UNIVERSITY

LEVEL's Degree in Communications Engineering

LEVEL's Degree Thesis

Latency-Aware DNN Inference with Adaptive Batching for Edge Task Offloading

Supervisors

Prof. Carla Fabiana CHIASSERINI

Dr. Corrado POLLIGHEDU

Candidate

Sadegh JAMISHI

September 2025

Summary

Edge computer-vision systems need to satisfy low-latency requirements, even under scarce computation and network resource availability. The novelty of this thesis is the investigation of how admission control, batching, and concurrency need to be jointly designed to jointly maximize the number of task completions without deadline violations.

First, we perform an empirical characterization of modern inference frameworks (e.g., PyTorch, NVIDIA TensorRT, YOLO). The findings show that batching and parallelism benefit throughput, but hit diminishing returns as host-side processing saturates. Inspired by this, we present a communication–computation model which subsumes rate-dependent uploads, limited bandwidth, and asynchronous task arrivals in a single compact form.

To address the scheduling problem, we introduce an algorithm Greedy-JBAS, a simple batching algorithm based on earliest-deadline-first ordering with upload and inference feasibility checks. It achieves high-completion ratios, plans in milliseconds, and almost matches the performance of more costly optimization-based formulations (e.g., Gurobi), thereby setting a new high-water mark for fixed-batch or mobile-edge-computing baselines. Overall, the contributions of this thesis include: (i) a reproducible empirical mapping of batching and concurrency behavior in modern inference stacks, (ii) a formal, yet practical, unified communication–computation model for edge inference, and (iii) a scalable scheduler that does not trade deployability for efficiency. These contributions aim to provide actionable guidance for building latency-aware edge AI pipelines, and open new doors to host-side parallelism opportunities. Keywords: Edge AI, deadline-aware scheduling, admission control, batching, concurrency, latency SLAs.

Acknowledgements

First and foremost, I would like to express my deepest gratitude to Prof. Carla Fabiana Chiasserini for her invaluable support, insightful advice, and constant encouragement throughout this journey. Her guidance helped me overcome the many challenges I faced, and it has been a true privilege to work under her supervision. I am also profoundly thankful to Dr. Corrado Pullighedu, whose mentorship, patience, and kindness have been a source of both professional growth and personal motivation. Without their help, this thesis would not have been possible.

I am grateful to HPC Polito for providing the computational resources that made my experiments feasible.

Finally, I would like to thank my family and friends for their kind support and encouragement. I also wish to thank my classmates at Politecnico di Torino for the good moments we shared, both in and outside the classroom. Even if our paths may never cross again, I will always carry these memories with me.

Table of Contents

List of Tables

List of Figures

Acronyms

AI
artificial intelligence

CV
Computer vision

ML
Machine learning

DNN
Deep Neural Network

IoT
Internet of thing

TRT
TensorRT

SLA
service level agreement

EDF
Earliest deadline first

OAS
Order and acceptance scheduling

SNR
Signal to noise ratio

AWGN

Additive white Gaussian noise

GPU

Graphics Processing Unit

CPU

Central Processing Unit

MILP

Mixed-Integer Linear Programming

Chapter 1

Introduction

1.1 Background and Motivation

Edge computer-vision (CV) systems are beginning to see widespread adoption, as they make it possible to process the continuous streams of images and videos directly on the data source, whether it is a mobile device, an embedded system, or at a localized edge server. These shifts in where and how we process data are being adopted to meet the demands of a wide range of application domains and use cases, including industrial inspection, autonomous robotics, traffic analytics, and safety monitoring. In the industrial setting, for instance, edge CV systems can help perform real-time quality control by processing images of items on a production line, making sure defects are detected and addressed on time. In autonomous robotics, edge CV systems allow machines to perceive the world around them with high accuracy, leading to more complex navigation and interaction with dynamic environments. In traffic analytics, edge CV can be leveraged to monitor vehicular flow, detect anomalies and patterns, and to optimize traffic management systems. In safety monitoring, these systems can be used to help keep public spaces safe, by processing the video feeds from surveillance cameras, spotting incidents and alerting the authorities as they occur. These use cases and many others show that edge CV systems promise a significant reduction of latency by keeping the distance that data has to travel to be processed as short as possible, and can help relieve the bandwidth bottleneck and cloud infrastructure limitations from having to send terabytes of video data to the cloud for processing and storage. [1]

In these and other settings, however, latency service-level agreements (SLAs) are as much at the core as accuracy: detections must arrive within tens of milliseconds to be of use, and sustained overload that lead to deadline misses can be far worse than temporary reductions in accuracy. For this to happen, the software that implements these systems needs to employ sophisticated algorithms and architectures that can

navigate the trade-offs between latency and accuracy at the edge, in real time. [2]

Meeting such SLAs is nontrivial for three reasons that are highly intertwined. First, the compute stack currently used in practice, deep neural networks deployed through PyTorch or NVIDIA TensorRT [3] and wrapped in inference frameworks such as Ultralytics YOLO or MMDetection, has non-trivial performance-resource trade-offs. Throughput generally increases with batch size and number of parallel streams, but only up to knee points beyond which additional batching or concurrency causes diminishing or even negative returns, due to memory pressure, launch overheads, and other host-side bottlenecks. Second, edge systems operate with limited and variable network capacity. Upload times are tied to link speed, which in turn depends on SNR and the particular bandwidth share allocated to the upload task; for multi-tenant edges, communication can often be the major contributor to the overall latency. Third, the incoming requests arrive at irregular intervals, with different deadlines, sizes, and inter-arrival frequencies.[4]

The server therefore needs to make hard choices about what tasks to accept and how to handle them during times of high load. Adaptive task offloading strategies can be put in place to deal with this, to dynamically change resource allocation in response to real-time network conditions and workload patterns, and may be used to boost the overall efficiency of edge CV systems, improving their timeliness and the performance of the system as a whole. Combined, these factors suggest that simply increasing the batch size or the number of streams will not be sufficient to guarantee SLA compliance. Edge inference requires a scheduling approach that is aware of such hardware bottlenecks, co-designing admission control with batching and concurrency choices, while taking into account the interaction with communication and compute delays.

1.2 Problem Statement

In this work, we consider a single edge node that needs to handle an input sequence of inference requests (often called jobs or tasks), each of which has an associated release time, a deadline, and an upload payload (e.g., a feature vector or an image). At a high level, the edge node can use three levers to control this processing: 1. Admission Control ($Y_k \in \{0, 1\}$): the binary choice of whether to accept or reject request (k), 2. Batching (π): the allocation policy for accepted requests to batches, each with a given size (B), 3. Concurrency (S): the number of parallel streams used to execute the batches. Then, the end-to-end latency (e2e) of each job (k) can be decomposed into two components: a communication time, which is determined by the uplink rate (r_k) as a function of SNR and bandwidth allocation, and a computation time, which is conditioned on the batch size and varies with the backend framework used (among others). From our empirical measurements,

we observe that given today's processing stacks, per-batch delay ($d(n)$) is well-approximated by the linear function $d(n) \approx an + b$ ($a > 0, b > 0$), with saturation effects that kick in for batch sizes ($B \approx 8$–16) and limited marginal gains from staggering launches beyond a certain point. We further find that concurrency provides near-linear gains only for values up to ($S \approx 5$–6), above which memory pressure and host-side overheads, largely in the form of Python-level bottlenecks in data preparation and model prediction, begin to dominate. [5, 6]

In this setting, our objective is to develop approaches to better coordinate the choice of (Y, π, B, S) to complete as many tasks as possible, which in turn translates into a higher application utility, all the while respecting each task's deadlines, as well as bandwidth and compute capacity constraints. Towards this goal, we first situate our work in the classical batched Order-Acceptance and Scheduling (OAS) problem setting: tasks come with deadlines and rewards, and the edge server has to jointly decide which tasks to accept and how to schedule accepted tasks. In this setting, "processing time" needs to be re-interpreted as a combination of upload and batch-execution delays. We then propose a simplified batched OAS model that has a natural and explicit representation of batching dynamics, and allows for closed-form batch-time updates derived directly from the empirical batch delay estimate inspired by what have been done in [7] and [8].

1.3 Research Questions

To address the many questions that emerge when tackling the problem of edge computing for CV applications in a rigorous manner, we formulate the following questions:

- RQ1. What are the interaction between batch size and number of parallel streams with respect to latency and throughput in real hardware (CPU and GPU) and under modern inference backends (e.g., PyTorch vs TensorRT)? More specifically, what are the critical performance metrics that one should aim to characterize to understand scalability limitations? For the edge AI community, understanding how hardware resources can best be used and combined in edge environments is important, as described in [9] and [10].

- RQ2. Is it possible to design a unified model that can accurately couple rate-dependent uploads (under realistic radio conditions) with batch-dependent computation, while maintaining sufficient accuracy for scheduling while being light enough to be computed in an efficient manner? If so, we could better tie together communication and compute and reduce inference complexity, as per the previous related works Bensalem et al. [11] and [12].

- Can a lightweight scheduling algorithm have near-optimal acceptance and completion rates compared to exact or relaxed MILP formulations, given the difficulty in scaling to realistic settings? What structural heuristics such as EDF algorithms with feasibility checks, can the algorithm leverage to gain competitive performance? For scheduling, we base our questions and sanity-checks in the literature [13, 14] and [15]

- How robust are our solutions to backend/architecture (i.e. PyTorch vs TensorRT) changes, outliers and errors in anticipated performance and delays and variations in radio conditions (bandwidth, SNR, etc)? What practical guidelines can be inferred for appropriate values of batch sizes and concurrency? We would like to formulate deployment guidelines that are useful and robust in practice, which has also been the goal of some related works [16], [17], and [18].

We hope to be able to answer all these questions and provide new insights into the design and operation of edge AI systems that are primed to face the challenge of real-time CV workloads.

1.4 Contributions

This thesis makes the following contributions to the state-of-the-art in edge AI and CV:(i) We provide a reproducible study of Ultralytics YOLO and MMDetection frameworks on two inference backends: PyTorch and TensorRT. Our experimental results lead to an empirical quantification of the region of optimal operating points for batch sizes and number of parallel streams ($B \approx 8$–16 and $S \approx 5$–6), and the host-side bottleneck that caps further scaling, with staggered offsets providing only marginal gains. These insights should inform model and algorithm design and development and are also useful to understand performance at the edge.[19, 20, 5]. (ii) We show how to cleanly extend the batched OAS model (JBAS) to include rate-dependent uploads and batch-dependent computation. This new model has a simple closed-form recursion for batch start times, even under asynchronous arrivals/deadlines. This represents a unified edge inference model that should facilitate more accurate scheduling. (iii) We present Greedy-JBAS, an algorithm built on an EDF batching policy, that is both simple and preserves the closed-form nature of timing updates. Greedy-JBAS simplifies feasibility checks for both upload and inference deadlines and is able to run in the order of milliseconds, adaptively tracking empirical stack behavior via ($d(n)$) and out-scale to problem sizes where no MILP can be found, due to problem size. [7] (iv) We provide performance evaluations of this approach in comparison to standard baselines with fixed batching/pipeline

and MEC that have been used in the past, or to MILP/relaxed formulations if a comparison is possible. In our experiments, we show that Greedy-JBAS is able to achieve exceptionally high rates of completed tasks, while having planning times orders of magnitude faster than a brute-force approach. (v) Finally, as part of this effort, we also include scripts to fully regenerate all timing fits, performance plots, and scheduling results from raw logs, as well as explicit configuration files of the hardware/software/dataset we use throughout our studies.

1.5 Thesis Outline

Chapter 2 provides a survey of edge inference, including the basics of batching/-concurrency behavior and communication-aware scheduling, with an emphasis on related works on batched Order-Acceptance and Scheduling (OAS). In particular, we identify and explicitly call out the key modeling and algorithmic gaps, that this thesis as a whole and our specific work in this area attempt to address [5, 7, 8] Chapter 3 lays out the system model we use throughout our investigation of the problem. In this section, we formalize how we go from a maximum-completion objective (hard deadlines) to simpler but more tractable formulations that cleanly couple rate-dependent uploads with our empirical batch delay model. We also present closed-form timing recursions required to support efficient batched execution. Chapter 4 presents the findings from our measurement study conducted on two frameworks (Ultralytics YOLO and MMDetection) and two backends (PyTorch and TensorRT). In this chapter, we show the empirical operating region of batch size (B) and concurrency (S), and characterize host-side bottlenecks via CPU/GPU profiling. Chapter 5 presents Greedy-JBAS, including its insertion logic (EDF based batching), feasibility check, algorithmic complexity, and implementation details (including how timing fits are used to drive the schedule). Chapter 6 presents the experimental results, including comparison to MILP formulations when possible as well as more traditional baselines. We perform sensitivity analysis on bandwidth/SNR variations and workload size, as well as ablation studies on batching, concurrency, and offsets. Chapter 7 discusses the implications of our work, as well as the limitations (e.g., backend dependence of the parameters (a) and (b), the single-node focus, and the effects of host-side parallelism, among others). We also present deployment guidelines that we extract from our results. Finally, Chapter 8 concludes the thesis by reflecting on our contributions and outlining possible future work. Some of these avenues include, but are not limited to, the investigation of a multi-model queue, formulation of energy-aware objectives, learned policies, and a robust C++ and multiprocess host pipeline.

Chapter 2

Background & Related Work

In this chapter, we situate the thesis in (i) edge inference architectures and associated SLAs (on-device vs. offload; batching & streams), (ii) scheduling/admission for DNN inference (batching, micro-batching, queueing), and (iii) communication modeling for uplink, rate, bandwidth sharing and SNR. We then relate it to existing Order-Acceptance and Scheduling (OAS) work — in particular batched OAS (JBAS-style) literature — and finally to framework backends (PyTorch vs TensorRT) in practice which make our empirical delay model in later chapters necessary. We end with a gap analysis explaining what has been done before and what we add to the literature.

This chapter will provide an in-depth analysis of existing literature and frameworks related to edge inference and scheduling. By comparing and contrasting various approaches and techniques, we aim to identify gaps and opportunities for further research in this field.

2.1 Edge inference & SLAs

Edge inference is the strategy of running trained DNNs close to the data source to improve performance by reducing end-to-end latency and offloading from the backhaul. Its novel deployment targets span a large range of nodes on the edge of the network from user devices such as mobile phones and robots, to micro-data centers in the base stations. The key design choice in the former edge inference is to run on-device, offload entirely to the cloud or use some partitioning that splits computation between the two. Collaborative intelligence or hybrid AI are examples where a framework (Neurosurgeon) automatically slices a trained network at the mobile/cloud boundary to trade off between latency and energy [21]. Comprehensive surveys on Edge AI have reviewed required system components in great detail (sensing, pre-processing and post-processing, on-device acceleration, and the role

of near-edge servers) and emphasized the need to respect latency SLAs as hard constraints during design [22].

Batching and multiple streams/instances are leveraged to efficiently use hardware accelerators within a given latency SLA. Batching amortizes the overheads of kernel and launch between multiple inputs. Streams or instances in a device allow multiple kernels to be run concurrently to exploit device concurrency and keep SMs busy. This has inherent bottlenecks (a knee point) after which adding more batches/streams can only hurt latency (lead to more memory pressure and host overheads). We show this in Chapter 4 quantitatively. Latency-aware batching has already been adopted in production systems and research prototypes to trade off the above benefits [23, 24]. We build our work on these motivations.

2.2 Scheduling/Admission for DNN Inference

Online prediction serving work has matured with focus on latency targets (SLO/SLAs) and maximizing throughput. Clipper built a system with model abstraction layer that uses adaptive, latency-aware batching per-model to achieve target latency while still enhancing throughput [23]. INFaaS extended the knobs to selection of model variants (accuracy/latency tradeoffs) and combines batching with SLO-aware admission and placement [25]. At the cluster-scale, Nexus shows that inference scheduling must consider batching and may even need to split a DNN across GPUs to hit SLOs while still retaining high utilization [24]. Very recently, Proteus and BCEdge have shown up to this adaptive batching with concurrency management and even DRL-based (learning-based) strategies to hit SLOs across workload dynamics and different device types [26, 16].

In the stream-processing literature, the complementary notion of latency-sensitive micro-batching was introduced to ensure all end-to-end deadlines for GPU pipelines even under peak load by offering algorithms for dynamically adjusting batch size coupled with analytical latency models [27]. These ideas — of adaptive batching, SLO-aware admission, and queueing — are the building blocks for this thesis; the innovation here is their integration into a batched-OAS model with explicit radio-aware upload times and a batch delay $d(n)$ that is empirically fit. We describe them in the coming sections.

2.3 Communication Model Basics

For the uplink, we model it by classical information-theoretic rate expressions (AWGN). For a single user with a given bandwidth W and instantaneous SNR γ, the corresponding Shannon capacity C can be computed as:

$$C = W \log_2(1 + \gamma). \tag{2.1}$$

As is well-known, this is the rate that can be supported under AWGN, see [28, 29]. In the shared edge setting with multiple users all active at the same time, each instantaneous rate r_k will also depend on that user's share of the bandwidth and the overall time-sharing among all k active users as well as their SNRs. This means that upload time needed for payload size ℓ_k for user k is:

$$t_k^{(o)} = \frac{\ell_k}{r_k}. \tag{2.2}$$

We use these standard wireless communication modeling expressions and treat them as our link-level surrogates for our analysis. In this problem, the downlink is disregarded because its impact is negligible. It only involves a JSON file being transmitted back from the edge server to the user, having minimal influence on timing and processing computations. Note that the Scheduling problem here is one of optimally sharing the bandwidth to hit deadlines while also allowing batching to be more advantageous on the compute side.

2.4 Batched-OAS Literatures

The Order-Acceptance and Scheduling (OAS) problem decides jointly which jobs to accept and how to sequence them on a machine under its capacity and delivery/time constraints to maximize profit/completions. There is foundational work that formalized MILP models and heuristics for the OAS on single-machine under generalizations like release dates, deadlines, sequence-dependent setups, and tardiness penalties [14, 30]. There have also been OAS variants that couple batching and delivery decisions or even logistics coupling [13, 31].

Instead of the manufacturing-centric OAS literature, we are in the setting of edge inference: (i) jobs arrive asynchronously with individual per-request deadlines; (ii) "processing time" is the sum of rate-dependent upload and batch-dependent inference delay; (iii) batches are run on a GPU with limited parallel streams. We thus use a batched OAS (JBAS-style) formulation with closed-form batch-time updates (Chapter 3) and then replace the original dual-update step with an EDF-based greedy insertion that checks feasibility against upload and compute deadlines. This ties the algorithm to our empirical $d(n)$ model in Chapter 4.

