Empirical Evaluation of Edge Computing for Smart Building Streaming IoT Applications

Talha Ghaffar

Thesis submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Masters of Science
in
Computer Science and Applications

Dongyoon Lee, Chair
Ali R. Butt
Changhee Jung

February 7, 2019
Blacksburg, Virginia

Keywords: Edge Computing, Stream Processing, Internet of Things, Apache Storm

Copyright 2019, Talha Ghaffar

Empirical Evaluation of Edge Computing for Smart Building Streaming IoT Applications

Talha Ghaffar

(ABSTRACT)

Smart buildings are one of the most important emerging applications of Internet of Things (IoT). The astronomical growth in IoT devices, data generated from these devices and ubiquitous connectivity have given rise to a new computing paradigm, referred to as "Edge computing", which argues for data analysis to be performed at the "edge" of the IoT infrastructure, near the data source. The development of efficient Edge computing systems must be based on advanced understanding of performance benefits that Edge computing can offer. The goal of this work is to develop this understanding by examining the end-to-end latency and throughput performance characteristics of Smart building streaming IoT applications when deployed at the resource-constrained infrastructure Edge and to compare it against the performance that can be achieved by utilizing Cloud's data-center resources. This work also presents a real-time streaming application to detect and localize the footstep impacts generated by a building's occupant while walking. We characterize this application's performance for Edge and Cloud computing and utilize a hybrid scheme that (1) offers maximum of around 60% and 65% reduced latency compared to Edge and Cloud respectively for similar throughput performance and (2) enables processing of higher ingestion rates by eliminating network bottleneck.

Empirical Evaluation of Edge Computing for Smart Building Streaming IoT Applications

Talha Ghaffar

(GENERAL AUDIENCE ABSTRACT)

Among the various emerging applications of Internet of Things (IoT) are Smart buildings, that allow us to monitor and manipulate various operating parameters of a building by instrumenting it with sensor and actuator devices (Things). These devices operate continuously and generate unbounded streams of data that needs to be processed at low latency. This data, until recently, has been processed by the IoT applications deployed in the Cloud at the cost of high network latency of accessing Cloud's resources. However, the increasing availability of IoT devices, ubiquitous connectivity, and exponential growth in the volume of IoT data has given rise to a new computing paradigm, referred to as "Edge computing". Edge computing argues that IoT data should be analyzed near its source (at the network's Edge) in order to eliminate high latency of accessing Cloud for data processing. In order to develop efficient Edge computing systems, an in-depth understanding of the trade-offs involved in Edge and Cloud computing paradigms is required. In this work, we seek to understand these trade-offs and the potential benefits of Edge computing. We examine end-to-end latency and throughput performance characteristics of Smart building streaming IoT applications by deploying them at the resource-constrained Edge and compare it against the performance that can be achieved by Cloud deployment. We also present a real-time streaming application to detect and localize the footstep impacts generated by a building's occupant while walking. We characterize this application's performance for Edge and Cloud computing and utilize a hybrid scheme that (1) offers maximum of around 60% and 65% reduced latency compared to Edge and Cloud respectively for similar throughput performance and (2) enables processing of higher ingestion rates by eliminating network bottleneck.

Dedication

Dedicated to my parents without whom this would not have been possible

Acknowledgments

I would like to express my gratitude to my advisor Dr. Dongyoon Lee, and the committee members Dr. Ali Butt and Dr. Changhee Jung for their guidance. I would also like to thank Dr. Pablo Tarazaga, Dr. Rodrigo Sarlo and Sa'ed Alajlouni from Virginia Tech Smart Infrastructure Laboratory (VTSIL) for their collaboration in this work.

Contents

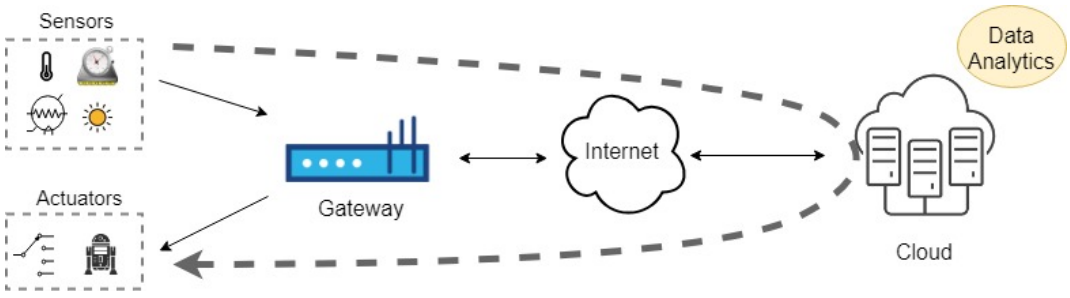
List of Figures

Chapter 1

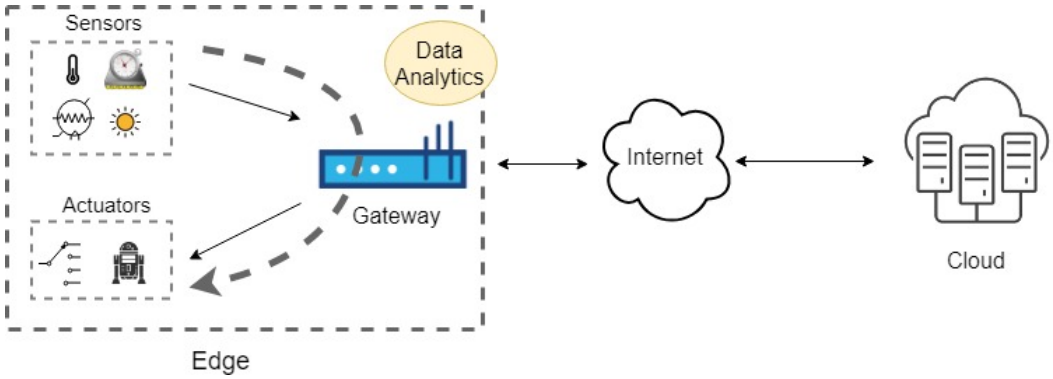
Introduction

Smart buildings are one of the most important emerging applications of Internet of Things (IoT) [52, 56]. The increasing availability of various sensor and actuator devices, and ubiquitous Internet connectivity has allowed us to instrument buildings with these devices (Things) to measure and manipulate the operating parameters of our buildings. These buildings may deploy sensor devices such as temperature, light, smoke, occupancy, motion, key-card readers and various others. These sensor devices continuously "read" data from their surrounding environment and generate unbounded data-streams that need to be processed in real-time. Analysis of these data streams in real-time can help us understand and monitor building's performance and in-door activities as they happen. Once this data is analyzed, various actuators deployed in the building can be triggered to adjust building parameters accordingly. For example, occupancy in a smart building can be analyzed for efficient power management [14]; fire suppression system can be activated if fire is detected [68] and occupant evacuations can be assisted in case of emergency [70].

Until recently, the processing paradigm to analyze data generated from IoT devices (such as sensor nodes in smart buildings) has been to utilize the computation resources available in the Cloud data centers [12, 40, 53, 77, 79]. Data captured from sensing devices is routed via local Gateway nodes to the Cloud data centers, where this data is analyzed. After analysis, the processed results are sent back to the IoT devices to trigger actuators accordingly or to monitoring dashboards for visualization purposes. This computing paradigm is termed as



(a) Cloud computing



(b) Edge computing

Figure 1.1: Edge and Cloud computing paradigms for streaming IoT applications (a) Cloud computing - Sensor data is streamed to the Cloud data centers for analysis and the processed results are sent back to the IoT devices to possibly trigger actuator states (b) Edge computing - Sensor data is streamed to local gateway nodes where it is processed and analyzed; reducing the network latency incurred in (a)

Cloud computing and is illustrated in Figure 1.1a. Cloud computing of IoT data introduces a challenge that sensor data potentially has to suffer from high latency before it can be processed in the Cloud. Furthermore, with astronomical increase in IoT devices and data [42], such deployment scheme also introduces significant scalability and bandwidth challenges [82].

To address these challenges, and to ensure that the IoT growth remains uncurtailed, computing paradigms have been proposed that argue for moving data processing and analysis to the network's edge, closer to the source of data [7, 38, 43, 47]. This paradigm is generally referred to as Edge computing and is illustrated in Figure 1.1b. Edge computing relies on the utilizing the computation resources of devices such as IoT Gateways, routers, access points

and small servers available at the Edge of the IoT infrastructure. From the sensor nodes, data is streamed to these gateway nodes where data processing services analyze this data. After analysis, instructions to actuator nodes are sent directly from these gateway nodes. By processing data closer to its source, IoT analytics can complete with very low latency; thus enabling near real-time IoT applications. In Smart buildings, this can be helpful by enabling timely evacuations, fire suppression and intrusion detection. Furthermore, processing at the Edge can also ease the increasing bandwidth requirements for the network by limiting the data that needs to be sent to the Cloud data centers.

Various works [43, 63, 75, 80] have motivated the need for Edge computing of IoT applications and analytically argued for its performance benefits against Cloud computing. However, these works do not attempt to offer empirical evidence for performance benefits of Edge computing. Unfortunately, little work has been done to empirically understand the performance benefits of Edge computing of streaming IoT applications. Most of these works have focused on mobile applications. For example, Zhuo et al. [31] show that offloading cognitive assistance applications from mobile or wearable devices to desktop-class resources (Cloudlets) can yield performance benefits, and Hassan et al. [49] show that nearby resource-constrained nodes can be used to offload tasks from mobile applications for better performance and can provide storage expansion for mobile devices.

The development and deployment of efficient Edge computing systems must be based on detailed and advanced understanding of the performance benefits that Edge computing can offer. On one hand, Edge computing can potentially eliminate the network latency cost at the cost of limited computation resources; while on the other hand, Cloud computing can offer resource-rich compute resources, albeit at the cost of high latency. The adage "If you can't measure it, you can't improve it" advises us to ask the question what potential benefits Edge computing can offer and to what extent can this performance can be better

than Cloud computing. In this work, we are motivated to understand the trade-offs of Edge and Cloud computing and seek an answer to the sources of potential benefits of Edge computing by performing an empirical examination of Edge and Cloud deployment of Smart buildings related IoT applications. Because of the immense diversity of IoT devices and their communication mechanisms, we restrict the scope of this study to a model more suited for the Smart buildings use case.

Edge Model: Various projects in the open source community are trying to provide a platform that can run on single node Edge devices such as Gateways, Routers or Embedded PCs and also on a distributed architecture of these Edge devices e.g. Kura [41], EdgeX [3], and OpenFog [7]. Similar to these IoT frameworks, in this work, we view the Edge network as a distributed collection of IoT Gateways deployed in the buildings that connect Things to the Cloud and enable local data analysis.

In the context of Smart buildings, these IoT devices and Gateways are reasonably stable and well connected - unlike mobile drones or vehicles - and can potentially communicate over wired local-area-networks (LAN) without involving mobile networks. Furthermore, these IoT Gateways deployed at multiple locations in a Smart building. Furthermore, in this work, we consider these IoT Gateways to have limited computing resources compared to the Cloud: few-core processors, little memory, and little permanent storage [24, 75]. However, they can have more resources than those available to embedded, wireless sensor networks [60], and thus can afford reasonably complex software like SPEs. For example, Cisco's IoT Gateways come with a quad-core processor and 1GB of RAM [33].

We will use this Edge model for the infrastructure and hardware used in the empirical evaluation of Edge computing in chapter 4. Following is the description of the IoT applications and their processing mechanism considered in this work.

Edge Application Model: Since IoT devices generate unbounded data streams, we consider that modern Stream Processing Engines (SPEs) (§2.2) are ideally suited to process & analyze sensor data-streams. These stream processing frameworks can also be used to develop and deploy general purpose IoT applications. We find that there are only a limited number of open-source streaming IoT applications available (§2.2.4). The Virginia Tech Campus also houses a model 'Smart building' that has been instrumented with accelerometer sensors to study its behavior under the influence of various stimulations that generate vibrations in the building. We see that this smart building can serve as a test-bed for real-world IoT applications. In this work, we use the data generated from accelerometer sensors in a walking experiment to develop a streaming Smart building application that can detect and localize the footsteps of the walking occupant.

In this work, we will use the existing streaming IoT applications (§2.2.4) and the Footstep Impact Localization smart building application, developed as a part of this thesis (§3.2), to perform empirical examination of Edge and Cloud deployment of streaming Smart building IoT applications.

1.1 Contributions

In this work, we present the following contributions: (a) Developing a real-time version for building occupant localization application that can be used as an IoT application to benchmark stream processing engines. (b) Empirical evaluation of the benefits offered by Edge computing for Smart building streaming IoT applications.

1.1.1 Real-time Occupant Footstep Impact Localization in Smart Buildings

Owing to the increasing availability of various sensing devices and capability to connect them to computation resources has given rise to various IoT application scenarios. We can now instrument our homes and buildings with these devices to monitor their operating parameters. The ability to analyze these parameters allows us to imagine futuristic buildings that communicate with their occupants to facilitate their activities, manage their resources efficiently, detect any suspicious activities and assist in emergency evacuations. The existence of such a Smart building on Virginia Tech's campus (Goodwin Hall) provides us a unique test-bed that can detect vibrations in the building's structure in real-time which can be used for various IoT applications (§3.2). With the ideals of futuristic smart buildings and data from Goodwin Hall, we attempt to develop a practical real-time streaming application for detecting and localizing occupants' movements in a smart building. The algorithm and heuristics for occupant tracking are based on vibration sensor data sampled from accelerometers [15].

In this work, we utilized and adapted this to develop the streaming version of the application. We created a synthetic data generator that simulates real-time generation of vibration data from accelerometers deployed in multiple corridors of a Smart building. To process this data in real-time, we developed an Apache Storm [20] topology that detects footstep impacts in the vibration signals. Once footsteps are detected, it estimates the location of the footstep impacts in the corridors. This work lays also the foundation for further work on similar smart building applications [25, 55, 72] and help realize the futuristic smart buildings.

1.1.2 Empirical Evaluation of Edge Computing for Streaming IoT Applications in Smart Buildings

The emergence of Internet of Things (IoT) and the subsequent astronomical growth in the data generated by IoT applications led to the emergence of Edge computing paradigms. One of the principle motivation for Edge Computing is to reduce the end-to-end latency of offloading all computation to the Cloud back-end. In this work, we empirically examine the end-to-end latency performance for Smart building related streaming IoT applications for Edge and Cloud computing. We considered an Edge infrastructure where resource-constrained Gateway devices, available at the Edge, are connected to IoT sensor nodes over a local network. These Gateway devices also provide a route to the Cloud data centers. At the Edge, we utilize these Gateway nodes are used to process data whereas in Cloud data-centers, VMs are deployed to process data. We employ Apache Storm as the streaming engine to process real time streams of IoT sensor data. In order to move data from sensor nodes to the processing nodes, we use message queue brokers as intermediary messaging middleware. This empirical study demonstrates that for Edge devices with limited resources, application's performance characteristics and data ingestion rates are important factor in determining whether the applications should be processed at the Edge nodes or the Cloud. At lower ingestion rates, the representative Edge devices can process the data at lower networking cost without overwhelming the deployed streaming engine. At higher ingestion rates, however, the representative Edge devices can no longer process the data under the given latency SLA and it's best to utilize Cloud data centers for processing. We also study that multiple gateway nodes at the Edge can also be utilized to handle higher ingestion rates at lower latency than Cloud.

