Serving AI using a Distributed Architecture

Final Year Project 2019/20 - Final Report

Waqas Ali (3035396771)

Supervisor: Dr. Heming Cui
Mentor: Shixiong Zhao

University of Hong Kong

May 3, 2020

Abstract

In recent years, artificial intelligence (AI) has penetrated multiple dimensions of people's daily lives by making the devices they use smarter. Fueled by data, AI programs imitate human intelligence in terms of their learning and behavioral capabilities. With such widespread usage, however, users demand improved functionalities and speed, pushing developers and data scientists to make their programs smarter amid industry competition. These smarter programs have to deal with more complexities, and developers consequently have to choose whether to prioritize the program's features or performance, posing a dilemma for them. This project proposes to design an AI application with a distributed architecture instead of a centralized architecture (the more common structure in the status quo) to improve its latency, efficiency, and throughput. As proof of concept, the project specifically examines a complex image analysis service. The project's objective is to develop tooling and foundation to automatically instantiate and compare distributed systems of a variety of specifications and scheduling algorithms. There are three milestones in this project. Firstly, the machine learning stage where test models have to be developed. Second, modifying the model serving to work in a distributed manner. Lastly, comparing distributed implementations which is the most crucial aspect of this project. The project shows that distributed implementations of AI applications lead to better latency, efficiency and throughput.

Acknowledgements

In addition to my supervisor and mentor who helped me with the technical aspects of my project, I would like to thank Ms. Mable Choi (HKU CAES) for her guidance in distilling and consolidating my work in a better way.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background

Artificial intelligence is an area of computer science that focuses on granting machines the ability to act intelligently [12]. It is a vast field with limitless applications and each application has its own unique solution. Machine learning, specifically, is a subset of artificial intelligence that learns from data. [13] Today we see ubiquitous applications of artificial intelligence such as spam filters [2], recommendations [11], virtual assistants, and self-driving.

Customers are demanding smarter and smarter capabilities in their machines, and this trend leads to a new set of software development challenges for AI developers. *Nvidia* summarises them with the PLASTER [16] framework:

- Programmability
- Latency
- Accuracy
- Size of Model
- Throughput
- Energy Efficiency
- Rate of Learning

These challenges carry over to the realm of machine learning since it is a subset of artificial intelligence. A machine learning application has two main stages:

1. Training (Learning from data)
2. Inference (Given an input, predicting an output)

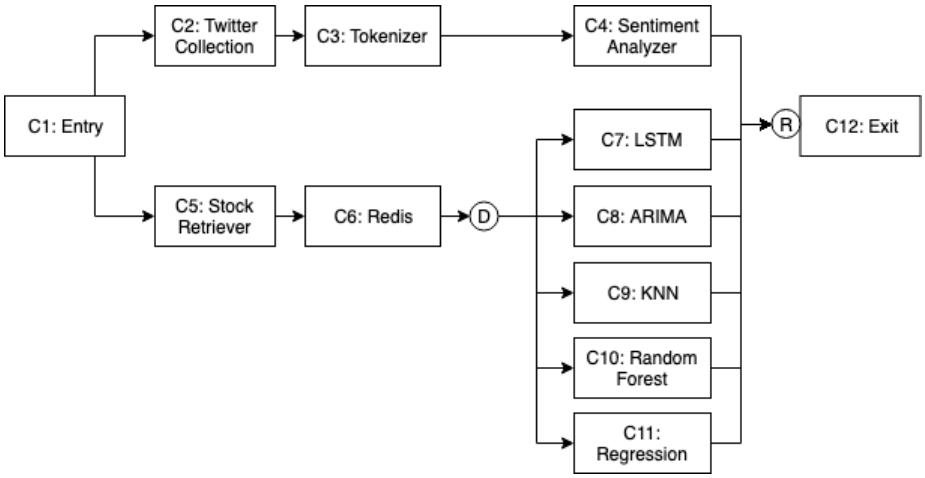

Figure 1.1: Inference pipeline of a stock price prediction service. C2 and C5 are data retrievers that fetch past stock data and twitter mentions of a specified stock symbol. C6 is a caching layer that allows skipping C7-C11 steps if a prediction has been recently made for a specific stock symbol. C7, C8, C9, C10 & C11 are ML models that compete against each other to predict a stock symbol's price. C4 is a sentiment analyzer whose results are taken into account for price prediction. As can be seen, it is a fairly complex pipeline composed of several different steps.

