Optimizing Microservice-based Applications for Cloud and Edge Networks

Undergraduate Thesis

by

Yan (Oscar) Yu

CS 4490Z
Thesis Supervisor: Hanan Lutfiyya
Course Instructor: Nazim Madhavi

Department of Computer Science
Western University, London, Ontario N6A 5B7, Canada
August 13, 2024

Abstract

Edge computing has been the subject of much attention in the software development space over the last several years as the limitations of traditional cloud computing models continue to be exposed by an increasing number of connected IoT and internet-enabled devices that require real-time computing. As this new computing paradigm becomes more prevalent in the industry, it is important that software is developed effectively to take advantage of the benefits that edge computing brings to the table.

In this study, we attempt to establish an understanding of performance patterns that will enable the effective design and development of distributed software systems that can be easily deployed and optimized for various architectures of computing models – primarily hybrid cloud edge networks.

Contents

1 Introduction

1.1 Edge Computing

Edge computing is a computing paradigm in which compute and data operations live on and are cached on the network edge, in addition to traditional data caching provided by CDNs in the cloud computing paradigm [4]. Findings show that the main advantage of the edge computing model is to reduce the time required to transfer large amounts of data to and from the cloud, as the amount of data produced by IoT and other devices on the network edge continues to grow every year [10]. This enormous amount of raw data being generated highlights weaknesses in the traditional cloud computing paradigm, as the bandwidth and network infrastructure required to transfer this becomes increasingly great.

For real time applications like video analytics, where the application demands the real-time ingest and processing of high-definition video sources [7], proposals have long been made to bring the cloud closer to the source devices [9]; thus reducing the necessary bandwidth and network latency of the system overall.

However, tradeoffs of bringing these operations to the network edge include limited computing resources, storage, and processing power. In addition, modern software applications are often hosted remotely on servers and compute units operated by large corporations such as Amazon's AWS, Google's GCP, or Microsoft's Azure through a PaaS (Platform-as-a-service) model, with edge resources being significantly more expensive than equivalent cloud offerings.

1.2 Microservice Architecture

Microservices are autonomous services deployed independently of each other, each performing some specific operation towards a single defined purpose [1]. Services can be developed and deployed independently of each other, maintaining only a shared communication interface. These properties make microservices ideal for deployment to distributed networks consisting of various edge or cloud nodes, as services can be independently deployed to the hardware or host that most optimally supports its performance characteristics without affecting other parts of the software system. These services are enabled by the use of containers or virtual machines (VMs) to support hardware agnostic deployments.

2 Research Objectives

(O1) Understand what applications may benefit from deployment to edge networks

(O2) Understand how to identify performance characteristics of individual components within a software application

(O3) Understand how applications can be developed to optimize for deployment across different network architectures

(O4) Understand how various network architectures can affect latency and user experience

3 Methodology

3.1 Application Development

Not all software is created equally, with a vast range of performance characteristics and behaviours across various applications when developed for differing use cases. For the purposes of identifying favourable conditions for edge deployments versus traditional cloud architectures, we developed a minimal microservice-architecture application for real-time object recognition. This design is an ideal representation of an edge-native application, demonstrating simultaneous requirements for high network bandwidth, low-latency interaction, and intensive compute across the application's various components. (O1)

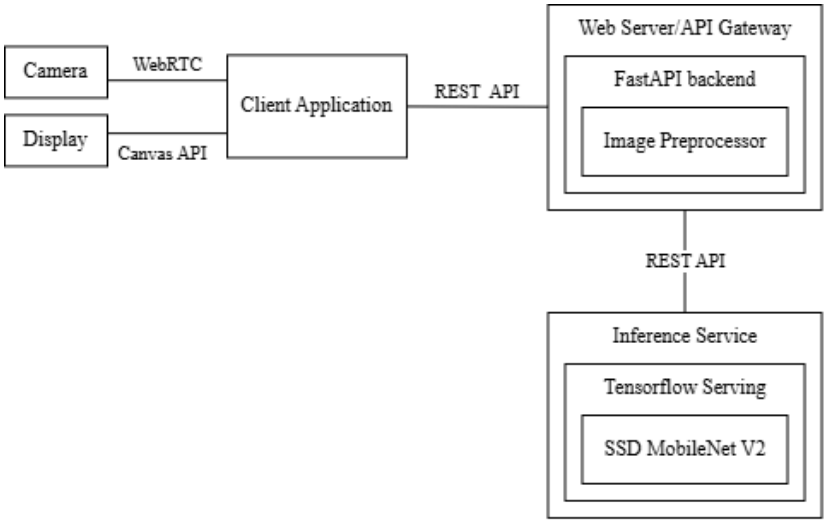

Figure 1: Sample application architecture for object recognition.

