MECBench: A Framework for Benchmarking Multi-Access Edge Computing Platforms

Omar Naman[1], Hala Qadi[1], Martin Karsten[1], Samer Al-Kiswany[1,2]
[1]*University of Waterloo, Canada*
[2]*Acronis Research*
{onaman, hqadi, mkarsten, alkiswany}@uwaterloo.ca

***Abstract*—We 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 a 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.**

We demonstrate MECBench's capabilities through two scenarios: an object detection scheme for drone navigation and a natural language processing application. Our evaluation shows that MECBench can be used to answer complex what-if questions pertaining to design and deployment decisions of MEC platforms and applications. Our 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.

I. Introduction

The fifth-generation (5G) mobile network architecture promises unprecedented characteristics in communication throughput, latency, and deployment density. These improvements enable new application domains that require these communication characteristics, such as autonomous driving, smart cities, drone applications, and Internet of Things (IoT). A central part of the 5G specification [1] is Multi-Access Edge Computing (MEC) [2], which refers to small compute clusters distributed across a mobile network to bring cloud services closer to users, thus avoiding the high latency imposed by contacting distant data centers. Service providers can design their own MEC or use a MEC service such as Amazon Outpost [3], Amazon Wavelength [4], and Azure Edge [5]. MEC plays a critical role in realizing the full potential of 5G and beyond mobile networks. MEC offers 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 service is complicated because of challenges in the MEC, the network, and the applications. First, there are no standard hardware or software specifications for the MEC, 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 the service providers' design and provisioning steps. These challenges complicate designing, provisioning, deploying, and billing MEC applications.

For instance, a typical scenario developers and service providers face is determining whether the provider can support a specific application with a specific service level agreement (SLA) in a particular locale. As an example of this scenario, a drone-based package delivery application could use a MEC-based mobile service to help with navigation. The operator expects up to 50 drones at any time and requires a response time of 100 ms for 95% of the requests sent by drones. The mobile service provider needs to answer many application- and platform-specific questions to support this application. For instance, the service provider needs to find 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? Which service level it needs to subscribe to in Amazon Outpost [3]? How much will it cost to support this application? The application developers need to explore questions related to the application's design. For instance, would data compression make it cheaper to run this application? How much would changing the drone speed help scale the system to support more drones? What is the accuracy/performance trade-off of different image resolutions?

To help application designers and service providers answer application- and deployment-specific questions, we present MECBench, an open-source benchmarking tool that can help answer what-if questions. MECBench takes an 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 mimic a range of network conditions, generate configurable client workloads, and tune the MEC resources available for a particular application. MECBench is designed to facilitate extending the benchmark with new datasets and applications. If an application cannot be deployed, MECBench offers a tunable synthetic workload generator that can mimic an application's compute and I/O load.

We demonstrate MECBench's capabilities through exploring two scenarios: an obstacle avoidance service for drones and a natural language processing (NLP) service for phone applications. In our evaluation, we explore questions related to the MEC hardware, such as the cost-performance trade-off

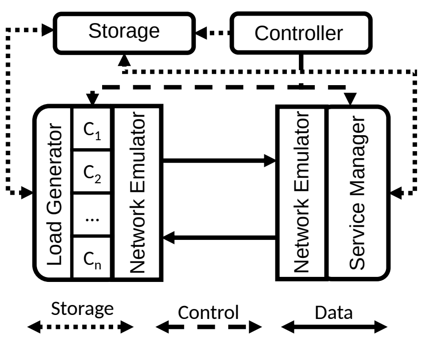

Fig. 1: MECBench architecture.

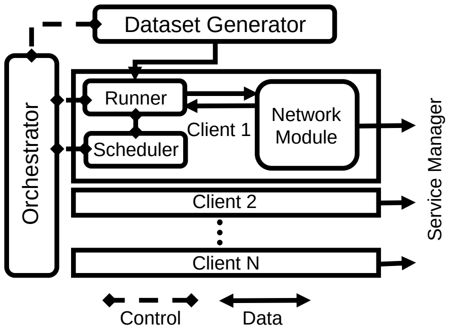

Fig. 2: LoadGen design.

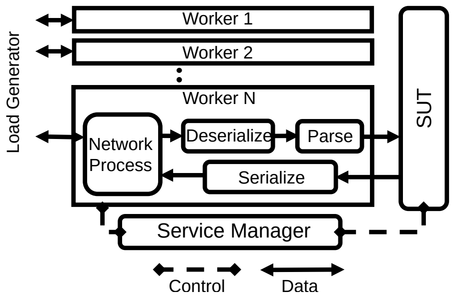

Fig. 3: Service Manager design.

of different hardware configurations; questions related to the network capabilities, such as exploring the impact of network conditions; and questions related to application design, such as exploring the impact of image resolution.

Our evaluation shows that MECBench is able to evaluate a range of scenarios and extract surprising insights about these applications. For instance, for the drone application, when measuring how many drones each AWS instance can support, we found that the cost per drone varies between different AWS instances with the smaller instances offering lower cost. Consequently, when supporting a large number of drones it is cheaper to use more lower-end AWS instances instead of large instances. Surprisingly, we found that GPUs do not bring performance gains to this use case, hence unnecessarily increase the deployment cost. Interestingly, while image compression improves performance for older networks, it brings little benefit for 5G networks. We discuss our findings in Section IV.