Take the example of an application that relies on a machine learning model to transcribe voice. Before the model can be used by the application, it needs to be trained. To do this, developers expose the model to hundreds of voice recordings to allow it to learn which sounds match to which words. Now, the application can use the model by sending it voice recordings and receiving transcribed text in return. In short, this process of predicting an output in response to an unseen input is inference, the second stage of machine learning as mentioned previously.

1.2 Motivation

Since end-users of machine learning applications are only concerned with inference, not training, the inference must be quick.

For inference, an input goes through multiple steps, known as a pipeline. Figure 1.1 is an example of a stock price prediction service's pipeline. As tasks in a pipeline increase in quantity and complexity, it can increase the *latency* (time taken) to execute all steps of the pipeline. Moreover, if a *centralized architecture* (single machine) executes the complete pipeline, it can create bottlenecks. For example, if a pipeline for input A is in progress, the pipeline for input B cannot start.

On a centralized architecture, all tasks have to be done sequentially (even

if they are independent of each other). This could take a long time and hence increase the *latency*.

Consequently, it is also possible for hardware resources to go underutilized and used for longer periods leading to decreased *efficiency*.

Moreover, until all tasks for a specific request have finished, processing for a new request cannot start. Thus, the service cannot handle a high number of requests in a given period i.e. *throughput*.

1.3 Objectives

Several methods could be considered to optimize latency, throughput, and efficiency of artificial intelligence applications. This project tackles this optimization problem by efficiently distributing the pipeline tasks of artificial intelligence applications over several machines.

Moving an artificial intelligence application from a centralized architecture to a distributed architecture not only requires deploying it on a network of multiple machines but modifying it to properly utilize the newly available resources.

A distributed application's success depends on how the application divides its tasks (job scheduling) and the quality/quantity of resources available for use. To figure out what works best we need a quick and reliable way of testing different job scheduling algorithms on networks of different sizes composed of machines of different specifications.

Consequently, the project's objectives are to develop the following programs:

1. An AI application with a complex inference pipeline that can work with different job scheduling algorithms.

2. A deployment method that can programmatically deploy an AI application according to provided specifications.

3. A web app to measure latency, efficiency and throughput in an architecture-agnostic way.

With the above-mentioned programs in place, we can confidently investigate and argue for or against using distributed systems for AI applications.

1.4 Contribution

With data, the project shows that latency, efficiency, and throughput of an AI application (concerns which were highlighted in the PLASTER framework [16]) can indeed be improved if a decentralized architecture is employed instead of a centralized architecture. Moreover, the project contributes tools to better conduct similar investigations.

1.5 Report Organization

Composed of artificial intelligence, distributed systems, and web development, the project's methodology involves several fields of computer science and software engineering to achieve the objectives mentioned in section 1.3. Naturally, that invites its own set of complexities and uncertainties. After literature review and project reasoning in chapter 2, chapter 3 accomplishes the important task of narrowing down and justifying the methodology. Following this, chapter 4 discusses in detail the implementation and technical details of the project. Most importantly, chapter 5 shows the results. Lastly, chapter 6 summarises this report and the project.

Chapter 2

Literature Review and Justification

2.1 Literature Review

Published in 2011, Mesos [9] was a pioneering work in resource-sharing across cluster-computing frameworks. It introduced a two-level scheduling mechanism that resulted in near-optimal performance and reduced resource under-utilization.

In 2017, Clipper [3] was introduced as a general-purpose prediction system that acted as a model selection/abstraction layer between applications and different implementations of a model. This ultimately allowed an AI model to be easily deployed on a distributed system while gaining benefits of modular architecture and increased performance.

Coming from the same lab as Clipper [3], InferLine [4], published in 2018, was a general-purpose specification designed to proactively optimize ML inference pipelines but also reactively control configurations. While generalizing across different frameworks, it achieved 7.6 times cost-savings and 32 times lower SLO miss-rate.

Lastly, GrandSLAm [10], published in 2019, was a general-purpose serving system built on a microservices architecture. It proactively optimized pipelines to meet latency requirements while improving throughput by 3 times.

2.2 Shortcomings

Mesos [9] is great at resource-optimization but it does not take into account the bigger picture. It is only concerned with how much resources the task at hand requires. Hence, it is not suitable for complex machine learning pipelines such as the one in figure 1.1.

