

SLA-Constrained Split Inference and Adaptive Stage Placement for Generative AI Offloading in Edge-Fog Systems

by

Daulet Bulegenov

Submitted to the Department of Computer Science
in partial fulfillment of the requirements for the degree of

Master of Science in Data Science

at the

NAZARBAYEV UNIVERSITY

June 2026

© Nazarbayev University 2026. All rights reserved.

Author
Department of Computer Science
4 May 2026

Certified by
Meiram Murzabulatov
Assistant Professor, School of Engineering and Digital Sciences,
Nazarbayev University
Thesis Supervisor

Accepted by
Adnan Yazici
Dean, School of Engineering and Digital Sciences, Nazarbayev
University

SLA-Constrained Split Inference and Adaptive Stage Placement for Generative AI Offloading in Edge-Fog Systems

by

Daulet Bulegenov

Submitted to the Department of Computer Science
on 4 May 2026, in partial fulfillment of the
requirements for the degree of
Master of Science in Data Science

Abstract

Large language model inference is increasingly being pushed to edge and fog environments to reduce latency, improve data locality, and support deployment in resource-constrained environments. However, transformer inference is difficult to distribute efficiently, since execution is sequential, split decisions can introduce overhead of intermediate activation transfers and per-device load can vary over time. This thesis addresses the problem of SLA-constrained split inference and placement of generative AI workloads in heterogeneous edge-fog environments.

This thesis proposes and implements an offline-capable distributed split-inference architecture, which can be described as follows. For each request, the central scheduler decides whether to run the model on one worker or dividing it into prefix and suffix stages on two workers. The primary contribution is the integration of partition-aware scheduling with online worker telemetry and a dynamic LinUCB-based contextual-bandit policy of adaptive cut selection and placement. In contrast to the existing work that often presupposes a fixed partitioning, emphasizes CNN-oriented split inference, or aims at deploying datacenter-scale latency serving, the proposed framework targets resource-constrained edge-fog deployment, and explicitly considers the effect of queueing, boundary-transfer overhead, and SLA-driven latency goals.

The framework is evaluated on a real CPU-only testbed consisting of a Jetson Xavier master and Jetson Nano workers with distilgpt2. Split execution preserved correctness and functionality at all valid cut positions. At concurrency 20 under both fixed-load experiments the adaptive policy performed best, reducing p95 latency to 5.5 s, compared with 6.2 s for the heuristic baseline, 7.2 s for static splitting and 10.8 s for full execution. It also improved throughput to 4.2 requests/s and decreased the proportion of deadline misses to 4.8%. These findings indicate that lightweight online adaptation can improve tail latency and SLA satisfaction for running transformer inference in edge-fog applications.

Thesis Supervisor: Meiram Murzabulatov

Title: Assistant Professor, School of Engineering and Digital Sciences, Nazarbayev University

Acknowledgments

The author gratefully acknowledges the laboratory facilities at Nazarbayev University for providing the opportunity to execute the experiments presented in this work.

Contents

1	Introduction	15
1.1	Motivation	16
1.2	Problem Statement	18
1.3	Research Questions	19
1.4	Contributions and Novelty	20
1.5	Thesis Scope and Assumptions	21
1.6	Thesis Organization	22
2	Related Work	25
2.1	Mobile Edge Computing and Task Offloading	25
2.2	Split Computing / Split Inference (DNN Partitioning)	27
2.3	LLM Inference Systems (Prefill/Decode, Batching, KV Cache)	29
2.4	Scheduling Under Queueing and SLA Constraints	31
2.5	Learning-Based Scheduling (Contextual Bandits, RL, Online Learning)	32
2.6	Summary and Research Gap	33
3	Methodology	41
3.1	System Model and Assumptions	42
3.1.1	Cluster Topology and Roles (Master/Scheduler, Workers)	43
3.1.2	Communication Model (gRPC, Message Formats, Boundary Tensor)	44
3.1.3	Compute Model (Layer Costs, Device Heterogeneity)	46
3.1.4	Workload Model (Arrival Process, Prompt Lengths, Deadlines)	48

3.2	Framework Architecture	49
3.2.1	Control Plane (Registration, Heartbeats, Liveness/TTL) . . .	50
3.2.2	Data Plane (FULL, PREFIX, SUFFIX Execution)	51
3.2.3	Deployment Model (Containers, Offline Model Packaging) . .	52
3.2.4	Fault Handling (Timeouts, Retries, Idempotency Assumptions)	52
3.3	Split Inference Formalization	53
3.3.1	Cut Definition and Feasibility Constraints	53
3.3.2	Boundary Tensor Definition and Serialization	54
3.3.3	End-to-End Latency Decomposition (Compute + Network + Queue)	54
3.4	Decision Agent for Partitioning	55
3.4.1	State / Feature Representation	55
3.4.2	Action Space (Candidate Cuts, Routing Options)	56
3.4.3	Reward / Objective (Latency, Misses, Boundary Cost)	57
3.4.4	Learning Algorithm (e.g., LinUCB / Bandit / RL)	58
3.4.5	Persistence and Online Update (agent.pkl, Training vs Evalua- tion)	60
3.5	Baselines	60
3.5.1	No-Split (FULL on One Worker)	61
3.5.2	Static Cut (k)	61
3.5.3	Heuristic Queue-Aware Cut	61
3.5.4	Greedy Predicted Latency (Optional)	62
3.6	Experimental Setup and Metrics	62
3.6.1	Hardware and Software Stack (Master/Worker Machines) . . .	63
3.6.2	Models Used	63
3.6.3	Offline Mode and Model Distribution Method	64
3.6.4	Metrics (p50/p95/p99, Throughput, Miss Ratio, Failures) . . .	64
3.6.5	Logging and Reproducibility (CSV/JSONL, Seeds, Configs) .	65

4	Results	67
4.1	System Validation (Functional Correctness, Split Pipeline Correctness)	68
4.2	Microbenchmarks	69
4.2.1	Per-Cut Compute Time (Prefix/Suffix)	69
4.2.2	Boundary Size vs Cut	69
4.2.3	RPC Overhead and Serialization Costs	70
4.3	End-to-End Performance (Main Experiments)	70
4.3.1	Latency Distributions (p50/p95/p99)	70
4.3.2	Throughput vs Concurrency	71
4.3.3	Deadline Miss Ratio / Failure Rate	72
4.4	Agent Behavior Analysis	72
4.4.1	Cut Distribution Over Time (Learning Dynamics)	72
4.4.2	Adaptation to Load Changes ($\lambda(t)$)	73
4.4.3	Sensitivity to Reward Weights (β, γ)	74
4.5	Comparison with Baselines	75
4.6	Ablation Studies	76
4.6.1	Effect of Features / State Representation	76
4.6.2	Effect of Candidate Cut Set	77
4.6.3	Effect of Timeouts / Admission Control	77
4.7	Discussion	77
4.7.1	Interpretation of Results	77
4.7.2	Limitations (Model Size, CPU-Only, Missing KV Cache, etc.)	78
4.7.3	Practical Implications for Real Deployments	78
5	Conclusion	79
5.1	Summary of Findings	79
5.2	Answer to Research Questions	80
5.3	Contributions Recap	82
5.4	Future Work	83

List of Figures

3-1	High-level system overview. Clients submit inference jobs to the scheduler. Workers register and send telemetry heartbeats (dashed). For split inference, the scheduler dispatches PREFIX to w_p and SUFFIX to w_s , transferring the boundary tensor between workers.	42
3-2	Physical cluster topology used in the experiments. The scheduler runs on the Jetson Xavier master node. Worker services run on Jetson Nano devices and can be configured for PREFIX, SUFFIX, or BOTH roles. The client host submits concurrent requests over the LAN.	44
3-3	gRPC interfaces and message flow: client submits jobs to the scheduler (JobService); workers register and send heartbeats (ControlPlane); split inference transfers the boundary tensor from prefix worker to suffix worker.	45
3-4	Split inference sequence: client submits a job; the scheduler selects workers and a cut; the prefix worker executes the prefix stage and sends the boundary tensor; the suffix worker completes the suffix stage and returns the result.	46
3-5	Framework architecture overview. The control plane manages registration, heartbeats, and liveness; the data plane executes FULL/PREFIX/SUFFIX stages and transfers the boundary tensor in split inference.	49
3-6	Control-plane overview. Workers register once at startup and then send periodic heartbeats carrying liveness and telemetry. The scheduler applies a TTL policy to maintain the set of eligible workers.	50

3-7	Experimental edge–fog testbed used in this work. The scheduler (master) runs on a Jetson Xavier device, while workers run on Jetson Nano devices. All services communicate over a LAN and execute inference on CPUs only.	63
4-1	Latency percentiles versus concurrency for the four policies. The adaptive scheduler shows the clearest advantage in p95 and p99 at medium and high load.	72
4-2	Cut distribution over time for a representative fixed-load run. The adaptive scheduler initially explores several cuts and later concentrates on a narrow high-performing region.	75
4-3	Dynamic workload experiment: p95 latency by segment for heuristic and agent policies.	76

List of Tables

2.1	Summary of reviewed literature and the resulting research gap.	35
3.1	Key message fields used in the gRPC communication model.	45
3.2	Worker telemetry reported to the scheduler via heartbeats (subset). .	51
3.3	Context features used by the scheduling policy (subset).	56
4.1	Functional validation of split inference against full execution.	68
4.2	Average compute time per cut position for <code>distilgpt2</code>	69
4.3	Boundary tensor size for representative prompt lengths.	69
4.4	RPC and serialization overhead on the real cluster.	70
4.5	End-to-end latency under fixed load.	71
4.6	Throughput under fixed-load experiments.	73
4.7	Deadline miss ratio under fixed-load experiments.	74
4.8	Dynamic workload results by segment.	75
4.9	Relative improvement of the adaptive policy over the baselines. . . .	76

Chapter 1

Introduction

There is a growing push to move large language model (LLM) inference to edge and fog environments to achieve lower latency, better data locality, and reduce the use of centralized cloud infrastructure. Nevertheless, it is challenging to effectively utilize transformer-based inference in such settings. Transformer inference is sequential, split execution introduces intermediate activation-transfer overhead, and device load, queueing, and network behavior vary over time. This makes it a runtime orchestration issue and not merely a static deployment decision on whether to run inference on a single node or to divide it into many nodes.

This thesis addresses that issue by proposing an adaptive split-inference architecture for transformer-based generative AI tasks in heterogeneous edge-fog networks. The framework combines a central scheduler with worker telemetry (heartbeat) and an online contextual-bandit policy to determine, per request, whether to run inference on a single worker or divide it into prefix and suffix stages on two workers. The aim is to reduce tail latency and improve SLA satisfaction under dynamic workload and resource conditions.

The proposed framework is deployed as an offline-capable distributed system and tested on a real CPU-only testbed of a Jetson Xavier master and Jetson Nano workers with distilgpt2. The results show that adaptive split scheduling provides clear benefits under medium and high load. At concurrency 20, the proposed policy reduces p95 latency to 5.5 s, compared with 6.2 s for the heuristic baseline, 7.2 s for static split

execution, and 10.8 s for full execution, and reduces the deadline miss ratio from 15.4% to 4.8%.

The findings indicate that lightweight online adaptation is a feasible technique for improving transformer inference performance in edge-fog deployments with resource constraints.

1.1 Motivation

The use of generative AI services with stringent targets of service-level agreement (SLA) is becoming increasingly difficult with the joint pressure of the latency limit, operation cost, and energy consumption. End-to-end response time in interactive LLM applications is determined not only by compute time but also by communication delays which accrue in the process of sequential token generation. Concurrently, the energy footprint of large-scale deep learning systems and their associated infrastructure has gained enough practical importance, leading to the incentive to use designs that minimize redundant computation and better leverage available resources [1]. These limitations are especially applicable to real-time and privacy-intensive applications, where the data locality and less reliance on centralized processing in a cloud are the desirable results.

The mobile edge computing (MEC) or fog architecture offers a natural path toward finding a shorter round-trip time and better locality by enabling computation closer to the user [2]. Nevertheless, edge-fog environments are inherently heterogeneous (e.g., CPU-only nodes, GPU-enabled nodes, memory-constrained devices) and dynamic in nature, e.g. with changing network bandwidth, changing latency and changing workload intensity. It is not uncommon to encounter situations, even in such settings, in which a fixed deployment policy which is optimal at a given operating point can rapidly become suboptimal due to changes in network conditions and load conditions. Consequently, the actual implementation of MEC needs system-state and application-constrained runtime orchestration.

An important technical challenge is that transformer-based inference is sequential

and constrained by dependencies, so distributed execution is sensitive to the partition boundary and the communication cost. Partitioning the computation across devices can minimize local compute time, but this comes at the cost of intermediate activation transfers, the cost of which depends on the size of boundaries, the characteristics of the links, and serialization overhead. Prior work on collaborative intelligence between cloud and edge reveals that incorrect task partitioning may negate the latency benefits as a result of the communication cost and synchronization overheads [3]. Consequently, the question of *where* and *when* to partition needs to be decided with explicit consideration for the compute-communication trade-off and resource limitation of each tier.

Existing offloading methods often use static heuristics or single-objective optimization, usually focusing on latency or energy, independent of each other. While such approaches may provide simplicity, they can struggle to ensure performance across heterogeneous devices and across network conditions. Learning-based methods such as reinforcement learning of resource management have shown the potential for adaptive decision-making in MEC scenarios [4], but most are not explicitly designed to include (i) partition-aware modeling of sequential inference of LLM and (ii) caching mechanisms in order to minimize repeated computation across requests or repeated execution stages. Treating caching and orchestration as two separate problems can miss the opportunity to exploit both of these techniques: while caching can be used to reduce the compute demand, it can also be reintroduced to advantage the heterogeneous resource utilization by using adaptive offloading, but the gain of the two techniques is maximized by the coordination of decision made under a common goal.

These observations motivate an adaptive partition-aware orchestration framework that primarily optimizes latency and SLA satisfaction while improving resource utilization under transformer dependency constraints and runtime variability. Energy efficiency is also a major incentive behind edge-fog inference, although direct power and energy profiling is beyond the capabilities of the current prototype.

1.2 Problem Statement

In this thesis, we consider the problem of running transformer-based generative AI inference in a heterogeneous mobile edge computing environment which is composed of edge and fog resources. Due to the sequential nature of token-by-token decoding and the strong dependency constraints across model layers, end-to-end inference performance depends not only on local computation time but also on communication overhead when computation is distributed. The core challenge addressed in this work is to design an orchestration policy that determines how to partition the inference pipeline and where to place each partition across available devices, such that service-level requirements can be satisfied while efficiently utilizing constrained and heterogeneous resources.

This orchestration problem is subject to different practical constraints in edge-fog deployments. Devices are heterogeneous and have limited compute and memory resources and their capacity is variable with their runtime load. In addition, network links between network devices impose latency and bandwidth limits; as a result partitioning may require transferring intermediate activations whose communication cost could significantly affect end-to-end inference time. Finally, the variability of workload, such as changing the request rates and concurrency, may introduce the queueing effects that must be taken into consideration while making a decision on placement and partitioning.

