POLITECNICO DI TORINO

Master Degree in Communications Engineering

Master Degree Thesis

Collaborative Multi-point User Localization with Joint Communication and Sensing

Supervisors

Prof. Carla Fabiana CHIASSERINI

Phd. Marco PALENA

Candidate

Zhenkai ZHOU

December 2025

Acknowledgements

I would like to express my gratitude to my supervisor, Prof. Carla Fabiana Chiasserini, for providing me with the opportunity and support to work on this thesis.

I am deeply indebted to Ph.D. Marco Palena for his continuous mentorship and unwavering support. His patience, technical expertise, and constructive feedback were indispensable during the long process of research and implementation. This work would not have been possible without his constant encouragement and detailed advice.

I extend my thanks to the colleagues at the University of Brescia for providing the real-world CSI dataset, which facilitated the experimental validation of this research.

Finally, I would like to express my deepest thanks to my family. Their unconditional love, understanding, and silent support have been my greatest motivation throughout these years of study. I am also affectionately grateful to the classmates and friends I met here in Turin. Thank you for your company, for the encouragement when things got tough, and for all the shared memories that made this journey truly unforgettable.

Abstract

Joint Communication and Sensing (JCS) is a key feature of 6G networks. Multi-point localization using Wi-Fi Channel State Information (CSI) is a prime example of JCS. However, deploying such applications at the network edge is complex. It requires fusing CSI data collected by multiple access points (APs) and managing significant computational workloads on resource-constrained devices. This creates a two-fold challenge: 1) At the application level, how much does collaboration (using multiple APs) actually improve performance, and how robust is this performance to sensor (AP) failure? 2) At the system level, how can we deploy these collaborative applications (which have different strategies, like model splitting or data fusion) in a dynamic and efficient way, instead of using static, manual configurations?

This thesis addresses both challenges. First, we implement and evaluate a complete indoor localization pipeline, we train this pipeline on a dataset of CSI traces collected from a multi-antenna, multi-anchor real-world scenario, creating two distinct versions: one for regression (predicting coordinates) and one for classification (predicting location labels). We use this pipeline to conduct a detailed robustness analysis. Our results provide a key finding: data fusion is not just beneficial, it is essential for reliable localization. We show that performance in a surveyed environment collapses from 99.99% accuracy (with 5 APs) to 55.64% (with 1 AP). Similarly, for regression task, the Mean Absolute Error (MAE) increases significantly from 3.62 cm (with 5 APs) to 73.59 cm (with 1 AP). Second, to manage such applications, we design and implement a novel distributed control-layer solution, the Collaborative Inference Manager (CIM). This is an orchestration service that uses cloud-native tools (Docker, Kubernetes) to simulate a distributed edge environment, where each container represents an independent computing node. It can take API requests and automatically deploy the CSI localization pipeline using two different strategies: one based on model partitioning and the other based on data-fusion.

The contribution of this thesis is therefore two-fold: we provide both a quantitative analysis that proves the necessity of multi-point data fusion for CSI localization, and a dynamically reconfigurable framework to deploy and manage such collaborative sensing applications efficiently within a simulated environment.

Keywords: Joint Communication and Sensing, Collaborative Inference, Wi-Fi CSI Localization, Edge Intelligence

Contents

Chapter 1

Introduction

1.1 Background and Motivation

As fifth-generation (5G) and emerging sixth-generation (6G) communication networks continue to evolve, the architecture and functionality of modern networks are undergoing a fundamental transformation. This evolution is defined not only by significant performance gains—such as higher data rates, ultra-low latency, and massive device connectivity—but also by a strategic shift in network capability. Traditionally, communication networks have primarily served as data transmission channels, responsible for delivering information between endpoints. However, with the rapid advancement of network technologies like massive MIMO, beamforming, and network slicing [1], this paradigm is expanding toward intelligent, context-aware platforms that integrate Joint Communication and Sensing (JCS) capabilities. [2, 3]. In this model, the same network infrastructure and wireless signals are used not only for data transmission but also for actively sensing the surrounding physical environment.

This dual JCS capability, combined with the performance of 5G/6G, enables a new class of sophisticated, real-time applications such as autonomous driving, immersive AR/VR, and Industrial IoT (IIoT). However, these applications cannot function by merely relying on a distant, centralized cloud, as the latency of round-trip data transmission is prohibitive. This limitation has given rise to Edge Intelligence (EI) [4], a paradigm that leverages Machine Learning (ML) execution directly at the network edge, closer to the source of data.

The motivation for Edge Intelligence is multi-faceted and represents a direct response to the failures of the traditional, centralized ML model in this new context. First, centralized models face an insurmountable bandwidth and latency bottleneck; requiring countless edge devices (e.g., sensors, vehicles, cameras) to stream raw, high-dimensional data to a central server for processing is both technically infeasible and economically unsustainable, while also failing to meet the millisecond-level latency requirements of real-time control loops. Second, the centralized approach introduces severe privacy and security risks. Transmitting raw, often sensitive, sensor data across the public internet to a third-party cloud creates a large attack surface and forces users to relinquish control of their data. Edge Intelligence directly addresses these challenges by processing data locally,

enhancing privacy, minimizing bandwidth consumption, and enabling near-instantaneous inference.

However, the practical realization of Edge Intelligence introduces its own fundamental tension: the applications that benefit most from it, such as real-time video analytics or radio-frequency sensing, increasingly rely on large, computationally expensive Deep Neural Network (DNN) models. This requirement stands in stark contrast to the defining characteristic of edge devices (like Wi-Fi Access Points, routers, or IoT gateways), which are, by design, resource-constrained in terms of computational power, memory, and energy budget.

Wi-Fi Channel State Information (CSI) localization emerges as a prime exemplar of this exact challenge. As an application, it is a compelling use case for JCS, offering high-precision, "device-free" indoor positioning. The high-dimensional nature of CSI data, which captures a detailed "fingerprint" of the multipath wireless environment, makes it ideal for sophisticated DNN-based analysis. However, this same data richness presents two distinct problems. First, it reinforces the computational challenge, as the DNNs required to process these complex inputs are too demanding for a single, low-power edge device. Second, robust localization in complex indoor spaces inherently requires a multi-point, collaborative approach; relying on data from a single Access Point (AP) often leads to significant ambiguity and poor performance due to signal fading and "blind spots."

This dual requirement—the need for computationally intensive models combined with the necessity for multi-point data collection—renders the traditional monolithic deployment (one large model on one node) infeasible. It directly necessitates the development of new Collaborative Inference (CI) paradigms. These paradigms are designed to allow multiple, geographically distributed, resource-limited devices to work in concert. This collaboration, however, is not monolithic itself; it presents a complex trade-off, forcing a choice between distinct strategies—such as model partitioning (splitting a single model) versus data fusion (combining multiple results). The design and management of these strategies, represent a necessary step for integrating advanced AI services into future communication networks.

1.2 Problem Statement

Achieving multi-point collaborative inference in an edge environment is essential. This high-level goal, however, introduces two distinct and often competing low-level problems:

1. **The Computational Problem:** How can the significant computational workload of a large DNN model be executed on edge devices that individually lack the necessary resources (CPU, memory)?

2. **The Fusion Problem:** How can data (or results) from multiple, distributed sensors (APs) be effectively and efficiently combined? For complex sensing tasks like CSI localization, relying on a single sensor's perspective is insufficient, often leading to signal ambiguity and a catastrophic collapse in performance. Therefore, a multi-point fusion strategy is not just beneficial, but essential for achieving robust and accurate predictions.

To address these problems, different competing strategies have emerged, each with its own significant trade-off.

- **Solution A: Model Partitioning (Split Computing)** This strategy directly addresses the Computational Problem. It involves vertically splitting a large, monolithic DNN model into sequential partitions (e.g., splitting at an intermediate layer), which are then distributed across different nodes to be executed in a pipeline sequence.
  - **Trade-off:** This solves the computation bottleneck but introduces a new communication bottleneck. Large intermediate tensors (the output from a layer) must be sent over the network between each split, which can add significant latency, especially on wireless links.
- **Solution B: Data Fusion (Ensemble Methods)** This strategy directly addresses the Fusion Problem for multi-source tasks like CSI localization. However, it forces a choice between three sub-strategies:
  - **Early Fusion:** Combine raw data at the input. This requires all APs to send their high-dimensional sensor data (such as unprocessed CSI subcarrier information) to a central node for aggregation.
    - **Trade-off:** This approach suffers from unacceptable communication overhead.
  - **Intermediate Fusion (Mid-Fusion):** A hybrid approach where each device locally processes raw data to extract intermediate feature representations. These features (which are significantly smaller than raw data but richer than a final prediction) are then sent to a central node for fusion before the final inference stage.
    - **Trade-off:** This offers a balance, reducing the high communication overhead of Early Fusion while avoiding the full computational redundancy of Late Fusion, but requires careful model design.
  - **Late Fusion (Ensemble):** Combine results at the decision layer. Each data source performs a full prediction by running a local version of the model. It then sends only a low-dimensional output (e.g., a coordinate or probability map) to a shared fusion server, which aggregates the individual results using a consensus algorithm to produce the final decision.
    - **Trade-off:** This solves the communication overhead but introduces high computational redundancy, as multiple nodes are all running the same full model.

Gap Analysis

This highlights the central problem addressed in this thesis: the choice between strategies such as model partitioning and early, intermediate, or late fusion is currently static and

manually defined. Implementations are usually hard-coded to a single strategy, making them unable to adapt to changing conditions.

However, the optimal strategy—a quantifiable trade-off between latency, accuracy, and computational resource usage—is not fixed. It strongly depends on real-time factors:

- **Network Conditions:** A high-bandwidth link makes Model Partitioning or Early/Intermediate Fusion more viable. A high-latency, low-bandwidth link makes Late Fusion preferable.
- **Node Status:** High CPU load on edge nodes makes the computational redundancy of Late Fusion an inefficient choice.

Therefore, the research gap is not merely the absence of a final automatic selection algorithm, but the absence of the foundational enabling technologies required to support such an algorithm. The field remains constrained by static, hard-coded implementations. A unified control framework that can dynamically manage, deploy, and reconfigure these competing strategies (CI vs. DF) on demand is a necessary prerequisite. This thesis aims to fill this specific gap by designing and implementing such an orchestration framework, moving the field away from static configurations and providing the essential groundwork for future, fully adaptive systems.

1.3 Research Objectives

Based on the problems identified, the core objective of this thesis is to design, implement, and evaluate a Collaborative Inference Manager. This is a distributed control-layer solution designed to fill the previously mentioned research gap.

The specific goals to achieve this objective are defined as follows:

1. **Design a Dynamic Orchestrator:** To implement a control service that can receive high-level inference tasks (e.g., perform CSI localization) and a set of constraints (e.g., prioritize latency). It must then dynamically generate all necessary deployment files to set-up the localization service.
2. **Support Dual-Strategy Deployment:** The manager must be able to automate the deployment of two different collaborative strategies:
    - **CI Strategy (Model-Parallelism):** Automatically split a given DNN at specified cut-points and deploy it as a K8s service chain.
    - **DF Strategy (Data-Parallelism):** Automatically deploy a variable number of model instances as parallel K8s workers, with a late-fusion server.
3. **Implement using Cloud-Native Tools for Simulation:** The solution must use containerization (Docker) and orchestration (Kubernetes, specifically using k3d) as cloud-native technologies. The main goal here is to simulate a realistic, multi-node edge environment on a local machine. This approach allows us to deploy and test our distributed services (CI and DF) in a controlled way, where each container (Pod) acts like a separate edge device (e.g., an AP or edge server).