1.2 Thesis Organization

The remaining of this thesis is organized as follows: In chapter 2, we present related works and the background for understanding this work. In chapter 3, we discuss the Smart building infrastructure set up in the Virginia Tech's Goodwin Hall building and describe the creation of a real-time occupant's footstep impact localization application that can also be used as a benchmark IoT application for Stream Processing Engines (SPEs). Chapter 4 focuses on the empirical examination of Edge and Cloud computing paradigms for Smart building related streaming IoT applications. We describe our experimental setting and model in detail and present the results for our evaluation. Finally, chapter 5 provides a conclusion to the thesis, summarizes our work and discusses the future directions.

Chapter 2

Background and Related Work

This work focuses on a comparative analysis of Edge and Cloud computing for IoT streaming applications and introduces a real-world Smart building application that can be used to characterize the performance of streaming engines for IoT applications. In this chapter, we discuss the background needed for understanding different aspects of this thesis. This chapter discusses the Edge computing paradigms and summarizes the current research conducted for Edge Computing. Since this work is focused on utilizing streaming engines for IoT applications, we also discuss the programming models they employ and state-of-the-art streaming engines. We also discuss the available IoT applications for benchmarking streaming engines. The following subsections discuss this background in detail.

2.1 Edge Computing

Over the last decade, Cloud computing has been significantly important in driving the growth in IT by providing scalable, centralized infrastructure and resources and thus supporting quick deployment and wider reach for applications. The emergence of IoT and rapid growth in data generated from IoT devices has caused problems with the centralized model employed by the Cloud. Furthermore, moving data to the far-away Cloud is a hindrance for low-latency IoT applications. These challenges have led to the emergence of computing paradigms that process data closer to the Edge.

Edge Computing, in general, refers to computing paradigms that argue for decentralized processing of closer near its source. In the context of IoT, *Edge* refers to the network's periphery where different sensors, actuators and IoT devices such as Gateways and routers are located. The IoT devices at the network's Edge have intermediate-class computing resources; few-core processors, limited memory, and little permanent storage [33]. They are, however, capable enough to process limited amounts of data. As these devices are just one network hop away, if data analysis is moved to these devices from the Cloud, significant latency benefits can be obtained.

Various existing works in recent years have focused on Edge computing paradigms that can be distinguished based on the architecture or resources they operate on. For example, Mobile Edge Computing [44] tries to harness the computation resources in mobile, portable devices connected in a distributed architecture that can communicate via cellular communication protocols. With the pervasiveness of Cloud Computing, emerged Mobile Cloud Computing [37, 45, 54] where mobile devices with limited resources offload computation to the Cloud to leverage the resource-rich Cloud infrastructure. Mobile Cloudlet Computing [74] tries to address latency problems in Mobile Cloud Computing by utilizing the computation resources of desktop-class computers connected in a network closer to the mobile devices. Yousefpour et al. [81] provide a detailed survey of these computing paradigms.

Fog computing [43] has emerged as a paradigm that tries to enable a *"horizontal architecture"* that moves the responsibilities of traditional Cloud; computing, storage, data management and networking, to distributed devices devices available along the *Things-to-Cloud* continuum. The OpenFog Consortium [7] is trying to enable and advance Fog Computing by defining standards and creating an open reference architecture. EdgeX [3] is another open-source effort to enable distributed Edge Computing on the Fog nodes along the Things to Cloud path.

Very recently, some works have tried to study the effectiveness of Edge Computing. Zhu et al. [31] have a similar motivation to this work. They try to empirically evaluate the effectiveness of Edge Computing in a Mobile-Cloudlet architecture. They consider wearable cognitive assistance applications for cellular devices that offload compute-intensive task to Cloudlets (desktop-class machines, one wireless hop away from mobile devices) over mobile networks or WiFi. In contrast, this work focuses on Edge nodes to be resource-constrained, stable and connected over Ethernet such as in Smart buildings (§3.2). Morabito [66] evaluates resource constrained devices for Edge Computing tasks under varying workloads for resource utilization and power consumption and shows that single-board computers (as used in this work) can be used for Edge computing tasks.

2.2 Stream Processing

In this section, we provide a background on the stream processing programming model (§2.2.1) and the software architectures used in existing SPEs (§2.2.2).

2.2.1 Programming Model

Stream Processing refers to processing of data streams that are continuous and unbounded in nature. These data streams could originate from different sources such as including event logs, video, time series or sensor data. To handle these streams of data, many stream processing systems employ the dataflow programming model [30, 78]. In this model, as shown in Figure 2.1, the data *tuples* flow through a directed acyclic graph (*topology*) from *sources* to *sinks*. Each inner node is an *operator* that performs arbitrary computation on the data, ranging from simple filtering to relatively complex operations like ML-based classification

algorithms. In the Edge context a source might be providing data from an IoT sensor, while a sink might be publishing data to a message broker for consumption by an actuator device or a monitoring unit for analysis visualization.

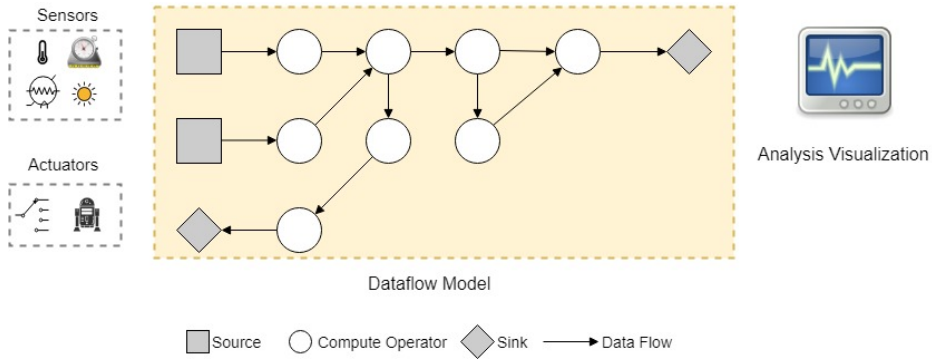


Figure 2.1: Dataflow Model for Processing IoT Applications

Though an operation can be arbitrary, they are preferred to perform computation tasks while I/O should be handled by source and sink nodes, and the inner operation nodes should perform only memory and CPU-intensive operations [5, 50]. This is based on the premise that the more unpredictable costs of I/O will complicate the scheduling and balancing of operations.

After defining the data flow abstraction, operators and how the data flows between operators, data engineers define a *physical plan* for the topology that describes the number of physical instances of each logical operator. In a distributed setting engineers can also indicate preferred mappings from operations to specific compute nodes. An SPE then deploys the operators onto the compute node(s), instantiates queues and workers, and manages the flow of tuples from one operator to another.

2.2.2 Stream Processing Engines

Here, we briefly touch upon the existing solutions for stream processing. Broadly speaking, current stream processing solutions follow two approaches to data stream processing: (a) the dataflow graph model where the streaming engines ingest data streams and each operator processes a single tuple at any given time; and (b) micro-batching model where incoming data tuples are grouped into micro-batches and are processed together as a batch.

The first generation of stream processing engines were proposed and designed by the database community in the early 2000s. These included systems such as Aurora [30], Stream [22], and Borealis [13]. The second generation of "modern SPEs" began with Apache Storm [20] (2012) as part of the democratization of big data. These second-generation SPEs have been mainly developed by practitioners with a focus on scalable Cloud computing and have achieved broad adoption in industry.

Under the hood, most of these modern SPEs are based on an architecture where the operations are connected by queues in a pipelined manner, and processed by its own workers. Some operations may be mapped onto different nodes for distributed computing or to take advantage of heterogeneous resources (GPU, FPGA, etc.). In addition, these SPEs monitor the health of the topology by checking the lengths of the per-operation queues. Queue lengths are bounded by a *backpressure* mechanism [35, 59], during which the source(s) buffer input until the operation queues clear.

Among these modern SPEs, **Apache Storm** [20] is one of the most popular engines and employs data-flow programming model. The streams of data are processed by operator nodes in the graph where each operator is executed by one or more threads. The operators are executed by worker processes which can be deployed across multiple nodes.

Apache Flink [17] supports both stream and batch processing and closely follows the

OWPOA model. Applications are constructed as dataflow graphs that consume data streams from one or more sources. Different operators apply transformations on to the stream and the execution ends at one or more sinks. To allow parallel execution, Flink allows both streams and tasks to be split into *stream partitions* and *subtasks* which can be executed independently.

Apache Samza [18] is an SPE that also allows creation of stateful applications. Samza's streaming application is composed by chaining multiple tasks where each task is an independent processing unit that consumes a partitioned data stream. Task(s) are executed in containers which can be distributed across multiple physical nodes.

Apache Spark [19], originally designed for batch processing, also supports streaming by discretizing the incoming data streams into micro batches. It employs a unified programming model and execution engine for both stream and batch processing. For better load balancing and fault recovery, Spark's tasks are dynamically allocated to workers based on data locality and available resources. These tasks operate on the micro-batches and output the results for other tasks.

2.2.3 Stream Processing at the Edge

Since "Things" generate continuous stream of data, Stream Processing Engines (SPEs) are well suited to processing these real-time data streams. With these SPEs deployed at Edge, we can process this data in a timely fashion.

Apache Edgent [2] is an SPE specifically tailored for the Edge. It is designed for data pre-processing at individual Edge devices rather than full-fledged distributed stream processing. It enables a simple form of data pre-processing at an end device, as evidenced by its simple, limited set of predefined operations: e.g., `filter` and `join`. This pre-processed data can

then be send to the Cloud for analytics and storage. SpanEdge [71] tries to provide a programming environment where geo-distributed streaming applications can be developed with certain programmer-specified operations can be deployed at the Edge. Papageorgiou et al. [69] try to extend Apache Storm to enable Edge-aware Storm so that some streaming tasks can be deployed at the Edge. We use similar inspiration in §4.3.5 to deploy tasks on both Edge and Cloud nodes to achieve better performance. Mobile Storm [67] ports Apache Storm to process real-time streaming data on mobile devices. Similar to [67], this work deploys Apache Storm on resource-constrained Edge nodes to process IoT streaming data in real-time.

2.2.4 IoT Benchmarks for Streaming Engines

Various benchmarks have been proposed to evaluate and compare the performance of distributed stream processing systems.

StreamBench [62] proposes 7 micro-benchmarks to represent simplistic stream computing scenarios such as wordcount, grep and sampling. The microbenchmarks operate on synthetic workloads from real-time web logs such as AOL Search data and Internet traces dataset. It focuses on measuring metrics for performance (throughput, latency), fault tolerance (Throughput/Latency penalty factors) and durability (Availability Ratio). The benchmark is available for Apache Storm and Spark Streaming. Similar to StreamBench [10] is Storm Benchmark which contain 9 Storm workloads (such as wordcount, Rolling count, Rolling Sort and Grep). The benchmark has the goal of evaluating Apache Storm's performance under resource pressure and also under relatively simple typical streaming use cases. It's design to use Kafka as the source for data ingestion.

At Yahoo, an advertisement campaign topology was created to benchmark the peformance of

Apache Storm, Flink and Spark [32]. The topology reads JSON events from Kafka, identifies and filters relevant events, extracts relevant fields from messages and performs a Join with data from Redis and stores a count of relevant events for each campaign in Redis. Besides these benchmarks, some works use micro-benchmarking streaming scenarios to quantify the performance of their work. For example, StreamBox [64] uses temporal join, tweets sentiment analysis for its performance study.

IoTAbench [23] is a benchmark for evaluating Big Data analytics platforms for IoT. It focuses on providing a synthetic data generator for smart-meter data that and SQL-based queries to analyze the generated data. The benchmark, however, is not designed for streaming use cases. CityBench [16] is another benchmark that focuses on IoT applications. It uses smart city applications that use IoT streams generated from sensors deployed in a Aarhus, Denmark as workload. The benchmark, however, is focused on evaluating the performance RDF Stream Processing.

RIoTBench [76] is a benchmark most suited to real-time processing of IoT data streams. It provides a benchmarking suite to evaluate the performance of streaming engines for IoT applications. It includes a set of IoT tasks as micro-benchmarks from various categories. It also IoT applications that depict some common IoT data processing patterns such as: (1) ETL (extract-transform-load) for incoming sensor data streams to filter any outliers and perform interpolation in case of missing data. Eventually, it publishes transformed data to a message queue to notify data availability to the subscribers and to permanent data storage in the Cloud. (2) Utilizing data from various sensors to perform predictive analytics using various machine learning models (e.g. predicting air quality from environment monitoring sensors' data). (3) Periodically training (and updating) machine learning models used for predictive analytics by using saved sensor data after ETL. (4) Performing various statistical analysis over incoming sensor data streams such as windowed-average, count, N^{th} order

moments or finding outliers. These benchmarking applications can use data from sources such as Smart City [29].

The RIoTBench applications, however, have only been evaluated using resource-rich Cloud VMs where data sources (synthetic sensor data generators) and data sinks are also deployed in the Cloud VMs which is not a realistic case for IoT applications. In this work, besides the Smart building application we present in §3.2, we will also use the RIoTBench applications for the empirical examination of Edge and Cloud computing.

Chapter 3

Real-time Occupant Footstep Localization in Smart Buildings

3.1 Introduction

The emergence of Internet of Things (IoT) has given rise to various application scenarios. These applications involve various sensing devices that continuously generate large volumes of data which then needs to be processed to gain actionable insights. One of such emerging applications of IoT is the Smart buildings. The increased availability of various types of sensing devices has enabled us to instrument the buildings to monitor different operating parameters of our buildings. Such sensing devices in the buildings could include temperature, light, smoke, air quality, occupancy, motion, key-card readers, door open/close and various others. Together, these sensing devices enable us to monitor the building's performance and in-door activities. If this data can be processed in real-time, we can imagine a building that communicates with its occupants to facilitate their activities. Such a building can enable efficient utilization of its resources, detect and alert about suspicious activities and assist in emergency evacuations from the building.

Various buildings in the literature such as [11, 28, 34] have been instrumented for research in building dynamics and structural health monitoring. [61] has been instrumented to allow real-time monitoring of the buildings' structure, temperature, air-flow, energy demand and

soil moisture etc.