2.5 Framework Backends

Modern CV stacks come with wide heterogeneity in run-time characteristics. PyTorch's imperative API and eager execution programming model support rapid prototyping and flexible design, which has led to its large adoption in research and deployment in practice [19]. For instance, MMDetection is a popular modular

toolbox in object detection that spans model families and inference utilities which lets users and researchers try out different backbones and model components easily to speed up the research/development process [32]. NVIDIA TensorRT takes trained models and converts them into optimized inference engines via a number of techniques including: layer/tactic selection, kernel fusion, precision calibration, etc, all of which are key to reducing latency. Production-grade server backends such as Triton are meant to support the dynamic batching and multiple concurrently running model instances at a time to utilize a device as well as its resources best [33]. These optimizations will have an impact on the shape of the batch-delay curve as well, which is a key figure of merit for inference systems. In our work, we have observed that TensorRT does a great job of minimizing per-batch overheads to bring out the useful batch size regime to about 8 to 16. It also truly enables concurrency by allowing multiple instances of the same model to be launched (Chapter 4). These are reasons behind us fitting an empirical $d(n)$ in Chapter 3 vs an idealized analytical kernel-time model. The empirical model is required to capture all the effects of the optimizations, PyTorch and TensorRT runtime characteristics, etc.

2.6 Gap Analysis

Individual components of the above problem have been studied previously, but a holistic solution is missing. Systems like Clipper[23], INFaaS[25], Proteus[26], and BCEdge[16] have tried to adapt batch size (concurrency in some cases) to hit SLOs. Research into micro-batching has provided important takeaways of latency-aware adaptation for GPUs in particular. The OAS literature has worked on the order of acceptance and sequencing jobs on abstract computational resources to minimize a specific objective.

The missing part, however, is a formulation and algorithm that couples along multiple axes: that of the radio-aware upload, mathematically formulated as $r_k = W \log_2(1 + \gamma_k)$, with bandwidth sharing and one that also describes batch-aware computation where simultaneously consider the batch size and the number of parallel streams, fit from state-of-the-art backend systems, while also giving importance to measure the CPU and GPU behavior through detailed usage profiling to understand their impact on the model. The formulation needs to handle the multiple dimensions of the problem at once in a tractable way. This includes coupling admission control, batching, and parallel stream management to make decisions within milliseconds at a scale representative of production environments. Bridging this gap between accurate MILP or OAS formulation on one hand and production inference servers on the other will be very important to design efficient systems.

To this end, this thesis contributes by (i) calibrating $d(n)$ on modern stacks (PyTorch vs. TensorRT over Ultralytics/MMDetection), (ii) embedding those measurements into batched-OAS model with closed-form timing, and (iii) a Greedy-JBAS algorithm that proposes an EDF-based scheduler that achieves high completion rate of tasks within a tractable runtime. Integrating this new algorithm yields practically deployable rules-of-thumb for B and S under network constraints which we experimentally validate in Chapters 3–5.

Chapter 3

Empirical Measurements

3.1 Hardware and Software

Experiments were performed on a dedicated edge node, with an AMD EPYC–class multi-core CPU and an NVIDIA Volta-class GPU (GV100). Limits on GPU memory dictated the maximum possible batch size, setting the size of the experimental operating window. To elicit host-side behavior, we also performed CPU affinity tests by pinning the inference process to a small number of cores.

We instrumented two representative computer-vision inference stacks:

- OpenMMLab / MMDetection (PyTorch).
- Ultralytics YOLO (PyTorch / TensorRT).

In total, we study three running modes:

1. PyTorch eager inference. This serves as the baseline implementation.
2. A TensorRT-compiled engine, taking advantage of kernel fusion, tactic selection, and lower kernel-launch overheads.
3. Single-model, single-GPU with streams. By launching multiple concurrent instances of the same model, we expose GPU concurrency.

The experimental methodology emphasizes reproducibility so that all runs used a fixed random seed, a fixed list of images, and warm-up iterations to stabilize auto-tuning, caches, and memory allocations. Each configuration was repeated at least 10 times to average the variance. We include full metadata for the entire experiment in Appendix B, covering the CUDA, cuDNN, TensorRT, and driver versions.

The switch to TensorRT mode dramatically reduces per-batch overheads relative to PyTorch, and also exposes a source of practical concurrency by supporting multiple independent inference streams on the same GPU. These features feed directly into the empirical delay model in Chapter 3.(Plots in the figure 3.1 show the superiority of TensortRT-based model)

3.2 Datasets and Pre-processing

We use the COCO validation dataset for all performance measurements.[34] To ensure fair comparisons between frameworks and backends, we pick a fixed subset of 1,024 images and use it throughout.

Input pre-processing follows the default routines implemented in each framework. Images are resized, letter-boxed to the model's canonical input size, normalized, reformatted, etc. Pre-processing was harmonized as much as possible across PyTorch and TensorRT modes to ensure comparability.

The timing protocol is as follows:

1. Run a short warm-up phase (not measured) to stabilize autotuning, caches, and memory allocations.
2. For each (batch size n) and (stream count S), process the full 1,024-image list.
3. Repeat each experiment at least 10 times.
4. Collect total wall-clock runtime, per-batch time,per-image time and throughput, and report mean values as well as 95th percentile (p95) statistics.

Another interesting thing we understood, behind selecting the 1024 images, is that we expect that the two aforementioned frameworks use padding in case the number of remaining images in the last batch is not divisible by the batch size. Yet, surprisingly, we found that while padding is a common methodology in most computer vision algorithms, neither the OpenMMLab nor the Ultralytics YOLO perform padding at the last batch, and they process the last batch as is. Moreover, as a fixed number of images, a fixed dataset, combined with warm-ups and repetitions, is essential to reproducible measurement of batch and concurrency effects. It rules out spurious variance in results due to dataset composition.

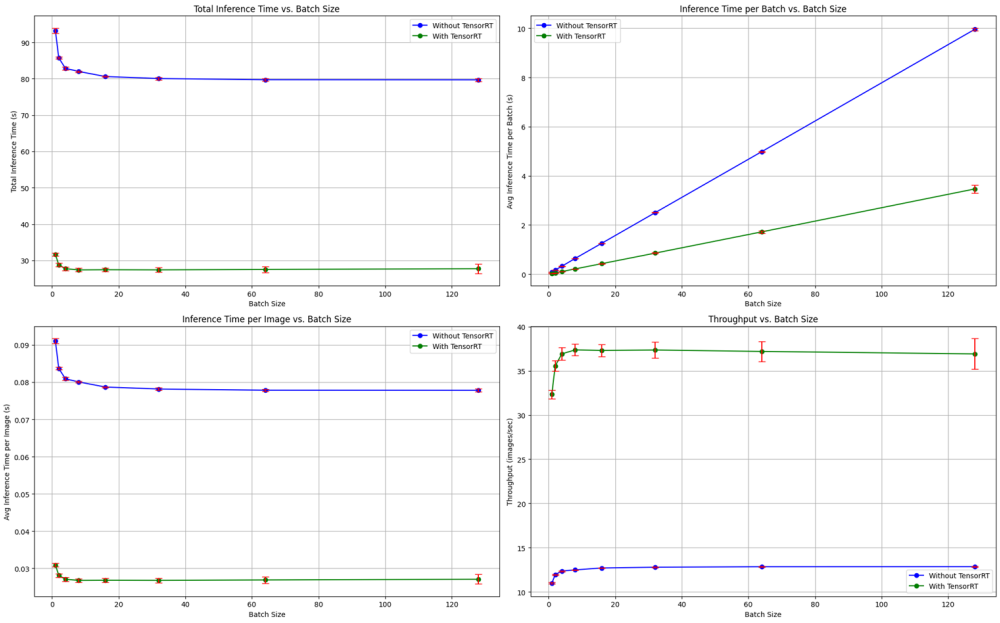

Figure 3.1: Performance evaluation of the two used frameworks: Blue curves are related to the TensorRT-based YOLOv11 model, the other is the YOLO V3 model on OpenMMLab

Figure 3.1 illustrates a baseline YOLO pipeline and its TensorRT-optimized variant under four views of batch–scaling. The *Total Inference Time vs. Batch Size* view (upper-left) shows the TensorRT curve lower and flattening sooner, corresponding to both less end-to-end overhead and more of the GPU being well-utilized as batch size increases. The *Inference Time per Batch* view (upper-right) helps explain why: both models take longer per batch with larger batch sizes, but the baseline scales nearly linearly (due to its weak amortization and kernel/launch overheads), whereas the TensorRT version has a reduced slope (due to the fused kernels, better memory access, and lower-precision execution that mitigate batch-size growth).

The *Inference Time per Image* view (lower-left) depicts the well-known amortization tradeoff: per-image latency decreases with batch size for both models, but is lower and more constant for the TensorRT pipeline, resulting in more predictable latency under load. The *Throughput* view (lower-right) completes the picture: throughput increases with batch size before saturating, with the TensorRT variant saturating at a higher point, corresponding to better GPU saturation and fewer bottlenecks. In practice, these trends mean that the TensorRT-optimized pipeline has more favorable latency–throughput operating points, particularly at moderate-to-large batch sizes where the baseline pipeline saturates with memory and launch overheads.

3.3 Batch-Size Effects

We begin by characterizing the effect of batch size n on per-batch delay $d(n)$. Under the Ultralytics YOLO + TensorRT backend, the empirical latency curve fits the linear model:

$$d(n) = a\,n + b,$$

with fitted parameters $a = 0.011982$ and $b = 0.006082$ in case that only one instance of the model is running. This confirms that GPU kernel fusion and mixed-precision optimizations preserve a nearly constant per-task slope, or fan-in factor a, over the practical range of small to moderate batch sizes.

The practical operating window exhibits two patterns. Throughput grows sharply up to around B, with more modest further gains up to B. Beyond this point, further batching yields diminishing returns, as overheads from memory pressure and host-side orchestration grow relative to GPU compute. In contrast, the PyTorch implementation has higher fixed overhead (larger b) and poorer scaling (larger effective a), consistent with less efficient batching.

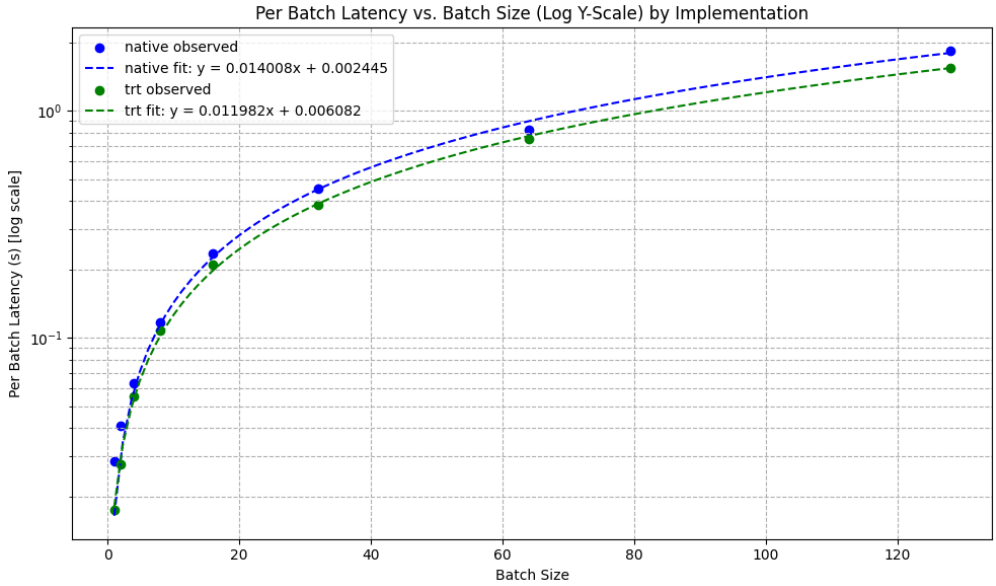

Figure 3.2: Measured per-batch inference latency per task as a function of batch size (logarithmic y–axis), comparing native (blue) and TensorRT (green) implementations and their fitted linear models.

The fitted parameters (a, b) directly parameterize the computation delay model in Chapter 3. They enter as empirical coefficients in the closed-form timing recursions that underlie the Joint Batching and Scheduling (JBAS) framework.

3.4 Concurrency Effects

We next characterize the effect of GPU concurrency by varying the number of parallel streams $S \in \{1, \ldots, 8\}$, where each stream corresponds to an independent model instance running on the same GPU. Under TensorRT, throughput scales nearly linearly with (S) up to about three to four streams, after which a distinct knee is encountered at $S \approx 6$. Beyond this point, throughput gains from adding additional streams are negligible, and in some cases cause a regression in throughput due to memory contention and host-side coordination overhead.

Aggregate throughput peaks around $S = 5$–6, representing the optimal level of concurrency given the underlying GPU's ability to schedule and overlap kernels. However, this concurrency ceiling is sensitive to batch size: as B grows, so does the amount of VRAM used per batch. For this reason, the most effective operating points combined moderate batch sizes ($B = 8$–16) with $S = 5$–6 streams.

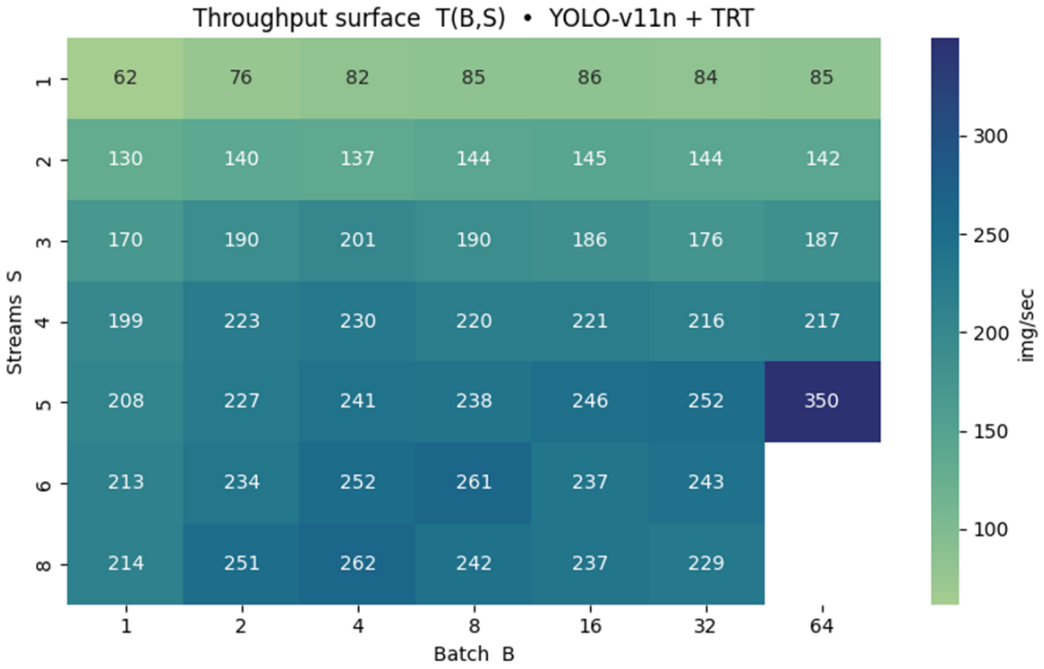

Figure 3.3: Aggregate throughput as a function of stream count S under different batch sizes. Markers indicate the concurrency knee.

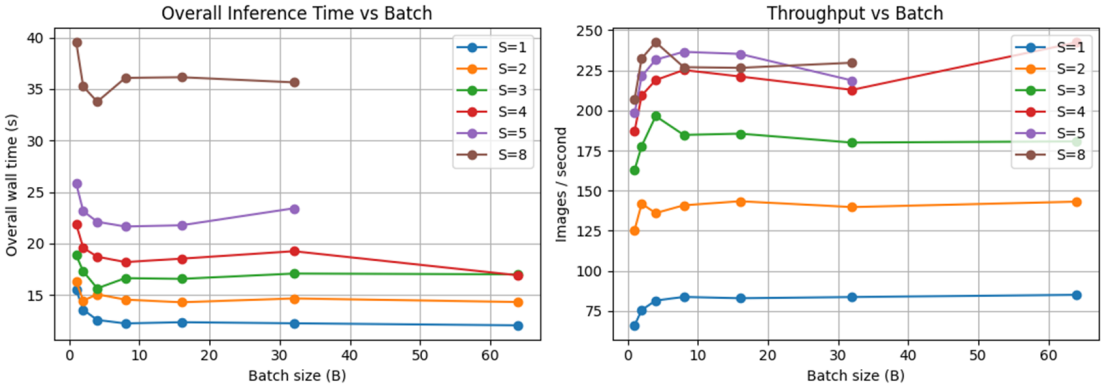

Figure 3.4: Throughput and total infernce time as a function of T(B,S) Using YOLO-v11n + TensorRT

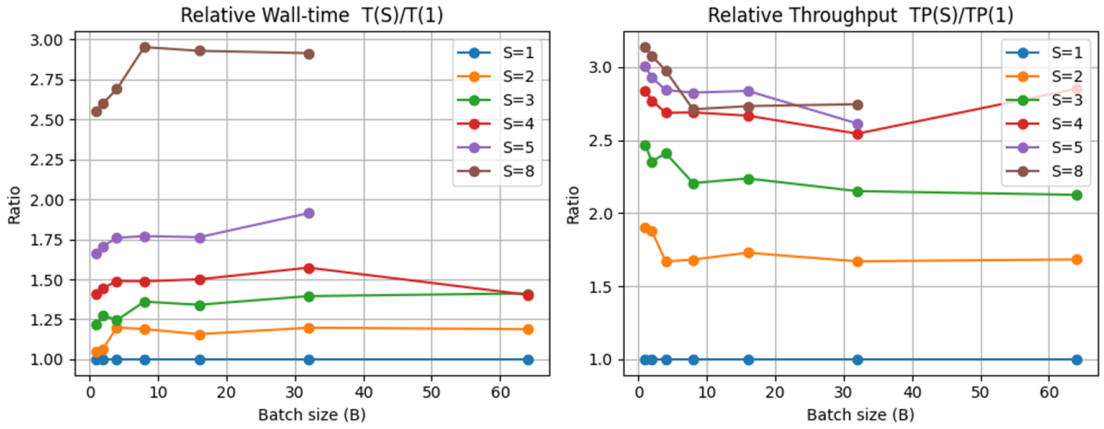

Figure 3.5: Relative throughput and total infernce time as a function of T(B,S) Using YOLO-v11n + TensorRT

The term *relative* in Figure 3.5 means that all curves in the figure are normalized by the single-stream baseline ($S = 1$). In other words, in the left panel (Relative Wall-time), $T(S)$ is normalized by $T(1)$; while in the right panel (Relative Throughput), $TP(S)$ is normalized by $TP(1)$. This way, each curve in the figure reflects the relative improvement of efficiency achieved by adding more streams. The results show that increasing the number of streams improves throughput up to a saturation point, while wall-time grows sublinearly with S. In particular, setting $S = 5$ parallel streams captures roughly 90% of the maximum achievable throughput across all settings. This implies that after 5 streams, there are diminishing returns, so $S = 5$ is a near-optimal choice for balancing utilization and overhead.

3.5 Offset Launches

While we try to measure the GPU Utilizatoin for running multiple concurrent tasks, after a batch completes, there's a gap while the CPU prepares the next batch (or while other overhead tasks complete), because the tasks are synchronized, these gaps often line up, causing simultaneous dips in GPU usage. So, a common strategy to mitigate this effect is introducing an offset for each concurrent task, meaning Task B only begins loading and processing its first batch after Task A has already started or even completed its first batch. This can reduce the periods of low GPU usage between batches.