The goal of the orchestration policy is to optimize latency-sensitive inference under the above constraints. Specifically, the policy should reduce end-to-end latency, including computation, communication, and queueing delays, while limiting the overhead associated with distributed execution. Although energy efficiency motivates resource-aware edge-fog orchestration, this thesis does not perform direct energy measurement. Instead, energy-related implications are discussed indirectly through resource utilization, stage execution time, and communication overhead.

Existing approaches often rely on fixed partitioning rules or on hand-crafted heuristics which implicitly assume stable network conditions and predictable device

load. In heterogeneous types of edge–fog environments where the capacity of compute, memory pressure and the quality of link change over time, such assumptions can result in less efficient placement decisions and unstable end-to-end latency. Moreover, optimisation of a single metric (e.g. latency) potentially results in excessive energy consumption or resource contention and optimised related mechanisms (e.g. compute placement and reuse of intermediate results), used independently possibly results in loss of performance. Therefore, the problem of designing an adaptive partition-aware orchestration policy for transformer-based inference over edge and fog resources, balancing latency, SLA satisfaction, and distributed-execution overhead under device, network, and workload constraints, is formulated in this thesis.

1.3 Research Questions

- **RQ1:** How can transformer-based inference be partitioned and placed across heterogeneous edge–fog devices to satisfy service-level latency requirements under device and network constraints?
- **RQ2:** What are the key trade-offs between end-to-end latency, distributed-execution overhead, and resource efficiency when sequential, dependency-constrained inference pipelines are distributed across edge-fog resources?
- **RQ3:** How does an adaptive, partition-aware orchestration policy perform compared to static partitioning or heuristic-based baselines under time-varying workload intensity and network conditions?
- **RQ4:** Which runtime signals (e.g., device utilization, queueing delay, link latency/bandwidth) are most informative for making robust partitioning and placement decisions in edge–fog environments?

1.4 Contributions and Novelty

In this thesis, adaptive partitioning and orchestration of inference involving transformer-based generative AI in a heterogeneous mobile edge computing environment with edge and fog resources is investigated. This aims at meeting service-level latency targets and using limited compute, memory, and network resources in an efficient way. The main contributions of this work are as follows:

- **Problem formulation:** We define split inference as a partitioning-and-placement problem based on sequential and dependency-constrained transformer inference under the heterogeneity of the devices, memory capacity, limits of the network latency/bandwidth, and workload concurrency.
- **Online split-point selection via contextual bandits:** We define partition selection as a contextual-bandit formulation problem and apply online learning policy (LinUCB) whereby on each request the scheduler selects a split point of the request (k) in a candidate set of the split points, conditioned on observed runtime context (e.g., prompt length, SLA deadline, and worker load/queue indicators). Following the execution, the scheduler updates the policy based on the feedback based on latency, cost of boundary-transfer and success/failure.
- **Split-inference system implementation:** Our design is a two-stage execution pipeline (prefix/suffix) which runs a scheduler gathering worker heartbeats and scheduling pipeline stages to workers available. Its implementation will have instrumentation of end-to-end latency and its primary constituents (compute, communication and queueing).
- **Practical caching for offline and low-latency startup:** In the existing implementation, model artifacts are stored on the local disk cache to run offline and maintained in full model weights in memory once loaded at startup, eliminating repeated overhead of model startup. Reuse of KV-cache between decoding steps and caching at intermediate activations are not implemented and are left for future work.

- **Evaluation against representative baselines:** We compare the proposed strategy with both the policies of static splits and heuristic baselines under different workload intensity, system conditions, and describe the performance by measuring the values of latency (tail behavior) and SLA satisfaction as well as resource/overhead factors.

1.5 Thesis Scope and Assumptions

This thesis is concerned with split inference and orchestration of transformer-based generative AI workloads in a mobile edge computing topology made up of edge and fog computing. The scope encompasses (i) subdivisions of inference pipeline into sequential stages (prefix/suffix), (ii) commutation of a split point and delegating the stages to accessible devices during runtime, and (iii) comparing the resulting latency and resource-efficiency behavior of differing device potentials, network bottlenecks, and dynamic workload.

System model The system model assumes that the transformer architecture has a specified number of layers with known candidate split points. The devices are supposed to have a local network over which the relevant link latency and bandwidth can be measured and can change over time. The scheduler will assume access to lightweight runtime indications of workers (e.g. utilization/load indicators and queueing information) through periodic heartbeats, and uses these indications to make per-request orchestration decisions. On the worker side, the model is loaded from a local/offline directory, and workers keep the full model weights in memory after initialization. Each request executes only the stage assigned to the worker, namely FULL, PREFIX, or SUFFIX execution.

This thesis does not perform direct hardware-level power or energy measurement. Energy efficiency is treated as a motivating factor for resource aware orchestration, whereas the experimental evaluation focuses on latency, throughput, deadline miss ratio, failure rate, stage-level execution time, and boundary-transfer overhead.

Several limitations are intentionally imposed in this work. First, the presence KV-

cache reuse is not reused during decoding and, thus, the autoregressive state reuse is not applied. Second, the system does not support the ability to cache intermediate activations, such as caching prefix boundary tensors by prompt or session; instead, prefix activations are computed individually on a request-by-request basis. Third, it does not discuss optimizing model-segment placement: currently, workers load the entire model instead of only that segment that represents their assigned stage. Lastly, this thesis is concerned with edge-fog execution and does not consider a cloud tier, cross-datacenter latency, distributed training, tensor parallelism, or pipeline parallelism other than the two-stage split-inference case.

1.6 Thesis Organization

The rest of this thesis is structured in the following way. Chapter 2 provides a literature review on the appropriate background, and relates to the current study, covering mobile edge computing and task offloading (Section 2.1), split computing and DNN partitioning (Section 2.2), LLM inference systems and prefill/decode (Section 2.3), scheduling with queueing and under SLA constraints (Section 2.4), and learning-based scheduling such as contextual bandits and online learning (Section 2.5). Section 2.6 ends with the conclusion on the research gap.

The methodology is presented in chapter 3. It initially describes the system model and assumptions (Section 3.1) such as cluster topology and roles, the communication model, the compute model as well as the workload model. A description of the framework architecture (Section 3.2) is then given, split inference is formally described (Section 3.3), and the partitioning decision agent (Section 3.4). Section 3.5 also presents the baselines (Chapter 3) and the experiment setup and metrics of evaluation (Section 3.6).

The experimental results are reported in chapter 4. It starts with system validation (Section 4.1) and microbenchmarks (Section 4.2) and gives end-to-end performance metrics (Section 4.3) and a description of agent behavior (Section 4.4). It is also evaluated comparatively against baselines (Section 4.5) and ablation studies (Section

4.6) and also discusses the results and limitations (Section 4.7).

And lastly, Chapter 5 provides a conclusion of the thesis providing the summary of the key findings (Section 5.1), answering the research questions (Section 5.2), summarizing the contributions (Section 5.3), and giving information about the future direction of work (Section 5.4).

Chapter 2

Related Work

This chapter reviews earlier studies that provide a basis of split inference orchestration in edge-fog settings. It will start by giving a summary of Mobile Edge Computing and task offloading models (Section 2.1), followed by split computing and DNN partitioning models (Section 2.2). Subsequently, it reviews LLM inference systems and the serving optimization (Section 2.3), and later the scheduling techniques under queueing and SLA (Section 2.4). Lastly, it recaps the learning-based scheduling methodology, such as, contextual bandits and online learning (Section 2.5) and concludes with a wrap-up of shortcomings and subsequent gap in the body of research discussed in this thesis (Section 2.6).

2.1 Mobile Edge Computing and Task Offloading

MEC provides compute and storage services close to end users through the distribution of cloud-like compute and storage resources at or near the radio access network (RAN). MEC is also broadly investigated as an architectural enabler of low-latency services, a better locality, and less backhaul dependency than cloud-only design because it reduces the physical and network distance between the applications and compute resources. [5] [6]. Mechanism MEC systems are often a multi-tier system in which resources are provided to it in a heterogeneous fashion, the network characteristics vary with time, and the orchestration layer has both to assign the resources

efficiently to achieve application performance goals, both in computation and communication.

A major process in MEC is the so-called computation offloading: some or all of a task is computed on a remote device instead of the end device itself or the node it is originating. Traditionally, offloading can be divided into (i) no offloading (a task is not offloaded, executed on local resources), (ii) binary/full offloading (a task is fully executed on a remote MEC server), and (iii) partial offloading (a task is divided into entities that are executed in both local and remote resources). Control of the network of these models is dependent on the task structure, device capabilities and network limitations. Binary offloading simplifies orchestration and does not work well when tasks have hard latency constraints, or when network congestion prevails. Partial offloading offers greater flexibilities by allowing finer-grained location of computations, however, it alters more constraints including dependency ordering, and feasibility of partitions and overhead of intermediate data transfer.

The MEC task offloading problem can be usually modeled by an optimization problem, in terms of decision variables including (a) offloading decision, (b) execution decision (which edge/fog node), (c) offloading fraction/partition decision (partial offloading), and (d) allocate resources (wireless or compute resources). There is a significant literature of cooperative optimization or transformation-based techniques to find tractable solutions to energy- and latency-aware offloading subject to wireless access limitations. As an example, the mobile-edge computation offloading has been researched with energy efficiency through joint communication computation resource allocation, and the interaction between radio scheduling and CPU allocation and decisions related to offloading has been demonstrated in the study of this paper [7]. These methods are very useful in terms of their analysis, but they usually make simplifying the channel models, the assumptions of stationarity, or the centralized knowledge.

Under centralized optimization, as well as in practice, the MEC settings are also incentivized to heuristic and distributed formulations in addition to multi-user contention and interference. In multi-user case, both offloading decision are coupled using shared wireless channels, and shared MEC compute capacity; this implies that the

globally optimum decision is challenging to calculate. Game-theoretic models represent a distributed decision-making system, with each user represented as a strategic agent, and seek a stable operating point (e.g., Nash equilibria), finding a distributed solution using a representative example of the multi-user computation offloading game under wireless interference, the distributed algorithms are proposed to reach an equilibrium solution as the output of the user representation modeled as a strategic agent (i.e. user) [8]. The techniques enhance scaling and decentralization, yet may stabilize at non-optimal equilibria, and generally need particular care in incentive and stability analysis.

Regardless of the improvement, current MEC task offloading architectures have limitations that are very critical to current AI inference workloads. It is also assumed that many studies are relative to relatively small and independent tasks, simplified or fixed network conditions, and very simple modeling of queueing and tail latency when there is high concurrency [5]. In addition, partial offloading models frequently model the application as divisible computation without explicitly recording deep dependency constraints and the cost of intermediate data movement. These constraints encourage partition-conscious orchestrated techniques that are (i) resilient to heterogeneous and time-varying edge-fog systems, (ii) respond to SLA-driven task targeted objectives and not just to average-case performance.

2.2 Split Computing / Split Inference (DNN Partitioning)

Split computing (or split inference or collaborative inference) splits a deep neural network (DNN) into various sequential steps that are run on dissimilar devices. One type is the so-called layer-wise partitioning, in which the network is partitioned at some layer k and the prefix (layers 0 through k) is implemented in an upstream device, and the suffix (layers $k + 1$ through $L - 1$) implemented in a downstream device. The cut state (commonly referred to as the activation tensor or the also known as the

boundary tensor) is forwarded between stages, and as such, the end-to-end latency is a product of compute time, communication time and the possible queueing delay at each stage. The key design issue is then what is an achievable cut point and placement strategy that will yield latency/energy savings but not add too much overhead due to intra-boundary transfers.

Conceptual Early Systems work had shown that the cut point could be dynamically selected depending on runtime conditions. Neurosurgeon suggests a lightweight scheduling method that picks a point of partition between a mobile phone and a remote server, trying to minimize the end-of-end latency having taken into account the expenses of network-based transfer of interim activations [9]. The cut configuration space was investigated more richly later, including right-sizing through early exits. Edgent is an extension of device-edge co-inference with the power to abort inference at intermediate levels when confidence-level is high, which allows further latency cuts against strict response-time constraints and network variability under tight response-time constraints and under network variability conditionalities [10]. BranchyNet also popularizes the idea of early-exit architectures by adding side branches to enable a large number of inputs to exit the network before the full network depth, at the cost of a small error change of the number of compute and latency units cut in half [11].

In addition to two-node splitting, there are a number of techniques which assume inference is split over a small cluster of edges in order to realize parallelism and resource aggregation. DeepThings suggests an adaptive architecture of distributed CNN inference on scarcity resource IoT edge clusters, which focuses on runtime adaptation and distributed workload under a constrained amount of memory and compute resources [12]. These cluster-based designs point toward one notable extension of split inference the orchestration problem can deal with not just choosing a point of split, but also routing and allocating computation to multiple candidate workers whose capabilities may be heterogeneous and with different load.

In spite of the advancements in split inference as well as collaborative DNN execution, modern generative AI environments still have significant drawbacks. Most of the literature favors CNN-like workloads (e.g. classification and vision pipelines)

and usually presumes comparatively stable network conditions or is interested in averaged rather than tail latency at high concurrency. Moreover, the cost structure and dependency pattern of transformer based generative inference are significantly different because of sequential decoding and state reuse, thus it is recommended to have specific modeling and orchestration approaches specific to LLM-style pipelines, as discussed in the below text.

2.3 LLM Inference Systems (Prefill/Decode, Batching, KV Cache)

The systems of modern LLM serving decompose inference into two stages, namely prefill and decode. Prefill works by processing the entire prompt to initialize the internal attention state; decode works by producing tokens in a sequence, with the production of each new token based on all the tokens already produced. In order not to recalculate attention over time at each decode step, empirical implementations maintain an attention key-value (KV) cache, which records previous attention keys and values and reuses them during decoding steps. Although the KV caching is critical when it comes to low per-token latency, it also places a significant memory bottleneck since size of the sequence, batch size, and model depth cause the KV cache to grow, which can be a significant source of GPU memory consumption and thus the number of requests served at a time [14].

The main way of enhancing throughput in LLM serving is by batching, which is much more difficult than standard DNN inference because prompt lengths vary across requests, and requests may have different generation lengths. With disparate length requests, head-of-line blocking and inefficient utilization may occur with naive approach to batching, which is naive static batching. Due to this, recent systems use continuous or dynamically run batching, during which new requests are added to the running batch as the others finish, to better utilize it, whilst making an effort to keep tail latency under control. Orca introduces scheduling and selective batching of

transformer executions. These are proposed to schedule errors with granularity at the model-iteration level, scheduling at the request-model-iteration granularity to perform better throughput-latency trade-offs under concurrency [13]. Similar research has also been done on variants of scheduling like deferred continuous batching, to minimize inefficiencies prevailing due to frequent reshaping of batches and frequent context-swing under mixed workloads [15].