The implementation of MECBench aims to make it easier for the community to adopt this tool. We made the tool extensible to integrate new workloads and new MEC services, simplified the tool deployment through containerization and integrating it with Kubernetes, and simplified accessing the tool through providing a RESTful API. For wider impact, we will release MECBench as an open source project.

The rest of this paper is organized as follows. Section II details the design of MECBench. Section III discusses implementation details. Section IV presents the scenario-based evaluation. Section V discusses the related work. The paper is concluded with a summary in Section VI.

II. Design

MECBench comprises five main components: load generator, service manager, network emulator, storage, and controller. Figure 1, shows the interaction between the five components. The service manager runs the service under test (SUT) subject to the evaluation. SUT is the user-provided MEC service under 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.

A. Load Generator

The load generator (LoadGen) generates the client requests that are sent to the service manager. Each request has one or more queries. The service manager processes these queries using the SUT and sends the response back to 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. LoadGen is notified, if a response misses a deadline. The LoadGen component is flexible and can be configured to mimic different workload patterns. It can be deployed on multiple nodes.

Figure 2 shows the design of LoadGen on a single node. The main component is the orchestrator, which loads an experiment configuration, sets the workload metadata, starts and controls clients, and keeps track of the response time for each request.

The dataset generator is a module that defines the data used in an experiment and provides a list of data items that clients can retrieve. The orchestrator queries the 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 module 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. The second thread receives the response and sends the response time to the orchestrator. The scheduler (Figure 2) instructs the runner to send requests at a configurable rate. 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 the service manager and performs post-processing, such as verifying the correctness or quality of the response, and returning the response metadata to the orchestrator.

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.

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 one or multiple concurrent requests at a pre-configured rate. This allows running an evaluation with different workload patterns.

Workload configuration. The orchestrator can be configured to change the number of concurrent clients, the number of requests sent per second, and the number of queries in each request. These configuration values are defined in the workload configuration file.

LoadGen can be run on multiple nodes, which allows the generation of workloads that can saturate system components. This multi-deployment option of LoadGen can also be configured to generate workloads with different patterns on each instance of LoadGen.

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

- **SingleStream**: This scenario represents applications concerned with response time. The requests are generated in a closed loop and contain a single query. 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 different sources. For example, the requests contain multiple queries representing multiple sensors, generated at a constant time interval between each query. LoadGen will skip generating a request if the last request does not complete in time and the maximum number of concurrent requests is reached.
- **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. LoadGen pushes all queries in a dataset to the service manager to process at once and measures how long it takes to process a complete batch. The collected results are then analyzed to find the throughput of the service manager measured in queries per second (QPS).

B. Service Manager

The service manager manages the SUT. Figure 3 shows the design of the service manager. 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 and sending the results of all the queries back to the service manager. The service manager provides flexible support for different SUT models. The SUT can run in a separate process and communicate using OS interprocess communication techniques, or as a library as part of the service manager process. While the service manager implements several parsers for different applications, it offers an API that can be extended to implement custom parsers.

C. Communication Layer

MECBench provides a communication layer between LoadGen and the service manager to abstract the communication details, including network protocols and serialization. Our implementation uses Google's remote procedure call framework *gRPC* [6] via its streaming API. Each client creates a gRPC stream connection to its corresponding thread at the service manager. The stream is used throughout the test to send all client requests. The same mechanism is used for concurrent requests, which simplifies open-loop clients. Furthermore, gRPC supports setting deadlines for requests. The communication layer also offers serialization and deserialization of requests and responses using the Protocol Buffers (Protobuf) [7] format.

D. 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) [8], using the network emulation (NetEm) [9] module. NetEm provides the ability to emulate a variety of network conditions, including:

- **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 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.

MECBench uses these capabilities to facilitate evaluating services under different network technologies and conditions. MECBench allows having 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.

E. 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 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.

F. MECBench Controller

This component is responsible for orchestrating the system's deployment, starting and configuring experiments and workloads. The controller exposes a RESTful 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.

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 using the same service manager deployment for multiple experiments.

G. 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 codebase.

LoadGen Extension. Creating a new evaluation use case is done by extending two main classes of LoadGen's API: $Dataset$ and $Runner$. The $Dataset$ API to be extended is defined as follows:

- **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. 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.
- **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 follows:

- **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 facilitate running mutliple clients on the same LoadGen node.
- **Constructor**: The constructor of $Runner$ receives the $Dataset$ instance to be used in the evaluation.

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.
- **parseQuery**: Deserialize the query received from the LoadGen client to a format that can be passed to the running SUT through **processQuery**.
- **processQuery**: Pass the deserialized/parsed query to the running SUT for further processing. The SUT returns 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.

III. Implementation

The implementation of MECBench aims to make it easier for the community to adopt this tool. We made the tool extensible to integrate new workloads and MEC services, simplified the tool deployment through containerization and integrating it with Kubernetes, and simplified accessing the tool through providing RESTful API.