For instance consider the case where run the model at batch size eqaul 256, so basically we will have four batches pass to the model, therefore we will have four big spikes in GPU usage, one for each batch, Each large spike in GPU utilization corresponds to the model actively processing one batch of size 256. Between these spikes, the GPU usage drops because it's waiting for the CPU to finish preparing/loading the next batch or for other overhead to complete. This creates the "periodic" pattern you're seeing in the plot.

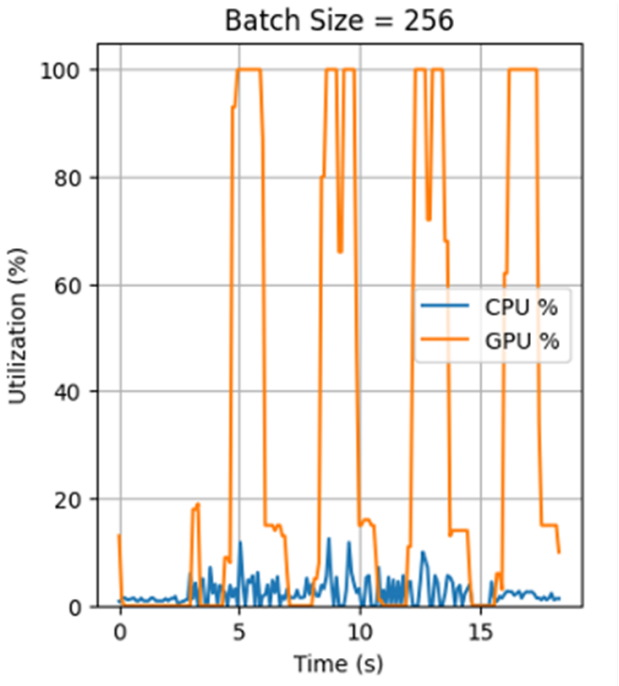

Figure 3.6: GPU profiling for concurrent streams

We calculate the offset duration based on below:

$$\text{Number of patterns} \triangleq \frac{\text{dataset\_size}}{\text{batch size}} \tag{3.1}$$

$$\text{Pattern Duration} \triangleq \frac{\text{Total Inference Time}}{\text{Number of patterns}} \tag{3.2}$$

$$\text{Offset Duration} \triangleq \frac{\text{Pattern Duration}}{\text{Number of patterns}} \tag{3.3}$$

Then, to further understand GPU utilization, we compare synchronized and staggered (offset) launch strategies across multiple streams. In the synchronized case, all streams begin at once, while in the offset case, `model.predict` calls are staggered by a fixed delay intended to smooth GPU load. After correcting the offset formula (offset = batch duration / number of streams), experimental results revealed only marginal improvements.

Measured gains from staggering were modest at best, on the order of 1–3% once S. GPU utilization traces confirm that offsets reduce idle gaps slightly, producing a smoother load pattern. These benefits did not translate into material throughput improvements in the presence of a concurrency knee at or above S.

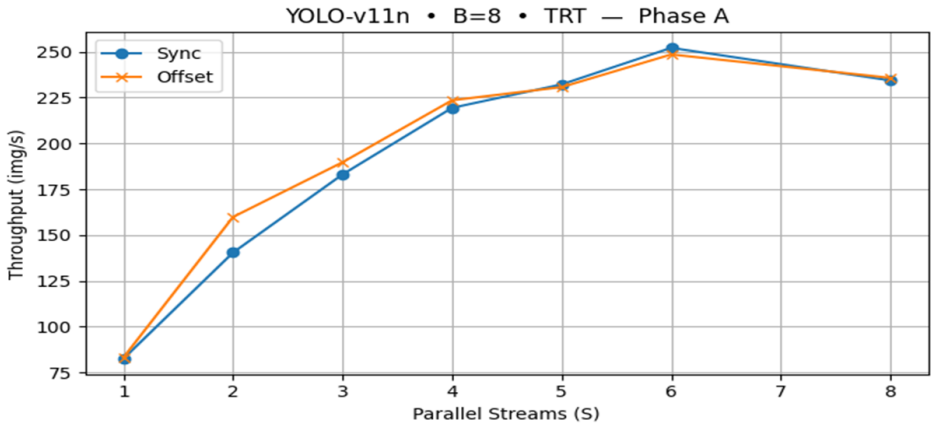

Figure 3.7: Throughput comparison of synchronized vs. staggered offsets at $S = 3,5,6$. Inset: representative GPU utilization traces for both models.

Implication. Offset scheduling has only a marginal impact on throughput in the studied regime. As a result, we do not model offsets in Chapter 4 formulation, and they can be viewed as an optional deployment-level refinement rather than a core optimization.

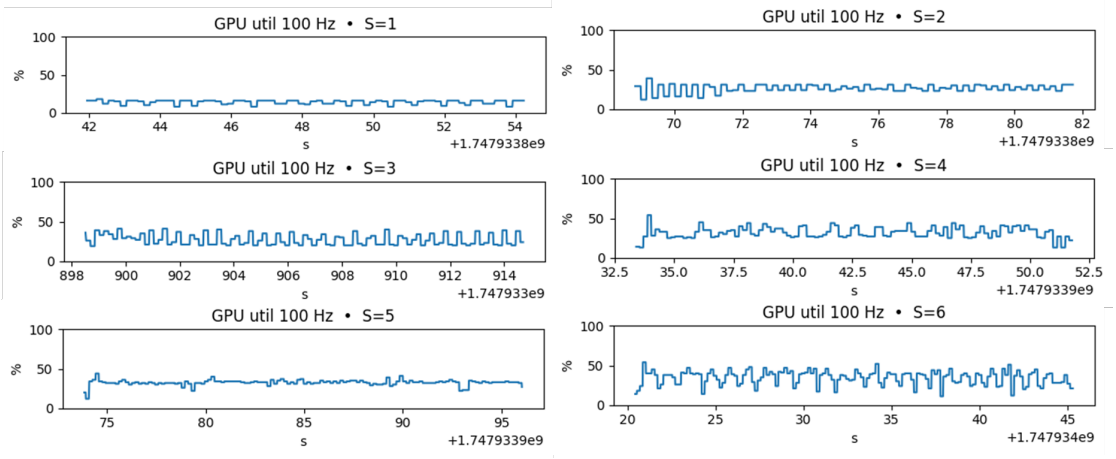

Figure 3.8: GPU usage comparison at $S = 1,2,3,4,5,6$ at 100Hz frequency for B = 8 TensorRT engine model.

Streams S	**1**	**2**	**3**	**4**	**5**	**6**	**8**
GPU util (mean)	$\sim 5\%$	$\sim 15\%$	$\sim 22\%$	$\sim 28\%$	$\sim 35\%$	$\sim 38\%$	$\sim 38\%$
GPU util (peaks)	$< 15\%$	$< 35\%$	$< 45\%$	$< 55\%$	$< 60\%$	$< 65\%$	$< 65\%$

Table 3.1: GPU utilization (mean and peaks) as a function of the number of streams S.

Table 3.1 shows that even at the best operating point, GPU utilization remains below 70%, indicating substantial hardware headroom; end-to-end latency is dominated by per-batch host work and/or TensorRT overhead.

3.6 CPU/GPU Profiling

To understand the reason behind throughput saturation before full GPU utilization, we performed fine-grained CPU and GPU profiling.

Per-core CPU usage: Looking at the profile of CPU usage by limiting the number of CPU used cores (either by using *taskset* command on Linux or by using the Python OS library, *os.sched_setaffinity*), we find that only one or two cores are significantly loaded. For instance, when running under a 10-core affinity mask (cores 0–9), typically one core would have sustained ∼60–75% utilization while another was ∼20–35%; all other cores are near-idle. Shifting the affinity to another set of cores (e.g., cores 40–43) simply relocates the bottleneck without actually engaging

any new cores. This suggests that the Python host loop, including pre-processing, post-processing, and all orchestration around the `model.predict` calls, effectively operates as a single-threaded or dual-threaded workload. The Global Interpreter Lock (GIL) in Python aggravates this condition.

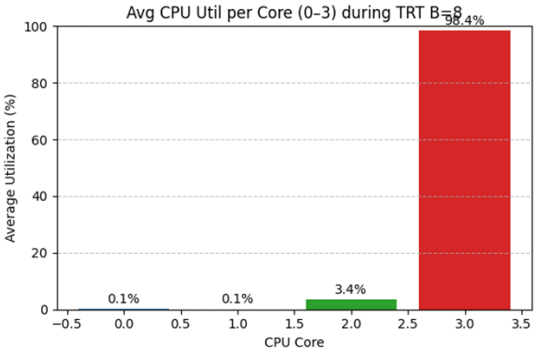

Figure 3.9: Per-core CPU utilization under affinity mask cores 0–3.

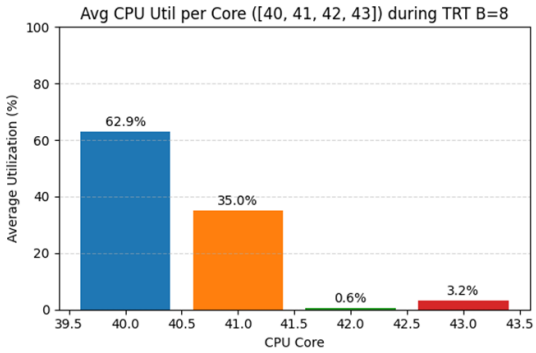

Figure 3.10: Per-core CPU utilization under affinity mask cores 40–43.

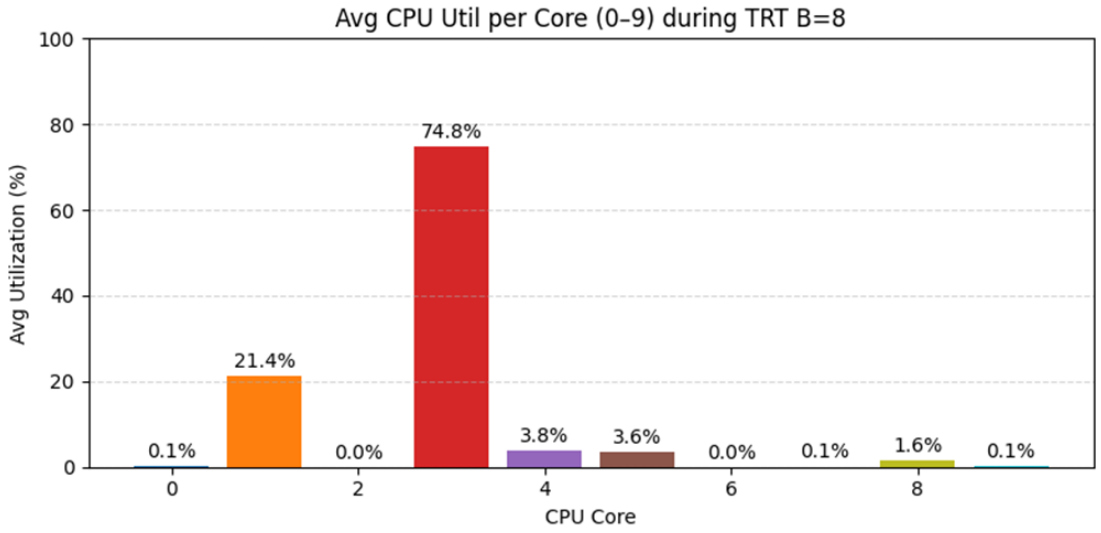

Figure 3.11: Per-core CPU utilization under affinity mask cores 0–9.

GPU utilization: Average GPU utilization, even at the concurrency knee ($S =$ 5–6), is rare to exceed 70%. GPU utilization traces show a roughly-periodic sawtooth pattern, with bursts of high activity interrupted by dips when the GPU stalls while waiting for the host to prepare new data or to finish data transfer. This finding supports the conjecture that the GPU is not in fact fully saturated, but

rather throughput is capped by a host-side bottleneck.

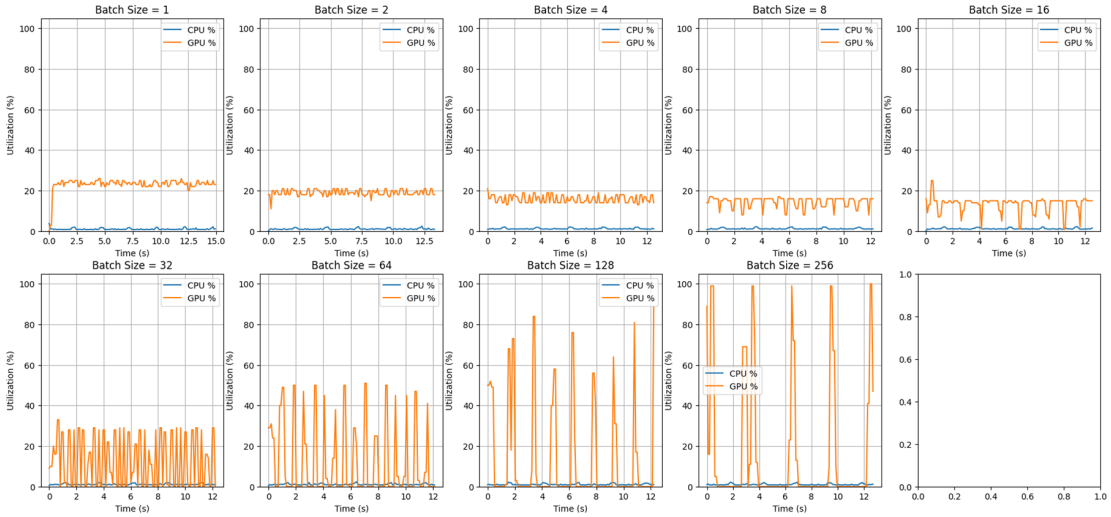

Figure 3.12: GPU utilization under different affinity masks; GPU utilization traces at $S = 3{,}5{,}6$.

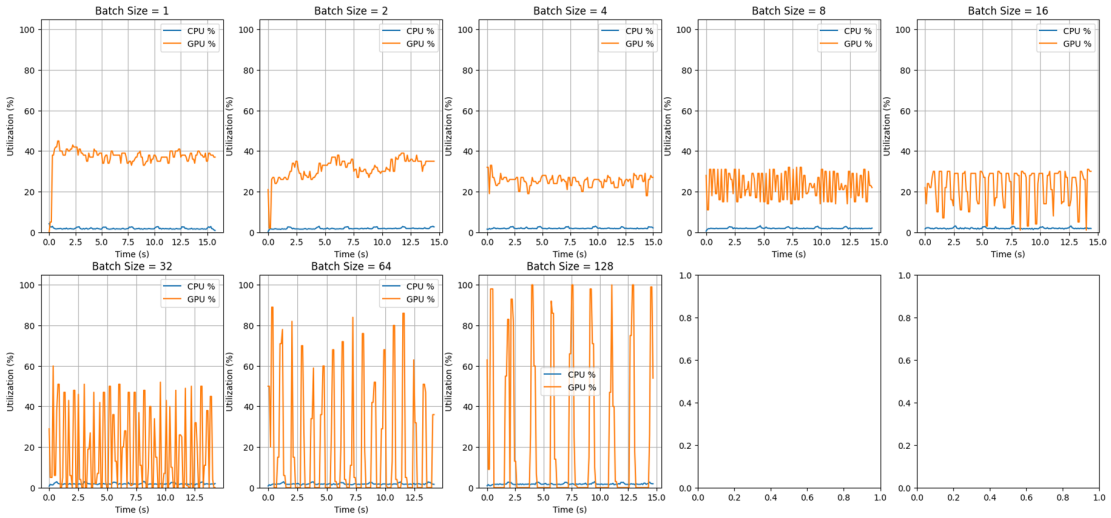

Figure 3.13: GPU utilization under different affinity masks; GPU utilization traces at $S = 3{,}5{,}6$.

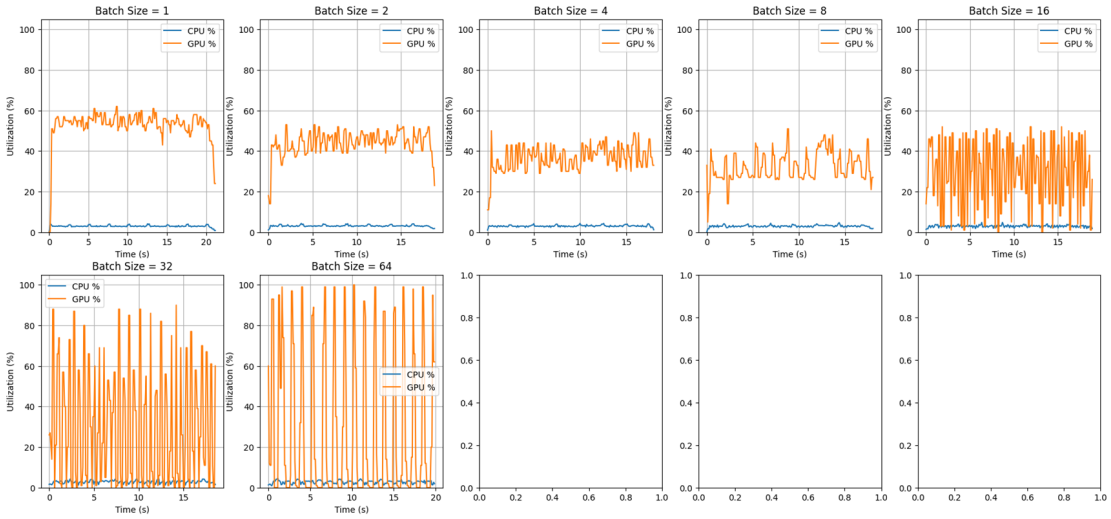

Figure 3.14: GPU utilization under different affinity masks; GPU utilization traces at $S = 3{,}5{,}6$.

While addressing the negative impact caused by CPU performance is beyond the scope of this thesis, exploring potential solutions remains valuable. It is worthwhile to investigate remedies for this situation, such as:

- Offload pre-/post-processing into C++ extensions or a multi-process pipeline to avoid the Python GIL.
- Use a framework such as PyTorch DataLoader with multiple workers and `pin_memory` enabled to parallelize data loading.
- Exploit async host-to-device transfer and prefetching to overlap computation and data staging.

The limiting factor beyond $S = 5$–6 is not GPU compute, but rather host-side bottlenecks related to data preparation and orchestration. This supports the treatment of S as a fixed, exogenous parameter in Chapter 3's problem formulation. Optimization is instead focused on admission control and batching strategies.

3.7 Batch Delay Formulation

In sections 3.3 and 3.4, we saw the effect of batch size and concurrency, respectively, on the model's total inference time. For instance, figure 3.2 shows the regression model fitted on the experimental measurement data in the case that we only have one concurrent stream. In the following, we try to find a general term for batch

delay to support testing the scenarios with a higher number of concurrent streams in our final algorithm.

