MECBench: A Framework for Benchmarking Multi-Edge Computing Systems

by

Omar Naman

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2023

Author's Declaration

This thesis consists of material all of which I authored or co-authored: see Statement of Contributions included in the thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Statement of Contributions

I am the main contributor to this thesis, I designed the MECBench, implemented all of its components, and implemented and evaluated the scenarios in the evaluation chapter. Hala Qadi helped with the design and development of the graphical user interface as described in Section 3.5.1.

Abstract

I present MECBench, an extensible benchmarking framework for multi-access edge computing. MECBench is configurable and can emulate networks with different capabilities and conditions, can scale the generated workloads to mimic large number of clients, and can generate a range of workload patterns. MECBench is extensible; it can be extended to change the generated workload, use new datasets, and integrate new applications. MECBench's implementation includes machine learning and synthetic edge applications.

I demonstrate MECBench's capabilities through three scenarios: an object detection processing for drone navigation, a natural language processing application, and a synthetic workload with configurable compute and I/O intensity. My evaluation shows that MECBench can be used to answer complex what-if questions pertaining to design and deployment decisions of MEC platforms and applications. My evaluation explores the impact of different combinations of applications, hardware, and network conditions as well as the cost-benefit tradeoff of different designs and configurations.

Acknowledgements

First and foremost, I would like to take this opportunity to express my great appreciation to my advisor Samer Al-Kiswany for being a great mentor, both professionally and personally. I would like to thank him for his continuous guidance, support, and encouragement which significantly helped me to develop my research skills and my character. This thesis would never be possible without him.

This thesis benefited a great deal from all the guidance and support I received from Professor Martin Karsten. Thank you Martin for providing valuable feedback and expertise to me throughout my research journey.

I would like to express my sincere gratitude to my friends and colleagues in Waterloo's Advanced Systems Lab (WASL); Ahmed and Ashraf for their great and continuous support, Hala for her continuous encouragement and her great help in developing the graphical user interface. I would also like to thank Aladdin and Paul from Rogers for their involvement in the project's growth and development.

And finally, thank you, Qadora, Ali, Anwar, Abdallah, and Basil for being there.

There are no words to describe my gratitude to my parents, my brother, and my sisters. I would have never been able to continue my studies without their unconditional love and support.

Dedication

This is dedicated to my loved ones; my parents, my brother, and my sisters. Thank you and I love you all.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

The fifth-generation mobile network promises unprecedented improvements in communication throughput, latency, and deployment density. This improvement enables new application domains that require these communication characteristics, such as autonomous driving, smart cities, drone applications, and Internet of Things (IoT) applications. The 5G specification includes Multi-Access Edge Computing (MEC) clusters used to overcome geo-distributed data centers' physical latency limitation [23]. MEC clusters are small and distributed across the mobile network to bring cloud services closer to the users and avoid the high latency imposed by contacting distant data centers.

MEC plays a critical role in realizing the full potential of 5G and beyond mobile networks. MEC provides a lower latency alternative to data centers, can mask network failures that disconnect clients from data centers, and provides resources for offloading complex application logic from resource-limited devices.

Building and deploying a MEC-supported application is complicated because of challenges in the MEC, the network, and applications. First, there are no standard hardware or software specifications for the MEC design, leaving application developers and service providers guessing what and how many resources should be provisioned. Second, the network capabilities differ between cities, even in a mobile network of the same provider. Third, as this field is in its infancy, there are no established applications to guide service providers' design and provisioning steps. These challenges complicate designing, provisioning, deploying, and billing MEC applications.

For instance, consider how a service provider can decide if they can support a specific service level agreement (SLA) for a given application in a particular city. More concretely, imagine a drone-based package delivery application that would like to purchase a MEC-based service to help with drone navigation. The application developers expect up to 50 drones at any time and require a response time of 100 ms for 95% of the requests. The service provider needs to answer many application- and platform-specific questions to

support this application. For instance, the service providers need to find out if the MEC hardware and network at the target city can support these requirements. What hardware upgrades, if any, are required to support this application? Furthermore, how much will it cost to support this application? The application developers need to explore questions related to its design. For instance, would data compression techniques make it cheaper to run this application? How much changing the drone speed can help scale the system to support more drones? What is the accuracy/performance trade-off of changing the resolution of the images captured by the drone?

To help application designers and service providers answer application- and deployment-specific questions, I present MECBench, a benchmarking tool that can help practitioners answer what-if questions. MECBench takes the target application, deploys it on the target MEC platform, then generates workloads to measure the application and platform performance. MECBench is highly configurable and extensible. It can be configured to mimic a range of network conditions, generate configurable client workloads, and tune the resources available for a MEC application. MECBench is designed to facilitate extending the benchmark with new datasets and applications. If an application is unavailable for deployment, MECBench offers a tunable synthetic application that can mimic an application's compute and I/O intensity.

In order to demonstrate MECBench's capabilities to help make informed decisions, I use MECBench to explore three application scenarios: An obstacle avoidance service for drones, a natural language processing (NLP) service for phone applications, and a synthetic application with varying compute and I/O loads. In my evaluation, I explore questions related to the MEC hardware, such as the cost-performance trade-off of different hardware configurations; questions related to the network capabilities, such as exploring the impact of network conditions on application performance; and questions related to application design, such as exploring the impact of data compression and image resolution on the application performance.

The rest of this work is organized as follows. Section 2 details the design of MECBench. Section 3 discusses implementation details. Section 4 presents the scenario-based evaluation. Section 5 discusses the related work. I conclude in Section 6.

Chapter 2

Design

MECBench comprises five main components: load generator, service manager, network emulator, storage, and controller. Figure 2.1, shows the interaction between the five components.

The service manager runs the service under test (SUT) subject to the evaluation. The load generator generates the workload representing one or more clients. The network emulator can emulate different network technologies and conditions. The storage component stores experiment descriptions, configurations, input data, and results. The controller is the main engine for starting and managing all components throughout the experiment. The rest of this section details the design of each of these components.

2.1 Load Generator

The load generator (LoadGen) component generates the system evaluation workloads. The LoadGen generates requests sent to the service manager. Each request may have one or more queries. The service manager processes these queries using the SUT and sends the response back to the LoadGen. The nature of the query is application specific. For instance, it could include an inference request for an ML-based SUT, a transaction request for a database, or a bookkeeping request for an IoT application. Each query has a deadline. If a request misses a deadline, it notifies the LoadGen. The LoadGen component is flexible and can be configured to mimic different workload patterns.

2.1.1 LoadGen Design

LoadGen can be deployed on multiple nodes. Figure 2.2 shows the design of LoadGen on a single node. The main component of LoadGen is the Orchestrator, which loads an

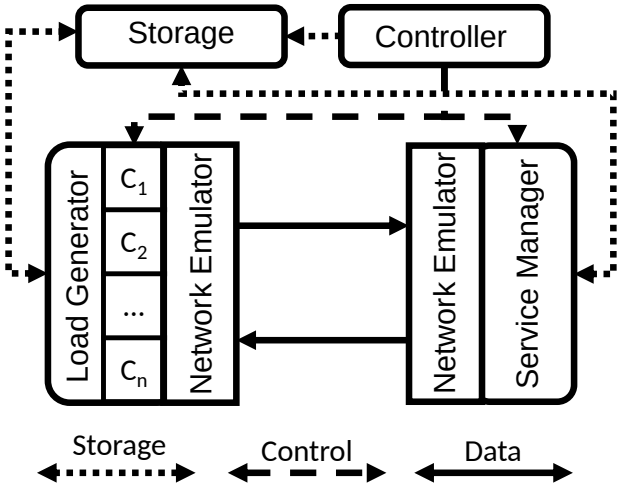

Figure 2.1: Architecture of MECBench, with the components (load generator, edge service, network emulator, storage, and controller) and their interactions.

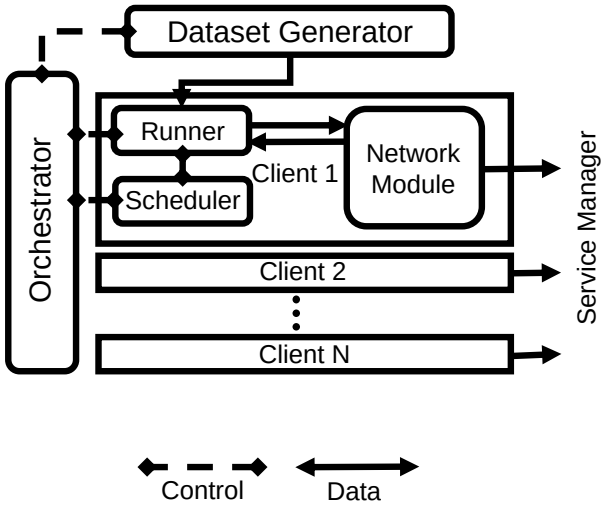

Figure 2.2: LoadGen design.

experiment configuration, sets the workload metadata, starts and controls clients, and keeps track of the response time of each request.

The Dataset Generator is a module that defines the data used in the evaluation and provides a list of data items that clients can retrieve during the evaluation.

The Orchestrator queries Dataset Generator to retrieve the metadata for the requests before the experiment starts. The Orchestrator uses the metadata to create requests and send these requests to Client modules.

The Client is the module that mimics a single client of the target edge service. The Client runs two threads. One thread generates a request and sends it to the service manager; another thread receives the response and sends the response time to the Orchestrator. A Scheduler (Figure 2.2) instructs the Runner to send requests at a configurable rate in the Client module. The Scheduler is configurable and can generate various workload patterns.

The Runner module generates a request and sends it to the service manager over the network. The Runner receives a request's metadata from the Scheduler, uses the request metadata to retrieve the request data from the Dataset Generator, then forwards the request to the Network module. The Runner also receives the response from service manager and can perform post-processing such as verifying the correctness or quality of the response. The Orchestrator keeps track of each request's response time by assigning a unique ID for each generated request and pairing it with its corresponding response. The Runner communicates the response time to the LoadGen Orchestrator. The Network module handles the details of the network communication. It sends a request to the service manager over the network and receives its result.

Clients can be configured as closed-loop clients that send one request at a time and wait for its response or open-loop clients that send multiple concurrent requests at a pre-configured rate. This allows the evaluation of different client behaviours and their impact on the performance of the SUT.

2.1.2 Workload Configuration

The workload constructed by the Orchestrator can be configured to change the number of concurrent clients sending requests, the number of requests sent per second, and the number of queries in each request. These configurations are defined in the workload configuration file that the Orchestrator reads at the start of the experiment.

LoadGen can be run on multiple nodes, each generating its workload. This allows LoadGen to generate workloads that can saturate system components that a single instance of LoadGen cannot saturate. This multi-deployment option of LoadGen can also help generate workloads with different patterns on each instance of LoadGen, further increasing its flexibility.

2.1.3 Scenarios

LoadGen is flexible and can be configured to generate a range of client workloads. To simplify the usage of LoadGen, I implemented the following application scenarios. The scenarios mimic real request patterns of online services and are configurable.

- **SingleStream**: This scenario represents applications concerned with response time. The requests generated contain a single query per request. The requests are generated in a closed loop. A typical test using this scenario will collect the end-to-end response time of the queries and analyse the response tail latency.
- **MultiStream**: This scenario represents an application with a constant rate of requests that carry queries from a set of sources. For example, the requests contain multiple queries representing multiple sensors, generated at a constant time interval between each query. The requests are generated in a closed or open loop. LoadGen will skip generating a request if the last request does not complete in time.
- **Server**: This scenario represents the workloads of online services that receive queries from multiple clients. The requests are generated following a Poisson distribution in an open loop. Each request has one query.
- **Offline**: This scenario represents applications that perform batch processing of data. LoadGen pushes all the queries in the dataset to the service manager to process at once, and measures how long it takes to process the complete batch. The collected results are often analyzed to find the throughput of the service manager.

2.2 Service Manager

The service manager is the component that manages the service under test. Figure 2.3 shows the design of service manager. The service manager receives requests from the LoadGen clients. The service manager parses the request and issues the queries to the loaded SUT. The service manager gets the response from the SUT and serializes it to the LoadGen. While the service manager implements several parsers for different applications, it offers an API that can be extended to implement custom parsers.