4. **Evaluate and Validate the System and its Trade-offs:** We must validate the system using a real-world CSI localization use case. This evaluation will be two-fold:
    - **Application-Level Justification:** First, we must provide the fundamental justification for a multi-point sensing approach (which is the premise for this entire thesis) by measuring how localization accuracy degrades as data from collaborating APs is lost.
    - **System-Level Evaluation:** Second, we must quantify the deployment trade-offs of the two strategies (CI vs. DF) managed by our orchestrator. This will be achieved by deploying the system in the simulated environment and measuring its online performance metrics, such as end-to-end latency, throughput, and resource consumption.

1.4 Contributions

In this thesis, our main goal was to design, implement, and evaluate a dynamic control-layer solution for collaborative inference, which we call the Collaborative Inference Manager. We wanted to move beyond static, manually-configured deployments and create a system that could automatically deploy different collaborative strategies (CI vs. DF) based on high-level API requests.

To achieve this goal, we developed several key components. First, we implemented a complete, multi-point CSI localization pipeline to serve as a realistic and complex reference application. We trained this pipeline CSI traces collected from a real-world scenario. This involved processing raw sensor data, aligning multi-node timestamps, and implementing a feature extraction and training workflow.

More importantly, we designed and implemented the orchestration system itself. This core contribution includes a novel orchestration service that acts as the "brain" managing API requests, and a templating engine (using Jinja2) that automatically generates all necessary `Dockerfile` and Kubernetes `YAML` manifests. This system is capable of deploying the two core collaborative strategies discussed in this thesis: a model-split (CI) pipeline and a late-fusion (DF) ensemble, the latter of which includes an asynchronous fusion server.

Finally, these tools allowed us to perform a quantitative analysis of these collaborative strategies. We provide a core trade-off analysis that evaluates system robustness against sensor (AP) failures. This analysis validates the necessity of our DF strategy and provides a strong baseline for future research into dynamic, adaptive orchestration.

1.5 Thesis Outline

This thesis is organized as follows:

- **Chapter 2: Background and Related Work** reviews related work in edge AI, collaborative inference (CI/DF), Wi-Fi CSI sensing, and cloud-native network orchestration.

- **Chapter 3: System Design and Methodology** details the architecture of the Collaborative Inference Manager and analyzes the automated deployment workflows for the CI and DF strategies.
- **Chapter 4: Experimental/numerical evaluation** describes the implementation of our reference CSI application (the data pipeline and models) and our evaluation methodology. It then presents and analyzes our key experimental findings, focusing on the AP robustness analysis (which quantifies the value of DF, showing a performance increase from 55.64 % to 99.99 %) and the performance trade-offs of the CI pipeline strategy.
- **Chapter 5: Conclusions and Future Work** summarizes the contributions, identifies current limitations, and proposes future research directions towards a proactive orchestrator.

Chapter 2

Background and Related Work

This chapter provides the technical background needed to understand this thesis. We review four key areas: machine learning at the network edge, the main collaborative inference methods, Wi-Fi CSI sensing as our use case, and cloud-native orchestration as our implementation base. Finally, we identify the research gap that this thesis aims to fill.

2.1 Machine Learning at the Network Edge

Traditionally, Machine Learning (ML) relied on powerful, centralized cloud data centers for training and inference. However, this model is not suitable for modern applications that need very low latency, data privacy, and efficient bandwidth use. As a result, Edge Intelligence has emerged. The goal of Edge Intelligence is to move AI computation away from the distant cloud [5] and closer to the data source, such as on 5G base stations, edge servers, or the devices themselves.

Federated Learning (FL) [6] is a well-known paradigm in Edge AI. It focuses on distributed training. Devices train models locally and only share the model updates (gradients), not their private raw data.

This thesis, however, focuses on a different but equally important area: Distributed Inference. We do not focus on how the model is trained. Instead, we study how a large, pre-trained model can be deployed across multiple resource-limited devices so they can work together to perform an inference task. Our Collaborative Inference Manager is a solution for this distributed inference problem.

2.2 Collaborative Inference Paradigms

Distributed inference can be done using different collaborative strategies. When multiple nodes in an edge network want to solve one task, they can choose to collaborate by "splitting" the model or by "fusing" their data.

2.2.1 Model Partitioning (Split Computing)

Split Computing, or model partitioning, is a model-parallel strategy. The main idea is to split a single, large DNN (like a large, complete model) vertically into several sequential parts, or "shards".

For example, a model can be split into a "head" and a "tail". The device (like a mobile phone) runs the head (the first few layers). It then sends the output, called an intermediate tensor, over the network to an edge server, which runs the tail (the remaining layers) to get the final result [7].

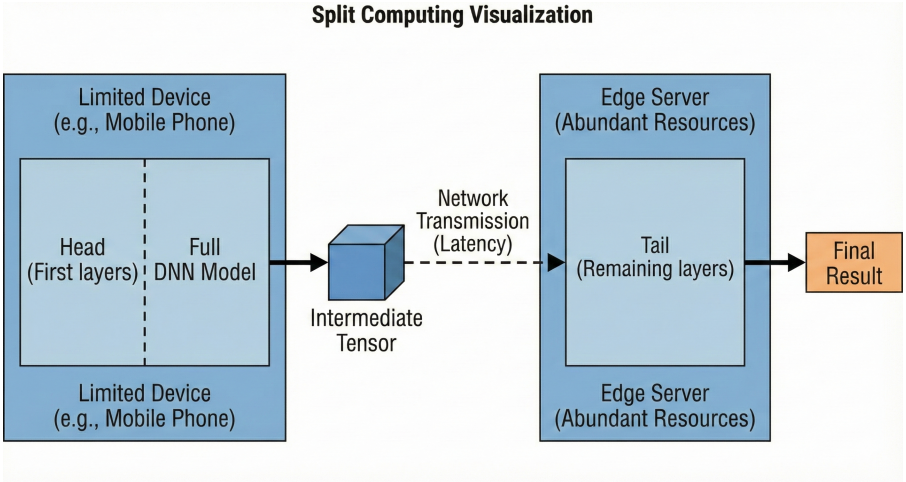

Figure 2.1. Visualization of Model Partitioning (Split Computing). The DNN is vertically split into a Head (executed on the limited device) and a Tail (executed on the edge server), transmitting an intermediate tensor over the network.

- **Pros:** This method reduces the computation and memory load on a single device, especially the one with the most limited resources.
- **Cons:** It creates a new communication overhead. The intermediate tensor can be very large. Sending it over the network (especially a wireless one) adds significant latency. This delay can cancel out the time saved by splitting the computation.

The CI (Collaborative Inference) strategy in this thesis is an implementation of this split computing paradigm.

2.2.2 Ensemble Methods and Data Fusion

Data Fusion (DF) is a data-parallel strategy. It is used when multiple sensors (like multiple Wi-Fi APs) are observing the same target. Combining their information can produce a single estimate that is more accurate and reliable than any single sensor. For CSI localization, fusing data from multiple APs is essential [8]. The main fusion strategies are:

- **Early fusion:** This is fusion at the input (or feature) level. All APs send their raw CSI data (or extracted features) to a central node. This node **concatenates** all features into one very large vector and feeds it into a single, large model.

  - *Pros:* Allows the model to find complex, low-level correlations between the data from different APs.

  - *Cons:* Creates a massive communication overhead because the raw, high-dimensional CSI data must be transmitted over the network.

- **Intermediate Fusion (Mid-Fusion):** This is a hybrid strategy where fusion occurs at the feature representation level. Each AP locally processes the raw data through the initial layers of a neural network to extract abstract intermediate features. These feature tensors are then transmitted to a central node, which aggregates them and executes the remaining layers of the model to produce the final prediction.

  - *Pros:* Offers a distinct balance in the trade-off spectrum. It significantly reduces communication overhead compared to Early Fusion (as extracted features are often more compact than raw data) while avoiding the full computational redundancy of Late Fusion.

  - *Cons:* Requires complex model design to identify the optimal "split point" for feature extraction. The communication cost, while lower than raw data, is still significantly higher than transmitting simple probability maps (Late Fusion).

- **Late fusion:** This is fusion at the decision (or probability) level. Each AP (or a nearby node) independently runs a full inference model locally. It produces its own low-dimensional output (e.g., a location probability map). A fusion server (like a central fusion server) then collects only these small outputs and combines them. It can use simple methods like `average` or Bayesian methods like `conflation` to get the final decision.

  - *Pros:* Has very low communication overhead (only the final results are sent). It is also very robust; if one node fails, the system can still work.

  - *Cons:* Causes high computational redundancy, as every node has to run the same complete model.

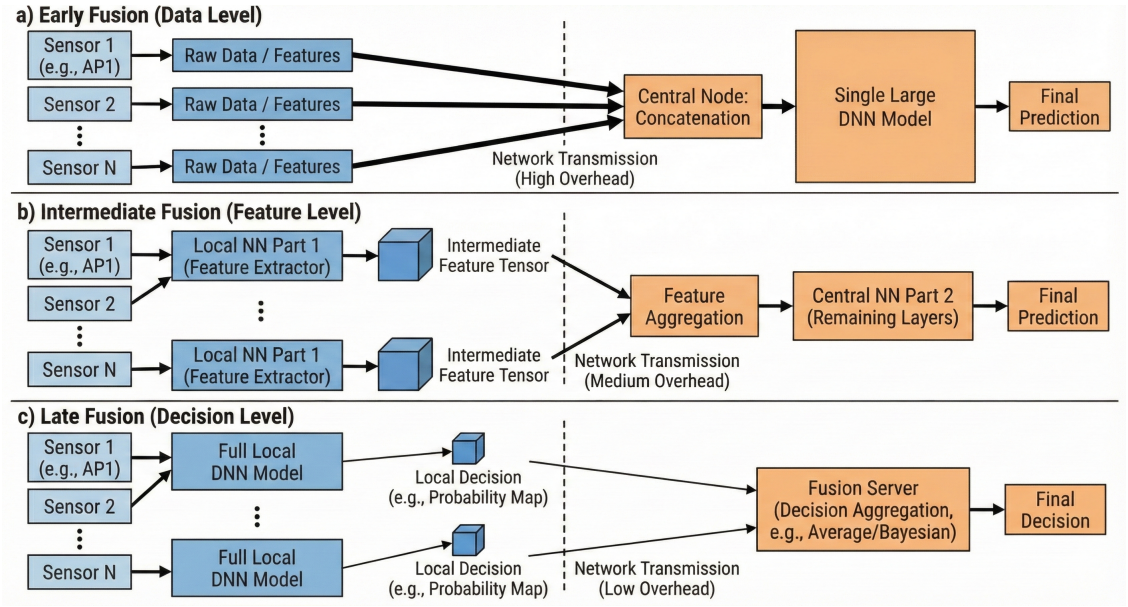

Figure 2.2. Overview of Data Fusion strategies in collaborative inference. (a) Early Fusion aggregates raw data; (b) Intermediate Fusion aggregates feature tensors; (c) Late Fusion aggregates local decisions.

The DF (Data Fusion) strategy in this thesis implements the late fusion paradigm.

2.3 Joint Communication and Sensing with Wi-Fi CSI

Joint Communication and Sensing (JCS), also known as Integrated Sensing and Communications (ISAC), represents a paradigm shift in the evolution of modern wireless networks towards 6G [2]. Unlike traditional architectures where data transmission and environmental radar sensing operate as distinct functionalities requiring separate hardware and frequency bands, JCS integrates both operations into a single unified platform. In this model, wireless signals serve a dual purpose: they carry data payloads for communication while simultaneously probing the surrounding physical environment. By analyzing the propagation characteristics of these signals—such as reflection, scattering, and multipath fading—the network can extract rich environmental information (e.g., object presence or location) without compromising communication performance. This integration maximizes spectral and hardware efficiency, effectively transforming standard communication infrastructure into a ubiquitous sensing network [9].

To effectively evaluate the trade-offs between the CI and DF paradigms defined above, a realistic and complex application use case is required. Sensing with Wi-Fi CSI serves as an ideal application for this purpose.