3.7.1 Dataset and Metrics

In figure 3.4, we profiled a GPU inference stack with varying batch sizes and concurrency (CUDA streams). We collected tuples $(B, S, \texttt{total\_s}, \texttt{img\_per\_s})$ where B is the batch size, S the number of concurrent instances, `total_s` the wall-clock time taken to process all images in the workload, and `img_per_s` the throughput (images/s).
Now, For each (B, S) we derive the *per-batch* processing time as:

$$d_S(B) \;=\; \frac{B}{\texttt{img\_per\_s}(B, S)}. \tag{3.4}$$

When the total number of processed images N_{imgs} is known, (3.4) is equivalent to

$$d_S(B) \;=\; \frac{\texttt{total\_s}(B, S) \cdot B}{N_{\text{imgs}}}. \tag{3.5}$$

In practice, we also verify consistency via $N_{\text{est}} = \text{round}(\texttt{total\_s} \cdot \texttt{img\_per\_s})$.

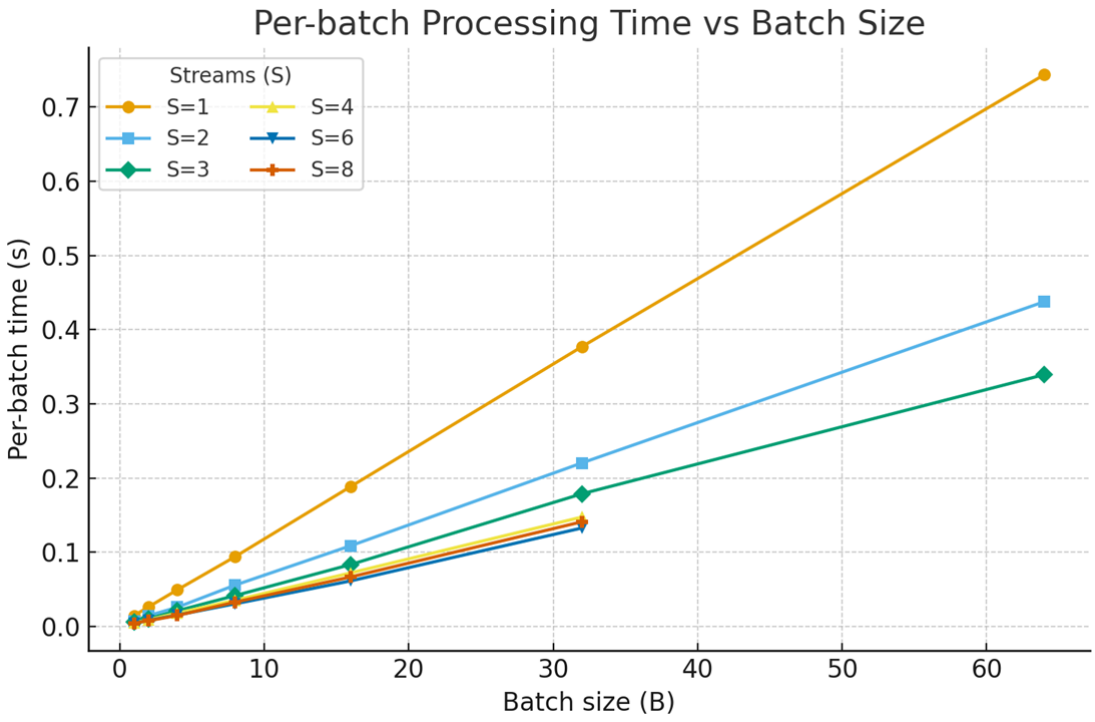

Figure 3.15: Per-batch delay time as a multivariate function of (B,S)

3.7.2 Empirical Curves

For each fixed S, we plot $d_S(B)$ vs. B and fit the affine model

$$d_S(B) \;\approx\; a_S\, B + b_S, \tag{3.6}$$

where a_S captures the *incremental* cost per image within a batch, and b_S aggregates per-batch fixed overheads (launch, sync, H2D/D2H setup).

Table 3.2 reports the least-squares fits (one line per S). The fits are near-perfect ($R^2 \approx 0.999$), supporting the standard affine delay model.

S (streams)	a_S (s/task)	b_S (s)	R^2
1	0.011590	0.003319	0.999967
2	0.006840	0.000492	0.999956
3	0.005344	0.000479	0.999195
4	0.004601	-0.000233	0.999795
6	0.004125	-0.000975	0.998303
8	0.004426	-0.001685	0.999118

Table 3.2: Per-stream affine fits $d_S(B) \approx a_S B + b_S$.

The per-stream slopes a_S decrease as S increases (sublinear speedup; contention and memory pressure), so a multiplicative form $S(\,aB + b\,)$ is *not* supported by the data. Instead, we adopt a saturating inverse scaling:

$$\boxed{d_S(B) \;=\; \frac{aB + b}{1 + \gamma(S-1)}} \tag{3.7}$$

where (a, b) are single-stream coefficients and $\gamma \in (0,1]$ quantifies concurrency efficiency. Fitting (3.7) jointly over all measured (B, S) yields

$$a \approx 0.01136 \text{ s/image}, \qquad b \approx 0.00502 \text{ s}, \qquad \gamma \approx 0.554. \tag{3.8}$$

Equation (3.7) enables extrapolation to unmeasured (S, B) pairs (useful where VRAM limits prevented experiments). Example predictions have been shown in the table 3.3:

	S=1	S=4	S=8	S=16
B=8	0.096	0.022	0.012	0.007
B=16	0.187	0.042	0.023	0.013
B=32	0.369	0.082	0.044	0.024
B=64	0.733	0.162	0.087	0.047

Table 3.3: Predicted per-batch time $d_S(B)$ (seconds) from (3.7) with parameters of equation (3.8).

From now, due to the flexibility of the interpolated model to compute the d(n), the so-called per batch delay, we use its formula to calculate the a & b parameters in our designed algorithm.

Chapter 4

System Model & Problem Formulation

The chapter formalizes the edge inference scheduling problem that we will solve in Chapters 5 and 6. We start by defining the entities and notation, and then the communication and computation models. In Section 4.4, we state the decisions/-constraints and objective ladder from (P1) to (P4). By default, we assume a single edge node with a single GPU. The concurrency (number of parallel streams S) is a fixed parameter that also affects the values of the delay fit coefficients.

4.1 Entities & Notation

We operate on a continuous time axis (units of seconds). The main sets, variables, and parameters are collected in Table 4.1.

- **Tasks/requests.** Index tasks by $k \in \{1, \dots, K\}$. Each task has an arrival time $T_k^{(a)} \in \mathbb{R}_{\geq 0}$, a deadline $T_k^{(d)} > T_k^{(a)}$, and an upload payload (feature size) ℓ_k (bits).
- **Radio state.** Each task has an associated SNR (linear scale) or channel gain γ_k used to determine its uplink rate r_k (bit/s) given an allocated bandwidth share (defined in §3.2).
- **Admission & assignment.** $Y_k \in \{0,1\}$ is a binary admission variable. If admitted, a task is assigned to exactly one batch $n \in \{1, \dots, N\}$ via indicator $\pi_{k,n} \in \{0,1\}$ with $\sum_n \pi_{k,n} \leq 1$. The batch size is $\pi_n \triangleq \sum_k \pi_{k,n}$, with a VRAM-driven cap $\pi_{\max}$ (cf. Chapter 4).
- **Batch timing.** Batches execute sequentially in the base model: t_n is the start time of batch n, and $d(\pi_n)$ its service time (seconds). We will use the empirical

linear model extracted from experimental measurements in Chapter 3:

$$d(n) = a\,n + b, \qquad a > 0,\ b > 0,\ n \in \{0,1,\ldots,\pi_{\max}\}.$$

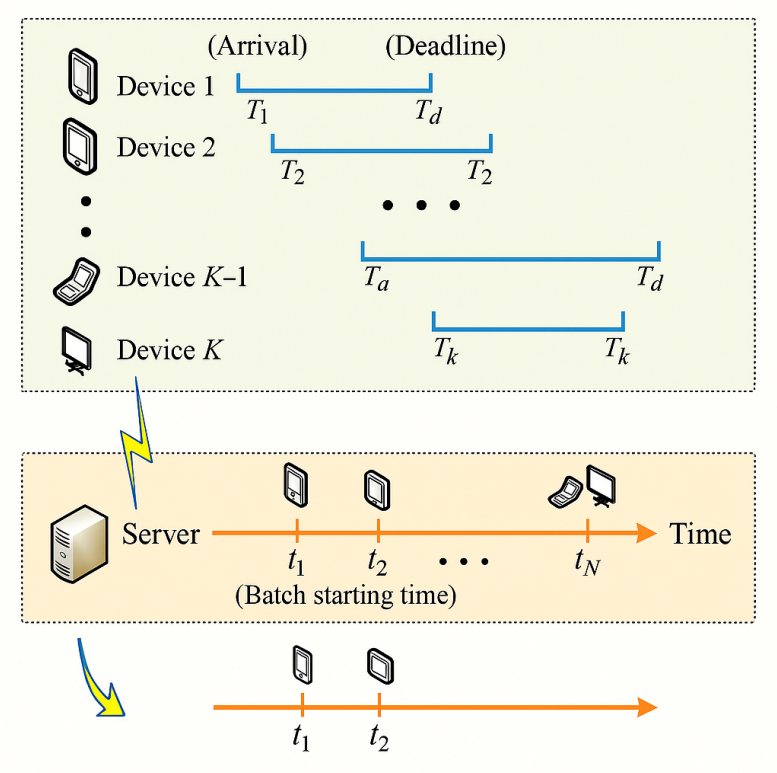

Figure 4.1: System model for asynchronous task arrivals.

The fitted coefficients (a, b) correspond to the TensorRT version of YOLOv11n at the chosen S (we denote them $(a, b)_{\text{TRT,YOLOv11n},S}$).

- **Concurrency.** The number of GPU streams/instances is $S \in \mathbb{Z}_{\geq 1}$. In the base problem we fix S (chosen near the knee $S \approx 5$-6, Chapter 4) and fold its effect into the measured (a, b). Extensions with explicit multi-server timing appear in the Appendix.
- **Bandwidth.** The total uplink bandwidth available to the edge is W (Hz). Within any upload window (see §4.2), users share W according to a specified policy (we adopt FDMA equal share by default in simulations).
- **Slack & priority.** Define slack $\sigma_k \triangleq T_k^{(d)} - T_k^{(a)}$. Each task has a *priority/utility weight* $\rho_k \geq 0$ (dimensionless; default 1) used in the objective to value completed tasks. Check the appendix B chapter for the GUI for more information regarding the priority values.

• **Safety & numerics.** We use a small $\epsilon > 0$ to avoid degenerate divisions in upload windows and an optional safety inflation $\delta \in [0,0.1]$ on $d(\cdot)$ for robustness.

Table 4.1: Notation & units (Chapter 3).

Symbol	Meaning	Units/Notes
K	Number of tasks in horizon	–
$T_k^{(a)}$, $T_k^{(d)}$	Arrival, deadline of task k	s
ℓ_k	Upload payload size	bits (convert bytes→bits uniformly)
γ_k	SNR (linear)	optionally dB in input; convert
W, w_k	Total and per-task bandwidth share	Hz
r_k	Uplink rate of task k	bit/s
Y_k	Admission variable	$\{0,1\}$
$\pi_{k,n}$, π_n	Assignment of k to batch n; batch size	$\{0,1\}$; integer
$\pi_{\max}$	Max batch size (VRAM)	integer
t_n, $d(\pi_n)$	Start time and service time of batch n	s
a, b	Delay fit coefficients (TRT+YOLOv11n, S fixed)	s/task, s
S	Parallel streams/instances	integer
ρ_k	Task priority/utility weight	dimensionless; default 1
ϵ, δ	Numerical slack; safety margin on d	s; fraction

To avoid a clash with batch size B used elsewhere, we denote bandwidth by W (Hz) and batch size by the batch cardinality π_n (or by B only in prose).

4.2 Communication Model

We adopt a standard uplink model: during its upload window $\left[T_k^{(a)}, t_n\right)$, a task k that is assigned to batch n is allocated a bandwidth share $w_k \geq 0$ such that $\sum_{i \in \mathcal{U}_n} w_i \leq W$, where $\mathcal{U}_n$ is the set of tasks uploading for batch n. The instantaneous rate (bit/s) is

$$r_k \;=\; w_k \log_2\Big(1+\gamma_k\Big), \tag{4.1}$$

with γ_k the (linear) SNR for task k; (4.1) is the AWGN surrogate used throughout wireless modeling. [28, 29]

The upload time required to transmit ℓ_k bits at rate r_k is

$$t_k^{(o)} \;=\; \frac{\ell_k}{r_k} \;=\; \frac{\ell_k}{w_k \log_2(1+\gamma_k)}. \tag{4.2}$$

Feasibility for assignment $\pi_{k,n} = 1$ requires that the upload finish before the batch starts:

$$T_k^{(a)} + t_k^{(o)} \;\leq\; t_n, \qquad t_n > T_k^{(a)} + \epsilon. \tag{4.3}$$

Within each window $[T_i^{(a)}, t_n)$, bandwidth shares must obey the capacity budget

$$\sum_{i \in \mathcal{U}_n} w_i \;\leq\; W, \qquad w_i \geq 0. \tag{4.4}$$

Sharing policy. In our simulations and experiments we use FDMA equal share: at any time within $[T_i^{(a)}, t_n)$, active uploaders share W equally (i.e., $w_i = W/|\mathcal{U}_n|$). This choice matches the implementation used to produce the results in Chapters 3 and simplifies feasibility checks. Other policies (proportional fair, TDMA) can be substituted without changing the formulation.

Optional overhead. A small constant uplink setup latency c_0 (e.g., link establishment) may be included by replacing (4.2) with $t_k^{(o)} = c_0 + \ell_k / r_k$. We set $c_0 = 0$ in our experiments for simplicity.

Units and guards. We enforce $t_k^{(o)} > 0$, $0 \leq w_k \leq W$, and convert any dB values of SNR input to linear γ_k before applying 4.1. When 4.3 cannot be satisfied for any batch n, the task must be rejected ($Y_k = 0$).

4.3 Computation Model

We model the GPU service time for a batch of size n by the empirical linear fit obtained in Chapter 4:

$$d(n) \;=\; a\,n + b, \qquad a > 0,\; b > 0,\; n \in \{0,1,\ldots,\pi_{\max}\} \tag{4.5}$$

where $\pi_{\max}$ is a VRAM-driven maximum batch size (from Chapter 4), and (a, b) are the coefficients of the regression-fitted model (based on TensorRT + YOLOv11n) measured at the chosen stream count S (fixed near the concurrency knee). In the initial setup, Chapter 4 reports representative values $a \approx 0.011982\,\text{s/task}$ and $b \approx 0.006082\,\text{s}$.

Sequential execution (base model). We treat the GPU as a single server in the base formulation: batches execute one after another, so start times obey the recursion

$$t_{n+1} \;=\; t_n \;+\; d(\pi_n). \tag{4.6}$$

This is the specialization (single server case) of the closed-form batch time update proven in Chapter 3/Appendix A; we keep the general theorem there and use (4.5) as the working relation here. (A multi-server extension with explicit $S > 1$ appears in Appendix A.)

Streams and offsets. We fix the number of parallel streams S (chosen near 5–6 from Chapter 4) and fold its effect into (a, b); this keeps the problem ladder (P1→P4) unchanged. Staggered offset launches were measured to provide only $\approx$ 1-3% benefit at $S \geq 4$, so we ignore offsets in the base model (Appendix C shows traces).

4.4 Problem Formulation

Inspired by [13, 8, 7], we first state the decision variables and the hard constraints of the problem. Defining $\pi_{k,n} \in \{0,1\}$ (assign task k to batch n), batch size $\pi_n = \sum_k \pi_{k,n}$, and batch start times $\{t_n\}$. Building on the model above, we aim to *maximize the weighted number of completed tasks* by choosing batch start times, assignments, and the number of batches:

$$\{t_n\}_{n=1}^{N}, \quad \{\pi_{k,n}\}, \quad N \in \mathbb{Z}^+,\, N \leq K.$$

Let $\pi_n \triangleq \sum_{k=1}^{K} \pi_{k,n}$ and $d(n) = an + b$ (measured at fixed S near the concurrency knee). The total uplink bandwidth is W (Hz). For a task k assigned to batch n, the *required bandwidth* to finish its upload by t_n is

$$b_k(n, t_n) \;\triangleq\; \frac{\ell_k}{\Big(t_n - T_k^{(a)}\Big)\,\log_2\!\Big(1+\gamma_k\Big)}, \qquad t_n > T_k^{(a)} + \epsilon. \tag{4.7}$$

cf. §4.2.

$$\text{(P1)} \qquad \max_{\{t_n\},\,\{\pi_{k,n}\},\,N} \; \sum_{k=1}^{K} \sum_{n=1}^{N} \rho_k\, \pi_{k,n}. \tag{P1}$$

Subject to:

(i) Causality (upload finishes before batch starts). For any assignment $\pi_{k,n} = 1$,

$$t_n \;>\; T_k^{(a)} + \epsilon. \tag{C1}$$

(ii) Deadline (compute finishes before task deadline).

$$t_n + d(\pi_n) \;\le\; T_k^{(d)}, \qquad \forall k \text{ with } \pi_{k,n} = 1. \tag{C2}$$

(iii) Sequential batches (single server).

$$t_n + d(\pi_n) \;\le\; t_{n+1}, \qquad n = 1, \ldots, N-1. \tag{C3}$$

(iv) Bandwidth feasibility (FDMA surrogate). For any batch n,

$$\sum_{i:\,\pi_{i,n}=1} b_i(n, t_n) \;\le\; W, \qquad t_n > T_i^{(a)} + \epsilon \;\; \forall i : \pi_{i,n} = 1. \tag{C4}$$

(v) Single assignment & domain.

$$\sum_{n=1}^{N} \pi_{k,n} \le 1, \quad \pi_{k,n} \in \{0{,}1\}, \quad 0 \le \pi_n \le \pi_{\max}, \quad N \le K. \tag{C5}$$

(The strict ordering $t_1 < \cdots < t_N$ is implied by (iii) with $d(\pi_n) > 0$ and therefore omitted.) P1 is a Mixed-Integer Linear Programming (MILP) problem and, while there are several heuristic algorithms for the simple OAS problem [14], we pursue the convex-relaxation approach introduced in [7].