MECBench is implemented in around 8,000 lines of C++ and 2,500 lines of Python code. We have implemented two versions of the service manager: a C++ version and a Python version to integrate machine learning models 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 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 [10], a single-node benchmark for machine learning models. The provided synthetic workloads mimic compute- and I/O-workloads. Kubernetes is used to deploy MECBench's components: 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.

A. LoadGen

The LoadGen implementation uses a pool of threads to run client modules.

Runner. The protocol used by the runner to communicate with the service manager carries a stream of bytes. 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 a response to a request and to record the query's latency.

Dataset. The main dataset module 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.

B. Service Manager

The service manager runs the SUT, usually deployed on a 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.

C. Support for Machine Learning

MECBench comes prebuilt with machine-learning models that can be used as SUTs for evaluation. These models are implemented using the Python service manager. We have implemented three machine learning SUTs starting from their implementation in the MLPerf benchmark [10]:

- **SSD-Mobilenet [11]**: An object detection model, designed for computationally limited devices, that utilizes a single-shot detection (SSD) algorithm to detect and label objects in images. The model is implemented in MECBench using the ONNX Runtime [12] library, utilizing the SSD-Mobilenet model used by MLPerf [10].
- **EfficientDet [13]**: 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. Higher resolution models require more computation but provide more accurate inference results. This suite of models is integrated in MECBench using their original TensorFlow [14] implementation.
- **SpaCy [15]**: SpaCy is a high-performance NLP library. SpaCy is not designed to run as a service, it only supports running a single task at a time. To create a SpaCy service that can handle multiple requests we explored a design with a pool of SpaCy processes ready for serving client requests. The service manager chooses the next available process to handle a new incoming request and uses inter-process communication to submit a request to the process.

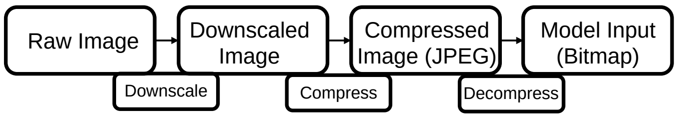

Fig. 4: Image pre-processing stages.

Machine Learning Datasets. Along with the machine learning SUT implementations, we have built a set of datasets that contain items used as requests for each machine learning SUT. An image pre-processing pipeline shown in Figure 4 prepares the images for consumption by the machine learning models.

- **Raw COCO and JPEG COCO.** We have pre-processed Microsoft's COCO dataset [16] for use in SSD-Mobilenet and EfficientDet SUTs. For *Raw COCO*, each image is downsized to match the input of each model and converted to a bitmap image ready to be passed to the model. On the other hand, *JPEG COCO* is composed of the same images as *Raw COCO* but compressed using JPEG. *Raw COCO* reduces the overall processing time of the SUT, but poses a bandwidth limitation due to a larger image size, while *JPEG COCO* is more bandwidth-efficient (Figure 5), but requires more processing time to decompress the images before passing them to the model.
- **Labeled Raw COCO.** We have also implemented a dataset that targets the accuracy of the models. We use the raw COCO dataset paired with the ground truth of the dataset. The ground truth is used in the post-processing step to measure the response accuracy.
- **SQuAD.** We have extracted the paragraphs from the SQuAD dataset [17], a question-answering dataset. We use these paragraphs as queries to be sent to the SpaCy SUT to perform Named Entity Recognition (NER). The size of the queries in this dataset is small compared to the other datasets, shown in Figure 6, in which 98% of the queries are less than 1500 bytes.

D. 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 evaluates the I/O throughput and latency of the service manager using a single file. Queries describe operation parameters such as total data, data written per operation, and whether to *fsync* data to disk after each write.
- **CPU Model**: This benchmark targets the performance of the CPU by continuously performing floating-point divisions for a set amount of time specified by the queries.
- **Requests Model**: This benchmark 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 time specified by the query.

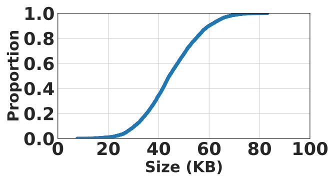

Fig. 5: Size of samples in JPEG COCO.

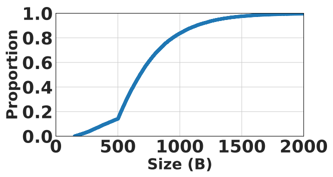

Fig. 6: Size of samples in SQuAD.

E. *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.

F. *Deployment*

In the current implementation of MECBench, its components are containerized and deployable using an 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 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, in run-server mode LoadGen is deployed as a Kubernetes service. 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.

IV. Evaluation

A. *Evaluation Setup*

We demonstrate the capabilities of MECBench using two different scenarios. For each scenario, we run a set of experiments that evaluate a specific capability of the service manager. For our evaluations we use AWS, nevertheless, MECBench can be deployed on other cloud platforms or edge platforms (e.g., Amazon outpost or Azure Edge). MECBench's components are deployed on EC2 instances, and the instances are controlled by AWS's Elastic Kubernetes Service (EKS).