The service manager supports concurrent requests from clients. The service manager dedicates a worker thread for each LoadGen client. A service manager worker receives the stream of requests from a LoadGen client, parses the requests, passes the requests to SUT, and sends the results back to the LoadGen client. After passing the request to the SUT, the SUT implementation is responsible for processing the request, managing the resources, and sending a response back to the service manager. The service manager

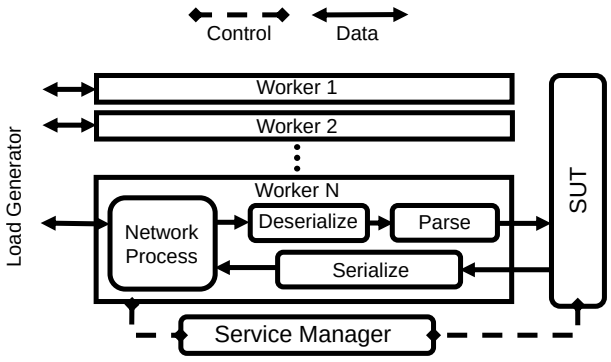

Figure 2.3: Service Manager design.

provides flexible support for different SUT models. The SUT can run in a separate process and communicate using OS inter-process communication techniques, or as a library as part of the service manager process.

2.3 Communication Layer

MECBench offers a communication layer to communicate between the LoadGen and service manager. This layer abstracts the network communication details from the clients, including communication protocols, request serialization, and response deserialization. In my implementation, LoadGen uses Google's Remote Procedure Call framework *gRPC* [11] using its treaming API. Each client creates a gRPC stream to its corresponding thread at the service manager. The stream is used throughout the test to send all the client requests.

The communication layer also offers serialization and deserialization of the requests and responses using the Protocol Buffers (Protobuf) [12] format. I also utilize gRPC's streaming to handle concurrent requests per client. This simplifies implementing open-loop clients. Furthermore, gRPC supports setting deadlines for requests.

2.4 Network Emulation

MECBench provides the ability to emulate the network conditions between the service manager and the LoadGen clients by utilizing Linux's Traffic Control (TC) [20], using the TC's Network Emulation (NetEm) [13] module. NetEm provides the ability to emulate a variety of network conditions, including adding delays to packets, introducing packet

loss, and limiting transfer rates. MECBench uses these capabilities to facilitate evaluating services under different network technologies and conditions.

MECBench's network emulation integrates the emulation of the following properties. These properties can have different settings depending on the direction of communication from LoadGen to the service manager and from the service manager to LoadGen. This facilitates emulating mobile networks that typically have different upload and download characteristics.

- **Delay and jitter**: The network emulator can add a delay to each outgoing request. The delay can be a fixed value or generated following a uniform distribution. This configuration may introduce jitter.
- **Packet loss**: The network emulator can drop packets. The packets are dropped following a uniform distribution. The rate of packet loss is configurable.
- **Transfer rate**: The network emulator can limit the throughput per LoadGen instance. Multiple LoadGen instances can be deployed per node to get a finer granularity control of the throughput.
- **Packet reordering**: The network emulator can send packets out of order. This property can control the percentage of outgoing packets sent out of order.

2.5 Storage

MECBench is typically deployed on multiple nodes. Consequently, datasets, configurations, and results must be accessible over the network. MECBench offers two storage services for different deployment platforms.

MECBench Storage. MECBench uses an SQL database service to save and aggregate results collected during the experiment, as well as the experiment configurations. This storage service can be queried using SQL queries to retrieve the results of a specific experiment after completion.

Blob Storage. To support cloud deployments of MECBench, the experiment data and configuration is stored on files on a network-accessible blob storage service.

2.6 MECBench Controller

This component is responsible for orchestrating the system's deployment, starting and configuring experiments and workloads.

The controller exposes a REST API that can be used to access most of the functionality of MECBench, allowing it to be extended by other automation scripts and graphical user interfaces. The controller abstracts the functionality of MECBench and its internal services by providing the ability to start, stop and configure experiments without knowing the underlying implementation details of the engine and the components.

To run an experiment, a JSON configuration is pushed to the controller. This configuration includes the description of the LoadGen parameters and the network emulation layer, alongside other parameters like the type of scenario to run, and how many times the experiment should be run.

The deployment of the service manager is separated from the deployment of the experiments to ensure its reusability. Since the service manager's configurations are often the same across different experiments, this allows less overhead from constantly deploying new instances for each experiment, adding time to the total experiment duration [16].

2.7 MECBench's Extensibility

One of the main objectives of MECBench is to provide a flexible and extensible platform for evaluating the performance of edge computing systems. Adding new client implementations, dataset definitions, and SUT servers is all done by extending the existing classes defined in the MECBench's codebase.

2.7.1 LoadGen Extension

Creating a new evaluation use case is done by extending two main classes of LoadGen's API: *Runner* and *Dataset*. The definition of a new dataset and a new runner should be done by implementing the *Dataset* and *Runner* APIs to match the requirements of the new use case.

The *Dataset* API to be extended is defined as:

- **loadDataset**: Load the metadata of the dataset into memory. This method is called once at the beginning of the evaluation. It loads information related to the generated queries, including the total number of queries and the path of the data related to them.
- **loadQueryData**: Load specific data related to a set of queries into memory to be used in the current evaluation cycle. LoadGen generates a set of indices to be used by the clients. The partial loading allows working with datasets that do not entirely fit into memory while avoiding extreme disk I/O delays.

- **getQueryData**: Retrieve data related to a specific query from the loaded datasets, given its index. This method is called each time the Runner adds a query to a request.
- **postProcess**: Perform any needed post-processing on the results of the queries, including accuracy tests and processing that a client will perform online as part of the evaluated application. The Runner calls this method each time a query response is received.
- **getNumberOfQueries**: Return the total number of queries in the dataset. This method is called by LoadGen to correctly generate the query indices based on the number of queries in the dataset.

The *Runner* API to be extended is defined as:

- **runQuery**: Handle the query generated by the LoadGen. This method is called by the LoadGen each time a query is scheduled containing the metadata of the set of samples the *Runner* is to send.
- **call**: Send a query to the service manager. This method is called inside the **runQuery** method after the query's data is retrieved from the *Dataset*.
- **clone** and **init**: These methods are used to spawn multiple instances of the *Runner* to be passed to the client threads, allowing the ability to define the separation and memory isolation of the clients.
- **Constructor**: The constructor of *Runner* receives the *Dataset* instance to be used in the evaluation.

2.7.2 Service Manager Extension

To add a new application to the service manager, one can extend the SUT class, which defines how the SUT is initiated, as well as the serialization and deserialization of the inputs and outputs. The API to be extended is:

- **load**: Load the SUT service. This method initializes the SUT, including spawning the SUT's processes if the SUT runs as a separate process, defining the communication tunnels between the SUT and service manager's workers, and loading any external data needed to run the SUT. This method is called once at the initialization of the service manager and can be left empty if the SUT does not require any external data or initialization.
- **parseQuery**: Deserialize the query received from the LoadGen client to a format that can be passed to the running SUT through **processQuery**.

- **processQuery**: Receives the deserialized/parsed query and passes it to the running SUT for further processing. The SUT will return the results in the format defined by the **serializeResponse** method.
- **serializeResponse**: Serialize the SUT's results to be streamed back over the network to the LoadGen client. The client will use its deserialization method to parse the results for further processing.

Chapter 3

Implementation

I implemented MECBench in 7810 lines of C++ code and 2475 lines of Python code. In addition to the C++ version of the service manager, I implemented a Python version of the service manager to help integrate machine learning models that are not available in C++. The controller and storage of MECBench are implemented in Python. Communication between LoadGen and the service manager is implemented using gRPC, utilizing Protobuf for serialization. The gRPC channels currently utilized in the communication are insecure, but can be easily swapped with SSL based gRPC channels. The controller and the storage can be contacted using their respective REST APIs. MECBench has prebuilt support for machine learning SUTs and synthetic benchmarks. The machine learning support is based on MLPerf [28], a single-node benchmark for machine learning models. The provided synthetic workloads mimic compute and I/O-heavy workloads. Kubernetes is used to deploy MECBench's components: the LoadGen, the service manager, and storage.

The rest of this section describes the implementation details for MECBench's components and the machine learning and synthetic benchmarks.

3.1 LoadGen

The LoadGen implementation uses a pool of threads to run the clients, two threads for each client as detailed below. The dataset implementation is extensible following the API discussed in Section 2.7.1. The dataset includes serialized queries. Each client thread uses the query index sent by the Orchestrator to fetch a query and send it to the service manager.

3.1.1 Runner

A Runner is a class responsible for defining how the data is communicated with the service manager, acting as the client in the evaluation. This includes communication protocols, query formats, and handling multi-query requests. My implementation of Runner uses the gRPC Streaming client to communicate with the server.

Each client spawns two threads, one for sending the requests to the service manager and the other for receiving the results. This implementation method allows the client to have multiple concurrent outgoing requests, allowing it to be an open-loop client. The client can be set to be a closed-loop client by setting the number of concurrent queries to one.

The protocol used by the Runner to communicate with the service manager carries a stream of bytes the SUT can parse. This stream of bytes is in a format defined by the Dataset API. The query also carries metadata describing the query; the request's id, and the byte stream's length. The service manager responds with a stream of bytes and query metadata; the id of the query and the length of the byte stream. The Runner sends the metadata to the Orchestrator to match the request-response and record the query's latency.

3.1.2 Dataset

The *Dataset* class is responsible for providing the data used in the evaluation. This includes the definition of the data format, how the items are loaded into memory and retrieved, and any pre-processing or post-processing that might be performed offline or at the client.

The main *Dataset* I implemented does not perform any pre-processing or post-processing of the items and assumes the items are already in the format that the SUT can process. It receives a system path that points to the dataset's directory. The directory is expected to contain one file per item, where the file's name is the item's id. The *Dataset* class loads the items into memory and retrieves them when the *Runner* requests.

3.2 Service Manager

The service manager is the component serving the SUT to be evaluated, usually deployed on its separate node to reduce resource contention with other processes. With the focus on providing a high level of flexibility and extensibility, MECBench's service manager is implemented twice; once in Python to allow the ease of integrating Python-based ML models and once in C++ to accommodate any C++-based models and synthetic evaluation models.

Similar to the *Runner*, the service manager serves the SUTs behind a gRPC service. For each incoming client stream of requests, the service manager spawns a new worker thread to handle the stream, parsing and passing the requests to the SUT and returning the results to the client. The service manager spawns a worker thread for each LoadGen client.

3.3 Machine Learning

MECBench comes prebuilt with machine-learning models that can be used as SUTs for evaluation. These models are implemented using MECBench's Python service manager. In this section, I describe each model's dataset and SUT implementations.

3.3.1 Machine Learning SUTs

I implemented three machine learning SUTs in MECBench. Their implementations are based on the ML support provided by MLPerf [28]. Following is a list describing the models and their implementation in MECBench:

- SSD-Mobilenet [15]: An object detection model that utilizes a single-shot detection (SSD) algorithm to detect and label objects in images. The Mobilenet suite of models targets deployment on low-end devices with limited computation power. The model is implemented in MECBench using the ONNX Runtime [10] library, utilizing the SSD-Mobilenet model used by MLPerf [28].

- EfficientDet [29]: The EfficientDet suite provides a set of models that target a wide range of accuracy and performance trade-offs by being trained on different input resolutions, requiring more computation for higher resolutions while providing more accurate inference results. This suite of models is implemented in MECBench using their original TensorFlow [5] implementation.

- SpaCy [14]: SpaCy is a natural language processing (NLP) library for high-performance production use. Originally, SpaCy was not designed to be used as a *Software as a Service* (SaaS) platform. However, due to its high performance and efficient resource utilization, I have implemented a SUT for SpaCy in MECBench. The service manager is implemented by forking a set of processes that load the SpaCy model and wait for requests to be communicated through memory pipes. For each request received from a client stream, the service manager chooses the following available process to handle the request.