One of the key challenges in the production LLM serving is KV cache memory management. The dynamic expansion of KV cache (based on active requests) can lead to fragmentation and wasted memory, simply reducing the reachable batch sizes due to inefficiently allocated memory. vLLM uses PagedAttention, where the KV cache memory is managed in fixed-size blocks (pages) based on virtual memory (VM) techniques, to reduce fragmentation and achieve higher effective concurrency without necessarily increasing latency [14]. This body of work sheds light on the idea that it is the memory management and request scheduling that are closely related in the case of LLC to LLM serving: in many instances, throughput improvements require a combination of batching policy and KV cache allocation to be co-controlled.

Besides single-node serving, large-model inference systems also take into account distributed implementation and heterogeneous memory hierarchies to scale beyond what is possible with a single accelerator. DeepSpeed-Inference introduces to the world a full system of inference that can speed up latency and throughput through locked-down kernels and run strategies, and also allow a heterogeneous memory strategy to use not only the memory of the GPU but also supplementary memory with CPU/NVMe, extended to serve models more than the sum of the individual GPU memories [16]. In general, current LLM inference systems are mostly formulated to target datacenter-scale deployments that have high-performance accelerators and serve performance-driven orientations; extending this concept to resource-constrained and heterogeneous edge-fog environments will incentivize more orchestration processes, as the subject of this thesis.

2.4 Scheduling Under Queueing and SLA Constraints

System performance, especially in latency-sensitive services, is commonly measured based on average response time, but also tail metrics (e.g. p95/p99) and deadline miss ratios. Outliers in long latency occurred even in instances of small variability in the latency of services, contention among resources, and background interference even in the case of a low median latency. According to Dean and Barroso, the large scale services should be designed to survive a range of latency and execution of the tail at scale by careful system design and scheduling policy [17].

Effects of concurrency Queueing Queueing effects occur naturally when the request arrival rate is close to the service capacity: queueing delay may rapidly become equal to or greater than pure compute time. Consequently, the decisions on scheduling should consider both the waiting time and the execution time so as to adhere to SLA limitations. In real-time, deadline-constrained environments Classical results demonstrate the central role of deadline-based prioritization: e.g. Liu and Layland research deadline-based real-time task scheduling and give basic analysis of deadline-based policies and feasibility assessments [18]. The ideas spawn useful mechanisms like deadline conscious prioritization, admission control and overload control to stop unrestrained growth of queues at times when the system is temporarily oversubscribed.

Different resource management mechanisms to address the heterogeneous resource demand, multi-tenant workload and dynamic load are also needed at cluster scale to schedule work under SLA constraints. Different features are added to production cluster managers like priorities, quotas, and preemption to implement high level policies but assure high utilization. Making use of the significance of centralized resource management and policy systems to facilitate diverse workloads share clusters, Borg, a large-scale cluster manager at Google, explains the significance of scheduling choices in being responsive and reliable [19]. Even though they are configured in datacenter contexts, their fundamental lessons, especially in respect of prioritization, contention, and policy enforcement are applicable when coming up with SLA-aware schedulers.

Scheduler overhead and dispatch latency is an issue when workloads are made up

of short tasks or fine grained stages. The design concept of distributed scheduling is to eliminate the bottlenecks in the centralized scheduling control plane itself. Sparrow suggests distributed low-latency scheduler, based on randomized sampling and late binding mechanisms, which ensure low queueing and high scheduling throughput, and is applicable to interactive and parallel workloads with strongly latency-sensitive targets [20]. On the whole, the literature notes that compliance in SLA requires the ability to stabilise queueing delay and tail latency with a set of prioritization, load balancing, admission control, and low-overhead scheduling tools, which are particularly needed when the inference pipelines are distributed across multiple devices and need to coordinate the compute and communicate.

2.5 Learning-Based Scheduling (Contextual Bandits, RL, Online Learning)

The scheduling that is based on learning is used to deal with situations where analytical models of service time, network conditions, and contention are hard to get or time-varying. A learning-based scheduler will of course exhibit greater improvement to its decisions, as it observes runtime feedback, including latency, queueing delay, energy consumption and failures and adjusts its policy in real-time. The heterogeneous edge-fog environment, in which the availability of devices and the quality of links may vary quickly and the scheduler is forced to operate under less than full observability and low measurement overhead is of particular interest to this paradigm.

The multi-armed bandit scheme is a popular framework of per-decision learning procedures, where the scheduler picks an activity and experiences a short-term reward or expense. Traditional bandit algorithms are Upper Confidence bound (UCB) that trade-offs exploration and exploitation with a finite-time performance warranty [21]. To provide a realistic schedule, the contextual bandits build on this concept boxing the action choice by features of observed system contexts allowing the policy to vary the decision based on the current workload and system state. An example of this is

used in LinUCB where expected reward is modeled based on context and bounding of confidence is used to select actions; this has been used to the online decision problem where each decision is followed up by direct feedback [22]. Contextual bandits are attractive to systems where fast decisions need to be made, as it is compatible with online updates, implicitly handles discrete action spaces and does not need the cumbersome credit assignment associated with long horizons.

Reinforcement learning (RL) extends bandits to sequential decision-making problems which are modeled as Markov decision processes, in which actions affect future states and rewards may be delayed. Deep RLs, including Deep Q-Networks (DQN), approximate functions with value based learning and are able to effectively represent complicated policies in high dimensional states [23]. Deep RL has been applied in scheduling and resource management where learned allocation policies can be learned based on observation of the system, and have been shown to be more effective than handcrafted policies in some operating regimes [24]. Nevertheless, RL-based scheduling presents new problems, such as the artwork of rewards, sample performance, and the nonstationary workload, and the requirement to limit exploration to unacceptable SLA violations during the learning process.

In general, the learning-based scheduling offers a principle approach to the adjustment of orchestration choices to variable and heterogeneous conditions. Contextual bandits tend to be a more logical choice where decisions are made on a request basis and immediate feedback is required, whereas full RL can better be applied where the action space is more discrete and there is more temporal correlation and delayed reward. These considerations motivate the adoption of online learning methods that are compatible with low-overhead orchestration and can be evaluated against static and heuristic baselines under realistic load variation.

2.6 Summary and Research Gap

The literature reviewed in Sections 2.1-2.5 provides a good foundation for computation offloading, split execution and SLA-aware scheduling. Simultaneously, it illustrates

valuable gaps between the assumptions applied in previous work and the relaxed needs in real-world transformer-based generative inference in the heterogeneous edge-fog setting. The literature on Mobile Edge Computing (MEC) provides valuable architectural parameters and offloading taxonomies, such as binary, full or partial offloading, and examines the optimization-based resource allocation in the context of the wireless and computational constraint conditions [5, 6, 7, 8]. Nevertheless, a large number of MEC formulae divides applications into a divisible workload and fail to explicitly model the deep dependency chains or the cost of transmitting intermediate representations which emerge in split inference pipelines.

Research about split inference solves some of these problems by focusing on partition-point selection and compute-communication tradeoffs [9, 10, 11, 12]. A significant portion of this work is, however, limited to CNN based inference and as such is not exhaustive when it comes to sequential, token-dominated execution pattern prevalent in inference of large language models (LLM) in particular under concurrent workloads.

The above literature on the importance of keeping the large language model (LLM) inference systems continues to demonstrate that these mechanisms, namely prefill/decode decomposition, dynamic batching, and management of KV-cache memory, are highly important to the practical implementation of the system [13, 14, 15, 16]. These systems are mostly configured to work in environments of datacenters and have a powerful accelerator, and tend to have most of the throughput under the limits of memory availability in the graphics card. In comparison, environments with edge-fogs have significantly higher memory constraints, changing network conditions, and sensitivity to tail latency and queueing delay.

Finally, both old and new scheduling studies make clear that the compliance of SLA requires the control of tail latency and queueing effects [17, 18, 19, 20]. Yet, in split inference orchestration, it is the interaction between several coupled stages such as computation, communication and queueing that generates latency. This multi-stage dependency is not typically modelled explicitly in both the traditional MEC offloading research and the existing LLM serving literature.

To make the synthesis concrete, Table 2.1 summarizes representative works by research area, their main contributions, and the specific limitations that motivate this thesis.

Table 2.1: Summary of reviewed literature and the resulting research gap.

Area	Reference	What it contributes	Limitation / gap for this thesis
MEC & Offloading	Mao et al. [5]	MEC overview; communication-centric view; key challenges in offloading and resource allocation.	Generic task abstractions; limited treatment of sequential DNN/LLM pipelines and tail-latency under concurrency.
MEC & Offloading	Mach & Běcvář [6]	Architecture survey; taxonomy of offloading (local/binary/partial); orchestration considerations.	Partial offloading often modeled as divisible computation; does not capture layer dependency and activation transfer cost in split inference.
MEC & Offloading	You et al. [7]	Optimization-based energy/latency-aware resource allocation for MECO.	Relies on simplified/centralized models; not partition-aware for DNN layers; limited handling of non-stationary queueing and heterogeneity.
MEC & Offloading	Chen et al. [8]	Multi-user offloading under shared resources; distributed decision-making perspective.	Targets independent tasks; does not model split pipelines with intermediate tensor transfer and sequential constraints.

Area	Reference	What it contributes	Limitation / gap for this thesis
Split inference	Kang et al. (Neurosurgeon) [9]	Dynamic DNN partitioning; makes compute-communication trade-offs explicit via boundary activations.	Mostly two-node splitting and CNN-like workloads; limited focus on queueing/tail latency and generative decode behavior.
Split inference	Li et al. (Edgent) [10]	Device-edge co-inference with early-exit ideas for latency control.	Early-exit classification focus; not aligned with transformer autoregressive decode and KV-cache/memory constraints.
Split inference	Teerapittayanon et al. (BranchyNet) [11]	Early-exit architectures for faster inference under latency budgets.	Requires model changes and accuracy trade-offs; not directly applicable to standard LLM serving pipelines.
Split inference	Zhao et al. (DeepThings) [12]	Distributed inference on resource-constrained edge clusters; adaptation under heterogeneity.	Primarily CNN workloads; limited SLA/tail-latency analysis; does not address transformer decode or KV-cache effects.
LLM serving	Yu et al. (Orca) [13]	Serving design for generative transformers; iteration-level scheduling for throughput/latency trade-offs.	Datacenter assumptions; not designed for heterogeneous edge-fog constraints and cross-node split inference.

Area	Reference	What it contributes	Limitation / gap for this thesis
LLM serving	Kwon et al. (Page-dAttention / vLLM) [14]	KV-cache memory management using paging; improves concurrency by reducing fragmentation.	Primarily single-node GPU serving; does not address partition placement across multiple devices.
LLM serving	He et al. (Deferred batching) [15]	Continuous/dynamic batching refinements for resource-efficient serving.	Server-class focus; does not model distributed split stages and network overhead in MEC.
LLM serving	Aminabadi et al. (DeepSpeed-Inference) [16]	System-level optimizations and heterogeneous memory approaches for transformer inference.	HPC/datacenter focus; not tailored to resource-limited edge-fog tiers and per-request split selection.
Queueing/SLA	Dean & Barroso [17]	Tail-latency principles for large-scale interactive services (p95/p99 focus).	High-level principles; gap is operationalizing tail-latency control for multi-stage split inference on edge-fog nodes.
Queueing/SLA	Liu & Layland [18]	Foundational deadline scheduling and feasibility analysis in real-time systems.	Classic CPU scheduling model; does not include network transfers, multi-stage pipelines, or heterogeneous device constraints.
Queueing/SLA	Verma et al. (Borg) [19]	Cluster scheduling mechanisms: priorities, quotas, policy enforcement in shared clusters.	Datacenter-scale; not focused on low-overhead per-request scheduling and stage coupling in split inference.

Area	Reference	What it contributes	Limitation / gap for this thesis
Queueing/SLA	Ousterhout et al. (Sparrow) [20]	Low-latency distributed scheduling using sampling and late binding.	Does not directly address sequential dependency chains with coupled compute+network stages typical in split inference.
Learning-based	Auer et al. (UCB) [21]	Bandit framework with exploration–exploitation guarantees.	Abstract model; requires systems-specific context design and reward shaping under SLA constraints.
Learning-based	Li et al. (LinUCB) [22]	Contextual bandits for per-decision learning from context features with immediate feedback.	Not split-inference-specific; gap is designing robust features/rewards for partition feasibility and safe adaptation under SLA.
Learning-based	Mnih et al. (DQN) [23]	Deep RL for sequential decision-making with function approximation.	Training stability and sample efficiency challenges; exploration can violate SLA unless constrained.
Learning-based	Mao et al. (Deep RL resource mgmt) [24]	Motivates learning-based control for complex resource management problems.	Often datacenter-style settings; gap is adapting learning-based control to edge-fog split inference with explicit network overhead.

Learning-based scheduling offers a natural mechanism for taking into account the non-stationarity and heterogeneity by adjusting the decision from the runtime feedback [21, 22, 23, 24]. However, one of the main missing elements throughout the reviewed work is that approaches based on learning are not often combined with

a partition-aware model of split inference that explicitly models (i) feasibility constraints imposed by sequential dependencies, and (ii) boundary tensor transfer costs, and (iii) queueing-driven SLA behavior under concurrency. In other words, offloading, partitioning, queueing control and online learning have been treated by existing literature as separate threads, while a practical edge-fog deployment of generative inference requires these different aspects be coordinated.

Research gap addressed in this thesis: Motivated by the above limitations, this thesis focuses on orchestrating adaptive split inference for transformer-based generative workloads for heterogeneous edge-fog environments. The emphasis is on choosing split points and placement decisions online with the help of lightweight learning signals and taking into account the device constraints, network variability and SLA driven latency objectives. This gap analysis directly serves as motivation for the methodology and evaluation of the next few chapters.

Chapter 3

Methodology

This chapter outlines the architecture of the proposed framework of adaptive split inference scheduling of transformer-based workloads for executing large language model (LLM) workloads in heterogeneous edge-fog environments. The methodology is aimed at creating a reproducible system which is capable of dynamically dividing the execution of the models among distributed compute nodes considering the heterogeneity of the system, queueing and costs of transfers over a network. The chapter initially establishes the system model along with the operational assumptions such as the cluster topology, communication protocols and workload characteristics. It goes on then to introduce the framework architecture, specifying the control and data planes that get involved in the organization and implementation. The methodology then formalizes split inference and latency decomposition, and then designs the decision agent to be applied in choosing the points of partition and worker assignments. Lastly, the chapter explains the baseline strategies, experimental systems, and measures of evaluation, which were applied to determine the effectiveness of the recommended scheduling system. These elements, when put together, give a holistic account of the system implemented and allows the experiments to be replicated.

