UNIVERSITÉ SAVOIE MONT BLANC

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ SAVOIE MONT BLANC

Spécialité : **STIC Informatique**

Arrêté ministériel : 25 Mai 2016

Présentée par

Youssouph FAYE

Thèse dirigée par **Mohammad-reza SALAMATIAN** et
codirigée par **Francesco BRONZINO**

préparée au sein du **Laboratoire LISTIC**
dans **l'École Doctorale Sciences Ingénierie Environnement**

Distributed edge cloud architecture for executing AI based applications

Thèse soutenue publiquement le **18 décembre 2025**,
devant le jury composé de :

M., Pierre, SENS
Professeur à l'Université de Sorbonne, Rapporteur
M., Eddy, CARON
Professeur à l'Université Lyon 1, Rapporteur
Mme, Elisabeth, BRUNET
Maître de conférences à Telecom Sud Paris, Examinatrice
Mme, Sara, BOUCHENAK
Professeur à l'INSA Lyon, Examinatrice
M., Mohammad-reza, SALAMATIAN
Professeur à l'Université de Savoie Mont-Blanc, Directeur de thèse
M., Francesco, BRONZINO
Maître de conférences à l'ENS de Lyon, Co-directeur de thèse

Distributed edge cloud architecture for executing AI based applications

Youssouph FAYE

November 4, 2025

Université de Savoie Mont Blanc

Acknowlegments

Contents

List of Figures

List of Tables

Abstract

Edge computing has emerged as a transformative paradigm for real-time video analytics, offering low-latency processing by relocating computation closer to data sources. This shift is especially critical in domains such as public safety, traffic management, and autonomous systems, where responsiveness and bandwidth efficiency are crucial. Yet, edge environments are inherently resource-constrained and must contend with dynamic workloads, particularly when incorporating mobile cameras. These cameras provide valuable, timely perspectives but introduce unpredictability in data volume and scene composition, challenging traditional analytics pipelines and static workload distribution strategies. To address these constraints, this thesis presents VideoJam, a decentralized load balancing framework that predicts short-term workload fluctuations and redistributes video traffic across edge nodes, enabling adaptive, low-latency performance without centralized coordination.

As edge systems evolve to support increasingly diverse Artificial Intelligence (AI) tasks, a second challenge arises, one that lies not in data flow but in model deployment. Indeed, colocating multiple Deep Neural Networks (DNNs) on shared Graphics Processing Units (GPUs) has become a practical necessity, driven by the need to maximize limited computing resources while maintaining real-time inference. However, this necessity poses a fundamental challenge because when multiple models share the same hardware, their simultaneous execution can lead to competition for limited resources, resulting in slower performance and reduced reliability in meeting application requirements. Existing orchestration frameworks tend to adopt reactive or coarse-grained strategies, often neglecting the nuanced interactions that occur during model execution. To address this, this thesis introduces Roomie, a kernel-aware orchestration system that profiles and anticipates interference between co-deployed models, enabling insightful placement decisions that preserve throughput and ensure consistent performance in resource-constrained deployments.

Together, these contributions advance the design of scalable, adaptive edge architectures capable of supporting high-fidelity video analytics across heterogeneous and dynamic deployments.

Résumé

Le calcul en périphérie (edge computing) est apparu comme un paradigme transformateur pour l'analyse vidéo en temps réel, offrant un traitement à faible latence en déplaçant les calculs plus près des sources de données. Ce changement est particulièrement critique dans des domaines tels que la sécurité publique, la gestion du trafic et les systèmes autonomes, où la réactivité et l'efficacité de la bande passante sont cruciales. Cependant, les environnements de calcul en périphérie sont intrinsèquement contraints en ressources et doivent faire face à des charges de travail dynamiques, en particulier lorsqu'ils intègrent des caméras mobiles. Ces caméras fournissent des perspectives précieuses et opportunes, mais introduisent une imprévisibilité dans le volume de données et les éléments visuels présents dans les scènes filmées, ce qui remet en question les pipelines (chaines de traitement) d'analyse traditionnels et les stratégies de distribution de charge de travail statiques. Pour relever ces contraintes, cette thèse présente VideoJam, une solution d'équilibrage de charge décentralisé qui prédit les fluctuations de charge de travail à court terme et redistribue le trafic vidéo entre les noeuds de calcul, permettant des performances adaptatives à faible latence sans coordination centralisée.

Alors que les systèmes de calcul en périphérie évoluent pour prendre en charge des tâches d'intelligence artificielle de plus en plus diverses, un deuxième défi se pose, qui ne réside pas dans le flux de données mais dans le déploiement de modèles. En effet, la co-localisation de plusieurs réseaux de neurones profonds (DNN) sur des accélérateurs de calcul (GPU) partagés est désormais une pratique courante, motivée par la nécessité de maximiser les ressources de calcul limitées tout en maintenant une inférence en temps réel. Cependant, cette nécessité pose un défi fondamental car lorsque plusieurs modèles partagent les mêmes resources, leur exécution simultanée peut entraîner une concurrence pour des ressources limitées, entraînant une performance plus lente et une fiabilité réduite pour répondre aux exigences des applications. Les cadres d'orchestration existants ont tendance à adopter des stratégies réactives ou à gros grains, souvent en négligeant les interactions nuancées qui se produisent pendant l'exécution du modèle. Pour résoudre ce problème, cette thèse introduit Roomie, un système d'orchestration conscient du noyau qui profile et anticipe les interférences entre les modèles déployés conjointement, permettant des décisions de placement éclairées qui préservent le débit et assurent des performances cohérentes dans les déploiements contraints en ressources.

Ensemble, ces contributions avancent la conception d'architectures de calcul en périphérie scalables et adaptatives capables de prendre en charge une analyse vidéo haute fidélité sur des déploiements hétérogènes et dynamiques.

Introduction 1

1.1 Motivation

Over the past decade, the widespread availability of data from various sources, including camera streams, has become a ubiquitous feature in urban environments and beyond [1, 2]. This data is increasingly being used to gather valuable insights, contributing to improved public safety, operational efficiency, and decision-making in a variety of fields. This wealth of information not only supports surveillance activities, but also plays a critical role in traffic management, crime prevention, urban planning, and other critical applications [3–6]. By leveraging this data, local authorities and organizations can make informed decisions that improve quality of life, safety, and optimize resource allocation. On the other hand, the rapid transformation of AI has made it a powerful tool that profoundly influences industries and societies. AI plays a crucial role in analyzing and interpreting these large data sets, facilitating a variety of applications such as video surveillance, medical diagnostics, autonomous systems, and real-time analysis. This expansion of AI applications requires infrastructure capable of efficiently and rapidly processing large computational workloads, while offering innovative solutions to contemporary challenges.

As video data becomes increasingly central to AI-driven applications, live video analytics plays a pivotal role in extracting real-time insights from visual streams [3, 4]. Live video analytics involves the real-time processing of camera feeds using algorithmic and computer vision techniques to derive actionable intelligence. These streams typically pass through a structured pipeline, often modeled as a directed acyclic graph, comprising sequential functions such as object detection, classification, and tracking [7–10]. Video analytics has become a fundamental part of intelligent automation, enabling systems to interpret and respond to visual data in real-time. Its applications span a wide range of areas, including automated traffic management, where it facilitates the detection of traffic jams and accidents, as well as surveillance, crowd control, and assisted and autonomous driving, *etc.* By leveraging the growing volume of video data and the insights it provides, video analytics enables individuals and organizations to make more informed, faster, and more strategic decisions.

Traditionally, video analytics has relied heavily on cloud-based processing, where raw video streams are transmitted to centralized data centers for analysis [11–13]. However, this approach introduces significant limitations. The massive volume of video data generated by high-resolution cameras places immense strain on network bandwidth, leading to latency issues and potential bottlenecks [9, 14]. These delays are especially problematic in time-sensitive scenarios such as emergency response, industrial safety, or autonomous navigation, where milliseconds matter.

To overcome these challenges, the field is increasingly shifting toward edge computing, processing data locally at or near the source [7, 8, 15].

This approach reduces latency, conserves bandwidth, and enhances the responsiveness of video analytics systems. Yet, edge computing introduces its own set of constraints. Despite the presence of advanced GPUs, edge devices have limited computational capacity and can quickly become overwhelmed by high-volume video streams, leading to data loss and degraded performance.

This thesis addresses two critical challenges in the design of scalable and efficient video analytics systems, each representing a distinct contribution to the field.

First, it explores the integration of mobile cameras with distributed edge computing architectures. Mobile cameras offer flexibility and dynamic coverage; however, their unpredictable nature, as they can appear and disappear at any time, requires efficient data processing to avoid longer response times. This work proposes a new approach strategy for distributing video workloads across multiple edge nodes to balance computational demands, reduce latency, and improve overall system responsiveness. By optimizing data flow and task allocation, the proposed approach enhances throughput while maintaining low-latency performance a key requirement for real-time analytics.

Secondly, this research examines the deployment of DNN models in resource-constrained environments, such as edge devices. In these settings, the co-location of multiple computing tasks is often required; however, if not managed effectively, this can result in performance degradation and reduced model accuracy. This study offers new insights into the efficient placement and execution of DNN components within these environments, ensuring that analytics pipelines maintain both accuracy and responsiveness despite stringent resource limitations.

In summary, this thesis focuses on the integration of mobile camera systems with intelligent workload distribution strategies oriented towards the periphery and the optimization of DNN deployment in resource-constrained environments. This work contributes to the advancement of scalable, highly accurate, and low-latency video analysis systems. It addresses the placement problem, namely the challenge of determining the optimal deployment locations for DNN functions in video analysis pipelines, taking into account resource constraints and heterogeneous camera workloads.

1.2 Challenges in Incorporating Mobile Cameras

Mobile cameras offer a dynamic and flexible viewpoint for video analytics, capturing scenes that fixed cameras might miss due to their static nature. However, this mobility poses significant challenges for video analytics architectures, especially when both types of cameras coexist in a deployment. The first major challenge lies in the heterogeneous performance profiles of processing pipelines. Fixed cameras benefit from stable backgrounds, which allow for effective object isolation using background subtraction. Mobile cameras, on the other hand, constantly change scenes, making background subtraction unreliable and often leading to false detections. To address this, mobile camera pipelines often

rely on single-pass object recognition models such as YOLO, which, while more consistent, are also more resource-intensive. This divergence in processing requirements underscores the importance of tailoring analysis pipelines to the specific camera type.

The second challenge involves managing highly variable workloads. Traditional workload balancing strategies rely on predictable patterns based on camera location and time of day. Distream [7]'s architecture attempts to optimize workload distribution using cross-device balancing and adaptive controllers. However, mobile cameras disrupt these assumptions by introducing unpredictable scene changes and intermittent presence in the network, making long-term prediction models ineffective. The third challenge centers on deployment configuration flexibility. Fixed camera setups typically allow for slow, deliberate changes based on long-term usage patterns. Mobile cameras, however, demand real-time adaptability, as they can join or leave the network at any moment. This necessitates an online load balancing system capable of reconfiguring processing pipelines within seconds, without requiring full system reboots. Together, these challenges highlight the need for a fundamentally more agile and responsive video analytics architecture.

1.3 Deployment Challenges on Edge-Cloud Devices

Optimizing resource utilization in edge computing environments, such as mobile platforms, low-power nodes, or remote devices, requires a modular approach to AI deployment. Decoupling the processing pipeline into separate functions (*e.g.*, background subtraction, object detection, tracking, and classification), as in live video analytics, allows each stage to process data independently and pass results downstream, providing greater flexibility and efficiency. However, due to the limited computing and energy resources available in these environments, colocating multiple AI models on shared hardware often becomes a necessity to meet real-time performance requirements and reduce deployment costs. This pragmatic choice, while effective in theory, introduces a series of challenges: resource contention, interference between models, and degradation of quality of service metrics. Existing solutions often overlook these dynamics, deploying entire pipelines on single devices or distributing them without considering the negative interactions between co-deployed models. To mitigate these risks and ensure fair, predictable, and high-performance AI inference, sophisticated planning and optimization strategies are essential.

A fundamental first step in this process is to recognize and measure interference between co-deployed models before assigning them to a target GPU resource. By doing so, we can strategically deploy functions across available resources, maximizing the efficiency of edge devices. This requires a good understanding of how the GPU execution model works during inference. Such an understanding hinges on grasping the internal mechanics of GPU architecture. Designed to manage intensive workloads, GPUs rely on a highly parallel structure made up of multiple Streaming Multiprocessors (SMs), each with its own set of processing

cores, registers, and shared memory. This configuration enables GPUs to excel at the training and inference tasks central to Machine Learning (ML) and DNN.

Frameworks like Compute Unified Device Architecture (CUDA) and Open Computing Language (OpenCL) have enhanced GPU utility by offering granular control over kernel execution, memory access, and thread parallelism. CUDA, in particular, has become the standard for programming NVIDIA GPUs, enabling researchers and engineers to utilize low-level optimization capabilities and achieve breakthrough AI performance. However, understanding how kernels are scheduled and executed in the GPU during inference is crucial. Once deployed, each model will execute several kernels successively, where each kernel requires a certain amount of register allocation, shared memory, and warps to run efficiently. This low-level view complements high-level planning strategies, enabling smarter placement of models, particularly in edge environments where resource constraints necessitate the co-deployment of DNNs.

1.4 Contributions of this Thesis

This thesis aims to introduce and design an innovative architecture tailored to the aforementioned challenges associated with deploying video analytics applications in resource-constrained environments. The proposed framework will establish efficient methods for distributing and managing processing pipelines to adapt to fluctuating workloads. By optimizing resource efficiency and streamlining task distribution, we will be able to improve the overall performance and reliability of video analytics systems in varied and constantly evolving scenarios.

This thesis presents several significant contributions to the fields of edge computing and video analytics:

Self-Balancing Architecture for Live Video Analytics. VideoJam implements a distributed load balancing system by deploying load balancers colocated with each task in the analytics pipeline. Each load balancer continuously monitors the incoming flow of frames or objects and periodically shares this information with neighboring load balancers. Based on the collected data, each load balancer independently decides how much traffic to process locally and whether to offload some of its workload to less-burdened neighbors using a lightweight ML model that predicts the incoming workload for each processing component and its neighbors in the near future. Additionally, the load balancers employ a congestion prevention signaling system to correct any prediction errors. VideoJam operates autonomously, adapting dynamically to changes such as new camera arrivals or departures without requiring system reboots, and balances incoming traffic accordingly. Our approach uniquely combines horizontal distribution and per-function type load balancing, driven by short-term forecasts of incoming loads.

Efficient Model Cohabitation in Edge Computing Model Serving. An orchestration architecture designed to maximize system performance in scenarios where model colocation is necessary, particularly in resource-constrained edge environments. The key innovation of Roomie is its kernel-aware interference profiling, which captures the sequential nature of GPU kernel execution patterns when multiple models share hardware resources. By analyzing how specific kernel sequences from different models interact, Roomie constructs accurate interference profiles that predict performance degradation under various colocation scenarios. This detailed approach allows Roomie to make optimal placement decisions, determining which models can efficiently coexist on the same hardware and which combinations should be avoided to ensure performance guarantees.

Building upon these contributions, the next chapter (Chapter 2) will explore the foundational background and relevant literature that inform our research. This contextual groundwork is essential for grasping the significance of our innovations and the methodological choices made throughout the thesis, thereby enabling a holistic understanding of our strategy for optimizing edge computing and video analytics.

Related work 2

This section reviews the latest research and advances in edge cloud computing, focusing on three main themes: video analytics, workload distribution, and AI application scheduling. We begin by defining cloud computing and the motivations for adopting edge-based solutions, particularly in the field of video analytics. The following sections explore deeper into research on live video analytics, followed by a detailed discussion on workload balancing (Section 2.2) and, finally, query processing, scheduling, and model placement strategies (Section 2.3).

2.1 Edge Computing and Video Analytics

Cloud computing is a model that uses centralized data centers to provide scalable storage and processing capabilities on a large scale over the Internet. Although this model has facilitated significant digital transformation, it presents several challenges for latency-sensitive and bandwidth-intensive applications, for emerging real-time applications.

The increasing need for real-time responsiveness, data privacy, and network efficiency has prompted a move toward edge computing, where processing occurs closer to the data source. This shift is especially beneficial for video analytics, which frequently relies on continuous, high-volume data streams from distributed sensors and cameras. This architectural transformation addresses several limitations inherent in centralized cloud infrastructures, notably by reducing latency, minimizing bandwidth consumption, enhancing data security, and enabling real-time responsiveness. Given that edge devices, such as sensors, cameras, and Internet of Things (IoT) nodes, typically operate under constrained computational and energy resources, a variety of architectural models have emerged. These include hybrid edge-cloud systems, which balance local and remote processing, as well as fully autonomous edge analytics platforms designed for domain-specific applications, particularly within industrial IoT environments.

Edge computing applications. Edge computing has emerged as a transformative technology across diverse sectors, enhancing operational efficiency and enabling real-time decision-making. In smart agriculture, it facilitates precision farming through sensor, drone-based crop monitoring, promoting optimal resource allocation and early disease detection [16–18]. For instance, AI can significantly improve efficiency in the following areas: crop cultivation, yield forecasting, disease monitoring, supply chain operations, food waste reduction, and resource management [19]. Urban areas reap considerable benefits from edge computing systems that manage traffic, pollution, temperature rises, and security, *etc.* These systems use real-time sensor data to reduce traffic congestion and improve public safety through proactive monitoring and response [5,

6]. Edge computing in industrial environments enables predictive maintenance by monitoring machines in real time to prevent breakdowns, while promoting seamless human-robot collaboration for safer and more efficient operations. These innovations are driving the vision of Industry 5.0, which advocates for smart, human-centered manufacturing [20]. In healthcare, wearable devices enable continuous monitoring of vital signs, allowing prompt medical responses to anomalies [21]. The overview highlights the crucial role of edge computing in improving responsiveness, accuracy, and strategic decision-making across various industries and applications.

Video Analytics as a Critical Edge Application. Among the diverse applications of edge computing, video analytics emerges as a particularly demanding and impactful use case. In domains such as smart cities, autonomous driving, and public safety, edge-based video analytics enables the timely interpretation of visual data [3, 4]. A typical video analytics pipeline comprises several stages: data ingestion, preprocessing, feature extraction, model inference, and post-processing. Each stage imposes distinct computational and memory requirements, which are often challenging to satisfy within the limited capabilities of edge devices.

Resource Constraints and Architectural Challenges. The computational limitations of edge devices, manifested in restricted processing power, limited storage capacity, and constrained communication bandwidth, pose significant challenges to the deployment of complex video analytics workloads [8, 22–24]. These constraints necessitate the development of innovative strategies aimed at optimizing resource utilization while preserving the real-time performance and locality benefits of edge computing. The authors in [23], proposes a new edge computing scheme using lightweight object recognition and adaptive transmission, which improves road traffic monitoring by significantly reducing data transmission while maintaining effective object identification. Chameleon [8] optimizes resource efficiency by adapting configurations based on temporal and spatial correlations in video data, significantly reducing resource consumption while maintaining effective accuracy. Similarly, Spatula [25] exploits spatial and temporal correlations among camera feeds to lower network and computation costs. While the article in [24] presents bandwidth-saving strategies for real-time drone video analysis using advanced computing and DNNs, demonstrating effective transmission reduction with minimal impact on accuracy and latency.
On top of these solutions for dealing with edge computing challenges, other studies have suggested different ways to improve performance while keeping its benefits. Techniques such as model compression and pruning have been employed to reduce the computational footprint of deep learning algorithms, thereby enabling their execution on resource-constrained devices [26]. Distributed processing frameworks facilitate the partitioning and collaborative execution of workloads across multiple edge nodes, improving scalability and fault tolerance [7, 10, 27]. Furthermore, adaptive scheduling algorithms and energy-aware resource management strategies have been developed to dynamically allocate tasks and balance performance with sustainability. Collectively, these approaches contribute to the realization of robust, efficient, and scalable

edge computing systems capable of supporting increasingly complex and latency-sensitive applications.

2.2 Load Balancing for Live Video Analytics

Recent advancements in video analytics have led to the development of various techniques aimed at enhancing application performance. These efforts span multiple dimensions, including architectural design, pipeline optimization, and data privacy. As the demand for real-time, scalable video processing grows, particularly in edge environments, thus efficient resource management and adaptability have become central challenges.

Deployment Strategies. To overcome the limitations of static architectures, newer frameworks have adopted distributed deployment models. Distream [7] exemplifies this shift by partitioning analytics pipelines between smart cameras and edge nodes, adapting to workload dynamics to maintain low latency and high throughput. The work presented in [27] shows that the intelligent distribution and processing of vision modules in parallel on available peripheral computing nodes can ultimately enable better use of resources and improve performance. Likewise, VideoStorm [10] which places different video functions across multiple available workers to satisfy users' requests. These strategies offer greater flexibility and scalability, making them more suitable for dynamic environments.