The experiments use a selection of the models and datasets described in Section III on different types of AWS instances [18]. These instances differ in the number of processors, the amount of RAM available, and the existence of hardware accelerators, as shown in Table I. All processors are from the 2^{nd} generation Intel Xeon Platinum 8000 series. LoadGen

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 I: AWS EC2 instance resources [18], [19]. The information was collected on August 8, 2022.

is deployed on a single instance of type m5.xlarge that can saturate the network link as well as the service manager resources. We run all our experiments in a single AWS region (us-east-2) in a single 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. The experiments are repeated 30 times. No online processing is performed by LoadGen during the experiments, and all datasets are preprocessed to a format that can be immediately consumed by the SUT and preloaded into memory.

B. *Drone Object Detection*

One of MECBench's goals is to evaluate the trade-off between offloading onboard processing to a more powerful edge node. 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 runs an edge service that processes photos coming from the drone and detects any objects in the drone's path. For the evaluation, we use the SSD-Mobilenet model. For this application, it is critical that the response time is low enough to leave sufficient time for the drone to avoid an obstacle. We set the condition that this scenario requires that the 95^{th} percentile latency of the response time must be less than 100ms.

To understand how best to deploy this application, service providers and application developers need to answer the following questions. Which AWS instance can support the application and is most cost-efficient? Should the service use more expensive instances with GPU support? How many cores should be allocated for this application?

Application developers can configure their application to reduce cost or improve performance. For instance, 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 does each networking technology

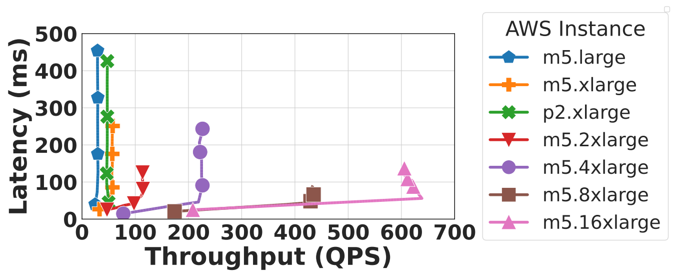

Fig. 7: Throughput-latency figure for the object detection scenario. The y-axis shows the 95th percentile latency.

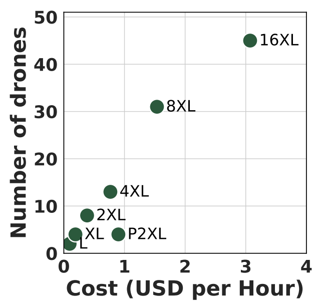

Fig. 8: Number of drones supported by different m5 instances with $100ms$ 95th percentile latency.

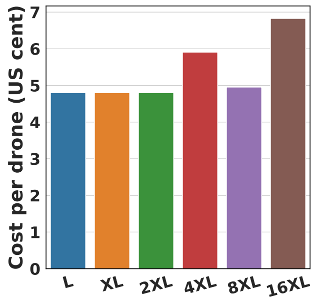

Fig. 9: The normalized cost per drone of different AWS m5 instances.

support? In this section, we show how we use MECBench to answer these questions.

1) Cost/Performance Trade-Off of AWS Instances: Cost/performance trade off is an important factor for practical deployments. In this section we start by exploring the cost/performance trade-off of different AWS instances (Table I). We note that AWS MEC solutions like Outpost [3] and Wavelength [4] offer a similar pricing models. LoadGen is configured to act as a set of closed-loop clients sending queries from the *Raw COCO* dataset. During the evaluation, we keep increasing the number of parallel clients until the service manager or the network is saturated. We also run SSD-Mobilenet on a GPU-based OnnxRuntime deployed on a p2.xlarge instance with an NVIDIA K80 GPU.

Figure 7 shows the throughput measured in Queries per Second (QPS) and 95th percentile latency of an edge service running an object detection service using the SSD-Mobilenet model. The figure shows the performance for different AWS instances, including p2.xlarge with a GPU. It shows that m5.large, m5.xlarge, m5.2xlarge, and p2.xlarge provide low throughput for a latency of 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 the 95th percentile latency being less than 100ms.

Figure 8 compares the different instances in terms of how many drones can be supported. The figure shows that only m5.4xlarge, m5.8xlarge, and m5.16xlarge can support more

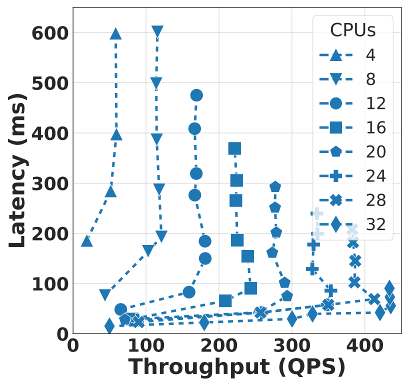

Fig. 10: Throughput-latency figure for the object detection scenario with different number of cores.