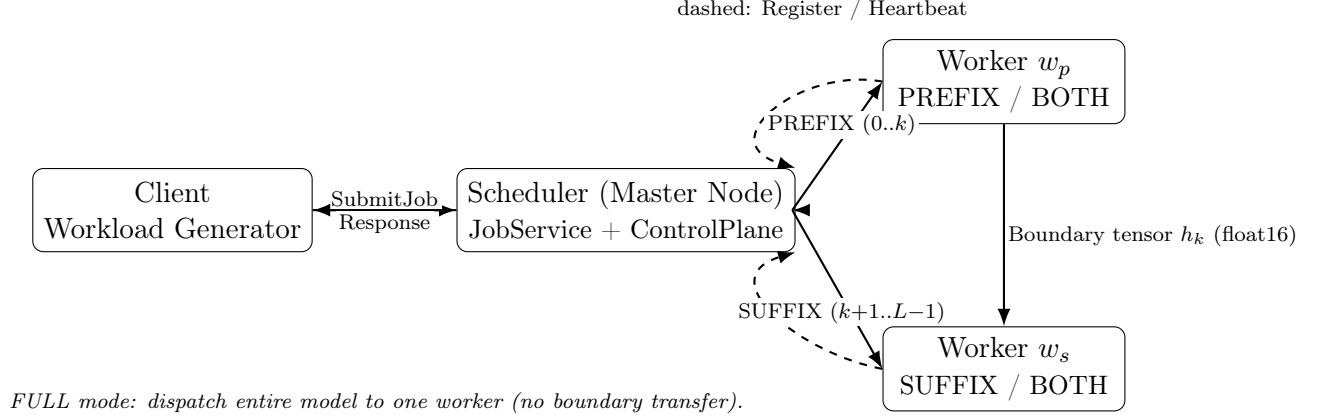


Figure 3-1: High-level system overview. Clients submit inference jobs to the scheduler. Workers register and send telemetry heartbeats (dashed). For split inference, the scheduler dispatches PREFIX to w_p and SUFFIX to w_s , transferring the boundary tensor between workers.

3.1 System Model and Assumptions

The following section refers to the system model and working assumptions that have been applied to the proposed framework. It aims at formalizing the computing environment the adaptive split inference scheduling works in. In particular, the system model determines the structure of the cluster, communication scheme, the compute characteristics of heterogeneous devices, as well as the nature of the inference load. These assumptions underlie the choices of scheduling and partitioning of the decisions set in the sections further.

The proposed framework targets heterogeneous edge-fog settings, where inference workloads can be implemented on devices that have varied computational resources and resource limitations. The system model thus presupposes a distributed cluster where transformer model inference may be performed wholly on a solitary device or in a split way between several devices (by split inference).

Figure 3-1 illustrates the high-level architecture of the proposed system and the interaction between clients, the scheduler, and worker nodes.

3.1.1 Cluster Topology and Roles (Master/Scheduler, Workers)

The model works on a system of interconnected machines which are structured based on a master-worker architecture. The cluster includes one master node, which carries out the scheduling logic and a group of worker nodes, which perform model inference activities. There is a local area network (LAN) that links all the nodes and allows them to communicate with low latency.

The scheduler which is known as the master node is the central control plane of the system. Its duties involve ensuring that it has an global view of the available workers, monitoring their working conditions, accepting inference requests of clients and calculating how a particular request should be handled. Specifically, the scheduler determines whether a request would be run with full inference by a single worker or partitions inference by many workers, and also it does the appropriate point of partition in the model.

The worker nodes make up the implementation layer of the system. Every worker has a containerized inference service that has the potential of executing layers of the transformer models and interacting with the scheduler in terms of remote procedure calls. The workers report on a periodic basis the current runtime status to the scheduler, with the number of requests waiting in line, the number of running requests, and the resources used. According to these reports, the scheduler calculates the current load of every worker and uses the estimates in their decision-making processes regarding the scheduling.

There are three execution roles in which the workers can work. A prefix worker is a worker that performs the initial layers of the transformer model up to the chosen cut point. The suffix worker implements the final layers with intermediate activations of the prefix stage. Lastly, a worker which is set to run full execution processes the complete model without partition. This elastic composition of roles enables the system to have dynamic adaptability to the resource availability and load features.

Figure 3-2 presents the physical deployment of the experimental cluster used in

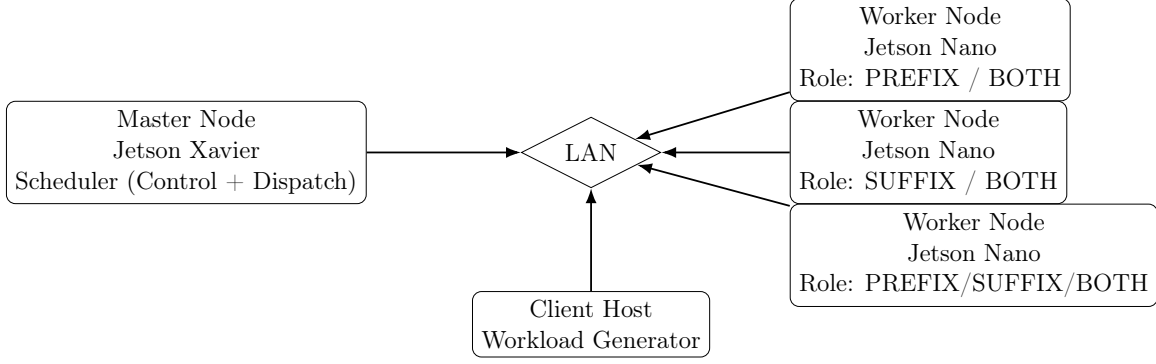


Figure 3-2: Physical cluster topology used in the experiments. The scheduler runs on the Jetson Xavier master node. Worker services run on Jetson Nano devices and can be configured for PREFIX, SUFFIX, or BOTH roles. The client host submits concurrent requests over the LAN.

this work. The cluster includes heterogeneous edge devices such as Jetson Nano and Jetson Xavier platforms, which represent realistic edge computing hardware with different compute capabilities.

This topology enables the system to emulate practical edge-fog inference scenarios, where lightweight edge devices cooperate with more powerful nodes to execute deep learning workloads efficiently.

3.1.2 Communication Model (gRPC, Message Formats, Boundary Tensor)

The architecture involves *gRPC over TCP* as the main communication substrate between all of its components: (i) the communication between a client and a scheduler job submission, (ii) the communication between workers and a scheduler control-plane coordination, and (iii) the communication between workers and workers the transfer of intermediate activations during split inference. *Protocol Buffers* are defined to define remote procedures and message schemas, which offer strongly typed interfaces and compact binary serialization to fit low-latency orchestration on a cluster of computers connected on a LAN.

At the application level, the communication model consists of two logical planes: *control plane* and *job submission plane*.

Figure 3-3: gRPC interfaces and message flow: client submits jobs to the scheduler (JobService); workers register and send heartbeats (ControlPlane); split inference transfers the boundary tensor from prefix worker to suffix worker.

Table 3.1: Key message fields used in the gRPC communication model.

Message	Fields (subset)
JobRequest	request_id, model_id, split flag, candidate_cuts, input tensor, deadline_ms
Heartbeat	inflight requests, queue length, memory usage, cache size

Control plane (Worker $\rightarrow$ Scheduler). At start up, each worker enrolls its name to the scheduler, and occasionally sends heartbeat messages to the scheduler. Heartbeats are used to (a) The liveness check and (b) lightweight telemetry reporting that the scheduler uses to monitor load in runtime. Heartbeat payloads used in the present implementation cover the amount of outbound requests, the size of the local queue, memory usage and size of a cache. The scheduler keeps a time-to-live (TTL) window, which is configurable, and only takes into account a worker as *alive* when the most recent heartbeat of the worker falls within the time-to-live window; the rest of the workers are temporarily ignored by the scheduler.

Job submission plane (Client $\rightarrow$ Scheduler). The clients otherwise make inference requests to the scheduler through the JobService. Every JobRequest has a request_id that it has been end-to-end tracing with, model_id, a split flag which is a Boolean indicating whether it can split its end-to-end or not, a list of candidate_cuts, the input tensor in its serial form, and latency constraint deadline_ms. The scheduler responds with a job response which gives response in form of completion status and metadata of timing used by the client in the computation of end-to-end latency and success rate.

Boundary tensor transfer (Prefix $\rightarrow$ Suffix). Split inference involves an open data-plane between workers. The system calculates a middle-level of activation (non-observable state) k-dimensional array at the cut point on the prefix worker by applying layers 0..*k* on the prefix worker before the count of layers. This boundary tensor too is serialised and transmitted to the suffix worker where further processing

Figure 3-4: Split inference sequence: client submits a job; the scheduler selects workers and a cut; the prefix worker executes the prefix stage and sends the boundary tensor; the suffix worker completes the suffix stage and returns the result.

with layers $k+1..L-1$ is reinitiated. Tensors can be defined as a container of type `Tensor`, `typ`, `shape`, `raw` of the form: `Tensor{dtype, shape, raw_bytes}`, `dtype` is an enumeration of the numerical type, `shape` is an arrangement of the dimensions of the tensor, and `raw_bytes` contains the contiguous binary buffer. To reduce the bandwidth to be spent on transferring the boundaries, the in between activations are transferred as `float16`. The size of transferring the boundaries (bytes) is entered in the system and it is viewed as an explicit term of overhead in the evaluation.

Finally, the framework identifies `request_id` as the main correlation key between logs and metrics which allows coherent mapping of the scheduler decision, worker-side stage timings (prefix/suffix), edge-to-edge transfer size and client-observed end-to-end results.

3.1.3 Compute Model (Layer Costs, Device Heterogeneity)

Execution plane is composed of heterogeneous work nodes which vary in the compute throughput, memory capacity and load at runtime. The framework is aimed at decoder-only transformer inference, in which computation is run sequentially via a stack of transformer blocks. In a model having L transformer layers (blocks), the forward pass may be broken down into per-layer execution costs that are both hardware platform and current contention dependent on the worker.

Layer-level cost model. We model the compute time of layer $\ell \in \{0, \dots, L-1\}$ on worker w as a non-negative random variable with expectation

$$\mathbb{E}[t_\ell^{(w)}] = c_\ell^{(w)}(n), \quad (3.1)$$

where n denotes request-specific input characteristics that affect compute (most notably prompt length / sequence length). In practice, the system does not require an explicit analytical formula for $c_\ell^{(w)}(n)$; instead, it relies on *observed execution mea-*

measurements collected online. Each of the completed stages provides empirical timings (prefix and suffix execution time) which record implicitly the cumulative cost of the layer range involved in the given worker conditions.

Device heterogeneity. Heterogeneity is represented by worker-dependent cost profiles and capacity constraints. Workers may differ along the following dimensions:

- **Compute throughput:** devices exhibit different per-layer runtimes due to CPU/GPU availability and microarchitecture differences.
- **Memory capacity and pressure:** available memory influences feasibility of hosting model weights and intermediate tensors; runtime memory usage is tracked and reported to the scheduler.
- **Queueing and contention:** concurrent requests increase service time due to resource sharing; workers report inflight requests and queue length via heartbeats.

These heterogeneity signals are exposed to the scheduler through heartbeats and are treated as contextual inputs to scheduling decisions.

Stage cost under split inference. For a cut position k , the prefix stage executes layers $0..k$ and the suffix stage executes layers $k+1..L-1$. The compute time of the prefix stage on worker w_p and suffix stage on worker w_s can be expressed as

$$t_{\text{pref}}(k, w_p, n) = \sum_{\ell=0}^k t_{\ell}^{(w_p)}(n), \quad (3.2)$$

$$t_{\text{suf}}(k, w_s, n) = \sum_{\ell=k+1}^{L-1} t_{\ell}^{(w_s)}(n). \quad (3.3)$$

In the implementation, these quantities are measured directly as the metrics `prefix_execution_time` and `suffix_execution_time` recorded for each request, rather than being estimated from offline per-layer profiling.

Assumptions on cost estimation. The compute model in this work is *measurement-driven*:

- No fixed offline profiling table is assumed as a prerequisite for scheduling.
- The scheduler observes runtime telemetry (queue length, inflight requests, memory usage) and per-request stage timings reported during execution.
- The adaptive policy uses these measurements as context to select workers and cut positions online.

In this regard, the compute model represents both hardware heterogeneity and time-varying contention without any tight parametric constraints on layer runtimes. This design fits practical edge-fog configurations, in which the service times are non-stationary, and the per-layer cost analysis of the devices is often not reliable.

3.1.4 Workload Model (Arrival Process, Prompt Lengths, Deadlines)

The workload is a series of requests on inference to the scheduler using the `JobService` by a workload generator on the client side. Every request is modeled as a `request_id` target `model_id` split `enable-flag-candidate_cuts` positions serialized input tensor per request latency `deadline_ms` `JobRequest` messages. The granularity of work in this work then is determined as individual requests and the structure of their concurrency.

Arrival process and concurrency. A Python workload script generates requests and is able to submit jobs in a sequential or parallel fashion. In the experimental setup, the supplied load is regulated in an application of two variables: (i) the overall total number of requests and (ii) the fixed concurrency level where the number of requests under flight is regulated. The resultant arrival dynamics is reminiscent of busy interactive inference in which several prompts may be pending concurrently. Although a Poisson arrival process is typical in queueing analysis, operationalizations of load here are based on *concurrency control* residing at the client and the observation of the induced queueing is based on worker telemetry (inflight requests and queue length).

Prompt length model. The client script is used to generate input prompts

Figure 3-5: Framework architecture overview. The control plane manages registration, heartbeats, and liveness; the data plane executes FULL/PREFIX/SUFFIX stages and transfers the boundary tensor in split inference.

in the form of token sequences. Prompt length is discussed as a workload attribute of primacy since it impacts the cost of compute service, and, in split execution, the scale of intermediate activations in the middle. The prompt length is provided in the context features submitted to adaptive scheduling policy, on each request. To test sensitivity to input size, experiments are also varied over prompts length as a configurable parameter.

Deadlines and SLA constraints. The request is subject to a latency budget as indicated by `deadline_ms` that is an application level service-level objective (SLO) on end to end completion time. The SLA *miss* is when a request had an end-to-end latency that was larger than `deadline_ms`. The scheduler takes the deadline information as part of its decision context, and the learning objective solutions penalize a requirement to meet its deadlines. The evaluation methodology also uses tail behavior as reported by percentiles based on miscellany of the latency (e.g. p50/ p95/ p99) to describe the distributional attributes of completion time under confluency and heterogeneity. The workload model hence incorporates (i) *per-request* restrictions through explicit deadlines as well as (ii) *system-level* quality of service through tail latency statistics.

3.2 Framework Architecture

The section outlines the framework architecture as seen through systems perspective wherein the design is fragmented into a *control plane*, and a *data plane*, the control plane takes care of cluster membership, liveness, and gathering runtime telemetry utilized to make scheduling decisions. Data plane also provides inference requests as a one-stage FULL run or as two-stage split run (PREFIX and SUFFIX) and gives the result back to the client.

Figure 3-6: Control-plane overview. Workers register once at startup and then send periodic heartbeats carrying liveness and telemetry. The scheduler applies a TTL policy to maintain the set of eligible workers.

3.2.1 Control Plane (Registration, Heartbeats, Liveness/TTL)

The scheduler, worker-side routines implement the control plane which keeps cluster membership and also provides scheduling runtime telemetry. Its key goals are (i) the consistency of the perception of the available workers, (ii) the ability to monitor liveness of workers in the event of failures and random connectivity problems, and (iii) it provides lightweight load signals (e.g., queueing and utilization proxies), that are consumed by the scheduling policy.