Load Balancing in Edge Video Analytics. As video analytics applications scale across distributed infrastructures, load balancing becomes essential for maintaining accuracy and low latency. Traditional methods relied on static configurations and centralized cloud offloading [9, 10, 15, 25]. VideoStorm [10] offers a system that uses approximation and delay tolerance to balance the load across the distributed system. It dynamically adapts the workload based on available resources and simultaneous network conditions to ensure effective load balancing. Alternatively, Spatula [25] utilizes spatial and temporal correlations between cameras to improve load balancing in large-scale camera networks. By selectively analyzing only the cameras and images most likely to capture the target identity (*e.g.*, people or vehicles) as they pass through different viewpoints, Spatula efficiently distributes the computing workload across the network. In contrast, VideoEdge [9] optimizes workload distribution across a hierarchy of clusters to maximize accuracy while balancing network and computing resource demands. Hetero-edge [15] uses Apache Storm and breaks edge applications into DAG-based tasks and map them across heterogeneous servers (CPUs, GPUs) for efficient execution. By intelligently distributing tasks based on server capabilities, it achieves efficient load balancing and significantly reduces latency for real-time vision applications.
Recent developments have introduced more dynamic load balancing mechanisms. Distream [7] introduces a distributed video analytics system that dynamically partitions tasks between smart cameras and edge clusters, enabling real-time scalability and responsiveness. Its pipeline-level

load balancing adjusts to fluctuating video content, delivering lower latency and higher throughput than conventional approaches.

Workload Prediction. To enhance load balancing, predictive models based on ML have been explored [7, 28, 29]. Distream [7] employs an LSTM-based workload predictor that anticipates future load spikes, thereby avoiding inefficient task migrations and maintaining a balanced workload distribution. Reinforcement learning, as applied in [28], supports real-time task assignment and shows promising results, though it typically involves considerable computational effort and ongoing training to manage evolving data patterns. Linear regression models, as used in [29], offer a simple and efficient way to forecast incoming tasks based on recent trends, making them a practical choice for scenarios with relatively stable scene dynamics. Given the constraints of peripheral devices and the unpredictability of data provided by mobile cameras, there is a growing need for lightweight prediction models that can predict short-term trends, run efficiently on resource-limited hardware, and support real-time decision-making.

The evolution of video analytics has moved from static, centralized systems to dynamic, distributed architectures. While significant progress has been made in deployment and load balancing strategies, future research must focus on developing lightweight, adaptive forecasting techniques. These solutions should be tailored to the constraints of edge environments and the variability introduced by mobile video sources, ensuring scalable and responsive video analytics.

2.3 Inference Serving and Resource Management in Edge Cloud Computing

With the rise of deep learning applications deployed as online services, efficiently managing inference workloads in GPU datacenters has become a pressing concern. Unlike training tasks, inference jobs present unique constraints, requiring high accuracy, low latency, and cost-effectiveness. These goals often conflict, making it essential to design scheduling systems that can balance trade-offs without compromising overall performance.

Inference Serving Systems. In the field of efficient resource allocation for ML inference systems, several studies have proposed relevant solutions. Clipper [30] proposes an inference service system designed for real-time applications, supporting a variety of ML frameworks and models to simplify their deployment and commissioning. TensorFlow-Serving [31], meanwhile, enables real-time model serving by dynamically adjusting the number of replicas based on traffic, promoting continuous adaptation to load variations. Clockwork [32] offers an approach focused on performance predictability, exploiting the deterministic nature of full GPU capacity DNN inference and orchestrating inference execution with data loading via separate CUDA streams. Proteus [33], a more recent work, introduces a high-throughput inference-serving system that dynamically scales model accuracy by optimizing model selection,

placement, and query distribution.
While these designs effectively optimize throughput and accuracy, their reliance on rigid mappings between models and devices tends to overlook opportunities for co-location and parallel execution, highlighting the need for more flexible placement strategies that better leverage shared GPU resources.

Multi-Tenant DNN Inference on Shared GPUs. A subsequent line of research investigates techniques for optimizing model performance in environments characterized by concurrent execution, where multiple models or processes must contend for shared GPU resources [34–37]. This body of work typically examines scheduling algorithms, resource allocation strategies, and system-level adaptations designed to maintain throughput and minimize latency under constrained computational conditions. For example, INFaaS [37] develops a model-less inference system that automates variant selection and resource allocation to achieve performance and accuracy targets, although it faces scalability challenges in profiling and predicting interference. Colti [34] offers a technique that improves the efficiency of training and inference tasks by colocating DNNs on GPUs, thereby increasing overall throughput and reducing execution times. Another work in [35], leverage the fact that DNN inference consists of a series of operators (convolution, dense, *etc.*) and exploits their independence between different DNNs to schedule their execution concurrently, assigning each model to a dedicated stream, splitting operator sequences into shorter steps, and orchestrating their execution to balance resources and minimize latency. REEF [38], a DNN inference serving system that enables efficient concurrent execution and kernel preemption on GPUs, allowing multiple models with different priorities to share resources while minimizing latency and maintaining predictable performance. Miriam [36], in the other hand, introduces a framework for GPUs that enables the simultaneous execution of DNN tasks with varying real-time requirements, using elastic kernels, which are smaller, more flexible units that can be dynamically scheduled and reassigned according to their criticality and priority. Overall, these approaches focus on post-deployment optimization of models sharing the same GPU, without addressing the issue of optimal initial placement or co-location strategy. Other work stands out by directly addressing this deployment phase, integrating co-location and inter-GPU planning strategies from the outset to anticipate interference and improve overall efficiency.

Interference-Aware Inference Serving. Recent studies have focused on proactive scheduling by modeling and predicting interference between concurrently running models. Some approaches, such as the work in [39], estimate latency degradation using coarse features like model utilization of global buffer and PCIE connection, which provides a unified way to predict latency degradation across platforms. In contrast, Scrooge [40] proposes profiling to find the optimal level of competition among identical DNNs in order to maximize throughput. This leads to an important conclusion: although effective for homogeneous configurations, this approach highlights the difficulty of scaling profiling across various combinations of different DNNs sharing resources. Operator-level scheduling

frameworks, like Abacus [41] addresses DNN interference by scheduling operators (*e.g.*, Convolution, ReLU) from multiple models to run concurrently. It treats each DNN as a sequence of operators and groups them for joint execution, aiming to maintain quality-of-service guarantees. This operator-level scheduling simplifies coordination and enables deterministic execution. More detailed analysis and low level consideration for DNN scheduling on GPUs, as iGniter [42], employs hardware-level metrics to characterize interference, but may not fully capture nuances of kernel launches and core scheduling. A lower-level GPU details like kernels, such as Usher [43], analyze achieved kernel occupancy and Dynamic Random-Access Memory (DRAM) usage during DNN inference and introduce model classification schemes, providing more accurate insights into interference. Further work is needed to develop more accurate and widely applicable interference-aware scheduling strategies that can operate in diverse environments, including edge computing scenarios, with a deeper understanding of their limitations.

VideoJam: Self-Balancing Architecture for Live Video Analytics

3

3.1 Introduction

Video camera flows are a pervasive source of information. Cities are increasingly deploying closed-circuit cameras that are used for safety, security, and traffic control applications [44–46]. To process the incoming videos streams in real-time, live video analytics architectures have been proposed to support a multitude of applications across navigation, safety, and control [7, 8, 11, 12, 47]. However, the increasing amounts of video data produced by available cameras has forced the community to move away from centralized cloud-based architectures [47, 48].

Edge-based live video analytics are a promising approach to reduce bandwidth overheads caused by the transmission of raw video streams to centralized clouds [3]. Unfortunately, in contrast with the quasi-infinite resources of data-centers, edge devices are often co-located with existing network equipment and deploy limited computational resources. As a result, they can rapidly become overloaded by the incoming video frames, causing data loss and reduced accuracy [8, 25]. To address this challenge, various solutions have been proposed to distribute the workload across locations, ranging from vertically splitting the processing between edge devices and the central cloud, wherein excess traffic is offloaded to the cloud [9, 14] to horizontally across different edge clusters, exploiting the dynamicity of processing requests incurred at each location [15, 25]. However, all these architectures assume that video flows are generated from fixed cameras (*e.g.*, traffic cameras deployed at street corners) and that their workflows present predictable patterns [8].

With the rising penetration of mobile cameras, an opportunity emerges to include these cameras in the design of video analytics architectures. Mobile devices, such as cars or drones, come equipped with high-quality camera sensors that have been demonstrated to be very effective for navigation, safety, and control applications [24, 46]. These cameras often have the unique advantage of being *in the right place at the right time*. Considering in particular that mobile cameras are generally owned by private individuals (*e.g.*, on-board car cameras), there is evidence of the growing use of these camera streams for personal and societal security [49, 50]. Integrating them into existing architectures will support analytics applications that interest camera owners, such as accident detection, crash analytics, and reconstruction. However, the scenes that mobile cameras capture vary more rapidly than for fixed cameras.

Owing to the high mobility and dynamic content of mobile cameras, we identify three key challenges in accommodating these cameras in video analytics architectures. First, the workload generated by mobile cameras is more dynamic and unpredictable, requiring constant adjustments to the processing infrastructure. Existing solutions such as Distream [7] recognize the need for adaptation to varying processing loads, but ultimately fall short of developing a solution that adjusts at the rate imposed by mobile cameras (we expose such problem in detail in Section 3.6).

Second, the continuously changing scenes captured by mobile cameras make customary processing pipelines ineffective. For example, a typical pipeline used to process a video feed incoming from a fixed camera might employ a lightweight background subtractor to isolate moving objects, reducing the need for deploying more expensive object detection modules [7]. Finally, as mobile cameras appear and disappear from the deployment, they generate constant changes in the deployment configuration and the number of sources to process. This introduces the need to deploy different processing pipelines for different cameras. Overall, existing video analytics architectures are ill equipped to handle such video traffic.

In this chapter, we tackle the challenge of designing a video analytics architecture capable of handling a mix of fixed and mobile video camera flows. We build on the recent idea [7] that the dynamicity of the workload generated by video cameras can be exploited to balance the load across different edge clusters. However, we observe that: (i) While highly dynamic, the workload generated by mobile cameras is not completely unpredictable. In fact, the scenes captured by mobile cameras reveal sufficient patterns to make it possible to predict the amount of objects that will need to be processed in the next few seconds. (ii) Enabling the coexistence of diverse video analytics pipelines requires dividing the processing problem into smaller components that function independently, even when pipelines later converge to the same set of modules. (iii) The constant changes in the deployment configuration, such as the addition or removal of processing components, require an online approach to load balancing that can adapt to these changes without requiring a complete reboot of the processing pipelines.

We leverage these observations to build VideoJam, a load balancing solution for live video analytics. VideoJam deploys a set of load balancers co-located with every task in the analytics pipeline. Each load balancer monitors the incoming flow of frames or objects to process and periodically shares this information with its neighbors. Based on the information collected, the load balancers take independent decisions on how much traffic to process locally and whether to offload some of their workload to less-loaded neighbors. To make these decisions, VideoJam uses a lightweight ML model to predict the incoming workload for each processing component for the near future, as well as for its neighbors. Finally, the load balancers recover to eventual prediction errors via a congestion prevention signalling system. VideoJam operates independently of deployed configurations and dynamically adapts to handle eventual changes (*e.g.*, new camera arrivals or departures) without requiring hard reboots balancing incoming traffic accordingly. Our approach features a unique combination of horizontal distribution and per-function type load balancing, powered by short-term forecasts of incoming loads.

We implement and evaluate VideoJam using a typical vehicular safety application, *i.e.*, vehicle plate detection, adapted to both fixed and mobile cameras. Our evaluation shows that VideoJam can achieve a 2.91× lower response time and reduce 4.64× video frames loss than a state-of-the-art approach [7], while also reducing network overheads. Further, we demonstrate the ability of VideoJam to dynamically adapt to changes in the deployment configuration, *i.e.*, change in number of function replicas or number of cameras in the system, without requiring any hard reboots.

VideoJam adapts to the new configuration in less than 28 seconds (*i.e.*, +27% the failure time), to the incurred change. We release VideoJam as open source software for the community to use and extend [1].

1: The VideoJam implementation code is freely available at https://github.com/ENSL-NS/VideoJam.git

3.2 Background and Motivation

In this section we present the characteristics of video analytics pipelines, providing motivation for integrating mobile video sources into the analytics architecture. We then present the key differences between fixed and mobile cameras, highlighting the challenges that we face when we integrate different types of cameras into such systems.

3.2.1 Live Video Analytics

Live video analytics center around analyzing video camera streams in real-time, using algorithmic and computer vision techniques to extract valuable information. Incoming frames traverse a series of modules (or *functions*) that perform different tasks, such as object detection, classification, and tracking. These functions are combined into a pipeline, conventionally represented by a directed acyclic graph, where the output of one function is the input of the next and the final output depends on the analytics application deployed. Figure 3.1 shows an example of video analytics pipelines for a typical traffic control application: vehicles' number plate detection.

Historically, video analytics research has focused its attention on the placement problem occurring at deployment time. Deploying a video analytics application involves taking an orchestration decision of where to place the instances of the application's pipeline. Placement decisions are taken based on the available resources in the compute infrastructure and the workload generated by available cameras. Early work on video analytics focused on how to efficiently transmit video traffic to centralized clouds for processing [11–13]. Yet, while centralized datacenters offer unbounded compute resources, transporting the increasing amounts of video streams to these locations can result in network bottlenecks, requiring either to preprocess video frames on premises [9, 14] or to reduce the quality of the video transported [51], potentially affecting the performance of deployed applications.

To tackle these challenges, recent work has focused on deploying video analytics pipelines at the edge of the network [7, 8, 15]. These approaches aim to take advantage of compute resources deployed close to video cameras and bypass the transmission to remote locations. However, edge compute devices are often co-located with the existing network equipment and deploy limited computational resources. As a result, they can rapidly become overloaded by incoming video frames causing data loss and reduced accuracy. To address this challenge, different solutions have been proposed to distribute the workload across locations. For example, Distream [7] exploits the inherent load dynamics present in video flows to split the processing pipeline between two hierarchical locations, *i.e.*, the camera and the edge compute machine. Chameleon [8]

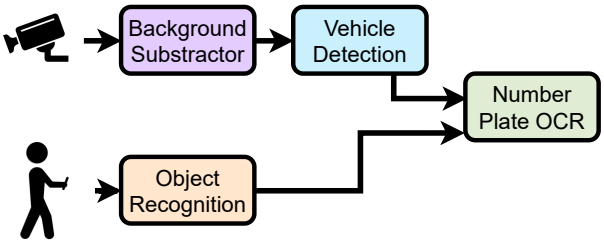
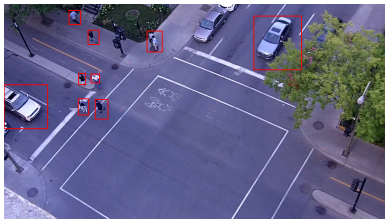
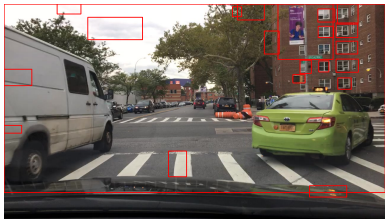


Figure 3.1: The functions composing a vehicles' number plate detection application pipeline. Different sources require different processing functions.



(a) Fixed camera with background subtractor.



(b) Mobile camera with background subtractor.

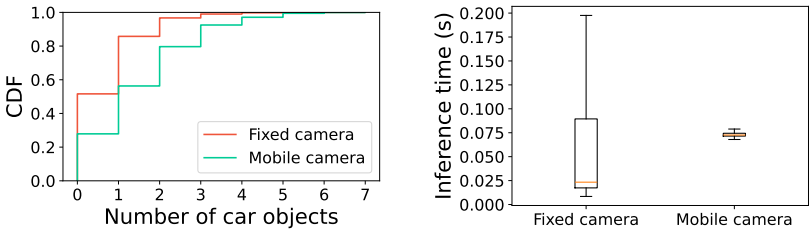
Figure 3.2: A comparative example on the performance of background substractor functions on fixed and mobile cameras.

instead optimizes the pipelines configuration based on temporal and spatial predictions on the contents of camera sources.

Yet, all these architectures assume that video flows are generated from fixed cameras (*e.g.*, traffic cameras deployed at street corners) and that their workflows present predictable patterns allowing reconfiguration decisions to be taken at longer timescales, for example when traffic conditions change during the day due to commute patterns. However, mobile cameras have become pervasive in the last decade. Mobile devices, from smartphones to cars and drones all come equipped with high quality camera sensors. These cameras often have the unique advantage of being in the right place at the right time, offering the potential to enhance existing architectures and improve application performance. Unfortunately, video feeds generated by mobile cameras are fundamentally different from fixed camera ones, posing unique challenges into the path for their integration.

3.2.2 Challenges in Incorporating Mobile Cameras

Mobile cameras bring the advantage of providing a unique perspective on the captured scenes. They can be in the right place at the right time, capturing scenes that would be otherwise invisible from fixed cameras. However, this benefit comes at the implicit cost of having to handle fundamentally different dynamics. Mobile cameras are constantly moving, capturing new scenes; they can appear and disappear from a deployment; and the scenes they capture can vary more rapidly than for fixed cameras. For these reasons, the differences between fixed and mobile cameras can greatly impact the deployment choices of a video analytics pipeline architecture that aims to process their video feeds. With the goal of designing a video analytics architecture that can handle both fixed and mobile cameras, we identify three main challenges that arise from the coexistence of these cameras in the same deployment.



(a) Number of detected vehicles.

(b) Inference time.

Figure 3.3: Vehicle detection performance for fixed (background substractor + detection) vs mobile (YOLOv5) cameras.

Challenge #1: Heterogeneous performance profiles. The majority of existing video analytics solutions [8] assume the homogeneity in the performance of the processing components deployed. Distream [7] relaxes this assumption by considering the presence of heterogeneous processing devices and accounts for this disparity in implementing load balancing policies within its architecture. However, mobile cameras are inherently in constant movement, quickly capturing new scenes. This makes the processing pipelines that are effective for fixed cameras ineffective for mobile cameras.

To exemplify the difference between fixed and mobile cameras, we consider a vehicles' number plate detection application. The goal of this application consists of detecting vehicles from a camera frame using a vehicle detection module and extract their number plate via an Optical Character Recognition (OCR) module. To lighten the load of the processing pipeline, the first step involves isolating objects in the frame to limit vehicle detection executions solely on cropped images [7]. This is typically done using a background subtractor module that compares the current frame with a background model and outputs a mask of the foreground objects [7]. Figure 3.2a shows the output of a background subtractor module applied to a fixed camera frame. We observe that the background subtractor is able to isolate the moving objects in the frame while still objects (*e.g.*, parked cars) are not detected as they are still in the frame. Unfortunately, while the application of a background subtractor is effective for fixed camera streams, the same pipeline becomes ineffective for mobile cameras. By nature, mobile cameras are constantly moving, capturing new scenes. As a consequence, when applied to a moving subject, the subtractor model is not capable of adapting to new scenes, causing erroneous detections or, in the worst case, detecting the entire frame as foreground (as shown in Figure 3.2b).

To compensate for the degraded performance of the background substractor module, existing solutions tailored for mobile cameras [52, 53] replace the early stages of the pipeline with an object recognition module (*e.g.*, YOLO [54]) that is capable of detecting and classifying objects in the frame in a single operation. However, this comes at a price, as detection models are more resource-intensive, especially for scenarios where no object is in the frame. Figure 3.3 shows the performance difference between the two approaches on two selected videos, a fixed and mobile one. We observe that, while the number of detected vehicles is relatively similar across videos, the characteristics of the inference times for the two approaches highly varies depending on the processed frame. In fact, while YOLO achieves consistent results due to its single-pass nature, the performance of the combination of background subtractor and vehicle detection varies according to the number of vehicles present in the frame.

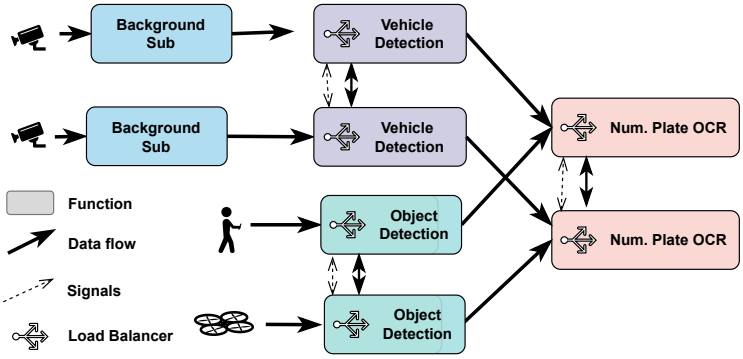


Figure 3.4: VideoJam architecture.

Ultimately, this leads to the conclusion that the two approaches should not be treated as interchangeable and that the choice of the pipeline to deploy should be tailored to the video source type.

Challenge #2: Highly variable workloads. Previous work has highlighted how video camera feeds differ in the amount of objects of interest they capture depending on their deployment location (*e.g.*, a building entrance vs. an emergency exit) and the time of capture (*e.g.*, at night vs during the day) [55, 56]. These differences generate variability in the workload that the video analytics pipeline needs to process and have been exploited to design more efficient processing pipelines [7, 8]. Distream [7] proposed to exploit this variability to dynamically balance the workloads across processing clusters, taking advantage of periods of lower usage from some nodes in the architecture to support overloaded ones. Their solution achieves this through the use of two main design elements: (i) A cross-device workload balancer that takes the cross-camera workload correlations and the heterogeneous compute capabilities of smart cameras and edge clusters to optimize cross-camera workload balancing via an optimization problem; and (ii) a workload adaption controller which triggers the cross-camera workload balancer when cross-camera workload imbalance is detected.