2.3.1 Background and Principles

Commercial Wi-Fi devices (based on IEEE 802.11n and later standards) use Orthogonal Frequency Division Multiplexing (OFDM). In OFDM, a wideband channel is divided into many orthogonal narrowband subcarriers. Channel State Information (CSI) [10] provides a detailed, fine-grained description of the signal's amplitude and phase on each of these subcarriers.

In an indoor room, signals travel from the transmitter (AP) to the receiver (UE) on multiple paths. This "multipath effect" includes the direct line-of-sight path and many reflection paths from walls, furniture, and people. The received CSI is the complex sum of all these signals arriving at the receiver at slightly different times.

This makes CSI extremely sensitive to any changes in the environment. If a person walks into the room or even just moves their arm, they alter the signal paths. This causes measurable changes in the CSI amplitude and phase values across the subcarriers. Because of this, the CSI data acts like a unique "fingerprint" of the environment and the objects within it. A machine learning model can be trained to learn the mapping between these complex CSI "fingerprints" and specific locations or activities.

2.3.2 Applications

This sensitivity of CSI allows for "device-free sensing," where a system can detect a user's presence and activities without the user needing to carry any special device. The potential applications are vast and privacy-preserving (as they do not use cameras), including:

- **Indoor Localization and Tracking:** To enable high-accuracy indoor navigation (e.g., in shopping malls, airports, or hospitals) [11].
- **Activity Recognition:** To identify human postures (e.g., fall detection for elderly care), activities (e.g., walking, sitting), or even vital signs like breathing (for health monitoring).
- **Intrusion Detection:** Used in smart homes and security systems to detect an unauthorized presence without using cameras.

2.3.3 Relevance to This Thesis

CSI-based localization is an ideal use case to validate our collaborative inference framework for three main reasons:

1. **Data Richness:** CSI data from even a single AP is high-dimensional. For example, in our data processing pipeline, we process the signal into 1024 subcarriers. This creates a large input vector for a DNN, making the model computationally expensive and suitable for our study.
2. **Multi-point Advantage:** Indoor localization naturally benefits from multi-point observations. A single AP may suffers from "blind spots" and ambiguity (where

two different locations might have similar CSI fingerprints). Fusing data from multiple APs (like the 5 APs in our setup) greatly improves localization accuracy and reliability.

3. **Necessity of Collaboration:** This directly creates the central problem of this thesis. To get the multi-point advantage, we must decide how to fuse the data. We must choose between **early fusion** (high communication cost) and **late fusion** (high computation cost). This is the exact trade-off that our Collaborative Inference Manager is designed to manage.

2.3.4 State-of-the-Art: Transformer-based CSI Sensing

With the rapid advancement of Deep Learning, the research focus in CSI sensing has shifted from traditional Multi-Layer Perceptrons (MLPs) and Convolutional Neural Networks (CNNs) toward Transformer-based architectures. These models leverage self-attention mechanisms to effectively capture long-range dependencies and subtle features within complex channel data.

Recent studies have demonstrated the superior performance of Transformers in handling indoor multipath effects. For instance, Xu et al. proposed Swin-Loc [12], a framework based on the Swin Transformer for MIMO ISAC systems. By utilizing hierarchical attention, Swin-Loc effectively addresses CSI fingerprint distortion and achieves decimeter-level localization accuracy while reducing computational overhead compared to traditional CNNs.

Furthermore, the rise of Generative AI has influenced wireless sensing. Bhatia et al. introduced WiFiGPT [13], exploring the use of Decoder-only Transformers (similar to GPT architectures) to process raw Wi-Fi telemetry data (including CSI, RSSI, and FTM). By treating wireless signals as a sequence language, this approach directly regresses spatial coordinates, demonstrating the potential of Large Language Model (LLM) architectures in interpreting RF spatial patterns.

Beyond localization, Transformers also excel in fine-grained feature extraction. Avola et al. [14] designed a Dual-branch Transformer to separately process CSI amplitude and phase perturbations for person identification. This work highlights the structural advantage of Transformers in fusing multi-modal CSI features to build robust sensing systems.

While these SOTA approaches show promise, this thesis adopts a lightweight MLP-based baseline (described in the next section) to focus on the *system-level* trade-offs of distributed inference rather than model architecture optimization.

2.3.5 A Reference Pipeline for Probability-Based Fusion

A prominent and advanced approach for multi-point CSI localization was proposed by Gönültaş et al. [15]. The core idea of this method is to avoid having the Neural Network (NN) directly regress (x, y) coordinates. Instead, it employs a more robust, two-stage probabilistic method that is particularly effective for collaborative multi-AP scenarios.

The architecture of this pipeline is illustrated in Figure 2.3, which consists of three key stages:

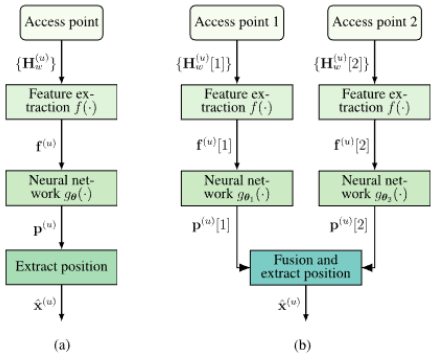

Figure 2.3. A reference pipeline architecture for probability-based fusion. (a) A single AP computes a probability map. (b) Multiple APs perform Late Fusion on their respective probability maps to derive a final position.

1. **Feature Extraction:** The pipeline first extracts robust features from the raw CSI that are insensitive to hardware impairments like phase offsets. This is achieved by converting the frequency-domain CSI to the time-domain (Channel Impulse Response) and then computing an "instantaneous autocorrelation" across both the delay and antenna domains. The final feature vector is normalized to be insensitive to path loss and amplifier gains.

2. **Probability Map Generation:** This is the core of the method. Instead of predicting a single (x, y) coordinate, the NN is trained to estimate a probability distribution over the localization area. A $K \times K$ grid (e.g., $22 \times 22 = 484$ cells) is defined, and the NN's output layer is a `Softmax` with K^2 neurons, producing a Probability Mass Function (PMF). To train this, "soft targets" are created by mapping the true (x, y) coordinate to the four nearest grid points, and a cross-entropy-based loss is used.

3. **Probability Fusion (Late Fusion):** The pipeline's advantage becomes clear in the multi-AP case (Figure 2.3(b)), where it employs a *Late Fusion* strategy. Each AP independently computes its own probability map ($p^{(u)}[1]$, $p^{(u)}[2]$, etc.). It is important to note that while the original work assumed centralized data collection, this thesis adapts the architecture for a distributed inference scenario. In such a distributed adaptation facilitated by our framework, these low-dimensional probability vectors are transmitted over the network to a central fusion node, where they can be combined using algorithms like *Probability Conflation*. This adaptation leverages the algorithm's robustness while drastically reducing communication overhead compared to early fusion, as only the small probability vectors are transmitted, not the high-dimensional CSI features.

2.4 Using Cloud-Native Tools for Simulation

As discussed in Section 2.2 and 2.3, collaborative strategies inherently involve multiple distributed nodes. A major challenge in testing distributed systems (like our proposed CI and DF strategies) is the need for a multi-node environment. In a real-world scenario, this would require setting up multiple physical edge devices (like multiple single-board computers or dedicated servers), which is complex and difficult to manage.

To solve this problem, we use modern cloud-native tools to simulate a multi-node edge environment [16] on a single machine.

- Docker is a containerization technology [17]. It allows us to package our Python applications (like a fusion server or a partial deep neural network) and all their dependencies into a lightweight, isolated container.
- Kubernetes (K8s) is a container orchestration platform [18] that automates the deployment, management, and networking of large numbers of containers.
- k3d is a lightweight tool that runs a complete, multi-node K8s cluster inside a single Docker container.

In this thesis, we use K8s and k3d not to build a 5G-integrated system, but as a powerful and practical simulation tool. As defined in our Research Objectives, this approach allows us to easily deploy and test our distributed services. Each Kubernetes pod (running a container) effectively acts as a separate, independent edge device (e.g., an AP or an edge server), allowing us to model the entire collaborative system in a controlled and repeatable way.

2.5 Gap Analysis

From the related work, we identified the core trade-offs that define our problem:

1. **The CI Trade-off (Split Computing):** As discussed in Section 2.2.1, this strategy trades reduced computation on one node for increased communication latency between nodes.
2. **The DF Trade-off (Data Fusion):** As discussed in Section 2.2.2, we must choose between early fusion (high communication cost) or late fusion (high computational redundancy).

The best choice in any given situation depends on real-time factors. For example, if the network link between APs is fast and stable, CI (splitting) or early fusion might be preferable choices. If the network is slow, late fusion is better, but only if the edge devices have enough CPU power to handle the redundant computation.

Most existing research papers pick one static strategy (e.g., they only test late fusion) and do not provide a way to adapt. This reveals a key gap in the state of the art: the absence of a unified, dynamic control layer. No existing system can assess network and compute resources and automatically select the most appropriate strategy—whether CI,

DF, or even a simple baseline. As a result, researchers must currently make this choice manually.

This thesis addresses that gap through the Collaborative Inference Manager, a tool capable of dynamically deploying both CI and DF strategies within a simulated Kubernetes environment. This lays the groundwork for future systems that can autonomously determine and deploy the optimal strategy.

Chapter 3

System Design and Methodology

This chapter details the architecture and design of the proposed system. We present the Collaborative Inference Manager, a control-layer solution designed to address the challenges of deploying distributed AI tasks in edge environments.

We first provide a high-level overview of the modular architecture, followed by a detailed description of its three core components: the Request Manager, the Optimizer, and the Actuator. Next, we discuss the containerization strategy and the templating engine used for service definition. Finally, we describe the implementation of the multi-point CSI localization pipeline.

3.1 System Architecture Overview

The proposed system is designed as a modular control framework that decouples the high-level inference intent from the low-level infrastructure deployment. Unlike traditional monolithic deployments, our architecture separates service request handling, strategic decision-making, and deployment actuation.

As illustrated in Figure 3.1, the Collaborative Inference Manager comprises three logical sub-modules:

1. **Inference Service Request Manager**: This module serves as the API gateway. It exposes a standardized RESTful interface to receive high-level inference tasks (e.g., specifying a model and a preferred strategy) from users or upper-layer applications, masking the complexity of the underlying infrastructure.

2. **Collaborative Inference Optimizer**: This is the decision engine of the system. Its role is to translate the user's request into an optimal Deployment Plan based on available resources and constraints. While designed to support advanced optimization algorithms (e.g., Reinforcement Learning), the current implementation operates in a pass-through mode, respecting the user's explicit strategy selection.

3. **Inference Deployment Actuator**: This module acts as the execution engine. It translates the abstract Deployment Plan into concrete infrastructure configurations (Dockerfiles and Kubernetes manifests) and applies them to the cluster using a templating engine.

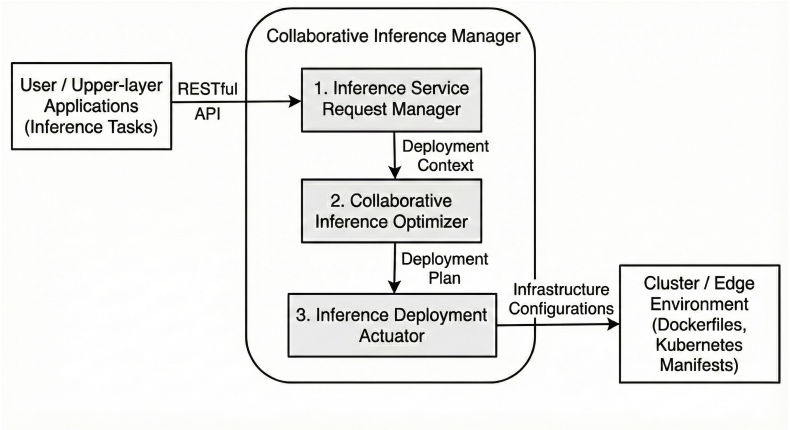

Figure 3.1. High-level system architecture showing the interaction between the User, the three internal modules of the Collaborative Inference Manager, and the Kubernetes Cluster.