Clipper [3] deals with different implementations of the same model. In the case where we have different models in a pipeline, we lose out on its benefits.

InferLine [4] and GrandSLAm [10] are interesting approaches to solving the problems highlighted in section 1.2. However, no standardized way exists to compare them. The scheduling techniques and architecture choices also vary. Not only that, but public implementations are also unavailable and hence further work cannot be done on them to improve them.

2.3 Project Justification

Regardless of how the pipeline optimization problem is solved, an implementation-agnostic way is needed to measure the results. In addition to its approach to solving the problem, this project also develops tools that can be used in the future to easily deploy and compare other solutions. Moreover, it builds a foundation, utilizing available resources at the University of Hong Kong, on which further work can be carried out.

Chapter 3

Methodology

From developing an artificial intelligence application to running it on a distributed architecture to comparing it to a traditional implementation, there are many steps to this project and without proper breakdown, it could get overwhelming quickly. The following sections describe and justify the steps through which the project aims to accomplish its objectives.

3.1 Choose AI Application for testing

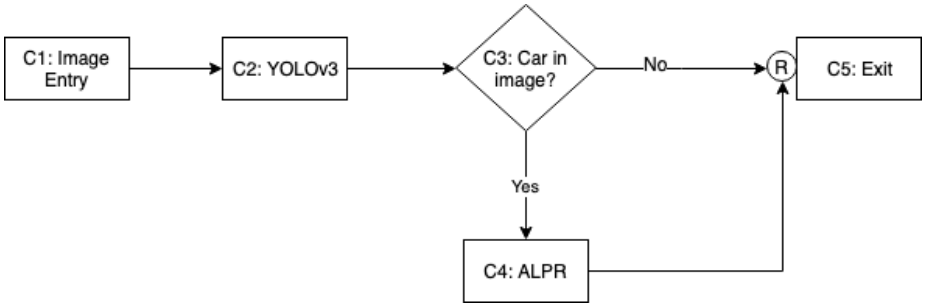

Figure 3.1: Inference pipeline of an image analysis service. It starts with C1 where an image is given as input. C2 runs the image through YOLOv3 [15] to detect the objects in it. C3 is a conditional where it checks if there is a car in the image. If true, the image goes through ALPR [14] to extract the nameplate details. The pipeline ends at C5 by returning the detected objects and nameplates in the image. With multiple models and branching, the pipeline represents real-world services e.g. Facebook image moderation, Google image labeling, etc.

As the project proposes a distributed architecture for artificial intelligence applications, our test AI application must be a sufficient representative of most if

not all AI applications for a fair investigation. Naturally, a fairly representative application is one with a pipeline composed of different kinds of tasks with a mix of mutually dependent and independent ones. Ergo, choosing a single AI application as a testing ground for our solution is an important task that requires studying popular AI techniques and implementations in the community.

3.2 Develop basic application

For any artificial intelligence application, first of all, machine learning models need to be trained and an inference pipeline needs to be developed. This requires studying current techniques for the application of our choice and using that knowledge to build and train good enough models. At this stage, accuracy is not important so we do not need to fine-tune the models. Once the model training is done, it needs to be ready for inference. Therefore, we need to ensure that all the steps required to accomplish inference on unseen inputs have been implemented at a satisfactory level.

3.3 Run on a centralized system

After having chosen a test AI application and trained basic models for it, we need to ensure we can successfully run inference on a single machine. This step is important for two reasons. Firstly, this gives us a baseline performance we can compare our distributed implementations with. Second, we will have a working implementation of our application and we can refer to it while converting our application to work with a distributed implementation.

3.4 Convert to a distributed system

Consequently, the next step is to convert the application from a centralized implementation to a distributed implementation. This will require modifying the source code to use distributed system techniques such as RPC (remote procedure call). To ensure consistency, we should ensure our distributed implementation running on one machine has the same performance as the centralized implementation from earlier.

3.5 Programmatic Deployment

The only way to test a distributed implementation is to deploy it on a cluster of computers and measure performance. We also need to vary the cluster specifications and test repeatedly. Moreover, we need to ensure all specifications are reproducible and give consistent performance. Doing all this manually can get complex and out of control quickly. Therefore, having a programmatic way of deploying and keeping track of distributed system deployments is crucial.