Unfortunately, this approach assumes that workloads have predictable profiles, either due to the cameras' relative locations, their time of capture [8], or, more generally, by training a prediction model based on previous patterns [7]. However, the introduction of mobile cameras generates a new level of variability in the workload that is not easily predictable. First, mobile cameras constantly vary their point of observation and might capture different scenes at different instances in time. Second, the inherent moving nature of mobile cameras causes them to appear or disappear from the deployment, generating sudden changes in the workload that the video analytics pipeline needs to process. Overall, relying on long term prediction models to infer incoming load is not sufficient, or even potentially counter-productive, to correctly balance the load in the presence of mobile cameras.

Challenge #3: Varying configurations. Early work in video analytics focused on the problem of optimizing the placement in the infrastructure of the functions that belong to the processing pipelines. The placement decisions behind this optimization are conventionally driven by the

available resources in the compute infrastructure, *i.e.*, the number of available servers or GPUs, and the workload generated by available cameras, *i.e.*, the number of camera flows to process [11–13]. Due to the overhead incurred, changes in the deployment configuration, such as the addition or removal of processing components, occur a longer time scales due long term pattern shifts (*e.g.*, day vs night scenes) [7, 8]. However, the presence of mobile cameras introduces a new level of dynamism in the deployment that has yet to be accounted for. Mobile cameras can appear or disappear from the deployment, and the processing pipeline needs to be able to adapt to these changes without requiring a complete reboot of the processing pipelines. Reboots occur when new containers are instantiated or, depending on the load balancing solution, to change load balancing policies. The first point is particularly problematic, as starting up or transferring containers can lead to long delays, causing the loss of frames that could otherwise have been processed. This introduces the need for an online approach to load balancing that can adapt to changes in the deployment configuration, such as the addition or removal of processing components, without requiring any hard reboots and quickly adapting, in less than a few seconds, to the incurred changes.

To address the challenges identified, we present in the following section VideoJam, a live video analytics solution aimed at (i) supporting the coexistence of fixed and moving cameras within the same processing pipeline; (ii) combining horizontal distribution and function-based load balancing, powered by short-term predictions of incoming loads.

3.3 Related work

In recent years, several techniques have been developed to improve the performance of video analytics applications [3, 4, 57]. Such a topic has been tackled from different perspectives, including the design of different data processing architectures [8, 12, 25], the improvement of pipelines' processing [13, 14, 58, 59] and the privacy of the extracted data [60–62].

Architecture scaling. Different approaches have been proposed to efficiently manage the computational resources for video analytics [8, 10, 12, 25]. Chameleon, presented in [8], frequently reconfigures the placement of video analytics pipelines to reduce resource consumption with small loss in accuracy. Another example is Spatula [25], which exploits the spatial and temporal correlations among different camera flows to reduce the network and computation costs. However, such solutions only consider video flows coming from fixed cameras.

Deployment strategies. Other solutions mainly focused on the deployment strategies of video analytics applications [7, 10, 27]. Distream [7] is a distributed framework based capable of adapting to workload dynamics to achieve low-latency, high-throughput and scalable live video analytics. Pipelines are deployed on both the smart cameras and the edge, and are jointly partitioned so that part is computed on the smart cameras, while the rest is sent towards the edge, which has more computing power at its disposal. The deployment of application pipelines is adapted to the

varying processing load, however there is a lack of adaptability required by the rate of mobile cameras. The work in [27] presents experimental results showing that smartly distributing and processing vision modules in parallel across available edge compute nodes, can ultimately lead to better resource utilization and improved performance. The same approach is also used by VideoStorm [10] which places different video functions across multiple available workers to satisfy users' requests. We assume a deployment of pipelines in line with this latter work given the higher flexibility, higher scalability and the better use of resources of this approach.

Load balancing strategies. Once video analytics applications have been deployed on a distributed edge infrastructure, load balancing strategies play a fundamental role in guaranteeing requirements of accuracy and efficiency. As a result, The concept of implementing load balancing for video analytics applications has gained popularity, particularly with the emergence of edge video analytics and its associated limitations. Historically, load balancing was performed between edge nodes and central clouds (*e.g.*, VideoStorm [10]), but this method became impractical due to increased network traffic and potential network bottlenecks. Load balancing between edge locations has been the subject of several works, including Spatula [25], Hetero-edge [15], and VideoEdge [9] (albeit VideoEdge still relies on offloading to remote clouds). However, these works focused on the production of static configurations, where each processed video is directed to a predetermined path through the deployed processing functions. As any change of configuration impacts the deployed functions, configuration updates occur over longer timescales, making these approaches unsuitable for highly variable loads. More recently, Distream [7] recognized the need for rapidly adapting to varying loads within a video, proposing an adaptable load balancing solution that splits the load balancing decisions at the pipeline level. However, it assumes that workflows present predictable longer-term patterns, allowing reconfiguration decisions only to be taken at longer timescales, for example when traffic conditions change during the day due to commute patterns.

Workload prediction. Workload predictions, based on ML models, have been proved to be effective in the design of load balancing policies [7, 29, 63, 64]. In [28], authors used reinforcement learning for performing real-time estimation for dynamic assigning task to the optimal server. While the work in [29], focused on load forecasting by using linear regression model. However, reinforcement learning solution, even though effective, require a significant amount of resources and continuous online training to avoid concept-drift problems [65]. Such a solution method is not suitable for the computational-constrained devices at the edge. In addition, the rapid changes in scenes captured by mobile cameras are more difficult to predict. Therefore, there is a need of lightweight forecasting models that can predict short-term trends, suitable for edge devices and fast enough for real-time prediction.

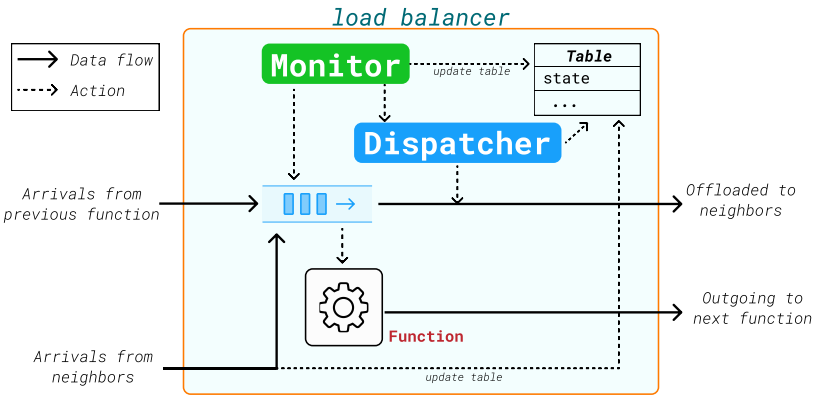


Figure 3.5: VideoJam's load balancer.

3.4 VideoJam Architecture

This section presents the general design of VideoJam, a self-balancing architecture for live video analytics. VideoJam is designed around three design guidelines:

1. **Per-function type-based load balancing.** To cope with the heterogeneous performance profiles and highly variable workloads, we design VideoJam to implement a decentralized set of load balancers on a per-function type basis.
2. **Short-term load forecasting.** To enhance frames offloading across functions but avoid errors from unreliable long-term trends, we design VideoJam to solely rely on short-term forecasting of incoming workload trends.
3. **Robustness to configuration changes.** Finally, to work independently from variations in configurations changes and camera arrivals and departures, we design VideoJam to solely rely on observed performance patterns (*e.g.*, incoming rate) rather than any pre-compute knowledge of deployment configuration.

In the rest of the section, we first present the global overview of the VideoJam architecture, then describe in details the components of the system.

3.4.1 System Overview

The core VideoJam's architecture leans on a set of load balancers, one associated with each and every replica the processing functions deployed among the set of available edge servers, as shown in Figure 3.4. In VideoJam, each load balancer (shown in Figure 3.5) communicates with the rest of load balancers installed on the functions of the same type (*e.g.*, background subtractor, object recognition, *etc.*), also called *neighbors*. The load balancer wraps a function replica and mainly consists of two components: the *Monitor* and the *Dispatcher*. Through these two modules the load balancer takes decisions about incoming frames from the previous functions in the pipeline and from the neighbors to compute the (i) the output to the next function in the pipeline, and (ii) a load balancing policy, *i.e.*, the amount of frames to be offloaded towards its neighbor. Such policy is calculated on the basis of a state table containing the information on the states of each neighbor, presented in rows. This

information includes the estimates of load for the neighbors derived from predictions generated by a forecasting model.

3.4.2 Message Exchange

In VideoJam's system of load balancers, each component computes its state and load balancing policy based on local information, *i.e.*, the incoming load from the previous function in the pipeline, and information collected from the neighbors acquired through a set of message interactions listed below. These mechanisms are designed to ensure that the load balancers converge to a common policy and compensate for potential errors in load forecasting.

Information Request. Load balancers submit a request for neighbor state information when their local data is missing or obsolete. In this case, a load balancer sends, together with its table, an information request to its neighbor, asking for information about its status. Based on the response (described below), the receiver then updates its table according to the table received, substituting the rows containing obsolete information.

State Update. A state update is always sent by a load balancer in one of two scenarios: (i) at bootstrap to announce its presence to neighbors; (ii) after receiving an information request from one of its neighbors. State responses contain a summarization of the local system state, including current and forecasted loads, as well as expected incoming loads from neighbors. State updates are also sent when a prediction error is detected after receiving an offloaded workload from one of the neighbors. Such an error quantifies the difference between the expected load (according to the predictions) and the actual load received from the neighbor.

Congestion Risk Signal. A congestion risk signal is sent by a load balancer to one of its neighbors only when it detects, after updating its table and according to the last computed policy, that a given neighbor is transmitting offloaded traffic to itself, while this was not expected given previous exchanges, *i.e.*, when the neighbor is not present in the list of neighbors from which it should receive offloaded video data.

Data Offloading. Beyond the actual offloaded workload, the data transferred from a load balancer to one of its neighbors may also contain the sender's table. It is worth noting that a load balancer cannot receive workloads from the neighborhood while it is offloading to another neighbor. Hence, the receiver first checks whether it is offloading to one or more neighbors according to its computed policy. In this case, it recomputes the load balancing policy and checks whether, according to the new policy, it is supposed to receive some data. If not, it sends a congestion risk signal back to the sender. Otherwise, it receives the workload, computes the prediction error and sends a status update if the error is above a certain threshold.

3.4.3 System state computation

In VideoJam, the system state is computed periodically and represents the current state of the load balancer based on the incoming load from the previous function in the pipeline, the processing rate, the queue size, and the offload rate to its neighbors. Here we formally describe the computation of the system state.

Operational Time Windows. Each load balancer associated to function c, operates over a time windows W_j^k, which consists of $k \in \mathbb{N}^+$ consecutive time-slots and is defined as

$$W_j = W_j^k = \{w_i\}_{i=1}^k,$$

where w_i is the i^{th} time-slot of the time window. All the time-slots within the time window have the same duration of Δ seconds. The next time window is denoted as W_{j+1}.

Each load balancer $S_{c,n}$ is characterized by its state, and keeps track of the states of all its neighbors (identified by $\mathcal{S}_c^*$) in a table. We further indicate with S_c the set of all load balancers associated to function c deployed among the edge nodes.

State. At each time window $W_j^k = \{w_i\}_{i=1}^k$, the state of load balancer $S_{c,n}$ is defined by (i) its processing rate $\mu_{c,n}$, (ii) the historical incoming load $\{\lambda_{w_i}^{c,n}\}_{w_i \in W_j^h}$ of $h \geq k$ previous time-slots, (iii) its current queue size $\varphi_{W_j}^{c,n}$, and (iv) the offload δ_n^c, *i.e.*, the total workload offloaded towards its neighbors. The table of each load balancer contains an estimation of all its neighbors' states. Such an estimation is based on the last information the load balancer has received from the neighborhood and it is updated every r time windows. When this time expired, *i.e.*, $r = 0$, the load balancer sends an information request to the neighborhood. As shown in Figure 3.5, the interaction of the two main components, *i.e.*, the *Monitor* and the *Dispatcher*, of each load balancer, determines the final offloading policy. We describe in detail how the di*Dispatcher*spatcher calculates the offloading policy in the next section.

The *Monitor* is in charge of monitoring the state and performance of the load balancer over time. More precisely, it supervises the incoming load from the prior function in the pipeline, and the processing rate. At each time-slot w_i of a given time window, the *Monitor* measures the amount of workload $\lambda_{w_i}^{c,n}$ coming from the previous function in the pipeline. At the end of each time window, the *Monitor* updates the historical incoming load of the state of the load balancer by shifting its $h - k$ values to the left and replacing the last k values with the load received in the window W_j. After such an update, the *Monitor* triggers the *Dispatcher* for the computation of the load balancing policy.

3.4.4 Load Balancer Algorithm

Here we describe in details the algorithm executed by each load balancer to compute its local offloading policy. The *Dispatcher* is in charge of computing the load balancing policy of VideoJam through the computation of the queue size and the prediction of future incoming workload for the load balancer. For a given function c, the policy is determined in such a way that the load is fairly distributed among all the load balancers in $\mathcal{S}_c$. In this manner, the load is processed at approximately the same time, as shown in [66]. Regarding the prediction of future incoming workload, given the limited capabilities of computational resources at the edge, state-of-the-art methods, such as Long Short-Term Memory (LSTM) [67], result to be computationally intensive and, hence, prohibitive for the described scenario [68]. Furthermore, when it comes to video analytics, learning the specific distribution of the load results to be a non-trivial task due to concept-drift problems [69], especially when dealing with mobile cameras. For these reasons, the *Dispatcher* relies on lightweight ML models to predict the incoming workload. In particular, we have developed a lightweight convolution-based neural network model based on a few convolution layers for predictions [70]. This model presents fast inference time and high accuracy metrics. The complete pseudocode of the *Dispatcher* is described in Algorithm 1.

Determine the Queue Size. Within a given time window W_j, each load balancer determines the queue size $\varphi^{c,n}_{W_{j+1}}$, which is the load at the beginning of the next window W_{j+1}, by counting the amount of load currently waiting for processing (line 2 of Algorithm 1). The queue size of each neighbor $S_{c,m} \in \mathcal{S}^*_c$ is estimated by adding (i) the difference between the incoming load and the processing rate, and (ii) the total amount of workload exchanged with the neighborhood, to the previous expected load (lines 3–7 of Algorithm 1). More formally,

$$\tilde{\varphi}^{c,m}_{W_{j+1}} = \tilde{\varphi}^{c,m}_{W_j} + \sum_{w_i \in W_j} (\lambda^{c,m}_{w_i} - \mu_{c,m}) + \delta^c_m, \tag{3.1}$$

where, $\delta^c_m = \sum_{p \in \mathcal{N}} \delta^c_{m|p}$, $\delta^c_{m|p}$ is the offload between $S_{c,m}$ and $S_{c,p}$, where $\delta^c_{m|p} < 0$ if data is offloaded from $S_{c,m}$ to $S_{c,p}$, and $\delta^c_{m|p} > 0$ if $S_{c,p}$ is sending data to $S_{c,m}$.

Predict Future Incoming Workload. The load balancer forecasts its load and the incoming load of its neighbors for the next time window. For such predictions we rely on a convolution-based neural network model used to predict short-term workload [29] (more details on the model are available in Section 3.5). The input of the predictive model is the historical incoming load W_h (*i.e.*, $\{\lambda^{c,n}_{w_i}\}$ consisting of $h \geq k$ previous time-slots) for all load balancers. While the output is the incoming load for the next window W_{j+1}. So when the predicted workload deviates from the actual workload during the monitoring phase, the *Monitor* can detect it and react to make adjustments at any time.

Offloading Policy Computation. Within a given time window W_j, each load balancer $S_{c,n}$ determines the offloading policy, *i.e.*, $\delta^c_{n|m}$ for each

$S_{c,m} \in \mathcal{S}_c$ for the next time-window W_{j+1}. Such a policy is computed through the following steps:

(i) First, the estimation of the global load, $\tilde{\phi}^{c,n}$, is computed according to

$$\tilde{\phi}_{c,n} = \tilde{\varphi}^{c,n}_{W_{j+1}} + \sum_{w_i \in W_{j+1}} \lambda^{c,n}_{w_i}, \tag{3.2}$$

i.e., the estimated queue size at the beginning of the next time window plus the expected incoming load. The queue size estimate is performed for all its neighbors in $\mathcal{S}^*_c$, whereas it can simply be obtained from the load balancer queue.

(ii) Then, based on the estimation of the global load, each load balancer $S_{c,n}$ estimates the actual load that every load balancer in $\mathcal{S}_c$ should handle:

$$\bar{\phi}_{c,n} = \frac{\sum_{n \in \mathcal{N}} \tilde{\phi}_{c,n} \times \mu_{c,n}}{\sum_{n \in \mathcal{N}} \mu_{c,n}}, \tag{3.3}$$

where $\tilde{\phi}_{c,n}$ is the global load that should be processed by $S_{c,n}$ with a process rate of $\mu_{c,n}$, and $\sum_{n \in \mathcal{N}} \mu_{c,n}$ is the total processing capacity of all the load balancers associated to function c.

(iii) The load balancer computes the *unbalanced* load for all the load balancers associated to function c (line 12 in Algorithm 1) as

$$\theta_{c,n} = \bar{\phi}_c - \tilde{\phi}_{c,n}. \tag{3.4}$$

Positive values for $\theta_{c,n}$ indicate that the function associated to the load balancer will be underutilized. In this case, it should receive workload from neighbors $S_{c,m}$ with negative values of $\theta_{c,m}$. Negative values for $\theta_{c,n}$, on the other hand, point out that the function will be overloaded requiring to offload some workload to the neighbors.

(iv) Finally, the amount of workload to be offloaded towards each neighbor $S_{c,m}$, *i.e.*, the offloading policy $\delta^c_{n|m}$, is finally computed based on $\theta_{c,n}$ as described from line 14 to 27 of Algorithm 1. Until all the θ's are set to 0, the *Dispatcher* takes the most loaded $S_{c,n}$ to balance to the least leaded $S_{c,m}$ in S_c. The load is transferred from $S_{c,n}$ to $S_{c,m}$ if there is enough room for the load, and the unbalanced load of $S_{c,n}$ is set to 0. Otherwise, $S_{c,n}$ transfers to $S_{c,m}$ the maximum load that can be received $\theta_{c,m}$ and the unbalanced load of the receiver is set to 0.

Algorithm 1: *Dispatcher* algorithm procedure

Function *dispatcher*(φ, λ, μ, δ, $\mathcal{S}_c$)
  /* 1. Determine the queue size */
  $\varphi^{c,n}_{W_{j+1}} = getQsize()$
  /* 1. Estimation load for neighbors, *i.e.*, $\mathcal{S}^*_c$ */
  for $S_{c,m} \in \mathcal{S}^*_c$ **do**
    $\tilde{\varphi}^{c,m}_{W_{j+1}} = \tilde{\varphi}^{c,m}_{W_j} + \sum_{w_i \in W_j}(\lambda^{c,m}_{w_i} - \mu_{c,m}) + \delta^c_m$;
  end
  for $S_{c,n} \in \mathcal{S}_c$ **do**
    /* 2. Forecast the future incoming load, *i.e.*, $\lambda^{c,n}_{w_i}, w_i \in W_{j+1}$ */
    $\lambda^{c,n}_{w_i} = model(\{\lambda^{c,n}_{w_i}\}_{w_i \in W^h})$;
    /* Estimation of global load */
    $\tilde{\phi}_{c,n} = \tilde{\varphi}^{c,n}_{W_{j+1}} + \sum_{w_i \in W_{j+1}} \lambda^{c,n}_{w_i}$;
  end
  /* Compute the unbalanced load */
  for $S_{c,n} \in \mathcal{S}_c$ **do**
    /* balanced load $\bar{\phi}_{c,n}$ */
    $\bar{\phi}_{c,n} = \frac{\sum_{n \in \mathcal{N}} \tilde{\phi}_{c,n} \times \mu_{c,n}}{\sum_{n \in \mathcal{N}} \mu_{c,n}}$;
    /* unbalanced load */
    $\theta_{c,n} = \bar{\phi}_c - \tilde{\phi}_{c,n}$
  end
  /* 3. Compute the offloading policy */
  while *any*($\theta_{c,n} < 0$) *and any*($\theta_{c,n} > 0$) **do**
    /* the most overloaded */
    $n = argmin(\theta_c)$;
    /* the less overloaded */
    $m = argmax(\theta_c)$;
    $q = |\theta_{c,n}|$;
    if $q < \theta_{c,m}$ **then**
      /* load from $S_{c,n}$ to $S_{c,m}$ */
      $\delta^c_{n|m} = q$;
      $\theta_{c,n} = 0$;
      $\theta_{c,m} = \theta_{c,m} - q$;
    else
      $\delta^c_{n|m} = \theta_{c,m}$;
      $\theta_{c,n} = \theta_{c,n} + \theta_{c,m}$;
      $\theta_{c,m} = 0$;
    end
  end
end

3.5 Implementation and Deployment Configuration

In this section we present the details on how we implement VideoJam, *i.e.*, the set of parameters of the architecture and the video analytics components. Additionally, we outline our evaluation framework, which encompasses the evaluation metrics, the datasets selected for system assessment, and the baselines used for comparison against VideoJam.