From (P1) to (P2): Decoupling Deadlines

Introducing the constant value $\Xi \triangleq \max_k T_k^{(d)} + d(K) = \max_k T_k^{(d)} + aK + b$. Then:

$$t_n + d(\pi_n) \;\le\; T_k^{(d)} + (1 - \pi_{k,n})\,\Xi, \quad \forall k, n,$$

which is tight when $\pi_{k,n} = 1$ and vacuous when $\pi_{k,n} = 0$. Thus:

$$\begin{aligned}
(\text{P2}): \quad \max_{\{t_n\},\{\pi_{k,n}\},N} \; & \sum_{k=1}^{K}\sum_{n=1}^{N} \rho_k\, \pi_{k,n} \\
\text{s.t. } & t_n > T_k^{(a)} + \epsilon \;\text{ if } \pi_{k,n} = 1, \;\forall k, n, \\
& t_n + d(\pi_n) \le T_k^{(d)} + (1 - \pi_{k,n})\Xi, \;\forall k, n, \\
& t_n + d(\pi_n) \le t_{n+1}, \; n = 1, \ldots, N-1, \\
& \sum_{i:\pi_{i,n}=1} b_i(n, t_n) \le W, \;\forall n, \\
& \sum_{n=1}^{N} \pi_{k,n} \le 1, \;\; \pi_{k,n} \in \{0{,}1\}, \\
& 0 \le \pi_n \le \pi_{\max}, \;\; N \le K.
\end{aligned}$$

From (P2) to (P3): Fixing the Number of Batches

By allowing *empty* batches, we can fix $N = K$ without loss of optimality:

$$
\begin{aligned}
(\mathrm{P3}): \quad \max_{\{t_n\},\{\pi_{k,n}\}} \ & \sum_{k=1}^{K}\sum_{n=1}^{K} \rho_k\, \pi_{k,n} \\
\text{s.t. } & t_n > T_k^{(a)} + \epsilon \ \text{ if } \pi_{k,n} = 1,\ \forall k, n, \\
& t_n + d(\pi_n) \le T_k^{(d)} + (1 - \pi_{k,n})\Xi,\ \forall k, n, \\
& t_n + d(\pi_n) \le t_{n+1},\ n = 1, \ldots, K-1, \\
& \sum_{i:\pi_{i,n}=1} b_i(n, t_n) \le W,\ \forall n, \\
& \sum_{n=1}^{K} \pi_{k,n} \le 1, \ \ \pi_{k,n} \in \{0{,}1\}, \\
& 0 \le \pi_n \le \pi_{\max}.
\end{aligned}
$$

From (P3) to (P4): Smoothing the Non-Smooth Delay

The batch delay

$$
d_n(\pi_n) = \begin{cases} a\,\pi_n + b, & \pi_n > 0, \\ 0, & \pi_n = 0, \end{cases} \tag{4.8}
$$

is non-smooth at $\pi_n = 0$. Using the log surrogate for the indicator (ℓ_0) with parameter $\delta > 0$,

$$
\|\pi_n\|_0 \approx \frac{\ln(1 + \delta^{-1}\pi_n)}{\ln(1 + \delta^{-1})}, \tag{4.9}
$$

and the first-order linearization around $\pi_n^{(r)}$ yields

$$
\|\pi_n\|_0 \le \theta_n^{(r)}\,\pi_n + \psi_n^{(r)}, \quad \tilde{d}_n^{(r)}(\pi_n) = \left(a + b\,\theta_n^{(r)}\right)\pi_n + b\,\psi_n^{(r)}. \tag{4.10}
$$

Relax $\pi_{k,n} \in \{0{,}1\}$ to $0 \le \pi_{k,n} \le 1$ and replace $d_n(\pi_n)$ by $\tilde{d}_n^{(r)}(\pi_n)$ to obtain:

$$
\begin{aligned}
(\mathrm{P4}): \quad & \max_{\{t_n\},\{\pi_{k,n}\}} \sum_{k=1}^{K}\sum_{n=1}^{K} \rho_k\, \pi_{k,n} \\
& \text{s.t.} \quad \pi_{k,n} = 0 \text{ or } t_n > T_k^{(a)} + \epsilon,\ \forall k, n, \\
& \qquad t_n + \tilde{d}_n^{(r)}(\pi_n) \le T_k^{(d)} + (1-\pi_{k,n})\,\Xi,\ \forall k, n, \\
& \qquad t_n + \tilde{d}_n^{(r)}(\pi_n) \le t_{n+1},\ n = 1, \ldots, K-1, \\
& \qquad \sum_{i=1}^{K} \pi_{i,n} \frac{\ell_i}{\left(t_n - T_i^{(a)}\right)\log_2(1+\gamma_i)} \le W,\ \forall n, \\
& \qquad \sum_{n=1}^{K} \pi_{k,n} \le 1, \quad 0 \le \pi_{k,n} \le 1.
\end{aligned}
$$

We explicitly guard denominators by requiring $t_n > T_i^{(a)} + \epsilon$ whenever $\pi_{i,n} > 0$.

Relaxation & Alternating-Optimization (P4)

At this stage, we have transformed the hard MINLP (P3) into a *convex* problem by (i) approximating the non-smooth delay $d_n(\pi_n)$ with its affine surrogate $\tilde{d}_n^{(r)}(\pi_n)$ (Section 4.4) and (ii) relaxing $\pi_{k,n} \in \{0,1\}$ to $0 \le \pi_{k,n} \le 1$. The resulting convex program is (P4) above.

Alternating-Optimization. Problem (P4) is convex but still high-dimensional in $\{\pi_{k,n}\}$ and $\{t_n\}$. We therefore split it into two tractable subproblems and iterate:

(i) Task–Batch Association (fix t, optimize π): With $\{t_n\}$ and surrogate weights $\{\theta^{(r)}, \psi^{(r)}\}$ fixed, (P4) reduces to a linear program in π with simple box and per-task sum constraints (dualizing deadlines, sequentiality, and bandwidth). The LP naturally prioritizes tasks via the objective coefficients ρ_k, choosing, for each task k, the batch n that maximizes the corresponding net gain.

(ii) Batch Starting Times (fix π, optimize t): With $\{\pi_{k,n}\}$ fixed, feasibility under causality, deadlines and sequentiality yields the closed-form backward recursion

$$
t_n^* = \begin{cases} \min\{\chi_n, t_{n+1}^*\} - d_n(\pi_n), & \pi_n > 0, \\ t_{n+1}^*, & \pi_n = 0, \end{cases} \qquad \chi_n = \min_{k:\pi_{k,n}=1} T_k^{(d)}. \tag{4.11}
$$

(iii) Surrogate-Weight Update: Update $\theta_n^{(r+1)}, \psi_n^{(r+1)}$ around the new π_n, and repeat.

Repeating (i)–(iii) until convergence yields a high-quality solution to the original JBAS MINLP.

4.5 Assumptions and Scope

This section summarizes the modeling assumptions used in Sections 4.1–4.4 and clarifies the scope of the problem and algorithms studied in later chapters. We assume a single edge node, with a single GPU, and no coordination across multiple edge servers or the cloud. The problem concerns asynchronous, single-shot inference tasks with no feedback after execution; the tasks have independent processing requirements and given (exogenous) arrivals and deadlines. The scheduler has a finite planning horizon with K tasks, and produces batch start times $\{t_n\}$ within the horizon.

For communication, we adopt the FDMA equal-share surrogate described in Section 3.2 within each upload window $[T^{(a)}, t_n)$. We assume a fixed total bandwidth W available within each window, and treat per-task SNR values as constants within each window. Each task uses the standard AWGN rate expression

$$r = w \log_2\Big(1 + \mathrm{SNR}\Big),$$

and there are no retransmissions, HARQ, or fast fading. We also assume no cross-traffic beyond the given set of tasks, so interference and MAC effects are abstracted into the parameters W and the per-task SNRs.

On the computation side, we use the empirical batch-delay model

$$d(n) = a\,n + b,$$

derived in Section 3.3 by fitting our experimental measurements obtained on our hardware at a fixed stream count S chosen near the concurrency knee. We apply this model only within $n \in [1, \pi_{\max}]$. In the base formulation of Section 4.4, batches execute sequentially on a single server, and batches do not overlap in time. Stream offsets (explained in detail in Chapter 3) are ignored because their measured benefit is negligible ($\approx$ 1–3%) in our setting. Host-side preprocessing and post-processing are outside of $d(\cdot)$; their effect is studied empirically in later chapters.

The scheduling model takes the form of admission plus batching with fixed streams S, and the effect of concurrency is folded into the fitted coefficients a and b. There is no explicit stream-level scheduling or preemption within a batch. Batch size is limited by $\pi_{\max}$ from VRAM, and each task is in at most one batch. The scheduling is deadline-aware and treats deadlines as hard constraints: if a task is in batch n, then $t_n + d(n)$ must be before the task's deadline. To improve robustness, a small safety margin $\delta > 0$ may be applied to $d(\cdot)$.

Energy and thermal constraints, fairness or age-based objectives, and multiple-model interactions such as multiple queues for detection plus tracking are not modeled. Time-varying SNR and measurement noise in the (a, b) values are also out of scope; we mitigate them by incorporating a safety margin and advocating for

regular reassessment. Multi-edge coordination, backhaul contention, and low-level optimizations in the host pipeline are left to future work.

Finally, we note that the coefficients (a, b) are backend- and model-specific, and should be re-measured for the target hardware and backend before deployment. The new (a, b) must be plugged into the formulation in place of the default YOLO values for correctness and efficiency. While modest mis-fit can be tolerated by the safety margin, correctness and efficiency improve with fresh measurements. Under the above assumptions, the theoretical statements in Section 4.4 (problems P1–P4 and the batch-time recursion) are valid; the experiments in later chapters instantiate them using TensorRT + YOLOv11n and the radio surrogate described in Section 4.2.

Chapter 5

Algorithms

This chapter presents the scheduling algorithms benchmarked in Chapter 5.4. It starts with deployment goals (latency SLAs and throughput/utility), establishes baselines, and then describes **Greedy-JBAS**, a *priority-aware* batching scheduler tailored to the empirical delay model from Chapter 3. We assume that the computation delay per batch is parameterized by the linear fit $d(n) = a\,n + b$ and treat the parallel stream count S as a design choice selected from the concurrency knee of Chapter 4.

N.B. The plots presented throughout this chapter represent the general scenario where S equals 1 and priority is set to 1.

5.1 Design Goals

The primary objective is to maximize the weighted number of completed (on-time) tasks, subject to per-request deadlines and a shared uplink bandwidth constraint. As main SLA/KPI targets, we evaluate (i) weighted completion (utility) U and its normalized form $U/\sum_k \rho_k$, (ii) end-to-end latency, (iii) throughput (req/s), and (iv) planner time (ms per decision horizon). Planning budget is critical because the scheduler must produce decisions in milliseconds to cope with bursty arrivals; this makes exact MILP solvers (e.g., Gurobi) infeasible at the scales we consider (see Chapter 5.4 runtime comparison) [7].

5.2 Baselines

We compare against two structure–independent baselines that enforce the same latency constraints as our method. For fairness and relevance, both baselines are intentionally parameterized to be competitive. Algorithmic details are omitted here and given in the appendix.

5.2.1 Traditional Fixed–Batch

Traditional Fixed–Batch (TFB) picks a fixed batch size $B \in \{8,16\}$ and a fixed stream count S (e.g., $S = 5$), keeps a FIFO queue of arrivals, and launches a batch whenever B tasks are available and jointly feasible; otherwise, it launches at a timeout with the largest feasible subset. Joint feasibility is checked per task k that could be assigned to batch n with start time t_n: (i) its upload must complete by the time the batch starts, and (ii) its inference must finish by the deadline, i.e., $t_n + d(B) \leq T_k^{(d)}$. Tasks that cannot satisfy (i)–(ii) for any batch with the fixed (B, S) are rejected. Admission and feasibility checks are $O(1)$ amortized per arrival since B and S are constant and the model $d(\cdot)$ is affine, but the policy is non–adaptive w.r.t. the workload variability. Reported performance is the weighted utility $U = \sum_k \rho_k Y_k$ where $Y_k \in \{0,1\}$ is an indicator of completion by the deadline.

5.2.2 MEC pipeline

The MEC pipeline is like running TFB with no batching (B=1, S=1). A task k is admitted if its upload could finish by the time service begins; it is then scheduled immediately (or at the next available slot), with start time $t_n = \max\{T_k^{(a)}, \text{server-available}\}$. Feasibility requires $t_n + d(1) \leq T_k^{(d)}$; otherwise the task is rejected. The computational overhead is trivial, but the accelerator is typically under–utilized due to lack of batching.

Both baselines respect latency constraints without exploiting joint admission–plus–adaptive batching. They therefore provide informative references for quantifying the benefit of the proposed scheduler under the weighted objective and realistic timing assumptions.

5.3 Greedy-JBAS

Greedy-JBAS instantiates the batched-OAS structure from Chapter 4 with a priority-first, EDF tie-break insertion policy and closed-form batch-time updates, using the empirical compute model $d(n) = an + b$ and the radio surrogate from Chapter 4. Our GUI implementation follows a *grid-seeded alternating* scheme: (i) assign jobs to pre-seeded batch start times via priority-first + feasibility (EDF for ties), then (ii) run a backward two-line update on $\{t_n\}$ to satisfy deadlines and arrivals. This reflects the practical path used to generate the figures. (The GUI intro and images are available in the appendix.)

5.3.1 Inputs and Parameters

- **Per task** k**:** arrival $T_k^{(a)}$, deadline $T_k^{(d)}$, payload size ℓ_k (bits), **spectral efficiency** s_k (bits/s/Hz) with $s_k = \log_2(1+\gamma_k)$ (cf. Eq. (4.1)), and **priority weight** $\rho_k \geq 0$ (default 1).

- **Global:** total bandwidth W (Hz), delay coefficients (a, b) (from Chapter 4), optional batch-size cap $\pi_{\max}$ (VRAM), guard $\epsilon > 0$, and the number of seeded batches N (default $N = K$).

- **Operating choice:** stream count S held fixed (its effect folded into (a, b) per Chapter 4).

5.3.2 Backward Batch-Time Recursion

Let $\chi_n \triangleq \min\{\, T_k^{(d)} : \pi_{k,n} = 1 \,\}$ be the tightest deadline in batch n and $\pi_n = \sum_k \pi_{k,n}$ its size. The GUI applies the following two-line update backwards for $n = N, \ldots, 1$:

$$\begin{aligned} t_n &\leftarrow \min\{\, \chi_n,\ t_{n+1} \,\} \ - \ \Big(a\,\pi_n + b\Big), \\ t_n &\leftarrow \max\Big\{\, t_n,\ \max\{\, T_k^{(a)} : \pi_{k,n} = 1 \,\} + \epsilon \Big\}. \end{aligned} \tag{5.1}$$

This enforces both deadlines and causality. (When a batch is empty, the code sets $t_n \leftarrow t_{n+1}$.)

5.3.3 Algorithm

We seed N batch start times uniformly between the earliest arrival and a slackened horizon, then alternate assignment and time updates for a small number of rounds.

Algorithm 1 Greedy-JBAS (Priority-first + EDF tie-break + Feasibility)

procedure JBAS-GUI($\{T_k^{(a)}, T_k^{(d)}, \ell_k, s_k, \rho_k\}_{k=1}^{K}$, W, (a, b), N, ϵ, `max_iters` [, $\pi_{\max}$] [, δ])
Require: s_k is spectral efficiency in **bits/s/Hz**; W in **Hz**; ℓ_k in **bits**; $T_k^{(a)}, T_k^{(d)}, t_n$ in **seconds**; $d(n) = an + b$ in **seconds**.
Require: $s_k = \log_2(1 + \gamma_k)$; streams S are fixed and folded into (a, b) (cf. Ch. 4).
Ensure: weighted utility $U = \sum_k \rho_k Y_k$, assignments $\pi_{k,n} \in \{0,1\}$, start times $\{t_n\}_{n=1}^{N}$.
    ▷ Initialization of batch start times (uniform grid over the horizon)
    $T_{\min} \leftarrow \min_k T_k^{(a)}$; $T_{\max} \leftarrow \max_k T_k^{(d)}$
    $X_{\max} \leftarrow T_{\max} + (aK + b)$ ▷ slackened tail so last batch can fit
    $\text{span} \leftarrow (T_{\max} - T_{\min}) + \epsilon$
    for $n = 1$ **to** N **do**
        $t_n \leftarrow T_{\min} + \text{span} \cdot \frac{n-1}{\max(1, N-1)}$ ▷ uniform grid
    end for
    $t_{N+1} \leftarrow X_{\max}$ ▷ sentinel for backward update
    $\pi_{k,n} \leftarrow 0$ for all k, n ▷ no assignments yet
    ▷ Alternating loop: priority-first assignment (π-step) then backward time update (t-step)
    for `iter` $= 1$ **to** `max_iters` **do**
        ▷ π-step: for each batch n, assign arrived jobs priority-first (EDF tie-break) under bandwidth/deadline constraints
        for $n = 1$ **to** N **do**
            $U_n \leftarrow 0$ ▷ used bandwidth (Hz) in batch n's upload window
            $\mathcal{E}_n \leftarrow \{\, k : (\sum_j \pi_{k,j}) = 0 \ \wedge \ T_k^{(a)} \le t_n \,\}$ ▷ unassigned and arrived by t_n
            sort $\mathcal{E}_n$ by *decreasing* ρ_k (higher priority first), then by *non-decreasing* $T_k^{(d)}$ (EDF tie-break)
            for each $k \in \mathcal{E}_n$ **do**
                $\tau_o \leftarrow t_n - T_k^{(a)}$ ▷ available upload time for k into batch n
                if $\tau_o \le 0$ **then** ▷ not yet arrived ⇒ skip
                    continue
                end if
                $\text{reqBW} \leftarrow \dfrac{\ell_k}{s_k \, \tau_o}$ ▷ Hz needed so ℓ_k bits finish by t_n
                if $U_n + \text{reqBW} > W$ **then** ▷ bandwidth budget
                    continue
                end if
                $m \leftarrow 1 + \sum_i \pi_{i,n}$ ▷ prospective batch size if k is added
                $\tilde{d}(m) \leftarrow (a\, m + b) \cdot (1 + \delta)$ ▷ optional safety inflation; GUI uses δ=0
                if $t_n + \tilde{d}(m) > T_k^{(d)}$ **then** ▷ deadline feasibility
                    continue
                end if
                if defined($\pi_{\max}$) **and** $m > \pi_{\max}$ **then** ▷ optional VRAM cap
                    continue
                end if
                $\pi_{k,n} \leftarrow 1$; $U_n \leftarrow U_n + \text{reqBW}$ ▷ admit k into batch n
            end for
        end for
        ▷ t-step: backward update to honor per-batch tightest deadline and causality
        for $n = N$ **down to** 1 **do**
            $\pi_n \leftarrow \sum_k \pi_{k,n}$; $\chi_n \leftarrow \begin{cases} \min\{\, T_k^{(d)} : \ \pi_{k,n} = 1 \,\}, & \pi_n > 0 \\ t_{n+1}, & \pi_n = 0 \end{cases}$
            if $\pi_n > 0$ **then**
                $t_n \leftarrow \min\{\chi_n, \, t_{n+1}\} - (a\, \pi_n + b)$ ▷ back up to fit compute
                $t_n \leftarrow \max\{\, t_n, \ \max\{T_k^{(a)} : \pi_{k,n} = 1\} + \epsilon \,\}$ ▷ respect arrivals (causality)
            else
                $t_n \leftarrow t_{n+1}$ ▷ empty batch inherits next start
            end if
        end for
    end for
    $U \leftarrow \sum_k \rho_k \cdot \mathbb{1}\{\sum_n \pi_{k,n} = 1\}$ ▷ weighted utility of accepted tasks
    return U, $\{\pi_{k,n}\}$, $\{t_n\}_{n=1}^{N}$
end procedure

Complexity. Each iteration does a priority-first (then EDF) sort per batch and linear scans over candidates; with $N \ll K$ and a small `max_iters` (e.g., 8–12), planner time remains in the *milliseconds* in our runs.