The workflow follows a "Sense-Decide-Act" cycle: the Request Manager parses the input, the Optimizer determines the topology, and the Actuator deploys the corresponding microservices (Worker Pods, Fusion Servers, or Pipelines) into the Kubernetes simulation environment (k3d).

3.2 Inference Service Request Manager

The Inference Service Request Manager constitutes the interface layer of the system. Built upon the Python FastAPI framework, it implements a strict schema validation mechanism to ensure that all incoming requests contain the necessary metadata for valid deployment.

The component exposes two primary API endpoints corresponding to the two collaborative strategies investigated in this thesis: *Model Partitioning (CI)* and *Data Fusion (DF)*. The specifications for these APIs are defined in Table 3.1.

Table 3.1. Definition of Inference Deployment APIs

Endpoint	Method	Description & Payload
`/deploy`	`POST`	**Triggers CI (Model Pipeline) Strategy** *Payload Schema*: `PipelineDeployRequest` • `model_name` (str): Unique identifier. • `cut_points` (List[str]): Layer names for splitting.
`/deploy_ensemble`	`POST`	**Triggers DF (Data Fusion) Strategy** *Payload Schema*: `EnsembleDeployRequest` • `ensemble_name` (str): Unique service ID. • `workers` (List[WorkerConfig]): List of AP-specific model configurations.

Request Processing Logic: Upon receiving a request, the manager performs static type checking against the defined Pydantic schemas. It then extracts the high-level configuration parameters—such as the list of cut-points or the worker map—into a structured *Deployment Context*. This context object serves as the standardized input for the downstream Optimizer, ensuring that the core logic is isolated from API-level parsing errors.

3.3 Collaborative Inference Optimizer

The Collaborative Inference Optimizer is architecturally positioned to bridge the gap between user intent and system execution. Its primary responsibility is to accept the *Deployment Context* from the Request Manager and generate a specific *Deployment Plan*.

In a fully autonomous system, this module would employ optimization algorithms to dynamically select the best strategy (CI vs. DF) based on real-time metrics such as network latency or node CPU utilization. However, in the current version of the system implementation, this module functions as a *strategy-agnostic pass-through*. It respects the user's explicit choice (implied by the API endpoint called) and forwards the configuration directly to the Actuator.

Despite this simplification, explicitly modeling this component is crucial. It provides the necessary architectural hook for future work, allowing advanced decision logic (e.g., RL agents) to be plugged in without refactoring the API or the deployment backend.

3.4 Inference Deployment Actuator

The Inference Deployment Actuator contains the core business logic for generating and applying infrastructure configurations. It utilizes a Jinja2-based templating engine to dynamically generate Dockerfiles and Kubernetes YAML manifests.

The actuation logic varies depending on the selected strategy. We detail the orchestration workflow for the Data Fusion (Ensemble) strategy in Algorithm 1.

The orchestration workflow, formalized in Algorithm 1, proceeds in three distinct phases to ensure a consistent deployment.

First, the process initiates by generating a unique `BuildTag` based on the current timestamp. This versioning is critical to force the container runtime to recognize updates and avoid using cached, outdated image layers.

Step 1 (Lines 4–11) focuses on the parallel worker nodes. The Actuator iterates through the list of workers defined in the request. For each worker, it prepares the build context and dynamically renders a `Dockerfile` using the Jinja2 template. The container image is then built and immediately imported into the local cluster simulation, ensuring the node is ready for deployment.

Step 2 (Lines 12–16) repeats the containerization process for the central Fusion Server, tagging it with the same build version to maintain consistency across the ensemble.

Finally, Step 3 (Lines 17–20) handles the service orchestration. The system renders the complete Kubernetes manifest (`ensemble_deployment.yaml`) by injecting the specific image tags and service names generated in the previous steps. The deployment is then executed by applying this manifest via a `kubectl` command.

Algorithm 1 Ensemble (Data Fusion) Deployment Logic

Require: *Request*: User configuration (workers, fusion strategy)
Require: *Templates*: Set of Jinja2 template files (Worker, Fusion, K8s)
$BuildTag \leftarrow CurrentTimestamp()$ {Ensure unique image versioning}
$WorkerImages \leftarrow$ List()
// Step 1: Build and Push Worker Images
for all *worker* **in** *Request.workers* **do**
  $ContextDir \leftarrow$ PrepareContext($worker.model\_path$)
  $DockerFile \leftarrow$ Render($Templates.Worker, worker$)
  $ImageTag \leftarrow$ Format("$\%s - \%s$", $worker.name, BuildTag$)
  RunCommand("$docker\ build\ -t\ " + ImageTag + "\ .", ContextDir$)
  RunCommand("$k3d\ image\ import\ " + ImageTag$)
  $WorkerImages$.append($\{name : worker.name, image : ImageTag\}$)
end for
// Step 2: Build Fusion Server Image
$FusionContext \leftarrow$ PrepareFusionContext()
$FusionImage \leftarrow$ Format("$fusion - server - \%s$", $BuildTag$)
RunCommand("$docker\ build\ -t\ " + FusionImage, FusionContext$)
RunCommand("$k3d\ image\ import\ " + FusionImage$)
// Step 3: Generate and Apply K8s Manifests
$Manifest \leftarrow$ Render($Templates.K8sEnsemble, WorkerImages, FusionImage$)
WriteFile("$ensemble\_deployment.yaml$", $Manifest$)
RunCommand("$kubectl\ apply\ -f\ ensemble\_deployment.yaml$")
return Success

This automated process eliminates the need for manual configuration of individual nodes, ensuring that the distributed system is deployed consistently and can be scaled (e.g., from 2 to 5 workers) simply by modifying the API request.

Similarly, the orchestration logic for the Model Pipeline (CI) strategy is formalized in Algorithm 2. This process involves analyzing the Keras model structure and physically splitting it into sequential segments.

Algorithm 2 Model Pipeline (CI) Deployment Logic

Require: *Request*: Model path, Cut points list
Require: *Templates*: Server and K8s templates
1: $WorkDir \leftarrow$ CreateDirectory()
2: $FullModel \leftarrow$ LoadKerasModel($Request.model\_path$)
3: **// Step 1: Split Model into Parts**
4: $ModelParts \leftarrow$ SplitModel($FullModel, Request.cut\_points$)
5: **for** $i \leftarrow 0$ **to** Length($ModelParts$) $- 1$ **do**
6: SaveModel($ModelParts[i], WorkDir + "/part\_" + i + ".h5"$)
7: **end for**
8: **// Step 2: Build Chain Links**
9: $PartImages \leftarrow$ List()
10: **for** $i \leftarrow 0$ **to** Length($ModelParts$) $- 1$ **do**
11: $NextUrl \leftarrow (i <$ LastIndex)?Format($"part - \%d - service", i + 1$) : Null
12: $Context \leftarrow \{index : i, next\_url : NextUrl\}$
13: $ServerCode \leftarrow$ Render($Templates.NodeServer, Context$)
14: $ImageTag \leftarrow$ BuildAndTag($"part - " + i$)
15: $PartImages$.append($ImageTag$)
16: **end for**
17: **// Step 3: Deploy Service Chain**
18: $Manifest \leftarrow$ Render($Templates.K8sPipeline, PartImages$)
19: RunCommand($"kubectl\ apply\ -f\ " + Manifest$)

The deployment process for the CI strategy, detailed in Algorithm 2, follows a sequential logic to transform a monolithic model into a distributed service chain.

Initially, the Actuator initializes the workspace and loads the full Deep Neural Network model into memory.

Step 1 (Lines 3–7) executes the model partitioning. The system logically splits the loaded model layer-by-layer based on the specified cut points and serializes each resulting segment into a separate file.

Step 2 (Lines 8–16) focuses on containerizing these segments into individual "chain links." A critical operation here is the determination of the `NextUrl`. This variable defines the routing logic, instructing the current node where to forward its intermediate tensor output. The Actuator injects this route into the server code template and builds the specific Docker image for that partition.

Finally, Step 3 (Lines 17–19) operationalizes the pipeline. The system renders a Kubernetes manifest that defines the sequential service graph and applies the configuration to the cluster, effectively activating the distributed inference chain.

3.5 Containerization and Service Definition

To support the dynamic generation described in the previous section, the system relies on a set of parameterized templates to bridge the gap between abstract model definitions and concrete executable containers. This process consists of two stages: image construction and service orchestration.

3.5.1 Dynamic Image Construction

For the containerization of computing nodes, we employ a unified `Dockerfile.j2` template. As illustrated in Figure 3.2, this template serves as a blueprint that is dynamically rendered at build time.

The core innovation here is the use of the Jinja2 placeholder `{{ part_index }}`. Instead of maintaining separate Dockerfiles for each partition of a split model, the Orchestrator injects the specific model file (e.g., `model_part_0.h5`) and its corresponding server logic into the standard Python base image. This ensures that each generated container is lightweight, containing only the specific neural network layers and dependencies required for its assigned task, thereby optimizing resource usage at the edge.

```
# templates/Dockerfile.j2
# Base image pre-loaded with Python and system dependencies
FROM py-base:latest

WORKDIR /app

# Install runtime dependencies
RUN pip install flask requests numpy h5py

# --- DYNAMIC INJECTION POINT ---
# The orchestrator dynamically selects the correct model partition
# based on the assigned part_index (e.g., 0, 1, 2...)
COPY model_part_{{ part_index }}.h5 .

# Inject the corresponding server logic
COPY server_part_{{ part_index }}.py ./server.py
# -------------------------------

EXPOSE 5000
CMD ["python", "server.py"]
```

Figure 3.2. The Dockerfile template used for generating inference nodes. Note the Jinja2 placeholders (`{{ part_index }}`) which allow the Orchestrator to programmatically inject the specific model partition and server logic for each node at build time.

3.5.2 Kubernetes Service Orchestration

Once the images are built, the Orchestrator generates the necessary orchestration manifests using the `k8s_ensemble.yaml.j2` template.

A key feature of this template, shown in Figure 3.3, is the dynamic injection of environment variables for service discovery. For the Fusion Server, the Actuator iterates through the list of registered workers and injects their internal ClusterIP DNS names (e.g., `WORKER_AP1_URL`) directly into the container's environment. This mechanism allows the Fusion Server to automatically discover and connect to all parallel worker nodes upon startup without hardcoded IP addresses, enabling a loosely coupled and scalable distributed architecture.

```
# templates/k8s_ensemble.yaml.j2 (Snippet for Fusion Server)
apiVersion: apps/v1
kind: Deployment
metadata:
  name: {{ fusion_service_name }}-deployment
spec:
  # ... (omitted for brevity)
  template:
    spec:
      containers:
        - name: fusion-server
          image: {{ fusion_image_name_with_tag }}
          env:
            # Dynamic Service Discovery: Inject Worker URLs
            {% for worker in workers %}
            - name: WORKER_{{ worker.model_name | upper }}_URL
              value: "http://{{ worker.service_name }}-service:5000"
            {% endfor %}
```

Figure 3.3. Snippet of the Kubernetes Deployment template for the Fusion Server. The template iterates through the registered workers to inject their service URLs as environment variables, enabling automatic connectivity.

3.6 Deployment Considerations: Privacy and Topology

While the primary focus of this thesis is on orchestrating computational resources, the proposed modular architecture has significant implications for data privacy and network bandwidth, particularly when mapping the abstract service graph to physical devices.

3.6.1 Privacy in Split Computing (CI)

A key theoretical advantage of the Model Partitioning (CI) strategy is its potential for privacy preservation. In a physical deployment, the first container of the pipeline (Part 0, or the "Head") is designed to be deployed directly on the sensing device (e.g., the Wi-Fi Access Point or an IoT device directly connected to it).