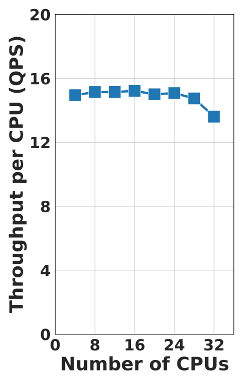

Fig. 11: Throughput per CPU of the SSD-Mobilenet model.

than 10 drones. Interestingly, for this application GPUs do not increase system throughput as p2.xlarge instance with a GPU costs 3 times more than m5.xlarge, while their performance is comparable. Figure 9 shows the normalized per-drone cost of each instance. The figure shows that m5.large, m5.xlarge, and m5.2xlarge have the lowest cost per drone of 4.8 cents per hour. This shows that when deploying the SSD-Mobilenet SUT on AWS, it is better to scale horizontally with more lower-end instances rather than vertically with more powerful instances.

2) Application Scalability: One important aspect of cloud applications is scaling to efficiently use all available cores in a machine to serve client requests. In this experiment, we evaluate the drone application's ability to use all cores of an m5 instance. Kubernetes makes it possible to specify how many cores are to be used per pod. We use this capability to vary the number of cores allocated for the service manager container.

Figure 10 shows the throughput and 95th percentile latency when using different numbers of cores on an m5.8xlarge AWS instance. Figure 11 shows the throughput normalized per core when using different numbers of cores. The drone object detection application leverages all the cores on a machine effectively. This result shows that this edge application benefits from allocating more cores at edge servers. 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 between threads.

3) Accuracy/Performance Trade-off of Image Resolution: 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. 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, we could not find a dataset of images captured by a flying drone and corresponding ML models. The model we found with varying input resolutions is

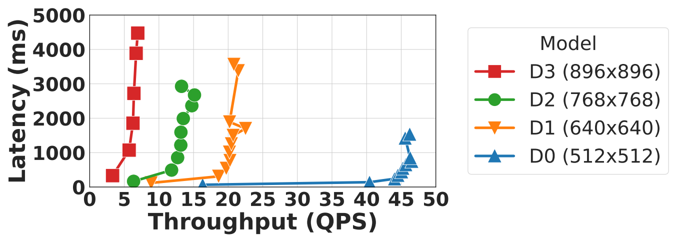

Fig. 12: Performance evaluation of EfficientDet deployments with an $m5.8xlarge$ service manager.

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

TABLE II: EffecientDet Model Detection Accuracy.

Google's EfficientDet [13] model suite. Table II shows the model names, input image resolution, and object detection accuracy. Although the team providing this suite states that these models are not suitable for latency-critical use [20], we use them to demonstrate MECBench's ability to explore the accuracy/performance trade-off of different models. The models in Table II use the same underlying dataset scaled to different image resolutions.

Figure 12 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 II shows the detection accuracy of different models.

Figure 12 shows that increasing the image resolution significantly reduces the system performance. Using the highest-resolution model reduces the peak throughput by 92% and the 95^{th} percentile latency is 500 ms. Table II 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 midpoint in terms of accuracy and performance between D0 and D3. This experiment demonstrates MECBench's ability to explore this trade-off.

4) Impact of Network Performance on Number of Drones: 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, we evaluate the performance of the SSD-Mobilenet model when deployed on an m5.8xlarge instance with different network conditions. We use MECBench's network emulation capabilities to emulate a variety of networks. Table III shows the characteristics of the network standards we use in our evaluation. We also emulate a set of synthetic networks (Table IV) offered publicly by different network providers due to the difficulty of deploying the 5G standard. We use the same methodology as in Section IV-B1 to evaluate the performance of the model.

Network Spec.	RTT (ms)	Download (Mbps)	Upload (Mbps)
5G spec. [1]	1	10,000	1,000
4G-LTE+ [21]	10	1000	500
4G-LTE [21]	10	100	50
WiMAX [22]	30	128	64

TABLE III: Network specifications used in our evaluation.

Network Name	RTT (ms)	Download (Mbps)	Upload (Mbps)
Net8.0	25	8,000	800
Net6.0	25	6,000	600
Net4.0	25	4,000	400
Net2.0	25	2,000	200
Net1.0	25	1,000	100
Net0.5	25	500	50

TABLE IV: Synthetic network specifications.

Figure 13 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. We see from the figure that networks with low bandwidth capabilities, like 4G-LTE and WiMAX, struggle to serve 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 14, which looks into the maximum number of drones that can be served by the edge service while maintaining a 95^{th} percentile latency of $100ms$.

5) Impact of Data Compression: 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 [23]. In this experiment, we evaluate the effect of image compression on performance. We use the *Raw COCO* dataset, which contains raw images, and the *JPEG COCO* dataset, which contains JPEG compressed images. We use the same methodology as in Section IV-B1 to evaluate the performance of the model on each dataset.

Figure 15 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 [1], when sending queries from the *Raw COCO* dataset. Figure 16 shows the results of the same experiment with the compressed *JPEG COCO* dataset. 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 longer transfer time in other

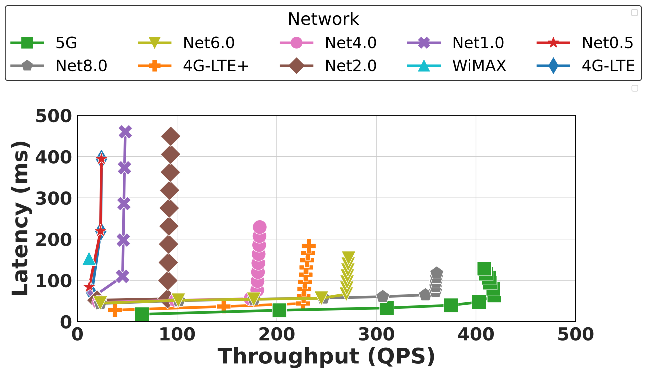

Fig. 13: Throughput-latency figure for the object detection scenario under different networks.