As shown Figure 1, the application is composed of a web interface, API gateway/web server, and the inference service. The web server and inference service were containerized using Docker in order to enable configurable, platform agnostic deployment. (O3)

The three components depend heavily on network bandwidth, as the client passes image data to the web server, which forwards it to the inference service for processing. Despite preprocessing on the client side to downscale snapshot resolution to 640 x 480, each request to the web server contains over 20 kilobytes of image data. (O2)

In a realistic scenario, 1080p real-time video analysis will require anywhere from 24-120 requests per second. With a 1920x1080 snapshot being around 2 megabytes, this

results in a lower bound for bandwidth of of 384 Mbps, or an upper bound of 1.92 Gbps. As a result of this bandwidth bottleneck, this type of application is an extremely good candidate for edge deployments [10], and could see significant improvements in various performance metrics versus a traditional cloud paradigm. (O1, O2)

3.1.1 Inference Service

To enable interfacing with a pre-trained object detection model, we leverage the open source TensorFlow Serving infrastructure to expose an interface for performing inference with the model through a REST API or gRPC calls. This is a production ready system that introduces minimal additional latency, making it suitable for the sample application.

The object detection model chosen for the sample application is a pre-trained copy of SSD MobileNet v2, a multi-layer object detection model based on Google's MobileNetV2 CNN [8] and the SSD (Single Shot MultiBox Detector) [6]. Sourced from the TensorFlow Model Zoo, this was trained on the COCO 2017 dataset. This model was chosen for its small size and ability to run efficiently on lower spec CPUs, as compute power and memory are often constrained on edge devices or servers.

Despite the limited scale of this model, inference through a CNN (Convolutional Neural Network) remains a compute intensive task that will likely be a performance bottleneck that scales with hardware performance on the host device, as the process relies on large numbers (see: billions) of parallel matrix operations [2]. (O2)

A note regarding the configuration of this service: while the option to run the service with GPU acceleration was available, the CPU-only deployment was chosen to reduce additional variables as a result of differing hardware configurations locally and on the cloud host.

Interfacing with the inference service was done through REST API calls, passing in image data through an HTTP POST request to the /predict endpoint with a body containing a preprocessed pixel array containing RGB colour channel values.

3.1.2 Web Server/API Gateway

The web server and API gateway functionality for this application are combined into a single package within this component. Built using Python on the FastAPI web framework, the web server accepts incoming HTTP POST requests to the /image endpoint. The logic for this endpoint is as follows:

1. Preprocesses the image using the Python PIL and NumPy libraries to parse the image data into a correctly shaped tensor represented by a multi-dimensional array,

which is then sent to the inference server for object detection.

2. After receiving a response from the inference server, the web server parses the JSON response and filters out objects with a confidence score of less than a set threshold; 0.5 for the tests performed for data collection.

3. The remaining objects are packaged back into JSON format before being sent back in the response to the POST request.

3.1.3 Client Application

The client application is a basic JavaScript application that uses WebRTC to capture the input video source from a connected device. It accepts video inputs of size 640x480 to 1920x1080, and uses the Canvas API to take snapshots of the input and downsize them to 640x480 before sending them to the web server via HTTP POST request. It parses the object data returned by the web server to draw bounding boxes over the video input as a graphical representation of the object detection process.

The source code for these components are located on Github in the following repositories:

1. `https://github.com/LordExodius/HCE-inference-server`

2. `https://github.com/LordExodius/HCE-web-server`

3. `https://github.com/LordExodius/HCE-client`

3.2 Deployment Architecture(s)

To demonstrate the feasibility of developing an application compatible with multiple network architectures and gather data regarding the performance of an identical application on different architectures, the web server and inference service were deployed in three configurations across a cloud host and locally, to emulate an edge server. (O3)

1. Local Web Server & Local Inference Service. This configuration represents a full edge deployment, with minimal geographical distance and high bandwidth between all components. The web server and inference service shared a resource pool consisting of an AMD Ryzen 9 5900X with 12 cores and 16GB of memory allocated to Docker.

2. Local Web Server & Remote Inference Service. This configuration represents a hybrid cloud edge deployment, with the inference service being deployed via Docker image to a Heroku instance running on a single Performance-L Dyno, with 8 CPU cores and 14GB of memory.