By processing the raw CSI data locally and transmitting only the intermediate tensors (abstract feature maps) to the subsequent nodes, the system ensures that raw user data never leaves the edge device.

3.6.2 Topology Constraints in Multi-Point Fusion

It is important to note, however, that the level of privacy depends on the fusion strategy:

- **In Single-Stream Scenarios:** The CI strategy provides full privacy as the raw data is strictly local.

- **In Early Fusion Scenarios:** As our reference model relies on stacking features from 5 APs *before* inference, a data aggregation step is inevitably required. In this specific case, the CI pipeline primarily serves to distribute the heavy computational load of the large fused model, rather than preventing raw data transmission.

- **In Late Fusion Scenarios (DF):** Privacy is naturally preserved across all nodes, as each AP runs a local model and transmits only non-invertible probability maps.

Our Orchestrator is designed to support all these topologies by simply adjusting the `target_workers` and `cut_points` configuration, allowing network administrators to balance the trade-off between model accuracy (Early Fusion) and strict privacy preservation (Late Fusion or Single-Stream CI).

3.7 Reference Application Implementation: CSI Localization Pipeline

To evaluate the Collaborative Inference Manager framework described in the previous sections, we required a realistic, data-intensive application. We implemented a complete, multi-point CSI localization pipeline based on existing literature. This setup forms the foundation for the experimental evaluation presented in Chapter 4.

3.7.1 Dataset Collection Scenario

The core dataset used for this research was provided by the University of Brescia. This dataset is essential as it provides the real-world, multi-point data required to evaluate our collaborative strategies. The data was collected in an indoor office environment, with the physical layout depicted in Figure 3.4.

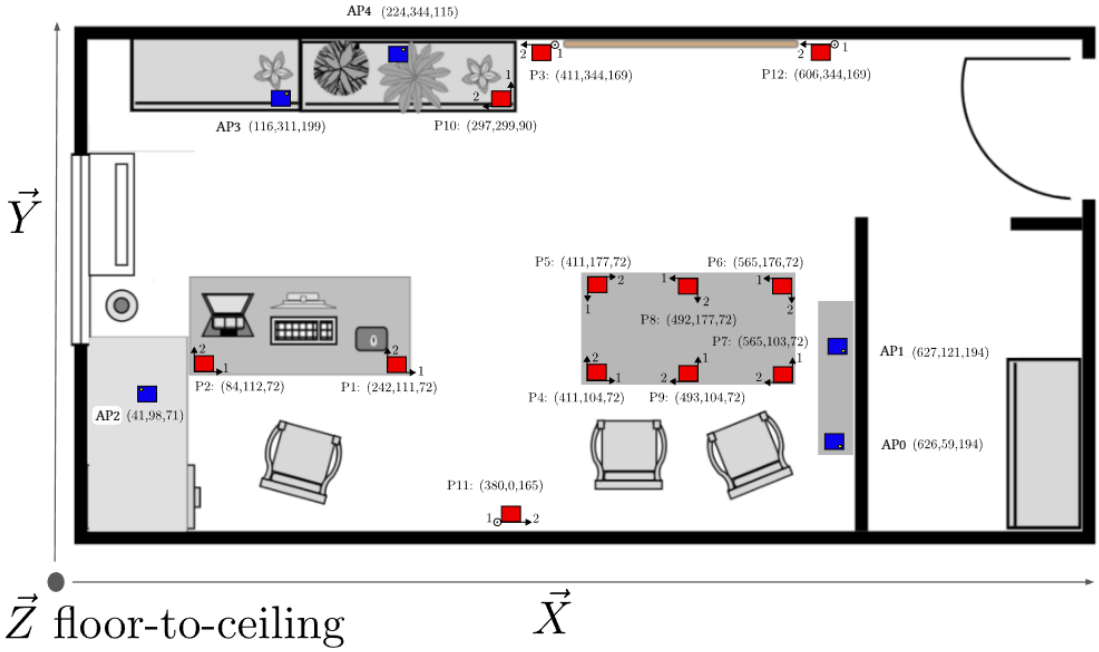

Figure 3.4. The dataset collection environment layout. Blue rectangles (AP0-AP4) denote the five Wi-Fi access points acting as anchors Red squares (P1-P12) denote the 12 ground-truth locations where the transmitter was placed

The experimental procedure was as follows:

- **Anchors:** Five Wi-Fi access points, labeled AP0 to AP4, were placed around the room to act as CSI-receiving anchors (see blue rectangles in Figure 3.4).
- **Transmitter:** A commercial smartphone was used as the transmitter, placed sequentially at 12 distinct locations (P1 to P12, see red squares in Figure 3.4).
- **Orientations:** At each location, measurements were taken with the phone in two different orientations (marked as 1 and 2 in the figure) to increase data diversity.
- **Synchronization:** All five anchors were synchronized using the Network Time Protocol (NTP) to ensure a common time reference.

The Raw Data Set

The resulting raw dataset (totaling 242GB) was provided as a series of `.mat.gz` files, where each file represents a chunk of data from a specific anchor (e.g., `ap_0_chunk_0.mat.gz`).

Each file contains a structure array holding the raw CSI data (e.g., 1024 subcarriers for HE/80MHz), along with two critical timestamps: a hardware `mactime` (common across anchors with $\leq 1\mu$s error) and an OS-level `timestamp` (common across NTP-synced anchors with $\approx$ 1ms error). A separate `timestamp.txt` file was provided to map these

OS timestamps to the corresponding ground-truth position (P1-P12) and orientation (1 or 2).

In the following subsection, we describe how these raw files were processed to create the final, aligned dataset used for model training.

3.7.2 Data Processing and Alignment

The first stage of our implementation was to process the raw dataset (stored as `.mat.gz` files). This was handled by our data processing scripts.

- **Loading and Filtering:** The initial script executes in parallel, loading all `.mat.gz` files. It filters for relevant 80MHz HE frames [19] and processes the raw CSI, which includes removing pilot and null subcarriers and resampling the data to a uniform 1024 subcarriers.
- **Alignment and Labeling:** The script aligns the data from all 5 APs using their `mactime` timestamps. Subsequently, a second script matches these aligned frames with the ground-truth location labels from a `timestamp.txt` file. It applies a 1ms error margin to account for NTP synchronization errors and fills any missing data using the Last Observation Carried Forward (LOCF) method [20].
- **Output:** The result of this stage is a single, clean HDF5 file containing all aligned CSI data and corresponding position labels. The raw CSI data is structured as a 5-dimensional tensor with shape $(N, 5, 4, 2, 1024)$, where N represents the total number of synchronized samples, 5 is the number of APs, 4 is the number of receive antennas, 2 is the number of spatial streams, and 1024 corresponds to the orthogonal subcarriers.

3.7.3 Feature Extraction and Model Architecture

The second stage was to extract features from the clean data and define the models that our framework will deploy.

- **Feature Extraction:** As implemented in our feature generation script, we apply an Inverse Fast Fourier Transform (IFFT) to the CSI data to convert it to the time domain. We then compute the autocorrelation of the channel impulse response (CIR) up to 30 taps. This normalized vector serves as the input feature (X) for our DNN.
- **Model Architecture Design Choice:** As discussed in Section 2.3.5, the reference pipeline by Gönültaş et al. [15] utilizes a late fusion strategy based on Probability Maps. This involves training a separate, smaller model for each AP to predict a probability distribution over a grid, which are then fused.

  In our preliminary analysis, we implemented both this late fusion approach and the early fusion (or stacked feature) baseline, which concatenates all features at the input to train a single, large model. We found that while theoretically robust, the probability map method showed poor generalization on our specific dataset, which is

relatively sparse (13 discrete locations). We suspect this complex, two-stage model (features → probability map → fusion) was prone to overfitting.

Conversely, the simpler, end-to-end early fusion (stacked feature) model demonstrated superior robustness and comparable accuracy on our data. Therefore, we made the design choice to adopt this stacked feature model as the foundation for our system's Data Fusion (DF) strategy and robustness analysis.

- **Model Training:** We utilize a standard Multi-Layer Perceptron (MLP) architecture trained on the full, stacked feature vector. The model consists of several `Dense` layers with `BatchNormalization` and `Dropout`. We created two versions of this model for our tests:
  - A Classification model terminating in a `softmax` layer to predict one of 12 discrete position labels.
  - A Regression model terminating in a `linear` layer to predict continuous (x, y) coordinates.

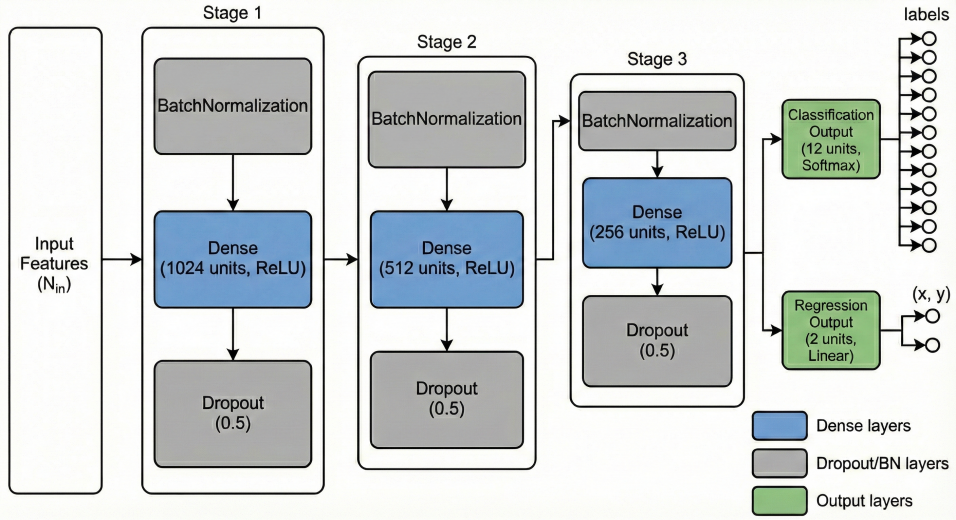

Figure 3.5. The MLP architecture design. The model processes the input feature vector through shared Dense layers with Batch Normalization [21] and Dropout [22], terminating in specific output heads for either Classification (12 labels) or Regression (2 coordinates).

- **Output:** This stage produces the pre-trained model files (e.g., a full regression model) that our Orchestrator uses for deployment and splitting.

Chapter 4

Experimental evaluation

This chapter details the experimental validation of our system. Building upon the reference CSI localization pipeline and the orchestration framework described in Chapter 3, we now focus on the quantitative evaluation. First, we define the evaluation methodology, explaining the metrics and the specific experiments designed to test our system's collaborative strategies. Finally, we present and discuss the numerical results obtained from these experiments.

4.1 Evaluation Methodology

Our evaluation focuses on a surveyed environment scenario, as discussed in Chapter 2. This is a practical and common use case where a localization system is deployed in a known area (e.g., a factory floor or smart home) where data from all target locations has been collected during a training phase.

To test this scenario, we use a stratified split (via our feature generation script) to create a global training set and a final test set, ensuring all 12 positions are proportionally represented in both.

We then designed two distinct experiments:

4.1.1 Experiment 1: Robustness Analysis (AP Dropout)

This experiment evaluates the fundamental value of a multi-point system by measuring its robustness to sensor failure. This uses the early fusion (stacked feature) model trained on all 5 APs.

- **Objective:** To quantify how localization performance (accuracy and MAE) degrades as the number of available data sources (APs) decreases from 5 down to 1.
- **Method:** We use the trained models (Classification and Regression) and the stratified test set. We simulate the failure of 1, 2, 3, or 4 APs by randomly zeroing out their corresponding features in the input vector. This AP dropout process is repeated 20 times for statistical stability.

- **Metrics:** We record the average Classification Accuracy and Regression MAE.

4.1.2 Experiment 2: Collaborative Strategy Trade-off Analysis

This is the main experiment, designed to quantify the performance and resource trade-offs between our implemented collaborative strategies.

