



SAPIENZA
UNIVERSITÀ DI ROMA

From CloudOps to EdgeOps: Design, Develop and Pilot CI/CD pipelines for ML on IoT Edge with AWS

Faculty of Information Engineering, Informatics and Statistics
Master Degree Program in Engineering in Computer Science

Candidate

Andrea Gaetano Ferrara
ID number 1887505

Thesis Advisor

Prof. Ioannis Chatzigiannakis

Academic Year 2020/2021

Thesis defended on 14 January 2022
in front of a Board of Examiners composed by:

Prof. Francesca Cuomo (chairman)

Prof. Ioannis Chatzigiannakis

Prof. Antonio Cianfrani

Prof. Fabrizio D'Amore

Prof. Umberto Nanni

Prof. Christian Napoli

Prof. Roberto Navigli

**From CloudOps to EdgeOps: Design, Develop and Pilot CI/CD pipelines for
ML on IoT Edge with AWS**

Master's thesis. Sapienza – University of Rome

© 2021 Andrea Gaetano Ferrara. All rights reserved

Version: January 6, 2022

Author's email: ferrara.1887505@studenti.uniroma1.it

*Dedicata a
i miei Genitori*

Abstract

Technological innovations are what make the world go round, thanks to them we experience every day new possibilities that we thought impossible just yesterday. In these, a major role is represented by Machine Learning and the Internet of Things world that for this reason are the backbone of our paper. Thanks to the help of these two technologies we can achieve new services that can be of great help to individuals or to society. Unfortunately, they don't always go well together since they have some rules that need to be followed to be functional and they are often conflicting. In today's society, the inclusion of every individual is becoming more and more important and, while the rights and necessities of many minorities and groups are now being considered more, there are still some obstacles that need to be overcome for a future where everyone is equal. Our take in this subject was this project where we wanted to use a Machine Learning Object Detection Model in order to recognize gestures from the International Sign Language shown to a webcam-equipped device to simplify communication between a speech-impaired person and anyone not knowledgeable about Sign Language. One of the main goals of this paper was to show how to effectively implement an ML model in an Edge-Based Environment so as to exploit the benefits of having computation closer to the device and transfer via internet connection only ingested data. To reach our objectives we created a cloud system so we could later transition to the edge system while highlighting all the differences in implementation and execution. The results show how different implementations may answer different needs regarding latency, costs, or ease of deployment. Moreover, the creation of CI/CD pipelines will allow keeping the project relevant in the years to come by providing tools and technologies for easy scalability of the system while also creating a place where the source code can be stored and updated regularly to fix possible bugs or for adding interesting features in the future. During the whole discussion, several services belonging to the AWS platform, which is one of the most used offering services for cloud computing, storage, training, and represents the go-to for most of the tasks they were created for.

Acknowledgments

I ringraziamenti per questa tesi ho deciso di scriverli in italiano, in quanto credo che ci siano cose che posso esprimere solo nella mia lingua madre.

Per prima cosa voglio ringraziare il professore relatore di questa tesi il Prof. Ioannis Chatzigiannakis che si è sempre mostrato estremamente disponibile durante tutta la creazione di questo elaborato. I suoi consigli e considerazioni hanno influito molto nella creazione di una tesi di cui potessi andare fiero.

Voglio inoltre ringraziare il mio collega Alessandro Migliore con la quale ho avuto il piacere di sviluppare questo progetto che segna la fine di questo corso Magistrale. Quando si è con un amico anche le ore di lavoro passano in un attimo tra risate e scherzi.

Un sentito ringraziamento va ai colleghi del corso di Engineering in Computer Science, in particolare a Giovanni, Lorella, Daniel e Michele che nonostante la situazione non proprio felice e la nostra prematura separazione hanno reso questi due anni davvero indimenticabili.

A tutti i miei amici con i quali ho condiviso avventure, gioie ma anche ansie e preoccupazioni e che in ogni momento mi hanno aiutato ad andare avanti.

Ringrazio il gruppo che ancora chiamo “i compagni delle superiori” ma che ormai potrei chiamare compagni di vita dato che mi sopportano da più di dieci anni e con i quali condivido dei preziosissimi ricordi.

Ai miei fratelli “Capybara” che considero ormai famiglia. So di poter sempre contare su di loro e loro potranno sempre contare su di me. Ed anche se il tempo e le circostanze potranno in futuro allontanarci fisicamente, avremo sempre una seconda casa nel canale Discord dove poter stare insieme a giocare e divertirci.

Agli “HydeG”, anche se siamo separati da centinaia di chilometri di distanza, la nostra è un’amicizia autentica e duratura. Grazie a voi mi sento a casa in ogni parte d’Italia.

Alla mia famiglia, che mi ha sempre sostenuto e che ha fatto il tifo per me in ogni fase della mia vita, universitaria e non.

A mio fratello Paolo, il mio fan #1 e la prima persona a criticarmi quando sbaglio. Sei la prima persona a cui penso quando ricevo una bella notizia, ma anche la spalla su cui sorreggermi quando mi sento giù. Ti voglio tanto bene.

Ai miei genitori ai quali dedico non solo questa tesi, ma anche qualsiasi altro traguardo riuscirò a raggiungere in futuro. Il loro appoggio ed il loro affetto mi hanno reso la persona che sono oggi. Spero che siano tanto fieri di me quanto io lo sono del fatto che sono i miei genitori.

In fine, vorrei tanto ringraziare mia nonna Alda e mia nonna Fina. So che avrebbero fatto di tutto per essere qui oggi, ma so anche che non mancheranno mai nei miei pensieri.