3.6 Compare architecture-agnostically

With all these different deployments, we need a reliable way of comparing each deployment performance that is completely decoupled from its intrinsic qualities. A fair and reliable way to compare is to create a web app that accepts a server URL and sends numerous requests to it. As the web app is run in the browser, it measures these metrics from the client-side and all it cares about is input and output. Thus, it does not matter for the web app if the server implementation is on a centralized or distributed architecture.

3.6.1 Test Cases

To study whether a distributed architecture can indeed improve latency, efficiency, and throughput, performance will be compared across systems of various specifications:

1. Centralized implementation (baseline)
2. 1 machine for n tasks (should be same as above)
3. Less than n machines for n tasks
4. n machines for n tasks (optimum)

3.7 Summary

To summarise, the project picks an AI application, builds it to run on a centralized architecture, converts the application to run on a distributed architecture, runs the application on distributed systems of different specifications and, lastly, compares the application across all these implementations to assess whether latency, efficiency, and throughput can be improved using a distributed architecture.

Chapter 4

Implementation

The next few sections highlight the implementation of the methodology discussed in chapter 3. It incorporates state-of-the-art open-source technologies and reproducible techniques while adapting to resource constraints.

4.1 AI Application of Choice

Figure 4.1: YOLOv3 Results. It shows two correctly identified cars and their location in image.

An image analysis service (Figure 3.1) was chosen for multiple reasons. First

of all, image services are quite common nowadays and thus representative of the production landscape of artificial intelligence. Second, there are several open-source models e.g. YOLOv3 [15] available so we can save time by not developing our own. Moreover, we can replicate real-world load by using them. Lastly, GPUs are fundamental to machine learning and our experiments need to take their usage into account. As image models heavily rely on GPUs, they are perfect candidates for being part of our inference pipeline.

4.2 Architecture (Simplified)

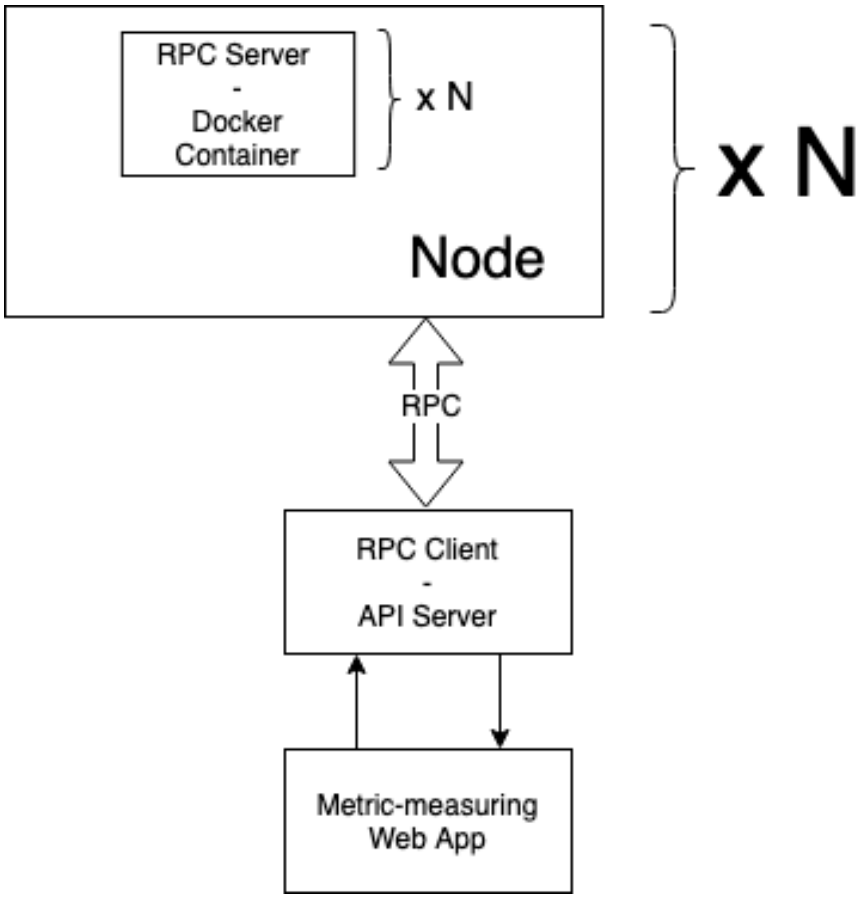

Figure 4.2: Project Architecture. A web app interacts with an API server which fulfills requests by communicating using RPC with pipeline task instances deployed across several machines.