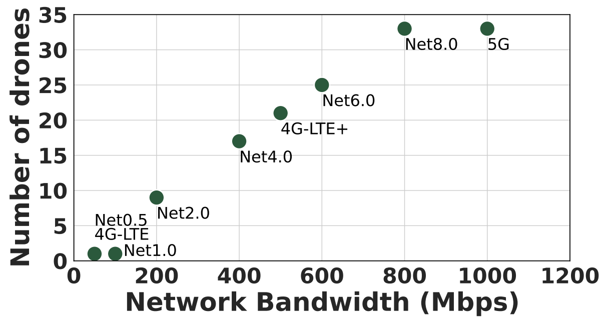

Fig. 14: Maximum number of drones that can be served by the service manager while maintaining a 95th percentile latency of 100ms under different networks.

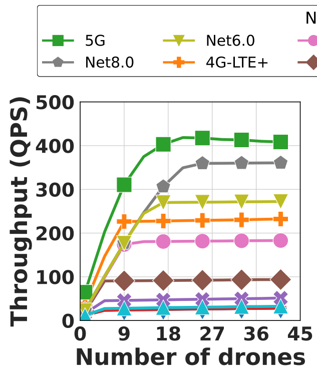

Fig. 15: Raw COCO throughput evaluation.

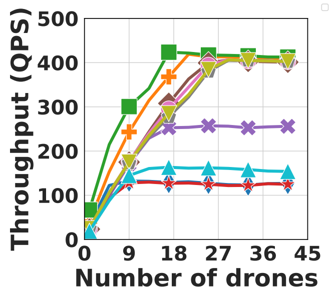

Fig. 16: JPEG COCO throughput evaluation.

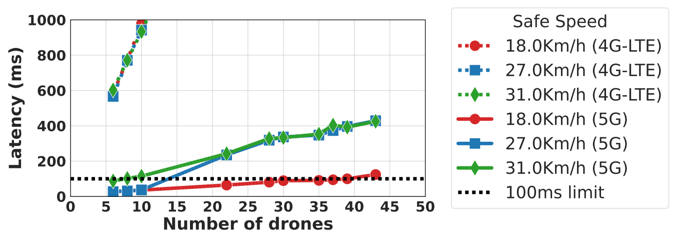

Fig. 17: The effect of varying the frame rate of the data being sent to the service manager on the system's performance.

networks.

6) Impact of Network Performance on Drone Speed: Sensor frame rates are usually controlled and limited to fit the needs of the application or hardware limitations. When reducing drones speed, we can reduce the sensors frame rate, hence reducing the load on the system and enables supporting more drones.

To evaluate the impact of network characteristics on drones speed, in this experiment, we emulate drone with a Multi-Stream scenario (Section II-A) and send requests at different rates to emulate different speeds. The server-side queue size is 4 requests per drone. We use an m5.8xlarge instance serving the SSD-Mobilenet model, using 4G and 5G emulated networks (Table III), using MECBench's network emulation capabilities, and with data rates varying from 10 frames per second (FPS) to 30 FPS. Figure 17 shows the response latency when increasing the number of drones with different data rates. By assuming that a drone should abide by a safety distance and its ability to instantly stop when it detects an obstacle, the drone must not travel more than the safety distance between each frame processed.

Equations 1 and 2 are used to convert between drone speed and perception-reaction time. Perception-Reaction (PR) is the time it takes for a drone to react to a perceived object by the sensors. We use the PR time to calculate the maximum speed of the drone in Equation (1). For instance, assume a safety distance of 1 meter must be maintained between the drone and the obstacles. Equations 1 and 2 tells us that a drone traveling $18km/h$ or $5m/s$ that is capturing a frame every $100ms\,(10FPS)$ needs a response from the server in under $100ms$. Following the same assumption, Figure 17 shows that the service manager can only serve this kind of service over a *5G* network. Over *4G-LTE*, drones traveling at the same speed cannot be served by the service manager due to its high latency $(600ms)$, the drones' speed must be less than 5Km/h to maintain the safety distance.

$$\text{Safe Speed} \times \text{PR Time} \leq \text{Safety Distance} \tag{1}$$

$$\text{PR Time} = \frac{1}{\text{Frame Rate}} + \text{Processing Time} \tag{2}$$

C. Text-Based NER Evaluation

The second evaluation scenario 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. The entire process should be completed as fast as possible to provide a seamless experience to the user; thus, the 95th 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

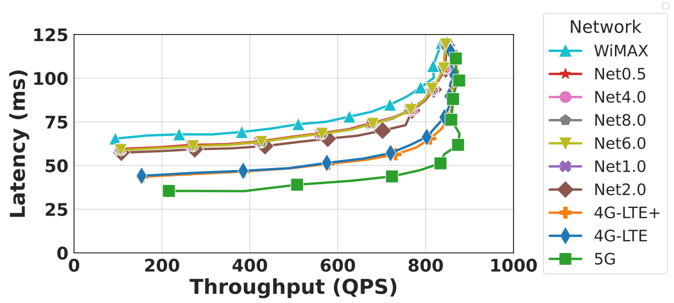

Fig. 18: Performance of SpaCy NER on different networks.

