. 136–146.

[5] A. Sadeghian *et al.*, "UnFaaSener: Latency and cost aware offloading of functions from serverless platforms," in *USENIX Annual Technical Conference (ATC)*, 2023, vERIFY author list and exact title.

[6] M. L. Pinedo, *Scheduling: Theory, Algorithms, and Systems*, 5th ed. Springer, 2016, weighted tardiness as a soft real-time / scheduling objective.

[7] P. H. Zipkin, *Foundations of Inventory Management*. McGraw-Hill, 2000.

[8] M. L. Puterman, *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. Wiley, 1994.

[9] M. Harchol-Balter, *Performance Modeling and Design of Computer Systems: Queueing Theory in Action*. Cambridge University Press, 2013.

[10] S. Halfin and W. Whitt, "Heavy-traffic limits for queues with many exponential servers," *Operations Research*, vol. 29, no. 3, pp. 567–588, 1981.

[11] S. Borst, A. Mandelbaum, and M. I. Reiman, "Dimensioning large call centers," *Operations Research*, vol. 52, no. 1, pp. 17–34, 2004.

[12] Amazon Web Services, "AWS Lambda managed instances," AWS re:Invent announcement, 2025, vERIFY exact product name, pricing, and citation form.

[13] K. Kaffes, N. J. Yadwadkar, and C. Kozyrakis, "Hermod: Principled and practical scheduling for serverless functions," in *ACM Symposium on Cloud Computing (SoCC)*, 2022.

[14] I. Grosof, Z. Hong, M. Harchol-Balter, and A. Scheller-Wolf, "The RESET and MARC techniques, with applications to multiserver-job analysis," *Proc. ACM Meas. Anal. Comput. Syst. (POMACS)*, 2023, vERIFY; multiserver-job scheduling under heavy traffic.