The Virginia Tech Campus also houses a model 'Smart building' - Goodwin Hall - an active building that contains various laboratories, lecture rooms, auditoriums and offices etc. The Virginia Tech Smart Infrastructure Laboratory (VTSIL) has instrumented the Goodwin Hall building with accelerometers with plans for adding other sensors such as strain gauges, thermocouples, and air flow sensors for further extension. The accelerometers in Goodwin Hall are mounted directly onto structural parts of the building. Currently, in Goodwin Hall, various labs, hallways and structural columns are installed with accelerometers. Using the vibration data collected from these sensors, VTSIL works on developing mathematical models to study occupant classification [25], structural health [72] and occupant tracking [15]. Since the sensors are deployed out-of-sight in the building's structure or underfloor, the real-time deployment of these works can help achieve the futuristic, non-invasive smart buildings.

In order to collect data from these deployed sensors, a distributed data acquisition (DAQ) system has been deployed in Goodwin Hall [48]. The sensors are directly connected to the DAQ units which are positioned throughout the building to limit the cable length to the sensors so as to limit noise and interference from voltage external sources. These DAQ units record measurements from the sensors and are connected to each other via CAT6 Ethernet cables on a local network. A GPS master clock is deployed to synchronize and timestamp the measurements from these sensors. The DAQ units are also connected to a local data acquisition server (deployed on the third floor of the building). This server gathers data from these units, processes and stores the data in Hadoop Distributed FileSystem (HDFS).

Currently, the processing mechanism is mostly offline and works in batch processing mode. The experiments are conducted in a controlled environment and data is collected and stored for the duration of the experiment. The algorithms developed for this sensor data can then be run offline on the collected data.

Based on such data acquisition mechanism, we can envision an Edge model where Gateway devices are deployed along with the Data Acquisition Units to process this data and then direct the processed results to centralized servers, monitoring units and storage systesm to save processed data for posterity.

3.1.1 Problem Statement

In this chapter, our objective is to develop the real-time streaming application for detecting and localizing human footsteps in a smart building. The localization application with realistic workload can also be used to benchmark performance of streaming engines for IoT applications.

3.1.2 Contributions

In this chapter, we present the following contributions: (a) Discussion of how the instrumented Goodwin Hall building on Virginia Tech's campus can use the Edge network to deploy real-time versions of the smart building applications, (b) Developing an Apache Storm application for real-time processing of accelerometer sensor data for human footstep detection and localization by adapting the algorithm presented in [15] and (c) discussion of how the developed occupant tracking and its sister applications are different from existing IoT benchmarking applications for Apache Storm [76].

3.2 Footstep Impact Localization

Any significant impact that happens on the floor of a building generates vibrations along the floor. Since the Goodwin Hall building is instrumented with an accelerometer sensor

network underneath the hallways, it is capable of detecting such vibrations. Triangulating these impacts can lead to the possibility of various 'smart' applications in the building. One such application scenario is the localization and tracking of the building's occupants. In this section, we briefly discuss the heuristic algorithm for footstep impact localization and describe how the offline, batch processing algorithm was adopted for real-time processing using Apache Storm as the processing engine. The other applications based on similar vibration data from accelerometers such as gunshot detection and classification [55], occupant classification [15] and operational modal analysis [73] can be developed and used in a similar manner. The impact localization application described in this section can also be used to benchmark the performance of stream processing engines as a real-world IoT application.

3.2.1 Heuristic Algorithm for Footstep Impact Localization

The localization algorithm is based on a heuristic method for estimating the location of the source from the vibration readings generated by the impact. The method is based on the fact that as a wave travels away from its source, its energy is attenuated. The algorithm uses raw data from the accelerometer sensors. The required data sampling rate for the purpose of footstep localization is only 1 kHz which is relatively small. Initially, the sensor data is used to detect any impact in the floor. The impact detection essentially detects peaks in the sampled data. After a footstep has been detected using readings from all sensors, the individual peaks for each sensor is then located. From these peaks, the algorithm goes back in time to find the beginning of the impact that generated the wave packet.

3.2.2 Sensor Data For Footstep Impacts

Data Collection

For footstep impact localization, the vibration data is collected from an experiment where the test subject is walking in a cordoned off, isolated 16x2 meter hallway that is instrumented with underfloor accelerometers. The data is collected for a total of 162 footsteps with 81 unique impact locations where each location was stepped upon twice while the subject walked from opposite directions. The data comprises of readings from 11 different sensors in the hallway and is collected at a sampling rate of 8 kHz.

Simulating Real-Time Generation Of Sensor Data

To simulate footstep impact localization in real-time, we used the collected accelerometer sensor data as traces and replayed these traces continuously at the rate they were sampled (8 kHz). Various methodologies could be employed to simulate this data generation. In our setting, we employed two different methodologies: (a) One sensor reading is treated as a single measurement and the data generator publishes this reading directly to the messaging middleware with the unique sensor identifier. (b) Since the data from sensors is being collected by the Data Acquisition units, we assume a small buffer at the DAQs that can hold data for 1 second. The data held in this buffer is then published to the message queue. In order to scale this data generation from one hallway to an entire building, we simulate multiple generators; each with a unique identifier for different corridors in the building (and for sensors in that corridor).

3.2.3 Designing Compute Units

The Impact Localization algorithm [15] is designed for a batch processing scenario. The impact data is collected for the walking experiments and the analysis is then executed on the entire dataset. The prototype for this application was designed in `MATLAB`. To create an Apache Storm application for this prototype, we needed to to convert the application to its streaming version and translate the code to Java (the language primarily used for creating Apache Storm topologies). As `MATLAB` API implementation was not available, we used reference documentation as a guide to implement corresponding signal processing APIs in Java that could be used by the Storm Topology. The figure 3.1 shows the Apache Storm topology for Footstep Impact Localization (FSL).

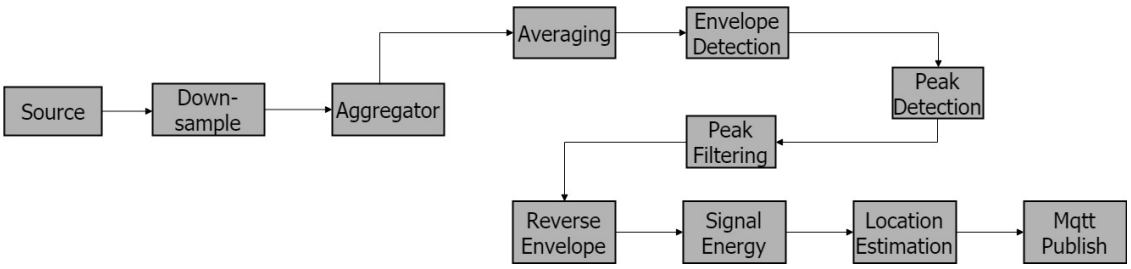


Figure 3.1: Footstep Impact Localization (`FSL`) Topology

Here, we describe the various components of the FSL topology.

Topology's Data Source

In Apache Storm, source of data for the topology is called *spout*. Spout is responsible for either generating data for the topology or reading data from external sources and feeding that data to the topology. The data generation code for the Storm's spout *nextTuple* is executed on a thread. When scheduled, the spout should emit one data 'tuple' into the topology. In our setup, during the topology initialization phase, the spout initializes a consumer for

the message broker. Since we use RabbitMq [8] as the message broker for this topology, two mechanisms are available to consume data from the broker (a) whenever the data is available at the broker, the broker 'pushes' the data to the consumer and (b) the consumer periodically 'polls' the data from the broker. For better performance, we set up the spout so that whenever available, the broker 'pushes' the data to the consumer. Upon receiving the data, the consumer adds timestamp to the data and adds this event to the event queue of the spout. When the spout is scheduled to execute *nextTuple*, it consumes events from this queue and emits a *tuple* to the downstream bolt of the topology.

Signal Downsampling

Downsampling is the first compute unit for the topology. This bolt receives a tuple containing corridor and sensor identifiers and data for that sensor and then down-samples the received signal to a 1 kHz signal. The bolt is connected to the spout via *Fields Grouping* on the corridor id, so that when multiple executors are deployed, the data from a particular corridor always goes to the same executor. The `MATLAB` api used for down-sampling uses a low-pass Chebyshev Type I IIR filter of order 8. We created an order 8 FIR filter in `MATLAB` and convolved that filter with the original signal to very similar down-sampling behavior. Then we used the filter parameters generated in `MATLAB` in this bolt and convolve the incoming signal with the filter to generate a low pass signal. Then, to down-sample the signal to 1 kHz, we choose every *Nth* data point in the signal to create a signal for the downstream bolt.

Sensor Data Aggregator

The data aggregator compute unit receives down-sampled signal from all the sensors. As the Footstep Detection algorithm needs to detect peaks in the combined averaged signal

from all sensors deployed in a corridor, this bolt is responsible to aggregate signals from a particular corridor. The downsampled signals from these corridors are aggregated into a matrix which is then emitted by this bolt. This bolt is also connected to its upstream bolt via *Fields Grouping* on corridor Id so that signals from a particular corridor are not received by different instances of the executors and can be merged correctly.

Sensor Signal Averaging

This compute unit receives an aggregated matrix of down-sampled signal from all the sensors in a corridor and calculates an averaged signal for the window such that each data-point in the signal is the average of the absolute values of the corresponding data points in readings from each sensor. This averaged signal is then emitted to the downstream compute units.

Signal Envelope Detection

The footstep detection algorithm then needs to determine the envelope of the averaged signal. This compute unit receives a signal that is averaged over the readings from all the sensors in a corridor. To detect envelope of the signal, this bolt performs cubic spline interpolation using a smoothing interval of $floor(T/4*f)+1$ where $T = 0.25$ seconds; the expected length of footstep waveform, and $f = 1kHz$; the frequency of the signal. The signal envelope is then emitted to the downstream bolt.

Peak Detection & Filtering

The peak detection compute unit receives the envelope of the averaged signal as its input and it determines the peaks (local maxima) in the signal. A peak is determined if the data point is greater than both neighbors (previous and the following data points). In case of a

flat peak, the bolt determines the first point to be the peak. The signal peaks are emitted to the peak filtering bolt.

From the peaks detected in the signal, the peak filtering bolt filters the peaks to determine an individual footstep/impact. It does so by filtering the input peaks based on their amplitude and relative distances. After the filtering, if a peak is detected in the signal, it is determined to be the footstep of a person. The detected peak locations along with the signal are emitted as a *tuple* from the bolt.

Signal Power Estimation

For each detected footstep impact, the Signal Energy compute unit tries to estimate energy of the signals received from all the sensors in a particular corridor. The signal energy is estimated using the formula:

$$Q_s(t_{bs}, t_{ps}) = \frac{1}{t_{ps} - t_{bs}} \times \sum_{t=t_{bs}}^{t_{ps}} sig_s(t)^2 \tag{3.1}$$

In the above equation 3.1, s represents the sensor id, $sig_s(t)$ denotes the reading from sensor s, and $[t_{bs}\ ,\ t_{ps}]$ denotes the time interval over which the signal power Q_s is calculated. Here, t_{bs} is the time when the beginning of the impact was detected, while t_{ps} is the time where peak of the signal's envelope is detected. The signal envelope here is the RMS (root-mean-square) envelope of the signal and is calculated by using a sliding window of 10 samples.

Location Estimation

Location estimation works on the premise that once an impact is generated, the sensors closer to the impact will detect a signal with higher average energy values while the far-

away sensors would detect signals with lower energies. Based on this premise, the impact's location is estimated by weighing the (already known) sensor locations in the hallway with the corresponding energies estimated in the signals detected by them. Equation 3.2 is used to estimate an impact's location:

$$\textbf{Location} = \frac{\sum_{s=1}^{N} Q_s \times L_s}{\sum_{s=1}^{N} Q_s} \tag{3.2}$$

Here, L_s represents the coordinates for sensor s and Q_s represents the signal power calculated in 3.1. The estimated location is a vector containing the x and y coordinates of the impact.

MQTT Publish (Topology's Sink)

We implement the sink for the FSL topology to be an MQTT publisher. Whenever any footstep is detected and its location is estimated by the topology, the sink publishes the estimated coordinates along with the corridor Id to the MQTT broker (set up at the Edge). Any monitoring unit can the subscribe to the broker and track location of the footsteps in the building's hallways.

3.3 Rate of Data Flow in the FSL Topology

In this section, we consider the RIoTBench applications [76] (discussed in §2.2.4) using the Smart City dataset [29] and the FSL topology from the perspective of how the data flows through their topologies. We compare and contrast the data flow in RIoT topologies with the data flow through the Footstep Impact Localization topology.

3.3.1 Experimental Setting

In order to conduct the experiments to measure data flow through the applications, we use a setup similar to 4.7a. We execute the applications using Apache Storm v1.1.0. We deploy Nimbus and Zookeeper for Apache Storm cluster on a desktop machine. We use Raspberry PIs as the processing nodes (Storm workers) where the application topologies are executed.

In Apache Storm, the data flows between different components of the topologies in the form of tuples where each tuple is a list of objects. We use Java's `Instrumentation` interface to approximate the memory consumed by each object in a tuple. We provide the `jar` file for the `Instrumentation` agent to the `worker.childopts` option in Storm's configuration file (`storm.yaml`) so that the agent instance can be loaded when the Storm *worker* is launched at the Raspberry Pi processing node. The rate of data flow from each component is calculated by multiplying the throughput of the component with the estimated size of the tuple object emitted by that component.

3.3.2 Results & Comparison

The figures 3.2 - 3.5 show the topologies for the RIoTBench applications along with the rate at which data is moving between the different components of these topologies. We make one observation that is consistent across all the RIoT topologies that the rate of data flow is relatively similar between the components of these RIoT topologies i.e. neither of the components in these topologies significantly increases or decreases the tuple size (and the data flow rate).

The figure 3.6 shows the rate of data flow through the Footstep Impact Localization (FSL) topology. We can observe a characteristic of the FSL topology that is different from the RIoTBench topologies shown earlier. We can observe that data flow is not uniform in this

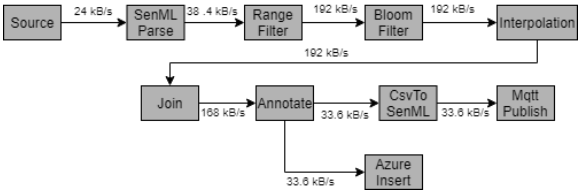


Figure 3.2: Rate of Data flow through `ETL` topology (Input Rate of 300 events/sec)

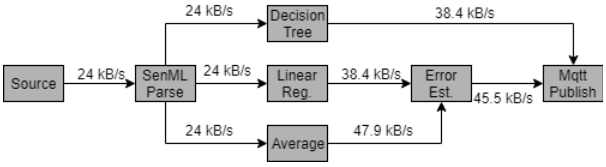


Figure 3.3: Rate of Data flow through `PRED` topology (Input rate of 300 events/sec)

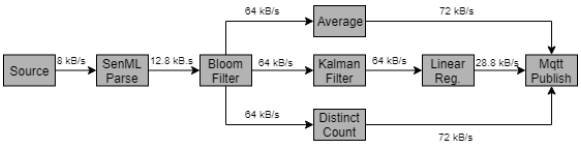


Figure 3.4: Rate of Data flow through `STAT` topology (Input rate of 100 events/sec)

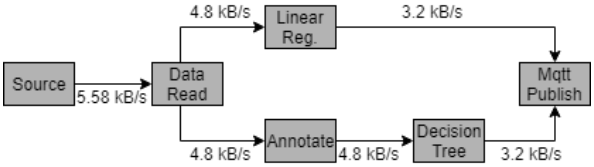


Figure 3.5: Rate of Data flow through `TRAIN` topology (Input rate of 50 events/sec)

topology. The experiment is conducted at the ingestion rate of 99 (9 corridors, with each sensor in the corridors emitting data at 8 kHz). After the first compute unit which down-samples the data from different sensors, the size of the tuples and the rate of data flow reduces significantly. Since, the aggregator compute unit aggregates results from all sensors in a corridor, the tuple size and the data size per second emitted by the tuple also increases.