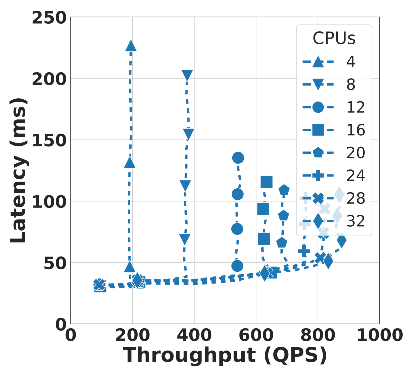

Fig. 19: Throughput-latency of SpaCy SUT using with different number of CPU cores.

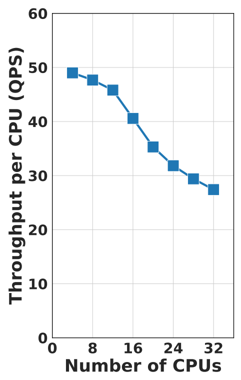

Fig. 20: Throughput of SpaCy with different number of cores.

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 application is less demanding in terms of network bandwidth but with a similar processing time compared to the drone application. The NER deployment uses the SpaCy model (Section III-C). We use an m5.8xlarge instance and configure it to start 33 SpaCy worker processes to perform the NER process on queries constructed from the *SQuAD* dataset (Section III-C).

1) Application Feasibility by Network Technology: This application is directly impacted by network latency. We evaluate the SpaCy model using MECBench on an m5.8xlarge instance with different network conditions.

Figure 18 shows the performance of the application when deployed on different networks (Tables III and IV). 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 that the results are clustered based on the round trip time (RTT) of the different network. *4G-LTE* and *4G-LTE+* networks show a 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.

Going back to our 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. The performance in the case of the 5G network is limited by the application implementation not the network capabilities.

2) Application Scalability: In this section, we look into the effect of allocating more cores to the application on the overall performance of the system. We utilize Kubernetes to gradually increase the number of available CPUs for the service manager running on an m5.8xlarge instance. We 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 19) 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 20, where the throughput per core drops by 55% when increasing the number of cores from 4 to 32. These results show a clear limitation in the scalability of the application and highlights the need for redesigning the application for higher concurrency.

V. Related Work

A few previous efforts proposed a benchmark tool for edge applications. ComB [24] provides a fixed load generator to evaluate edge applications. It includes an object tracking application and uses TrackEval [25] metrics that are often used to measure the performance of this application domain.

Defog [26] targets evaluating the impact a MEC may have on cloud applications. Defog evaluates a set of applications under different deployment modes: cloud-only, edge-only, and cloud-edge combined mode. The framework offers a fixed load generator and a set of applications that can be deployed on different cloud/edge configurations.

EdgeBench [27] evaluates the performance of edge platforms, such as AWS's IoT Greengrass [28] and Azure's IoT Edge [29], using speech-to-text, image recognition, and scalar value generator applications. BenchFaas [30] and EdgeFaasBench [31] explore the different methodologies of evaluating the performance of OpenFaas platforms. EdgeBench, BenchFaas, and EdgeFaasBench are geared towards evaluating serverless edge platforms using a set of implemented applications and workloads.

pCamp [32] focuses on comparing machine learning packages on commercially available edge devices. pCamp measures the latency, memory footprint, and energy usage of ML packages on an edge machine like a MacBook pro.

Unlike previous efforts, MECBench offers more flexible, extensible, and deployable options. MECBench offers a flexible and extensible workload generator that can be configured to generate a wide range of workloads, it uses containers to simplify deployments, and uses network emulation to emulate different network conditions. MECBench also allows evaluations in controlled environments without actual edge hardware, using network emulation and orchestration engine capabilities to limit resources available to edge nodes.

VI. Conclusion

We present MECBench, a framework for benchmarking edge computing applications. MECBench's design is centered around high configurability and extensibility. 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. We demonstrate the utility of MECBench in answering a number of what-if questions in an edge application. We are 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, we can compare performance-accuracy tradeoffs for the EfficientDet model suite. MECBench's source-code can be found at https://github.com/UWASL/MECBench

Acknowledgment

We thank Aladdin Saleh and Paul Howarth for their feedback. This research was supported by an NSERC Alliance grant and a grant from Rogers Communications.

References

[1] *Detailed specifications of the terrestrial radio interfaces of International Mobile Telecommunications-2020*, International Telecommunications Union Std., February 2020.

[2] "Driving forces for multi-access edge computing (mec) iot integration in 5g," *ICT Express*, vol. 7, no. 2, pp. 127–137, 2021.