3.3.2 Machine Learning Datasets

Along with the machine learning SUT implementations, I built, using the *Dataset* class described in Section 3.1.2, a set of datasets that generate items to be used as requests for each machine learning SUT.

I define an image pre-processing pipeline shown in Figure 3.1 to prepare the images for consumption by the machine learning models.

Raw COCO

I pre-processed Microsoft's Common Objects in Context (COCO) dataset [22] to be used as a dataset for the SSD-Mobilenet and EfficientDet SUTs. Each image in the dataset is downsized to match the resolution of the input of each model, e.g., 300×300 for SSD-Mobilenet. The images are then converted to *Model Input (Bitmap)*, the last stage of the image pre-processing pipeline, producing files ready to be consumed by the models without further processing. Due to it being a bitmap image, the size of all queries in the 300×300 dataset is $264KB$. This large query size can be problematic in most wireless standards due to their limited bandwidth but can be instantly passed to the model without decompression on the SUT side.

Labeled Raw COCO

I also implemented a dataset that targets the accuracy of the models. I use the raw COCO dataset, which receives a path to a directory containing the queries and a path to a ground truth file that describes the objects in the images and their bounding boxes. The ground truth is used in the post-processing step to measure the response accuracy.

JPEG COCO

I pass the images through the image pre-processing pipeline again. However, this time I stop at the third stage of the pipeline, producing compressed (JPEG) images expected to be decompressed by the SUT before being passed to the model. The distribution of the size of queries in the 300×300 dataset is shown in Figure 3.2, where 90% of the queries are smaller than $60KB$. This dataset alleviates the problem of large queries saturating the wireless link but requires the SUT to decompress the images before passing them to the inference model, which could introduce additional processing latency.

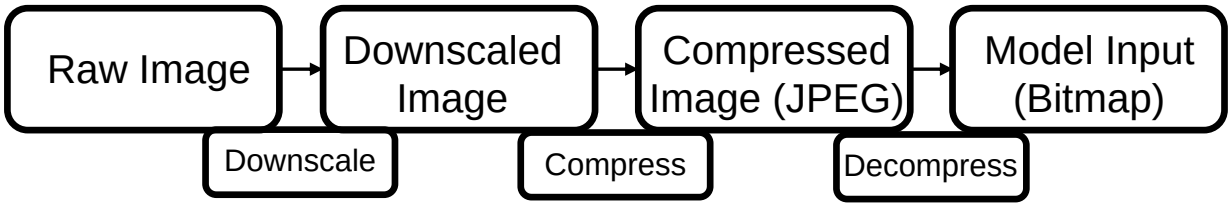

Figure 3.1: Image pre-processing stages.

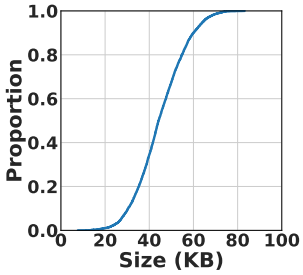

Figure 3.2: Size of samples in JPEG COCO.

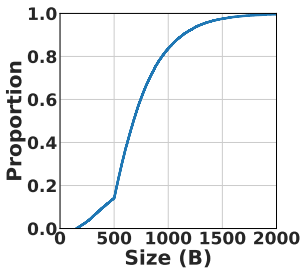

Figure 3.3: Size of samples in SQuAD.

SQuAD

I extracted the paragraphs from the *SQuAD* dataset [27], a question-answering dataset that contains a set of questions and paragraphs that answer the corresponding question. I use these paragraphs as queries to be sent to the SpaCy SUT to perform Named Entity Recognition (NER) on the paragraphs. The size of the queries in this dataset is small compared to the other datasets, shown in Figure 3.3, where 98% of the queries are less than 1500 bytes.

3.4 Synthetic Benchmarks

MECBench also implements a set of synthetic SUTs that mimic request, compute, and I/O-intensive applications. These synthetic benchmarks are implemented in C++ as follows:

- I/O Model: This benchmark targets the evaluation of the I/O throughput and latency of the service manager by performing a set of I/O operations on a single file. The model receives a request that specifies the total amount of data to be written, the amount of data written per operation, and whether the data should be *fsynced* to disk after each write. The model then creates a temporary file, writes to it based on the request's parameters, closes, and unlinks the file.

- CPU Model: This benchmark targets the performance of the CPU by continuously performing floating-point operations for a set amount of time. The model receives a request that specifies the total amount of time to perform the operations and does floating-point division until the requested time elapses.
- Requests Model: This model targets the throughput of the service manager in terms of the number of requests it can handle. Each request represents a sleep operation that sleeps for a specified time. The request contains the sleep period padded by zeros to reach a set size to evaluate the effect of the requests' size on performance.

The *Dataset* of these benchmarks is represented by a single configuration file specifying the benchmark's parameters. The path to the file is passed to the *Dataset* on startup, where it gets parsed, and the parameters are used to configure the requests sent to the service manager.

3.5 MECBench's web services

MECBench's web services: System Controller, Blob storage, and MECBench storage are all implemented as RESTful web servers using Python. Each service exposes a set of stateless endpoints that can be used by other services to ease the interaction.

3.5.1 Graphical User Interface

As part of the development, MECBench provides a graphical user interface (GUI) that allows users to easily access the functionality of MECBench without the need to closely interact with the intricate parts of the system.

I constructed a web service that interacts with the controller's RESTful API using Angular [17], a Javascript framework for building front-end applications and used an open-source dashboard template [30]. The GUI allows users to easily create, configure, and run experiments using an intuitive interface that supports most of the functionality of MECBench. The GUI also allows users to save and load experiment configurations as profiles that can be saved and loaded for future use.

The user is presented with a dashboard that shows a list of MECBench's components that can be configured inside the GUI. Starting with the LoadGen, the user can configure several parameters of the LoadGen, such as the scenario, the number of clients, and the dataset used in the evaluation, as shown in Figure 3.4.

The user can also configure the service manager parameters, such as the number of threads the SUT can use and the model loaded by the SUT, as shown in Figure 3.5. The

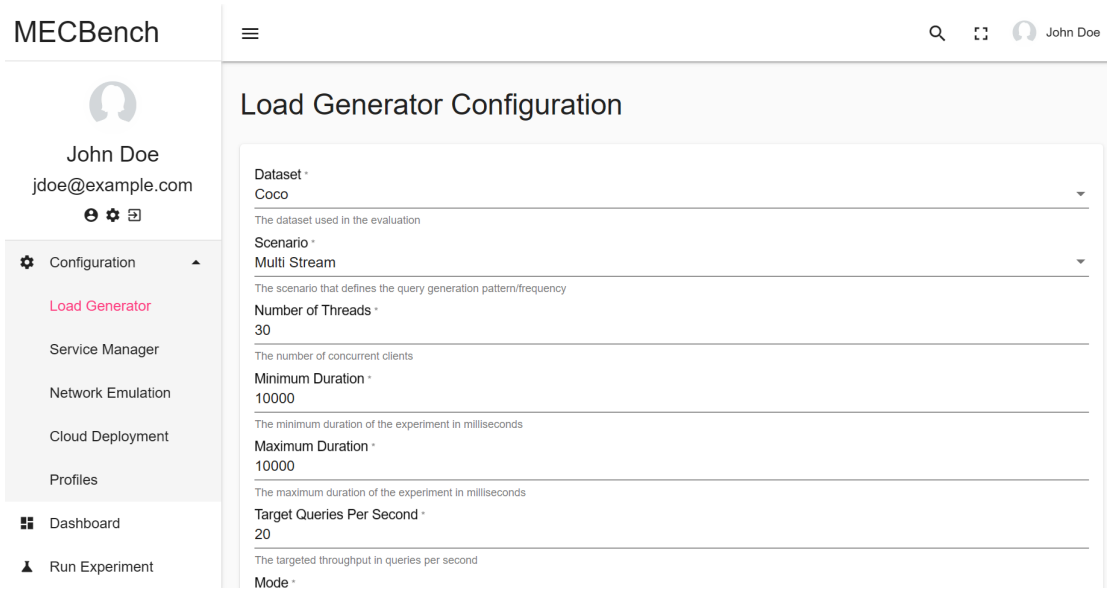

Figure 3.4: LoadGen configuration page in the GUI.

service manager configuration GUI page is mainly designed around the machine learning SUTs currently implemented in MECBench, as described in Section 3.3.1.

Following the configuration of the LoadGen and service manager components of MECBench, the user can configure the network emulation layer. The user can choose the network conditions for the LoadGen and service manager components, such as bandwidth, latency and jitter, and packet loss, as shown in Figure 3.6.

Since MECBench is integrated with AWS, the user can configure the cloud deployment of MECBench. The user can choose which cloud provider to use, and the type of instance to deploy the service manager on, as shown in Figure 3.7.

When assessing the user experience of the GUI, I found that filling the long list of parameters in the GUI can be tedious and error-prone, especially when the user wants to change a single parameter or conduct experiments over multiple sessions. To address this issue, I implemented a feature that allows the user to save the current configuration as a profile and load it later. This feature allows the user to save the configuration of the LoadGen, service manager, and network emulation layer, and load it later to run the experiment, see Figure 3.8. I use MECBench's storage to store the profiles and map them to previous experiments.

Finally, the user can specify an id for the experiment that can be used to retrieve its results from the MECBench storage later, as seen in Figure 3.9. Currently, the GUI does not support plotting the experiment results due to the number of parameters that can

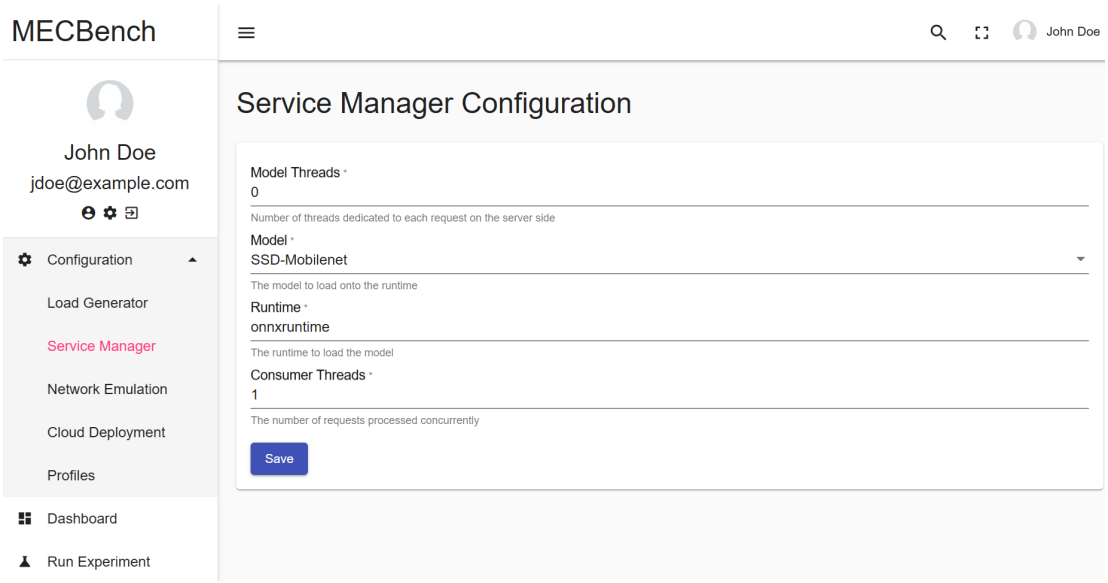

Figure 3.5: Service Manager configuration page in the GUI.