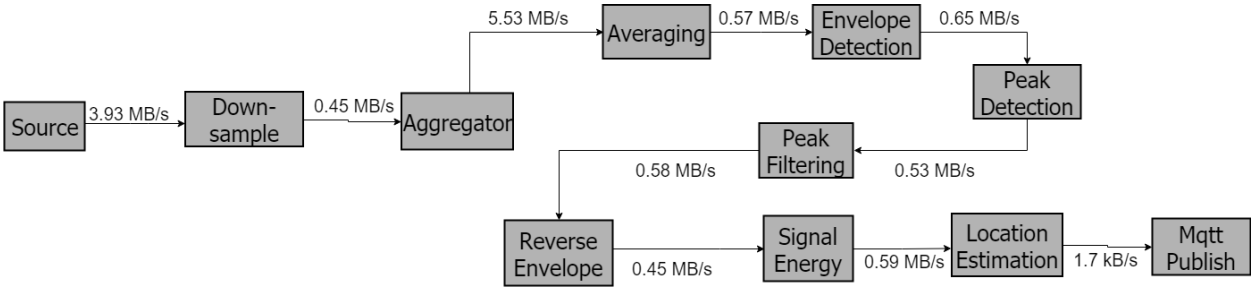


Figure 3.6: Rate of Data Flow through (`FSL`) Topology (Measured at input rate of 99)

3.4 Chapter Summary

In this chapter, we discussed a building occupant localization algorithm that can track occupants walking in an instrumented building's hallways. We adopted the algorithm to develop a real-time streaming application in Apache Storm. Other algorithms developed based on building vibration data can be similarly adapted to realize the futuristic smart buildings. This application can also be used with realistic workload to benchmark the performance of stream processing engines. Finally, we distinguish vibration data based applications considered here from some existing IoT benchmarking applications with regards to how data flows through these applications.

Chapter 4

Empirical Evaluation of Streaming Smart Building Applications

4.1 Introduction

The last decade has seen Cloud computing rise to become a popular computing paradigm to facilitate the growth of reliable, efficient and scalable applications that are ubiquitous today by providing Infrastructure as a Service (IaaS) for on-demand computing. Moreover, Cloud has been extremely successful in providing Software as a Service where entire application stack is managed by the vendors and clients. More recently, we have seen the emergence of Internet of Things (IoT) as a transformative force in various domains such as Smart buildings, Smart cities, Transportation, Healthcare and Industrial Manufacturing [36, 51]. The IoT applications have also tried to leverage this power of the Cloud by connecting everyday devices and sensors to the services hosted at the Cloud [12, 40, 77, 79]. In such a deployment model, the IoT devices at the infrastructure's edge gather data from various sensor devices and send it to the application(s) hosted in the Cloud. After processing the received data, the Cloud applications can send the processed results to monitoring services to gather actionable insights or back to the IoT devices at the Edge in order to trigger an actuator's state. However, connecting these IoT devices to Cloud for all kinds of services can raise certain challenges. The IoT devices, today, are generating huge volumes of data

and this data volume is growing at an exponential rate[4]. Sending all this data to Cloud for computation can cause challenges such as network congestion at the Edge and increased processing latency and thus, adversely impacts application performance.

In order to counter these challenges and sustain the projected data requirements of IoT, the community has proposed to utilize the Gateway devices available at the Edge to perform some or all the data analysis; and thereby moving the frontier of computation and services from the network core, the Cloud [24], to its Edge [43, 75], where the Things and Gateways reside.

Moving this computation to the Edge can be significantly beneficial for IoT applications' performance as this would enable near real-time data analysis, reduced network traffic to the data centers and reduced latency to triggering the actuator's state at the Edge. The emergence of Edge computing paradigm has made it critically important to empirically understand and evaluate the potential performance benefits that can be achieved by performing IoT data analysis at the Edge. With this in mind, this chapter focuses on performing this empirical examination of Edge and Cloud computing for an emerging class of IoT applications; Smart buildings.

4.1.1 Problem Statement

In this chapter, our objective is to perform an empirical examination of Edge and Cloud computing paradigms for Smart building related streaming IoT applications. We try to empirically understand the potential performance benefits achievable by processing these IoT applications at the Edge. We discuss our Edge architecture and choice of processing engine, and use realistic IoT benchmarking applications from chapter 3 and RIoTBench to conduct this evaluation.

4.1.2 Contributions

We present the following contributions of our work in this chapter: (a) We discuss our implementation the Edge IoT system for a Smart building setting; comprising of a cluster of 12 Raspberry Pi IoT devices, that we use to study the benefits of Edge computing. (b) We present our empirical examination the advantages of Edge computing against Cloud for relatively stable IoT scenarios such as Smart buildings. We simulate the scale of deployed applications by varying the rates at which sensors emit data. Our results demonstrate that for lower to medium scale deployments of these applications, the Edge devices can process sensor data at considerably low latency cost. The results further show that distributed Edge devices can be utilized to process large-scale deployments of such applications. (c) We show that application performance for large-scale deployment of RIoTBench [76] applications is bottlenecked by computation cost and for `FSL`(§3.2) it is bottlenecked by the network bandwidth. We deploy hybrid processing for `FSL` so that data pre-processing reduces data sent over the network and alleviate the network bottleneck to improve performance.

4.2 Experimental Methodology

In this section, we discuss the methodology we used for conducting the experiments to evaluate the performance of Edge, Hybrid and Cloud schemes for various IoT applications. First, we discuss the choice of processing engine that can execute the IoT applications in a real-time manner both on single-nodes or in a distributed architecture. Next, we discuss the IoT benchmark applications we considered for our evaluation and finding an optimal physical topology for them. Finally, we discuss the metric collection mechanism we employed for our evaluations.

4.2.1 Processing Engine

A key component for the architecture we've discussed is the choice of a processing engine to analyze the IoT data. With an increased availability of sensing devices, we are witnessing an explosion in the volume and rate of data generation. For an efficient and easier way of life, as we continue to engineer always-on, smart devices, it is easy to imagine an exponential increase in generated data [4]. Since the IoT devices continuous, unbounded streams of data, the engine must be able to process this data efficiently and reliably in real-time. Distributed processing and scalability of the engine are also critical since a single node at the Edge or a VM in the Cloud might not be able to handle data at a very high velocity. The considered architecture also requires the engine to integrate easily with various messaging systems. Furthermore, a good processing engine must also provide a framework that allows flexibility and freedom in developing a variety of IoT applications.

Several open-source stream processing engines (SPE) have been proposed for real-time processing of streaming data in an efficient and scalable manner. Most of these frameworks use a dataflow-based programming model where user-defined operators are connected by queues in a pipelined manner. Once connected, these operators represent a directed acyclic graph (DAG) that represents the flow of incoming streams of data. Among the modern SPEs, Apache Storm is the most popular framework for data science [26]. In this work, we use Apache Storm as the processing engine for both the Edge and Cloud computation.

Apache Storm

Apache Storm [20] is a distributed real-time stream processing engine for processing high-velocity, unbounded streams of data. Storm allows developers to create their applications as a *Topology* which is, essentially, a Directed Acyclic Graph (*DAG*). A *Topology* comprises

of *Spouts*; the data sources that either generate data or read data from external sources and emit to the topology for processing, and *Bolts*; the processing units that process the data. Storm allows users to specify how the data flows through the topology. A unit of data that flows between compute units is called a *Tuple* which is an ordered list of values.

Apache Storm employs a *Master-Slave* architecture where the master node (*Nimbus*) schedules and distributes the topology components to run on the Slave nodes (*Workers*). Each slave node in Storm runs a *Supervisor* daemon. *Nimbus* coordinates with Apache ZooKeeper [21] to manage the cluster state by monitoring cluster health via heartbeat messages, keeping track of topology execution and relocate any compute units in case of any slave node going down.

Before a Storm Topology is deployed into a cluster, the developers needs to statically determine the parallelism for each spout and bolt to improve the performance. The parallelism determines how many threads of execution will be assigned to this spout or bolt.

4.2.2 Benchmarks

For our experiments, we used the RIoTBench benchmark suite [76], a real-time IoT stream processing benchmark implemented for Apache Storm. The RIoTBench provides various IoT tasks as microbenchmarks. Each microbenchmark is developed as a single bolt topology where the bolt implements the task logic which is fed data-stream from the topology's spout. Besides these microbenchmarks, RIoTBench also provides IoT applications that perform various analyses on real-world sensor data-sets. The data-set we use for our experiments is the Smart Cities data stream [29] which includes data from various sensors such as outdoor temperature, humidity, light, dust and air quality. The data is collected from seven cities for two months.

The microbenchmarks provided by RIoTBench are not suitable for our experiments as we are interested in realistic IoT applications. Therefore, we only use the IoT applications from the suite. The logical topologies for RIoTBench's applications are shown in figures 3.2, 3.3, 3.4 and 3.5. The figures also show the rate of data flow through these topologies at certain input rates.

For our experiments, we also made various modifications to the RIoTBench benchmarks such as: (1) We patched various bugs and inefficiencies. (2) Replaced any Cloud-based services with lab-based ones; (3) To enable a controlled experiment, we implemented a timer-based input generator that reads data from a replayed trace at a configurable input rate.

Besides RIoTBench applications, we also use the Footstep Impact Localization application discussed in §3.2 to compare Edge, Cloud and Hybrid schemes.

4.2.3 Messaging Middleware

Owing to their different capabilities, IoT devices can use different communication mechanisms and protocols to communicate with each other and with the Internet. If the sensor nodes do not need to be mobile, they can be hardwired and connected to local Gateway devices via Ethernet such as in the smart buildings scenario discussed in chapter 3. On the other hand, mobile nodes would need to communicate via wireless protocols such as Bluetooth Low Energy (BLE), WiFi or Zigbee [57] (based on IEEE802.15.4). The Gateway devices, in turn, use various messaging middleware to communicate with each other and with the Cloud. To establish communication between sensing devices, Gateway nodes and the Cloud, a messaging middleware is required. In this work, we consider the popular IoT protocols such as AMQP [1] and MQTT [6] for moving data between these devices. We use RabbitMq [8] as the message broker for AMQP's implementation. We chose RabbitMq be-

cause it's one of the standard, lightweight implementations available for the AMQP protocol and can be used without requiring disk space for message routing (if message persistence is not needed) [39]. We also use MQTT broker to receive the processed results. The actuator nodes subscribe to and receive these processed results.

4.2.4 Tuning Storm Topologies

Measuring Operator Utilization

As we have discussed in §4.2.1, Apache Storm employs the dataflow programming model where the application is structured as a directed acyclic graph with nodes representing the bolts (compute units) and edges representing the flow of data between them. Each bolt has an associated input queue. For each tuple in the bolt's queue, the task logic is executed by the worker thread(s) and the resulting tuple is moved to the queue of the next bolt in the pipeline. Considering this, Storm topologies can also be viewed as a queuing network [46, 58] - a directed acyclic graph of stations where Widgets (tuples) enter the network via the queue of the first station. Once the widget reaches the front of a station's queue, a server (worker) operates on it, and then it advances to the next station.

In Queuing Theory [46, 58], server utilization (fraction of time a server is busy) is modeled as comprising of two factors: (a) the input rate λ to the server and (b) the service rate μ for each input. For a server i, the server utilization ρ_i is defined to be

$$\rho_i = \frac{\lambda_i}{\mu_i} \tag{4.1}$$

A queuing network is considered to be stable when $\rho_i < 1$ for all servers i. If there is a server to which widgets arrive more quickly than they are serviced, the queue of that server will grow

unbounded and such a server will become the bottleneck for the network's performance. To alleviate this bottleneck for the network, the input rate λ_i has to be decreased or the service rate μ_i needs to be increased. In order to maximize the performance of such a network, the service rate for the server i can be increased by adding multiple parallel units to process the incoming widgets.

In Apache Storm, bolt utilization can be estimated by using a metric called *Capacity*. During the execution of the topology, this metric is periodically calculated as:

$$Capacity = \frac{Execute\_Count \times Execute\_Latency}{Time} \tag{4.2}$$

In equation 4.2, *Execute_Count* represents the number of tuples executed by a specific bolt in a given time window, *Execute_Latency* is the per tuple average computation latency for the bolt and *Time* represents the window duration during which *Execute_Count* and *Execute_Latency* metrics are collected. The fraction, $\frac{Execute\_Count}{Time}$ is equivalent to λ, since throughput is equal to input rate for a balanced queue; and *Execute_Latency* is equivalent to $\frac{1}{\mu}$. The metric *Capacity* can therefore be used to represent the utilization for each bolt to find the bottleneck of a Storm topology.

Tuning Methodology

The topologies we have considered for our experiments have the characteristic of different computation cost and varying input rates at different nodes. At certain input rates that stress the topology sufficiently, this leads to an imbalanced topology where the queues of the bottleneck bolt exceed the acceptable threshold; thus triggering the backpressure and reducing the inflow to the topology and affecting performance. Given such logical topologies, in order to get the best use of available hardware resources, we need to alleviate any per-

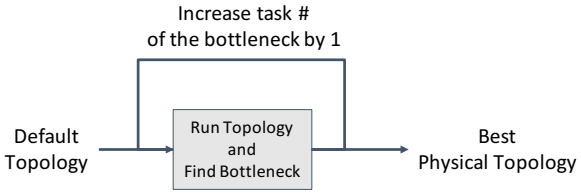


Figure 4.1: Iterative tuning of Topologies

formance bottlenecks in the topology while also not over-burdening the system with worker processes or executors. Therefore, we tried to find an "optimal" configuration for each logical topology before conducting our experiments.

In Apache Storm, various configuration parameters exist that can impact the application performance. Primarily, these parameters include the number of Workers (JVM processes that run the topology) and number of parallel executors for spouts and bolts. Fine-tuning can be done by adjusting the input and output queue lengths for spouts and bolts or the backpressure triggering thresholds.