3. Remote Web Server & Remote Inference Service. This configuration represents a cloud native deployment, with everything except the client running on independent Heroku instances. Both component are running on Performance-L Dynos, with access to 8 CPU cores and 14GB of memory each.

4 Results

4.1 Data

The graphs below show the latency data collected from deploying the sample application across 3 different configurations of cloud and local hosting, which was used as a proxy for an edge server given the time and resource constraints during the data collection stage.

End-to-end (E2E) latency as shown here can also be interpreted as frametimes in a graphics processing context, as it also represents the time since the last frame drawn for the detection box overlay. We can invert it to calculate frames per second (FPS), which may be a more intuitive metric for how "smooth" or performant the system is:

$$\text{FPS} = \frac{1000\text{ms}}{\text{E2E Latency (ms)}}$$

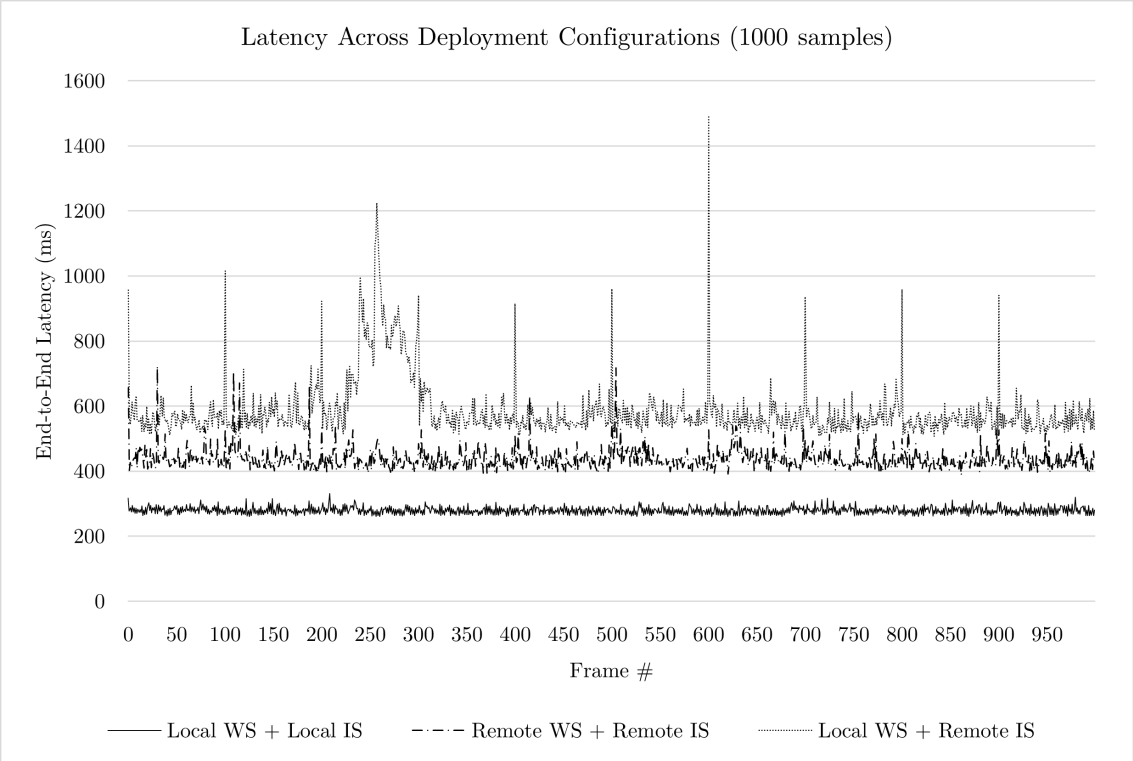

Figure 2: Graph showing E2E latency of a single video frame starting from the application client over 1000 samples. (WS: Web Server; IS: Inference Service)

Latency values were collected from the client application by calling the performance.now() method to return a high resolution timestamp in milliseconds prior to each web server request, and immediately after drawing the detection boxes on the output canvas. As such, this measurement includes time taken to capture a frame snapshot, the inference request, and the detection box rendering.