5.4 Experimental Evaluation

We evaluate Greedy-JBAS against relevant baselines and, where feasible, exact and/or relaxed formulations. We use the hardware-calibrated computation model from Chapter 3 and the radio model from Chapter 4. Unless otherwise stated, we fix the inference backend to Ultralytics YOLO-11n + TensorRT and use the batch-delay fit $d(n) = an + b$ estimated from Chapter 3.

We generate single-edge, asynchronous-arrival workloads. The set of tasks is K, each k associated with a tuple $(T_k^{(a)}, T_k^{(d)}, \ell_k, r_k, \rho_k)$. Here $T_k^{(a)}$ and $T_k^{(d)}$ are arrival and deadline times; ℓ_k is the upload payload (feature size); r_k is the uplink rate from the radio model in Chapter 4; and ρ_k is the priority weight. We sweep the number of tasks K and also sweep SNR and bandwidth to stress the communication side. The following implementation scenarios can be imagined:

- **Bandwidth sweep:** a small grid of edge bandwidth budgets (low/med/high). Exact values are listed in Appendix B (Table B.x).
- **SNR sweep:** per-task SNRs are sampled from the specified range; we report results by SNR quantiles as well as by mean SNR.
- **Load sweep:** K varied (e.g., $20 \rightarrow 100$). We also report sensitivity to the number of batches N allowed (Section 5.5).

Computation model. We use the fitted TensorRT coefficients from Chapter 3:

Concurrency. The number of streams S is fixed near the concurrency knee (Chapter 3): $S \in \{5{,}6\}$ unless stated otherwise. VRAM determines the set of admissible (B, S) pairs.

Baselines. (i) Traditional Fixed-Batch (TFB) with $B \in \{8{,}16\}$ and EDF within each batch; (ii) MEC pipeline with $B = 1$. Definitions are in this chapter.

Metrics. We report **weighted completion (utility)** $U = \sum_k \rho_k Y_k$ and $U/\sum_k \rho_k$, throughput (req/s), p95 end-to-end latency, and planner runtime (ms). All results are averaged over ≥ 10 runs; we show the mean with 95% CI unless otherwise stated.

5.5 Sensitivity Studies

We quantify performance sensitivity for feature size, bandwidth, SNR, load (K), the number of batches (N) and the delay of the tasks, all for the case K = 100, S = 1 and priority = 1 (although they can be easily changed in the GUI). Unless otherwise noted, plots report the *weighted* completion fraction $U/\sum_k \rho_k$ (trends are qualitatively similar for unweighted completion).

- **Weighted acceptance vs. feature size (Fig. 5.1).** Acceptance decreases *monotonically* as the per-task upload size ℓ_k grows: larger ℓ_k use more bandwidth per task, so fewer tasks finish before their deadlines. The curve exhibits a steep drop at very small sizes, then a long, gradual tail toward zero.
- **Weighted acceptance vs. total bandwidth.(Fig. 5.2).** Acceptance increases *monotonically* with total bandwidth B, starting low at small B and rising toward a saturation plateau once the communication bottleneck is removed; rapid gain in the low-B regime, then flattening for $B \gtrsim$ 20–40 MHz.

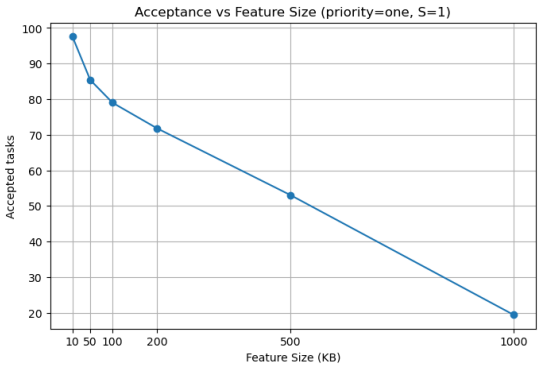

Figure 5.1: Acceptance vs. feature size.

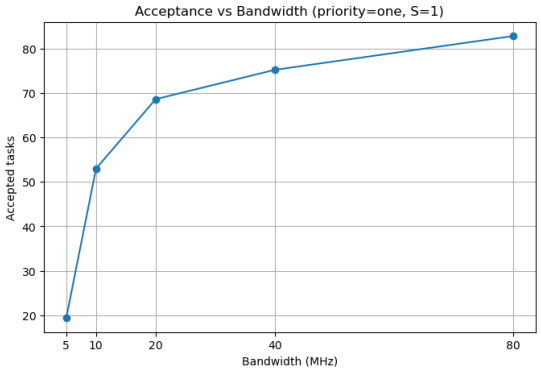

Figure 5.2: Acceptance vs. total bandwidth.

- **Weighted completion vs. number of tasks K (Fig. 5.3).** The normalized utility $U/\sum_k \rho_k$ *decreases monotonically* as K grows, reflecting increased competition for finite communication and compute resources.
- **Effect of minimum delay requirement (Fig. 5.4).** Tight deadlines (e.g., 50 ms) produce low weighted completion, since only the earliest arrivals and fastest links can finish both upload and inference. Relaxing the minimum delay to a few hundred milliseconds causes a rapid jump; beyond ∼1 s the curve *plateaus near 100%* once both communications and compute are easily accommodated.

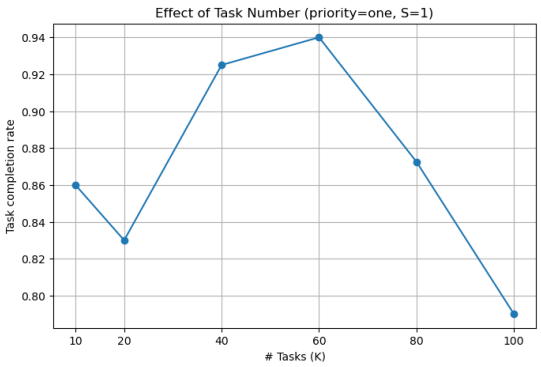

Figure 5.3: Completion vs. number of tasks K.

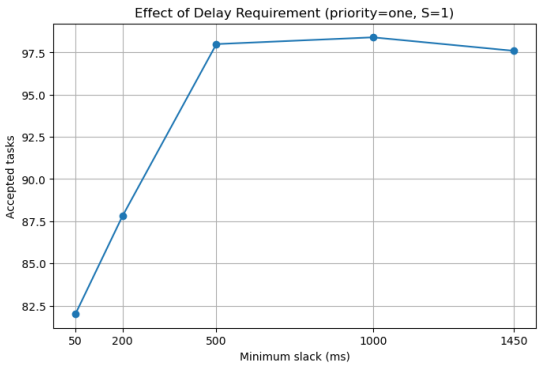

Figure 5.4: Completion vs. minimum delay requirement.

- **Effect of transmit SNR (Fig. 5.5).** The weighted completion follows an S-curve as SNR increases:
  - *Low SNR* (< 10 dB): spectral efficiencies r_k are very small, so uploads take too long and most tasks miss their deadlines.
  - *Mid SNR* (10–30 dB): modest power gains yield large rate improvements, sharply reducing upload times and boosting completion.
  - *High SNR* (> 30 dB): communications stop being the bottleneck—compute latency $a\,\pi + b$ dominates—so the curve flattens near its ceiling.

- **Effect of Batch Count N (with $K = 60$, Fig. 5.6).** The completion rate rises with the number of batches:
  - *Few batches* ($N \ll K$): forcing many tasks together inflates both upload and compute delays, yielding low throughput.
  - *Moderate N* (20–40): more slots spread tasks out, reducing per-batch size (compute delay) and giving late arrivals extra chances—acceptance climbs steeply.
  - *Large $N \gtrsim 40$*: additional batches add only marginal flexibility; overhead of tiny batches can even slightly hurt, so the curve saturates.

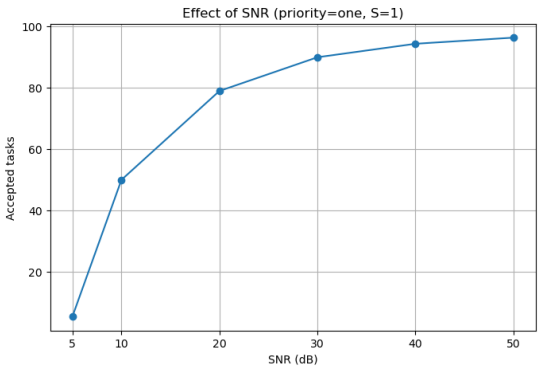

Figure 5.5: Completion vs. transmit SNR.

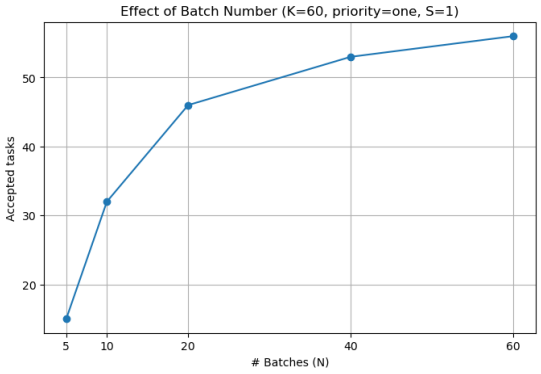

Figure 5.6: Completion vs. number of batches N ($K = 60$).

5.6 Optimality Gaps

Starting by figure 5.7, we compare the solve times of our greedy-JBAS against the mixed–integer formulation (P1) problem.

- **JBAS-Greedy** Solve time remains nearly constant (about 0.01–0.04s) even as K grows to 100, reflecting its simple combinatorial updates.
- **Direct MILP (P1)** Time starts increasing rapidly once $K \approx 40$, consistent with nonconvex coupling between π and t and large integer spaces, leading to prohibitively long runtimes in the large-K regime.

Next, we compare the mixed–integer formulation (P1) from Sect. 4.4 — solved on an *anchored time grid* — against the Greedy Alternating JBAS (Alg. 1). The batch time here is $d_S(B) = a_S B + b_S$ with S=1 (measured coefficients). The upper bound (UB) is computed from the LP relaxation on the *same* grid. For larger instances, we retain the UB and compare it to the heuristic only.

For small number of tasks K, (where MILP is feasible) we report:

$$\text{Relax gap } (\%) = \frac{\text{UB} - \text{OPT}}{\max\{\text{UB}, \varepsilon\}}, \qquad \text{Heuristic gap } (\%) = \frac{\max\{0, \text{OPT} - \text{Greedy}\}}{\max\{\text{OPT}, \varepsilon\}}$$

with $\varepsilon = 10^{-9}$. For large K (no MILP), we use

$$\text{Heuristic gap vs UB } (\%) = \frac{\max\{0, \text{UB} - \text{Greedy}\}}{\max\{\text{UB}, \varepsilon\}}$$

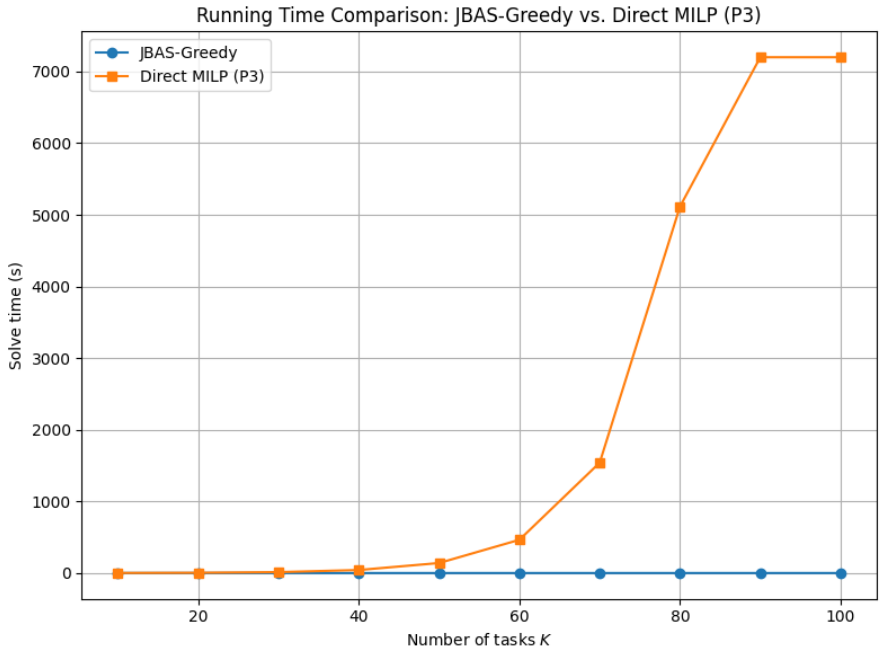

Figure 5.7: Solve-time comparison as a function of task count K: JBAS-Greedy remains below 0.05s, while the direct MILP solver time explodes beyond $K \approx 50$.

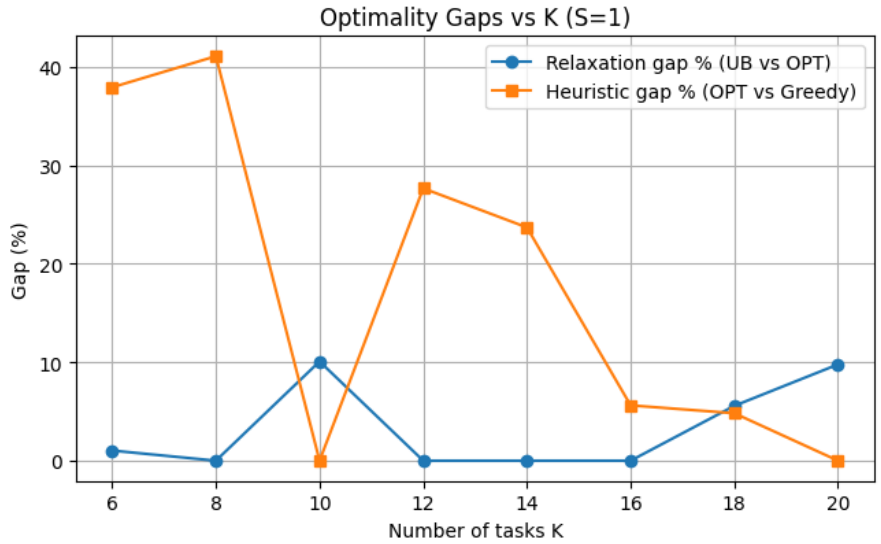

Figure 5.8: Small-K results (S=1). LP relaxation is often tight (gap $\leq 10\%$), and Greedy Alternating is often within 5–40% of MILP optimum, and optimal at some K where the grid matches optimal batch starts well.

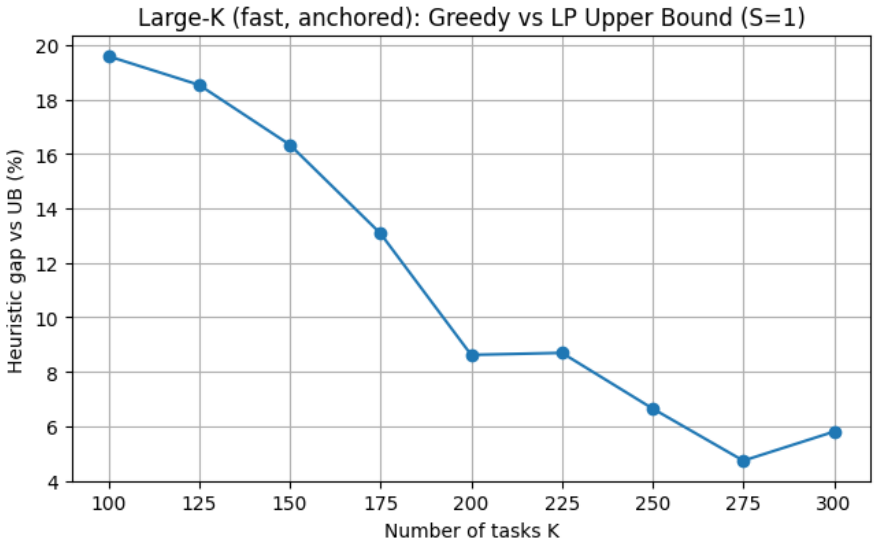

Figure 5.9: Large-K fast experiment (S=1): Greedy vs. LP upper bound. The gap decreases from $\approx$ 19% (K=100) to $\approx$ 6%–9% for $K \in [200,300]$, indicating that batching opportunities and the anchored UB both tighten at scale.

Table 5.1 reports optimality gaps in the small- and large-K regimes. For small K we solve both the MILP and its LP relaxation on an adaptive *anchored* grid of candidate batch start times seeded at arrivals/deadlines and locally refined; these yield, respectively, a grid-consistent integer optimum (OPT) and an upper bound (UB).

Our Greedy Alternating JBAS is naturally formulated in continuous time. In many instances, the relaxation gap (UB − OPT)/UB is near zero, which is an empirical indication that the LP relaxation is tight on the anchored grid. The heuristic gap $\max\{0, \text{OPT} - \text{Greedy}\}/\text{OPT}$ varies with deadline tightness and bandwidth but often shrinks with K as batching opportunities increase. When Greedy exceeds the grid-based UB, this is due to discretization (i.e., Greedy may place batch starts at arbitrary points between anchors); we therefore conservatively clip negative gaps to zero rather than claim super-optimality.

Small-K Results					
K	UB	OPT	Greedy_cont	RelaxGap (%)	HeurGap (%)
6	4.5249	4.4781	2.7798	1.03	37.92
8	6.6513	6.6513	3.9190	0.00	41.08
10	8.1538	7.3320	7.3320	10.08	0.00
12	9.9932	9.9932	7.2287	0.00	27.66
14	10.8335	10.8335	8.2669	0.00	23.69
16	11.7780	11.7780	11.1155	0.00	5.62
18	14.7291	13.9089	13.2385	5.57	4.82
20	15.5258	14.0068	14.7624	9.78	0.00

Large-K Results			
K	UB	Greedy	Gap vs UB (%)
100	81.64	65.65	19.58
125	98.01	79.85	18.53
150	112.65	94.25	16.33
175	128.53	111.72	13.07
200	139.45	127.43	8.62
225	160.42	146.48	8.69
250	161.85	151.09	6.65
275	170.77	162.69	4.73
300	185.26	174.51	5.80

Table 5.1: Optimality-gap summaries for Small-K where LP relaxation (UB) and MILP (OPT) are solved on an adaptive anchored time grid. For Large-K we only reported Greedy vs. LP–UB

5.7 Comparison of Different Algorithms

Figure 5.10 shows how the number of tasks K impacts the completion rate of three approaches: JBAS-Greedy, the MEC Pipeline heuristic, and a Traditional Batch scheduler. Through it we can observe that JBAS-Greedy holds high completion (≈ 0.9) up to $K = 60$, then gently falls to ≈ 0.75 at $K = 100$. MEC Pipeline declines steadily from ≈ 0.9 at $K = 10$ to ≈ 0.25 at $K = 100$. Traditional Batch collapses from ≈ 0.3 at $K = 10$ to below 0.1 by $K = 100$.JBAS-Greedy dynamically balances communication and compute across batches. The MEC Pipeline's fixed slot assignments suffer growing congestion as K increases. Traditional Batch's rigid grouping cannot adapt to larger K, so most tasks miss their deadlines.