As figure 4.2 shows, the application architecture involves remote procedure

calls to instances of pipeline tasks running across multiple machines.

4.2.1 Container for each pipeline task.

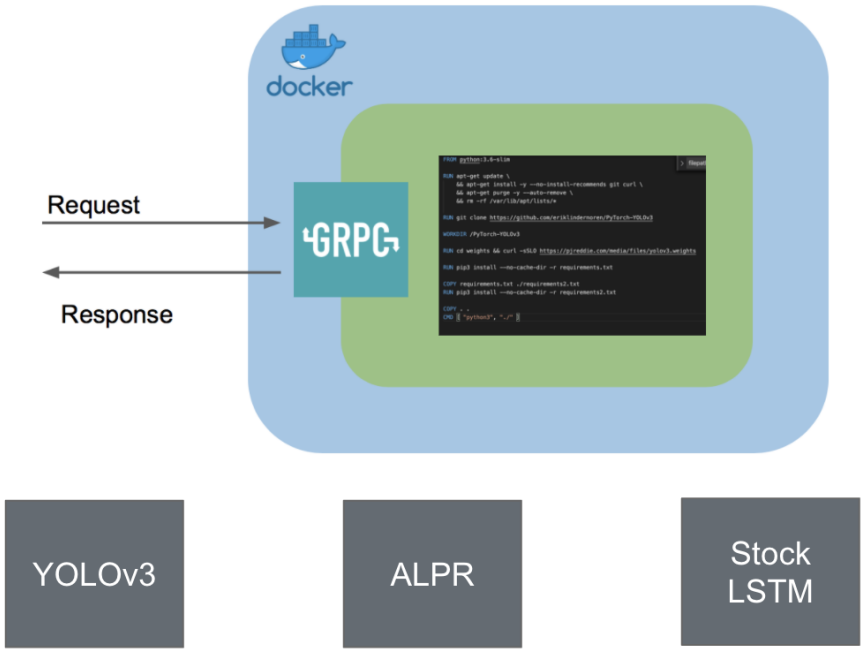

Figure 4.3: Containerization using Docker. Docker has been used to run isolated gRPC server containers for each task of the pipeline.

Docker [5] has been used to run separate *containers* for each task of our pipeline e.g. YOLOv3. This makes it easy to develop, package, and deploy each task reliably. As long as we have Docker running on our cluster nodes, we can run any task on any node. If OpenALPR requires a specific version of Linux but YOLOv3 requires something else, we don't have to worry about finding common ground. Using Docker, we can isolate each task into its runtime and start/stop/restart it at any time. Moreover, we can scale up or down the number of tasks with just a single instruction.

4.2.2 Remote Procedure Calls

For remote procedure calls, gRPC [7] is used because it is open-source, modern (uses HTTP/2), efficient and cross-platform (uses Protocol Buffers as the interface description language). To implement gRPC, every *task container* of the pipeline is a gRPC server. Our API server acts as a gRPC client when communicating with any task instance. Because gRPC is cross-platform, we are

not limited by our choice of language/framework to build our RPC servers. All we need is to define our task as a gRPC service using *Protocol Buffers* [6] (a method of serializing structured data).

4.2.3 API Server

Our API server is the bridge between our client and our task instances. It implements the *REST* architecture style using *Python* and *Flask*. As it is responsible for managing our task instances, all the *scheduling* logic lies here. By changing how it fulfills the requirements of incoming requests, we can increase/decrease the performance of our distributed system.

In addition to accepting inputs for our image analysis pipeline, it also accepts inputs to the scheduling logic. This allows us to easily modify its behavior while conducting experiments.