3.5.1 Prototype Implementation

We implement VideoJam with about 400 lines of Python 3 code, using the asyncio [71] library to handle I/O operations of incoming frames and objects to process, and OpenCV v4.5.3 with CUDA v11.2.2 support for various vision models. With its simple design, VideoJam offers flexibility in incorporating any existing project, since only the communication component and the actual function (*e.g.*, a DNN model), are needed to be adapted to the existing one. The library is designed to easily support a variety of existing video analytics applications. We integrate the system in a docker image can be pulled from a public docker hub or built from the Dockerfile available in our public GitHub repository [2]. To evaluate the design effectiveness, we implement the functions of a typical traffic control application: vehicles' number plate detection.

2: The VideoJam implementation code is freely available at `https://github.com/ENSL-NS/VideoJam.git`

Video Analytics Components. We implement the vehicles' number plate detection pipeline by integrating the following video analytics functions: video source and decoder, background subtractor and vehicle detection (for fixed camera sources), You Only Live Once (YOLO) object detection (for mobile sources), and number/license plate recognition.

The decoder represents the entry point of the pipeline and takes as input an encoded stream (for the evaluation in this chapter we use pre-recorded videos, yet the system supports live streams as well). The decoded video frames are then passed on to the next function for further processing.

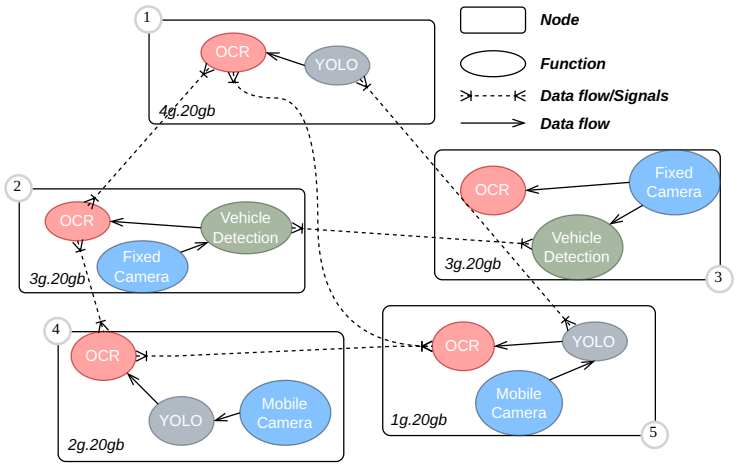


Figure 3.6: An example of heterogeneous deployment of VideoJam. Note that not all links between functions are represented to reduce image complexity.

For fixed cameras, frames are transmitted to the background subtractor. The background subtractor is a function that generates a foreground mask using a static camera. This mask is then used on the current image to subtract the static scene, *i.e.*, the background, while each moving scene is detected and classified as an object of interest. Extracted objects are then passed to the vehicle detection function, which embeds a ML model trained to detect vehicles within an image [72]. Detected vehicles are passed along to the next function, while other objects are discarded.

For detecting vehicles in mobile sources, we integrate a YOLOv5 model [54]. YOLOv5 is the fifth version of the YOLO object detection model. It performs detection and classification, and returns a box for each object in the image taken in input, along with their classes with a high degree of accuracy. This is computationally intensive and is generally used on GPUs to achieve fast and accurate results in real-time. There are many pre-trained check-points available, as well as input size pixel images. For our purposes, we use YOLOv5 with a high (640) and low (416) input pixels size.

Finally, detected vehicles are passed to an OCR function that used to detect license plates on cars. Numerous frameworks have been developed for this task. In our implementation we integrate Tesseract [73], an open source OCR engine that combines traditional image processing techniques with modern ML methods to accurately recognize and convert text from images into a digital format.

We use these functions to create a heterogeneous application with two different pipelines. The first takes as input fixed video camera feeds passed along to the background subtractor, which forwards the data to the vehicle detector for classification before the final function, the license plate detection. The second pipeline takes data from a moving camera and processes them using YOLO, then passes the result to the license plate detection. This last function is shared by both pipelines for reuse and optimization. With such a deployment, a situation of workload imbalance can arise at any time, and forecasting becomes more challenging. The complete application used for deployment is presented in Figure 3.6. All functions (except sources and background subtractor) of the same kind run load balancing and implement the VideoJam *Monitor* and *Dispatcher*. We have avoided drawing all the lines between the pairs (as specified in the caption) to avoid increasing the complexity of the figure.

System configuration and model tuning. We configure the VideoJam load-balancing system using the parameters summarized in Table 3.1. Regarding the predictive model, we opt for an architecture that minimizes inference time while guaranteeing acceptable performance. We choose a neural network with a single dense layer of 512 units trained over 100 epochs. The model takes in input a window of size h and can predict a window of size k (see Table 3.1 for the values of these parameters). We also tried out different architectures, such as a Convolutional Neural Network (CNN), and LSTM. However, such models result to be difficult to use since (i) CNN requires a significant amount of time for the inference, although it has high levels of accuracy; (ii) in our case, LSTM presents poor performance in terms of both accuracy and inference time given its specific use on time series (different from our case).

Table 3.1: VideoJam configuration

Parameter	Value and description
Δ	1 second (the monitoring duration)
k	10, represents a monitoring window of $10 \times \Delta = 10$ seconds
h	50, the history for short-term forecasting
model	short-term forecasting: DNN for Vehicle detection and Number plate OCR, none for YOLOv5

3.5.2 Evaluation Setup

Baselines. We evaluate VideoJam compared against three different baselines: (i) a video analytics pipeline that does not implement any load balancing, (ii) one that implements Weighted Round Robin (WRR), and (iii) Distream [7], a state-of-the-art solution. In WRR, neighbors determine their processing rates based on an initial estimation, share this information with their neighbors, and collectively assign weights based on their capacity to create a load-balancing policy. They then apply this policy to distribute incoming workload to their local queues or to neighbors, with offloaded work being placed directly in a neighbor's local queue. For Distream, we start with the version available at the project repository [3]. We then transform the Golang code into Python for integration into our deployment framework. Finally, we also adapt the code to support batch processing, mentioned in the article but not implemented in the open source version. The final code is also available on our public GitHub. Note that Distream does not support heterogeneous pipelines, thus we solely integrate the pipeline with YOLO into its architecture.

3: https://github.com/AIoT-MLSys-Lab/Distream/tree/main

Deployment Infrastructure. We conduct experiments deploying multiple docker containers on a server equipped with Nvidia A100 GPUs (full specifications are shown in Table 3.2). For functions requiring access to GPU resources, we leverage the Multi-Instance GPU (MIG) that Nvidia GPU offers to split the available GPUs into multiple instances. We emulate a heterogeneous edge infrastructure by splitting the GPU into six nodes with heterogeneous compute cores (*i.e.*, *4g.20gb*, 2× *3g.20gb*, *2g.10gb* and 2× *1g.5gb*) [74]. Given the lack of enough Central Processing Unit (CPU) cores we leave all containers to concurrently use all available CPUs. While this reduces the realism in terms of CPU isolation, the implemented functions mostly rely on the GPU for heavier computations, thus not introducing unwanted bottlenecks to the setup. Finally, regarding network connectivity between containers, we emulate a realistic network configuration to the extent possible. For this reason, we not only limited the links between nodes to 1gbps, but also configured these links to have realistic latencies and bursts. For information, we used the following *tc* command: *rate 1gbit burst 16kbit latency 10ms*.

Evaluation metrics. We evaluate the performance of VideoJam and the other baselines using three metrics: (i) the response time, (ii) the loss rate, and (iii) the total bandwidth utilization. The response time for a frame refers to the time elapsed from its introduction into the system

Table 3.2: Server configuration used for experiments.

Model	Dell PowerEdge R7525
CPU	AMD EPYC 7452 (Zen 2), 2 CPUs/node, 32 cores/CPU
Memory	128 GB
GPU	2 x Nvidia A100-PCIE-40GB, Compute capability: 8.0

Table 3.3: Video cameras used for experiments.

Type of camera	Duration (min)	Resolution	Total videos
Mobile	11-80	720p	9
Fixed	5-60	720p	11

(*i.e.*, from the source) to its complete processing by the last function in the pipeline. It can also be measured at the level of a specific pipeline function. For example, the response time for a frame measured at the vehicle detection level corresponds to the time elapsed between its entry into the pipeline and its exit from this function. A low response time is an indicator of the system's ability to quickly extract the information generated by the application. The percentage of losses corresponds to the number of objects lost over the total number of objects to be processed. In general, each function has a queue with a maximum number of items that can be held in it. When the incoming load exceeds a function's capacity, it begins to accumulate load in its queue. When the queue is full, any new incoming objects arrive they are dropped. Since we do not focus on function selection, losses become the main indicator of accuracy reduction. Finally, total bandwidth utilization corresponds to the total amount of data transmitted during load balancing between functions. This is an important measure, as it enables us to measure the impact of the different approaches used on network resources.

Datasets. We use public real-world videos from YouTube for both fixed and mobile cameras. We selected a custom dataset of videos from YouTube for three reasons: (i) Existing mobile video datasets did not meet our requirements. For example, the BDD100k dataset [4], widely used for analytics tasks, contains a good collection of traffic datasets from an on-board camera on cars in different weather conditions. However, the dataset contains only short sequences of videos lasting less than a minute or using under-sampled images. (ii) Using YouTube videos is a common standard in the state-of-the-art [7, 75–78]. (iii) The selected dataset meets the diversity of requirements for the evaluation in this chapter. The videos we selected include 11 videos from dashboard cameras mounted on cars traveling through various cities (*e.g.*, Los Angeles [5]), and nine fixed cameras mounted on street corners in the same cities [6]. These recordings come in a variety of conditions, including normal to heavy traffic, as well as empty and crowded locations. As a result, the number of vehicles (*e.g.*, cars, trucks) in the frames of these videos are variables, with minimum, maximum, mean, and median respectively of 0, 15, 3, and 3 per frame. Further, these videos are longer than the ones described in the previous dataset, and they span between 5 and 80 minutes (see Table 3.3).

4: http://bdd-data.berkeley.edu/

5: https://www.youtube.com/watch?v=CwOlb-nqSNE8

6: https://www.youtube.com/watch?v=wqctLW0Hb-0

3.6 Evaluation

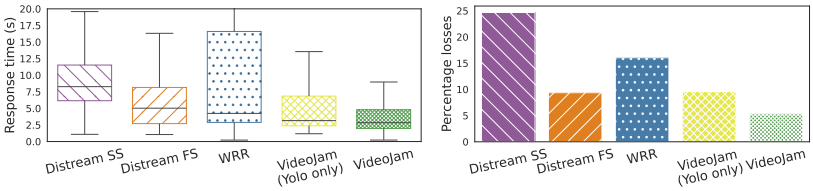
We evaluate VideoJam in different scenarios and against the three baselines previously described. First, we compare it against Distream [7] to demonstrate the benefits of localized load balancing at function level, rather than a centralized approach for global load balancing. Next, we evaluate the system under different levels of loads to measure the performance of VideoJam as the workload increases. In addition, we test VideoJam's ability to adapt to system configuration changes or failures, by subjecting it to critical situations such as the failure of a function or the failure of a node. Finally, we test the system's ability to deal with mobile cameras leaving and joining the architecture, demonstrating VideoJam's ability to adapt to forecasting errors caused by sudden changes of video content.

3.6.1 Comparison with Distream

Distream's load balancing architecture is based on two main key concepts: the cross-camera workload and the partition point. The cross-camera workload determines the workload balance among cameras (also called *Ends*) only. The partition point defines which functions of each pipeline are processed by Ends, while other functions are then executed on the server (called *Edge*). The choice of offload proportion is handled in one of two different ways by the Ends: either a *Full-Stochastic* (FS) or *Semi-Stochastic* (SS) partitioning. In full-stochastic partitioning, the Ends generate a random partitioning proportion based on a Bernoulli random variable with probability set proportionally to the compute power of the Edge and End nodes. At every step of the pipeline, Ends draw a value from this variable and determine whether to process the function locally or offload it to the Edge. In semi-stochastic, the random value is drawn only at the partition point. In this case, the partition point is calculated as the point in the pipeline that evenly splits it proportionally to the compute power of the two components.

VideoJam outperforms baselines in heterogeneous deployments. We compare VideoJam and other baselines in dealing with mixed traffic of mobile and fixed video cameras. To carry out this experiment, we deploy Distream using five GPU-equipped nodes (as described in Section 3.5). We deploy one instance of YOLO and one instance of OCR on each node, as required by Distream. We use the same five nodes for VideoJam, but we partition nodes hosting the pipeline for fixed and mobile cameras. In particular, we deploy two vehicle detection instances, three YOLOs, and five OCRs. We also deploy two other baselines, WRR and a VideoJam version that solely employs YOLOs for detection, and using the same configuration as VideoJam. Finally, we use four video sources, two mobile cameras and two fixed ones, draw randomly from the dataset presented in Section 3.5 and set to 20 frames per second.

Figure 3.7 shows that VideoJam outperforms all other solutions, both in terms of response time, with 2.91× and 1.77× less response time compared to Distream's solutions, and reduces objects loss by 19% and 4%, respectively. This highlights the advantages of using a localized load



(a) Response time.

(b) Percentage losses.

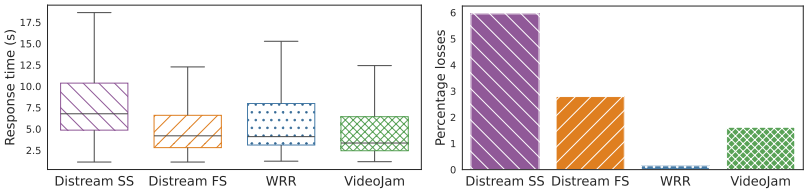
Figure 3.7: Evaluation on a heterogeneous architecture shows a response time up to 2.91× lower than comparative approaches, and with fewer losses.

balancing technique like VideoJam, and the limitation of approaches like Distream or a simplistic WRR. Further, the figure highlights that the use of mixed pipelines for different sources of traffic improves response time. Indeed, the use of dedicated pipelines for fixed and mobile cameras, *i.e.*, background substractor vs YOLO as explained Section 3.2, improves classification performance incurring less load depending on the number of objects in each frame. Consequently, if the amount of objects in the video scene is low, no action, *i.e.*, inference, will be taken, whereas detection techniques such as YOLO will take action even if no objects are present in the frames. This is supported by the performance improvement observed when comparing the two approaches on VideoJam (*i.e.*, the one using vehicle detection over the one using only YOLOs). Furthermore, the results show that VideoJam has a response time 1.11× lower than VideoJam with YOLO only.

VideoJam outperforms Distream in mobile only scenarios. In the previous experiment we have shown that VideoJam outperforms baselines when processing mixed types of video traffic. We now explore whether our solution can still outperform Distream when solely processing video traffic generated by mobile cameras. We do so to evaluate VideoJam's load balancing technique, understanding whether the advantages presented previously are to be solely attributed to the use of different pipelines for different types of traffic or to the load balancing as well. In this experiment, we use the same experimental setup as in the previous experiment, the only difference being that we use YOLOs throughout the deployment for all baselines, and we use four mobile cameras.

Figure 3.8 shows the performance of WRR, Distream, and VideoJam. We can observe that the semi-stochastic version of Distream incurs the worst performance in terms of both response time and loss: about 2.02× and 3.68×, respectively, compared to VideoJam. Indeed, given the design of Distream, it is difficult to define the load balancing policy when considering the pipeline as a whole. In fact, two different functions (*e.g.*, YOLO and OCR) on different nodes may become overloaded as traffic loads vary in time. This makes it very difficult to define an optimal load balancing policy for load distribution. Furthermore, in our observation, the considerable loss observed is due to the fact that a large portion of the video traffic remains continuously blocked in the Edge, which is unable to empty it quickly enough.

Nevertheless, we observe similar response time results between the full-stochastic approach and VideoJam. The reason for this lies behind the simple offload policy implemented in this approach: as the pipeline consists of only two functions, the partition is only necessary at either YOLO or OCR, often leaving the Edge in charge of the full processing. Yet, this simplicity can incur increased loss, when these nodes become



(a) Response time.

(b) Percentage losses.

Figure 3.8: Evaluation on mobile cameras shows that VideoJam's response time is 1.25× shorter than Distream's.

overloaded (about 3% loss). We also observe that WRR experiences a lower percentage of loss compared to VideoJam. This is due to WRR's simpler load-balancing policy, which in some cases can be beneficial to loss, when frames spend more time being transmitted between nodes, slowing down the volume of traffic reaching the OCRs. In fact, we observe that WRR tends to balance load more aggressively across all available instances, thus increasing network usage (more details on this later in this section). Consequently, these frames are not queued fast enough to fill the OCR queues, explaining the small loss for WRR. This also explains the WRR's lower performance in terms of response time (1.22× lower response time than WRR). Ultimately, this shows that in certain instances, there might be a tradeoff to explore between response time and information loss. We leave this exploration for future work.

VideoJam better handles node failures. The aim of this experiment is to determine the impact of node failure on the performance of Distream, WRR, and VideoJam. To do this, we simulate two types of failure: the first is an End failure, the second an Edge failure.

Figure 3.9 summarizes the performance of each solution. We can observe a significant loss for Distream (about 15% of the total traffic). The reason for this large loss is that after the Edge has failed, the last policy calculated by the Edge, *i.e.*, the partitioning point, is still executed by the Ends and is not updated during the Edge's absence. So when the Edge comes back, the computation it is supposed to be dealing with since the last partition point update suddenly arrives, saturating the Edge in the process and causing several losses. Also, Distream's low response time is due to the fact that fewer frames are waiting in the queue to be processed after a large proportion of them have been lost.

Furthermore, we see the ability of VideoJam to be robust to failures and to react a recovery. Indeed, when a node failure occurs, VideoJam adapts the load balancing policy calculation to the available resources. When the failed node returns, the policy is recomputed and all the workload already accumulated is redistributed. This explains the low losses of around 2% which are 6× lower than Distream's ones. WRR, on the other hand, does not present the same ability of robustness to failures, even though its policy is updated every time the system is stressed. And since its policy does not take into account the workload of the instances, the load previously accumulated during the downtime is not redistributed.

3.6.2 Load Balancing Ablation Study

In this section, we evaluate the effectiveness of VideoJam's load balancing algorithm with respect to two alternative baselines, no load balancing

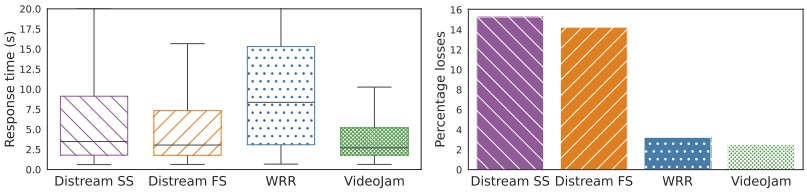
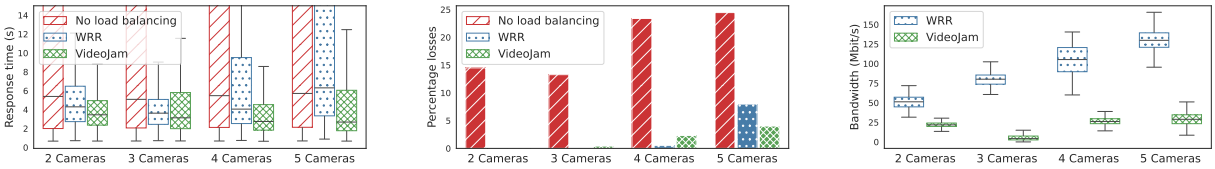


Figure 3.9: Evaluation of Distream and VideoJam in the event of node failure, with the latter recording fewer losses while maintaining better response time.

(a) Response time.

(b) Percentage losses.

and WRR.

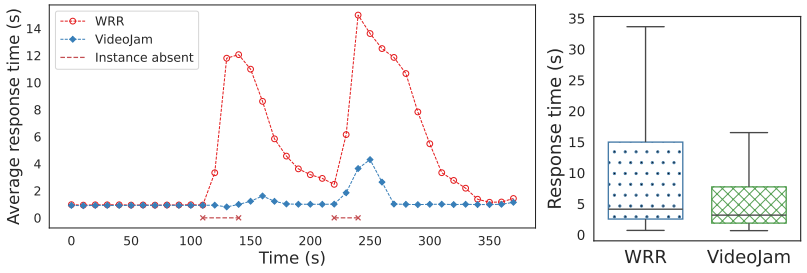


(a) Response time of the system.

(b) Percentage of frame lost.

(c) Bandwidth used for offloading to neighbors.

Figure 3.10: Evaluation on several configurations shows the need for a load-balancing technique and the effectiveness of VideoJam in achieving lower response times with less bandwidth usage.



(a) Averages for frames exiting YOLO.

(b) Response time.

Figure 3.11: Evaluation under node failure conditions: WRR's poor performance and VideoJam's adaptability under failures.

VideoJam Under Different Workloads. To evaluate the sensitivity of VideoJam to the level of workload, we subject it to different workloads, increasing the number of source cameras given a fixed function placement (deployment). In this scenario, we use six GPU-equipped nodes, and we set the number of YOLO and OCR instances to three and five respectively, while we increase the number of mobile cameras from two to five with a rate of 15 frames per second.