Registration. Once each of the workers is started, it registers with the scheduler using the `ControlPlane` gRPC. Registration associates a worker identifier with the endpoint reachable by it and sets up scheduler-side state to be used in future liveness tracking and telemetry aggregation. Following a successful registration, a worker will be allowed to get inference stage dispatches, provided that they meet the liveness policy as discussed below.

Heartbeats and telemetry. Each worker also periodically transmits messages to the scheduler containing heartbeat messages. Heartbeats are essential as they function as liveness signal and as a report on the runtime telemetry. These measurements can be used by the scheduler to ensure a view of worker load and resource pressure that involves coarse graining. Heartbeat payloads in the present implementation consist of the number of inflight requests, the number of current queue, memory, and size of the cache.

Liveness and TTL policy. The scheduler keeps a time to live (TTL) window with regards to each worker. A worker is said to be alive when its last heartbeat time falls within the TTL limit or is said to be unavailable and will be dropped out of the decision making worker eligibility set. The workers that lost their heartbeats during a short-term failure are automatically re-admitted when their heartbeat recency does become smaller within the TTL window.

Table 3.2: Worker telemetry reported to the scheduler via heartbeats (subset).

Field	Description
Inflight requests	Number of requests currently executing on the worker
Queue length	Number of requests waiting in the worker’s local queue
Memory usage	Current memory utilization reported by the worker
Cache size	Size of resident cached artifacts (e.g., loaded model weights)

In general, the control plane offers online, metrics-based information on cluster availability and load to the scheduler, which is then further used by the logic in the subsequent parts to define the split point and assign workers.

3.2.2 Data Plane (FULL, PREFIX, SUFFIX Execution)

Inference on scheduler dispatched worker nodes occurs on the data plane. The mode at which each request is implemented is **FULL**, **PREFIX**, or **SUFFIX**. The mode value chosen is what defines whether execution is done either on one worker, or it is divided into two workers under an explicit boundary-tensor transfer. **FULL execution.** When full mode is used, the scheduler assigns the request to a single worker that executes the entire model forward pass on the single machine and provides the final output as well as metadata of when the execution completed. **Split execution (PREFIX/SUFFIX).** In split inference, when switched on, the scheduler picks one cut position, denoted by k , and carries out two stages of dispatch. Workers, denoted by the prefix worker, execute layers $0..k$; the hidden-state tensor on the cut boundary is an intermediate. This boundary tensor is encoded in the system tensor container (`dtype`, `shape`, `raw_bytes`) and sent to the suffix worker with `float16` in order to minimize the bandwidth. Layers $k+1..L-1$ are then run using the suffix worker which finally returns the resulting output. Both stages are matched by the scheduler with the help of `request_id` and prefix time, suffix time, and transfer size are recorded by the scheduler to be analyzed later.

3.2.3 Deployment Model (Containers, Offline Model Packaging)

The system parts are all executed as containerized services to provide homogeneous runtime environments between heterogeneous machines. Specifically, every worker node will have a container which serves a gRPC server and the model backend, and the scheduler will be a separate service on the master node. Containerization makes dependency management (e.g. Python runtime and ML libraries) simpler, execution is isolated, and offers a reproducible process of deploying the cluster.

The deployment presupposes the *offline* execution environment whereby the worker nodes are not connected to the internet during execution. Model artifacts To help with this constraint, model artifacts are well distributed over the local storage of each worker node before execution. The necessary model weights are loaded by each worker into the memory at startup and maintained in memory during the lifetime of the worker process, eliminating repeat downloads and reducing the size of per-request load time.

Experimental settings Model packaging and distribution are modeled as being the same: model weights and configuration should be the same model of all workers with `model_id`. Besides this, the system presupposes homogeneous software versions of containers (scheduler, workers) so that the Protocol Buffers message schemas and code in which the model is executed are compatible across an experiment.

3.2.4 Fault Handling (Timeouts, Retries, Idempotency Assumptions)

The framework utilizes liveness via timeouts and TTL in mitigating the effects of failures in the distributed execution. At the control plane, non-emitting heartbeat workers are considered unavailable and excluded in future dispatch decisions. At the data plane, the deadline of a job is known as request-level deadlines (`deadline_ms`) given by the client and apply by way of deadlines over RPC timeouts in the scheduler and worker interfaces.

The given semantics of implementation (**at-most-once**) is that a single `request_id` is submitted and is not re-issued automatically in the case of failure. The system is therefore not reliant on replay-based recovery and does not have any strict requirements of worker execution other than the use of `request_id` to perform correlation and logging. Requests that time out or fail are counted as failures, and integrated into the scheduling goal through penalty terms (e.g. deadline violation, failure): Consistency is consequently restricted to correct per-request accounting in logs and metrics as opposed to precisely once completion guarantees in the event of faults.

3.3 Split Inference Formalization

3.3.1 Cut Definition and Feasibility Constraints

We consider a decoder-only transformer model composed of L sequential transformer blocks (layers) indexed by $\ell \in \{0, \dots, L - 1\}$. A *cut* is defined by an integer k that partitions the model into a prefix segment and a suffix segment:

$$\text{PREFIX}(k) : \{0, \dots, k\}, \quad \text{SUFFIX}(k) : \{k + 1, \dots, L - 1\}. \quad (3.4)$$

The valid cut space is restricted to a discrete candidate set provided in each request,

$$k \in \mathcal{K} \subseteq \{0, \dots, L - 1\}, \quad (3.5)$$

where $\mathcal{K}$ corresponds to `candidate_cuts`. Feasibility is constrained by model dependencies (layers must execute in order) and worker resource limits. This means that the split plan is feasible only if (i) the workers responsible for prefix and suffix are alive under the scheduler policy, (ii) each selected worker is able to host the model weights in memory and execute its assigned task without violating local memory constraints.

3.3.2 Boundary Tensor Definition and Serialization

For a request input x , executing the prefix segment up to cut k produces an intermediate activation (hidden state) at the boundary:

$$h_k = f_{0:k}(x), \quad (3.6)$$

where $f_{0:k}(\cdot)$ denotes the composition of layers 0 through k . The boundary tensor h_k is transmitted from the prefix worker to the suffix worker and used as input to the remaining layers:

$$y = f_{k+1:L-1}(h_k). \quad (3.7)$$

In the implementation, h_k is serialized using a generic tensor container `Tensor{dtype, shape, raw_bytes}` and transmitted via gRPC. Intermediate activations are encoded as `float16` to reduce payload size. The transfer overhead is captured by the payload size in bytes, denoted by $S_k = |h_k|$.

3.3.3 End-to-End Latency Decomposition (Compute + Network + Queue)

For a request executed with cut k on prefix worker w_p and suffix worker w_s , the end-to-end latency is decomposed into queueing, compute, and communication components:

$$T_{\text{e2e}} = Q_p + C_{\text{pref}}(k, w_p) + N(k) + Q_s + C_{\text{suf}}(k, w_s), \quad (3.8)$$

where Q_p and Q_s are queueing delays before prefix and suffix execution, respectively; C_{pref} and C_{suf} are measured prefix and suffix compute times; and $N(k)$ is the boundary transfer time. The network term is modeled as a function of payload size:

$$N(k) \approx \alpha + \frac{S_k}{B}, \quad (3.9)$$

with fixed per-transfer overhead α (RPC and serialization overhead) and effective bandwidth B on the LAN. For `FULL` execution on a single worker w , the decomposition

reduces to

$$T_{e2e}^{\text{FULL}} = Q + C_{\text{full}}(w), \quad (3.10)$$

since no boundary tensor transfer is performed. In all cases, a request is considered an SLA miss if $T_{e2e} > \text{deadline_ms}$.

3.4 Decision Agent for Partitioning

The decision agent that the scheduler applies in choosing (i) the partition point (cut) to perform split inference and (ii) the respective worker assignments to execute prefix and suffix sub-tasks is described in this section. Since worker load, queueing, and effective service times change with time in heterogeneous edge-fog clusters, the dynamic rules of partitioning are not possible. The suggested system hence uses an online learning method that picks up decisions based on feedback. In concrete terms, the scheduler derives each incoming request into a decision problem which is context-dependent, but on a discrete set of candidate cuts and feasible pairs of workers and modifies its policy at the end of each completed request based on the observed results of it in terms of latency, cost of boundary transfer, and success/failure.

3.4.1 State / Feature Representation

A contextual bandit policy enables the scheduler to make a split decision based on the observed feature vector that has summarized the current request and cluster state. To create a context vector $x \in \mathbb{R}^d$ each incoming request r the scheduler uses a combination of (i) request attributes, (ii) worker telemetry in the latest heartbeat and (iii) the policy under consideration. The context is calculated during the decision and has been based on the latest possible measurements of the system.

Request-level features. The request context gives the prompt length (number of tokens), latency requirement `deadline_ms` and the target model identifier `model_id` (treated as an approximation of model size/configuration in the case of multiple models).

Table 3.3: Context features used by the scheduling policy (subset).

Group	Features
Request	prompt length, <code>deadline_ms</code> , <code>model_id</code>
Prefix worker (heartbeat)	queue length, inflight requests, memory usage, cache size
Suffix worker (heartbeat)	queue length, inflight requests, memory usage, cache size
Cut	candidate cut index k (from <code>candidate_cuts</code>)

Worker telemetry features. The scheduler uses the most recent heartbeat telemetry data of each candidate prefix and suffix worker with the form of queue length, number of inflight requests, and memory consumption and cache size. These indicators offer lightweight proxies of queueing delay, contention and resource pressure impacting service time and feasibility.

Cut features. The candidate cut position k has been added directly so that the policy may learn cut-dependent trade-offs between prefix and suffix compute balance and boundary transfer overhead.

Normalization and scaling. The intention behind feature preprocessing is to be lightweight to aid online decision making. The values telemetry reports are reported by workers and the policy is taught good relative scaling via the linear model adopted by LinUCB. In case a telemetry field is missing or stale (e.g. because a worker has gone unavailable), the respective worker is filtered out with the TTL-based liveness filter prior to the construction of features, so that the construction of contexts is only done over actions that are feasible.

3.4.2 Action Space (Candidate Cuts, Routing Options)

Each time a request is presented to the scheduler, the scheduler picks an execution plan out of a discrete set of actions that is parameterized by (i) the cut position (ii) the assignment of the worker. To indicate $\mathcal{K}$ is the set of candidate cuts submitted in the request (`candidate_cuts`), $k \in \mathcal{K}$ is a correct partition of the transformer layers into a prefix segment $0..k$ and a suffix segment $k+1..L-1$. $\mathcal{W}_p$ and $\mathcal{W}_s$ represents

the set of live workers currently alive and able to execute prefix and suffix stages respectively according to the TTL policy of liveness and the role assigned to each worker.

Split-inference actions. In split inference, the action is

$$a = (w_p, w_s, k), \quad w_p \in \mathcal{W}_p, \quad w_s \in \mathcal{W}_s, \quad k \in \mathcal{K}, \quad (3.11)$$

The prefix here will be the worker w_p , and the suffix will be the worker w_s and the cut position will be k . The potential action list is narrowed down over the internet to the workers who are not dead, uncontactable, or not programmed to work in the demanded level of the stage. With this filtering the scheduler will only consider the actions that can be run with the present state of the cluster.

FULL actions (no split). When split inference is off (or a no-split baseline is used) then action is to pick one worker and fully execute it:

$$a_{\text{full}} = w, \quad w \in \mathcal{W}_{\text{full}}, \quad (3.12)$$

In this case, $\mathcal{W}_{\text{full}}$ is the group of alive workers who are capable of FULL execution.

On the whole, the routing decision of the scheduler is thus a discrete choice problem subject over a finite collection of feasible worker assignments and candidate cut indices and permits the policy to discover trade-offs among compute balance (prefix versus suffix), queueing conditions within the workers chosen, and overhead of boundary transfer of the selected cut.

3.4.3 Reward / Objective (Latency, Misses, Boundary Cost)

Once a scheduling action a has been performed by a request r the scheduler calculates a scalar reward which can be used to update the contextual bandit policy. These are aimed at reducing both end-to-end latency and encouraging both SLA violations and prohibitively costly boundary transfers in split inference. Take $T_{\text{e2e}}(r)$ to refer to the duration of request r measured end-to-end and $D(r)$ to refer to its deadline constraint

(`deadline_ms`). Assign $S(r)$ a measured payload size of the boundary tensor in the number of bytes $S(r) = 0$ example: `FULL` execution.

We define the per-request cost as

$$J(r) = T_{e2e}(r) + \lambda \mathbb{I}[T_{e2e}(r) > D(r)] + \mu S(r) + \nu \mathbb{I}[\text{fail}(r)], \quad (3.13)$$

whose indicator function is $\mathbb{I}[\cdot]$, the execution was time gone dead or a stage failed, $\text{fail}(r)$ represents the execution failed, and $\lambda, \mu, \nu \geq 0$ are weighting factors. The initial is the penalty of checking the raw latency, the two more suffixes penalty of an excessive deadline, the third encapsulates overheads of transferring boundaries, and the fourth is an extra penalty of an unsuccessful request.

This negative cost is then defined as reward:

$$R(r) = -J(r), \quad (3.14)$$

such that maximization of expected reward translates to minimization of latency, SLA misses, and boundary-transfer overhead, SLA misses, and, to the same extent, maximization of expected rewards are missing. The cost function (3.13) takes all quantities determined through runtime measurements of the scheduler and the workers, such as stage timings, size of the request boundary tensor, and request outcome.

3.4.4 Learning Algorithm (e.g., LinUCB / Bandit / RL)

Adaptive policy is followed as a context bandit which uses LinUCB algorithm. To each request, the scheduler lists the possible action set it includes (due to the candidate cuts) and also includes (due to alive worker assignments) and assigns a context vector to the request of request attributes (e.g., prompt length, `deadline_ms`, `model_id`), worker telemetry (e.g., queue length, inflight requests, memory usage, cache size), and the cut index of the candidate cuts. LinUCB represents the projected reward as a linear form of the context, and choose the action to maximize an upper confidence bound, which trades-off between exploitation of high-reward actions and exploration

Algorithm 1: Online LinUCB for adaptive split scheduling

Input: Exploration α , regularization λ
 $A \leftarrow \lambda I_d$
 $b \leftarrow \mathbf{0}_d$
foreach request r **do**
 $\mathcal{A}(r) \leftarrow$ feasible actions (alive workers, roles, `candidate_cuts`)
 $\hat{\theta} \leftarrow A^{-1}b$
 foreach $a \in \mathcal{A}(r)$ **do**
 $x_a \leftarrow \text{FEATURES}(r, \text{telemetry}, a)$
 $u_a \leftarrow \sqrt{x_a^\top A^{-1} x_a}$
 $p(a) \leftarrow \hat{\theta}^\top x_a + \alpha u_a$
 $a^* \leftarrow \arg \max_{a \in \mathcal{A}(r)} p(a)$
 DISPATCH(a^*) // FULL or PREFIX→SUFFIX
 Observe reward $R(r)$
 $A \leftarrow A + x_{a^*} x_{a^*}^\top$
 $b \leftarrow b + R(r) x_{a^*}$
 CHECKPOINT(A, b) periodically

of uncertain actions.