[3] "Aws outposts family," https://aws.amazon.com/outposts/, 2022.

[4] "Aws wavelength," https://aws.amazon.com/wavelength/, 2022.

[5] "Azure private multi-access edge compute (mec)," https://azure.microsoft.com/en-us/solutions/private-multi-access-edge-compute-mec/overview, 2022.

[6] I. G. I. Dropbox, S. Ltd., and W. C. Inc., "grpc," https://grpc.io/docs/.

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

[8] A. N. Kuznetsov and B. Hubert, *tc(8) Linux User's Manual, Traffic Control*, December 2001. [Online]. Available: https://man7.org/linux/man-pages/man8/tc.8.html

[9] S. Hemminger, F. Ludovici, and H. P. Pfeifer, *tc-netem(8) Linux User's Manual, Network Emulator (NetEm)*, November 2011. [Online]. Available: https://man7.org/linux/man-pages/man8/tc-netem.8.html

[10] V. J. Reddi, C. Cheng, and e. a. Kanter, David, "Mlperf inference benchmark," in *ACM/IEEE Annual International Symposium on Computer Architecture (ISCA)*, 2020, pp. 446–459.

[11] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, "Mobilenets: Efficient convolutional neural networks for mobile vision applications," 2017. [Online]. Available: https://arxiv.org/abs/1704.04861

[12] O. R. developers, "Onnx runtime," https://onnxruntime.ai/, 2021, version: 1.7.0.

[13] M. Tan, R. Pang, and Q. V. Le, "Efficientdet: Scalable and efficient object detection," *arXiv:1911.09070 [cs.CV]*, 2019. [Online]. Available: https://arxiv.org/abs/1911.09070

[14] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, and et al., "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015, software available from tensorflow.org. [Online]. Available: https://www.tensorflow.org/

[15] M. Honnibal and I. Montani, "spaCy 2: Natural language understanding with Bloom embeddings, convolutional neural networks and incremental parsing," 2017, to appear.

[16] T.-Y. Lin, M. Maire, S. Belongie, J. Hays, P. Perona, D. Ramanan, P. Dollár, and C. L. Zitnick, "Microsoft coco: Common objects in context," in *Computer Vision – ECCV 2014*, D. Fleet, T. Pajdla, B. Schiele, and T. Tuytelaars, Eds. Cham: Springer International Publishing, 2014, pp. 740–755.

[17] P. Rajpurkar, R. Jia, and P. Liang, "Know what you don't know: Unanswerable questions for squad," 2018. [Online]. Available: https://arxiv.org/abs/1806.03822

[18] "Amazon ec2 m5 instances," https://aws.amazon.com/ec2/instance-types/m5/, 2022.

[19] "Amazon ec2 p2 instances," https://aws.amazon.com/ec2/instance-types/p2/, 2022.

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

[21] *Detailed specifications of the terrestrial radio interfaces of International Mobile Telecommunications-Advanced*, International Telecommunications Union Std., May 2012.

[22] *IEEE Standard for Local and Metropolitan Area Networks - Part 16: Air Interface for Fixed and Mobile Broadband Wireless Access Systems - Amendment for Physical and Medium Access Control Layers for Combined Fixed and Mobile Operation in Licensed Bands*, Std., 2006.

[23] D.-U. Lee, H. Kim, M. Rahimi, D. Estrin, and J. Villasenor, "Energy-efficient image compression for resource-constrained platforms," *IEEE Transactions on Image Processing*, vol. 18, pp. 2100 – 2113, 10 2009.

[24] S. Bäurle and N. Mohan, "Comb: A flexible, application-oriented benchmark for edge computing," in *Proceedings of the 5th International Workshop on Edge Systems, Analytics and Networking*, 2022.

[25] A. H. Jonathon Luiten, "Trackeval," https://github.com/JonathonLuiten/TrackEval, 2020.

[26] J. McChesney, N. Wang, A. Tanwer, E. de Lara, and B. Varghese, "Defog: Fog computing benchmarks," in *Proceedings of the 4th ACM/IEEE Symposium on Edge Computing*, 2019.

[27] A. Das, S. Patterson, and M. P. Wittie, "Edgebench: Benchmarking edge computing platforms," 2018. [Online]. Available: https://arxiv.org/abs/1811.05948

[28] I. Amazon Web Services, "Aws iot greengrass documentation," https://docs.aws.amazon.com/greengrass/index.html, 2022.

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

[30] F. Carpio, M. Michalke, and A. Jukan, "BenchFaaS: Benchmarking serverless functions in an edge computing network testbed," *IEEE Network*, pp. 1–8, 2022.

[31] K. R. Rajput, C. D. Kulkarni, B. Cho, W. Wang, and I. K. Kim, "Edgefaasbench: Benchmarking edge devices using serverless computing," in *2022 IEEE International Conference on Edge Computing and Communications (EDGE)*, 2022, pp. 93–103.

[32] X. Zhang, Y. Wang, and W. Shi, "pCAMP: Performance comparison of machine learning packages on the edges," in *USENIX Workshop on Hot Topics in Edge Computing (HotEdge 18)*, 2018.