For our experiments we limit the number of Workers to the number of available physical nodes in order to avoid the overhead of inter-worker communication (via Netty). After that, we only focus on adjusting the number of parallel bolt executors to find a better performing physical topology. Increasing the bolt's executor threads can have the impact of reducing server utilization by increasing the service rate and thereby leading to a balanced network. Therefore, we adopted a simple iterative approach to achieve a balanced topology.

Tuner's Work Flow

To begin with, we identify a range of input rates for each topology - from reasonably low to sufficiently high rates such that reasonable performance cannot be achieved with the underlying hardware. For each selected input rate within the range, we assign an initial

bolt instance vector to the topology, with all instances set to 1 (no parallel executors). We initially submit these topologies to the Storm cluster and allow them to run for a fixed duration. During the execution, we collect heuristics to identify the bottleneck node in each topology. Guided by this identification, we increase the parallelism for the bottleneck node by 1 and run the again topology with the new parallelism assignment. This procedure is repeated iteratively. The goal of this iterative procedure is to find a balanced topology that can achieve good performance using the given resources. Therefore, during execution, we monitor variance in the bolts' capacities to estimate how unbalanced the network is. We also monitor the throughput and latency performance of the executed topolgoy.

Certain termination conditions are needed to stop this iterative procedure. We can stop this procedure when (a) the difference between bolts' variances during two iterations is less than a certain threshold T_{vDiff} or (b) none of the bolts is the topology bottleneck i.e. the *Capacity* for all the bolts is less than a certain threshold. We repeat this procedure until the performance difference between 2 parallelism assignment is less than a predefined threshold T_{cap}. For our purposes, we only terminate if all bolts have capacities below T_{cap} and ignore (a) to further observe the behavior. Instead, we limit on the number of iterations to terminate the procedure's execution.

At the end of this procedure, we have a collection of topologies; explored for each selected input rate. We run each unique physical topology with the selected input rates to observe their throughput-latency performance. From analyzing the throughput-latency curves, we choose the topologies that achieve high throughput under a given latency threshold as the candidate physical topologies.

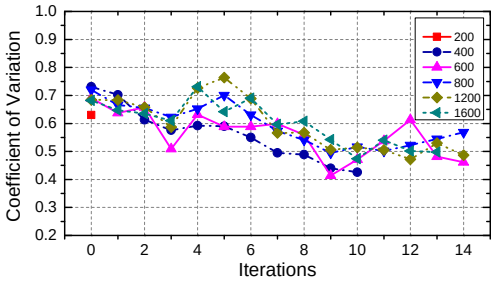


Figure 4.2: Decrease in Coefficient of Variation for `ETL` Topology across iterations

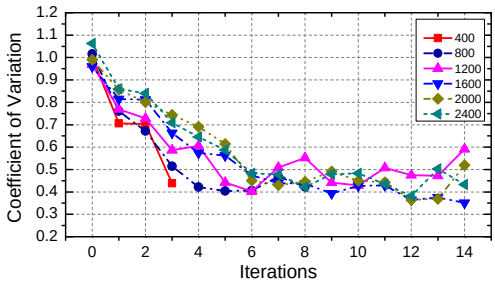


Figure 4.3: Decrease in Coefficient of Variation for `PRED` Topology across iterations

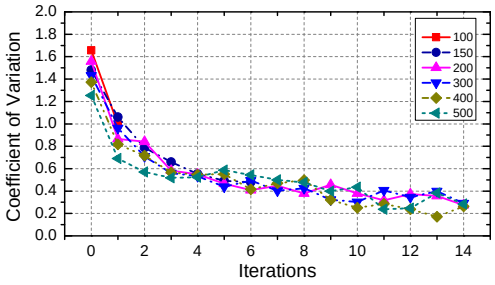


Figure 4.4: Decrease in Coefficient of Variation for `STAT` Topology across iterations

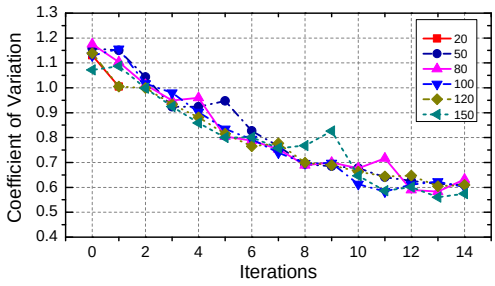
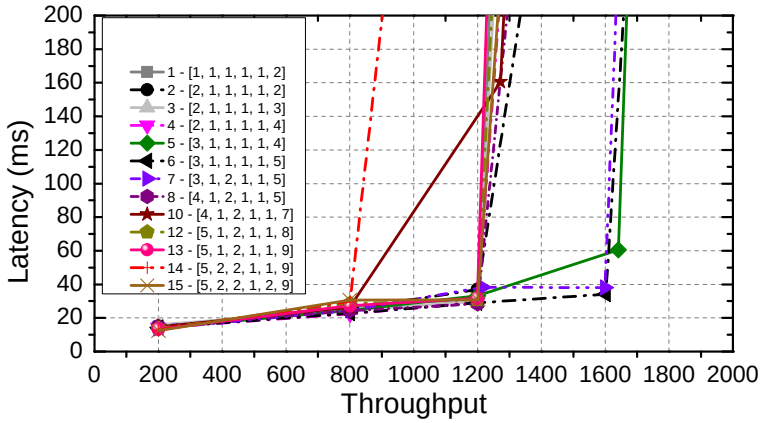


Figure 4.5: Decrease in Coefficient of Variation for `TRAIN` Topology across iterations

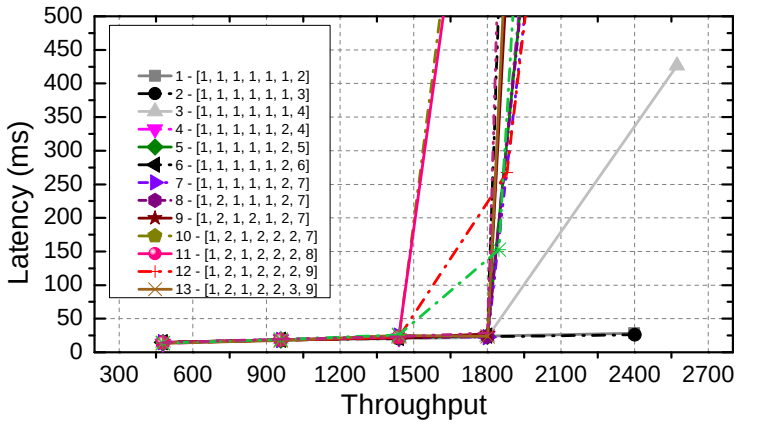
Variance in Bolt Capacity

We argued previously that balancing effective server (operation) utilization would yield gains in topology's performance. To measure the extent to which this tuning procedure balances server utilization, we calculated the windowed utilization ρ_w of each operation and then computed the coefficient of variation ($CV = \frac{stddev}{avg}$) of this vector. A lower utilization CV means more balanced server utilization.

The figures 4.2 - 4.5 plot the coefficient of variation for Storm across different iterations for selected input rates. The legend shows these input rates for each topology. We can see from these results that this tuning decreases the CV in the first few iterations; yielding a relatively balanced network. From then onward, we can observe that decrease is either not



(a)



(b)

Figure 4.6: Throughput-Latency curves for explored physical topologies (a) `PRED` (b) `STAT`

significant or in the case of `ETL` adversely impacts the balance. Some input rates only yield a small number of iterations because these lower input rate do not stress the topology enough and thus the network is already balanced; eliminating the need for tuning.

The figures 4.6a and 4.6b show the throughput-latency curves for different physical topologies `PRED` and `STAT` applications. The legend in the figures show the iteration for which these topologies were observed and the bolt instance vector. We can observe that some topologies can offer higher throughput at lower latency cost while others suffer from the bottleneck

at lower throughput. Here, we can observe a correlation between the topologies offering better performance, and improved balancing as shown by figures 4.3 and 4.4 for `PRED` and `STAT` topologies. The physical topologies offering "optimal" performance can be seen as corresponding to iterations where the coefficient of variance in bolts' capacities has reduced significantly and before the CV curve tapers off.

We can observe that this tuning procedure did not need to go through a large number of iterations to discover topologies that give reasonable performance. However, this is strictly dependant on the available hardware resources. Here, we only needed a few iterations because the topologies were executed on resource constrained Raspberry Pi devices. A large number of bolt instances would strain these constrained devices; causing further resource contention and extensive context switching between bolt executors. However, for a Storm cluster where *Supervisor* nodes have Cloud-class resources, this tuning methodology would need to be modified to have a different initial bolt instance vector. Furthermore, a unit increase in bolt instance of the bottleneck might prove to be prohibitively slow for the tuning procedure.

4.2.5 Metrics Measurement

The key performance metrics that we target for this study are throughput and end-to-end latency for the duration of the application. As has been discussed, our target application scenario is Stream Processing of IoT applications where unbounded data-streams are processed indefinitely. However, for the purposes of our experiments, we need limit the duration of each application run. Therefore, we run each experiment for 5 minutes and take measurements during the one minute of steady-state runs, after discarding the first two minutes of initial phase.

We measure *throughput* at the completion of execution by counting the number of tuples

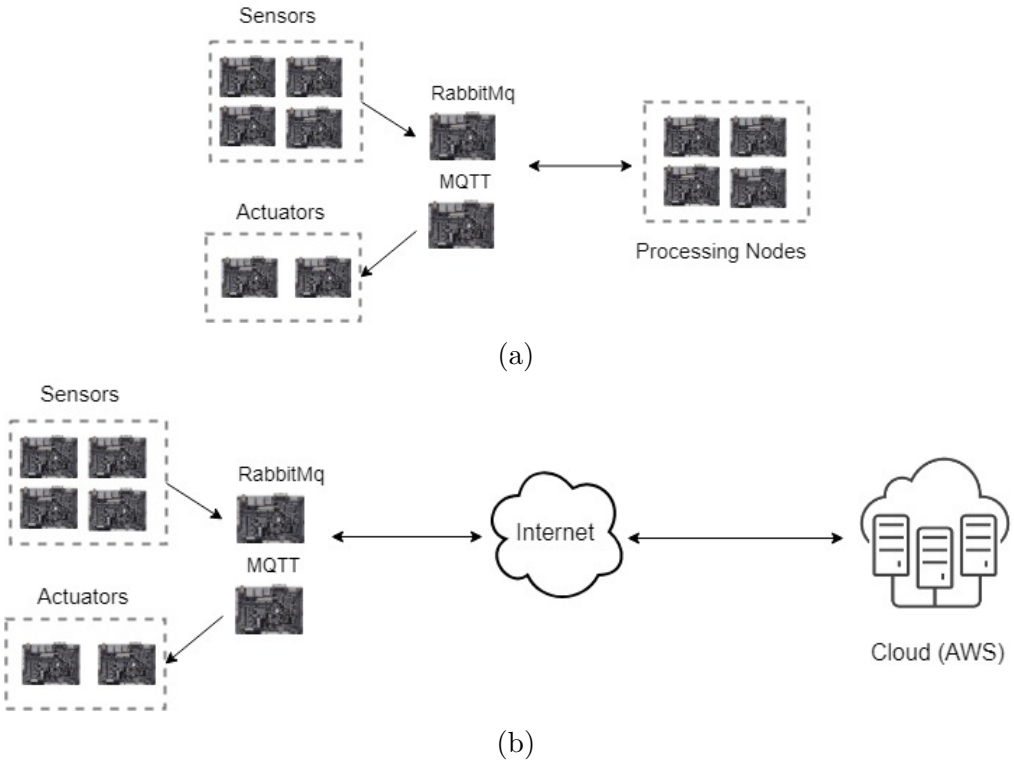


Figure 4.7: Experimental Setting for Edge and Cloud deployment (a) Processing Engine (Apache Storm) runs on the Raspberry Pi nodes deployed in the lab to realize our Edge model (b) Processing Engine runs on the EC2 instances in the AWS data center

that reach the `MQTT Publish` sink. Measuring throughput at the sink can result in different throughput rates for each topology at a given input rate, since different topologies have different *selectivity ratio* (input-output tuple ratios): e.g., 1:2 in `PRED`, 1:10 in `STATS`. We measured *latency* by sampling 5% of the tuples, assigning each tuple a unique ID and comparing timestamps at source and the same sink used for the throughput measurement. For each configuration, we performed each experiment 5 times and report the average. The error bars, where added, indicate one standard deviation from the average.

4.2.6 Experimental Setup

Recall that, we consider Edge IoT Gateway devices to have intermediate-class computing resources; few-core processors and little memory. Therefore, for this empirical study, we use the most common and popular single-board devices representative of this class. Specifically, we use Raspberry Pi 3 Model B devices [9], which have a 1.2GHz quad-core ARM Cortex-A53 with 1GB of RAM and install Raspbian GNU/Linux 8.0 v4.1.18 on them. We set up an infrastructure comprising of a cluster of 12 Raspberry Pi 3s connected on a LAN via Netgear switch through Ethernet cables.

For our experiments, we run our synthetic data generators on Raspberry Pi devices to simulate sensor nodes. On separate Raspberry Pi nodes, we deploy the message queue brokers to distribute sensor data to the consumer processing devices (Raspberry Pi nodes for Edge deployment and `AWS` data center for execution on Cloud) or send processed results to actuator or monitoring nodes.

For Edge experiments, we run Apache Storm *Supervisors* on the Raspberry Pi nodes to process streaming data. For Cloud experiments, we deploy Apache Storm *Supervisor* on EC2 instances in AWS data centers. We chose AWS-East for our deployments as it offers the minimum latency from our lab set up. Choosing the closest data-center is a reasonable choice for Smart Building or Smart City applications as the location of sensor nodes will not change. We chose t2.micro EC2 instances as they represent one particular Cloud setting for this study. We deploy the Storm *Nimbus* and *Apache Zookeeper* on a local desktop to coordinate the Storm worker nodes. All the machines used in our experiments are synchronized using Network Time Protocol [65].

The Figure 4.7a shows our realization of the Edge model that we will use to conduct our Edge experiments and Figure 4.7b shows the setting we use for our Cloud experiments.

4.3 Results and Discussion

This section presents the results of our experiments for this comparative study. First, discuss the performance of the RIoTBench applications by comparing their throughput and latency performance for the Edge and Cloud schemes on single-node at the Edge against a single VM in the Cloud. Then, we study the impact of data prefetching by the consumers from the Message Queues and discuss how that impacts the performance for single and multiple consumer cases. After that, we study the scalability of using distributed Gateways for the Edge processing at the Edge and compare this performance against the Cloud scheme. Finally, we consider the Footstep Impact Localization application presented in §3.2 for this performance study. We also employ a hybrid mechanism by using Edge and Cloud cooperatively and discuss how that impacts the application's performance.

4.3.1 Throughput & Latency Performance of RIoT applications

In this section, we present and discuss the results of the experiments where the RIoTBench applications were processed on single nodes at the Edge and compare it against processing on the Cloud. As discussed, the Cloud experiments use a single AWS EC2 instance for experiments.