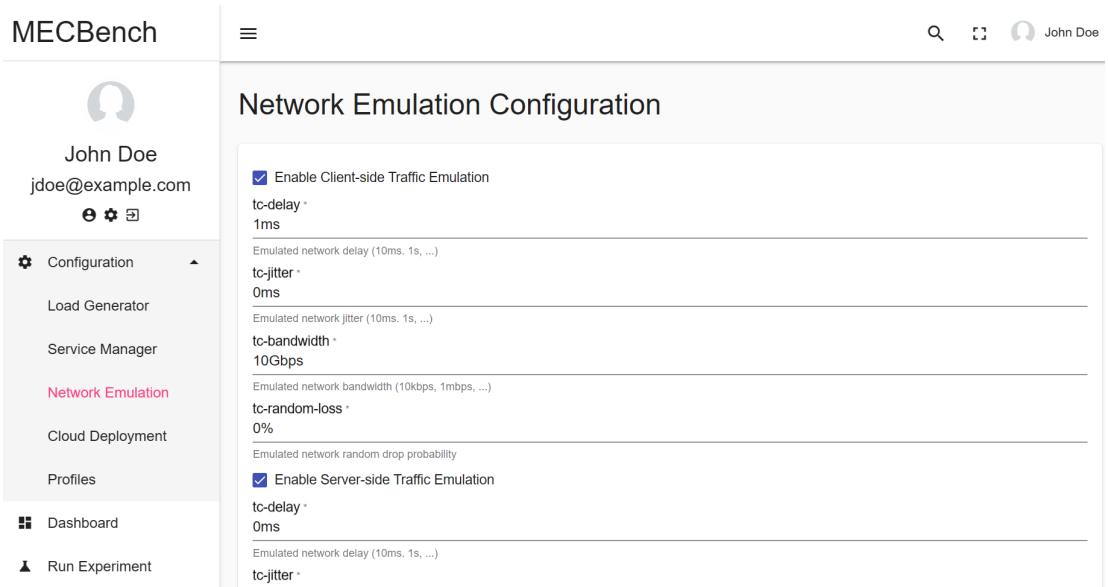

Figure 3.6: Network emulation configuration page in the GUI.

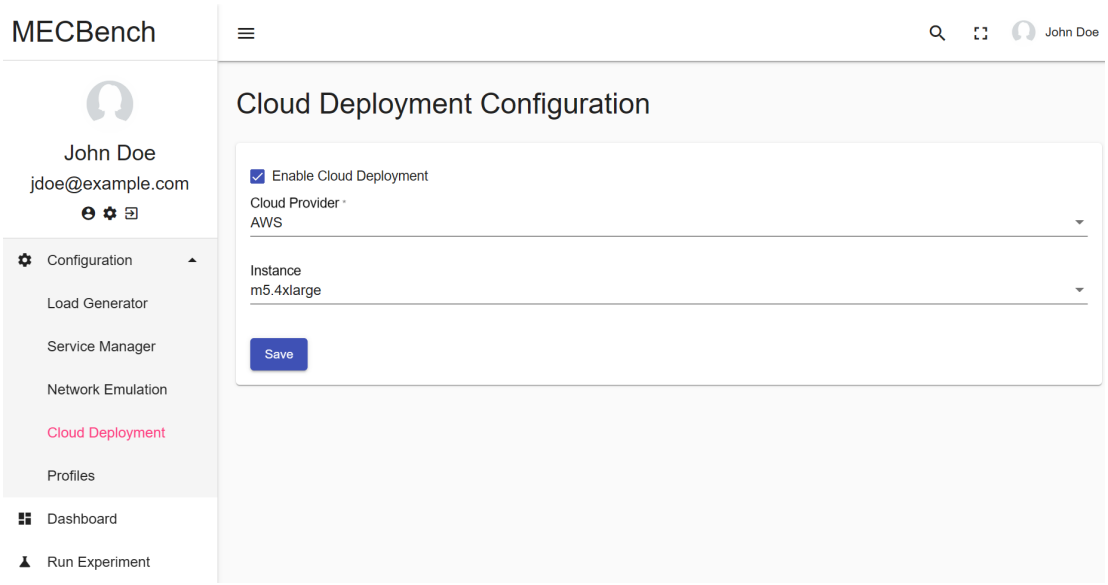

Figure 3.7: Cloud configuration page in the GUI.

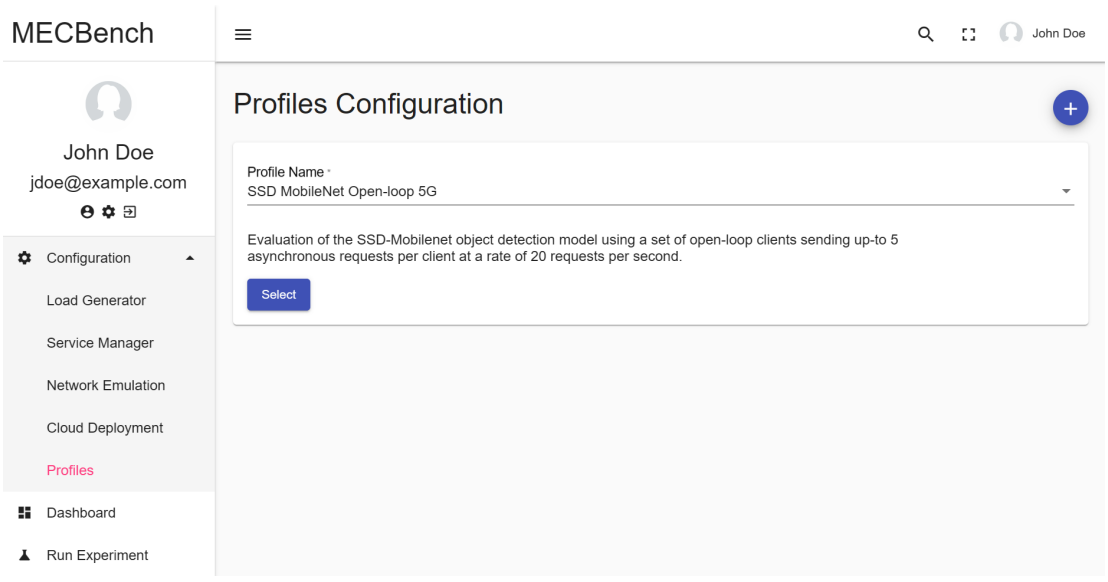

Figure 3.8: Profile configuration page in the GUI.

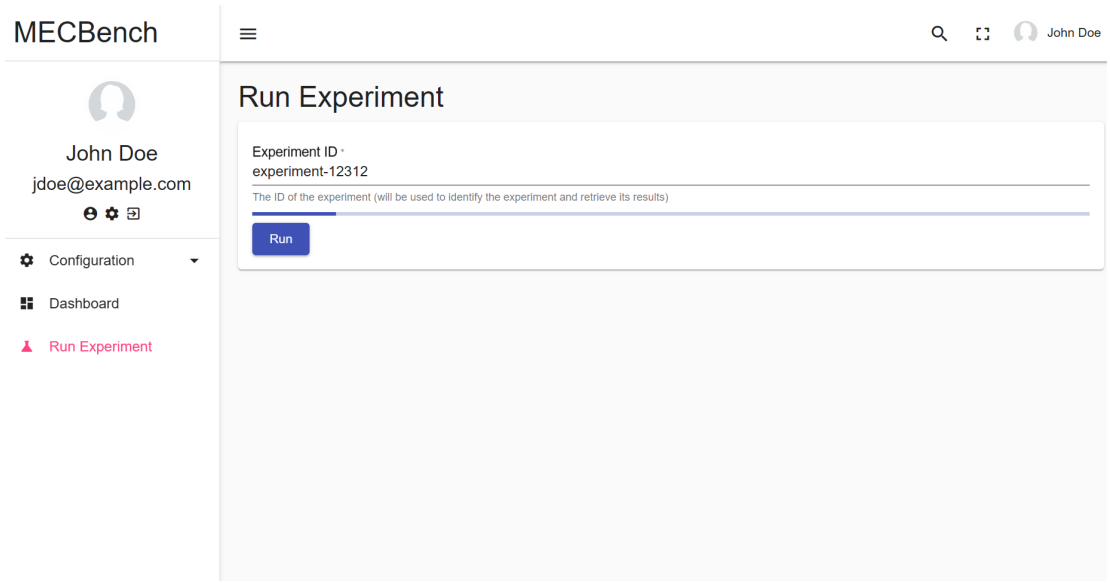

Figure 3.9: Run experiment page in the GUI.

change between experiments and the complexity of the results. However, I have plans to implement a dashboard that allows the user to create and customize plots of the results of the experiments.

3.6 Deployment

In the current implementation of MECBench, its components are containerized and deployable using a container orchestration system. Each component has a corresponding Docker image that can be pulled and deployed as a pod in a Kubernetes cluster. All the communication between network-connected components is done through the Kubernetes network, deploying the components as services with addresses that can be resolved at runtime via the Kubernetes Domain Name System (DNS).

MECBench's LoadGen can be deployed in a Kubernetes cluster in two ways; single-run or run-server, depending on the level of isolation needed between experiments. In the single-run mode, LoadGen is deployed as a single Kubernetes job, clearing the resources of the pods that ran the experiment after the experiment is completed. The single-run mode is used for experiments that could interfere with previously run experiments on the same node, such as experiments that leave a trace on the node's file system or network stack. On the other hand, the run-server mode deploys LoadGen as a Kubernetes service that receives experiment requests from the System Controller. The primary use of this mode

is to minimize the overhead of pod creation and deletion, allowing for a faster experiment turnaround time. It is also used to allow the synchronization of running an experiment on multiple nodes at the same time.

Amazon Web Services

MECBench uses a set of AWS services to deploy the system. Utilizing AWS's Elastic Kubernetes Service (EKS) for deploying and managing all the system's components. Images of each Kubernetes-deployable component are stored in AWS's Elastic Container Registry (ECR), and the deployed pods will pull the designated image from the ECR.

AWS's block storage (S3) is used as a storage service for MECBench's. LoadGen and the service manager can pull the data used in the evaluation from the S3 bucket and store it in the local filesystem.

Chapter 4

Evaluation

4.1 Evaluation Setup

I demonstrate the capabilities of MECBench using AWS's infrastructure with three different scenarios. For each scenario, I run a set of experiments that evaluate a specific capability of the service manager. MECBench's components are deployed on EC2 instances, and the instances are controlled by AWS's Elastic Kubernetes Service (EKS); Amazon's container orchestration service.

The evaluation experiments use a selection of the models and datasets described in section 3 on different types of AWS instances. These instances differ in the number of 2^{nd} generation Intel Xeon Platinum 8000 series processors [2], the amount of RAM available, and the existence of hardware accelerators, as shown in Table 4.1. The LoadGen is deployed on a single instance of type m5.xlarge that can saturate the network link as well as the service manager resources. All instances involved in the evaluation, LoadGen, the service manager, and storage are located in the same AWS region (us-east-2) in the same availability zone.

The experiments are set to generate requests for 10 seconds following the SingleStream closed-loop client workload generation and then wait for all the requests to be completed, where the jobs were run sequentially. The experiments are repeated 30 times. No online processing is performed by the LoadGen during the experiments, and all the datasets are preprocessed to a format that can be immediately consumed by the SUT and preloaded into memory.

Instance	Cores	vCPUs	RAM (GiB)	Price (USD per hour)	GPU
m5.large	1	2	8	0.096	-
m5.xlarge	2	4	16	0.192	-
m5.2xlarge	4	8	32	0.384	-
m5.4xlarge	8	16	64	0.768	-
m5.8xlarge	16	32	128	1.536	-
m5.16xlarge	32	64	256	3.072	-
p2.xlarge	2	4	61	0.900	K80

Table 4.1: AWS EC2 instance resources [2, 3]. The information was collected on August 8, 2022.

4.2 Drone Object Detection

One of MECBench's goals is to evaluate the trade-off of offloading onboard processing to a more powerful edge node, following the premise of edge computing. This scenario addresses the viability of running an object detection model at a MEC to support fast-moving autonomous drones. In this scenario, a drone sends a photo collected by its front camera to the closest MEC server. The MEC server is running an edge service that processes photos coming from the drone and detects any objects in the drone's path. For this evaluation, I use the SSD-Mobilenet model. For this application, it is critical that the response time be low to leave enough time for the drone to avoid an obstacle. I will set a condition that this scenario requires that the 95^{th} percentile latency of the response time be less than 100ms. In this scenario, I assume AWS instances are used as MEC servers.