Figure 3.10 shows the system response time, the percentage of lost frames, and the bandwidth used during the offloading. VideoJam generally shows better performance with respect to the other compared solutions. The poor performance of no-balancing strategy highlights the need of load balancing solutions in this context. WRR reasonably shows longer response times as the load increases since such solution struggles in presence of network congestion. This is also highlighted by the increase in the bandwidth utilization showed in Figure 3.10c. VideoJam, on the other hand, even in the event of congestion, only offloads the difference between instances to prevent two instances from sending load to each other, which explains the low bandwidth used. Furthermore, given the efficient collaboration among neighbors, VideoJam maintains stable performance as the workload increases.

In summary, this experiment has shown that VideoJam balances the workload without compromising performance, while minimizing system response time. In addition, while WRR statically balances incoming

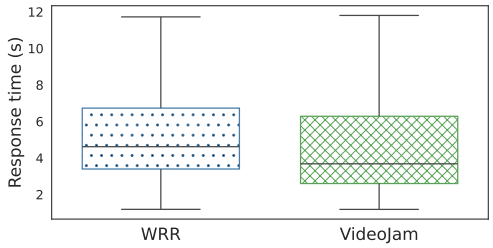


Figure 3.12: Little effect of source mobility and abrupt flow changes on VideoJam performance.

workload to neighbors, VideoJam adapts to the current situation by computing the most appropriate policy, and prevents bandwidth wastage due to bidirectional load migration.

VideoJam's Adaptation to Functions Placement. We evaluate the impact of different placement strategies (configurations) on VideoJam, particularly in the case of real-time variations. Note that there is a fundamental difference between workload variations and configuration changes or function placement strategies. Whereas in the previous experiment, we studied workload variation, which applies to the number of objects to be processed in each video stream, in this experiment we will study the case of configuration changes, which applies to the number of streams and processing modules available in the system.

Many studies have been carried out to find a better solution for placing components or functions to maximize the use of hardware resources while maintaining good precision [9, 10]. As VideoJam is agnostic of the placement, we aim to evaluate if performance remains stable across different configurations. To do this, we start with four sources (15fps), three low-resolution YOLOs and five OCRs. First, we deliberately remove a YOLO function before redeploying it, but this time with a higher resolution, but with a higher inference time (lower throughput). This corresponds to a configuration change that can occur when placement techniques are used. In a second step, we kill all components (YOLO and OCR) on another node before restarting them after about 15 seconds. This simulates a node failure that can occur at any time when dealing with edges.

The average response time and the system response time of frames, as they leave the YOLO component, are shown in Figure 3.11a and Figure 3.11b respectively. Initially, YOLOs maintain a consistently low response time, as they efficiently handle the incoming workload that is below their capabilities. At ~100s (Figure 3.11a), a YOLO instance fails, with the source previously connected to it now sending its stream to another instance. This results in a sudden increase in the load that is badly distributed between instances for WRR. In contrast, in with VideoJam, instances remain capable of handling all the load generated by the sources. When the function is reintroduced into the system with a different configuration (higher resolution), WRR shows a gradual and slow decrease in response time, as it is unable to redistribute the loads already allocated to instances.

A similar behavior is observed at ~220s, where a YOLO and an OCR deployed in the same node are killed due to node failure. Here again, the response time increases for WRR and VideoJam. In this particular case, the remaining YOLOs (one high-resolution and one low-resolution)

are unable to handle the incoming workload, resulting in an increase in response time. In contrast, VideoJam excludes the outgoing instance from the load-balancing policy until it returns, or new functions are added, allowing previously accumulated overload to be efficiently redistributed between instances.

Impact of Mobility. One of the key features of mobile cameras is their ability to be in the right place to retrieve useful information. We aim to emulate mobile cameras, such as dash cams mounted on cars, entering and leaving the system. We emulate this scenario in our experiment as follows: at first, we have a camera streaming mobile content (*i.e.*, a dash cam content as described in Section 3.5.2, in the paragraph related to the datasets); after a fixed amount of time, the camera leaves the system, emulating the car leaving the area currently being processed. Finally, a new video starts streaming, emulating a new car entering the system. For the experiment, we used three moving cameras (20fps) which send their streams to four YOLOs, which in turn send them to five OCR instances. At a given time, one camera stops sending its stream to the next function for a short period of ~15 seconds. This represents a camera leaving the system and has a direct impact on the load, as the incoming workload correlation is broken. We observe the system response time in Figure 3.12. When a source stops sending its data stream to the next function, the latter sees its incoming load drop drastically and then fails VideoJam's prediction. Even though WRR is agnostic to these events as it continues to distribute frames independently of sources, VideoJam's load balancing compensates for eventual forecasting errors, maintaining 1.25× better response time.

3.7 Conclusion

We present VideoJam, a new approach designed to meet the challenges of video analytics applications that integrate heterogeneous camera sources, *i.e.*, both fixed and mobile. VideoJam responds to scenarios incurring high load variability (such as mobile cameras) by integrating short term load prediction and performing load balancing at function level. Further, the system adapts to varying deployment configurations, not requiring any hard reboots to compensate for them. Thanks to its design, VideoJam reduces response times by 2.91× lower response time, while reducing video data loss by more than 4.64× and generating lower bandwidth overheads.

In future work, we plan to tackle the need of accounting of additional constraints in the analytics pipeline. Currently, VideoJam does not consider inter-function link bandwidths to determine load balancing policies. In heterogeneous network environments, where link speeds differ, this omission can have an impact on overall system efficiency. In addition, while VideoJam works independently of the existing deployment configuration (*e.g.*, number of replicas for each function), it does not compensate for scenarios where the load exceeds the existing processing capabilities (*e.g.*, too many video sources to process). Furthermore, it could be interesting to treat neighborhood cases with functions that are not necessarily identical, but rather functions with identical or similar objectives but

slightly different implementation (*e.g.*, YOLO and SSD [79] are similar). VideoJam, by default, can handle this heterogeneity, although they are treated as identical functions with different performances. However, these functions have many more differences, for example in terms of accuracy, size, inference time, *etc.*, and therefore raise more challenges. Future work will address these limitations, with the aim of improving resource utilization and exploring more adaptive deployment strategies.

ROOMIE: Efficient Model Cohabitation in Edge Computing Model Serving

4

4.1 Introduction

ML inference serving has become a foundational task for a variety of domains, with organizations increasingly deploying ML models that applications spanning from computer vision to natural language processing [80]. Unfortunately, the growing demand for ML inference requests is now outpacing hardware availability, creating a critical resource gap that forces organizations to maximize utilization of existing computational infrastructure. To process the incoming data streams in real-time while working within these constraints, scalable model serving architectures have been proposed to support a multitude of applications across video analytics, language understanding, recommendation systems, anomaly detection, and more [31, 33, 37, 39, 43]. The increasing data volume demands of state-of-the-art models, coupled with tight Service Level Objectives (SLOs) (*e.g.*, latency), has made efficient resource utilization a core challenge across all deployment scenarios.

To address these resource constraints, recent work has explored deploying ML inference pipelines across diverse computational environments, ranging from cloud-grade server clusters to resource-constrained edge devices [30, 33, 39–41, 43]. For instance, cloud platforms like AWS SageMaker or Google Cloud AI offer scalable compute and storage, enabling high-throughput inference for applications such as real-time fraud detection or large-scale recommendation systems. However, these deployments often suffer from network latency and raise data privacy concerns, especially in domains like healthcare or finance where sensitive data must remain local. Conversely, edge deployments—such as running inference on NVIDIA Jetson modules embedded in traffic cameras or smart manufacturing sensors—can reduce latency and mitigate privacy risks by processing data near its source. Yet, these edge devices typically have limited compute and memory, as they are co-located with existing infrastructure like routers or Internet of Things (IoT) gateways. Regardless of the deployment environment, efficiently managing available resources requires intelligent distribution of inference requests across the infrastructure [31, 37]. Unfortunately, existing ML inference serving frameworks, such as TensorFlow Serving or TorchServe, often assume homogeneous, resource-rich environments and overlook the constraints of edge settings, leading to suboptimal hardware utilization and degraded performance.

While significant research has addressed orchestrating model placement and query distribution [30–32], these approaches yield limited benefits when the performance profiles of deployed models are imprecisely characterized or when models operate concurrently. This is particularly problematic in resource-constrained environments where multiple models must share limited hardware and can significantly interfere with each other's execution. Recent work such as Usher [43] attempted to address this gap by analyzing models at the kernel level to better understand performance under interference conditions. However, their

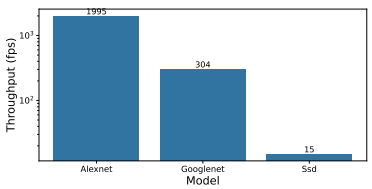
approach fails to account for the complex execution patterns of models on GPU architectures, resulting in inaccurate performance predictions (as discussed in detail in Section 4.2). This limitation becomes especially critical in resource-constrained deployment scenarios, where even minor performance estimation errors can dramatically impact overall system efficiency, potentially rendering carefully orchestrated deployments infeasible. A more nuanced understanding of model execution characteristics is therefore essential to enable truly efficient inference serving across all computational environments.

In this chapter, we present ROOMIE, a model serving orchestration architecture that maximizes system performance in scenarios where colocation is necessary across resource-constrained environments. ROOMIE's key contribution is its kernel-aware interference profiling that captures the sequential nature of GPU kernel execution patterns when multiple models share hardware resources. By understanding how specific kernel sequences from different models interact, ROOMIE builds accurate interference profiles that predict performance degradation under various colocation scenarios. This fine-grained approach enables ROOMIE to make better informed placement decisions, identifying which models can efficiently coexist on the same hardware and which combinations should be avoided to maintain performance across both goodput and latency.

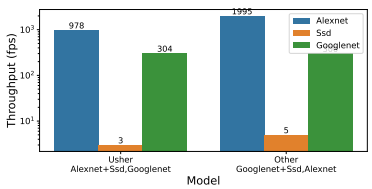
Our experimental evaluation demonstrates ROOMIE's effectiveness across a diverse set of inference models and deployment scenarios. In cloud deployments, ROOMIE achieves up to 17× lower latency and sustains over 97% processing rate, outperforming state-of-the-art solutions [37, 43] through interference-aware colocation and resilience to saturation. On edge devices, it maintains up to 9× lower response times and 1.5× higher throughput, effectively navigating resource constraints. These improvements come from ROOMIE's smart scheduling, which stays close to the best possible setup even under high concurrency. By precisely characterizing interference patterns and adapting to system dynamics, ROOMIE enables scalable, high-efficiency inference serving across modern GPU platforms.

4.2 Motivation and Challenges

In this section, we will discuss the importance of taking model interference into account when inference serving, as demonstrated by the suboptimal performance of existing approaches.



(a) Throughput when the model operates in isolation, without any interference.



(b) An alternative to co-locating three models on two GPU, offering a better compromise than Usher [43].

4.2.1 Limitations of Existing Inference Serving Strategies

The deployment of ML models for inference serving has become increasingly critical across domains such as computer vision, natural language processing, and recommendation systems. As demand for real-time inference grows, organizations are compelled to maximize the utilization of existing computational infrastructure, particularly in resource-constrained environments. While scalable serving architectures have been proposed to meet SLOs, such as latency and throughput, many existing systems rely on simplistic heuristics for model placement and colocation. These approaches typically reference offline profiling data or prioritize devices with the most available memory, overlooking the nuanced performance degradation that arises when multiple models share GPU resources.

The assumption that memory availability is a reliable proxy for inference performance is mistaken. During inference, models consume significantly less memory compared to training, as intermediate states and gradients are not retained. Consequently, memory-centric placement decisions fail to account for the dynamic interference between concurrently executing models. Systems such as Usher attempt to address this by analyzing low-level GPU kernels to estimate resource demands. However, their methodology does not capture the complex interactions between kernels from different models, leading to suboptimal colocation choices.

We conducted an experimental study to evaluate the performance of three inference models: AlexNet, GoogLeNet, and Single Shot Multibox Detector (SSD), on two Jetson Xavier devices. The performance of each DNN when operating in isolation is illustrated in Figure 4.1a. Our primary objective was to identify the optimal co-location configuration among all possible combinations. Our findings revealed that Usher's proposed approach of co-locating AlexNet with SSD is not necessarily the most effective option. In fact, an alternative configuration, GoogLeNet paired with SSD—demonstrates a superior balance in performance (refer to Figure colocation).

We have determined that Usher's misjudgment stems from the method used to assess model compatibility. Merely classifying models based on their computing or memory capacity does not provide a comprehensive understanding of their performance. A more nuanced approach that considers the resource demands of each model over time is essential for accurate evaluation.

4.2.2 The Importance of Kernel-Level Analysis

DNNs perform inference by executing a sequence of GPU kernels, each responsible for specific low-level computations and defined by distinct resource requirements such as shared memory, register usage, and execution time. These kernels, rather than the high-level model architecture, constitute the true computational footprint on the GPU. Profiling tools such as Nsight-Systems [81] and Torch Profiler [82] enable fine-grained observation of kernel behavior, revealing patterns of resource occupancy and execution timing that are often obscured at the model level. Empirical analysis shows that individual kernels frequently leave portions

of the GPU underutilized, suggesting that with careful orchestration, multiple models can be deployed simultaneously to improve overall throughput.

However, when models are executed concurrently on the same GPU, their kernels may overlap in time and compete for limited hardware resources. Although this parallelism can enhance utilization, it also introduces a critical challenge: interference. This occurs when the combined resource demands of overlapping kernels exceed the GPU's capacity, forcing the hardware to serialize execution or delay kernel launches. For instance, if two kernels simultaneously require more shared memory or registers than the GPU can allocate, contention arises, leading to extended execution times and degraded performance. The severity of interference is shaped not only by the type of resources consumed but also by the duration and temporal alignment of kernel execution. Without precise modeling of these interactions, colocation decisions risk undermining system efficiency rather than enhancing it.

Effective colocation requires more than simply identifying underutilized resources; it demands a precise understanding of how kernels from different models interact during concurrent execution. The configuration of each kernel (its use of shared memory, registers, and other architectural resources) determines its potential for interference with others. Predicting these interferences cannot be based solely on aggregated model statistics or layer-level abstractions, as these overlook the fine-grained execution dynamics that govern actual performance. Instead, kernel-level analysis must consider both static resource requirements and temporal execution behavior to assess how overlapping workloads compete for limited GPU capacity. By modeling these interactions, it becomes possible to identify model pairs that minimize conflicts and make informed colocation decisions that preserve throughput and latency under constrained conditions.

4.2.3 Challenges in Modeling and Predicting Interference

Determining the best colocation strategy for DNNs sharing a GPU requires a nuanced understanding of how their kernels interact during inference. The goal is to identify model pairings that maximize throughput and minimize latency by avoiding harmful interference. Achieving this demands accurate performance modeling at the kernel level, where resource contention and execution overlap directly impact runtime behavior. However, building such models introduces several practical and computational challenges that must be addressed to ensure scalability and reliability.

Profiling overhead and trace collection. Profiling is essential for capturing the fine-grained execution characteristics of DNN kernels, including resource occupancy and duration. These metrics form the backbone of any interference-aware colocation strategy. Yet, collecting accurate traces is inherently time-consuming and introduces overhead that can distort the very measurements it seeks to record. Profiling tools rely on instrumentation and callbacks that add latency to kernel execution, often resulting in total recorded durations that exceed the actual inference time.

In practice, we observed that the ratio between profiled kernel durations and actual inference time can vary considerably, sometimes exceeding the true runtime by a wide margin, highlighting how profiling overhead may distort performance measurements and must be carefully considered. Despite these limitations, profiling remains indispensable, as predicting kernel behavior analytically is highly challenging due to the variability in runtime configurations and hardware-specific optimizations. What's more, CUDA Deep Neural Network library (cuDNN) [83] provides highly tuned implementations for standard routines such as convolution, attention, matrix multiplication, pooling, and normalization. For the same operator type, it may execute different kernels depending on the GPU architecture and available resources, with varying configurations in register usage, shared memory, and execution strategy. While this process is time-consuming, it remains feasible due to the limited number of GPU variants and the ability to profile models only in isolation.

Combinatorial Complexity of Kernel Overlap Scenarios. A key challenge in optimizing DNN colocation lies in defining and identifying the interference itself. Interference is not a static property: it arises dynamically when the kernels of different models overlap during execution and compete for shared GPU resources. To determine which kernels are likely to interfere, it is necessary to analyze not only their resource requirements, but also their temporal alignment and execution context. This is particularly difficult because interference depends on both the type and timing of resource usage, which can vary across deployments and hardware configurations.
This challenge is compounded by the vast number of possible overlapping scenarios. In realistic service environments, models do not begin inference in a synchronized manner; each can start at any point in its kernel sequence, leading to a vast space of potential execution alignments. Taking into account all combinations of starting positions across multiple models leads to exponential growth in the number of scenarios as the number of models increases. This combinatorial explosion makes exhaustive evaluation impractical for real-time decision-making. The challenge lies in estimating the impact of interference without simulating all possible alignments, while maintaining sufficient fidelity to guide effective colocation strategies.

To support efficient colocation of DNNs on shared GPU, we introduce Roomie, a kernel-level profiling and interference estimation strategy that balances precision with scalability. By analyzing execution traces in isolation and simulating representative overlap scenarios, Roomie enables informed deployment decisions without incurring prohibitive computational cost.

4.3 Related Work

With the proliferation of deep learning-based applications offered as online services, managing and scheduling large-scale inference workloads in GPU data centers has become increasingly critical. Unlike resource-intensive training workloads, inference tasks have distinct characteristics

that require specialized scheduling solutions. The objectives of inference scheduling (accuracy, latency, and cost-effectiveness) are closely linked. Accuracy efficiency involves selecting the most suitable model for each query and allocating resources intelligently. Latency efficiency requires meeting response time constraints, even in the face of irregular or fluctuating loads. Cost efficiency aims to minimize financial expenditure, particularly in public cloud environments. These objectives are often contradictory, requiring flexible and comprehensive planning systems capable of balancing trade-offs between different performance dimensions.

Inference Serving Systems. Several systems have been proposed to address the challenges of inference scheduling. Clipper [84] supports multiple ML frameworks and simplifies model deployment for real-time applications, but does not account for resource interference. TensorFlow-Serving [31] adapts to traffic changes by scaling replicas, yet similarly overlooks interference among colocated models. Clockwork [32] offers predictable performance by executing one inference at a time with full GPU capacity, but this design underutilizes GPU resources due to limited concurrency. Proteus [33] introduces adaptive batching and dynamic model variant selection to optimize throughput and accuracy. However, its restriction of one model variant per device limits co-location and parallel execution, leaving GPU resources under exploited.

Multi-Tenant DNN Inference on Shared GPUs. To improve resource utilization, recent work has explored concurrent execution of multiple models on shared GPUs. INFaaS [37] enables model-less serving by dynamically selecting variants based on query load and performance constraints. It co-locates variants and scales workers reactively, but struggles with profiling complexity and lacks proactive interference prediction. Colti [34] enhances throughput by colocating models for training and inference. Yu et al. [35] exploit operator-level independence to schedule concurrent execution across streams, reducing latency. REEF [38] supports kernel preemption and priority-based sharing to maintain predictable performance. Miriam [36] introduces elastic kernels for real-time prioritization, enabling flexible scheduling across tasks with varying criticality. While these systems optimize post-deployment execution, they often neglect initial placement strategies that could preemptively mitigate interference.

Interference-Aware Inference Serving. Recent studies have focused on proactive scheduling by modeling and predicting interference between concurrently running models. Mendoza et al. [39] propose a latency degradation model based on global buffer and PCIe utilization, but the coarse granularity limits predictive accuracy. Scrooge [40] profiles concurrency thresholds for identical DNNs to determine optimal co-location, yet its approach is infeasible for heterogeneous model combinations due to profiling overhead. Abacus [41] schedules operators from multiple models jointly to maintain QoS, but its hardware-agnostic duration model and reactive execution lead to underutilization and increased latency. iGnifer [42] adopts a low-level perspective, using GPU metrics

such as L2 cache usage and core launch counts to characterize interference. However, coarse indicators like power consumption prove less predictive. Usher [43] refines interference modeling by analyzing kernel occupancy and DRAM usage, distinguishing between compute- and memory-intensive workloads. Both Usher and iGnifer rely on NVIDIA's Multi-Process Service (MPS) [85] for spatial sharing, which limits their applicability in edge environments such as NVIDIA Jetson, where MPS is unsupported.

4.4 Kernel Interference-Aware Scheduling

This section explains the main concept behind Roomie. First, we introduce the idea of interference and describe a method for estimating it. Next, we present an algorithm that efficiently calculates the interference among various models. Finally, we detail a placement algorithm that utilizes the interference estimation from the first procedure to efficiently allocate new incoming models to the available GPUs.

4.4.1 Kernel Execution and Occupancy Modeling

Modern deep learning inference on NVIDIA GPUs relies on the efficient execution of CUDA kernels—parallel routines that process data across thousands of threads. Each layer (such as convolution, activation, pooling, or normalization) of a model typically launches one or more kernels. Put simply, DNN inference is the sequential execution of these kernels on the GPU, and their performance depends heavily on how well they utilize the GPU's hardware resources. Understanding this execution model is essential for analyzing and optimizing GPU occupancy, which reflects how many computational units are actively engaged during kernel execution. In this section, we describe the computation of the theoretical occupancy in three key steps. For further details on occupancy modeling and constraints, we invite the reader to refer to the CUDA documentation [86] and kernel tuning studies such as [87].