4.2.4 Web App

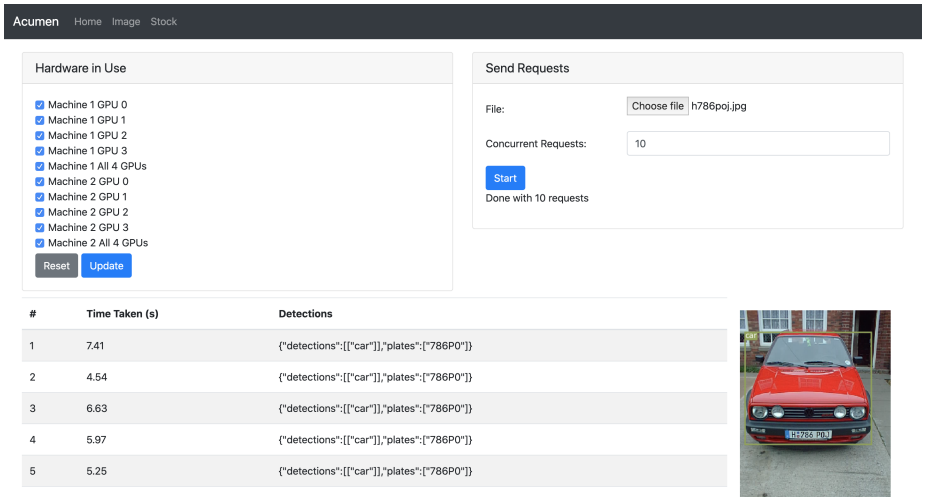

Figure 4.4: Metric-measuring Web App. The distributed system's specifications can be modified using the "Hardware In Use" panel on top left. The "Send Requests" panel on the right is used to control the number of concurrent requests. The results section shows the output and response time for each request.

The purpose of the web app is three-fold. First, we can use it to send requests and measure response time. Second, we can modify the type/number of requests we send. Third, we can set the scheduling logic of the API server. With these three functions, we can conduct all kinds of experiments and collect valuable data for our project.

The web app is a fully client-side and all the required *HTML*, *CSS* and *JavaScript* is fetched once in the beginning. This ensures our API server is not

wasting valuable resources rendering web pages. It is built using *JavaScript* and *React*. As it does not know how the API server works other than the kind of input it accepts and the output it sends back, our web app is fully architecture-agnostic.

4.3 Architecture (Actual)

As with any real-world implementation, we are constrained by the resources we have at hand. To overcome them, the architecture has been slightly modified as shown in figure 4.5.

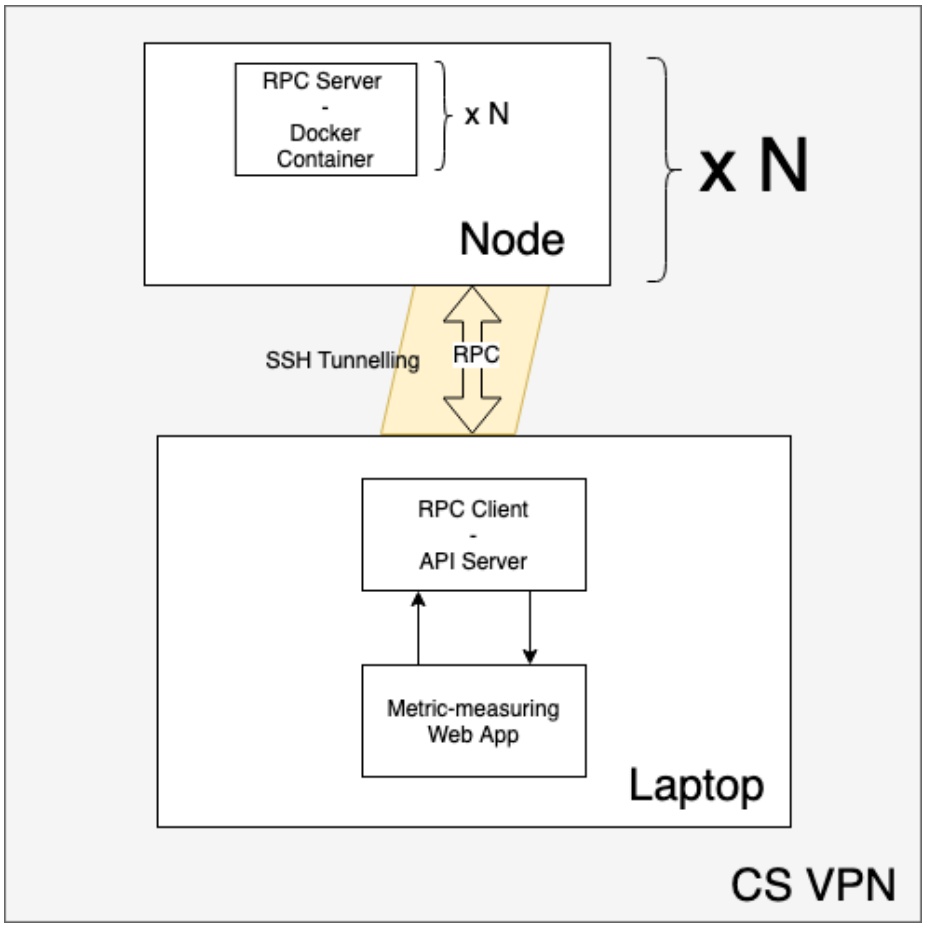

Figure 4.5: Modified architecture to overcome resource constraints. SSH tunneling has been used to port forward API server's ports to task instances. The entire architecture runs on the CS VPN.