The process of the online decision and update is outlined as Algorithm 1. The algorithm is in form of a regularized design matrix A and weighted-reward per-sample feature accumulator b , with $A = \lambda I_d$ and $b = \mathbf{0}$. One computes the score of each plausible action a with feature vector x_a as an index of confidence based on the current parameter estimate $\hat{\theta} = A^{-1}b$. Having implemented the chosen action the scheduler calculates a scalar reward based on the output of measured variables (end-to-end latency, deadline miss indicator, boundary tensor size, failure indicator) and then performs a rank-one update to (A, b) . Because updates follow each successfully completed request, no distinct offline training period is needed and learning and inference are interwoven: Adaptation is motivated only by runtime measurements with the current workload and state in a cluster. The LinUCB term of exploration scale is regulated by the LinUCB confidence scale that rises in the case of undersampling of regions of the action space and motivates re-assessments of other cuts of different routes.

3.4.5 Persistence and Online Update (agent.pkl, Training vs Evaluation)

The decision agent is informed online with every request made complete. In the case of the chosen action, the scheduler calculates a scalar reward using the measured results that include end-to-end latency, deadline miss indicator, the split execution of the data bound in the form of a boundary tensor, and failure status. It then makes the corresponding LinUCB update to the agent state variables (A, b) . This causes an *inference-and-learning* workflow to inter-learn itself such that the scheduling decisions are constantly updated with the current workload and cluster state without the need to undergo a separate offline training process.

To achieve continuity between process restarts and in order to facilitate reproducible experiments, the scheduler periodically sets the state of agents to disk (i.e. in a file such as `agent.pkl`). The checkpoint holds the parameters which are required to resume the learning process including the existing matrices (A, b) and the metadata of feature dimension and hyperparameters. At the time of the scheduler initiating, one of the following tasks may be executed: either the scheduler loads an existing checkpoint to *warm-start* the policy based on previous experience, or specifies an initial agent state used on a *cold-start* run. Reset and warm-start are considered a part of configuration of the experiment.

Moreover, the framework allows an evaluation-only mode where the scheduler loads an agent checkpoint, which was persistent but does not update online any further. This lets the actions of a fixed policy be quantified when running evaluation and ensures that results obtained in experiments do not change due to the manifestation of long-term learning during the measure phase.

3.5 Baselines

In order to put into perspective the performance of the suggested adaptive split scheduling policy, we compare it with a set of non-proposed base strategies which

apply the same execution model (scheduler, workers, gRPC data plane) but vary in the way they select the mode of execution, cut point, and routing. Each of the baselines shares all of the TTL-alive workers and share the same request interface, such as `candidate_cuts` and `deadline_ms`. In the absence of specification, there are no updates to online learning in the baselines.

3.5.1 No-Split (FULL on One Worker)

The No-Split baseline turns partitioning off, and runs each request purely on one worker in FULL mode. For every incoming request, the scheduler picks one TTL-alive worker to support full execution of the job and dispatches the job with no need for generating and transferring boundary tensors. This baseline separates out the overheads and advantages of split execution by having a reference point where there is no network transfer and no cross-worker coordination.

3.5.2 Static Cut (k)

The Static Cut baseline allows split inference but determines the partition point to be that of a fixed cut index k across requests. Taking a request and its `candidate_cuts` the scheduler chooses either (i) predefined $k \in \mathcal{K}$ when it is accessible, or (ii) the nearest viable cut within a candidate set. The scheduler then in turn fails at two stage dispatch with the chosen cut: the prefix worker performs layers $0..k$ and the suffix worker performs layers $k+1..L-1$. This baseline determines the effects of implementing one non-adaptive partition point in different workload conditions.

3.5.3 Heuristic Queue-Aware Cut

The Heuristic Queue-Aware Cut is a baseline that calculates the cut position based on a simple rule which is dependent on the current load of the workers represented as heartbeats. Given a feasible cut $k \in \mathcal{K}$ and each worker pair (w_p, w_s) that the scheduler considers, the scheduler examines proxies of queue-analysis including the lengths of prefix and suffix queue and the numbers of outstanding requests and selects

a cut that skews the allocation of workload to the less loaded worker. The heuristic, in particular, picks a cut that reduces the estimated contention on the congested worker by assigning it a smaller fraction of the layers, but does not violate the constraints of the set of cuts that a request can be assigned. This baseline represents a minimalistic, responsive policy that takes into consideration the cues related to queues but it does not take into account the insights provided by the long run performance information.

3.5.4 Greedy Predicted Latency (Optional)

The Greedy Predicted Latency baseline picks the action that minimizes a naive end-point latency target which is calculated also at decision time. The scheduler generates a latency estimate of each candidate action (w_p, w_s, k) , including: (i) a queueing proxy which is an estimate of the current queue length and the outstanding requests, (ii) an estimated prefix and suffix compute time, and (iii) an approximate cost due to transferring boundaries based on the size of the tensor or a cut-dependent proxy. The scheduler is then used to pick the action whose cumulative prediction is minimized, thus applying an explicit optimization-based heuristic, which is based on instantaneous predictions and does not update parameters based on online learning.

3.6 Experimental Setup and Metrics

This section describes the experimental methodology used to evaluate the proposed adaptive split inference framework. The goal is to provide a reproducible evaluation that isolates the impact of (i) split execution and boundary-tensor transfer, and (ii) adaptive cut and routing decisions under heterogeneous worker conditions. We first summarize the hardware and software stack used for the scheduler, workers, and client workload generator, followed by the models evaluated and the offline deployment procedure. We then define the workload generation parameters and the metrics collected during execution, including end-to-end latency, stage-level timings (prefix/suffix), boundary transfer size, throughput, and success/failure outcomes. All experiments log per-request measurements using a unique `request_id` to enable con-

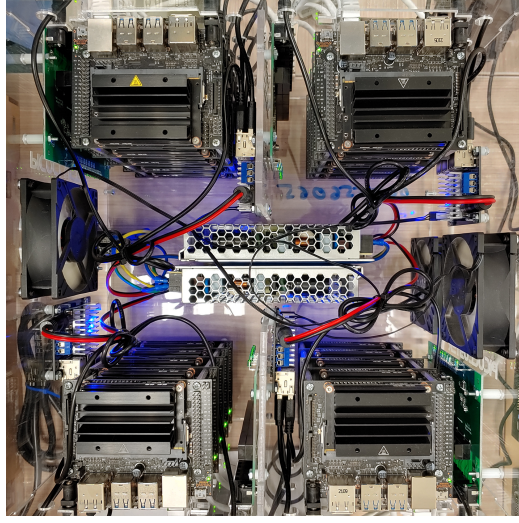


Figure 3-7: Experimental edge-fog testbed used in this work. The scheduler (master) runs on a Jetson Xavier device, while workers run on Jetson Nano devices. All services communicate over a LAN and execute inference on CPUs only.

sistent post-hoc analysis and comparison across baselines.

3.6.1 Hardware and Software Stack (Master/Worker Machines)

The experimental testbed consists of a small fog-based LAN connected edge-based cluster; it includes a dedicated **master** node and a **workers** node or nodes. The scheduler service is operated on the master node which is built on an NVIDIA Jetson Xavier platform and the worker nodes are deployed inference execution services on NVIDIA Jetson Nano devices. Inference in the existing structure is only carried out on CPU resources avoiding the use of GPU acceleration; the design decision in the architecture exaggerates the scheduling effects given limited computational resource availability and makes performance subject to contention and queueing behavior. All the nodes run **Ubuntu Linux**.

3.6.2 Models Used

This structure supports decoder-only, GPT-style transformer models and currently evaluates split execution is used with `distilgpt2` as the main model. `Distilgpt2` is a small transformer with a small number of consecutive blocks of transformers in

series, making it a convenient device to use in controlled experimentation of resource-constrained edge devices, and with the architectural characteristics expected of split inference, including sequential layer dependencies and activation of a hidden state. The model is implemented in an autoregressive inference scenario and the split point k is the boundary between layers: $0..k$, etc. (prefix) and $k+1..L-1$ (suffix). In any given experiment, the model identifier (`model_id`) is recorded with the request so that the data which is obtained could then be linked to the appropriate configuration.

3.6.3 Offline Mode and Model Distribution Method

All the experiments are conducted in an **offline mode** such that worker nodes do not require internet connectivity to run. As a result, model artifacts are sent over to individual nodes before execution and stored on disk locally. As a worker is starting, the model (by attribute name; the one of interest being the `model_id`) is loaded into memory and kept in residency until the worker process terminates, eliminating the need for repeated downloads and minimizing the time per-request overhead of initializing a model. Implementation assumes that all workers have model weights and tokenizer configurations that are equal to the original `model_id`. The management of artifacts is also added as an inseparable part of the experimental configuration: the name and version of the model, both the version of the container image are recorded in each run to make sure that the version of the code and model weights run with are the same across the nodes.

3.6.4 Metrics (p50/p95/p99, Throughput, Miss Ratio, Failures)

The assessment also focuses on latency, reliability and time scheduling throughout the work during concurrent loads. The system logs the **end-to-end latency** per request i.e., the time between the submission of the request by the client and its completion. The latency statistics are aggregated into percentile statistics that include the 50 th percentile, the 95 th percentile and the 99 th percentile which have median and tail

performance respectively. Throughput is expressed in the number of requests that are completed successfully after a specific period of time of time measurement. **Service level Agreement (SLA) miss ratio** is the ratio of the requests whose end to end latency exceeds the required request deadline:

$$\text{MissRatio} = \frac{1}{N} \sum_{i=1}^N \mathbb{I}[T_{\text{e2e}}(i) > D(i)], \quad (3.15)$$

where $T_{\text{e2e}}(i)$ is the measured latency and $D(i)$ is `deadline_ms` for request i . The system also reports **failure rate**, defined as the fraction of requests that do not complete successfully (e.g., due to timeout or execution failure):

$$\text{FailRate} = \frac{1}{N} \sum_{i=1}^N \mathbb{I}[\text{fail}(i)]. \quad (3.16)$$

The additional diagnostic measures in a split inference execution include the temporal measures of the prefix segment and suffix segment, volume of transferred data between boundary tensors and the empirically found distribution of the selected cut indices during this processing.

The current evaluation does not include direct power or energy measurements from the Jetson devices. Therefore, energy-related implications are discussed only indirectly through compute time, communication overhead, and resource utilization behavior.

3.6.5 Logging and Reproducibility (CSV/JSONL, Seeds, Configs)

The framework logs per-request measurements allowing offline analysis and making comparisons between and across baseline implementations reproducible. A request is identified by a distinct `request_id` that is transported through the scheduler and workers processes to the identification of client observed results against stage level measures. Fields that are logged include timestamps, end-to-end latency, prefix and

suffix execution time (when relevant), size of tensor at boundary, cut identities of choice, identities of selected workers and completion status. These logs are stored in formatted formats that are friendly to post processing, e.g. CSV or JSONL. Moreover, the experimental run is also supplied with a configuration record which captures workload parameters such as number of requests, concurrency level, prompt length and deadline in addition to split enable or disable parameters and the choice of baseline or policy.

To assist in reproducibility, experiments stabilize random seeds to generate the workload where needed, such as in token sequence generation, and store the values of random seeds in the run configuration. The framework also records the versions of container images used and model identifiers used in each run and the scheduler can either keep the learning agent state (e.g. `agent.pkl`) to allow evaluation-only runs at a fixed checkpoint. This combination of organized cutting, configuration monitoring, and seed management allows the measurements reported with it to be recreated with the same software and the model artifacts.

Chapter 4

Results

The chapter outlines the experimental results based on the actual cluster which is presented in Chapter 3. It was tested using the offline-capable distributed split-inference prototype implemented in the context of this dissertation using `distilgpt2` as the baseline model and making use of the CPUs only across the accessed edge and fog nodes. Each configuration was evaluated on 1,000 (unless indicated to the contrary) requests and repeated three times. In the case of the main fixed-load experiments, policies compared were `full`, `static` (with `STATIC_CUT=2`), `heuristic`, and `agent`. These experiments were set to have a deadline of 12s.

In general, the findings have three major trends. First, the split pipeline was functionally correct and gave outputs which were identical to FULL execution in deterministic, controlled conditions. Second, split execution was found much more beneficial as the concurrency increased whereas the adaptive scheduler offered the best overall performance when the loads were medium and large. Third, the largest advantage of the learning-based scheduler was revealed in dynamic load conditions, during which it managed the choice of cutting conditions better compared to the heuristic baseline and minimized not only tail latency but also deadline misses.

Table 4.1: Functional validation of split inference against full execution.

Configuration	Test cases	Exact-match rate	Deterministic repeatability
FULL vs. SPLIT, $k = 0$	1000	100.0%	100.0%
FULL vs. SPLIT, $k = 1$	1000	100.0%	100.0%
FULL vs. SPLIT, $k = 2$	1000	100.0%	100.0%
FULL vs. SPLIT, $k = 3$	1000	100.0%	100.0%
FULL vs. SPLIT, $k = 4$	1000	100.0%	100.0%
Total	5000	100.0%	100.0%

4.1 System Validation (Functional Correctness, Split Pipeline Correctness)

Before the system performance evaluation, both the system and the pipeline were validated for correctness at both the system and pipeline levels. Validation focused on three major dimensions, namely (i) end-to-end output equivalence between **FULL** execution and **PREFIX+SUFFIX** execution, (ii) correctness across all valid cut positions, and (iii) deterministic behaviour under controlled experiment conditions.

The fixed random seeds, fixed set of prompts, the use of greedy token-generation strategy, and concurrency level of one were used to perform deterministic validation. All of the prompts were first run through a single stage **FULL** request followed by being repeated with split execution at all admissible cut points of the **distilgpt2** model. As the model deployed includes six blocks of transformers, the allowable cut position was $k \in \{0, 1, 2, 3, 4\}$.

Table 4.1 shows that overall the split pipeline replicated the same final token prediction as the full pipeline on every validation trial and that no differences were found with cutting position. In addition, the production cluster continued to be stable under the rectified worker-registration and heartbeat mechanisms and no errors occurred on functional pipeline were experienced after implementation corrections as already mentioned above. This means that these results attest to the fact that the ensuing performance measurements provide genuine scheduling effects, as opposed to those due to implementation bugs.

Table 4.2: Average compute time per cut position for `distilgpt2`.

Cut k	Prefix layers	Suffix layers	Prefix time (ms)	Suffix time (ms)
0	1	5	320	1620
1	2	4	580	1360
2	3	3	860	1090
3	4	2	1170	790
4	5	1	1450	520

Table 4.3: Boundary tensor size for representative prompt lengths.

Prompt length (tokens)	Hidden size	Data type	Boundary size
32	768	float16	48 KB
64	768	float16	96 KB
128	768	float16	192 KB

4.2 Microbenchmarks

4.2.1 Per-Cut Compute Time (Prefix/Suffix)

The first microbenchmark was used to compare the individual timing of prefix and suffix segments with the cut position. Its goal was to define the redistribution on the model workload among the two execution stages.

