Optimizing Microservice-based Applications for Hybrid Cloud 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 12, 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 paper, we attempt to establish an understanding of core principles 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

2 Background and Related Work

This thesis is based on existing work in multiple fields, including

2.1 Cloud Computing

2.2 Edge Computing

2.3 Microservice Architecture

3 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 hosting patterns can affect latency and user experience

4 Methodology

4.1 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)

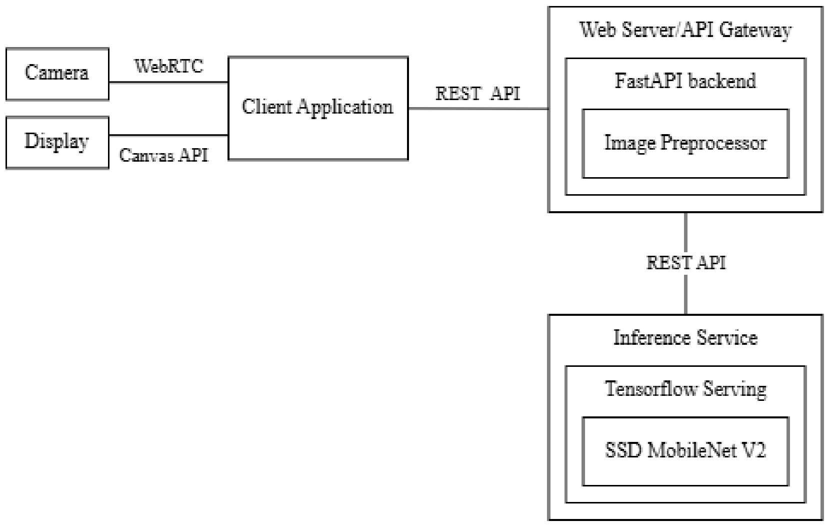

Figure 1: Sample application architecture for object recognition.

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

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.

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, this type of application is an extremely good candidate for edge deployments, and could see significant improvements in various performance metrics.

4.1.1 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 and the SSD (Single Shot MultiBox Detector). 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 on mobile CPUs, which was a constraint due to limited cloud hosting resources available at the time of this thesis.

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.

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.

4.1.2 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.

4.1.3 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`

4.2 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.

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 24 threads 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 Standard x2 Dyno, which is limited to 1GB 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 components have independent access to 1GB of memory and a limited share of the server's CPU.

5 Results

5.1 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:

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

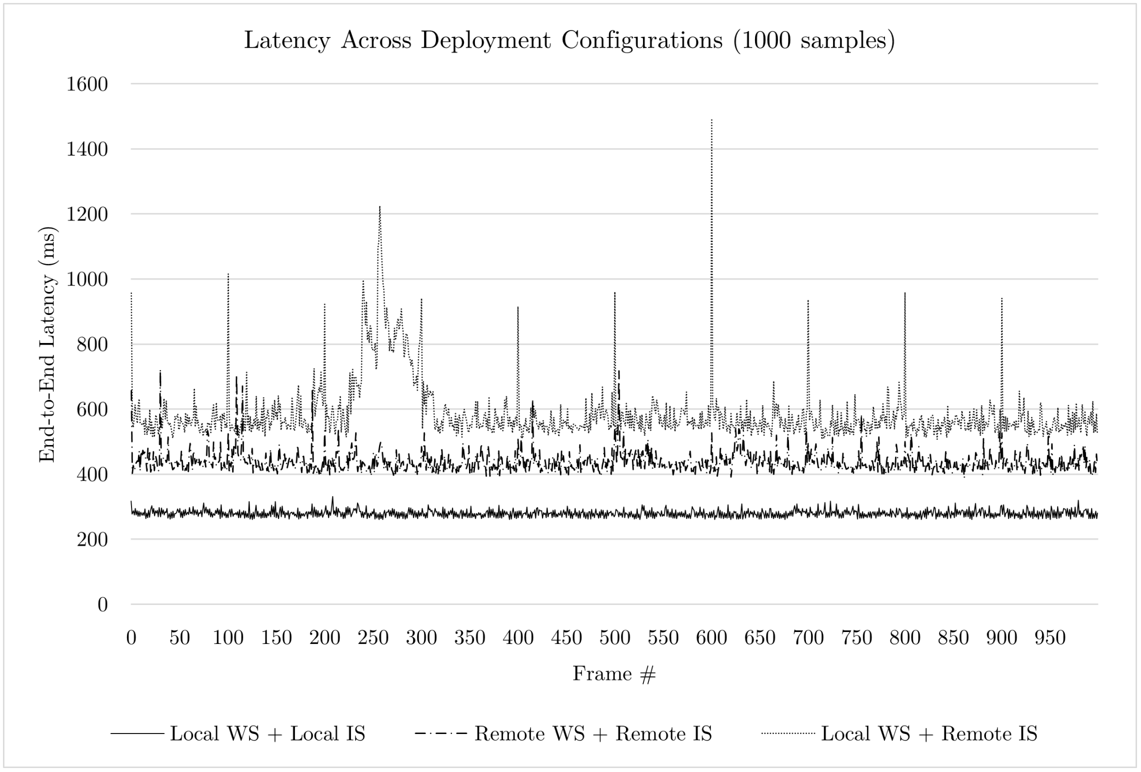

Figure 2: Graph showing E2E latency of a single video frame starting from the application client over 1000 samples. (WS: Web Server; IS: Inference Service)

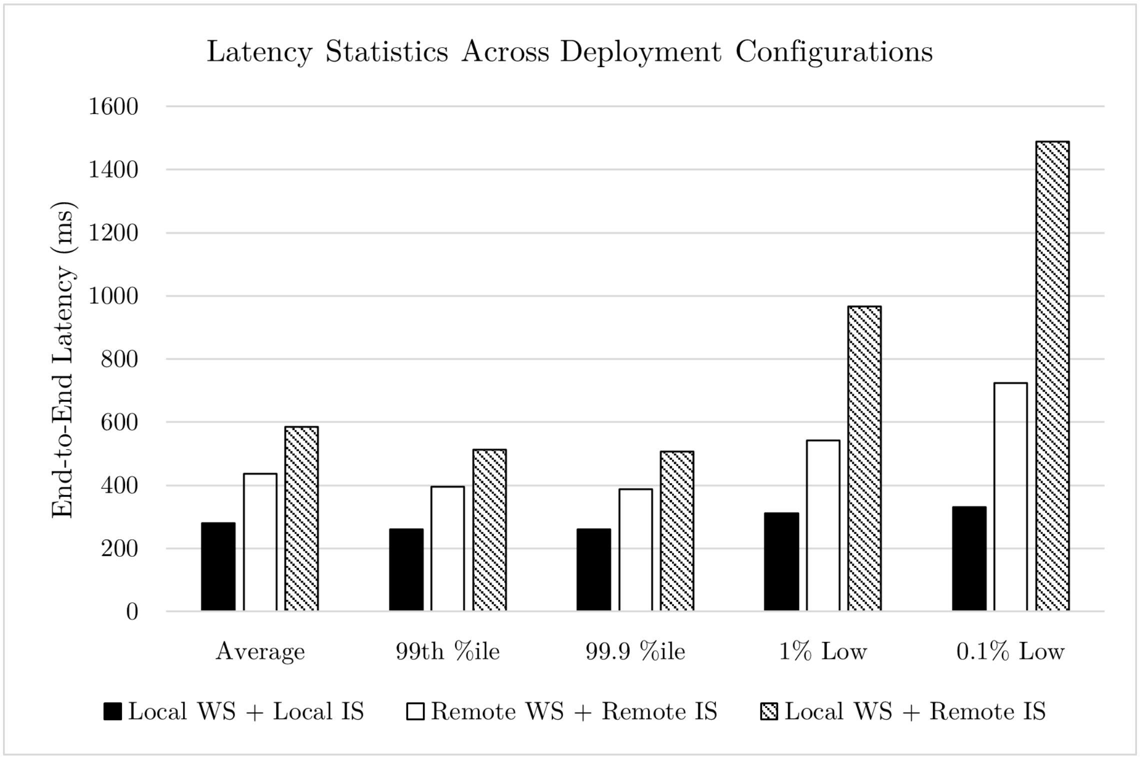

Figure 3: Common benchmarking performance metrics for real time graphics.

5.2 Latency Differences

6 Discussion

6.1 Implications

6.2 Limitations and Generalizations

- Hardware limitations - Limited model selection - Relatively simple system design - Lack of edge deployments - No service level latency monitoring - Lack of cloud hardware control - Only latency measurements, no hardware usage or identification of other bottlenecks

7 Conclusions and Future Work

- Perform more analysis with more performance analytics

8 Reference List