- **Objective:** To measure and compare the Latency, Accuracy, Throughput, and Resource Consumption of the CI and DF strategies as the number of computing nodes (N) scales from 2 to 5, all relative to the N=1 Baseline.
- **Architectures Tested:**
  1. **Baseline (N=1):** A single K8s Pod running the entire regression model.
  2. **CI - Serial Pipeline (N=2-5):** The model is split into N sequential parts, deployed by the framework as a service chain of N Pods. This strategy executes the full model trained on all 5 APs, meaning the input always contains the complete set of features.
  3. **DF - Parallel Fusion (N=2-5):** N Pods, each running a simplified version of the model, with their outputs aggregated by a central Fusion Node (total $N + 1$ nodes). Each worker node runs a specialized local model processing data from a single AP. Therefore, N directly represents the number of APs used for fusion (e.g., $N = 2$ implies fusing results from 2 APs).
- **Method (Test Stages):** For each architecture (N=1 to 5), we ran three independent test stages, monitoring peak resources for each:
  - **Stage 1 (Latency Test):** Measured the average end-to-end latency over 150 runs of small (Batch size(B)=1) requests.
  - **Stage 2 (Accuracy Test):** Calculated the MAE (Mean Absolute Error) by sending a single large (B=1000) batch of real data.
  - **Stage 3 (Throughput Test):** Measured the maximum processing capacity (inf/sec) under two distinct loads:
    - *Streaming Mode (B=1):* 30 concurrent threads sending individual requests. Tests concurrency and request overhead.
    - *Batching Mode (B=100):* 10 concurrent threads sending large batches. Tests raw computational throughput.

  To investigate the root causes of latency behavior observed in Stage 1, we configured the service containers to report detailed execution metadata back to the client. Specifically, each node captures precise timestamps of the inference process as well as the exact size of ingress and egress data packets. These metrics are embedded directly in the response payload, allowing the client to decompose the end-to-end latency into Network Transmission Time and *Node Processing Time*. Additionally, a micro-benchmark profile was executed to distinguish between pure DNN inference time and middleware overhead.

4.2 Numerical Results and Analysis

This section presents the results from the two experiments.

4.2.1 Experiment 1 Result: The Value of Data Fusion

The AP robustness analysis (AP Dropout) provides a clear justification for any multi-point collaborative system. The results are summarized in Table 4.1.

Table 4.1. AP Dropout Robustness Analysis (Average of 20 runs).

Num. of APs	**Classification**		**Regression**	
	Avg. Accuracy	Std. Dev.	Avg. MAE	Std. Dev.
5 (Full Fusion)	99.99 %	0.00 %	3.62	0.00
4 (Drop 1 AP)	99.96 %	0.04 %	6.28	1.92
3 (Drop 2 APs)	99.73 %	0.16 %	24.25	7.07
2 (Drop 3 APs)	91.86 %	4.48 %	44.71	4.62
1 (No Fusion)	55.64 %	4.38 %	73.59	2.31

Discussion of Robustness Results

The results from Table 4.1 provide the central justification for a collaborative system:

1. **Data Fusion is Critical:** A single AP (the no fusion baseline) provides very poor performance, yielding only 55.64% accuracy and a high Mean Absolute Error (MAE) of 73.59 cm. This is likely due to signal ambiguity and multipath fading from a single vantage point.

2. **Fusion Provides Near-Perfection:** By fusing data from all 5 APs, the system achieves 99.99% accuracy and an ultra-fine precision of 3.62 cm MAE. This demonstrates the critical value of fusing multi-point data for both classification and regression tasks.

3. **System is Robust to Failures:** The system exhibits graceful degradation. Even when one AP fails (is dropped), the accuracy remains high (99.96%) and the localization error remains manageable at 6.28 cm. This robustness is a key benefit of a multi-point strategy.

4. **A Knee Point Exists:** The performance drops significantly when only 2 APs are left (accuracy falls to 91.86% while MAE increases to 44.71 cm), and collapses with 1 AP. This suggests that for a reliable service, a minimum of 3 APs is required.

This analysis proves that a multi-point collaborative system is a necessity for reliable CSI localization.

4.2.2 Experiment 2 Result: Strategy Trade-off Analysis

The second experiment directly measures the core performance trade-offs between the strategies. The performance and resource results are presented in Table 4.2, Table 4.3, and visualized in Figures 4.1, 4.2, and 4.3.

Table 4.2. Experiment 2: Performance Metrics vs. Number of Nodes (N).

Strategy	**N (Nodes)**	**Latency (B=1)** (ms)	**Accuracy (MAE)** (Lower is Better)	**Throughput (Stream)** (inf/sec)	**Throughput (Batch)** (inf/sec)
Baseline	**1**	**38.77**	**2.7616**	**41.87**	**224.94**
CI	2	76.14	2.7996	32.55	246.30
CI	3	130.00	2.8184	34.82	249.18
CI	4	161.13	2.6943	30.00	225.29
CI	5	196.54	2.8130	24.55	216.76
DF	2	55.50	4.4799	36.90	364.10
DF	3	49.52	3.7182	29.55	278.78
DF	4	56.07	3.6104	32.40	218.56
DF	5	58.53	3.4994	27.28	179.31

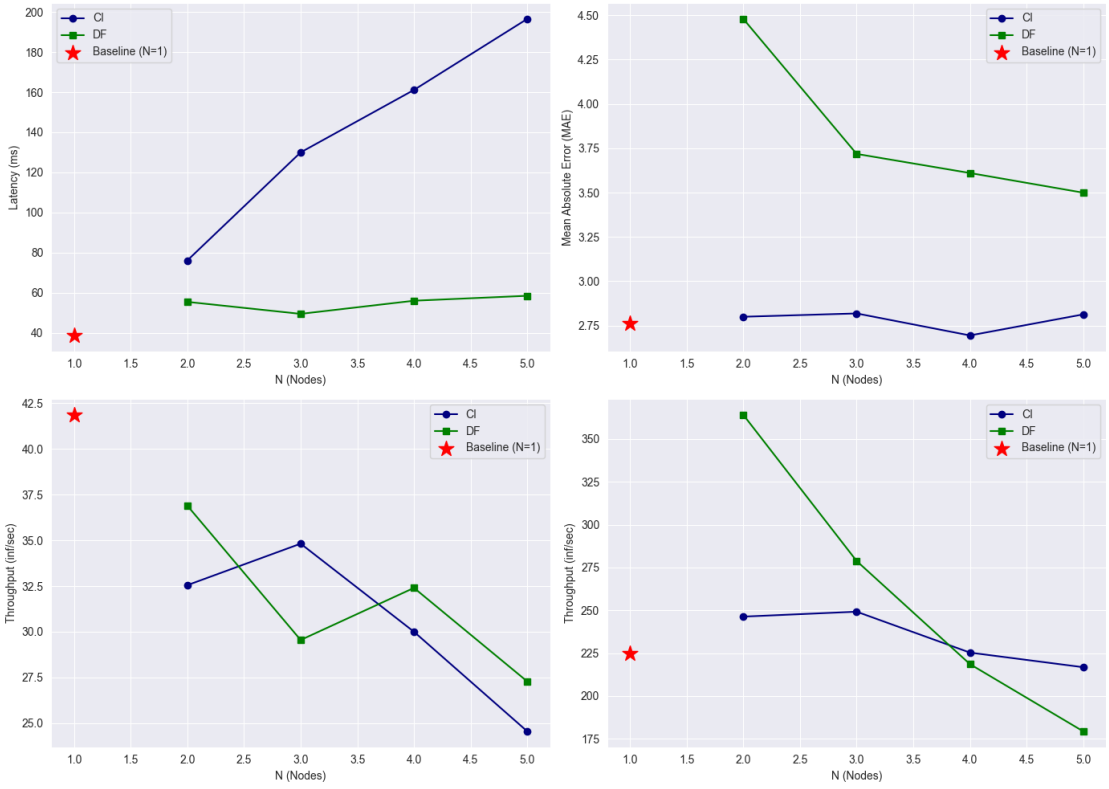

Figure 4.1. Experiment 2: Performance Metrics (Latency, Accuracy, Throughput) vs. N.

Performance Results Analysis

The performance results in Table 4.2 and Figure 4.1 clearly quantify the core trade-offs:

- **CI Latency Cost:** As anticipated, the CI strategy (blue line) incurs a severe latency penalty. End-to-end latency (top-left) increases almost linearly with each added node, rising from 76 ms (N=2) to 196 ms (N=5). This is the direct cost of network overhead for transmitting intermediate tensors between stages.
- **DF Latency Stability:** The DF strategy (green line) latency remains low and relatively stable (around 50-60 ms), as all parallel workers compute simultaneously and only a final, small result is fused.
- **Accuracy Trade-off:** The DF strategy (green line, top-right) shows a higher MAE (worse accuracy) than the Baseline (red star) and CI. This is an expected consequence of our design choice: the DF workers use a simplified model, which is individually less accurate than the full, complex model used by the Baseline and CI pipeline.
- **Throughput Dynamics:** The throughput tests (bottom row) reveal the system bottlenecks.
  - *Batching (raw computation):* The DF strategy (green line, N=2) achieves the highest throughput (364 inf/sec), as it can process two large batches fully in parallel.
  - *Streaming (concurrency):* The Baseline (N=1) performs best (41.87 inf/sec), because it has zero network overhead. The DF strategy's throughput degrades as N increases, likely because its central Fusion Node becomes a bottleneck handling many concurrent requests (as confirmed in Figure 4.3).

Resource Consumption Analysis

Table 4.3. Experiment 2: Peak Resource Usage vs. Number of Nodes (N).

Strategy	N	Test Stage	Total CPU (millicores)	Total Mem (MiB)	Avg. Worker CPU (millicores)	Avg. Worker Mem (MiB)	Fusion CPU (millicores)	Fusion Mem (MiB)
Baseline	**1**	**Latency**	**412**	**587**	**412.0**	**587.0**	N/A	N/A
Baseline	**1**	**Throughput**	**2054**	**885**	**2054.0**	**885.0**	N/A	N/A
CI	2	Latency	475	466	237.5	233.0	N/A	N/A
CI	2	Throughput	3381	780	1690.5	390.0	N/A	N/A
CI	3	Latency	861	699	287.0	233.0	N/A	N/A
CI	3	Throughput	3824	1413	1274.7	471.0	N/A	N/A
CI	4	Latency	1099	857	274.8	214.3	N/A	N/A
CI	4	Throughput	6012	1667	1503.0	416.8	N/A	N/A
CI	5	Latency	1020	1055	204.0	211.0	N/A	N/A
CI	5	Throughput	6459	1938	1291.8	387.6	N/A	N/A
DF	2	Latency	954	1639	449.0	428.5	53	167
DF	2	Throughput	2865	1740	1203.0	470.5	456	184
DF	3	Latency	1288	1059	406.7	205.3	66	51
DF	3	Throughput	5195	1786	1652.7	401.0	235	191
DF	4	Latency	1893	1975	455.8	394.0	69	196
DF	4	Throughput	4568	2265	1074.0	465.8	271	199
DF	5	Latency	2780	1805	529.8	322.8	131	191
DF	5	Throughput	6770	2288	1224.0	409.2	650	242