To understand how best to deploy this application, service providers need to answer the following critical questions: Which AWS instance should I use to support this application? Which instance is most cost-efficient? Should I use the more expensive instances with GPU support? How many cores should I allocate for this application? Cameras can be adjusted to take lower-resolution photos, which introduces an interesting trade-off: lower-resolution photos can be processed faster as, they have lower transfer and processing time, but they may lead to lower object detection accuracy. Which resolution should the system designers use to reduce the response time while providing adequate accuracy? What are the network requirements to support this application? Will it run over 4G networks? How does data loss affect application performance? Given that a drone's speed is correlated with the rate of queries the drone issues to the MEC, this raises the question: What drone speeds or query rate does each networking technology support? In this section, I show how I use MECBench to answer these questions.

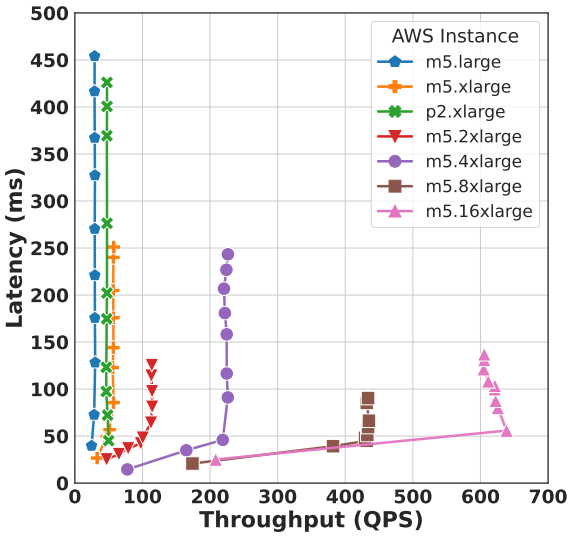

Figure 4.1: Throughput-latency figure for the object detection scenario. The y-axis shows the 95^{th} percentile latency.

4.2.1 What is the Cost/Performance Trade-Off of AWS Instances?

I start by exploring the cost/performance trade-off of different AWS instances. I evaluate the performance of the model on different types of instances, each with a different number of CPU cores and a different amount of RAM. The LoadGen is configured to act as a set of closed-loop clients sending queries from the *Raw COCO* dataset. During the evaluation, I keep increasing the number of parallel clients until the service manager or the network is saturated.

I also compare the performance of the SSD-Mobilenet running on a GPU-based OnnxRuntime deployed on a p2.xlarge instance with an NVIDIA K80 GPU.

Figure 4.1 shows the throughput and 95^{th} percentile latency of an edge service running an object detection service using the SSD-Mobilenet model. The figure shows the performance using different AWS instances, including p2.xlarge with a GPU. The figure shows that m5.large, m5.xlarge, m5.2xlarge, and p2.xlarge provide a low throughput for a latency less than 100 ms. This indicates that these instances cannot support deployments with a large number of drones. m5.4xlarge, m5.8xlarge, and m5.16xlarge achieve a throughput of over 200 queries per second, with m5.16xlarge achieving around 600 queries per second with 95^{th} percentile latency less than 100ms.

Figure 4.2 compares the different instances in terms of how many closed-loop drones

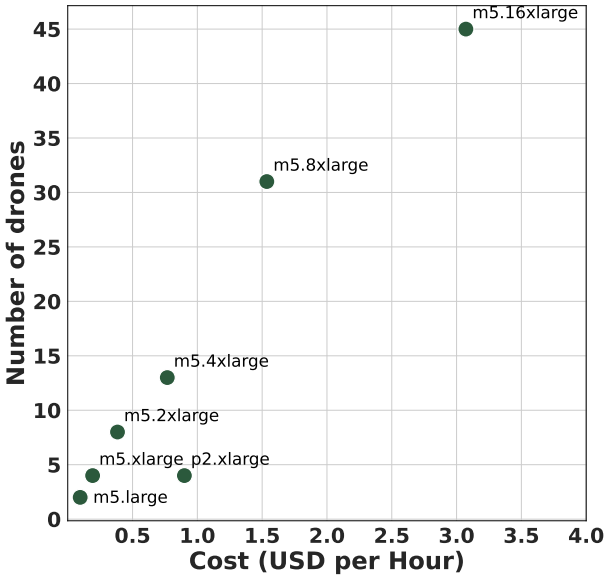

Figure 4.2: Maximum number of drones that can be served by the service manager while maintaining a 95^{th} percentile latency of $100ms$.

can be supported while keeping the 95^{th} percentile latency less than 100ms. The figure shows the cost/capacity tradeoff of the different instances. The figure shows that, while the p2.xlarge instance with a GPU costs 3 times more than the m5.xlarge, their performance is comparable because each can support only around 4 drones. The m5.8xlarge instance is the best fit for this scenario, as it has the lowest cost per supported drone. It can support 32 drones and costs $1.5 per hour, which brings the cost per drone to less than 5 cents per hour.

4.2.2 Does the Application Scale to Use Multiple Cores?

In this section, I explore the question of how many cores one should allocate for this application. One important aspect of cloud applications is to scale to efficiently use all available cores in a machine to serve client requests. In this experiment, I evaluate the drone application's ability to use all the cores available in an instance. Kubernetes makes it possible to specify how many cores are to be used per pod. I use these capabilities to vary the number of cores allocated for the service manager container.

Figure 4.3 shows the throughput and 95^{th} percentile latency when using different numbers of cores on an m5.8xlarge AWS instance. Figure 4.4 shows the throughput normalized per core when using different numbers of cores. The result is that the drone object detection

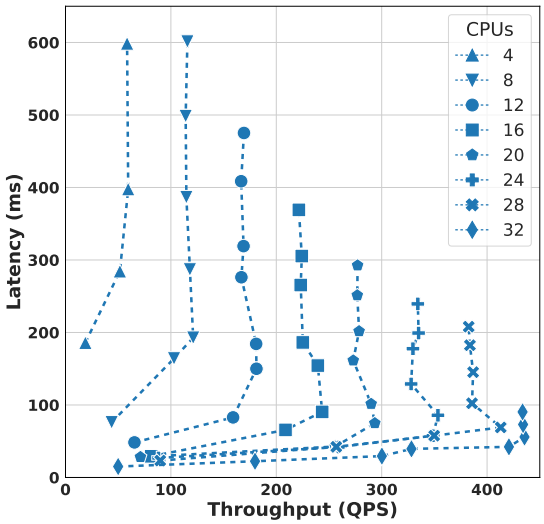

Figure 4.3: Price performance evaluation of the SSD-Mobilenet model using K8s' resource management.

application leverages all the cores on a machine effectively. The application experiences a slight dip in performance when using all the cores on the machine; this is due to the implementation causing frequent context switches in the threads. I explore this more in Section 4.4. This result shows that this edge application benefits from allocating more cores at edge servers.

4.2.3 What is the Impact of Image Resolution on Accuracy and Performance?

Drone cameras can take photos at different resolutions. Photo resolution presents a trade-off between response time and object detection accuracy. Larger images take longer to transfer to the MEC and longer to process, but are expected to lead to a higher object detection accuracy. When using lower-resolution images, smaller objects are harder for the models to detect due to the data lost when capturing or resizing the images. Using higher-resolution images can help detect smaller objects or, in the case of autonomous drone decision-making, objects that are further away, creating a larger time window for the vehicle to react.

Unfortunately, I could not find a dataset of images captured by a flying drone and

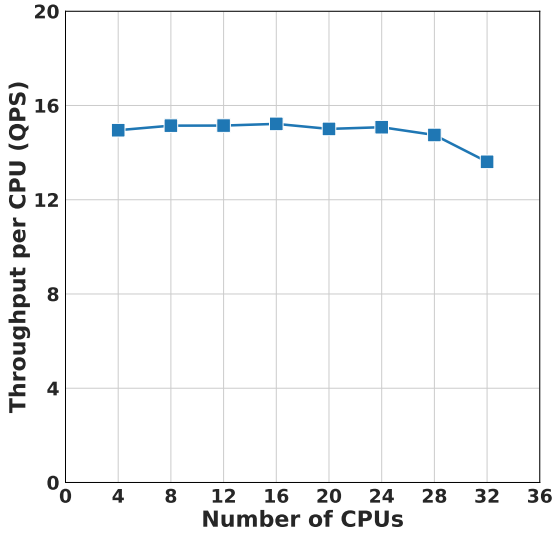

Figure 4.4: Throughput per CPU of the SSD-Mobilenet model using K8s' resource management.

corresponding ML models. The model I found with varying input resolutions is Google's EfficientDet [29] model suite. Table 4.2 shows the model names and the model image resolution. Although the team providing this suite states that these models are not suitable for latency-critical use [4], I use them just to demonstrate MECBench's ability to help explore the accuracy/performance trade-off of different models. The models in Table 4.2 use the same dataset. However, the resolution of images in the dataset is reduced to match the model's resolution.

Figure 4.5 shows the throughput and 95^{th} percentile latency for object detection using the four different models. The experiment uses the m5.8xlarge instance with 32 cores. Table 4.2 shows the detection accuracy of different models.

Figure 4.5 shows that increasing the image resolution significantly reduces the system performance. Using the highest-resolution model reduces the peak throughput by 92% the 95^{th} percentile latency is 500 ms. Table 4.2 shows the object detection rate of the EfficientDet suite when evaluated using the COCO Dataset. The table shows a noticeable improvement in the number of objects detected in the dataset, increasing with dataset resolution and reaching up to a 25% better detection rate in the highest-resolution model when compared to the model with the lowest resolution. The D1 model offers a mid point in terms of accuracy and performance between D0 and D3. This experiment demonstrates MECBench's ability to explore this trade-off.

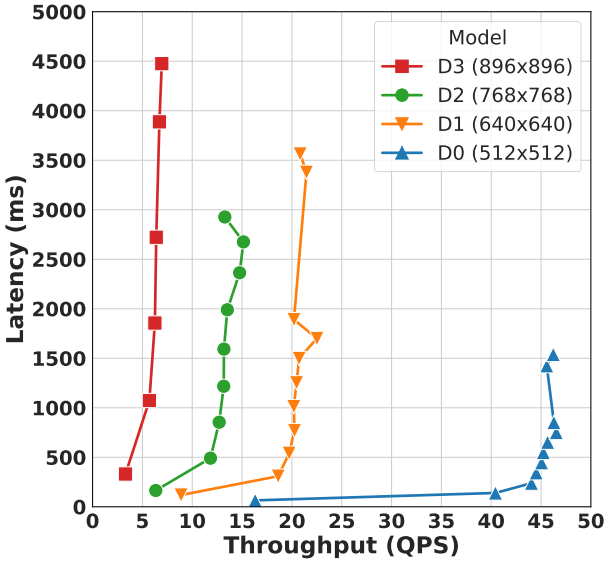

Figure 4.5: Performance evaluation of EfficientDet deployments with an $m5.8xlarge$ service manager.

Model Name	Resolution	Detection
D0	512×512	0.466
D1	640×640	0.519
D2	768×768	0.558
D3	896×896	0.632

Table 4.2: EffecientDet Model Detection Accuracy.

Network Condition	RTT (ms)	Download (Mbps)	Upload (Mbps)
5G	1	10,000.0	1,000.0
4G-LTE+	10	1000.0	500.0
4G-LTE	10	100.0	50.0
WiMAX	30	128.0	64.0

Table 4.3: Mobile network specifications used in our emulation.

Network Condition	RTT (ms)	Download (Mbps)	Upload (Mbps)
Net8.0	25	8,000.0	800.0
Net6.0	25	6,000.0	600.0
Net4.0	25	4,000.0	400.0
Net2.0	25	2,000.0	200.0
Net1.0	25	1,000.0	100.0
Net0.5	25	500.0	50.0

Table 4.4: Synthetic 5G network emulation specifications.