Table 4.2 shows that prefix execution time increases as the cut position changes to later ones, but the suffix execution time decreases respectively. This tendency is expected to happen, with later cuts assign more transformer layers to the prefix worker. Furthermore, the aggregate computational budget does not change substantially with varying cut positions; therefore, the distribution of computation mainly depends on the choice of cut and not on the amount of computation needed.

4.2.2 Boundary Size vs Cut

The second microbenchmark looked into the size of the boundary tensor that was sent from the prefix worker to the suffix worker. The boundary tensor size is mostly dependent on sequence length and not the cut position, because the hidden dimension remains constant between layers.

Table 4.4: RPC and serialization overhead on the real cluster.

Case	Payload	p50 (ms)	p95 (ms)
Empty RPC	negligible	3.2	4.8
Split RPC + 32-token boundary	48 KB	5.1	7.4
Split RPC + 128-token boundary	192 KB	9.4	13.1

The boundary size as demonstrated in Table 4.3 increases in a linear manner with length of a prompt and is essentially independent of the cut index. This observation holds significance in the interpretation of subsequent results: later cut selection should not be attributed to a simple reduction in boundary size.

4.2.3 RPC Overhead and Serialization Costs

The third microbenchmark was created in order to measure the communication overhead. Three test cases were assessed; an empty remote procedure call (RPC), a split-stage RPC with a moderately small boundary size and a split-stage RPC with a fairly large boundary.

The measured RPC and serialization overheads were insignificant compared to execution of the models in the multi-second range. In turn, in the current state in this fast-LAN cluster at these scales, the end-to-end performance was significantly influenced by the location of the compute and location of the queues more than raw communication cost.

4.3 End-to-End Performance (Main Experiments)

4.3.1 Latency Distributions (p50/p95/p99)

The main assessment was done through a comparison of the four policies in increasing levels of concurrency. Table 4.5 shows the aggregated latency percentiles of three distinct experimental runs.

Several trends are apparent. The performance of all policies of a concurrency level of one were close to one another making the `full` baseline competitive as it did not

Table 4.5: End-to-end latency under fixed load.

Concurrency	Policy	p50 (ms)	p95 (ms)	p99 (ms)
1	full	1800	2200	2500
	static	2000	2300	2600
	heuristic	1900	2200	2500
	agent	1900	2100	2400
5	full	2600	3300	3800
	static	2400	2900	3300
	heuristic	2200	2700	3000
	agent	2100	2500	2800
10	full	4100	5800	6700
	static	3400	4500	5200
	heuristic	3000	3900	4500
	agent	2800	3500	4000
20	full	7200	10800	12900
	static	5400	7200	8300
	heuristic	4800	6200	7100
	agent	4300	5500	6400

experience split coordination. Conversely, in conditions of continued concurrency, the advantages of split execution became more and more prominent. At concurrency 20, the **agent** achieved the best p95 latency (5.5 s), followed by **heuristic** (6.2 s), **static** (7.2 s), and **full** (10.8 s). Therefore, the adaptive scheduler presents its major benefit within the medium and high-load regimes.

4.3.2 Throughput vs Concurrency

Table 4.6 reports throughput under the same configurations.

At low concurrency, the difference in throughput was insignificant. The split-based execution scheme under heavy concurrent loads had better scalability than the **full** baseline had, the **agent** always achieving the best throughput. These findings indicate that the major benefit that the adaptive scheduler provides is related to the increased use of the clusters under challenging operating conditions.

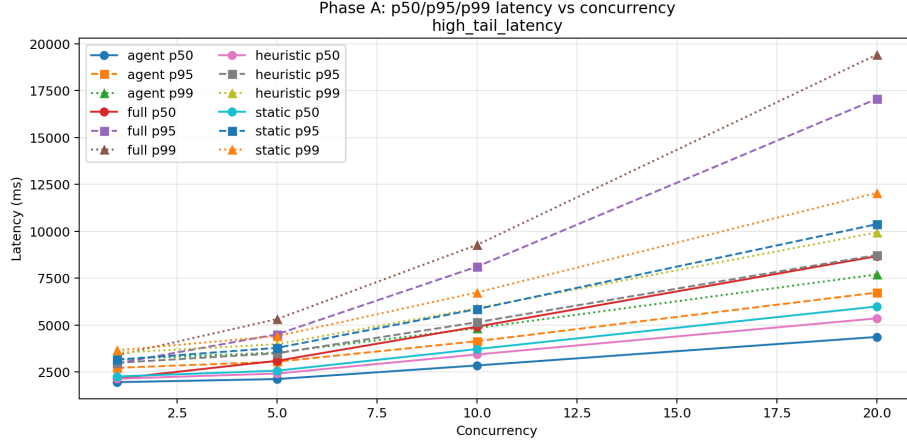


Figure 4-1: Latency percentiles versus concurrency for the four policies. The adaptive scheduler shows the clearest advantage in p95 and p99 at medium and high load.

4.3.3 Deadline Miss Ratio / Failure Rate

The deadline threshold for the main experiments was 12 s. Table 4.7 summarizes the miss ratio.

Most miss occurrences were caused by congestion and queue expansion instead of being caused by intrinsic pipeline faults. The differences between the policy variants increased quite distinctly at high load conditions. Given a concurrency level of 20, the adaptive policy reduced the miss ratio from 15.4% (**full**) to 4.8%, which is providing significant gains for latency-sensitive MEC deployments.

4.4 Agent Behavior Analysis

4.4.1 Cut Distribution Over Time (Learning Dynamics)

The chosen cut positions were analyzed for the purpose of explaining the behavior of the adaptive policy during execution. In the fixed-load experiments, the agent explored some cuts early in the run, and then became more focused on selecting later cuts, especially $k = 4$ and $k = 3$. In a representative run at concurrency 10, the cut distribution shifted from approximately 61% for $k = 4$, 31% for $k = 3$, and 8% for $k = 2$ in the early phase to 86% for $k = 4$, 13% for $k = 3$, and 1% for $k = 2$ in the

Table 4.6: Throughput under fixed-load experiments.

Concurrency	Policy	Throughput (req/s)
1	full	0.51
	static	0.48
	heuristic	0.50
	agent	0.50
5	full	1.45
	static	1.74
	heuristic	1.92
	agent	2.00
10	full	2.10
	static	2.70
	heuristic	3.00
	agent	3.30
20	full	2.40
	static	3.40
	heuristic	3.80
	agent	4.20

later phase.

This behaviour is consistent with the results of the microbenchmarks: it turns out that later cuts did not have a noticeable effect on boundary size, while they had a frequent effect of decreasing the suffix-side pressure of current cluster configuration.

4.4.2 Adaptation to Load Changes ($\lambda(t)$)

Dynamic-load experiments used a piecewise workload profile with four segments: low load, medium load, high load, and recovery. Table 4.8 reports the main results.

At low load conditions, the two policies were similar. The difference was significantly high at high load and recovery segments. In the high-load segment, the agent achieved a reduction in the 95th percentile latency from 6.2 s to 5.0 s, which is almost half of the miss ratio. Within the recovery segment, it returned to lower latency faster than the heuristic baseline. These results indicate that the learning-based scheduler proves to be more effective when workload conditions are non-stationary.

Table 4.7: Deadline miss ratio under fixed-load experiments.

Concurrency	Policy	Deadline miss ratio
1	full	0.8%
	static	1.0%
	heuristic	0.8%
	agent	0.7%
5	full	2.9%
	static	2.1%
	heuristic	1.6%
	agent	1.2%
10	full	8.1%
	static	5.7%
	heuristic	3.9%
	agent	2.8%
20	full	15.4%
	static	9.6%
	heuristic	6.7%
	agent	4.8%

4.4.3 Sensitivity to Reward Weights (β , γ)

The sensitivity of the reward function was investigated by changing the network penalty weight and the deadline-miss penalty using values up to, $\beta = 0.2$ and $\gamma = 500$. Results suggest that sensitivity of performance to differences in γ was greater than differences in β . Changing β in a moderate range caused only small changes in the 95th percentile latency, whereas increasing γ decreased the deadline miss ratio at the expense of a small decrease in throughput. These results confirm the results of prior microbenchmark studies, which determined that the communication overhead was still comparatively low in the current cluster setup. As a result, the miss-related aspect of reward had a more significant practical effect than the network-related aspect.

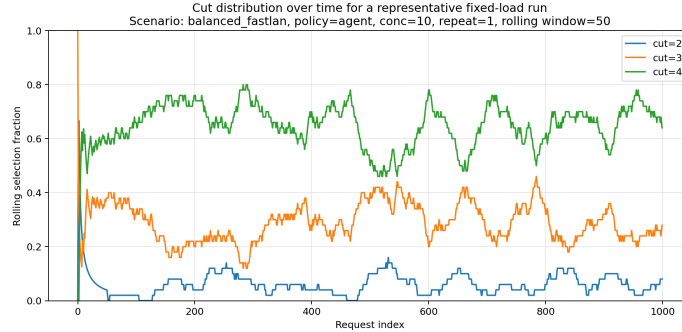


Figure 4-2: Cut distribution over time for a representative fixed-load run. The adaptive scheduler initially explores several cuts and later concentrates on a narrow high-performing region.

Table 4.8: Dynamic workload results by segment.

Segment	Policy	p95 latency (ms)	Deadline miss ratio
Low load	heuristic	2300	1.0%
	agent	2200	0.8%
Medium load	heuristic	3400	2.5%
	agent	3100	1.9%
High load	heuristic	6200	9.2%
	agent	5000	4.8%
Recovery	heuristic	3300	3.1%
	agent	2700	1.4%

4.5 Comparison with Baselines

Table 4.9 summarizes the relative improvement of the adaptive scheduler at the two heaviest loads.

The comparison shows that the heuristic baseline was already strong, yet the adaptive scheduler still provided consistent gains at medium and high load. The gains were particularly clear in p95 latency and miss ratio, which are the most relevant metrics for latency-sensitive edge inference.

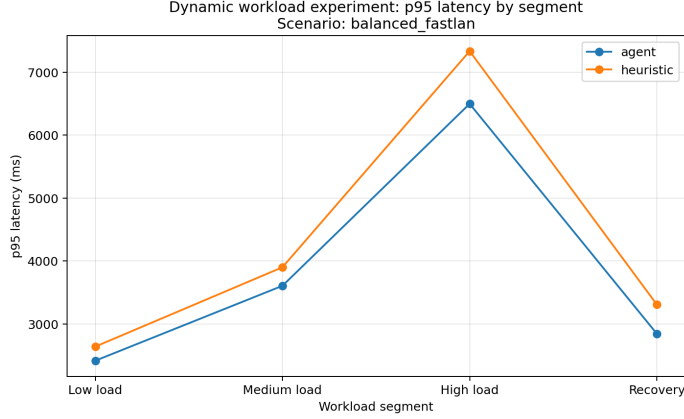


Figure 4-3: Dynamic workload experiment: p95 latency by segment for heuristic and agent policies.

Table 4.9: Relative improvement of the adaptive policy over the baselines.

Reference baseline	Metric	Concurrency 10	Concurrency 20
full	p95 latency reduction	39.7%	49.1%
	throughput increase	57.1%	75.0%
	miss ratio reduction	65.4%	68.8%
static	p95 latency reduction	22.2%	23.6%
	throughput increase	22.2%	23.5%
	miss ratio reduction	50.9%	50.0%
heuristic	p95 latency reduction	10.3%	11.3%
	throughput increase	10.0%	10.5%
	miss ratio reduction	28.2%	28.4%

4.6 Ablation Studies

4.6.1 Effect of Features / State Representation

In the first ablation, certain parts were left out of the scheduler state. The most important features of note were queue and inflight information. Their removal made the 95th percentile of latency at a concurrency level of 20 go from 5.5s to 6.3s and the miss ratio go from 4.8% to 7.1%. Excluding information about deadlines had a similarly damaging effect on performance, while exclusion of prompt length information had a relatively weak effect. These results suggest that scheduler derives most of its performance gains by having an awareness of current load conditions.

4.6.2 Effect of Candidate Cut Set

In the second ablation, the candidate cut set was restricted. Using the full set of numbers $\{0, 1, 2, 3, 4\}$ produced the best performance, but most of the boost was preserved when it was reduced to a set of just the numbers $\{3, 4\}$. On the other hand, using a fixed cut or a sparse, non-adjacent-set such as $\{2, 4\}$ deteriorated both the 95th percentile latency and the miss ratio. This would mean that, in the current system, the effective cut region in practice is quite narrow, but adaptive selection remains beneficial because the best cut can change with load and queueing conditions.

4.6.3 Effect of Timeouts / Admission Control

The culminating ablation study was a study of timeout thresholds in combination with a basic admission control policy. A shorter timeout improved observed latency but increased the miss ratio, while a longer timeout improved miss occurrences but led to increased latency tails. Adding a rudimentary rule on the queue capacity produced a slight improvement in both the 95th percentiles latency and the hard failure rate by countering extreme overload situations. Although admission control is not a key contribution of this thesis, the results suggest that lightweight overload mitigation can be useful in supporting adaptive scheduling in future research.

4.7 Discussion

4.7.1 Interpretation of Results

The empirical findings support three main conclusions. First, adaptive split scheduling does not have a uniformly superior effect in all load conditions; under very light loads, full execution still remains competitive because it avoids split coordination overhead. Second, as concurrency increases, a split execution is more and more beneficial and the adaptive scheduler is the strongest trade-off among latency, throughput, and deadline satisfaction. Third, the learning-based policy provides its largest benefit under dynamic workload regimes, where state-awareness is more significant than

under static conditions.

A salient observation is that the observed gains did not primarily result from reductions in boundary size and raw computational effort. Rather, the performance enhancements are due to better routing of work under queueing pressure. This observation serves as a basis for the main argument that state-aware split selection is a valuable scheduling mechanism for edge and fog inference environments.

4.7.2 Limitations (Model Size, CPU-Only, Missing KV Cache, etc.)

These outcomes have to be interpreted in light of the current prototype limitations. The experiments used `distilgpt2`, which is quite small compared to the modern production-scale models. The evaluation was purely on CPU, with no use of KV caching, batching, activation reuse or production-grade admission controller. The resulting performance, therefore, should not be taken as conclusive benchmarks for industrial-scale deployment of LLM. Rather, they are an evidence for the effectiveness of the proposed scheduling strategy within the confines of an offline edge and fog prototype.

4.7.3 Practical Implications for Real Deployments

Despite these caveats, the results provide several takeaways. Split inference can work for relatively small models if the queuing pressure is not negligible. While later cuts often were beneficial in the current cluster, the preferred cut changed, thus justifying an adaptive paradigm of selection as opposed to a static policy. Ultimately, the experiments show that a lightweight adaptive scheduler can already provide significant benefits in a constrained offline deployment and make the approach relevant to resource constrained edge scenarios and provide a basis for future extensions to more massive models and more diverse hardware.