Figure 5.11 shows the effect of the minimum delay requirement on the same three methods. In this figure, we can see that JBAS-Greedy reaches 100% once the delay exceeds ≈ 500 ms (with a slight dip at 500 ms). MEC Pipeline climbs from ≈ 0.48 at 50 ms to ≈ 0.65 at 2000 ms. Traditional Batch slowly rises from ≈ 0.16 to ≈ 0.26.Relaxed deadlines allow JBAS-Greedy to fit all tasks into feasible slots. The MEC Pipeline benefits from extra slack but remains limited by its fixed-pipeline structure. Traditional Batch's inflexible grouping can't exploit larger delays, so throughput stays low.

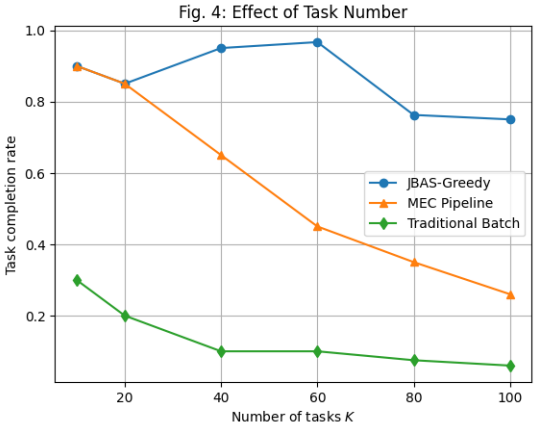

Figure 5.10: Completion rate vs. number of tasks K for JBAS-Greedy, MEC Pipeline, and Traditional Batch.

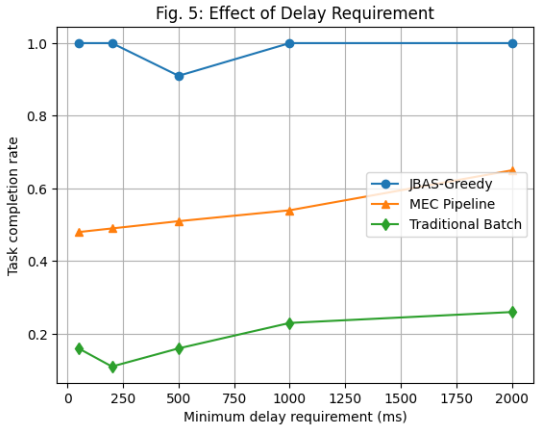

Figure 5.11: Completion rate vs. minimum delay requirement for JBAS-Greedy, MEC Pipeline, and Traditional Batch.

In figure 5.12, JBAS-Greedy shows a classic S-curve, very low (≈ 0) at 5 dB, then steep rise between 10–30 dB, saturating near 1.0 by 50 dB. MEC Pipeline climbs more gradually from ≈ 0.03 to ≈ 0.50. Traditional Batch remains near zero, reaching only ≈ 0.14 at 50 dB. At low SNR, upload rates $r_k = \log_2(1+\text{SNR} \times h_k)$ are too small—most tasks miss their deadlines. As SNR rises to 10–30 dB, each extra dB dramatically speeds uploads, boosting acceptance. Above $\sim$30 dB, compute latency $a\,p_n + b$ dominates, so JBAS-Greedy reaches $\approx 100\%$ and MEC Pipeline plateaus at its structural limit. Traditional Batch, with no adaptive scheduling, cannot exploit higher rates.

In figure 5.13, JBAS-Greedy jumps from ≈ 0.80 at $B = 5$ MHz to ≈ 0.99 by $B = 20$ MHz, then fully saturates at 1.0. MEC Pipeline increases from ≈ 0.34 to ≈ 0.56, with diminishing gains beyond 40 MHz. Traditional Batch rises modestly from ≈ 0.08 to ≈ 0.16. At low B, bandwidth scarcity throttles all algorithms—JBAS-Greedy still adapts batches to maximize throughput. As B passes $\sim$20 MHz, uploads finish quickly and JBAS-Greedy hits 100% acceptance. MEC Pipeline and Traditional Batch remain constrained by their fixed slot structure and compute delays, so their acceptance rates plateau below 1.0.

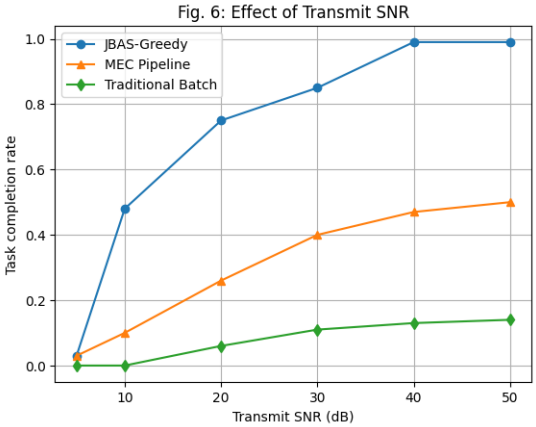

Figure 5.12: Completion rate vs. transmit SNR for JBAS-Greedy, MEC Pipeline, and Traditional Batch.

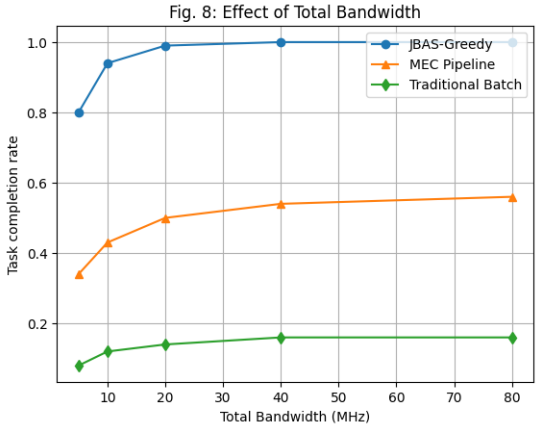

Figure 5.13: Completion rate vs. total bandwidth B for JBAS-Greedy, MEC Pipeline, and Traditional Batch.

Chapter 6

Conclusion & Future Work

Real-world edge inference is limited not only by the raw throughput of deployed models, but also by host and network constraints. By reasoning about upload windows, batch delays, and deadlines directly, our algorithm achieves both predictable latency and high throughput at the cost of only implementation-level complexity, a useful sweet spot between heuristic batching and intractable optimal search.

6.1 Summary of Contributions

In this thesis we have sought to understand how deadline–constrained vision workloads can be scheduled on an edge node that must juggle both radio uploads and GPU batching. The problem setting is very common in practice, since: (1) tasks arrive asynchronously, each with an arrival time and a hard deadline; (2) radio uploads complete at SNR–dependent rates; and (3) the accelerator has classic batching behavior with a fixed overhead and linear per–item cost. The objective has been to admit and batch these tasks so that the *weighted* number of timely completions is maximized, while keeping scheduling latency small enough to be deployable.

To study this problem in a principled way, we began by building a simple but faithful model linking communication and computation. On the radio side, each task's upload rate is characterized by an AWGN surrogate dependent on its SNR and the available bandwidth. On the compute side, we calibrated the per–batch delay of a modern stack (TensorRT + YOLO) and showed that it is well described by an affine curve with a concurrency scaling that explains the observed "knees" in batch size and streams. These two ingredients then let us formulate a joint admission–and–batching problem with hard deadlines and sequential batches.

Solving this problem directly as a mixed–integer program is instructive but turns out to be impractical at scale. We therefore took a two–pronged approach. On the *theory* track, we adapted an anchored–grid relaxation from batched OAS: by solving a linear relaxation and an integer program on the same set of candidate start times, we can get a reliable upper bound (UB) and an exact optimum (OPT) *on that grid*, which can then be used for diagnosis and benchmarking. On the *algorithm* track, we designed *Greedy–JBAS*, a lightweight scheduler that orders tasks by deadline and priority, admits only bandwidth– and deadline–feasible assignments, and places batches via a closed–form backward timing recursion. In effect, we retain the structure of the optimality conditions but avoid the heavy solvers, yielding millisecond–scale decisions.

Our experiments demonstrate that this combination is both effective and practical. Greedy–JBAS consistently produces deadline–respecting schedules and tracks the grid–based UB/OPT closely, while being orders of magnitude faster than MILPs. As the number of tasks increases, there are more batching opportunities and the upper bound tightens, so the heuristic's gap to UB shrinks. At smaller scales and tight slack, the heuristic may be more conservative but it remains feasible and predictable. Beyond aggregate metrics, the measurements also help to clarify *when* batching is helpful (ample slack, healthy ingress) and *when* it is harmful (tight slack, slow links), leading to simple guardrails—slack–aware waiting caps, a two–queue design for urgent jobs, and safety–inflated feasibility checks—that preserve performance without increasing complexity.

Finally, this work is also the first to contribute a number of practical artifacts: a calibrated compute model and concurrency scaling that can be re–fit on new hardware; an anchored–grid tool to generate UB/OPT for analysis; and an implementable scheduling policy suitable for real deployments. The scope was intentionally narrow—single edge node, single model, fixed coefficients, and simplified host pipeline—which keeps the problem tractable and the insights clear. These limitations, together with opportunities for multi–model scheduling, multi–edge coordination, energy awareness, and learning–augmented control, are discussed in the sections that follow.

6.2 Limitations

We study a *single* edge node without multi-edge coordination or placement; extending admission with routing is future work. The compute model uses hardware–specific (a, b) from TensorRT + YOLO-11n; different backends/models shift these coefficients. We mitigate via a safety factor δ and recommend periodic

re–fitting. The host pipeline is simplified (Python/C++ preprocessing and I/O variability are abstracted), and energy is unmodeled; adding power/thermal constraints would enable energy–aware scheduling. Fairness is not enforced—EDF can starve very loose jobs under sustained load—suggesting weight/age–based refinements.

Threats to validity include dataset and hardware dependence (COCO subset; EPYC+GV100 class machine), framework variance (Ultralytics/MMDetection) and TensorRT build choices (precision/tactics), and MILP solver sensitivity (parameters/time limits). We standardize configurations, apply warm–ups, report versions, and use a common time budget for fairness, but local re–fitting and sensitivity checks remain advisable. Finally, UB/OPT are computed on a finite *anchored* grid; finer grids can only lower UB and raise OPT, so reported gaps are conservative for the continuous–time problem.

6.3 Future Work

We view this work as a first step in the right direction. A natural follow-up is to move past simulations to a real edge–AI platform where traces can be replayed with realistic latencies, transient failures, and non-ideal topologies. A field-collected dataset of long traces of arrivals, payload sizes, and radio conditions from an IoT/camera fleet, replayed under measured and transient failures and non-ideal topologies, would reveal inadequacies in our current abstractions (e.g., AWGN rate surrogate, affine batch model), and be a useful platform for tuning the scheduler for both scalability and robustness in production environments.

A systems re-engineering effort around the host pipeline can likely produce quick wins. Our measurements point to Python-side preprocessing and orchestration as a concurrency knee. We believe a C++ or multi-process replacement of this portion of the path should eliminate this knee and improve completions under the same SLAs. Telemetry and instrumentation of the pipeline with fine-grained queueing delays, per-stage service times, and CPU/GPU utilizations will be important to close the loop on online adaptation and regression testing.

A more holistic scheduling study would also address multi-model and multi-queue scenarios. Many edge stacks co-host models (e.g., detection and classification) with significantly different delay curves $d_m(n)$ and priorities. A logical extension of the policy is to jointly allocate batches across models, while enforcing fairness guarantees across SLA classes. This makes the approach applicable to a much richer set of workloads. At larger scales, the scheduling problem couples with placement: coordination of admission and routing decisions across multiple edge nodes, mobility and handoff management, and shared-splitting of radio uplink bandwidth and inter-edge traffic.

Energy and sustainability metrics are also natural first-class objectives. Augmenting the objective with Joules per task or some carbon proxy, and coupling with device power telemetry or calibrated power models, could open the door to explicit tradeoffs between timeliness and energy. This would make possible energy-aware admission (e.g., throttling batch size in high-carbon periods) while still honoring hard deadlines.

Learning-augmented control is a promising direction. Contextual bandits or lightweight RL can be used to adapt batch size B, effective streams S, and safety margins online, seeded by Greedy–JBAS decisions and trajectories. The key is to treat feasibility as a hard constraint: the learned policy must never schedule a job that cannot meet its deadline. With appropriate guardrails, learning can track slow drift in workload and network conditions without sacrificing predictability.

Lastly, we leave some interesting directions on tighter analysis and stronger bounds. On the optimization side, we can work to tighten the anchored-grid construction (e.g., data-driven anchors, column generation) to improve the LP upper bound and shrink the MILP search space. On the robustness side, rolling re-fits of (a, b) and distributionally robust variants that hedge against rate and slack uncertainty could provide conservative but easily computable feasibility certificates in the wild. We also plan to work towards establishing approximation guarantees for the greedy policy under more realistic traffic models to better understand when and why the policy is near-optimal.

Appendix A

Hyperparameters, Versions, and Configs

Hardware	
CPU	`x86_64` (multi-core)
RAM	252 GB
GPU	NVIDIA Quadro GV100
VRAM	31.73 GB
Operating System	
Distribution	Ubuntu 22.04.1 LTS
Kernel	Linux 6.8.0-57-generic (#59~22.04.1-Ubuntu)
Drivers / Runtimes	
CUDA toolkit	11.5 (Cuda compilation tools, release 11.5, V11.5.119)
cuDNN	8.9.2
TensorRT	8.6.1
Optimization Solver	
Gurobi (gurobipy)	*12.0.2*
Frameworks	
PyTorch	2.2.2
Ultralytics / MMDetection	8.3.99 / Not installed

Table A.1: Hardware/software summary used for all experiments.

A.1 Inference Model Settings

Table A.2: Model and input settings

Setting	YOLO-11n	YOLO-11m	Notes
Input size	640×640	640×640	Standard Ultralytics default
Precision	FP16	FP16	Engines built with mixed-precision
NMS	IoU 0.6, conf 0.25	IoU 0.6, conf 0.25	Defaults from Ultralytics

A.2 TensorRT Engine Build

Table A.3: TRT build configuration

Parameter	Value	Notes
Max workspace	4096 MB	Typical default; limits temporary memory
Tactics	Enabled (default search)	TensorRT autotunes kernels
Precision	FP16	Chosen for throughput; INT8 not used
Dynamic shapes	Enabled	Engine supports variable batch sizes
Max batch ($\pi_{\max}$)	32	Export limited to 32 images per call

A.3 YAML/JSON configs

Below are minimal, copyable configs used to reproduce experiments. Replace with your actual files or include via `\lstinputlisting`.

Listing A.1: Minimal YAML config

```yaml
experiment:
  backend: trt
  model: yolov11n
  batch_candidates: [1,2,4,8,16]
  streams: 5
  delay_fit: { a: 0.011982, b: 0.006082, delta: 0.05 }
  pi_max: 16
  epsilon: 0.001
  seeds: [1,2,3,4,5]
  dataset: coco_val_1024
```

Appendix B

GUI Overview and Screenshots

B.1 Purpose and scope

This is a desktop app that simulates our *Greedy–JBAS* of algorithm 1 under resource-constrained uploads and deadline constraints. It stochastically generates instances of tasks (arrivals, SNR-driven rates, slack windows) and assembles them into batches via a greedy EDF policy, subject to feasibility checks under both link and compute models. The GUI exposes the main knobs (feature size, bandwidth, SNR, slack ranges, number of batches N, and streams S) and plots seed–averaged outcomes such as *accepted tasks* and *weighted utility*. Priorities are optional and user-selectable from the UI.

Model (enforced in simulator). Per task k assigned to batch n with start time t_n:

$$\text{(upload feasibility)} \quad \frac{F_{\text{bits}}}{(t_n - T_k^{(a)})\, r_k} \;\le\; B_{\text{tot}}, \qquad r_k = \log_2\!\Big(1 + \text{SNR}_k\Big),$$

$$\text{(compute feasibility)} \quad t_n + d_S\Big(|\mathcal{B}_n|\Big) \;\le\; T_k^{(d)}, \qquad d_S(B) = \max\{0,\; a_S B + b_S\}.$$

Here F_{bits} is the feature payload, B_{tot} the total uplink bandwidth (Hz), and $|\mathcal{B}_n|$ the batch size. The GUI *automatically* chooses (a_S, b_S) for the chosen streams S from measured coefficients when available, and otherwise falls back to the generalized concurrency surface

$$a_S \approx \frac{a_1}{1 + \gamma(S-1)}, \qquad b_S \approx \frac{b_1}{1 + \gamma(S-1)},$$

with fixed γ from calibration. The plotted KPI is either the number of accepted tasks or the *weighted utility* $U = \sum_k \rho_k Y_k$ (acceptance indicator $Y_k \in \{0,1\}$).

B.2 Architecture

The app is implemented in Python 3 using Tkinter (UI) and Matplotlib (plots); the core scheduler is vectorized NumPy. A lightweight worker thread runs simulations and marshals updates back onto Tk's main loop to keep the UI responsive. Major components of the app are as below:

1. **Instance generator:** draws arrivals $T^{(a)}$, SNRs (Rayleigh path loss $\rightarrow$ SNR), per-task rates r_k, and deadlines $T^{(d)}$ from user-specified slack ranges; also generates priorities ρ_k by mode (`one/low/med/high/rnd`).

2. **Greedy assignment step:** within each tentative batch start, ranks eligible tasks by a priority-weighted value-per-bandwidth score (EDF tie-break), admits only if both link and deadline constraints pass.

3. **Alternating timing:** after assignment, a backward "just-in-time" recursion places batch start times t_n to respect deadlines; steps (assignment/timing) repeat a few iterations (milliseconds runtime at typical sizes).

4. **UI & export:** contextual parameter panes (scrollable), a Matplotlib canvas, PNG/CSV export, a timestamped log, and a progress bar.

B.3 Screenshots

For each selected action, the user will have a wide variety of options to choose the parameters to be tuned, such as the number of streams, the KPI, and the priority index. In the output plots, "Accepted" is the count of tasks scheduled without violating upload or compute feasibility. "Weighted" sums ρ_k over accepted tasks. Trends versus feature size, bandwidth, SNR, K, N, and S reflect the same constraints as the theory chapters.