4.2.4 How Many Drones Can be Supported Using Different Network Technologies?

One of the main concerns when deploying an application on the edge is the required network performance in terms of throughput, latency, and packet loss. Applications often do not clearly express their network requirements or how network performance affects application performance. It is especially challenging to estimate the effect of network performance on an application in mobile networks because they offer asymmetric downlink and uplink performance. Using MECBench, I evaluate the performance of the SSD-Mobilenet model when deployed on an m5.8xlarge instance with different network conditions. I use MECBench's network emulation capabilities described in Section 2.4 to emulate a variety of networks. Table 4.3 shows the characteristics of the network standards I use in my evaluation. I also emulate a set of synthetic 5G networks (Table 4.4) offered publicly by different network providers due to the difficulty of deploying the 5G standard. I use the same methodology as in Section 4.2.1 to evaluate the performance of the model on each network condition.

Figure 4.6 shows the throughput and 95^{th} percentile latency of the same edge service running the object detection service on an $m5.8xlarge$ AWS instance. The figure shows the performance of the system under different network conditions. I see from the figure that networks with low bandwidth capabilities, like 4G-LTE and WiMAX, struggle to serve

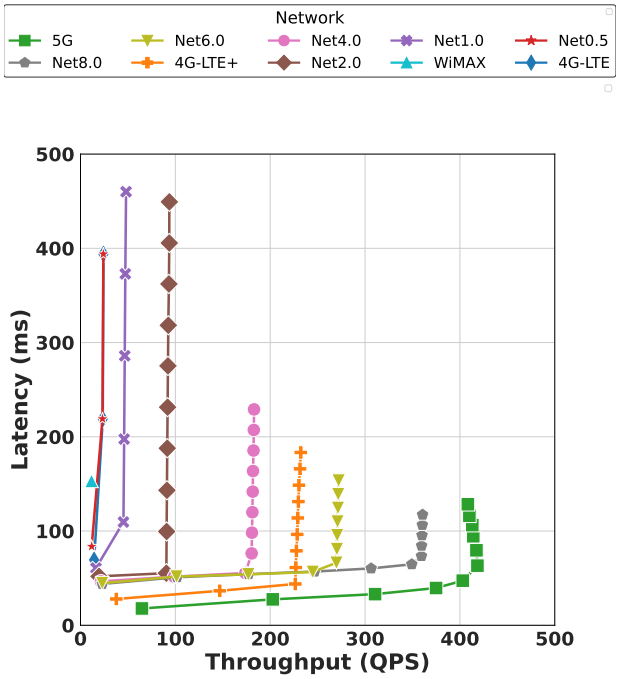

Figure 4.6: Throughput-latency figure for the object detection scenario under different networks.

any number of drones for request latencies less than $100ms$. On the other hand, networks like 4G-LTE+ and 5G, which have higher bandwidth capabilities, can serve around 20 and 32 drones, respectively, as shown in figure 4.7, which looks into the maximum number of drones that can be served by the edge service while maintaining a 95^{th} percentile latency of $100ms$.

4.2.5 What is the Impact of Data Compression on Application Performance?

Drone cameras produce images that are processed before sending them to the service manager. Compressing images before sending them to the edge service reduces the amount of data transferred over the network, but increases the computational overhead on the edge service [21]. In this experiment, I evaluate the effect of image compression on performance. I use the *Raw COCO* dataset, which contains raw images, and the *JPEG COCO* dataset, which contains JPEG compressed images. I use the same methodology as in Section 4.2.1 to evaluate the performance of the model on each dataset.

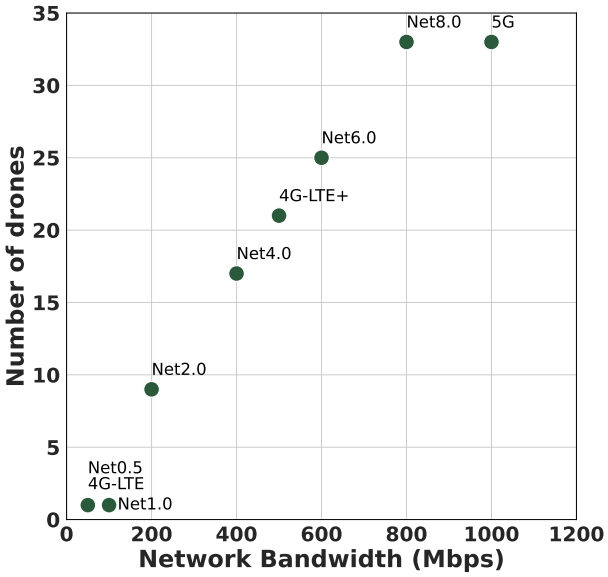

Figure 4.7: Maximum number of drones that can be served by the service manager while maintaining a 95^{th} percentile latency of 100ms under different networks.

Figure 4.8 shows the system throughput when using different network conditions. The figure shows that the system's performance is limited by the network bandwidth, where it only saturates the instance's resources when using the theoretical limits of a 5G network, when sending queries from the *Raw COCO* dataset. I ran the same experiment with the *JPEG COCO* dataset. Figure 4.9 shows the results of the experiment. The results show that image compression significantly improves performance for all networks except for the 5G network. Surprisingly, for the 5G network, compression reduces the system throughput by 5% compared to the *Raw COCO* dataset; that is because compression increases the computational overhead and introduces a performance bottleneck. This computational overhead was masked by the network bandwidth in other networks.

4.2.6 What is the Impact of Packet Loss on Application Performance?

To further explore the effect of changing the network conditions, I look into the effect of the conjunction of reliability and latency of the network on the system's overall performance. To conduct this experiment, I utilize MECBench's network emulation to inject variable probability of packet loss into the network on top of the network conditions provided in tables 4.3 and 4.4. This experiment's main objective was to evaluate the tradeoff of

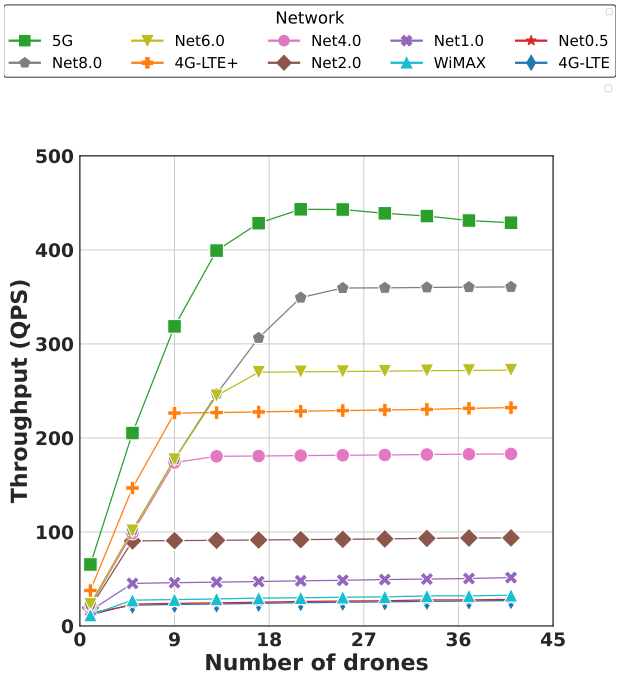

Figure 4.8: Raw COCO throughput evaluation.

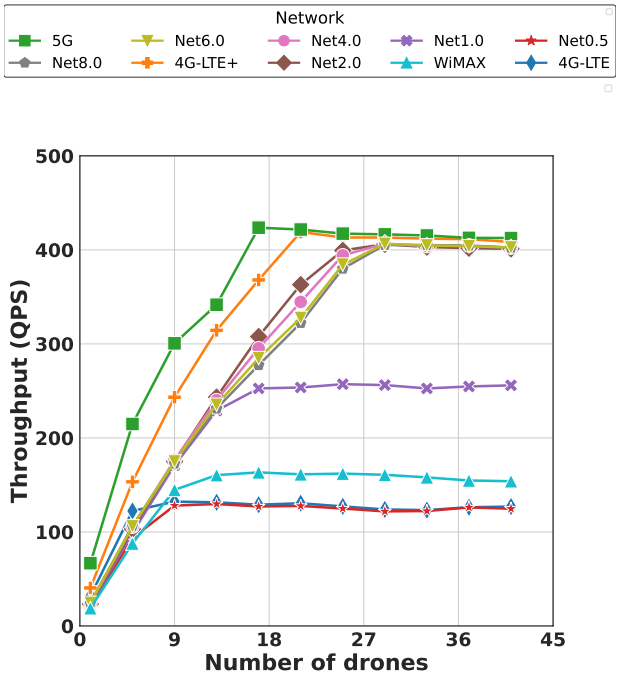

Figure 4.9: JPEG COCO throughput evaluation.

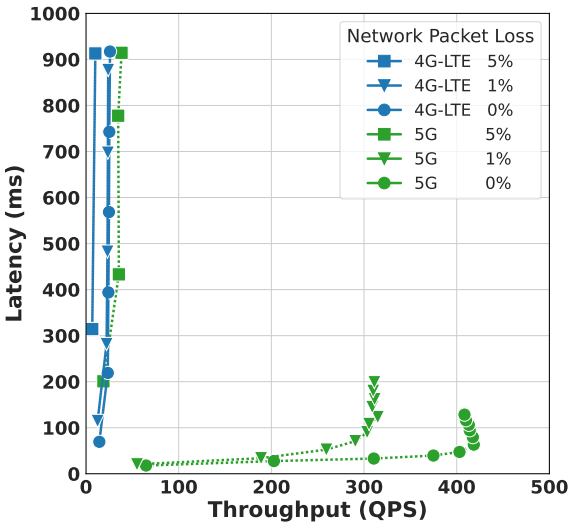

Figure 4.10: The effect of variable probability of packet loss on an application's performance.

using a reliable network with higher latencies versus a less reliable network with lower latencies. Figure 4.10 shows the throughput and 95^{th} latency when using *5G* and *4G-LTE* networks with different packet loss probabilities. The results show that, even with a high packet loss probability of 1%, a network offering the performance of the 5G specification outperforms 4G networks. Interestingly, a *4G-LTE* network with a packet loss probability of 1% performs similarly to a 5G network with a packet loss probability of 5% due to the increased number of re-transmissions that overshadows the 1ms RTT of the 5G network.

4.2.7 At What Speed Should the Drone Fly Under Different Network Technologies?

Sensor frame rates are usually controlled and limited to fit the needs of the application or the hardware limitations. Increasing the drone speed requires an increase in the query rate. Thus, reducing the drone's speed and its request rate is a parameter that can be modified to reduce the network bandwidth consumption per client at the price of slower inference rates on the drones, which in turn slows the reaction time of the drones.

In this experiment, I use the MultiStream scenario described in Section 2.1, configured to let the clients send data at a controlled rate, with a server-side queue of size 4 per client. The evaluation was conducted on an m5.8xlarge service manager instance serving

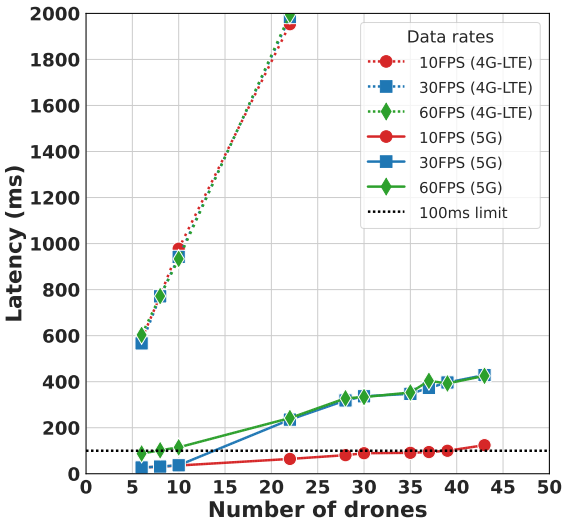

Figure 4.11: The effect of varying the frame rate of the data being sent to the service manager on the system's performance.

the SSD-Mobilenet model, using 4G and 5G emulated networks and with data rates varying from 10 frames per second (FPS) to 30 FPS. Figure 4.11 shows the response latency when increasing the number of drones with different data rates. The results show that a strict latency of $100ms$ can only be achieved when using a frame rate of $10FPS$ on a $5G$ network.