Warp configuration. CUDA threads are organized into warps, groups of 32 threads that execute instructions in lockstep. These warps are further grouped into blocks, which serve as the basic scheduling units on the GPU. The number of warps per block (W_b) depends on the kernel's thread configuration and remains fixed for that kernel. This value is the starting point for occupancy analysis, as it determines how many warps each block contributes to the SM. Understanding this structure is crucial because it defines the granularity of parallelism and the baseline resource demand per block.

Maximum blocks given by SM constraints. After establishing the number of warps per block (W_b), the next step is to determine the mximum number of blocks (b) that can be scheduled concurrently on a SM. This scheduling is governed by a combination of hardware limits and resource demands. While the SM has a fixed upper bound on the number of warps it can support (W), this limit is not always the only

element that influences b. Indeed, each thread within a block consumes registers and shared memory, and these resources are finite per SM. If the required registers or shared memory are exhausted before reaching the warp limit, the number of blocks that can be scheduled is reduced accordingly. In such cases, we say the kernel is *limited by registers* or *limited by shared memory*. The most restrictive constraint—whether warp count, register usage, or shared memory—ultimately determines the maximum number of blocks b that can be accommodated, as explained in [87].

Occupancy calculation. With both the warps per block (W_b) and the maximum number of blocks (b) determined, we can now compute the total number of active warps on the SM as $W_b \times b$. The theoretical occupancy (o) of a kernel is then calculated by dividing this value by the maximum number of warps the SM can support:

$$o = \frac{W_b \times b}{W} \tag{4.1}$$

Note that the occupancy formulation and its constraints are based on the model described in [87].

This ratio expresses the proportion of active warps relative to the SM's maximum capacity. Theoretical occupancy offers a clean upper bound on how effectively a kernel can utilize GPU resources—assuming ideal conditions and exclusive access to the hardware. Yet in practice, such isolation is rare. Real-world deployments often involve multiple kernels or models running concurrently, each competing for shared resources like registers, shared memory, and warp slots.

This concurrency introduces a new layer of complexity. The assumptions behind theoretical occupancy begin to break down as resource contention reshapes the effective scheduling landscape. To understand how performance degrades under these conditions, we must move beyond static occupancy formulas and examine how interference alters kernel execution.

4.4.2 Interference and Adjusted Occupancy

Interference occurs when one kernel prevents another from reaching its intended performance level due to simultaneous scheduling. In simpler terms, it arises when the total resource demand from multiple kernels exceeds the GPU's capacity. Formally, interference is present when the cumulative resource demand across all active kernels exceeds the hardware budget:

$$\sum_{k \in K} \varphi_k > \Phi \tag{4.2}$$

where K is the set of active kernels, φ_k denotes the resource usage of kernel k, and Φ represents the total resource capacity of the GPU. This formulation captures the condition under which resource contention begins to affect execution.

A kernel designed to operate at a certain occupancy may be forced to share resources, resulting in fewer warps or blocks than expected. This leads to reduced performance and increased latency. Depending on the severity of contention, kernels may still execute concurrently but with diminished efficiency—for example, a kernel that would normally schedule the maximum number of blocks allowed by its resource configuration may be restricted to fewer due to limited shared memory or register availability. In more extreme cases, one kernel may delay the other entirely, forcing sequential execution and increasing overall latency.

This degradation can be quantified by recomputing the occupancy of each kernel based on the reduced share of resources it receives under interference. When multiple kernels are scheduled concurrently, the number of blocks that can reside on an SM—and thus the number of active warps—is constrained by the availability of shared memory, registers, and warps. These resources, allocated across all active kernels, may limit each kernel to fewer blocks than its standalone configuration would allow. The recomputed occupancy, referred to as *adjusted occupancy*, captures how resource contention alters the kernel's execution profile and provides a basis for estimating performance loss in multi-kernel deployments.

The adjusted occupancy also depends on the specific scheduling policy used by the GPU to re-allocate resources for the models' kernels. However, as reported in Nvidia CUDA documentation [86], it is difficult to rely on a fixed policy for handling different concurrent operations. This translates to the impossibility of determining in advance a well defined order for the kernels' processing. For this reason, we approximate scheduling policies using simplified classical strategies such as equal partitioning, priority-based assignment, or first-come-first-served ordering. While these policies do not reflect the precise behavior of the GPU scheduler, they offer a practical basis for estimating performance under interference and evaluating deployment scenarios.

Let $\tilde{b}_k$ denote the adjusted number of blocks that kernel k can launch under constrained resources and this is determined, as derived from the GPU SM resource constraints outlined in Section 4.4.1. We compute the adjusted occupancy accordingly. With $\tilde{b}_k$, we can derive the new occupancy $\tilde{o}_k$ using the same formulation as in Equation 4.1. This allows us to estimate the new execution time of the kernel under interference:

$$\tilde{d}_k = d_k \times \frac{o_k}{\tilde{o}_k} \tag{4.3}$$

where d_k is the execution time of kernel k in isolation, and o_k is its original theoretical occupancy. In this way, the new execution time of kernel k is inversely proportional to its adjusted occupancy $\tilde{o}_k$, thereby modeling the effect of interference, where reduced kernel occupancy results in increased execution time. If $\tilde{o}_k = o_k$, then no interference occurs and $\tilde{d}_k = d_k$.

Interference ends once the first kernel completes its execution. Assuming at most two kernels are interfering, the remaining kernel continues alone without interference. We define the interference period as the time required for the first kernel to finish, $\Delta = \min \left\{\tilde{d}_k\right\}_{\forall k \in K}$. For the remaining kernel, its total duration is composed of two phases: the initial

interference period Δ, during which it runs with reduced occupancy $\tilde{o}_k$, and the remaining portion of its execution, which proceeds at full occupancy o_k. To account for the change in execution speed, we adjust the remaining time accordingly. Specifically, the portion $\tilde{d}_k - \Delta$, originally computed under reduced occupancy, is scaled by $\frac{\tilde{o}_k}{o_k}$ to reflect the normal execution once interference ends. Formally expressed:

$$\tilde{d}_k = \begin{cases} \tilde{d}_k & \text{if } \tilde{d}_k \leq \Delta \\ \Delta + \left(\tilde{d}_k - \Delta\right) \times \frac{\tilde{o}_k}{o_k} & \text{otherwise} \end{cases} \tag{4.4}$$

This unified notation allows us to express the adjusted duration for all kernels, whether they complete during the interference window or continue beyond it.

Performance Drop. Now that we have established a method for estimating interference among simultaneously executing kernels, we can generalize this approach to the inference phase of multiple DNNs. Each DNN launches a sequence of kernels, and we begin by aligning the first kernel of each DNN to form an initial set of concurrently executing kernels. This set is evaluated using the interference model described above. Once the first kernel in the set completes, it is replaced by the next kernel from the same DNN, and the process continues iteratively until all models have completed their execution, that is, until all final kernels have been processed.

For a model with an original inference time T and a total of q kernels, the new inference time under interference is given by:

$$\tilde{T} = \sum_{i=1}^{q} \tilde{d}_{k_i} \tag{4.5}$$

The performance drop experienced by model m due to interference is quantified by the relative increase in inference time. This is computed as:

$$\mu = \frac{\sum_{i=1}^{q} (\tilde{d}_{k_i} - d_{k_i})}{\sum_{i=1}^{q} d_{k_i}} = \frac{\tilde{T} - T}{T} \tag{4.6}$$

This formulation captures the cumulative slowdown introduced by resource contention across all kernels in the model's execution pipeline.

Kernel Alignment. We present a detailed analysis of how kernels from distinct DNNs interfere when executed concurrently on a shared GPU, and how this interference contributes to performance degradation. To fully characterize this behavior, it is necessary to consider how interference arises during execution. A key factor is the temporal overlap of kernel executions—referred to as alignment—which can occur at any point within a model's kernel sequence. This is the first point in the paper where we actually introduce the meaning of alignment. Perhaps we should give an idea of this concept also in the previous sections where we use this term? In other words, alignment determines which kernels from each model are likely to execute simultaneously and thus compete

for GPU resources. Each alignment scenario can lead to a distinct performance drop, making it essential to explore a wide range of configurations. Yet, the number of possible alignments grows rapidly with the number of models and kernel positions, making exhaustive evaluation impractical. This concept of alignment is central to understanding and modeling multi-model interference.

To address this, we introduce a greedy heuristic that approximates the impact of alignment without exhaustively enumerating all possibilities.

4.4.3 Greedy Algorithm for Estimating Model Interference

The analytical framework developed above provides a way to estimate performance degradation due to kernel interference across multiple DNNs sharing a GPU. However, applying this model exhaustively, by evaluating all possible combinations of kernel alignments across models, is computationally infeasible. For instance, while our previous formulation assumed that all models begin execution with their first kernel simultaneously, a more general scenario would allow each DNN to start from any of its i-th kernels (where $1 \leq i \leq q$). Enumerating all such combinations would require constructing a full Cartesian product of starting indices, which leads to exponential growth in the search space. For N concurrent DNNs, this results in a combinatorial explosion, making real-time evaluation impractical.

To address this, we propose a heuristic algorithm that approximates the interference impact efficiently. The pseudocode is presented in Algorithm 2. The key idea is to reduce the search space by:

1. Limiting the number of starting points per model to a subset of n evenly spaced indices from the full set of q kernels. 2. Focusing on *pairwise interference* rather than evaluating all N-way combinations.

For each model pair (m_i, m_j), we define a reduced set of starting indices S_i and S_j, and construct the set of concurrent execution scenarios as:

$$\mathscr{C}_{i,j} = S_i \times S_j \tag{4.7}$$

Each scenario corresponds to a pair of starting indices (s_i, s_j), which determine the positions in the kernel sequences where concurrent execution begins (line 8 in Algorithm 2). These serve as the basis for simulating localized interference effects between the two models.

This greedy, pairwise strategy dramatically reduces computational overhead while preserving the fidelity needed to estimate performance degradation due to kernel interference. This approximation is motivated by the need to avoid the exponential complexity of evaluating all possible kernel alignments across models. By considering only selected pairs of starting indices, we retain enough coverage to capture meaningful interference effects while keeping the simulation tractable.

Interference Simulation. For each pair (m_i, m_j), $i \neq j$, and each starting index pair $c_{i,j} = (s_i, s_j)$, we simulate kernel-by-kernel execution. Interference occurs when the combined resource demand of two concurrently executing kernels exceeds the available GPU capacity, as defined by the condition in Equation 4.2 : $\varphi_{k_i} + \varphi_{k_j} > \Phi$.

In such cases, the delay added to k_{s_i} (*i.e.*, the kernel identified by the starting point s_i) is:

$$\delta_{k_{s_i}} = d_{k_{s_i}} \cdot \frac{o_{k_{s_i}}}{o_{k_{s_i}} + o_{k_{s_j}}} \tag{4.8}$$

The total delay (additional time) for a given starting pair is:

$$\Delta^{c_{i,j}} = \sum_{s_i \leq t \leq q_i} \delta_{k_t} \tag{4.9}$$

We can define the representative additional duration as the median:

$$\Delta_{i,j} = \text{median}\left(\{\Delta^{c_{i,j}}\}_{\forall c_{i,j} \in \mathcal{C}_{i,j}}\right) \tag{4.10}$$

To account for the amount of time two models interact during execution, we introduce a scaling factor that reflects their relative kernel sequence lengths:

$$\gamma_{i,j} = \max\left(\frac{q_i}{q_j}, 1\right) \tag{4.11}$$

Determine the new duration The new duration after interference of model m_i is:

$$\tilde{T_{m_i}} = T_{m_i} + \sum_{\substack{j=1 \\ j \neq i}}^{N} \gamma_{i,j} \cdot \Delta_{i,j} \tag{4.12}$$

Finally, the performance drop can be determined as defined in Equation 4.6.

4.4.4 Placement Algorithm

The performance drop resulting from GPU resource sharing can be exploited to efficiently place new arriving models after deployment query for a new model variant or in case of upscaling meet the workload demand. When a new model arrives, the objective is to place it in a GPU g so that the average performance drop, denoted by $\bar{\mu}^g$, of all running models and the incoming model is minimal. Algorithm 3 describes the proposed procedure to place a new model that needs to be deployed. When a new model m_{arr} arrives, Algorithm 2 is applied sequentially to each GPU g (line 4 in Algorithm 3). The algorithm monitors the average performance drop across all deployed models to ensure it does not exceed an acceptable value, λ, which serves as a tunable parameter. The value of

Algorithm 2: Greedy Estimation of Model Interference

Function *performance_drop(M)*
  Generate starting index S_i for each model as Cartesian product of starting indices across all models.
  Initialize performance drop $\mu \leftarrow []$
  foreach *model* $m_i \in M$ **do**
    Initialize $\tilde{T_{m_i}} \leftarrow \sum d_k$
    foreach *model* m_j *where* $j \neq i$ **do**
      Initialize delay set $\mathcal{D}_{i,j} \leftarrow []$
      foreach *pair* $(s_i, s_j) \in S_i \times S_j$ **do**
        $k_i \leftarrow s_i, k_j \leftarrow s_j, \delta \leftarrow 0$
        while $k_i < q_i$ *and* $k_j < q_j$ **do**
          if $\varphi_{k_i} + \varphi_{k_j} > \Phi$ **then**
            Compute additional duration:
            $\delta \leftarrow \delta + d_{k_i} \cdot \frac{o_{k_i}}{o_{k_i} + o_{k_j}}$
          end
          Increment k_i, k_j
        end
        Append δ to $\mathcal{D}_{i,j}$
      end
      Compute overlap factor $\gamma_{i,j} \leftarrow \max\left(\frac{q_i}{q_j}, 1\right)$
      Update $\tilde{T_{m_i}} \leftarrow \tilde{T_{m_i}} + \gamma_{i,j} \cdot \text{median}(\mathcal{D}_{i,j})$
    end
    Finally, the performance drop.
    $\mu_{m_i} \leftarrow \frac{\tilde{T}_m - T_m}{\tilde{T}_m}$
    Append μ_{m_i} to μ
  end
  return μ
end

this parameter is chosen to ensure that the arriving model will not cause a significant performance drop in the already running models. Among all the candidate GPUs that satisfy the constraint $\bar{\mu}^g < \lambda$ (lines 7–9), the one that offers the lowest average performance drop is selected as the final deployment target. It is worth noting that this algorithm can be easily adapted to other objectives, *e.g.*, selecting the GPU that offers the highest throughput or lowest latency. If no suitable GPU is identified, the deployment is deferred.

Algorithm 3: Model Placement Algorithm

Function *schedule*(m_{arr}, G, λ)
  Initialize $p \leftarrow []$ be sequence of performance drops of new model m_{arr}.
  Initialize performance drop $\mathcal{P} \leftarrow []$
  foreach *GPU* g **do**
    $M^g \leftarrow M^g \cup \{m_{arr}\}$
    $\mu^g \leftarrow performance\_drop$(M)
    /* Ensure no variant has a performance drop beyond λ. */
    if $\bar{\mu^g} < \lambda$ **then**
      Append $\mu^g_{m_{arr}}$ to p
      Append μ^g to $\mathcal{P}$
    end
  end
  Peak lowest average performance drop.
  $g* \leftarrow \min \{\bar{\mu}^g\}_{\forall \mu^g \in \mathcal{P}}$
  return g^*
end

4.5 Evaluation

4.5.1 Experimental Setup

Baselines. We evaluate ROOMIE against two different state-of-the-art baselines: INFaaS and Usher. INFaaS performs accuracy scaling by scaling variants within the same worker to meet demand. Usher, on the other hand, groups compute-heavy models with memory-heavy models within a group. We have used different models from two family groups, namely classification and detection, to evaluate our solution and baselines. We consider that each incoming query is intended for a single inference model.

Deployment Infrastructure. We conducted our experiments using two distinct deployment types: a cluster of Jetson Nanos and a cluster of larger GPUs. The first consisted of 3× machines equipped with 4× Nvidia *A100-SXM4-40GB (40 GiB)*. The second consisted of 12× *Nvidia Jetson AGX Xavier* GPUs. Each GPU is assigned to a docker to form a server, making a total of 12 servers for each deployment. We also used 2× *HPE Proliant DL360 Gen10+* the client requesting the model and a controller responsible for serving requests to workers. The full specifications are presented in Table 4.1.

Datasets. We evaluated our system and baseline methods using both synthetic and real-world workloads. For the real workload, we adopt the Twitter trace 2020 dataset [88], as it is particularly suitable for modeling inference services, as tweets are commonly subjected to Deep Neural Network (DNN) processing before publication [33, 37]. Since the trace is aggregated at a coarse temporal granularity of one second, we apply a Poisson process to model intra-second arrival times and use a Zipf distribution to distribute queries among models, in line with established methodology [33, 37]. For synthetic workloads, we generated average request rates per second using a Gaussian process and applied the same

Model	CPU	GPU
Nvidia Jetson AGX Xavier	1 CPU/node, 8 cores/CPU	Nvidia AGX Xavier, Compute Capability (CC): 7.2
HPE Proliant DL360 Gen10+	x86_64, 2.40GHz, 2 CPUs/node, 16 cores/CPU	
Apollo 6500 Gen10+	x86_64, 1 CPU/node, 32 cores/CPU	4× Nvidia A100-SXM4-40GB (40 GiB), CC: 8.0
DL360 Gen10+	x86_64, 2.60GHz, 2 CPUs/node, 32 cores/CPU	

Table 4.1: Server configuration used for experiments.

Zipf-based model allocation. To ensure our evaluation captures a broad spectrum of inference behavior, we selected a diverse and representative set of DNN models. These include both high-performance classification architectures and widely adopted object detection frameworks, enabling us to rigorously assess system behavior under varied computational and latency profiles. The full list of models is summarized in Table 4.2, reflecting the breadth and relevance of our evaluation design.

Category	Models
Classification Models	`vgg19`, `alexnet`, `maxvit_t`, `resnet152`, `googlenet`, `densenet201`, `squeezenet1_1`, `mobilenet_v3_large`, `shufflenet_v2_x2_0`, `inception_v3`, `wide_resnet101_2`, `resnext101_32x8d`, `efficientnet_v2_l`, `convnext_large`
Object Detection Models	`ssd300_vgg16`, `fcos_resnet50_fpn`, `retinanet_resnet50_fpn_v2`, `fasterrcnn_resnet50_fpn_v2`, `ssdlite320_mobilenet_v3_large`

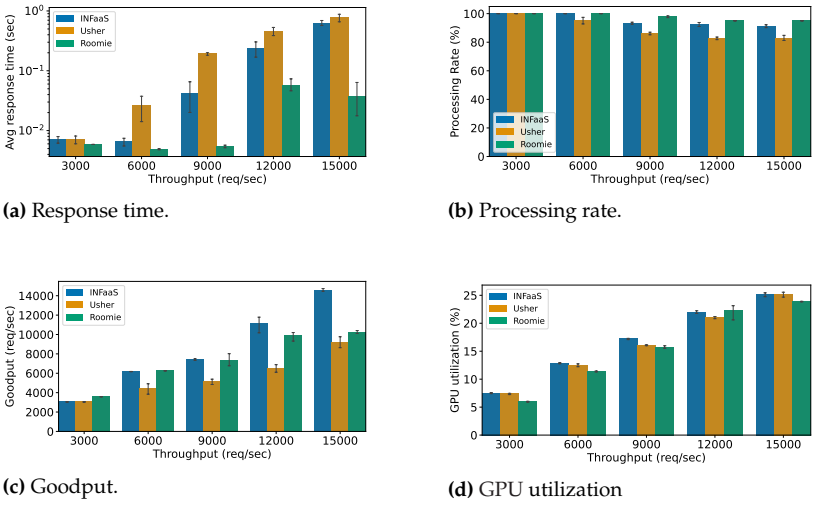
Table 4.2: Categorization of Deep Neural Network Models Used in Evaluation

Evaluation Metrics. To evaluate the effectiveness of each DNN deployment strategy, the assessment focused on four key performance indicators. Response time measured how quickly the models delivered results, which is a critical indicator of real-time performance. The processing rate indicated the percentage of requests successfully processed, highlighting the reliability of the system for different workloads. Throughput measured the number of requests processed in a given time frame, providing insight into overall capacity and scalability. Finally, GPU utilization monitored the degree of hardware resource usage during model execution.

4.5.2 Performance Evaluation of Cloud-Based GPU Cluster Solutions

To assess the effectiveness of our proposed deployment strategy, we conducted a comprehensive evaluation using a cloud-based GPU cluster comprising 12× GPUs and all DNN models detailed in Table 4.2. The experiments were performed using two distinct datasets: real-world

Twitter data and synthetically generated data. These synthetic workloads were constructed to mimic real-world traffic patterns while allowing precise manipulation of query rates. This enabled a more granular analysis of system behavior under stress. These datasets were used to simulate varying workload intensities, beginning with low traffic levels that all models could handle and gradually increasing until system saturation was observed.



(a) Response time.

(b) Processing rate.