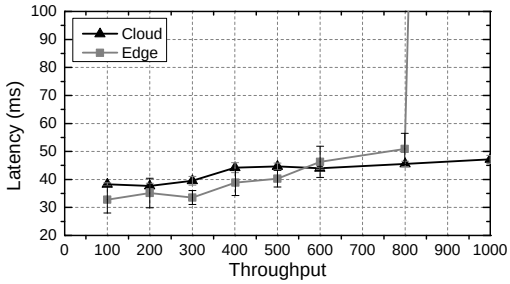
We measure the throughput-latency performance curve for each of the RIoTBench applications on a Raspberry Pi processing node across a range of input rates. The throughput-latency curves for `ETL`, `PRED`, `STATS`, and `TRAIN` are shown in figures 4.8a, 4.8c, 4.8e and 4.8g respectively. Observing these, we can see that Edge and Cloud offer different throughput-latency performance for the RIoTBench topologies. An important thing to observe is the different latency behavior at different throughput rates (and consequently input rates as input and throughput are linearly related).

Based on the latency performance, the input rates for these topologies can be roughly split in to three ranges - low, medium and high. At lower input rate ranges, Edge offers better latency performance for all the applications; for `PRED`, `STAT`, `ETL` and `TRAIN` the Cloud offers 59%, 90%, 15% and 40% higher latency compared to Edge. As the input rates increase, this difference becomes less pronounced. At considerably higher input rates, the latency for the topologies grows exponentially in the throughput-latency curves as the Raspberry Pi Edge node is no longer able to process the data at increased input rates because of frequent triggering of backpressure. The cut-off point where Edge stops offering the better performance varies based on application characteristics. For example, we can observe that for `TRAIN` topology, Edge can only be suitable for processing data at very low input rates. However, if data needs to be ingested at higher rates, a single Edge node becomes prohibitive in achieving good performance. All in all, the key observation from these experiments is that the application and ingestion rates determine where it is best to process the IoT data.

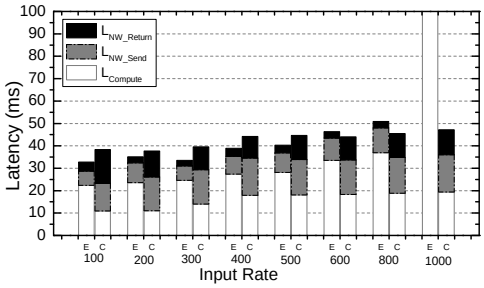
4.3.2 Latency Breakdown for RIoT Topologies

Recall that our model for IoT Applications considers the total processing latency of sensor data as from the time data is emitted by a sensor device to the time when the data is received back at some actuator or monitoring unit. The total processing latency, therefore, comprises of: (a) Deliver Latency (L_{NWSend}): Time from when the data is emitted by the sensor to the time when it is received at the processing node, (b) Compute Latency ($L_{Compute}$): Time taken to process the data by the streaming engine and (c) Return Latency ($L_{NWReturn}$): Time from transmitting the processed result to the time when the result is received at some actuator or a monitoring unit monitoring unit).

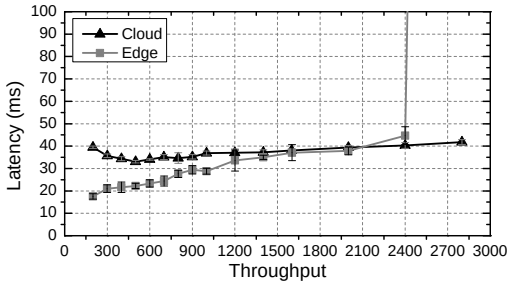
The figures 4.8b, 4.8d, 4.8f 4.8h provide the breakdown of the processing latency for the



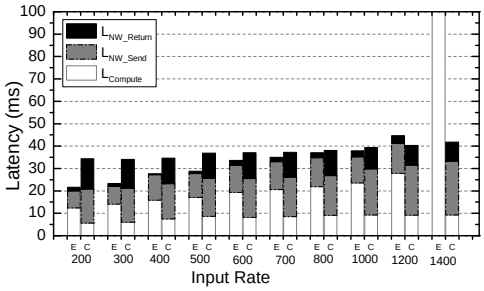
(a) ETL - Throughput vs Latency



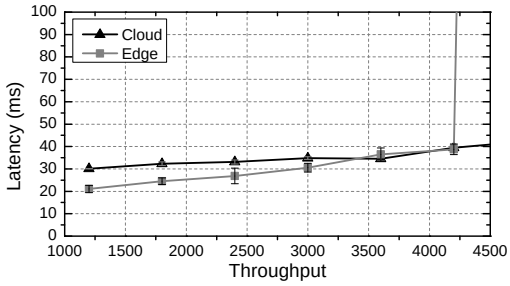
(b) ETL - Latency breakdown



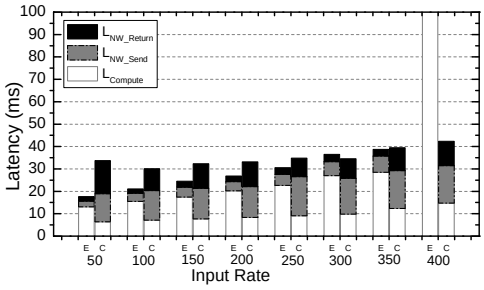
(c) PRED - Throughput vs Latency



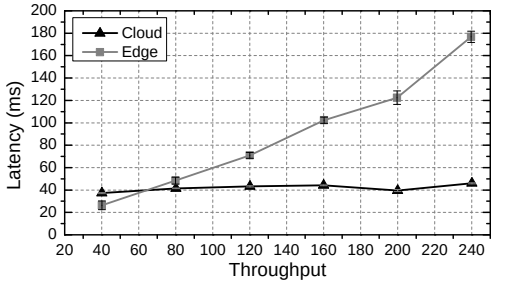
(d) PRED - Latency breakdown



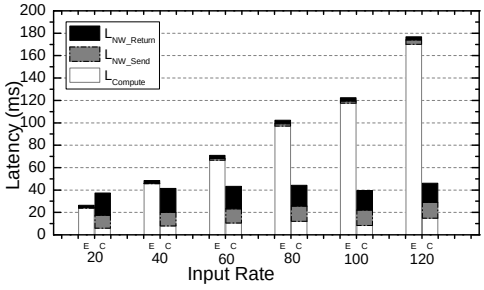
(e) STAT - Throughput vs Latency



(f) STAT - Latency breakdown



(g) TRAIN - Throughput vs Latency



(h) TRAIN - Latency breakdown

Figure 4.8: Performance of `ETL`, `PRED`, `STAT` and `TRAIN` application topologies on single-node Edge & single node Cloud. Input Rate and Throughput are linearly related (scaled by Selectivity ratio for the application) until the exponential blow-up of computation latency. (a), (c), (e), (g) show Throughput vs end-to-end Latency curves for all topologies (b), (d), (f), (g) show the Latency breakdown into its constituting components ($L_{Compute}, L_{NWSend}$ and $L_{NWReturn}$)for different ingestion rates

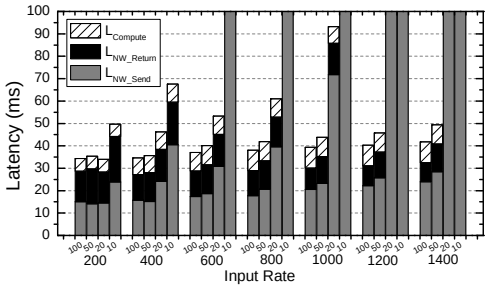
RIoTBench applications into its constituting components. From these, we can observe that at very high input rates, the exponential increase in latency for `ETL`, `PRED` and `STAT` applications is caused by the increased computation latency as the Edge processing node is unable to process data at such high rates because the backpressure is triggered frequently.

The benefit offered by the relatively powerful EC2 instance is observable as there is more than 2x decrease in computation latency when topologies are executed in the Cloud VM. However, this benefit is offset by the high network overhead in moving the data to the Cloud and back to the actuators.

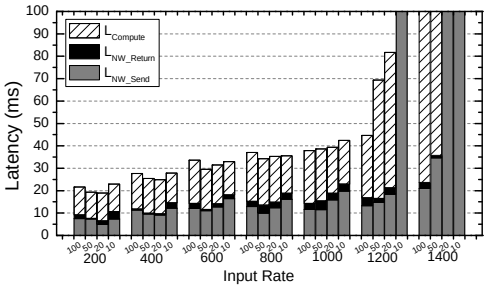
4.3.3 Sensitivity to Data Prefetching

This section discusses the impact of data prefetching from the Message queue (RabbitMq in our experiments) on application performance. In order to have guaranteed delivery to the processor nodes, the consumers need to acknowledge the message delivery or processing. When the consumers acknowledge the data reception to the broker, the broker can delete the acknowledged message. A lower prefetching value is needed for practical deployment of the applications in order to have balanced computing over multiple nodes with different processing capabilities and processing latency of data is important. In such cases, higher prefetching values can be counter-productive and it is recommended to have prefetching value in the range of 20-30 [27]. The experiments prefetching sensitivity were conducted for various prefetch values ranging from 1 to 1000. Here, we present the sensitivity results for values in the range from 10 to 100.

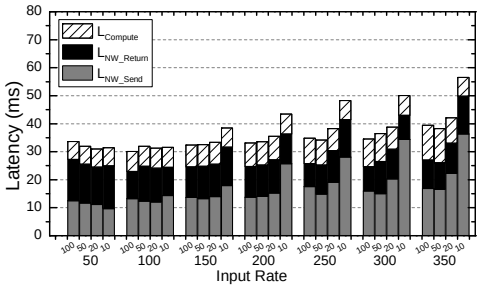
For execution on the Cloud, the network latency is a major contributing factor to application's latency performance. The figures 4.9a and 4.9c demonstrate the impact of different prefetch values on `PRED` and `STAT` topologies while executing them on the Cloud. Here, we



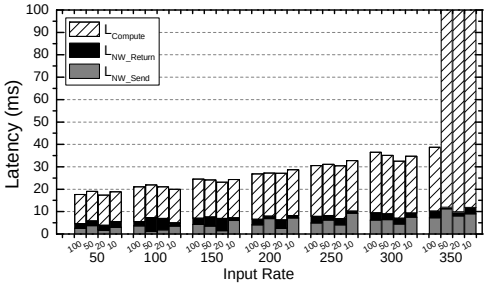
(a) PRED - Cloud computation



(b) PRED - Edge computation



(c) STAT - Cloud computation

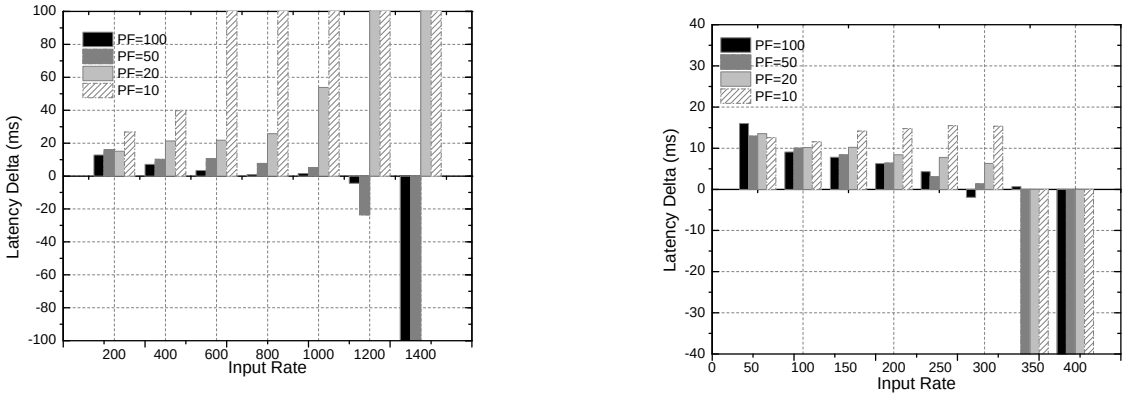


(d) STAT - Edge computation

Figure 4.9: Latency Sensitivity to Data Prefetching for `PRED` and `STAT` topologies (a), (c) show single-node Cloud computation for `PRED` and `STAT` topologies (b), (d) show single-node Edge computation for `PRED` and `STAT` topologies.

can observe the lower prefetching values adversely impacting the latency performance of the applications. This impact is largely driven by the comparatively greater network latency for processing on the Cloud. The impact on the `STAT` topology is not considerably high because the topology achieves the throughput of 4200 events/sec at comparatively lower input rates because of its higher selectivity ratio.

In comparison, from figures 4.9b and 4.9d we can observe that since Edge devices are closer to the sensor devices, network latency is a smaller contributing factor to total latency. The latency performance of these applications is not highly sensitive to data prefetching. The comparison also shows that at lower input rates, data prefetching has a smaller impact while at high input rates, it increases the total latency significantly.

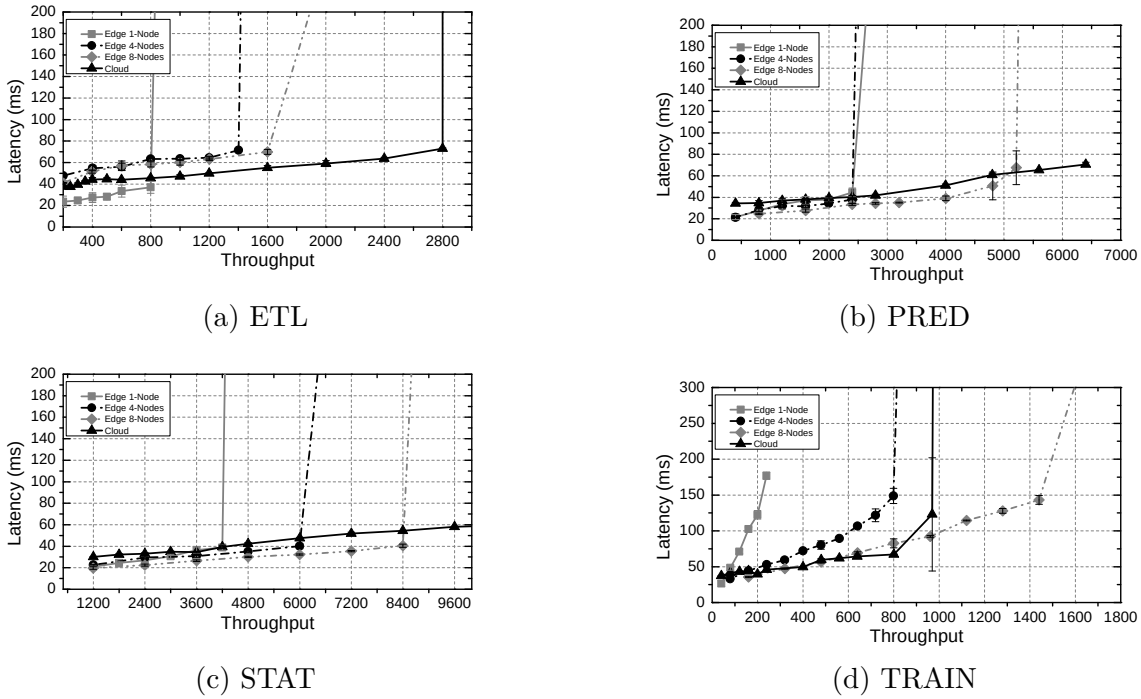


(a) PRED (b) STAT

Figure 4.10: Latency Delta ($L_{Cloud} - L_{Edge}$) demonstrating the difference in end-to-end latency of `PRED` and `STAT` topologies for prefetch values of 10, 20, 50, 100

The Figure 4.10 shows the end-to-end latency delta $L_{Cloud} - L_{Edge}$ for `PRED` and `STAT` topologies across different input rates for different prefetching values. For `PRED` topology, we can observe that for higher prefetching values, the Latency delta $L_{\Delta} = L_{Cloud} - L_{Edge}$ decreases with input rates; while for lower prefetching values, the L_{Δ} is increasing. This is because when prefetching value is larger, L_{NWSend} is not a significant contributing factor to L_{Cloud} and thus L_{Cloud} does not increase significantly; while L_{Edge} increases because of higher computation cost at high input rates. The net effect of this is that the L_{Δ} decreases for higher while increases for lower prefetching values. This suggests that Cloud can be a better choice for processing with higher input rates but with lower prefetching values (needed for balanced computing over heterogeneous compute nodes), Edge can be a better choice for processing.