Another way to interpret this result is by looking at drone speed. If you assume that the drone's safety radius is half a meter and it can stop instantly when it detects an obstacle, then the drone should not travel more than $0.5m$ between each frame taken. This means that the drone serviced by the SUT cannot travel faster than $2.5m/s$ or $9km/h$ on a $5G$ network due to the $200ms$ reaction time provided by the system; 100ms to capture a new frame, and 100ms to process the frame. If the application is able to accept $30FPS$, the reaction time of the system is reduced to $133ms$, which allows the drone to travel at a speed of up to $3.8m/s$ or $14km/h$.

4.3 Text-Based NER Evaluation

The second evaluation scenario I conducted is to evaluate the performance of the service manager when serving a text-based NER model. This scenario studies the viability of serving a NER model on the MEC to assist grammar-checking applications, commonly

found in smartphone devices. In this scenario, a smartphone sends a paragraph of English text to the closest MEC server, which then extracts all the entities from the text and returns them to the smartphone to assist any applications that may require the information to correct the text. The entire process should be completed as fast as possible to provide a seamless experience to the user; thus, the 95^{th} percentile response time should be under 70 ms. Compared to the drone application, this application sends less data but can generate a higher number of requests per second.

The same questions brought up in the drone evaluation come up when deploying the application, with emphasis on the effect of increasing the number of clients on the response latency due to smartphones being more available and widespread than drones. Which AWS instance provides support for the greatest number of clients? Is allocating more resources for the application beneficial? Are all the networks capable of supporting the application? And what is the most important aspect of the network to consider when deploying the application?

This deployment allows the utilization of a less network bandwidth-demanding dataset while keeping similar processing requirements to the image-based deployment discussed in Section 4.2. The NER deployment serves my implementation of the SpaCy model, described in the model list in Section 3.3.1, deployed on an m5.8xlarge instance configured to start 33 SpaCy worker processes to perform the NER process on queries constructed from the *SQuAD* dataset (Section 3.3.2).

4.3.1 Which Networks Are Capable of Supporting the Application?

This application is directly impacted by network latency. I evaluate my implementation of the SpaCy model using MECBench on an m5.8xlarge instance with different network conditions, utilizing its network emulation capabilities.

Figure 4.12 shows the performance of the application when deployed on different networks. The figure shows that the system reaches service manager saturation before being limited by the bandwidth of the network, even in the case of the lower-end $WiMAX$ network. This is due to the small size of the queries being sent to the service manager which is typically in the range of a few KB.

The evaluation shows a grouping in the types of networks used in the evaluation based on their round trip time (RTT); the *4G-LTE* and *4G-LTE+* networks showed similar performance due to the similarity in their RTT of $10ms$, with a 28% drop in the throughput of the system when compared to the $5G$ network with 5 clients connected and gradually reaching the same throughput with the increase of the number of connected clients due to reaching the hardware limitation of the system.

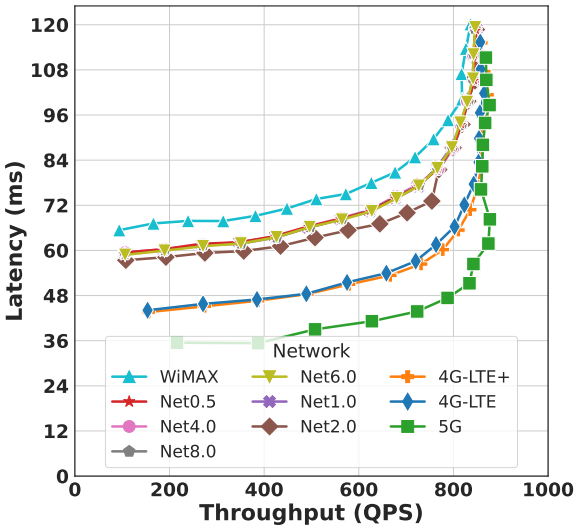

Figure 4.12: Performance of SpaCy NER on different network conditions.

Going back to my latency requirement, we can see that even the $WiMAX$ network, with a $30ms$ RTT, can support up to 33 concurrent clients with a 95^{th} percentile latency of $75ms$. On the other hand, the $5G$ network, with $1ms$ RTT, can support up to 41 concurrent clients but provides 25% faster responses when serving 33 clients, being limited by the performance of the application.

4.3.2 How Well Does the Application Scale with More Cores?

In this section, I look into the effect of allocating more cores to the application on the overall performance of the system. I utilize Kubernetes to gradually increase the number of available CPUs for the service manager running on an m5.8xlarge instance. I set SpaCy's worker processes to be one more than the number of CPUs available to the service manager, i.e., if the number of CPUs available is 4, the number of worker processes is 5.

The results (Figure 4.13) show diminishing returns on the throughput of the system as the number of CPUs available to the service manager increases. The growth in the number of clients supported per 4 cores drops from 9 clients to 4 clients. This is clearer when normalizing the throughput of the system per core, as shown in Figure 4.14, where the throughput per core drops by 55% when increasing the number of cores from 4 to 32. These results show a clear flaw in the scalability of the application on a single instance

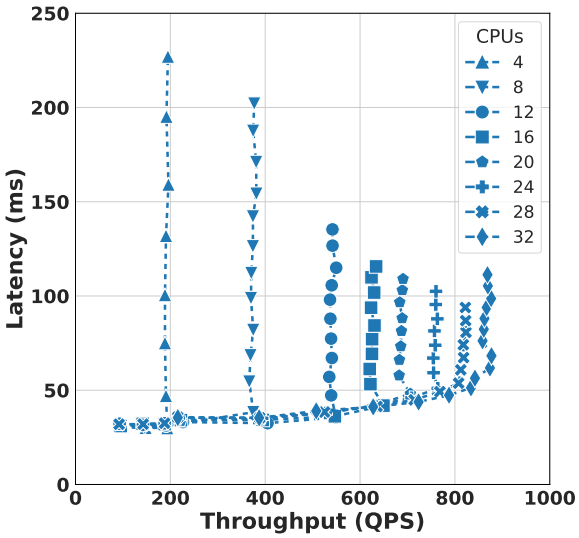

Figure 4.13: Price performance evaluation of the SpaCy model using K8s' resource management.

due to the inefficient task distribution of the service manager and the contention on the memory-shared task queues, which is not the case for the SSD-Mobilenet model.

4.4 Synthetic Service Manager Evaluation

I utilize the synthetic benchmarking models (Section 3.2) to further increase the granularity of the evaluation of the service manager instances. The evaluation looked into the effect of increasing the number of CPU cores and the amount of memory available to the service manager on the I/O performance and the CPU performance of the system. The goal of this section is to demonstrate MECBench's ability to generate a synthetic benchmark with varying CPU and I/O intensities.

The evaluation looked into two different types of instances to assess the impact of varying the service manager resources. I deployed the synthetic service manager on an m5.8xlarge instance, which has 32 vCPUs and 128 GB of RAM, and on an $m5.4xlarge$ instance, which has 16 vCPUs and 64 GB of RAM. The evaluation uses a SingleStream scenario with closed-loop clients. I disabled network emulation and used AWS's network with up to 10 Gbps throughput.

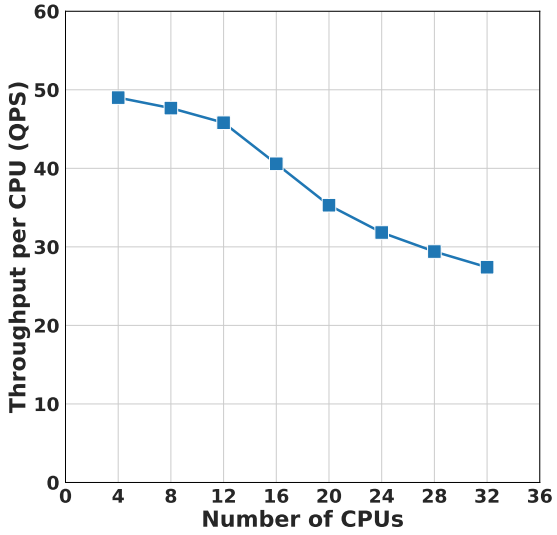

Figure 4.14: Throughput per CPU of the SpaCy model using K8s' resource management.

4.4.1 Does the I/O Throughput Depend on the Instance Type?

For the I/O evaluation of the system, I deploy the same storage device on both instances: AWS's General Purpose (gp3) SSD, capable of $3000IOPS$ and $125MB/s$ at its baseline [1]. For each instance, I conduct three experiments, varying the amount of data written per request and the number of clients connected to the service manager. For each request, the service manager generates and writes the specified amount of data to the storage device in a single write operation and *fsyncs* the data to disk before sending the response to the client. The amount of data written per request varies from $100KB$ to $10MB$. The results of the experiments are shown in Figure 4.15.

The results show that the service manager's storage throughput limit is similar in both instances, not being affected by the increase in available computing resources. The $125MB/s$ throughput is achieved when running 9 concurrent clients sending requests for $100KB$ writes and is immediately reached by a single client sending requests for $10MB$ writes.

4.4.2 CPU Performance Evaluation

In the CPU performance evaluation, I deploy the synthetic service manager on both AWS instances and run the SingleStream scenario with closed-loop clients, varying the process-

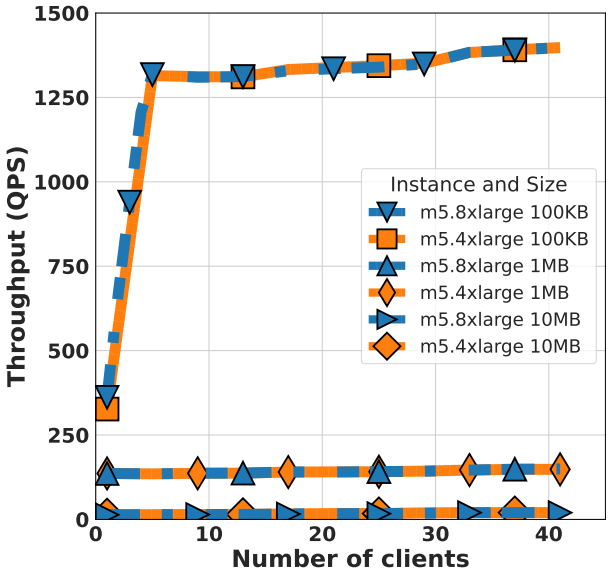

Figure 4.15: I/O performance of the service manager on different instances.

ing time of each request and the number of clients connected concurrently to the service manager. The results of the evaluation are shown in Figure 4.16.

Prior to the evaluation, I expected the results to show pseudo-linear scaling for the throughput of the system due to the calculation being limited by a specific amount of time, not by the number of floating-point operations performed. The results do show that trend when examining experiments with longer processing times. Sending $200ms$ requests to the service manager yielded throughput equal to $5\times$ the number of connected clients, and $2s$ requests yielded throughput of $0.5\times$ the number of connected clients. However, when experimenting with lower processing times, the results show that this is not the case for the service manager. Sending $20ms$ requests to the service manager yielded different results for the two instances, with the m5.8xlarge instance showing better scaling than the m5.4xlarge instance when increasing the number of clients. The m5.8xlarge instance was able to achieve 92% of the expected throughput with up to 32 concurrent clients, eventually dropping to 81% when reaching 40 clients. On the other hand, the m5.4xlarge instance performance dropped to 67% of the expected throughput at 32 clients and reached 55% at 40 clients. This is due to frequent context switching between the CPU cores, the impact of which is more pronounced on the m5.4xlarge instance due to the lower number of available cores.

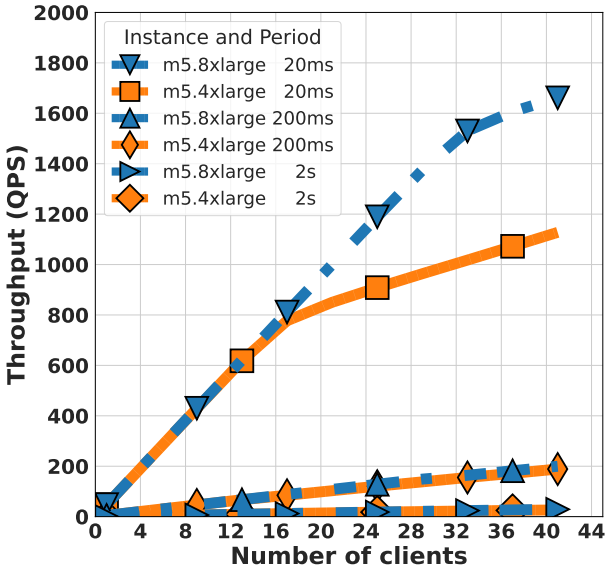

Figure 4.16: CPU performance of the service manager on different instances.