4.2 Latency Differences

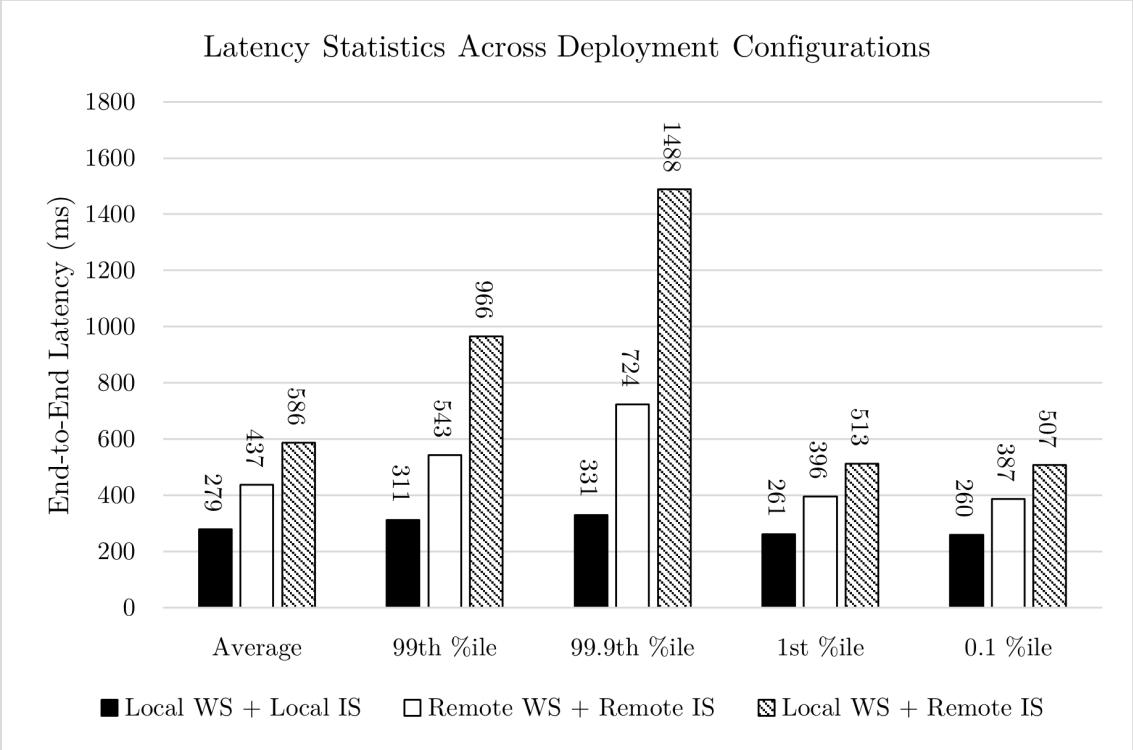

Figure 3: Common benchmarking performance metrics for real time graphics.

In Figure 3, we show several metrics for the latency samples collected from the three configurations of deployments described previously. These correspond to common graphics benchmarking metrics used for video games - another application in which users are highly sensitive to latency [5].

Percentile values were generated in Microsoft Excel using the PERCENTILE.EXC() function to exclude the first and last values from the 1000 samples collected, as the first frame sample appeared to be non-representative of the set as a result of having to establish the initial WebRTC connection from camera to client and the Session object for TCP

connection between the web server and inference service.

99th and 99.9th percentile values help illustrate consistency in E2E latency or frame timing, as they represent a reasonable lower bound for expected latency in the system. Large deviations between average and 99 percentile values indicate the presence of stutters or spikes in latency that result in much longer frame times, degrading user experience. We can observe these visually in Figure 1 as large vertical spikes, particularly in the Local WS + Remote IS deployment.

5 Discussion

5.1 Implications

These results can be used as a starting point to optimize real-time, microservice based applications. The specific latency results and performance data acquired from the testing performed are primarily applicable to real-time video analysis, particularly for those using machine learning inference in order to perform image or object recognition.

From these observations of the latency data, there appears a clear connection between the latency and the network architecture of an application – especially in the case where the performance characteristics are skewed towards network I/O operations and bandwidth. We can use similar statistical metrics to further analyze other software applications, identifying the optimal or most performant network architectures for any specific use-case or business requirement.

5.2 Limitations and Generalizations

Lack of flexibility in compute resources (see: no cloud GPU/TPU acceleration) limited the ability of this study to model common scenarios where traditional cloud computing offers similar or greater compute power at a reduced cost. With additional options for hardware acceleration, we could perform similar tests with varying levels of computing power for each component to provide a significantly more nuanced analysis of how to extrapolate an optimal network architecture based on various server configurations.