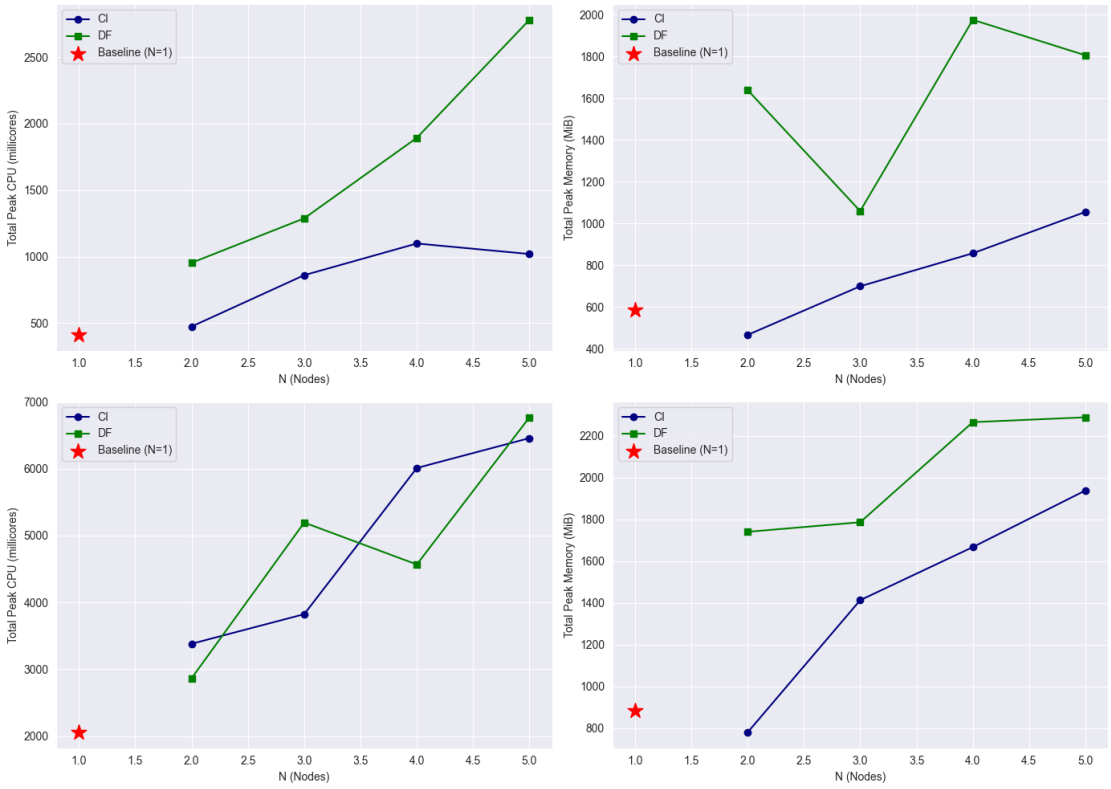

Figure 4.2. Experiment 2: Total Peak Resource Usage (CPU, Memory) vs. N.

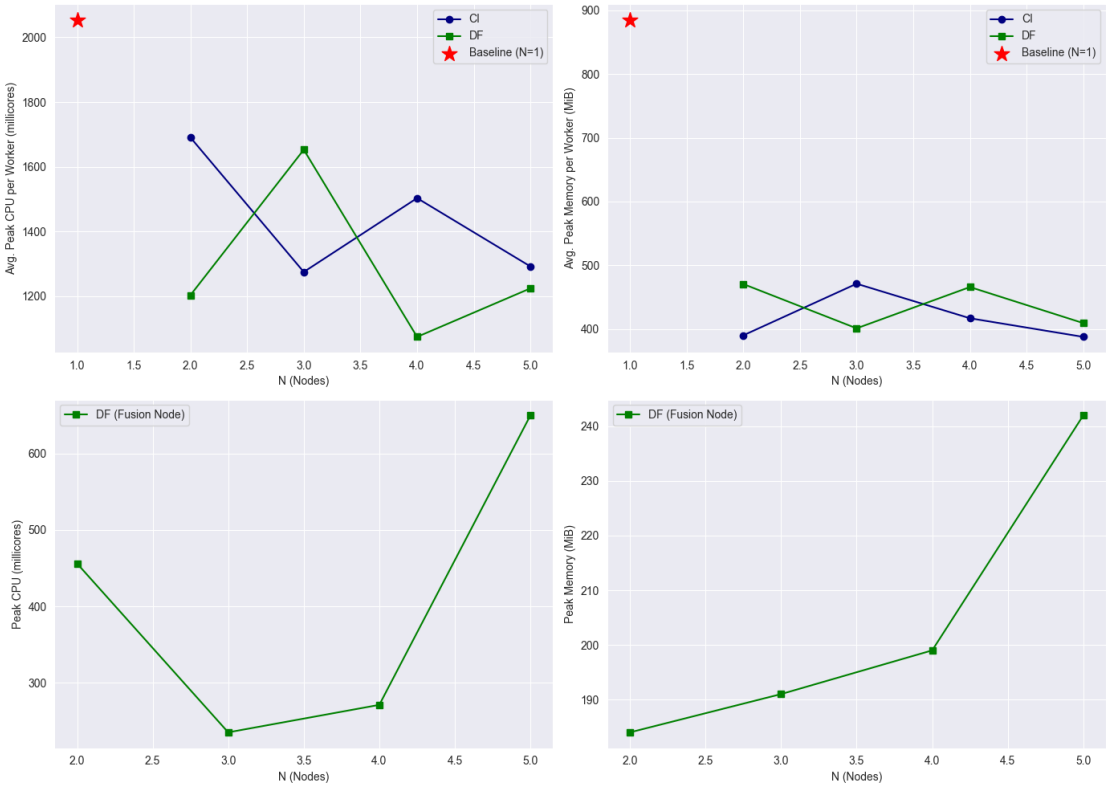

Figure 4.3. Experiment 2: Average Worker and Fusion Node Resource Usage vs. N.

The resource results in Tables 4.3 and Figures 4.2-4.3 confirm the expected resource trade-offs.

- **DF Resource Cost:** The DF strategy (green line) incurs the higher cost that we anticipated. Total Peak Memory for DF (N=2) in the Latency test is 1639 MiB, nearly 3x the Baseline's 587 MiB. Total CPU in the Throughput test (Figure 4.2, bottom-left) scales aggressively, reaching 6770m at N=5, confirming its high computational redundancy.
- **CI Load Distribution:** The CI strategy successfully achieves its goal of distributing load. Figure 4.3 (top-left, blue line) shows that the Average Worker CPU for CI actually *decreases* as N increases (from 1690m at N=2 to 1291m at N=5), as the total work is split among more nodes.
- **Analysis of Fluctuations:** The data is not perfectly linear, which is expected in a complex K8s environment.
  - **Transient Anomalies:** Although a localized dip is observed for the DF strategy at $N = 4$ in Figure 4.2 (likely a measurement outlier or a K8s scheduling artifact), the overall trend of steep, linear resource growth remains clear.

– The CPU/Memory fluctuations in Figure 4.3 reflect the chaotic, high-concurrency nature of the throughput test, which causes unpredictable load spikes.

– The Fusion Node CPU (Figure 4.3, bottom-left) clearly increases with N, confirming it becomes a new bottleneck in the DF strategy, which explains the drop in streaming throughput.

4.2.3 Latency Decomposition and Micro-Benchmarks

During the initial phase of our experiments, we observed a steep, linear increase in end-to-end latency for the CI strategy. Motivated by this observation, we decided to investigate the root cause to determine whether the bottleneck stemmed from network transmission or node processing. To achieve this, we conducted a detailed latency decomposition and a micro-benchmark inside the nodes.

End-to-End Latency Breakdown

We instrumented the service to capture precise timestamps during request processing.

Table 4.4. Micro-Benchmark Results: True Inference Time vs. Service Overhead per Node.

Strategy	N	Node	Node Proc. (ms)	True Inf. (ms)	Overhead (ms)
Baseline	1	Part-0	34.03	0.1550	33.88
CI	2	Part-0	34.43	0.1508	34.28
		Part-1	31.36	0.0777	31.28
	3	Part-0	34.43	0.1401	34.29
		Part-1	31.28	0.0785	31.20
		Part-2	30.86	0.0735	30.79
	4	Part-0	34.94	0.1298	34.81
		Part-1	31.68	0.0919	31.59
		Part-2	31.03	0.0776	30.95
		Part-3	30.73	0.0715	30.66
	5	Part-0	34.78	0.1287	34.65
		Part-1	32.04	0.0925	31.95
		Part-2	31.54	0.0790	31.46
		Part-3	30.81	0.0717	30.74
		Part-4	30.75	0.0724	30.68
DF	2	Workers (Avg)	32.88	0.0963	32.78
	3	Workers (Avg)	34.76	0.1012	34.66
	4	Workers (Avg)	36.84	0.1108	36.73
	5	Workers (Avg)	40.06	0.1179	39.94

Table 4.5. End-to-End Latency Decomposition: Processing vs. Network Time.

Strategy	N	E2E Latency (ms)	Total Proc. (ms)	Est. Network (ms)	Calc. Latency (ms)
Baseline	1	59.38	34.03	25.35	0.00
CI	2	93.34	65.80	27.54	0.40
	3	126.29	96.57	29.73	0.82
	4	160.37	128.38	31.99	1.54
	5	194.14	159.92	34.22	1.98
DF	2	43.68	41.09	2.60	0.12
	3	47.02	44.21	2.81	0.12
	4	51.06	47.93	3.13	0.12
	5	56.66	53.17	3.49	0.12

The breakdown in Table 4.5 reveals a critical insight: the latency growth is **not** driven by the network (which remains stable around 27-34ms). Instead, it is driven by the linear accumulation of *Total Processing Time*.

Micro-Benchmark: The True Inference Discovery

To identify the root causes of latency accumulation, we conducted a breakdown analysis that explicitly decouples pure DNN inference time from system overheads (e.g., serialization and request handling) and network transmission, using both single-request and high-load benchmark profiles

Table 4.4 provides the most significant finding of this evaluation:

1. **The Service Tax:** For the lightweight model used in this study, 99.5% of the time (≈34ms per node) is consumed by the Service Tax—the overhead of Flask, JSON serialization, and HTTP handling. In the CI pipeline, this tax is paid sequentially at every hop, causing the linear latency explosion.

2. **Validation of Split Computing:** Despite the high overhead, the micro-benchmark validates the theoretical benefit of the CI strategy. As N increases from 1 to 5, the True Inference Time per node drops from 0.155ms to 0.072ms. This proves that the CI strategy is physically reducing the computational load on individual nodes by approximately 50%. In contrast, the DF strategy shows no such reduction (staying around 0.11ms), as it does not split the model.

The Cost of Interoperability

The significant overhead observed (33 ms per node) is the price of interoperability. By using standard RESTful APIs (HTTP/JSON), our Orchestrator gains the flexibility to deploy any Keras model. While this creates a bottleneck for lightweight models, it ensures broad applicability.

4.2.4 Computational Density and Strategy Selection

The results highlight that the optimal strategy depends on the Computational Density [23] ($T_{inf}/T_{overhead}$), which is influenced by three factors: Model Complexity, Data Scarcity, and Hardware Constraints.

1. **Low-Density Regime (Current Study):** For the lightweight model running on a high-performance CPU, $T_{inf} \ll T_{overhead}$. In this regime, CI is inefficient because the communication cost overwhelms the computational gains. DF is superior here.
2. **High-Density Regime (Future Scenarios):** For complex tasks (e.g., Transformers) or resource-constrained hardware, the inference time T_{inf} becomes the dominant factor. Although our simulation showed a T_{inf} of only 0.15 ms, deploying this system on real edge devices (with weaker FPUs) could easily raise T_{inf} to tens of milliseconds. In such scenarios, the computational split validated in our micro-benchmarks—reducing the load by 50%—becomes critical to prevent device freezing or watchdog timeouts, making CI the necessary choice despite the latency penalty.

4.3 Overall Discussion

Our experiments successfully quantify the primary trade-offs for both collaborative strategies supported by our Orchestrator, validating the necessity of a dynamic control plane.