For `STAT` topology, we can observe that the latency L_{Δ} is negative at high input rates for all prefetch values. This is because the input rates are not so high as to make L_{NWSend} a major contributing factor to the end-to-end latency while the Raspberry Pi node at the Edge is unable to handle these input rates and L_Edge increases significantly. The results for `PRED` and `TRAIN` topologies are similar to those of `ETL` and `STAT`.



(a) ETL (b) PRED

(c) STAT (d) TRAIN

Figure 4.11: Throughput vs end-to-end Latency performance on distributed Edge (1, 4 and 8 nodes) against single-node `AWS` VM in Cloud. (a), (b), (c) and (d) show the performance for `ETL`, `PRED`, `STAT` and `TRAIN` topologies respectively.

4.3.4 Performance on Distributed Edge

The Edge or Fog architecture comprises of a number of Gateway devices at the Edge that connect the IoT devices at the Edge with each other and bridge them with the Cloud. IoT applications can certainly benefit from harnessing the processing power of the distributed Gateway devices available at the Edge.

This section focuses on the scalability benefits that can be obtained by distributed processing at the Edge and compares the gains against those achieved at the Cloud.

We deployed the `RIoTBench` applications across 4 and 8 Raspberry Pi nodes connected on an Ethernet lab network. We simply increased the physical operation instances of the topologies proportional to the number of Raspberry Pi nodes and assigned them uniformly across the

nodes.

The Figure 4.11 shows the throughput-latency curves for the `RIoTBench` topologies. We can observe that distributed deployment at the Edge can allow the topologies to handle higher ingestion rates under a given latency budget. Considering the end-to-end latency *SLA* requirement to be 100 ms, we can quantify the maximum throughput achievable under this latency budget. By using 8 distributed Edge nodes, we can observe that `PRED`, `ETL` and `STAT` achieve around 2x higher throughput compared to single node whereas for `TRAIN` the 8x more throughput is obtained.

For `TRAIN` topology, 8-node deployment can offer better throughput at lower end-to-end latency in comparison to the single-node Cloud while for the remaining topologies, single-node Cloud outperforms the distributed Edge deployment. We can also observe that, at lower input rates, the end-to-end latency for single-node Edge is less than for distributed execution as no networking cost is involved.

4.3.5 Edge Processing of Footstep Impact Localization

In this section, we focus on evaluating the performance of occupant's footstep localization application developed in §3.2 As has been discussed in chapter 3, to realize the futuristic smart buildings, real-time processing needs to be done on vibration data from accelerometers. Mathematical models developed for different applications require different sampling rates; for example, Gunshot Classification [55] requires sampling at 25.6 kHz, occupant impact localization [15] requires 1 kHz while modal analysis only requires sampling at 256 Hz. In a real-world setting, if data from same sensors needs to be used for all applications deployed in a buildings, the common denominator for sampling rate would be very large. It can be argued that sending this data sampled at very high rates to a centralized Cloud storage would strain

the network infrastructure and should best be processed locally at the Edge. Therefore, we consider Edge Computing as a processing scheme and compare it against Cloud processing.

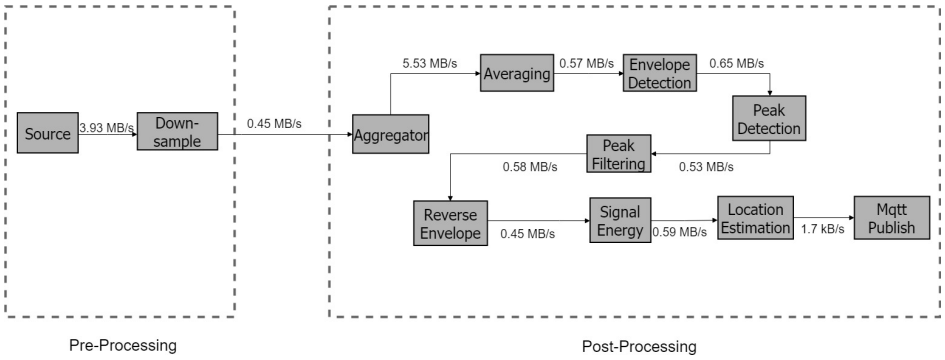


Figure 4.12: Splitting `FSL` topology to deploy pre-processing at Edge in order to limit the data sent over network to the Cloud

Besides, processing at the Edge & Cloud, we also consider a Hybrid processing model where Cloud-class computation resources in data centers can also be used for processing. Because of the non-uniform data flow through the impact localization topology, we split the topolgoy so that signal downsampling to the application's required sampling rate is handled at the Edge which would lead to a considerably lower data rate that needs to be sent to the Cloud. If reasonably complex computation is required for the application, processing in a hybrid manner can even be more beneficial compared to Edge.

In order to deploy Footstep Impact Localization in a hybrid manner, we create two *Storm topologies*. The first topology is responsible for handling the data generation and down-sampling the data to the application's required sampling rate (1 kHz). After down-sampling, this topology publishes the data to the RabbitMq message queue. The second phase of the processing happens at the *Cloud* where the second *topology* is deployed. The topology's *Spout* launches a RabbitMq consumer that handles the data from message queue. After timestamping the received data with arrival time, it pushes the data to the *spout's* queue. The spout consumes this data, creates and sends data *tuple* to the downstream bolt for

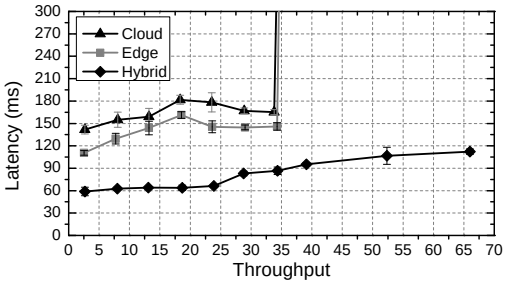


Figure 4.13: FSL Topology, Throughput vs Latency Comparison

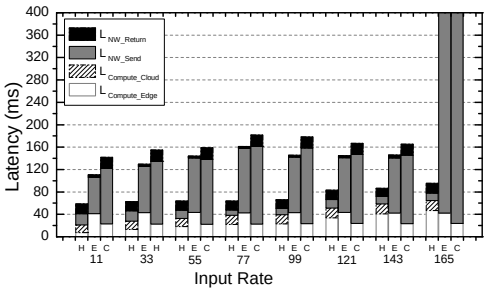


Figure 4.14: FSL Topology, Latency Breakdown

analysis. Processing data in this hybrid manner is advantageous because the data the data sent over the network is reduced by 8x and Cloud's powerful computation resources are being leveraged for processing.

Figure 4.13 shows the throughput-latency curve while the figure 4.14 shows the latency breakdown for the Impact Localization topolgoy. From 4.13, we can see that Hybrid deployment can offer higher throughput as it can ingest data at higher rates. This is elaborated by the latency breakdown in 4.14 that shows that the at higher ingestion rates, sending the large amounts of data over the network is becoming the bottleneck which is avoided by split processing in Hybrid execution.

Furthermore, at lower input rates, Hybrid scheme can process the generated data at lower latency. We can observe the maximum benefit at throughput of around 20 events/sec where Hybrid offers around 60% and 65% less latency compared to the Edge and Cloud execution. Mainly, this latency benefit is coming from the data sent to the Message Queue. The computation cost at the lowest input rates ($L_{ComputeEdge} + L_{ComputeCloud}$) is similar to the Edge computation cost and grows as the input rate increases. The $L_{ComputeCloud}$ cost remains relatively similar throughput while the increase is caused by the $L_{ComputeEdge}$ of latency. This cost is increasing as the synthetic data generator at the Raspberry Pi node along with the pre-processing *Storm* topology is generating memory pressure which triggers garbage

collection and thus affecting the performance. The cost for the downsampling component of pre-processing stays relatively similar across the input rates.

4.4 Chapter Summary

In this chapter, we attempted to empirically understand the end-to-end latency performance for IoT applications in the context of Edge and Cloud processing. We considered an Edge infrastructure where sensor devices are connected to resource-constrained IoT Gateway devices in a local network. These Gateway devices also provide a path to Cloud data centers where powerful processing resources are available. As IoT devices generate unbounded data streams, we employ a popular stream processing engine - Apache Storm - for processing of IoT sensor data. In our infrastructure, we consider message queue brokers as intermediary messaging middleware for moving data between sensor nodes, Gateway devices and Cloud. Our empirical study demonstrates that given resource-constrained Edge devices, application's performance characteristics and data ingestion rates are important factor in determining whether the applications should be processed at the Edge nodes or the Cloud. At lower ingestion rates, the representative Edge devices can process the data at lower networking cost without overwhelming the deployed streaming engine. At higher ingestion rates, however, the representative Edge devices can no longer process the data under the given service-level-agreement (SLA) for latency and it could be best to utilize Cloud data centers for processing. We also study that the availability of multiple gateway nodes at the Edge can be effectively utilized to handle higher ingestion rates at lower latency than Cloud.

Chapter 5

Conclusion & Future Work

5.1 Summary

The key goal in this thesis is to perform an empirical evaluation of the potential benefits achievable from Edge Computing and to develop a streaming application for impact localization of human footsteps. We discussed an impact localization algorithm implemented for vibration data in smart buildings that can localize footstep impacts and help track occupants in a smart building. We adapted this to develop a real-time streaming version of the application that lays the foundation for further work on similar smart building applications and help realize the futuristic smart buildings. Furthermore, since there is a lack of realistic streaming benchmarks in the IoT domain, this real-time streaming application can also be used as a realistic workload to benchmark the performance of stream processing engines.

We also performed empirical evaluation to understand the end-to-end latency performance for IoT applications in the context of Edge and Cloud processing. We considered an Edge infrastructure of resource-constrained Gateway devices; represented by Raspberry Pi devices, that are connected to IoT sensor nodes over a local network. At the Edge, we use Gateway nodes are used to process data.The Gateway devices also provide a route to the Cloud data centers where VMs deployed in the Cloud are used to process data in the Cloud scheme. We employ Apache Storm as the streaming engine to process real time streams of IoT sensor data. In order to move data from sensor nodes to the processing nodes, we use message

queue brokers as intermediary messaging middleware. This empirical study demonstrates that for Edge devices with limited resources, application's performance characteristics and data ingestion rates are important factor in determining whether the applications should be processed at the Edge nodes or the Cloud. At lower ingestion rates, the representative Edge devices can process the data at lower networking cost without overwhelming the deployed streaming engine. At higher ingestion rates, however, the representative Edge devices can no longer process the data under the given latency SLA and it's best to utilize Cloud data centers for processing. We also study that multiple gateway nodes at the Edge can also be utilized to handle higher ingestion rates at lower latency than Cloud.

5.2 Future Directions

Various avenues exist that can be explored to improve and advance the work presented in this thesis. These range from the deployment of Edge architecture in the VT smart building to a more detailed empirical study.

The creation of the Footstep Impact Localization as a streaming application as discussed in Section 3.2 can allow online processing of data in smart buildings. This is an important contribution of this work and one of the important works that can be done in the future is to develop streaming versions of other applications based on vibration sensor data collected from the instrumented Goodwin Hall [25, 55, 72]. Streaming versions of these applications could be practically deployed to lay the foundations of futuristic smart buildings. Furthermore, an Edge architecture can be deployed in "Goodwin Hall" to process vibration data in real time with lower latency budget. Moreover, this deployment can also serve as a test-bed to study Edge and Fog architecture for smart buildings and can prove to be a useful avenue for further research. These applications could also be used to understand and benchmark the

performance of the stream processing engines and Edge frameworks.

The empirical evaluation conducted in this work is focused on the Edge architecture that is relevant to Smart building scenarios; sensor and gateway nodes are stationary & physically connected. Furthermore, the representative gateway nodes used in this are homogeneous (Raspberry Pi 3 nodes). One possible extension to this work could be to consider applications & workloads that might require heterogeneous gateway nodes e.g. IoT applications that can benefit from using GPUs. Furthermore, in this work, we considered Gateway nodes to be connected in an Ethernet network in accordance with Goodwin Hall's instrumentation set up. Another extension to this study can be to include scenarios where sensor data is transmitted over different communication protocols such as over Wifi. In this work, we only considered Cloud VM instances in the AWS-East data centers as they offered lowest latency from our lab set up. Our set up happens to be in close vicinity of a data-center and the Cloud computing scenario suffers from considerably low latency. However, this would likely not be the case for many other locations. This work can be extended to consider Cloud VMs in other regions to study their impact on end-to-end latency for deploying streaming IoT applications in other locations.

Bibliography

[1] Amqp, an application layer protocol for message-oriented middleware.

[2] Apache edgent, an open source programming model and runtime for edge devices. https://edgent.apache.org/.

[3] Edgex foundry, an open framework for iot edge computing.

[4] The iot data explosion: How big is the iot data market? https://priceonomics.com/the-iot-data-explosion-how-big-is-the-iot-data/.

[5] Ispout api documentation.

[6] Mqtt, machine-to-machine (m2m)/iot connectivity protocol. http://mqtt.org.

[7] OpenFog, An Open Reference Architecture for Fog Computing,. https://www.openfogconsortium.org/.

[8] Rabbitmq.

[9] Raspberry Pi.

[10] Storm benchmark.

[11] Ucla. the ucla factor building seismic array.