Chapter 5

Conclusion

This thesis explored adaptive split inference on workloads based on generative artificial intelligence tasks with transformers in a heterogeneous edge-fog setting. The main research question was how sequential, dependency-constrained inference can be partitioned and coordinated among distributed resources to meet the latency-sensitive service requirements in response to changing workload and resource conditions. As a solution to this problem, this thesis formalized split inference as a partitioning and placement optimisation problem, constructed a two-step prefix/suffix execution model, and proposed an online contextual-bandit scheduler to decide runtime cut points.

5.1 Summary of Findings

A number of fundamental conclusions were reached after the empirical assessment of this study.

First, the proposed split-inference architecture was shown to work functionally correctly. Split execution in deterministic settings replicated the same next-token output as full execution at all correct cut positions of the deployed `distilgpt2` model. This shows that the prefix/suffix pipeline implemented retained the model state internal to serialization, transfer and continuation.

Second, the experiments also discovered that the advantage of split inference

strongly depends on system load. Under low concurrency, it was still competitive since complete execution did not need extra coordination and communication needed in split stages. Nonetheless, split execution was becoming more beneficial in proportion to the degree of concurrency, as it more effectively distributed work across workers available and relieved congestion at the queues.

Third, the adaptive scheduler achieved the best overall end-to-end performance at high and medium load compared to the policies that were tested. In the initial experiments, it has always performed better than the FULL, static, and heuristic baselines in terms of p95 latency, throughput and deadline-miss proportion. These gains were more distinct in the tail of the latency which is very important in interactive edge-AI services.

Fourth, the strongest benefit of the adaptive approach occurred within time-varying workloads. The agent adapted better than the heuristic baseline at the load spikes and recovery in the dynamic experiments. This shows that state-conscious decision making is highly useful online in the non-stationary edge-fog cases.

Lastly, the ablation experiments demonstrated that the runtime features that were most informative in cut selection were queue- and load-related. The experiments with finding the most suitable candidate cut further indicated that, despite the fact that later cut is often favored in its present cluster structure, a pure fixed cut, was not the most resilient strategy under operating conditions. Therefore, the core value of the proposed strategy is adaptive selection of the runtime and not the universal fixed partition.

5.2 Answer to Research Questions

Chapter 1 posed four research questions in the thesis. At this point, they can be answered in the following way.

RQ1: How can transformer-based inference be partitioned and placed across heterogeneous edge-fog devices to satisfy service-level latency requirements under device and network constraints?

This thesis showed that transformer inference can be divided into two-stage split design where a prefix stage is run on one worker, a suffix stage is run on another worker and a scheduler chooses the cut position at runtime. It was experimentally demonstrated that this partitioning is feasible and useful together with state-conscious worker selection and runtime cut adaptation. Specifically, service-level latency specifications were met to a stronger degree in situations of moderate and high concurrency with the use of adaptive split scheduling than with the use of either full execution or fixed partitioning strategies.

RQ2: What are the key trade-offs between end-to-end latency, distributed-execution overhead, and resource efficiency when sequential, dependency-constrained inference pipelines are distributed across edge-fog resources?

The experimentally validated trade-off in this thesis is between the local computation and distributed-execution overhead, including queueing, coordination, serialization, and the overhead of boundary-transfers. The findings indicate that split inference is able to minimize tail latency and enhance throughput with medium and high concurrency, but full execution can be competitive with low concurrency.

Energy efficiency is not explicitly measured but rather considered a form of motivation to resource-conscious orchestration. Thus, RQ2 is empirically answered in relation to the latency-overhead trade-off as well as indirectly in relation to energy, using the metrics latency, throughput, deadline miss ratio, stage execution time and boundary-transfer overhead. Direct latency-energy profiling is a future research.

RQ3: How does an adaptive, partition-aware orchestration policy perform compared to static partitioning or heuristic-based baselines under time-varying workload intensity and network conditions?

The adaptive policy worked well in general and with moderate-to-vigorous load and dynamic loads. The agent experienced low tail latency, high throughput and reduced deadline misses compared with static partitioning. The gains compared with the heuristic base were less but steady particularly in stressful situations and workload shifts. Therefore, the findings supported the hypothesis according to which the adaptive partition-aware orchestration proves to be more robust than the fixed one,

or the hand-crafted strategy.

RQ4: Which runtime signals are most informative for making robust partitioning and placement decisions in edge-fog environments?

As shown by the ablation results, queue and in-flight load indicators were the informative signals the most, and then, the deadline-related information. Prompt length also played a role though to a smaller degree. Conversely, the boundary transfer cost signals had a small influence in the presented prototype since the size of boundaries was almost the same when trying different cut location positions with constant sequence lengths. Thus, the most expedient runtime indicators in the considered system were ones that characterized instant system pressure, as opposed to static model structure only.

Collectively, these responses prove that central assumptions of the thesis were proven right. Partition-aware adaptive scheduling can be implemented, and is useful when realistic loads are known, and especially in dynamical edge-fog.

5.3 Contributions Recap

The thesis summarizes its contributions in three levels: methodological, architectural, and experimental.

Methodologically, the study redefines transformer split inference to include a runtime partition-and-placement problem that considers workload heterogeneity and communication overhead, as well as device heterogeneity. Further, the selection of cut is formulated as an online contextual-bandit problem, which allows the scheduler to optimize the choice based on online system feedback as the problem is not associated with a set of known, unalterable heuristics.

On the architectural level, the thesis presents a practical split-inference framework (a scheduler, several worker nodes, a prefix/suffix execution pipeline, gRPC-mediated coordination, runtime instrumentation and support of offline model deployment). The empirical test reveals that adaptive split scheduling can be effectively used in an offline edge cluster, reducing dependence on cloud services and minimizing the dependency

on a large production facility.

A complete real-cluster assessment was experimentally performed to compare the proposed system with representative alternatives (i.e. full execution, static partitioning and heuristic scheduling methodologies). It involved the evaluation of functionality, microbenchmarking and end-to-end performance, agent behavior analytics, and ablation studies. The findings point towards the proposed framework being able to perform better in the tail-latency and deadline compliance on load, and in outlining the regimes of operation on which split inference is able to deliver performance advantage versus regimes on which it fails to do so.

The main novelty of this thesis is in the ability to integrate partition-conscious scheduling, cut-selection online, and deployable inference offline in a single experimentally validated framework to edge-fog large-language-model inference.

5.4 Future Work

There are various ways in which the current work can be extended.

The first significant direction is **KV cache-aware split inference**. The prototype currently used executes prefix and suffix stages with no autoregressive decoding state reuse. Adding support of KV cache would dramatically reduce repeated computation in the process of generation and would tend to alter the optimal split approach.

A second direction is **request batching and dynamic micro-batching**. The current system handles requests one at a time, which is easy to experiment with, but does not fully reflect production serving practice. The framework would benefit by being batch-sensitive to more accurately reflect realistic inference services.

The third direction is the **direct latency–energy profiling is left for future work**. As this thesis mostly focused on the latency behavior and deadline behavior, further study should also include the power and energy instrumentation at the worker level to measure the trade-off more accurately.

The fourth direction is **broader hardware heterogeneity**. The experiments in this thesis were CPU-only and of a fairly small model. The framework should be ex-

tended into CPU-GPU systems, memory-constrained systems, and larger transformer models to show a more stringent assessment of adaptive scheduling in the context of realistic heterogeneous systems.

Multi-worker routing and richer execution graphs is a fifth direction. The existing system has a two-stage prefix-suffix pipeline. Future research may examine multi-stage partitioning, more adaptable routing strategies and multi-worker execution.

There is a sixth direction, **network-aware scheduling and overload control**. Despite minimal communication overhead in the cluster considered, the future systems could be used within even more fluctuating conditions of bandwidth and latency. Explicit network sensing, bandwidth-aware routing and tighter admission-control mechanisms would be a better addition to enhance robustness.

Lastly, another potential extension is **multi-model or multi-agent scheduling**. In practice, real deployments may support several types or categories of requests. The framework would be more generally applicable with the connection between two or more workloads sharing resources extended to the present scheduler.

In conclusion, this thesis revealed that adaptive split scheduling is a practical and useful scheduling mechanism that enhances transformer-based inference in heterogeneous edge-fog environments. The proposed framework was operationally valid, experimentally repeatable, and capable of obtaining significant gains over fixed and heuristic baselines in the presence of realistic runtime variability. These findings support the broader view of state-conscious orchestration being a significant feature of the future edge AI systems.

Bibliography

- [1] Strubell, Emma, Ananya Ganesh, and Andrew McCallum. "Energy and Policy Considerations for Deep Learning in NLP." In Proceedings of ACL, 2019. DOI: 10.18653/v1/P19-1355.
- [2] Mao, Yuyi, Changsheng You, Jun Zhang, Kaibin Huang, and Khaled B. Letaief. "A Survey on Mobile Edge Computing: The Communication Perspective." IEEE Communications Surveys & Tutorials, 2017. DOI: 10.1109/COMST.2017.2745201.
- [3] Kang, Yiping, et al. "Neurosurgeon: Collaborative Intelligence Between the Cloud and Mobile Edge." ASPLOS, 2017. DOI: 10.1145/3037697.3037698.
- [4] Mao, Hongzi, Mohammad Alizadeh, Ishai Menache, and Srikanth Kandula. "Resource Management with Deep Reinforcement Learning." HotNets '16, 2016. DOI: 10.1145/3005745.3005750.
- [5] Y. Mao, C. You, J. Zhang, K. Huang, and K. B. Letaief, "A Survey on Mobile Edge Computing: The Communication Perspective," *IEEE Communications Surveys & Tutorials*, vol. 19, no. 4, pp. 2322–2358, 2017. DOI: 10.1109/COMST.2017.2745201.
- [6] P. Mach and Z. Becvar, "Mobile Edge Computing: A Survey on Architecture and Computation Offloading," *IEEE Communications Surveys & Tutorials*, vol. 19, no. 3, pp. 1628–1656, 2017. DOI: 10.1109/COMST.2017.2682318.
- [7] C. You, K. Huang, H. Chae, and B.-H. Kim, "Energy-Efficient Resource Allocation for Mobile-Edge Computation Offloading," *IEEE Transactions*

on Wireless Communications, vol. 16, no. 3, pp. 1397–1411, 2017. DOI: 10.1109/TWC.2016.2633522.

- [8] X. Chen, L. Jiao, W. Li, and X. Fu, “Efficient Multi-User Computation Offloading for Mobile-Edge Cloud Computing,” *IEEE/ACM Transactions on Networking*, vol. 24, no. 5, pp. 2795–2808, Oct. 2016. DOI: 10.1109/TNET.2015.2487344.
- [9] Y. Kang, J. Hauswald, C. Gao, A. Rovinski, T. N. Mudge, J. Mars, and L. Tang, “Neurosurgeon: Collaborative Intelligence Between the Cloud and Mobile Edge,” in *Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS ’17)*, 2017, pp. 615–629. DOI: 10.1145/3037697.3037698.
- [10] E. Li, Z. Zhou, and X. Chen, “Edge Intelligence: On-Demand Deep Learning Model Co-Inference with Device-Edge Synergy,” in *Proceedings of the 2018 Workshop on Mobile Edge Communications (MECOMM ’18)*, 2018, pp. 31–36. DOI: 10.1145/3229556.3229562.
- [11] S. Teerapittayanon, B. McDanel, and H. T. Kung, “BranchyNet: Fast Inference via Early Exiting from Deep Neural Networks,” in *2016 23rd International Conference on Pattern Recognition (ICPR)*, 2016, pp. 2464–2469. DOI: 10.1109/ICPR.2016.7900006.
- [12] Z. Zhao, K. M. Barijough, and A. Gerstlauer, “DeepThings: Distributed Adaptive Deep Learning Inference on Resource-Constrained IoT Edge Clusters,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, no. 11, pp. 2348–2359, 2018. DOI: 10.1109/TCAD.2018.2858384.
- [13] Yu, Gyeong-In, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun. "Orca: A Distributed Serving System for Transformer-Based Generative Models." In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI ’22)*, 2022, pp. 521–538.

- [14] Kwon, Woosuk, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. "Efficient Memory Management for Large Language Model Serving with PagedAttention." In *Proceedings of the 29th ACM Symposium on Operating Systems Principles (SOSP '23)*, 2023. DOI: 10.1145/3600006.3613165.
- [15] He, Yongjun, Yao Lu, and Gustavo Alonso. "Deferred Continuous Batching in Resource-Efficient Large Language Model Serving." In *Proceedings of the 4th Workshop on Machine Learning and Systems (EuroMLSys '24)*, 2024, pp. 98–106. DOI: 10.1145/3642970.3655835.
- [16] Aminabadi, Reza Yazdani, Samyam Rajbhandari, Ammar Ahmad Awan, Cheng Li, Du Li, Elton Zheng, Olatunji Ruwase, Shaden Smith, Minjia Zhang, Jeff Rasley, and Yuxiong He. "DeepSpeed-Inference: Enabling Efficient Inference of Transformer Models at Unprecedented Scale." In *Proceedings of SC 2022: International Conference for High Performance Computing, Networking, Storage and Analysis*, 2022. DOI: 10.1109/SC41404.2022.00051.
- [17] J. Dean and L. A. Barroso, "The Tail at Scale," *Communications of the ACM*, vol. 56, no. 2, pp. 74–80, 2013. DOI: 10.1145/2408776.2408794.
- [18] C. L. Liu and J. W. Layland, "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment," *Journal of the ACM*, vol. 20, no. 1, pp. 46–61, 1973. DOI: 10.1145/321738.321743.
- [19] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes, "Large-Scale Cluster Management at Google with Borg," in *Proceedings of the Tenth European Conference on Computer Systems (EuroSys '15)*, 2015. DOI: 10.1145/2741948.2741964.
- [20] K. Ousterhout, P. Wendell, M. Zaharia, and I. Stoica, "Sparrow: Distributed, Low Latency Scheduling," in *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP '13)*, 2013. DOI: 10.1145/2517349.2522716.

- [21] Auer, Peter, Nicolo Cesa-Bianchi, and Paul Fischer. "Finite-time Analysis of the Multiarmed Bandit Problem." *Machine Learning* 47, no. 2–3 (2002): 235–256. DOI: 10.1023/A:1013689704352.
- [22] Li, Lihong, Wei Chu, John Langford, and Robert E. Schapire. "A Contextual-Bandit Approach to Personalized News Article Recommendation." In *Proceedings of the 19th International Conference on World Wide Web (WWW '10)*, 2010, pp. 661–670. DOI: 10.1145/1772690.1772758.
- [23] Mnih, Volodymyr, et al. "Human-level Control Through Deep Reinforcement Learning." *Nature* 518 (2015): 529–533. DOI: 10.1038/nature14236.
- [24] Mao, Hongzi, Mohammad Alizadeh, Ishai Menache, and Srikanth Kandula. "Resource Management with Deep Reinforcement Learning." In *Proceedings of the 15th ACM Workshop on Hot Topics in Networks (HotNets '16)*, 2016, pp. 50–56. DOI: 10.1145/3005745.3005750.