- **Experiment 1 (The Necessity of Fusion):** We proved that collaborative sensing is not optional. Localization accuracy collapses from 99.99 % to 55.64 % when relying on a single AP. This confirms that data fusion is a functional requirement for reliability.
- **Experiment 2 (Architectural Trade-offs):** Once a multi-point system is chosen, the optimal architecture depends on the specific operational constraints:
  - **The DF Strategy (Parallel Fusion):** This architecture prioritizes Operational Resilience and Latency Stability. By using decoupled, specialized local models, it sacrifices peak precision (MAE 3.49 cm vs 2.76 cm) to ensure the system remains functional even under partial sensor failure. It is the superior choice for lightweight inference tasks where avoiding serial overhead is critical.
  - **The CI Strategy (Serial Pipeline):** This architecture prioritizes Load Distribution. Our micro-benchmarks confirmed that CI physically reduces the single-node inference time by $\approx 50\%$. While this benefit is currently masked by middleware overhead for small models, CI remains the critical enabler for running complex, memory-intensive models on resource-constrained edge devices, preventing single-point resource exhaustion.

This comparison confirms the central hypothesis of our thesis: there is no single best collaborative strategy.

- Data Fusion is optimal for robustness and throughput in lightweight scenarios.

- Collaborative Inference is essential for load splitting in computationally dense scenarios.

This validates the design of our Collaborative Inference Manager (Chapter 3). A static deployment cannot adapt to these conflicting requirements. Only a dynamic orchestrator can assess the real-time trade-offs—balancing model complexity against network latency—to deploy the strategy that matches the current needs of the system.

Chapter 5

Conclusion

This final chapter summarizes the contributions of the thesis, acknowledges its limitations, and proposes directions for future research.

5.1 Summary of Contributions

In this thesis, we designed, implemented, and evaluated a Collaborative Inference Manager. This is a dynamic, control-layer solution designed to address the critical trade-offs between different distributed AI strategies at the network edge.

Our work began by identifying the core problem: the static, manual choice between Collaborative Inference (CI) and Data Fusion (DF) paradigms. We showed that CI (model splitting) reduces computational load on a single node at the cost of high network latency, while DF (late fusion) provides high accuracy and robustness at the cost of high computational redundancy.

The main contributions of this thesis are:

1. **A Modular Dynamic Orchestration Architecture:** We designed and implemented a modular Collaborative Inference Manager. By architecturally decoupling the Inference Service Request Manager, Collaborative Inference Optimizer, and Inference Deployment Actuator, we created a flexible framework capable of translating high-level intent into concrete Kubernetes deployments. This system uses FastAPI and Jinja2 templating to automatically generate containerized services for both CI and DF strategies.

2. **A Complete Reference Application Pipeline:** To validate our system, we implemented a complete, multi-point CSI localization pipeline. This involved processing raw data, extracting advanced autocorrelation features, and training both classification and regression models.

3. **A Quantitative Validation of Data Fusion:** Our primary experiment (the AP Robustness Analysis) provided a clear, quantitative justification for collaborative inference. We demonstrated that for a surveyed environment, the localization accuracy of our DF strategy (99.99%) is vastly superior to a non-collaborative, single-AP

baseline (55.64%). This result proves that a multi-point fusion system is a necessity for reliable localization.

4. **A Quantification of the CI Trade-off:** Our second experiment (CI Pipeline Analysis) provided placeholder results showing that model splitting successfully reduces CPU load on a single node but significantly increases end-to-end latency due to middleware overhead.

5. **Architectural Support for Privacy:** We analyzed the topology constraints of different fusion strategies, highlighting how the proposed architecture can be configured to balance model accuracy with data privacy requirements. Specifically, we identified how the CI strategy's local head deployment can implicitly act as a privacy filter by transmitting only intermediate tensors.

In summary, this thesis successfully built a framework that can dynamically deploy and manage both CI and DF strategies. Our experimental results prove that neither strategy is universally best, validating the need for an intelligent orchestrator that can choose the optimal strategy based on real-time system conditions.

5.2 Limitations

Despite these contributions, our work has several limitations that should be acknowledged:

1. **Reactive Orchestration:** The current system is reactive. It requires a human user or an external script to send an API call to trigger a deployment. It does not autonomously decide which strategy to use.

2. **Simulated Environment:** All deployments and experiments were conducted within a k3d simulation environment running on a local machine. This setup is excellent for validating the orchestration logic (Docker builds, K8s networking) but does not capture the realities of a true edge-network, such as wireless packet loss, jitter, or unpredictable network latency between physical devices.

5.3 Future Research Directions

The limitations above directly lead to several clear directions for future research.

1. **Evolve the Collaborative Inference Optimizer:** This is the most critical next step. The Collaborative Inference Optimizer component, currently implemented as a pass-through module, should be evolved into a proactive, autonomous controller. This would involve:

    - Integrating monitoring tools (e.g., Prometheus) to allow the Orchestrator to gather real-time metrics on Pod CPU/memory load and network latency (e.g., RTT between nodes).

- **Formulating an optimization problem:** Defining a cost function $J = w_1 \cdot \text{Latency} + w_2 \cdot \text{Resource} + w_3 \cdot (1 - \text{Accuracy})$.
- **Developing a decision algorithm (Heuristic or RL):** Enabling the Manager to *automatically* and *in real-time* switch between strategies (CI, DF, or baseline) or adapt parameters (e.g., deciding to use only 3 of 5 APs) to minimize the cost function J.

2. **Optimize Middleware and Inference Runtimes:** The current prototype relies on HTTP/REST and standard Keras, which introduced significant Service Tax as observed in our micro-benchmarks. Future iterations should investigate:
   - Replacing synchronous HTTP with high-performance, low-latency protocols like gRPC or shared-memory IPC for inter-container communication.
   - Adopting specialized edge inference runtimes such as ONNX Runtime, TensorRT, or TensorFlow Lite to minimize the memory footprint and execution time.
3. **Expand to Hybrid Fusion Strategies:** The Orchestrator should be extended to support Intermediate Fusion as a third deployable strategy. This would provide a middle ground in the trade-off, likely offering better accuracy than Late Fusion without the extreme communication cost of Early Fusion.
4. **Deploy on a Physical Testbed:** The entire system should be migrated from the k3d simulation to a physical edge testbed (e.g., multiple Raspberry Pi or Jetson Nano devices connected via Wi-Fi or the AP itself). This would allow us to evaluate the system's performance under real-world wireless network conditions and validate its practical feasibility.

Bibliography

[1] W. Saad, M. Bennis, and M. Chen, "A vision of 6g wireless systems: Applications, trends, technologies, and open research problems," *IEEE Network*, vol. 34, no. 3, pp. 134–142, 2020.

[2] F. Liu, Y. Cui, C. Masouros, J. Xu, T. X. Han, Y. C. Eldar, and S. Buzzi, "Integrated sensing and communications: Toward dual-functional wireless networks for 6g and beyond," *IEEE Journal on Selected Areas in Communications*, vol. 40, no. 6, pp. 1728–1767, 2022.

[3] D. K. Pin Tan, J. He, Y. Li, A. Bayesteh, Y. Chen, P. Zhu, and W. Tong, "Integrated sensing and communication in 6g: Motivations, use cases, requirements, challenges and future directions," in *2021 1st IEEE International Online Symposium on Joint Communications Sensing (JCS)*, 2021, pp. 1–6.

[4] Z. Zhou, X. Chen, E. Li, L. Zeng, K. Luo, and J. Zhang, "Edge intelligence: Paving the last mile of artificial intelligence with edge computing," *Proceedings of the IEEE*, vol. 107, no. 8, pp. 1738–1762, 2019.

[5] E. Li, L. Zeng, Z. Zhou, and X. Chen, "Edge ai: On-demand accelerating deep neural network inference via edge computing," *IEEE Transactions on Wireless Communications*, vol. 19, no. 1, pp. 447–457, 2020.

[6] B. McMahan, E. Moore, D. Ramage, S. Hampson, and B. A. y Arcas, "Communication-efficient learning of deep networks from decentralized data," in *Artificial intelligence and statistics*. PMLR, 2017, pp. 1273–1282.

[7] Y. Kang, J. Hauswald, C. Gao, A. Rovinski, T. Mudge, J. Mars, and L. Tang, "Neurosurgeon: Collaborative intelligence between the cloud and mobile edge," in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 615–629. [Online]. Available: https://doi.org/10.1145/3037697.3037698

[8] D. Lahat, T. Adali, and C. Jutten, "Multimodal data fusion: An overview of methods, challenges, and prospects," *Proceedings of the IEEE*, vol. 103, no. 9, pp. 1449–1477, 2015.

[9] J. A. Zhang, M. L. Rahman, K. Wu, X. Huang, Y. J. Guo, S. Chen, and J. Yuan, "Enabling joint communication and radar sensing in mobile networks—a survey," *IEEE Communications Surveys Tutorials*, vol. 24, no. 1, pp. 306–345, 2022.

[10] Y. Ma, G. Zhou, and S. Wang, "Wifi sensing with channel state information: A survey," *ACM Comput. Surv.*, vol. 52, no. 3, Jun. 2019. [Online]. Available: https://doi.org/10.1145/3310194

[11] F. Zafari, A. Gkelias, and K. K. Leung, "A survey of indoor localization systems and technologies," *IEEE Communications Surveys Tutorials*, vol. 21, no. 3, pp. 2568–2599, 2019.

[12] X. Xu, F. Zhu, S. Han, Z. Yu, H. Zhao, B. Wang, and P. Zhang, "Swin-loc: Transformer-based csi fingerprinting indoor localization with mimo isac system," *IEEE Transactions on Vehicular Technology*, vol. 73, no. 8, pp. 11 664–11 679, 2024.

[13] N. S. Bhatia and K. Obraczka, "Transforming decoder-only transformers for accurate wifi-telemetry based indoor localization," 2025. [Online]. Available: https://arxiv.org/abs/2505.15835

[14] D. Avola, A. Bernardini, F. Danese, M. Lezoche, M. Mancini, D. Pannone, and A. Ranaldi, "Transformer-based person identification via wi-fi csi amplitude and phase perturbations," 2025. [Online]. Available: https://arxiv.org/abs/2507.12854

[15] E. Gönültaş, E. Lei, J. Langerman, H. Huang, and C. Studer, "Csi-based multi-antenna and multi-point indoor positioning using probability fusion," *IEEE Transactions on Wireless Communications*, vol. 21, no. 4, pp. 2162–2176, 2022.

[16] C. Pahl and B. Lee, "Containers and clusters for edge cloud architectures – a technology review," in *2015 3rd International Conference on Future Internet of Things and Cloud*, 2015, pp. 379–386.

[17] D. Merkel, "Docker: lightweight linux containers for consistent development and deployment," *Linux J.*, vol. 2014, no. 239, Mar. 2014.

[18] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes, "Borg, omega, and kubernetes," *Commun. ACM*, vol. 59, no. 5, p. 50–57, Apr. 2016. [Online]. Available: https://doi.org/10.1145/2890784

[19] E. Khorov, A. Kiryanov, A. Lyakhov, and G. Bianchi, "A tutorial on ieee 802.11ax high efficiency wlans," *IEEE Communications Surveys Tutorials*, vol. 21, no. 1, pp. 197–216, 2019.

[20] J. Shao and R. R. Sitter, "Bootstrap for imputed survey data," *Journal of the American Statistical Association*, vol. 91, no. 435, pp. 1278–1288, 1996. [Online]. Available: http://www.jstor.org/stable/2291746

[21] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," in *Proceedings of the 32nd International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, F. Bach and D. Blei, Eds., vol. 37. Lille, France: PMLR, 07–09 Jul 2015, pp. 448–456. [Online]. Available: https://proceedings.mlr.press/v37/ioffe15.html

[22] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: a simple way to prevent neural networks from overfitting," *J. Mach. Learn. Res.*, vol. 15, no. 1, p. 1929–1958, Jan. 2014.

[23] J. Shao and J. Zhang, "Communication-computation trade-off in resource-constrained edge inference," *IEEE Communications Magazine*, vol. 58, no. 12, pp. 20–26, 2020.