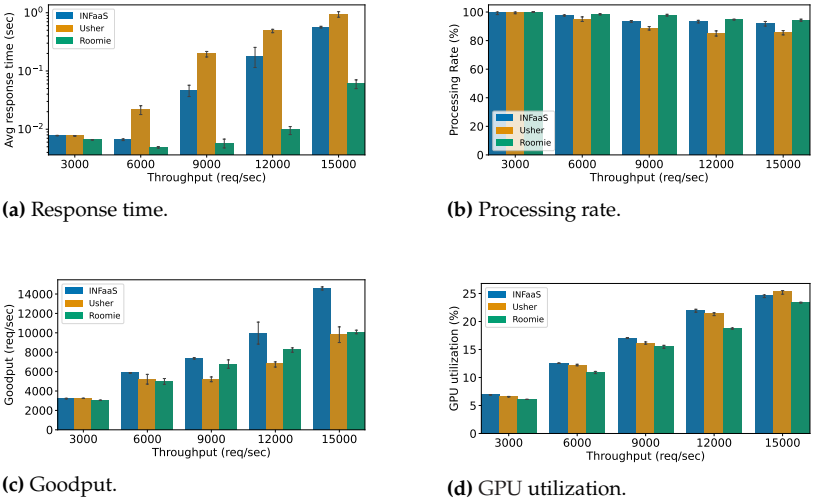
(c) Goodput.

(d) GPU utilization

Figure 4.2: ROOMIE achieves up to 17× faster response times while delivering similar processing rates in a cloud-based evaluation using the Twitter dataset, outperforming INFaaS and Usher under high workload conditions.

Evaluation on Twitter dataset. Figure 4.2 shows the performance results obtained from the Twitter database. Under low workload conditions, all approaches showed comparable response times. However, as the workload increased, significant performance disparities emerged. Notably, ROOMIE achieved response times up to 17× faster than rival solutions and maintained a processing rate above 97%. In contrast, Usher's strategy of co-locating large models with lightweight models failed to deliver satisfactory performance under high load, resulting in increased latency and reduced goodput. INFaaS, which scales DNNs on workers already hosting a copy, showed moderate goodput performance. However, this approach is agnostic to model interference, leading to latency increases of up to 17× and a decline in processing rate.

Interestingly, despite low GPU utilization across all approaches, we observe a significant increase in response time as workload intensifies. This counterintuitive behavior can be attributed to the saturation of large models, which become unable to keep pace with incoming queries. As these models reach their computational limits, they begin to queue requests, leading to latency explosions—even though the GPU itself remains underutilized. Additionally, interference between co-located models can further degrade responsiveness, compounding delays without a corresponding rise in GPU activity. One potential mitigation strategy is to increase batch sizes, which can improve utilization by amortizing overhead across multiple queries. However, this approach introduces a trade-off: larger batches may inflate response times, making it unsuitable for latency-sensitive applications. These findings underscore the need for deployment strategies that go beyond raw utilization metrics and account for model saturation dynamics and cross-model interference.



(a) Response time.

(b) Processing rate.

(c) Goodput.

(d) GPU utilization.

Figure 4.3: In cloud-based evaluation using synthetic workloads, Roomie yields 9.2× faster response time and higher processing rate, confirming its deployment efficiency in controlled stress scenarios.

Evaluation on synthetic dataset. Figure 4.3a illustrates the results of the evaluation conducted with a synthetic dataset designed to emulate diverse and controlled workload scenarios.
The performance trends observed with the synthetic dataset closely mirrored those seen with the Twitter data. Roomie consistently outperformed other deployment strategies, achieving a 9.2× reduction in response time and superior throughput and processing rates. These gains are attributed to Roomie's intelligent model deployment and colocation strategy, which minimizes interference and maximizes resource efficiency.

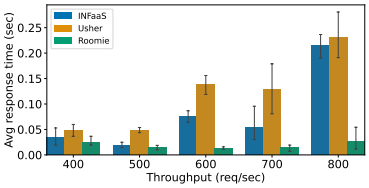
Overall, across both datasets, Roomie demonstrated robust performance under varying workload conditions. It consistently achieved lower response times—up to 17× faster, and higher processing rate than competing approaches, validating its effectiveness in cloud-based GPU environments.

4.5.3 Performance Evaluation on Edge Devices Using Jetson Xavier GPUs

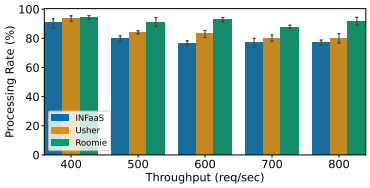
To further validate our approach, we conducted a second set of experiments using a cluster of 12 Jetson Xavier GPUs, representative of resource-constrained edge computing environments. As in the cloud-based evaluation, we deployed 12 models (from Table 4.2) and tested performance using Twitter data and synthetic data, gradually increasing the workload intensity.

Evaluation on twitter dataset. The Figure 4.4 presents the results obtained with the Twitter dataset on Jetson Xavier devices. While Usher and INFaaS exhibited similar throughput levels, INFaaS achieved lower latency, whereas Usher maintained a higher processing rate. Roomie, nevertheless, significantly outperformed both, achieving an 8.3× reduction in latency compared to INFaaS under high workload conditions. It also delivered superior throughput and processing rates due to its proactive colocation strategy.

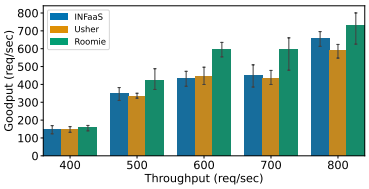
Edge environments impose more severe constraints on model deployment due to limited computing resources and reduced scalability. In this con-



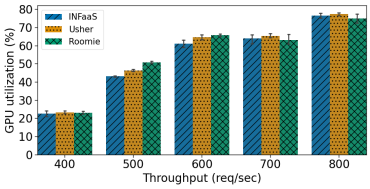
(a) Response time.



(b) Processing rate.



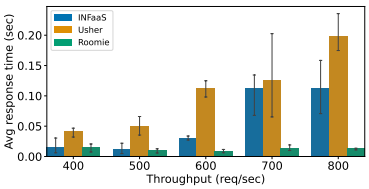
(c) Goodput.



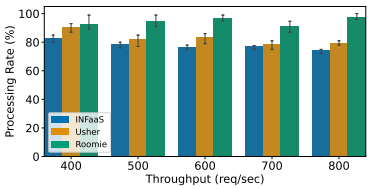
(d) GPU utilization.

Figure 4.4: Edge-based evaluation on the Twitter dataset shows ROOMIE delivering 8.3× lower latency and superior throughput on Jetson Xavier GPUs, validating its proactive colocation strategy under constrained resources.

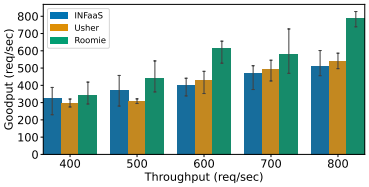
text, ROOMIE's colocation strategy offers a significant advantage, effectively balancing resource allocation and minimizing performance degradation. As workload intensity increases, GPU utilization increases accordingly, reaching levels significantly higher than those seen in cloud-based configurations. This underscores the critical importance of intelligent colocation policies in edge scenarios, where resource efficiency has a direct impact on system responsiveness and throughput.



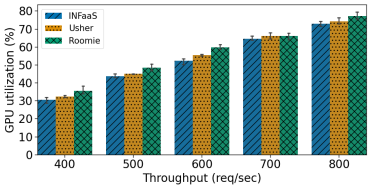
(a) Response time.



(b) Processing rate.



(c) Goodput.



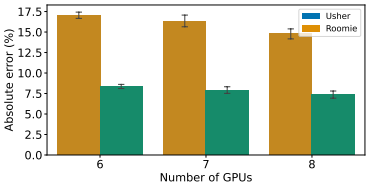
(d) GPU utilization.

Figure 4.5: Under synthetic edge evaluation, ROOMIE sustains 9× faster response time and 1.5× higher throughput, demonstrating robust performance in resource-limited environments.

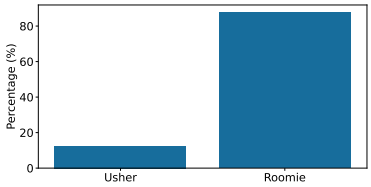
Evaluation on synthetic dataset. The synthetic dataset evaluation on Jetson Xavier GPUs yielded consistent results with those observed on the Twitter dataset. The results are shown in Figure 4.5. ROOMIE again demonstrated superior performance, achieving a 9× reduction in response time and a 1.3× increase in processing rate. Throughput was also 1.5× higher than competing approaches.

These findings confirm that ROOMIE is the most effective DNN deployment strategy in edge computing contexts where colocation is necessary and resources are limited. Its ability to maintain low latency and high throughput under constrained conditions makes it a compelling solution for real-time inference workloads.

4.5.4 Evaluating Roomie Deployment Accuracy Against Optimal Strategies



(a) Performance Gap Relative to Optimal Deployment.



(b) Proportion of Trials Where Roomie Outperforms Usher.

Figure 4.6: Roomie maintains deployment error within 7–8% of the optimal and outperforms Usher in nearly 90% of evaluated scenarios, demonstrating robust accuracy and generalization under high concurrency.

This section investigates the effectiveness of Roomie for the deployment of DNNs across a varying number of GPUs, specifically between six and eight. For each configuration, the number of DNNs to be deployed was randomly selected from a range of 2–3× the number of GPUs, guided by predefined options outlined in Table 4.2. More than 1500 randomized evaluations were conducted to ensure comprehensive coverage of deployment scenarios. In each evaluation, all feasible deployment permutations were thoroughly assessed to determine the configuration that resulted in the minimal average performance drop, defined as the optimal baseline. Both Roomie and Usher were then applied to the same scenarios, and their absolute errors relative to the optimal were recorded. The results are summarized in Figure 4.6. The Figure 4.6a illustrates the average performance gap across configurations, while the Figure 4.6b quantifies the proportion of trials in which Roomie outperforms Usher.

The comparative analysis highlights Roomie's consistent superiority in deployment accuracy across all concurrency levels. Its success in nearly 90% of randomized trials reflects a design that is not only structurally aware but also resilient to the practical limitations of kernel-level modeling. Unlike Usher, which applies static heuristics that overlook the dynamic nature of interference, Roomie adapts to the complexities introduced by concurrent execution. Crucially, the residual error observed in Roomie's deployments stems not from heuristic misalignment, but from the inherent challenges of profiling-based estimation. Tools such as Nsight-Compute, while indispensable for capturing fine-grained kernel behavior, introduce latency and measurement distortion that complicate performance inference. Roomie's strategy, based on the analysis of isolated traces and representative overlap simulation, effectively manages these distortions without resorting to exhaustive enumeration. Moreover, as concurrency increases, Roomie demonstrates robustness in the face of combinatorial explosion, where kernel alignment across models creates an exponentially growing space of interference scenarios. That Roomie maintains bounded error under these conditions affirms its capacity to balance fidelity with scalability, offering a principled alternative to heuristics that fail to account for architectural nuance.

4.6 Conclusion

Roomie consistently delivers high-performance deployments across both cloud and edge GPU environments. In cloud settings, it achieves up to 17× lower latency and maintains over 97% processing rate, outperforming

competing strategies through interference-aware colocation and saturation resilience. On edge devices, ROOMIE sustains its advantage with up to 9× lower response times and 1.5× higher throughput, effectively navigating resource constraints. ROOMIE's heuristic maintains deployment error within 7–8% of the optimal, even under high concurrency, despite challenges such as profiling overhead and architectural variability. These results hold across real-world and synthetic datasets, confirming ROOMIE's robustness and scalability. By intelligently minimizing cross-model interference and adapting to resource constraints, ROOMIE proves to be a practical and interference-aware solution for multi-model inference in modern GPU systems.

Conclusion 5

In conclusion, this thesis has presented two main contributions in the field of video analytics on distributed cloud platforms at the edge. We have introduced a new load balancing approach that adapts to heterogeneous camera sources, including fixed and mobile inputs. Furthermore, we developed a fine-grained model placement strategy that takes into account low-level GPU operations and proactively addresses interference between models. The remainder of this section revisits the central research question, summarizes our proposed solutions, discusses the limitations of our work, and offers recommendations for future research directions.

5.1 Research Problem and Summary of Contributions

We have discussed in this manuscript some issues related to live video analytics and, more importantly, when dealing with resource-constrained infrastructures. In Chapter 2, we discussed the many fields of research and application of edge-cloud computing, particularly video analytics applications, which are beginning to show their presence and importance in today's society. In this thesis, we have focused on two key points in particular: challenges to balancing computational demands, distributing video workloads across edge nodes with the integration of mobile cameras, and the deployment of DNN models in resource-constrained environments, where the co-location of multiple computing tasks is often required.

Our first contribution, presented in Chapter 3, identifies critical limitations in integrating mobile cameras with edge computing, primarily due to their unpredictable availability and dynamic movement, which compromise background subtractor and lead to unreliable object detection. These constraints are further exacerbated by fluctuating workloads and the need for rapid reconfiguration of processing pipelines as mobile cameras join or leave the network. To address these challenges, we introduce VideoJam, a novel system designed to integrate heterogeneous camera sources, both fixed and mobile, while maintaining low-latency performance. VideoJam incorporates short-term load prediction and function-level load balancing to manage high variability in computational demand. Empirical evaluations demonstrate that VideoJam achieves 2.91× lower response times, reduces video data loss by over 4.64×, and minimizes bandwidth overhead, thereby enabling a more adaptive and resilient video analytics architecture.

In the second part of our work, presented in Chapter 4, we examine deployment strategies for DNNs in resource-constrained edge environments, particularly in mandatory co-location scenarios. Although many approaches have been proposed to optimize inference performance, most

remain reactive, focusing on post-deployment planning or coarse profiling, and often fail to anticipate interference between models sharing GPU resources. These methods typically neglect the fine-grained execution characteristics of DNNs and rely on cloud-centric infrastructure, limiting their applicability to edge platforms. To address these shortcomings, we introduce Roomie, a kernel-aware orchestration system that profiles GPU kernel execution sequences to accurately model interference patterns. By leveraging these profiles, Roomie intelligently manages model placement, scheduling, and resource allocation across edge clusters. Experimental results demonstrate that Roomie significantly improves overall throughput compared to state-of-the-art systems.

5.2 Limitations

While VideoJam and Roomie successfully address key challenges related to edge video analysis and DNN deployment, both systems have limitations that leave room for improvement and further research. VideoJam does not take into account the bandwidths of inter-functional links when determining load balancing policies. In heterogeneous network environments, where link speeds vary significantly, this omission can adversely affect the overall efficiency of the system. Furthermore, VideoJam operates independently of the underlying deployment configuration (*e.g.*, the number of function replicas), but it lacks mechanisms to handle overload scenarios in which incoming video sources exceed available processing capacity. Moreover, VideoJam treats functions with similar objectives but different implementations (*e.g.*, YOLO vs. SSD) as identical entities, overlooking critical differences in accuracy, model size, and inference latency, factors that could significantly impact resource planning and allocation.

Roomie, on the other hand, faces scalability constraints, as its convergence time increases with the number of DNNs considered for interference. This poses challenges for real-time applications that require rapid adaptation. Furthermore, Roomie's current approach to profiling focuses primarily on kernel-level parameters, such as thread block size, shared memory, register usage, and occupancy. Its design does not yet incorporate a broader set of system-level factors that influence execution behavior, including PCIe bandwidth, L1/L2 cache performance, the number of SMs available to the GPU, and interleave latency between cores. These parameters, while less directly related to the core configuration at launch, play a crucial role in interference dynamics and overall system responsiveness.

Overall, these limitations highlight areas for future research and improvement, and we plan to address them in our future work.

5.3 Recommendations and Future Work

5.3.1 Recommendations

Our proposed systems, VideoJam and Roomie, have made a significant contribution to edge-based video analytics, but they are not without lim-

itations. We propose potential solutions to overcome these challenges.

VideoJam offers an efficient workload balancing technique. In a situation where all functions are overloaded, we could implement additional features such as a dynamic resource allocation mechanism that can adjust resource allocation in real-time based on changing system conditions and application requirements. Furthermore, the interference-aware scheduling algorithm, Roomie, could be used by VideoJam for these tasks to contribute to efficient and optimized DNN scheduling and minimize interference. These two solutions are highly complementary and could be integrated to improve the efficiency and effectiveness of DNN deployments in edge-based video analytics. Additionally, considering network state would enable the determination of the best load balancing policy by collecting transfer latencies (transmission) to know the available bandwidth.

To minimize the exponential latency of Roomie's algorithm, we could use a machine learning-based model, such as the one developed by Usher [43], to determine kernel duration. A possible initial choice would be a model that determines interference and predicts the final duration of any given kernel likely to interfere. This would then allow batch processing to be leveraged to minimize response latency. Alternatively, a model that takes into account all model sequences at once and predicts performance degradation for each of them could be used. We could utilize LSTM or Traffic Control Network (TCN) [67, 70], both of which have proven effective for time series data.

5.3.2 Future work

The next phase of this research will focus on building a unified orchestration framework that treats each application as a pipeline—a structured sequence of interdependent processing stages rather than isolated DNN deployments. This distinction is critical, as real-world video analytics applications typically involve multi-stage processing chains composed of heterogeneous functions, where tasks such as decoding, detection, tracking, and classification are tightly coupled within a unified pipeline. While VideoJam already embraces this pipeline-centric view, future work will extend it to support dynamic deployment, scaling, and coordination of entire pipelines under resource-constrained conditions. This direction also addresses limitations observed in prior systems such as SplitStream [89] and VideoStorm [10], which assume an abundance of edge devices and often overlook the complexities of co-location, interference, and adaptive scheduling.

Instance Lifecycle Management and Scaling Strategy. Inspired by the T-second lease mechanism introduced in SplitStream, which removes underutilized instances after a fixed interval, we propose a more structured approach to instance lifecycle management. For each application, a minimal set of instances, referred to as *primitive* instances, would be deployed and preserved regardless of system state. These serve as the baseline service assignment and are selected based on variant performance characteristics. Additional instances, introduced through scaling,

would be designated as *reserved* and subject to removal under two conditions: when the application is no longer saturated, thereby allowing resource reclamation, or when another instance requires upscaling and would yield greater performance benefits. Any downscaling event would trigger a reconfiguration of DNN placement to rebalance computational load and minimize interference across the system.

Pre-emptive Scaling and Rollback Mechanisms. To improve responsiveness under dynamic conditions, we envision a pre-emptive scaling strategy that anticipates saturation by identifying critical pipeline segments likely to require expansion. This includes instances positioned mid-pipeline or near the output stage, as well as those deployed last during initial configuration. Placement decisions within this framework will be guided by Roomie, whose kernel-aware profiling capabilities enable interference-aware scheduling across edge clusters. However, as identified in our limitations, Roomie may occasionally fail to provide optimal placement due to its reliance on kernel-level parameters and limited system-level context. To mitigate the impact of such suboptimal deployments, we propose the integration of a rollback mechanism. Following the deployment of a new variant, system performance will be monitored; if throughput degrades beyond acceptable thresholds, the orchestration system will revert to a previous configuration or selectively remove the variant(s) responsible. This feedback loop will enhance system adaptability and ensure that placement decisions remain aligned with performance goals.

Dynamic and Static Strategies for Batch Management. To further improve performance under dynamic deployment conditions, we propose two complementary approaches to batch size management. The first involves runtime batch size adaptation, where each variant enters a transient state following any change in worker configuration and dynamically recalibrates its batch size to optimize throughput and latency. This recalibration is performed collaboratively, with variants synchronizing to determine the most effective batch size for each, thereby avoiding contention and ensuring balanced resource utilization. The second approach leverages Roomie's batch-aware profiling capabilities to determine a fixed batch size during placement. Since Roomie captures how different batch sizes influence kernel execution patterns, it can select a configuration that minimizes interference and maximizes performance from the outset. By supporting both adaptive and static batch sizing strategies, the orchestration framework can tailor its behavior to the specific demands of each deployment scenario.

Special Terms

C
CNN Convolutional Neural Network. 32
CPU Central Processing Unit. 33
CUDA Compute Unified Device Architecture. 8, 14, 31, 49
cuDNN CUDA Deep Neural Network library. 47

D
DNN Deep Neural Network. viii, 1, 3, 6, 8, 14–16, 31, 33, 45, 46, 48, 49, 52, 53, 56–58, 60, 61, 63–66

G
GPU Graphics Processing Unit. v–vii, 1, 3, 6–9, 13–16, 23, 32–35, 38, 43–53, 56–64

I
IoT Internet of Things. 11, 43

L
LSTM Long Short-Term Memory. 28, 32, 65

M
ML Machine Learning. 8, 28

O
OCR Optical Character Recognition. 21, 32, 33, 35–40
OpenCL Open Computing Language. 8

S
SLO Service Level Objective. 43, 45
SM Streaming Multiprocessor. 8, 49, 50, 64
SSD Single Shot Multibox Detector. 41, 45, 64

T
TCN Traffic Control Network. 65

W
WRR Weighted Round Robin. vii, 33, 35–40

Y
YOLO You Only Live Once. vii, 21, 31–33, 35–41, 64

Bibliography

[1] Danlin Yu and Chuanglin Fang. 'Urban remote sensing with spatial big data: A review and renewed perspective of urban studies in recent decades'. In: *Remote Sensing* 15.5 (2023), p. 1307 (cited on page 5).

[2] Stéphane CK Tékouabou et al. 'Identifying and classifying urban data sources for machine learning-based sustainable urban planning and decision support systems development'. In: *Data* 7.12 (2022), p. 170 (cited on page 5).

[3] Miao Hu et al. 'Edge-Based Video Analytics: A Survey'. In: *arXiv preprint arXiv:2303.14329* (2023) (cited on pages 5, 12, 17, 23).