4.3.1 Challenges

The machines available for use at the Computer Science department of the University of Hong Kong have highly powered CPUs (Intel(R) Xeon(R) Silver 4116 CPU @ 2.10GHz) and GPUs (4 x GeForce RTX 2080 Ti). However, using them involved a few challenges:

1. All ports are blocked
2. Only SSH is allowed
3. CSVPN is required

As RPC communication between our API server and the task instances running on these machines requires several ports to be open, it was a make-or-break situation. But on the other hand, it would have been costly not to use these machines so overcoming these challenges was important.

4.3.2 CS VPN

The first solution was to use the same machine to run the API server and browse the web app. If connected to CS VPN, the machine allows us to communicate with other department machines via SSH.

4.3.3 SSH Tunneling

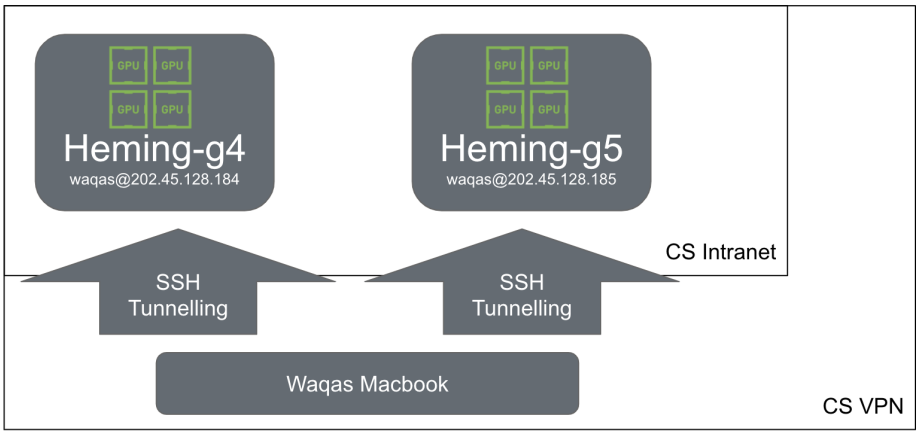

Figure 4.6: SSH Tunneling to connect to task instances from my Macbook running the API server.

Because all ports are blocked on department machines except SSH (22), SSH tunneling was used for port forwarding (Figure 4.6). What that means is that the API server acts as if the task instances are running on the same machine as itself but the port requests are forwarded to cluster machines via SSH tunneling.

4.4 Deployment & Source Code

4.5 Terraform

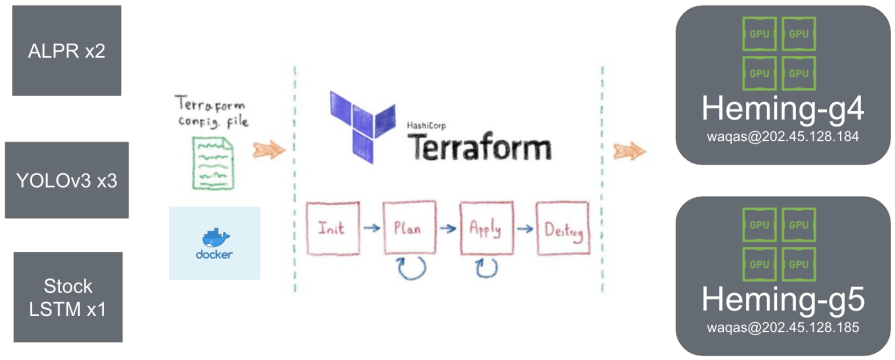

Figure 4.7: Deployment using Terraform. By writing our infrastructure as code in config files, we can easily track, manage, scale and deploy our experiments.