Additionally, the test results in this study may have been impacted by the use of a local deployment in lieu of a dedicated edge compute server. By deploying the local components to an actual edge service like AWS Wavelength, we could rerun the tests to greatly improve the accuracy and applicability of the results.

The results of this study can be generalized to other software applications utilizing CNNs to perform video analysis, and the conclusions reached regarding the effects of network architecture on latency can be used to support the optimization of other distributed microservice-architecture applications. Testing performed in this study can be repeated for any containerized microservice to determine the effect of various network architectures on the E2E latency of the system to support the conclusions found here.

6 Conclusions and Future Work

In performing this study, we aimed to identify factors that contributed towards the performance of a software application when deployed to the cloud and edge devices in various configurations. Objectives included identifying the types of applications that benefit from edge deployments and the performance characteristics of those applications (O1, O2); understanding the development process of an application designed for deployment across different network architectures (O3); and understanding how combinations of cloud and edge deployments can affect latency and user experience (O4).

Key results and conclusions:

1. Not all applications are equal. Performance characteristics such as heavy bandwidth and CPU resource dependencies can greatly impact the influence of network architectures on overall latency.

2. E2E latency of a networked system is heavily influenced by the proximity and latency of bandwidth heavy components. This is supported by the latency values being measurably lower for configurations where the web server component and inference service were deployed on a shared network, despite a lower bandwidth connection to the client in the full cloud deployment scenario.

3. Edge deployments exhibit significantly lower and more consistent latency results than cloud solutions, supporting the conclusion that edge computing offers significant advantages over a cloud computing paradigm, especially for real-time applications such as video or image analytics.

4. Network architecture optimization can be an effective tool to improve the performance of a software system, as supported by the significant differences in latency seen between the edge native, hybrid cloud edge, and cloud native deployments.

Future extensions to the testing and research conducted in this thesis include:

- Incorporating additional performance metrics such as *motion-to-photon* latency [3]. Additional metrics such as these help provide a more accurate picture of how the user experience is affected by changes in network architecture.

- Perform testing on commercial edge computing platforms such as AWS Wavelength to identify whether local deployments can be an effective proxy for the edge during application development and optimization.

7 Reference List

References

[1] Florian Auer, Valentina Lenarduzzi, Michael Felderer, and Davide Taibi. From monolithic systems to Microservices: An assessment framework. *Information and Software Technology*, 137:106600, September 2021.

[2] Michael Copeland. What's the Difference Between Deep Learning Training and Inference?, August 2016.

[3] S. George, T. Eiszler, R. Iyengar, H. Turki, Z. Feng, J. Wang, P. Pillai, and M. Satyanarayanan. Openrtist: End-to-end benchmarking for edge computing. *IEEE Pervasive Computing*, 19(04):10–18, oct 2020.

[4] Kacper Kubiak, Grzegorz Dec, and Dorota Stadnicka. Possible Applications of Edge Computing in the Manufacturing Industry—Systematic Literature Review. *Sensors (Basel, Switzerland)*, 22(7):2445, March 2022.

[5] Shengmei Liu, Atsuo Kuwahara, James J Scovell, and Mark Claypool. The Effects of Frame Rate Variation on Game Player Quality of Experience. In *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*, CHI '23, pages 1–10, New York, NY, USA, April 2023. Association for Computing Machinery.

[6] Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott Reed, Cheng-Yang Fu, and Alexander C. Berg. SSD: Single Shot MultiBox Detector. volume 9905, pages 21–37. 2016. arXiv:1512.02325 [cs].

[7] Sabuzima Nayak, Ripon Patgiri, Lilapati Waikhom, and Arif Ahmed. A review on edge analytics: Issues, challenges, opportunities, promises, future directions, and applications. *Digital Communications and Networks*, 10(3):783–804, 2024.

[8] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. MobileNetV2: Inverted Residuals and Linear Bottlenecks, March 2019. arXiv:1801.04381 [cs].

[9] Mahadev Satyanarayanan, Paramvir Bahl, Ramon Caceres, and Nigel Davies. The Case for VM-Based Cloudlets in Mobile Computing. *IEEE Pervasive Computing*, 8(4):14–23, October 2009. Conference Name: IEEE Pervasive Computing.

[10] Weisong Shi, Jie Cao, Quan Zhang, Youhuizi Li, and Lanyu Xu. Edge computing: Vision and challenges. *IEEE Internet of Things Journal*, 3(5):637–646, 2016.