[12] Mohammad Aazam, Imran Khan, Aymen Abdullah Alsaffar, and Eui-Nam Huh. Cloud of things: Integrating internet of things and cloud computing and the issues involved. In *Proceedings of 2014 11th International Bhurban Conference on Applied Sciences & Technology (IBCAST) Islamabad, Pakistan, 14th-18th January, 2014*, pages 414–419. IEEE, 2014.

[13] Daniel J Abadi, Yanif Ahmad, Magdalena Balazinska, Ugur Cetintemel, Mitch Cherniack, Jeong-Hyon Hwang, Wolfgang Lindner, Anurag Maskey, Alex Rasin, Esther Ryvkina, et al. The design of the borealis stream processing engine. In *Cidr*, volume 5, pages 277–289, 2005.

[14] Yuvraj Agarwal, Bharathan Balaji, Rajesh Gupta, Jacob Lyles, Michael Wei, and Thomas Weng. Occupancy-driven energy management for smart building automation. In *Proceedings of the 2nd ACM workshop on embedded sensing systems for energy-efficiency in building*, pages 1–6. ACM, 2010.

[15] Mohammad Albakri, Pablo Tarazaga, et al. Impact localization in dispersive waveguides based on energy-attenuation of waves with the traveled distance. *Mechanical Systems and Signal Processing*, 105:361–376, 2018.

[16] Muhammad Intizar Ali, Feng Gao, and Alessandra Mileo. Citybench: A configurable benchmark to evaluate rsp engines using smart city datasets. In *International Semantic Web Conference*, pages 374–389. Springer, 2015.

[17] Apache. Apache flink. https://flink.apache.org/.

[18] Apache. Apache samza. http://samza.apache.org/.

[19] Apache. Apache spark. a fast and general engine for large-scale data processing.

[20] Apache. Apache storm. an open source distributed realtime computation system. http://storm.apache.org/.

[21] Apache. Apache zookeeper. an open source server that enables highly reliable distributed coordination. "https://zookeeper.apache.org/".

[22] Arvind Arasu, Brian Babcock, Shivnath Babu, Mayur Datar, Keith Ito, Itaru Nishizawa, Justin Rosenstein, and Jennifer Widom. Stream: The stanford stream data manager

(demonstration description). In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, SIGMOD '03, pages 665–665, New York, NY, USA, 2003. ACM.

[23] Martin Arlitt, Manish Marwah, Gowtham Bellala, Amip Shah, Jeff Healey, and Ben Vandiver. Iotabench: an internet of things analytics benchmark. In *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering*, pages 133–144. ACM, 2015.

[24] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. A view of cloud computing. *Commun. ACM*, 53(4):50–58, April 2010.

[25] Dustin Bales. Characteristic classification of walkers via underfloor accelerometer gait measurements through machine learning. Master's thesis, Virginia Polytechnic Institute and State University, 2016.

[26] Sean Boland. Ranking popular distributed computing packages for data science, 2018. https://blog.thedataincubator.com/2018/02/ranking-popular-distributed-computing-packages-for-data-science/.

[27] Sigismondo Boschi and Gabriele Santomaggio. *RabbitMQ cookbook*. Packt Publishing Ltd, 2013.

[28] Samuel Case Bradford, John F Clinton, J Favela, and TH Heaton. Results of millikan library forced vibration testing. 2004.

[29] Data Canvas. Sense your city: Data art challenge.

[30] Don Carney, Ugur Cetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Greg Seidman, Michael Stonebraker, Nesime Tatbul, and Stan Zdonik. Monitoring streams:

a new class of data management applications. In *Proceedings of the 28th international conference on Very Large Data Bases*, pages 215–226. VLDB Endowment, 2002.

[31] Zhuo Chen, Wenlu Hu, Junjue Wang, Siyan Zhao, Brandon Amos, Guanhang Wu, Kiryong Ha, Khalid Elgazzar, Padmanabhan Pillai, Roberta Klatzky, Daniel Siewiorek, and Mahadev Satyanarayanan. An empirical study of latency in an emerging class of edge computing applications for wearable cognitive assistance. In *Proceedings of the Second ACM/IEEE Symposium on Edge Computing*, SEC '17, pages 14:1–14:14, New York, NY, USA, 2017. ACM.

[32] Sanket Chintapalli. Benchmarking streaming computation engines at yahoo!, December 2015. [Online; posted Dec-2015].

[33] Cisco. Cisco Kinetic Edge & Fog Processing Module (EFM). https://www.cisco.com/c/dam/en/us/solutions/ collateral/internet-of-things/kinetic-datasheet-efm.pdf.

[34] Meredith Claeys. Instrumentation of buildings to enhance student learning - a case study at marquette university's discovery learning compex. Master's thesis, Marquette University, 2011.

[35] Rebecca L Collins and Luca P Carloni. Flexible filters: load balancing through backpressure for stream programs. In *Proceedings of the seventh ACM international conference on Embedded software*, pages 205–214. ACM, 2009.

[36] Mckinsey & Company. The internet of things: Mapping the value beyond the hype, 2015.

[37] Eduardo Cuervo, Aruna Balasubramanian, Dae-ki Cho, Alec Wolman, Stefan Saroiu, Ranveer Chandra, and Paramvir Bahl. Maui: Making smartphones last longer with

code offload. In *Proceedings of the 8th International Conference on Mobile Systems, Applications, and Services*, MobiSys '10, pages 49–62, New York, NY, USA, 2010. ACM.

[38] Alex Davies. Cisco pushes iot analytics to the extreme edge with mist computing. https://rethinkresearch.biz/articles/cisco-pushes-iot-analytics-extreme-edge-mist-computing-2/.

[39] Philippe Dobbelaere and Kyumars Sheykh Esmaili. Kafka versus rabbitmq. *CoRR*, abs/1709.00333, 2017.

[40] Charalampos Doukas and Ilias Maglogiannis. Bringing iot and cloud computing towards pervasive healthcare. In *2012 Sixth International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing*, pages 922–926. IEEE, 2012.

[41] Eclipse. Kura. open-source framework for iot. http://www.eclipse.org/kura/.

[42] Dave Evans. The internet of things; how the next evolution of the internet is changing everything.

[43] J. Zhu S. Addepalli F. Bonomi, R. Milito. Fog computing and its role in the internet of things. *In Proceedings of the First Edition of the MCC Workshop on Mobile Cloud Computing, MCC '12, pages 13–16, 2012.*

[44] George H. Forman and John Zahorjan. The challenges of mobile computing. *Computer*, 27(4):38–47, 1994.

[45] Mark S. Gordon, D. Anoushe Jamshidi, Scott Mahlke, Z. Morley Mao, and Xu Chen. Comet: Code offload by migrating execution transparently. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 93–106, Berkeley, CA, USA, 2012. USENIX Association.

[46] Donald Gross. *Fundamentals of queueing theory*. John Wiley & Sons, 2008.

[47] Karim Habak, Ellen W Zegura, Mostafa Ammar, and Khaled A Harras. Workload management for dynamic mobile device clusters in edge femtoclouds. In *Proceedings of the Second ACM/IEEE Symposium on Edge Computing*, page 6. ACM, 2017.

[48] Joseph M. Hamilton. Design and implementation of vibration data acquisition in goodwin hall for structural health monitoring, human motion, and energy harvesting research. Master's thesis, Virginia Polytechnic Institute and State University, April 2015.

[49] Mohammed A Hassan, Mengbai Xiao, Qi Wei, and Songqing Chen. Help your mobile applications with fog computing. In *Sensing, Communication, and Networking-Workshops (SECON Workshops), 2015 12th Annual IEEE International Conference on*, pages 1–6. IEEE, 2015.

[50] HortonWorks. Hortonworks best practices guide for apache storm. https://community.hortonworks.com/articles/550/ unofficial-storm-and-kafka-best-practices-guide.html.

[51] International Electrotechnical Commission (IEC). Iot 2020: Smart and secure iot platform. http://www.iec.ch/whitepaper/pdf/iecWP-loT2020-LR.pdf.

[52] McKinsey Global Institute. The internet of things: Mapping the value beyond the hype, June 2015.

[53] Lihong Jiang, Li Da Xu, Hongming Cai, Zuhai Jiang, Fenglin Bu, and Boyi Xu. An iot-oriented data storage framework in cloud computing platform. *IEEE Transactions on Industrial Informatics*, 10(2):1443–1451, 2014.

[54] Yiping Kang, Johann Hauswald, Cao Gao, Austin Rovinski, Trevor Mudge, Jason Mars, and Lingjia Tang. Neurosurgeon: Collaborative intelligence between the cloud and mo-

bile edge. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, pages 615–629, New York, NY, USA, 2017. ACM.

[55] M Kasarda, P Tarazaga, M Embree, S Gugercin, A Woolard, B Joyce, and J Hamilton. Detection and identification of firearms upon discharge using floor-based accelerometers. In *Special Topics in Structural Dynamics, Volume 6*, pages 45–53. Springer, 2016.

[56] Surabhi Kejriwal and Saurabh Mahajan. Smart buildings: How iot technology aims to add value for real estate companies. *Deloitte Center for Financial Services*, 2016.

[57] Patrick Kinney et al. Zigbee technology: Wireless control that simply works. In *Communications design conference*, volume 2, pages 1–7, 2003.

[58] Leonard Kleinrock. *Queueing systems, volume 2: Computer applications*, volume 66. wiley New York, 1976.

[59] Sanjeev Kulkarni, Nikunj Bhagat, Maosong Fu, Vikas Kedigehalli, Christopher Kellogg, Sailesh Mittal, Jignesh M Patel, Karthik Ramasamy, and Siddarth Taneja. Twitter heron: Stream processing at scale. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 239–250. ACM, 2015.

[60] Franck L Lewis et al. Wireless sensor networks. *Smart environments: technologies, protocols, and applications*, 11:46, 2004.

[61] Michael R Lightner, Lawrence Carlson, Jacquelyn F Sullivan, Michael J Brandemuehl, and Rene Reitsma. A living laboratory. *Proceedings of the IEEE*, 88(1):31–40, 2000.

[62] Ruirui Lu, Gang Wu, Bin Xie, and Jingtong Hu. Stream bench: Towards benchmarking modern distributed stream computing frameworks. In *Proceedings of the 2014*

IEEE/ACM 7th International Conference on Utility and Cloud Computing, UCC '14, pages 69–78, Washington, DC, USA, 2014. IEEE Computer Society.

[63] Tom H Luan, Longxiang Gao, Zhi Li, Yang Xiang, Guiyi Wei, and Limin Sun. Fog computing: Focusing on mobile users at the edge. *arXiv preprint arXiv:1502.01815*, 2015.

[64] Hongyu Miao, Heejin Park, Myeongjae Jeon, Gennady Pekhimenko, Kathryn S McKinley, and Felix Xiaozhu Lin. Streambox: Modern stream processing on a multicore machine. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 617–629, 2017.

[65] David L Mills. Internet time synchronization: the network time protocol. *IEEE Transactions on communications*, 39(10):1482–1493, 1991.

[66] Roberto Morabito. Inspecting the performance of low-power nodes during the execution of edge computing tasks. In *Consumer Communications & Networking Conference (CCNC), 2017 14th IEEE Annual*, pages 148–153. IEEE, 2017.

[67] Qian Ning, Chien-An Chen, Radu Stoleru, and Congcong Chen. Mobile storm: Distributed real-time stream processing for mobile clouds. In *Cloud Networking (CloudNet), 2015 IEEE 4th International Conference on*, pages 139–145. IEEE, 2015.

[68] Juhwan Oh, Zhongwei Jiang, and Henry Panganiban. Development of a smart residential fire protection system. *Advances in Mechanical Engineering*, 5:825872, 2013.

[69] A. Papageorgiou, E. Poormohammady, and B. Cheng. Edge-computing-aware deployment of stream processing tasks based on topology-external information: Model, algorithms, and a storm-based prototype. In *2016 IEEE International Congress on Big Data (BigData Congress)*, pages 259–266, June 2016.

[70] Christian Richter, Antonio Sanso, David Rockett, and Harshita Nersu. System and method for dynamically and efficently directing evacuation of a building during an emergency condition, August 25 2009. US Patent 7,579,945.

[71] Hooman Peiro Sajjad, Ken Danniswara, Ahmad Al-Shishtawy, and Vladimir Vlassov. Spanedge: Towards unifying stream processing over central and near-the-edge data centers. In *Edge Computing (SEC), IEEE/ACM Symposium on*, pages 168–178. IEEE, 2016.

[72] Rodrigo Sarlo and Pablo A Tarazaga. Modal parameter uncertainty estimates as a tool for automated operational modal analysis: Applications to a smart building. In *Dynamics of Civil Structures, Volume 2*, pages 177–182. Springer, 2019.

[73] Rodrigo Sarlo, Pablo A Tarazaga, and Mary E Kasarda. Operational modal analysis of a steel-frame, low-rise building with l-shaped construction. In *Sensors and Smart Structures Technologies for Civil, Mechanical, and Aerospace Systems 2017*, volume 10168, page 101682H. International Society for Optics and Photonics, 2017.

[74] Mahadev Satyanarayanan, Victor Bahl, Ramón Caceres, and Nigel Davies. The case for vm-based cloudlets in mobile computing. *IEEE pervasive Computing*, 2009.

[75] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu. Edge computing: Vision and challenges. *IEEE Internet of Things Journal*, 3(5):637–646, Oct 2016.

[76] Anshu Shukla, Shilpa Chaturvedi, and Yogesh Simmhan. Riotbench: A real-time iot benchmark for distributed stream processing platforms. *CoRR*, abs/1701.08530, 2017.

[77] Moataz Soliman, Tobi Abiodun, Tarek Hamouda, Jiehan Zhou, and Chung-Horng Lung. Smart home: Integrating internet of things with web services and cloud computing. In

2013 IEEE 5th international conference on cloud computing technology and science, volume 2, pages 317–320. IEEE, 2013.

[78] Michael Stonebraker, Uğur Çetintemel, and Stan Zdonik. The 8 requirements of real-time stream processing. *ACM SIGMOD Record*, 34(4):42–47, 2005.

[79] Fan TongKe. Smart agriculture based on cloud computing and iot. *Journal of Convergence Information Technology*, 8(2), 2013.

[80] Luis M Vaquero and Luis Rodero-Merino. Finding your way in the fog: Towards a comprehensive definition of fog computing. *ACM SIGCOMM Computer Communication Review*, 44(5):27–32, 2014.

[81] Ashkan Yousefpour, Caleb Fung, Tam Nguyen, Krishna Kadiyala, Fatemeh Jalali, Amirreza Niakanlahiji, Jian Kong, and Jason P. Jue. All one needs to know about fog computing and related edge computing paradigms: A complete survey. *CoRR*, abs/1808.05283, 2018.

[82] Ben Zhang, Nitesh Mor, John Kolb, Douglas S Chan, Ken Lutz, Eric Allman, John Wawrzynek, Edward A Lee, and John Kubiatowicz. The cloud is not enough: Saving iot from the cloud. In *HotStorage*, 2015.