With complex pipelines, several machines and tens of task instances, it is highly important to track deployment. Through *Terraform* [8], we can programmatically deploy and manage our infrastructure. If we need to increase or decrease the number of YOLOv3 instances across multiple machines (while keeping track of their port numbers), we only need to modify a single config file, and Terraform handles the rest. Moreover, this brings infrastructure under version control, allows reproducibility and increases scalibility (more machines can be easily added).

4.6 Source code

To ensure the project is replicable and open to further improvement, the source code has been made available. [1] With the tools the project publishes, it hopes to contribute to further research in distributed systems at the University of Hong Kong.

Chapter 5

Results

Table 5.1: Response times for Image Analysis pipeline with a car image as input. n represents the number of concurrent requests. Each column represents the number of task instances in the distributed system. n/2 for n in the case of n = 10 means that there were 5 instances each of YOLOv3 and OpenALPR available for use. The response time is the total time it took for all requests to finish (averaged over three trials)

Requests (n) / Task instances	1 for n (baseline)	n/2 for n	n for n (optimum)
1	4.61s	4.60s	4.64s
10	46.41s	9.30s	4.59s
50	231.22s	9.29s	4.61s

Chapter 6

Conclusion

The project has three crucial parts: machine learning, tooling, and distributed architecture. As shown in chapter 4, a functional image analysis service has been developed and deployed which closely resembles machine learning applications in the real world. Moreover, reliable tooling has been successfully developed for metric measurement and deployment that can be used outside of this project. Adding to that, a distributed implementation has been developed. As mentioned in chapter 3, the hypothesis mentioned in section 1.4 (distributed architecture is better than traditional architecture) has been tested using a production-representative machine learning application and reliable metric tooling. With the data obtained from these tests (Table 5.1), the project has shown that the latency, efficiency, and throughput of AI applications can be improved using a distributed architecture.

Bibliography

[1] Waqas Ali. *Serving AI using a Distributed Architecture*. https://github.com/WaqasAliAbbasi/HKU-FYP. 2020.

[2] Ion Androutsopoulos et al. "Learning to Filter Spam E-Mail: A Comparison of a Naive Bayesian and a Memory-Based Approach". In: *CoRR* cs.CL/0009009 (2000). URL: http://arxiv.org/abs/cs.CL/0009009.

[3] Daniel Crankshaw et al. "Clipper: A Low-Latency Online Prediction Serving System". In: *Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation*. NSDI'17. Boston, MA, USA: USENIX Association, 2017, pp. 613–627. ISBN: 9781931971379.

[4] Daniel Crankshaw et al. "InferLine: ML Inference Pipeline Composition Framework". In: *CoRR* abs/1812.01776 (2018). arXiv: 1812.01776. URL: http://arxiv.org/abs/1812.01776.

[5] Docker. *Docker*. https://www.docker.com/products/container-runtime. 2020.

[6] Google. *protobuf 3.11.4*. https://github.com/protocolbuffers/protobuf. 2020.

[7] gRPC. *gRPC 1.28.1*. https://github.com/grpc/grpc. 2020.

[8] HashiCorp. *Terraform*. https://www.terraform.io/. 2020.

[9] Benjamin Hindman et al. "Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center". In: *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*. NSDI'11. Boston, MA: USENIX Association, 2011, pp. 295–308.

[10] Ram Srivatsa Kannan et al. "GrandSLAm: Guaranteeing SLAs for Jobs in Microservices Execution Frameworks". In: *Proceedings of the Fourteenth EuroSys Conference 2019*. EuroSys '19. Dresden, Germany: Association for Computing Machinery, 2019. ISBN: 9781450362818. DOI: 10.1145/3302424.3303958. URL: https://doi.org/10.1145/3302424.3303958.

[11] George Lekakos and Petros Caravelas. "A hybrid approach for movie recommendation". In: *Multimedia tools and applications* 36.1-2 (2008), pp. 55–70.

[12] John McCarthy. *What Is Artificial Intelligence?* Tech. rep. Stanford University, 2007.

[13] Thomas Mitchell. *Machine Learning (McGraw-Hill Series in Computer Science)*. McGraw-Hill Education, 1997.

[14] OpenALPR. *OpenALPR 2.3.0*. `https://github.com/openalpr/openalpr`. 2016.

[15] Joseph Redmon and Ali Farhadi. "YOLOv3: An Incremental Improvement". In: (Apr. 2018).

[16] David A. Teich and Paul R. Teich. *PLASTER: A Framework for Deep Learning Performance*. Tech. rep. TIRIAS Research, 2018.