Chapter 5

Related Work

In this section I survey previous efforts on building benchmark tools for edge computing.

As the edge computing industry evolves and matures, the need for a standard benchmark that can be used to evaluate the performance of edge computing systems has fueled efforts to develop new benchmarking frameworks. Muñoz et al. [26], TPCx-IoT, YCSB [8], and MLCommons [25] all suggest their benchmarking tools and frameworks to standardize the evaluation process. I discuss MLCommons MLPerf [28] due to its relevance in the machine-learning inference community; evaluating applications that are prevalent in edge computing systems and its relevant structure to MECBench's design.

MLPerf Inference [28] is a one-of-kind benchmarking tool developed to answer the calls of hundreds of organizations looking for a reliable, comprehensive methodology to assess their machine-learning inference chips and systems. MLPerf describes a set of rules and guidelines that clarify the comparison of different systems and their performance, aiming to be the new standard for evaluating any machine-learning inference system while providing the flexibility and adaptability that accommodates the rapid evolution of machine-learning hardware and models.

MLPerf provides a framework that allows assessors to easily integrate their machine-learning models and associated datasets into a benchmarking process that is repeatable and reproducible. MLPerf alleviates the need for the assessor to develop workload generation algorithms by providing a set of community-inspired evaluation scenarios that are designed to represent realistic end-user scenarios. Along with a few other advantages that MLPerf has over other benchmarking frameworks, the previous points lead me to believe that MLPerf has proven to be a prominent benchmarking tool for machine-learning inference systems.

Currently, MLPerf is designed to have both ends of the evaluation process; workload generation and workload execution, deployed on the same machine. This design imposes

a set of limitations on the evaluation process, as it does not help evaluate the effect of hardware heterogeneity on the overall performance of the system, nor does it help evaluate the effect of the network, which are two key considerations for edge computing systems.

MECBench utilizes MLPerf's scheduling scenarios to evaluate the performance of distributed systems, something that the current implementation of MLPerf does not address. Although MLPerf's scenarios have been used in the evaluation of edge hardware [25], I believe that it is missing some aspects that can only be present in a benchmarking framework that is specifically designed for edge computing, i.e., network conditions. MECBench's distributed nature, along-side the network emulation layer and the network-specific API extensions, allow it to take into consideration crucial parameters that MLPerf did not natively support.

Bäurle and Mohan [7] also suggest a new benchmarking suite that explicitly targets the evaluation of edge infrastructures and their applications. ComB uses the same load generation and service separation as MECBench, representing a bipartite graph that connects load generators with the nodes that are able to service them. With this design, ComB can report the evaluation results of the distributed system after a result-aggregation process. The metrics that ComB provides, TrackEval [18], are concerned with the performance of the object-tracking application used in their evaluation, but can be extended using a custom-written script [19].

MECBench aspires to provide capabilities that I believe ComB is missing. One of the main drivers for the development of MECBench is the lack of a workload generation and scheduling standard in edge computing. ComB uses custom-written workload scheduling that differs between experiments, requiring the implementation of a new scheduling algorithm when extending its evaluation capabilities. MECBench combines the distributed nature and dataset extensibility of ComB with the fine-grained workload generation and scheduling capabilities of MLPerf [28] to provide a benchmarking framework that can be easily extended to evaluate a plethora of easily customizable scenarios using realistic datasets.

MECBench also addresses the orchestration of the evaluation components differently, utilizing already-existing orchestration engines to provide a more flexible and scalable deployment that ComB mentions as planned future work. This orchestration concept is further enhanced by the extra layer of network emulation, discussed in Section 2.4, that MECBench provides to evaluate the performance of edge networks, enabling it to assess non-existent deployments with ease.

EdgeBench [9] is a framework that evaluates the performance of the available edge platforms, like AWS's IoT Greengrass [6] and Azure's IoT Edge [24], using a set of key applications that are representative of edge computing use cases. EdgeBench implements three applications: a speech-to-text application, an image recognition application, and a scalar value generator application, and utilizes their workload characteristics to design

different offloading and preprocessing scenarios that help shed light on the effect of edge computing on the performance of, currently, cloud-only applications.

MECBench aims to extend the evaluation capabilities of EdgeBench by providing more flexible and scalable deployment options that utilize the existing orchestration engines, while also providing a greater level of extensibility and reproducibility of the evaluation process using a well-defined workload generation and scheduling library.

MECBench also provides the ability to conduct the evaluations in controlled environments without the need for a real deployment of edge hardware, utilizing its network emulation module as well as the capabilities of the orchestration engines to limit the resources available to the edge nodes.

Chapter 6

Conclusion

I present MECBench, a framework for benchmarking edge computing applications. MECBench's design is centered around high configurability and extensibility that focuses on reproducibility. MECBench facilitates the extension of the framework with new applications that can be used to evaluate the performance of Service managers. If an SUT is not available, MECBench can mimic the application workload using a pre-built synthetic benchmark. I demonstrate the utility of MECBench in answering a number of what-if questions in an edge application. I am able to detect bottlenecks in a selection of network conditions as well as assess the cost efficiency of the pricing of AWS instances on object detection and NLP models. Furthermore, I can compare performance-accuracy tradeoffs for the EfficientDet model suite.

MECBench plans to continue to expand with new features to keep up with the rapid evolution of MEC applications. I plan on extending the network communication layer to include other communication middlewares, e.g. MQTT, **[write explanation]**. Another planned extension is looking into implementing a security layer for the communication between MECBench's main components, LoadGen and Service Manager, due to the impact of the computational requirements of cryptography's encryption and decryption algorithms, as well as allowing MECBench to be deployed in public environments. I am also looking into potential MEC applications designed by the research community that look into using edge computing and fifth-generation network to target localization and civil architectural opportunities.

References

[1] Amazon ebs general purpose volumes. https://aws.amazon.com/ebs/general-purpose/, 2022. 40

[2] Amazon ec2 m5 instances. https://aws.amazon.com/ec2/instance-types/m5/, 2022. xi, 23, 24

[3] Amazon ec2 p2 instances. https://aws.amazon.com/ec2/instance-types/p2/, 2022. xi, 24

[4] efficientdet/d0 - tensorflow hub. https://tfhub.dev/tensorflow/efficientdet/d0/1, 2022. 28

[5] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org. 14

[6] Inc. Amazon Web Services. Aws iot greengrass documentation. https://docs.aws.amazon.com/greengrass/index.html, 2022. 44

[7] Simon Bäurle and Nitinder Mohan. Comb: A flexible, application-oriented benchmark for edge computing. In *Proceedings of the 5th International Workshop on Edge Systems, Analytics and Networking*, EdgeSys '22, page 19–24, New York, NY, USA, 2022. Association for Computing Machinery. 44

[8] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM*

Symposium on Cloud Computing, SoCC '10, page 143–154, New York, NY, USA, 2010. Association for Computing Machinery. 43

[9] Anirban Das, Stacy Patterson, and Mike P. Wittie. Edgebench: Benchmarking edge computing platforms, 2018. 44

[10] ONNX Runtime developers. Onnx runtime. https://onnxruntime.ai/, 2021. Version: 1.7.0. 14

[11] Inc. Google Inc. Dropbox, Skyscanner Ltd., and WeWork Companies Inc. grpc. https://grpc.io/docs/. 7

[12] Google. Protocol buffers. http://code.google.com/apis/protocolbuffers/. 7

[13] Stephen Hemminger, Fabio Ludovici, and Hagen Paul Pfeifer. *tc-netem(8) Linux User's Manual, Network Emulator (NetEm)*, November 2011. 7

[14] Matthew Honnibal and Ines Montani. spaCy 2: Natural language understanding with Bloom embeddings, convolutional neural networks and incremental parsing. To appear, 2017. 14

[15] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications, 2017. 14

[16] Teodor Alexandru Ionita. A two-tier storage interface for low-latency kubernetes deployments. Master's thesis, University of Waterloo, 2022. 9

[17] Nilesh Jain, Ashok Bhansali, and Deepak Mehta. Angularjs: A modern mvc framework in javascript. *Journal of Global Research in Computer Science*, 5(12):17–23, 2014. 17

[18] Arne Hoffhues Jonathon Luiten. Trackeval. https://github.com/JonathonLuiten/TrackEval, 2020. 44

[19] Daniel Kang. The tengo language. https://github.com/d5/tengo, 2021. 44

[20] Alexey N. Kuznetsov and Bert Hubert. *tc(8) Linux User's Manual, Traffic Control*, December 2001. 7

[21] Dong-U Lee, Hyungjin Kim, Mohammad Rahimi, Deborah Estrin, and John Villasenor. Energy-efficient image compression for resource-constrained platforms. *Image Processing, IEEE Transactions on*, 18:2100 – 2113, 10 2009. 31

[22] Tsung-Yi Lin, Michael Maire, Serge Belongie, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C. Lawrence Zitnick. Microsoft coco: Common objects in context. In David Fleet, Tomas Pajdla, Bernt Schiele, and Tinne Tuytelaars, editors, *Computer Vision – ECCV 2014*, pages 740–755, Cham, 2014. Springer International Publishing. 15

[23] Madhusanka Liyanage, Pawani Porambage, Aaron Yi Ding, and Anshuman Kalla. Driving forces for multi-access edge computing (mec) iot integration in 5g. *ICT Express*, 7(2):127–137, 2021. 1

[24] Microsoft. Azure iot edge documentation. https://learn.microsoft.com/en-us/azure/iot-edge/?view=iotedge-1.4, 2022. 44

[25] MLCommons. Inference: Edge v2.1 results. https://mlcommons.org/en/inference-edge-21/, 2022. 43, 44

[26] Manuel Osvaldo Jesús Olguín Muñoz, Junjue Wang, Mahadev Satyanarayanan, and James Gross. Demo: Scaling on the edge – a benchmarking suite for human-in-the-loop applicationss. In *2018 IEEE/ACM Symposium on Edge Computing (SEC)*, pages 323–325, 2018. 43

[27] Pranav Rajpurkar, Robin Jia, and Percy Liang. Know what you don't know: Unanswerable questions for squad, 2018. 16

[28] Vijay Janapa Reddi, Christine Cheng, David Kanter, Peter Mattson, Guenther Schmuelling, Carole-Jean Wu, Brian Anderson, Maximilien Breughe, Mark Charlebois, William Chou, Ramesh Chukka, Cody Coleman, Sam Davis, Pan Deng, Greg Diamos, Jared Duke, Dave Fick, J. Scott Gardner, Itay Hubara, Sachin Idgunji, Thomas B. Jablin, Jeff Jiao, Tom St. John, Pankaj Kanwar, David Lee, Jeffery Liao, Anton Lokhmotov, Francisco Massa, Peng Meng, Paulius Micikevicius, Colin Osborne, Gennady Pekhimenko, Arun Tejusve Raghunath Rajan, Dilip Sequeira, Ashish Sirasao, Fei Sun, Hanlin Tang, Michael Thomson, Frank Wei, Ephrem Wu, Lingjie Xu, Koichi Yamada, Bing Yu, George Yuan, Aaron Zhong, Peizhao Zhang, and Yuchen Zhou. Mlperf inference benchmark. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 446–459, 2020. 12, 14, 43, 44

[29] Mingxing Tan, Ruoming Pang, and Quoc V. Le. Efficientdet: Scalable and efficient object detection. *arXiv:1911.09070 [cs.CV]*, 2019. 14, 28

[30] Zongbin. Ng-matero. https://github.com/ng-matero/ng-matero, 2019. 17