[4] Renjie Xu, Saiedeh Razavi, and Rong Zheng. 'Edge Video Analytics: A Survey on Applications, Systems and Enabling Techniques'. In: *IEEE Communications Surveys & Tutorials* (2023) (cited on pages 5, 12, 23).

[5] Haowen Xu et al. 'A mobile edge computing framework for traffic optimization at urban intersections through cyber-physical integration'. In: *IEEE Transactions on Intelligent Vehicles* 9.1 (2023), pp. 1131–1145 (cited on pages 5, 11).

[6] SK Alamgir Hossain, Md Anisur Rahman, and M Anwar Hossain. 'Edge computing framework for enabling situation awareness in IoT based smart city'. In: *Journal of Parallel and Distributed Computing* 122 (2018), pp. 226–237 (cited on pages 5, 11).

[7] Xiao Zeng et al. 'Distream: scaling live video analytics with workload-adaptive distributed edge intelligence'. In: *Proceedings of the 18th Conference on Embedded Networked Sensor Systems*. 2020, pp. 409–421 (cited on pages 5, 7, 12–14, 17–19, 21–24, 33–35).

[8] Junchen Jiang et al. 'Chameleon: scalable adaptation of video analytics'. In: *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*. 2018, pp. 253–266 (cited on pages 5, 12, 17, 19, 21–23).

[9] Chien-Chun Hung et al. 'VideoEdge: Processing Camera Streams using Hierarchical Clusters'. In: *2018 IEEE/ACM Symposium on Edge Computing (SEC)*. IEEE. 2018, pp. 115–131 (cited on pages 5, 13, 17, 19, 24, 39).

[10] Haoyu Zhang et al. 'Live Video Analytics at Scale with Approximation and Delay-Tolerance'. In: *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. Boston, MA: USENIX Association, Mar. 2017, pp. 377–392 (cited on pages 5, 12, 13, 23, 24, 39, 65).

[11] Lixiang Ao et al. 'Sprocket: A serverless video processing framework'. In: *Proceedings of the ACM Symposium on Cloud Computing*. 2018, pp. 263–274 (cited on pages 5, 17, 19, 23).

[12] Haoyu Zhang et al. 'Live video analytics at scale with approximation and delay-tolerance'. In: *14th USENIX Symposium on Networked Systems Design and Implementation*. 2017 (cited on pages 5, 17, 19, 23).

[13] Sadjad Fouladi and al. 'Encoding, fast and slow:{Low-Latency} video processing using thousands of tiny threads'. In: *Proc. of USENIX NSDI*. 2017, pp. 363–376 (cited on pages 5, 19, 23).

[14] Tiffany Yu-Han Chen et al. 'Glimpse: Continuous, real-time object recognition on mobile devices'. In: *Proceedings of the 13th ACM Conference on Embedded Networked Sensor Systems*. 2015, pp. 155–168 (cited on pages 5, 17, 19, 23).

[15] Wuyang Zhang et al. 'Hetero-edge: Orchestration of real-time vision applications on heterogeneous edge clouds'. In: *Proc. of IEEE INFOCOM*. 2019, pp. 1270–1278 (cited on pages 5, 13, 17, 19, 24).

[16] Mohammad Nishat Akhtar et al. 'Smart sensing with edge computing in precision agriculture for soil assessment and heavy metal monitoring: A review'. In: *Agriculture* 11.6 (2021), p. 475 (cited on page 11).

[17] Apri Junaidi et al. 'Deep Learning and Edge Computing in Agriculture: A Comprehensive Review of Recent Trends and Innovations'. In: *IEEE Access* (2025) (cited on page 11).

[18] Sourour Dhifaoui, Chiraz Houaidia, and Leila Azouz Saidane. 'Cloud-fog-edge computing in smart agriculture in the era of drones: a systematic survey'. In: *2022 IEEE 11th IFIP International Conference on Performance Evaluation and Modeling in Wireless and Wired Networks (PEMWN)*. IEEE. 2022, pp. 1–6 (cited on page 11).

[19] Dhananjay K Pandey and Richa Mishra. 'Towards sustainable agriculture: Harnessing AI for global food security'. In: *Artificial Intelligence in Agriculture* 12 (2024), pp. 72–84 (cited on page 11).

[20] Megha Sharma, Abhinav Tomar, and Abhishek Hazra. 'Edge computing for industry 5.0: Fundamental, applications, and research challenges'. In: *IEEE Internet of Things Journal* 11.11 (2024), pp. 19070–19093 (cited on page 12).

[21] Alexandru Rancea, Ionut Anghel, and Tudor Cioara. 'Edge computing in healthcare: Innovations, opportunities, and challenges'. In: *Future internet* 16.9 (2024), p. 329 (cited on page 12).

[22] Elarbi Badidi, Karima Moumane, and Firdaous El Ghazi. 'Opportunities, applications, and challenges of edge-AI enabled video analytics in smart cities: a systematic review'. In: *IEEE access* 11 (2023), pp. 80543–80572 (cited on page 12).

[23] Pavana Pradeep Kumar, Amitangshu Pal, and Krishna Kant. 'Resource efficient edge computing infrastructure for video surveillance'. In: *IEEE Transactions on Sustainable Computing* 7.4 (2021), pp. 774–785 (cited on page 12).

[24] Junjue Wang et al. 'Bandwidth-efficient live video analytics for drones via edge computing'. In: *2018 ieee/acm symposium on edge computing (sec)*. IEEE. 2018, pp. 159–173 (cited on pages 12, 17).

[25] Samvit Jain et al. 'Spatula: Efficient cross-camera video analytics on large camera networks'. In: *2020 IEEE/ACM Symposium on Edge Computing (SEC)*. IEEE. 2020, pp. 110–124 (cited on pages 12, 13, 17, 23, 24).

[26] Bailey J Eccles et al. 'DNNShifter: An efficient DNN pruning system for edge computing'. In: *Future Generation Computer Systems* 152 (2024), pp. 43–54 (cited on page 12).

[27] Sri Pramodh Rachuri, Francesco Bronzino, and Shubham Jain. 'Decentralized modular architecture for live video analytics at the edge'. In: *Proceedings of the 3rd ACM Workshop on Hot Topics in Video Analytics and Intelligent Edges*. HotEdgeVideo '21. New Orleans, Louisiana: Association for Computing Machinery, 2021, pp. 13–18. DOI: 10.1145/3477083.3480153 (cited on pages 12, 13, 23, 24).

[28] Hao Yuan et al. 'Online dispatching and fair scheduling of edge computing tasks: A learning-based approach'. In: *IEEE Internet of Things Journal* 8.19 (2021), pp. 14985–14998 (cited on pages 14, 24).

[29] Roland Kotto Kombi, Nicolas Lumineau, and Philippe Lamarre. 'A preventive auto-parallelization approach for elastic stream processing'. In: *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*. IEEE. 2017, pp. 1532–1542 (cited on pages 14, 24, 28).

[30] Daniel Crankshaw et al. 'Clipper: A Low-Latency Online Prediction Serving System'. In: *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. Boston, MA: USENIX Association, Mar. 2017, pp. 613–627 (cited on pages 14, 43).

[31] Christopher Olston et al. *TensorFlow-Serving: Flexible, High-Performance ML Serving*. 2017. URL: https://arxiv.org/abs/1712.06139 (cited on pages 14, 43, 48).

[32] Arpan Gujarati et al. 'Serving {DNNs} like clockwork: performance predictability from the bottom up'. In: *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation*. OSDI'20. USA: USENIX Association, 2020 (cited on pages 14, 43, 48).

[33] Sohaib Ahmad et al. '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. DOI: 10.1145/3617232.3624849 (cited on pages 14, 43, 48, 56).

[34] Jaiaid Mobin, Avinash Maurya, and M Mustafa Rafique. 'Colti: Towards concurrent and co-located dnn training and inference'. In: *Proceedings of the 32nd International Symposium on High-Performance Parallel and Distributed Computing*. 2023, pp. 309–310 (cited on pages 15, 48).

[35] Fuxun Yu et al. 'Automated runtime-aware scheduling for multi-tenant DNN inference on GPU'. In: *2021 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*. IEEE. 2021, pp. 1–9 (cited on pages 15, 48).

[36] Zhihe Zhao et al. 'Miriam: Exploiting elastic kernels for real-time multi-DNN inference on edge GPU'. In: *Proceedings of the 21st ACM Conference on Embedded Networked Sensor Systems*. 2023, pp. 97–110 (cited on pages 15, 48).

[37] Francisco Romero et al. 'INFaaS: Automated Model-less Inference Serving'. In: *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. USENIX Association, July 2021, pp. 397–411 (cited on pages 15, 43, 44, 48, 56).

[38] Mingcong Han et al. 'Microsecond-scale preemption for concurrent {GPU-accelerated}{DNN} inferences'. In: *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. 2022, pp. 539–558 (cited on pages 15, 48).

[39] Daniel Mendoza et al. 'Interference-aware scheduling for inference serving'. In: *Proceedings of the 1st Workshop on Machine Learning and Systems*. 2021, pp. 80–88 (cited on pages 15, 43, 48).

[40] Yitao Hu, Rajrup Ghosh, and Ramesh Govindan. 'Scrooge: A cost-effective deep learning inference system'. In: *Proceedings of the ACM Symposium on Cloud Computing*. 2021, pp. 624–638 (cited on pages 15, 43, 48).

[41] Weihao Cui et al. 'Enable Simultaneous DNN Services Based on Deterministic Operator Overlap and Precise Latency Prediction'. In: *SC21: International Conference for High Performance Computing, Networking, Storage and Analysis*. 2021, pp. 1–15. DOI: 10.1145/3458817.3476143 (cited on pages 16, 43, 48).

[42] Fei Xu et al. 'iGniter: Interference-Aware GPU Resource Provisioning for Predictable DNN Inference in the Cloud'. In: *IEEE Transactions on Parallel and Distributed Systems* 34.3 (2023), pp. 812–827. DOI: 10.1109/TPDS.2022.3232715 (cited on pages 16, 48).

[43] Sudipta Saha Shubha, Haiying Shen, and Anand Iyer. '{USHER}: Holistic Interference Avoidance for Resource Optimized {ML} Inference'. In: *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. 2024, pp. 947–964 (cited on pages 16, 43, 44, 49, 65).

[44] Rocco Parascandola. *New NYPD surveillance cameras to cover stretch of Upper East Side not easily reached by patrol cars*. Dec. 2018. URL: https://www.nydailynews.com/new-york/nyc-crime/ny-metro-argus-cameras-east-20181024-story.html (cited on page 17).

[45] Jonathan Ratcliffe. *How many CCTV Cameras are there in London? (Update for 2020/21)*. Nov. 2020. URL: https://www.cctv.co.uk/how-manfIPOy-cctv-camerfSECas-are-there-in-london/ (cited on page 17).

[46] Giulio Grassi et al. 'Parkmaster: An in-vehicle, edge-based video analytics service for detecting open parking spaces in urban environments'. In: *Proc. of the IEEE/ACM Symposium on Edge Computing (SEC)*. 2017 (cited on page 17).

[47] Ganesh Ananthanarayanan et al. 'Real-time video analytics: The killer app for edge computing'. In: *IEEE Computer* 50.10 (2017), pp. 58–67 (cited on page 17).

[48] Shanhe Yi et al. 'Lavea: Latency-aware video analytics on edge computing platform'. In: *Proceedings of the Second ACM/IEEE Symposium on Edge Computing*. 2017, pp. 1–13 (cited on page 17).

[49] Roy V Rea et al. 'Dash Cam videos on YouTube™ offer insights into factors related to moose-vehicle collisions'. In: *Accident Analysis & Prevention* 118 (2018), pp. 207–213 (cited on page 17).

[50] Elena Giovannini et al. 'Importance of dashboard camera (Dash Cam) analysis in fatal vehicle–pedestrian crash reconstruction'. In: *Forensic Science, Medicine and Pathology* 17.3 (2021), pp. 379–387 (cited on page 17).

[51] Chrisma Pakha, Aakanksha Chowdhery, and Junchen Jiang. 'Reinventing video streaming for distributed vision analytics'. In: *10th USENIX workshop on hot topics in cloud computing (HotCloud 18)*. 2018 (cited on page 19).

[52] Hang Qiu et al. 'Kestrel: Video analytics for augmented multi-camera vehicle tracking'. In: *2018 IEEE/ACM Third International Conference on Internet-of-Things Design and Implementation (IoTDI)*. IEEE. 2018, pp. 48–59 (cited on page 21).

[53] Jian He, Ghufran Baig, and Lili Qiu. 'Real-time deep video analytics on mobile devices'. In: *Proceedings of the Twenty-second International Symposium on Theory, Algorithmic Foundations, and Protocol Design for Mobile Networks and Mobile Computing*. 2021, pp. 81–90 (cited on page 21).

[54] Glenn Jocher. *YOLOv5 by Ultralytics*. Version 7.0. If you use YOLOv5, please cite it as below. DOI: 10.5281/zenodo.3908559 (cited on pages 21, 32).

[55] Enes Yildiz et al. 'Optimal camera placement for providing angular coverage in wireless video sensor networks'. In: *IEEE transactions on computers* 63.7 (2013), pp. 1812–1825 (cited on page 22).

[56] Hongpeng Guo et al. 'CrossRoI: cross-camera region of interest optimization for efficient real time video analytics at scale'. In: *Proceedings of the 12th ACM Multimedia Systems Conference*. 2021, pp. 186–199 (cited on page 22).

[57] Nada Ibrahim et al. 'A survey of performance optimization in neural network-based video analytics systems'. In: *arXiv preprint arXiv:2105.14195* (2021) (cited on page 23).

[58] Arthi Padmanabhan et al. 'Towards memory-efficient inference in edge video analytics'. In: *Proceedings of the 3rd ACM Workshop on Hot Topics in Video Analytics and Intelligent Edges*. 2021, pp. 31–37 (cited on page 23).

[59] Arthi Padmanabhan et al. 'Gemel: Model Merging for {Memory-Efficient},{Real-Time} Video Analytics at the Edge'. In: *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. 2023, pp. 973–994 (cited on page 23).

[60] Frank Cangialosi et al. 'Privid: Practical, Privacy-Preserving Queries on Public Video'. In: *19th USENIX Symposium on Networked Systems Design and Implementation*. 2022 (cited on page 23).

[61] Rishabh Poddar et al. 'Visor:{Privacy-Preserving} Video Analytics as a Cloud Service'. In: *29th USENIX Security Symposium (USENIX Security 20)*. 2020, pp. 1039–1056 (cited on page 23).

[62] Hao Wu et al. 'Pecam: privacy-enhanced video streaming and analytics via securely-reversible transformation'. In: *Proceedings of the 27th Annual International Conference on Mobile Computing and Networking*. 2021, pp. 229–241 (cited on page 23).

[63] Thomas Heinze et al. 'Auto-Scaling Techniques for Elastic Data Stream Processing'. In: *Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems*. DEBS '14. Mumbai, India: Association for Computing Machinery, 2014, pp. 318–321. DOI: 10.1145/2611286.2611314 (cited on page 24).

[64] Buğra Gedik et al. 'Elastic scaling for data stream processing'. In: *IEEE Transactions on Parallel and Distributed Systems* 25.6 (2013), pp. 1447–1463 (cited on page 24).

[65] Hang Zhang, Weike Liu, and Qingbao Liu. 'Reinforcement online active learning ensemble for drifting imbalanced data streams'. In: *IEEE Transactions on Knowledge and Data Engineering* 34.8 (2020), pp. 3971–3983 (cited on page 24).

[66] Ruchir Shah, Bhardwaj Veeravalli, and Manoj Misra. 'On the design of adaptive and decentralized load balancing algorithms with load estimation for computational grid environments'. In: *IEEE Transactions on parallel and distributed systems* 18.12 (2007), pp. 1675–1686 (cited on page 28).

[67] Klaus Greff et al. 'LSTM: A search space odyssey'. In: *IEEE transactions on neural networks and learning systems* 28.10 (2016), pp. 2222–2232 (cited on pages 28, 65).

[68] Varsha S Lalapura, Joseph Amudha, and Hariramn Selvamuruga Satheesh. 'Recurrent neural networks for edge intelligence: a survey'. In: *ACM Computing Surveys (CSUR)* 54.4 (2021), pp. 1–38 (cited on page 28).

[69] Romil Bhardwaj et al. 'Ekya: Continuous learning of video analytics models on edge compute servers'. In: *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. 2022, pp. 119–135 (cited on page 28).

[70] Philippe Remy. *Temporal Convolutional Networks for Keras*. https://github.com/philipperemy/keras-tcn. 2020 (cited on pages 28, 65).

[71] *asyncio — Asynchronous I/O*. https://docs.python.org/3/library/asyncio.html. 2024 (cited on page 31).

[72] Abdallah Wagih Ibrahim. *Vehicle Detection, CNN*. (Visited on 2023) (cited on page 32).

[73] *Tesseract*. (Visited on 2024) (cited on page 32).

[74] *NVIDIA Multi-Instance GPU User Guide*. URL: https://docs.nvidia.com/datacenter/tesla/mig-user-guide/index.html (visited on 2024) (cited on page 33).

[75] Lei Zhang et al. 'Batch adaptative streaming for video analytics'. In: *IEEE INFOCOM 2022-IEEE Conference on Computer Communications*. IEEE. 2022, pp. 2158–2167 (cited on page 34).

[76] Ziliang Lai et al. 'Top-K Deep Video Analytics: A Probabilistic Approach'. In: *Proceedings of the 2021 International Conference on Management of Data*. 2021, pp. 1037–1050 (cited on page 34).

[77] Shibo Wang, Shusen Yang, and Cong Zhao. 'SurveilEdge: Real-time video query based on collaborative cloud-edge deep learning'. In: *IEEE INFOCOM 2020-IEEE Conference on Computer Communications*. IEEE. 2020, pp. 2519–2528 (cited on page 34).

[78] Tarek Elgamal et al. 'Sieve: Semantically encoded video analytics on edge and cloud'. In: *2020 IEEE 40th International Conference on Distributed Computing Systems (ICDCS)*. IEEE. 2020, pp. 1383–1388 (cited on page 34).

[79] Wei Liu et al. 'Ssd: Single shot multibox detector'. In: *Computer Vision–ECCV 2016: 14th European Conference, Amsterdam, The Netherlands, October 11–14, 2016, Proceedings, Part I 14*. Springer. 2016, pp. 21–37 (cited on page 41).

[80] Mohammad Mustafa Taye. 'Understanding of machine learning with deep learning: architectures, workflow, applications and future directions'. In: *Computers* 12.5 (2023), p. 91 (cited on page 43).

[81] NVIDIA. *Nsight Systems Documentation*. Accessed: 2025-10-01. 2025. URL: https://docs.nvidia.com/nsight-systems/ (cited on page 45).

[82] PyTorch. *Torch Profiler Documentation*. Accessed: 2025-10-01. 2025. URL: https://pytorch.org/docs/stable/profiler.html (cited on page 45).

[83] NVIDIA Corporation. *cuDNN: GPU-Accelerated Library for Deep Neural Networks*. Version 8.2.1. 2025. URL: https://developer.nvidia.com/cudnn (cited on page 47).

[84] Daniel Crankshaw et al. 'Clipper: A Low-Latency Online Prediction Serving System'. In: *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. Boston, MA: USENIX Association, Mar. 2017, pp. 613–627 (cited on page 48).

[85] NVIDIA Corporation. *Multi-Process Service (MPS) Documentation*. Version v575. Accessed August, 2025. 2025 (cited on page 49).

[86] NVIDIA Corporation. *CUDA C++ Programming Guide*. Accessed: September 29, 2025. 2025 (cited on pages 49, 51).

[87] Robert V. Lim, Boyana Norris, and Allen D. Malony. *Autotuning GPU Kernels via Static and Predictive Analysis*. 2017. URL: https://arxiv.org/abs/1701.08547 (cited on pages 49, 50).

[88] Archive Team. *Twitter Streaming Traces*. Dataset. Mar. 2020. URL: https://archive.org/details/archiveteam-twitter-stream-2020-03 (cited on page 56).

[89] Yu Liang, Sheng Zhang, and Jie Wu. 'SplitStream: Distributed and workload-adaptive video analytics at the edge'. In: *Journal of Network and Computer Applications* 225 (2024), p. 103866 (cited on page 65).

UNIVERSITÉ SAVOIE MONT BLANC

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ SAVOIE MONT BLANC

Spécialité : **STIC Informatique**

Arrêté ministériel : 25 Mai 2016

Présentée par

Youssouph FAYE

Thèse dirigée par **Mohammad-reza SALAMATIAN** et
codirigée par **Francesco BRONZINO**

préparée au sein du **Laboratoire LISTIC**
dans **l'École Doctorale Sciences Ingénierie Environnement**

Distributed edge cloud architecture for executing AI based applications

Thèse soutenue publiquement le **18 décembre 2025**,
devant le jury composé de :

M., Pierre, SENS
Professeur à l'Université de Sorbonne, Rapporteur
M., Eddy, CARON
Professeur à l'Université Lyon 1, Rapporteur
Mme, Elisabeth, BRUNET
Maître de conférences à Telecom Sud Paris, Examinatrice
Mme, Sara, BOUCHENAK
Professeur à l'INSA Lyon, Examinatrice
M., Mohammad-reza, SALAMATIAN
Professeur à l'Université de Savoie Mont-Blanc, Directeur de thèse
M., Francesco, BRONZINO
Maître de conférences à l'ENS de Lyon, Co-directeur de thèse