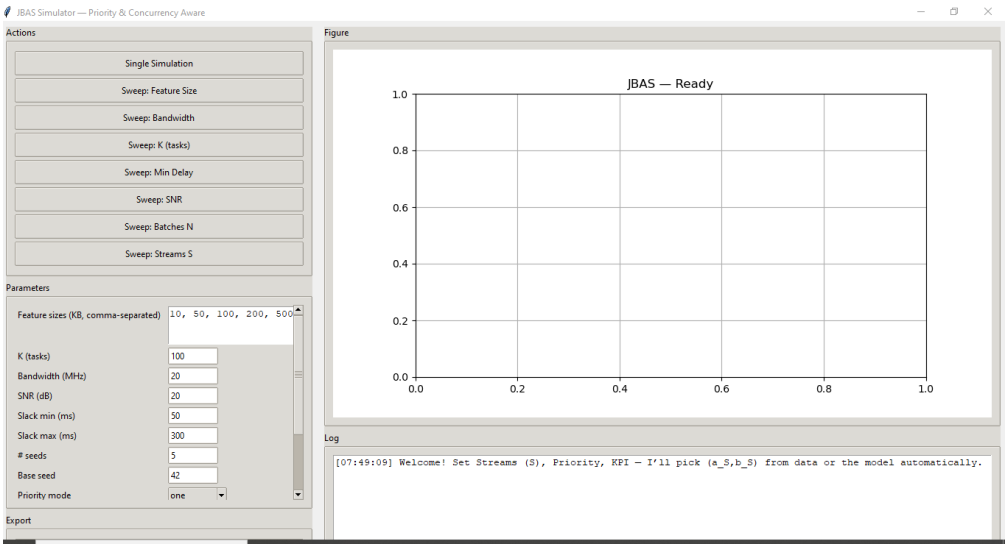

Figure B.1: Main dashboard with actions, scrollable parameters, plot canvas, export, and log.

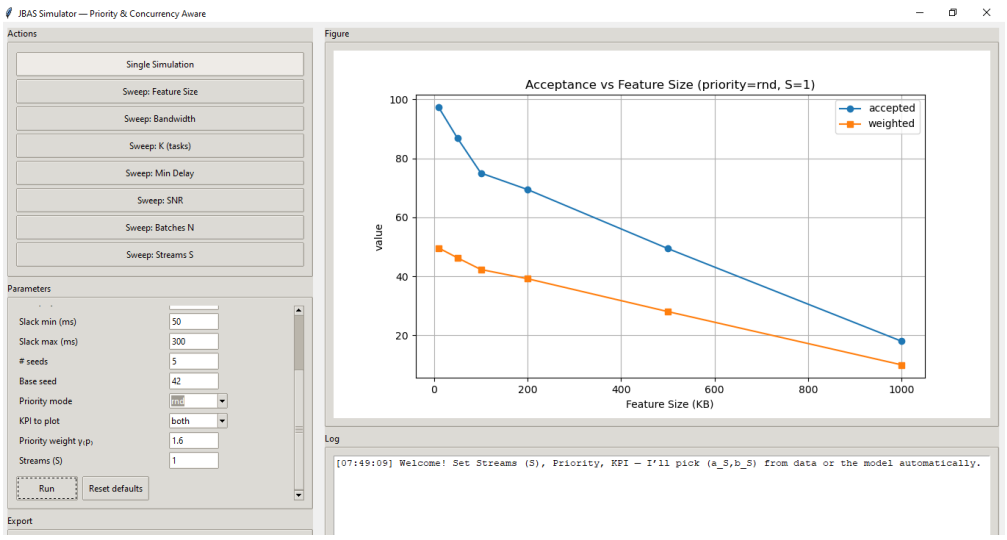

Figure B.2: Acceptance vs. feature size (seed–averaged).

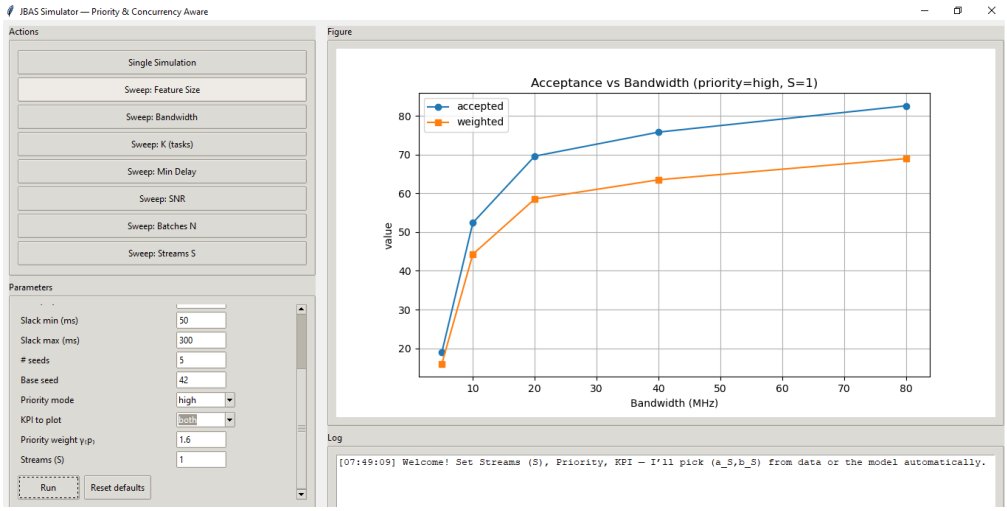

Figure B.3: Acceptance vs. bandwidth (seed–averaged).

Appendix C

Baseline algorithms

Here, we provide an explanation to the two baseline batching algorithms that were used to evaluate the performance of our novel algorithm with respect to them.

1. Traditional Batching

Goal: Form exactly one batch at the optimal start time to maximize the number of tasks completed before their deadlines.
Algorithm:

1. Collect candidate start times $\{t_1\}$ from each task's arrival $T_k^{(a)}$.
2. For each t_1:
   - Greedily fill one batch by selecting tasks with $T_k^{(a)} \leq t_1$, in order of earliest deadline.
   - For each selected task, check:
     - Uplink bandwidth: $\sum \frac{L}{r_k\,(t_1 - T_k^{(a)})} \leq B_{\text{tot}}$.
     - Deadline constraint: $t_1 + d(b) \leq T_k^{(d)}$.
   - Compute the fraction accepted.
3. Choose the t_1 that yields the highest acceptance rate.

2.MEC Pipeline

Goal: Serve tasks one by one (no batching), consuming communication and compute time for each task in arrival order.
Algorithm:

1. Sort tasks by increasing arrival time $T_k^{(a)}$.
2. Initialize clock $now \leftarrow 0$.

3. For each task k in arrival order:
   - Start upload at $t_{\text{start}} = \max(now,\, T_k^{(a)})$.
   - Upload duration: $\tau_k^o = \dfrac{L}{r_k\, B_{\text{tot}}}$.
   - Inference duration: $d(1) = a + b_0$.
   - Finish time: $t_{\text{finish}} = t_{\text{start}} + \tau_k^o + d(1)$.
   - If $t_{\text{finish}} \le T_k^{(d)}$, count task as completed.
   - Advance clock: $now \leftarrow t_{\text{finish}}$.

Bibliography

[1] Vítor Teixeira, Carlos Pires, Fernando Pinto, João Freitas, Miguel Sales Dias, and Eduarda Mendes Rodrigues. «Towards Elderly Social Integration using a Multimodal Human-computer Interface». In: *Proceedings of the 2nd International Living Usability Lab Workshop on AAL Latest Solutions, Trends and Applications - Volume 1: AAL, (BIOSTEC 2012)*. INSTICC. SciTePress, 2012, pp. 3–13. ISBN: 978-989-8425-93-5. DOI: 10.5220/0003852800030013 (cit. on p. 1).

[2] Robert Nishihara et al. «Real-Time Machine Learning: The Missing Pieces». In: *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*. HotOS '17. Whistler, BC, Canada: Association for Computing Machinery, 2017, pp. 106–110. ISBN: 9781450350686. DOI: 10.1145/3102980.3102998. URL: https://doi.org/10.1145/3102980.3102998 (cit. on p. 2).

[3] Yuxiao Zhou and Kecheng Yang. «Exploring TensorRT to Improve Real-Time Inference for Deep Learning». In: *2022 IEEE 24th Int Conf on High Performance Computing Communications; 8th Int Conf on Data Science Systems; 20th Int Conf on Smart City; 8th Int Conf on Dependability in Sensor, Cloud Big Data Systems Application (HPCC/DSS/SmartCity/DependSys)*. 2022, pp. 2011–2018. DOI: 10.1109/HPCC-DSS-SmartCity-DependSys57074.2022.00299 (cit. on p. 2).

[4] Ziyang Zhang, Yang Zhao, Huan Li, and Jie Liu. «BCEdge: SLO-Aware DNN Inference Services With Adaptive Batch-Concurrent Scheduling on Edge Devices». In: *IEEE Transactions on Network and Service Management* 21.4 (2024), pp. 4131–4145. DOI: 10.1109/TNSM.2024.3409701 (cit. on p. 2).

[5] NVIDIA. *TensorRT Batching and Optimization Guide*. Developer Documentation. 2023. URL: https://docs.nvidia.com/deeplearning/tensorrt/archives/tensorrt-861/developer-guide/index.html#appendix (visited on 09/11/2025) (cit. on pp. 3–5).

[6] Ultralytics. *Ultralytics YOLO Documentation*. Software documentation. 2023. URL: https://docs.ultralytics.com (visited on 09/11/2025) (cit. on p. 3).

[7] Yihan Cang, Ming Chen, and Kaibin Huang. *Joint Batching and Scheduling for High-Throughput Multiuser Edge AI with Asynchronous Task Arrivals*. 2023. arXiv: 2307.14350 [eess.SP]. URL: https://arxiv.org/abs/2307.14350 (cit. on pp. 3–5, 30, 31, 36).

[8] Ziming Yang, Zichuan Zheng, Liyou Deng, Shan Zhang, Zhiyuan Wang, and Hongbin Luo. «ACBatch: Adaptive and Cooperative Batching for Edge Inference». In: *IEEE INFOCOM 2025 - IEEE Conference on Computer Communications*. 2025, pp. 1–10. DOI: 10.1109/INFOCOM55648.2025.11044583 (cit. on pp. 3, 5, 30).

[9] Yoshiaki Inoue. «Queueing Analysis of GPU-Based Inference Servers with Dynamic Batching: A Closed-Form Characterization». In: *CoRR* abs/1912.06322 (2019). arXiv: 1912.06322. URL: http://arxiv.org/abs/1912.06322 (cit. on p. 3).

[10] Vyacheslav Zhdanovskiy, Lev Teplyakov, and Philipp Belyaev. «Efficient single- and multi-DNN inference using TensorRT framework». In: *Sixteenth International Conference on Machine Vision (ICMV 2023)*. Ed. by Wolfgang Osten. Vol. 13072. International Society for Optics and Photonics. SPIE, 2024, p. 1307215. DOI: 10.1117/12.3023487. URL: https://doi.org/10.1117/12.3023487 (cit. on p. 3).

[11] Mounir Bensalem, Jasenka Dizdarević, and Admela Jukan. «Modeling of Deep Neural Network (DNN) Placement and Inference in Edge Computing». In: *2020 IEEE International Conference on Communications Workshops (ICC Workshops)*. 2020, pp. 1–6. DOI: 10.1109/ICCWorkshops49005.2020.9145449 (cit. on p. 3).

[12] Jiawei Shao and Jun Zhang. «Communication-Computation Trade-Off in Resource-Constrained Edge Inference». In: *CoRR* abs/2006.02166 (2020). arXiv: 2006.02166. URL: https://arxiv.org/abs/2006.02166 (cit. on p. 3).

[13] İstenç Tarhan and Ceyda Oğuz. «Generalized order acceptance and scheduling problem with batch delivery: Models and metaheuristics». In: *Computers Operations Research* 134 (2021), p. 105414. ISSN: 0305-0548. DOI: https://doi.org/10.1016/j.cor.2021.105414. URL: https://www.sciencedirect.com/science/article/pii/S0305054821001775 (cit. on pp. 4, 8, 30).

[14] Ceyda Og˘uz, F. Sibel Salman, and Zehra Bilgintürk Yalçın. «Order acceptance and scheduling decisions in make-to-order systems». In: *International Journal of Production Economics* 125.1 (2010), pp. 200–211. ISSN: 0925-5273. DOI: https://doi.org/10.1016/j.ijpe.2010.02.002. URL: https://www.sciencedirect.com/science/article/pii/S0925527310000514 (cit. on pp. 4, 8, 31).

[15] Ying Chen, Yongchao Zhang, and Xin Chen. «Dynamic Service Request Scheduling for Mobile Edge Computing Systems». In: *Wireless Communications and Mobile Computing* 2018.1 (2018), p. 1324897. DOI: https://doi.org/10.1155/2018/1324897. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1155/2018/1324897. URL: https://onlinelibrary.wiley.com/doi/abs/10.1155/2018/1324897 (cit. on p. 4).

[16] Author Zhang et al. «BCEdge: SLO-Aware DNN Inference Services with Adaptive Batching on Edge Platforms». In: *Proceedings of ACM/IEEE Conference*. 2024 (cit. on pp. 4, 7, 9).

[17] Nasir Abbas, Yan Zhang, Amirhosein Taherkordi, and Tor Skeie. «Mobile Edge Computing: A Survey». In: *IEEE Internet of Things Journal* 5 (2018), pp. 450–465. URL: https://api.semanticscholar.org/CorpusID:31429854 (cit. on p. 4).

[18] Wazir Zada Khan, Ejaz Ahmed, Saqib Hakak, Ibrar Yaqoob, and Arif Ahmed. «Edge computing: A survey». In: *Future Generation Computer Systems* 97 (2019), pp. 219–235. ISSN: 0167-739X. DOI: https://doi.org/10.1016/j.future.2019.02.050. URL: https://www.sciencedirect.com/science/article/pii/S0167739X18319903 (cit. on p. 4).

[19] Adam Paszke et al. «PyTorch: an imperative style, high-performance deep learning library». In: *Proceedings of the 33rd International Conference on Neural Information Processing Systems*. Red Hook, NY, USA: Curran Associates Inc., 2019 (cit. on pp. 4, 8).

[20] OpenMMLab. *MMDetection Documentation*. Software documentation. 2023. URL: https://mmdetection.readthedocs.io (visited on 09/11/2025) (cit. on p. 4).

[21] Yiping Kang, Johann Hauswald, Cao Gao, Austin Rovinski, Trevor Mudge, Jason Mars, and Lingjia Tang. «Neurosurgeon: Collaborative Intelligence Between the Cloud and Mobile Edge». In: *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '17. Xi'an, China: Association for Computing Machinery, 2017, pp. 615–629. ISBN: 9781450344654. DOI: 10.1145/3037697.3037698. URL: https://doi.org/10.1145/3037697.3037698 (cit. on p. 6).

[22] Raghubir Singh and Sukhpal Singh Gill. «Edge AI: A survey». In: *Internet of Things and Cyber-Physical Systems* 3 (2023), pp. 71–92. ISSN: 2667-3452. DOI: https://doi.org/10.1016/j.iotcps.2023.02.004. URL: https://www.sciencedirect.com/science/article/pii/S2667345223000196 (cit. on p. 7).

[23] Daniel Crankshaw, Xin Wang, Giulio Zhou, Michael J. Franklin, Joseph E. Gonzalez, and Ion Stoica. *Clipper: A Low-Latency Online Prediction Serving System*. 2017. arXiv: 1612.03079 [cs.DC]. URL: https://arxiv.org/abs/1612.03079 (cit. on pp. 7, 9).

[24] Haichen Shen, Lequn Chen, Yuchen Jin, Liangyu Zhao, Bingyu Kong, Matthai Philipose, Arvind Krishnamurthy, and Ravi Sundaram. «Nexus: a GPU cluster engine for accelerating DNN-based video analysis». In: *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. SOSP '19. Huntsville, Ontario, Canada: Association for Computing Machinery, 2019, pp. 322–337. ISBN: 9781450368735. DOI: 10.1145/3341301.3359658. URL: https://doi.org/10.1145/3341301.3359658 (cit. on p. 7).

[25] Francisco Romero, Qian Li, Neeraja J. Yadwadkar, and Christos Kozyrakis. «INFaaS: Automated Model-less Inference Serving». In: *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. USENIX Association, July 2021, pp. 397–411. ISBN: 978-1-939133-23-6. URL: https://www.usenix.org/conference/atc21/presentation/romero (cit. on pp. 7, 9).

[26] Sohaib Ahmad, Hui Guan, Brian D. Friedman, Thomas Williams, Ramesh K. Sitaraman, and Thomas Woo. «Proteus: A High-Throughput Inference-Serving System with Accuracy Scaling». In: *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*. ASPLOS '24. La Jolla, CA, USA: Association for Computing Machinery, 2024, pp. 318–334. ISBN: 9798400703720. DOI: 10.1145/3617232.3624849. URL: https://doi.org/10.1145/3617232.3624849 (cit. on pp. 7, 9).

[27] Charles M. Stein, Dinei A. Rockenbach, Dalvan Griebler, Massimo Torquati, Gabriele Mencagli, Marco Danelutto, and Luiz G. Fernandes. «Latency-aware adaptive micro-batching techniques for streamed data compression on graphics processing units». In: *Concurrency and Computation: Practice and Experience* 33.11 (2021), e5786. DOI: https://doi.org/10.1002/cpe.5786. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/cpe.5786. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.5786 (cit. on p. 7).

[28] Andrea Goldsmith. *Wireless Communications*. Cambridge, UK: Cambridge University Press, 2005. ISBN: 978-0521837163 (cit. on pp. 8, 29).

[29] David Tse and Pramod Viswanath. *Fundamentals of Wireless Communication*. Cambridge, UK: Cambridge University Press, 2005. ISBN: 978-0521845274 (cit. on pp. 8, 29).

[30] Susan A. Slotnick. «Order acceptance and scheduling: A taxonomy and review». In: *European Journal of Operational Research* 212.1 (July 2011), pp. 1–11. DOI: None. URL: https://ideas.repec.org/a/eee/ejores/v212y2011i1p1-11.html (cit. on p. 8).

[31] N. Hall and C. Potts. «Supply Chain Scheduling: Batching and Delivery». In: *Operations Research* (2003) (cit. on p. 8).

[32] Kai Chen et al. *MMDetection: Open MMLab Detection Toolbox and Benchmark*. 2019. arXiv: 1906.07155 [cs.CV]. URL: https://arxiv.org/abs/1906.07155 (cit. on p. 9).

[33] NVIDIA. *Triton Inference Server: Dynamic Batching Guide*. Developer documentation. 2024. URL: https://docs.nvidia.com/deeplearning/triton-inference-server/user-guide/docs/introduction/index.html (visited on 09/11/2025) (cit. on p. 9).

[34] Tsung-Yi Lin, Michael Maire, Serge Belongie, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C. Lawrence Zitnick. «Microsoft COCO: Common Objects in Context». In: *Computer Vision – ECCV 2014*. Ed. by David Fleet, Tomas Pajdla, Bernt Schiele, and Tinne Tuytelaars. Cham: Springer International Publishing, 2014, pp. 740–755. ISBN: 978-3-319-10602-1 (cit. on p. 12).