Contents

1	Introduction	1
1.1	Overall Problem	1
1.2	The approach and goals of the thesis	1
1.3	Structure of the thesis	2
2	Model Deployment at the Edge	5
2.1	Architectures	6
2.1.1	Cloud-Centric	6
2.1.2	Edge-Based	7
2.2	Cloud-based ML Technologies	7
2.2.1	Machine Learning	7
2.2.2	Amazon SageMaker	8
2.2.3	Amazon SageMaker Ground Truth	9
2.2.4	Apache MXNet in AWS	9
2.2.5	OpenCV	10
2.3	Cloud-based Storage Services	10
2.3.1	Amazon S3	10
2.3.2	Amazon DynamoDB	11
2.4	Cloud-based Access Control and Accounting	11
2.4.1	AWS Identity and Access Management (IAM)	11
2.4.2	Amazon CloudWatch	12
2.5	Compute	12
2.5.1	AWS Lambda	12
2.5.2	Amazon Elastic Container Registry(ECR)	12
2.5.3	Docker	13
2.6	Edge technologies	13
2.6.1	Internet of Tings	13
2.6.2	AWS IoT Greengrass	15
2.7	CI/CD	15
2.7.1	GitHub Repository	15
2.7.2	AWS CodeCommit	16
2.7.3	AWS CodePipeline	16
2.7.4	AWS CloudFormation	16

3	Use Case	17
3.1	Background	17
3.2	International Sign Language	17
3.3	Thesis Objectives	18
3.4	Model Training	19
3.4.1	Data Preparation	19
3.4.2	Labeling with AWS GroundTruth	20
3.4.3	Training with AWS SageMaker	22
3.5	Creating Deployable Model	24
4	Cloud-Based Services	27
4.1	Introduction	27
4.2	Images collection and upload	28
4.3	The Lambda Function	29
4.3.1	Lambda with Container Image	29
4.3.2	Lambda Configuration and Trigger	35
4.4	Conclusions and Tests	36
5	Edge-Based Services	39
5.1	Introduction	39
5.2	AWS IoT Greengrass Core software and Greengo Deployment	39
5.3	Creating your inference pipeline in AWS IoT Greengrass Core	41
5.4	Lambda Functions	42
5.4.1	VideoIngest	42
5.4.2	BlogInfer	43
5.4.3	DynamoItem	45
5.4.4	Configure Lambda functions	46
5.5	Install machine learning dependencies on the device	47
5.6	Local and Machine Learning Resources	48
5.6.1	Local Resources	48
5.6.2	Machine Learning Resources	50
5.7	Subscriptions	52
5.8	MQTT Test Client	52
6	Continuous Integration and Delivery	55
6.1	Introduction	55
6.2	AWS CodeCommit	56
6.3	AWS CodePipeline	57
6.4	AWS CloudFormation and Stack	60
6.4.1	Introduction	60
6.4.2	Design and implementation of the template in YAML language	61
7	Conclusions and Future Works	71
7.1	Future works	73

Chapter 1

Introduction

1.1 Overall Problem

It is safe to say that today any tech enthusiast or even any technology user has in some way benefitted from machine learning. Facial recognition technology allows social media platforms to help users tag and share photos of friends. Optical character recognition (OCR) technology converts images of text into movable types. Recommendation engines, powered by machine learning, suggest what movies or television shows to watch next based on user preferences. It is clear then that Machine Learning is a reality that influenced deeply our approach to technology and our life in general. Thanks to ML, we can make computers train on data inputs and use statistical analysis in order to output values that fall within a specific range. Unfortunately, one of the limitations of Machine Learning is that to have a reliable model that performs its task correctly, you need to train it properly. This requires a great amount of data and good computation power. This kind of constraint, up until now, was satisfied easily only in a Cloud Environment where you could use all the available services hosted by some big data center anywhere in the world; on the contrary, if you wanted to exploit the great advantages of using a device close to the "edge", things were not that simple. In the meantime, we can easily see that the world of "Smart Things" is overwhelmingly making its way into our lives. Internet of Things (IoT) is the networking of physical objects that contain electronics embedded within their architecture to communicate and sense interactions amongst each other or concerning the external environment. It is clear that finding a way to make these two technologies "work together" will bring life to new systems that will be of great use for today's society.

1.2 The approach and goals of the thesis

Thankfully, nowadays the computational power of smart devices increased greatly and this allows us to shift configurations that were only possible in a cloud-centric way, in an edge-based system. With this said, the thesis aims to design, develop and pilot a machine learning model for AI-powered Smart Cameras connected using the IoT service AWS bringing what once was possible almost exclusively in a cloud environment closer to the edge to take advantage of its benefits.

Then, since a radical change of implementation like this brings its consequences in several fields like integration and delivery, which will be looked into more in-depth later, we will work to make our project as easily implementable and scalable as possible to allow for new devices to be added to the system. In fact, in the cloud environment, it is easy to maintain the code and scale the implementation since all the logic is being kept in one place and uploaded to hosts that can be accessed virtually infinite times. On the other hand in an edge environment, the topic gets more complicated since any new device must be provided with all the logic it needs and this might be time-consuming. Thankfully, using services like CloudFormation and CodeCommit we are able to create a repository where all the functions are stored and we can create the whole group with a single click assuring a CI/CD approach to the whole system. To be more precise, in our case, we want to develop and create through these new technologies a device that is able to help and involve in society all people with speech impairment, making it possible for them to collaborate or socialize quickly with anyone. To do so, the first and most important step will be the construction of the model, trained using a dataset of images that allows to accurately recognize the sign language gesture. The thesis will show how to create a properly trained model since one of the goals of the paper is to have a general approach so that anyone could make their own to bring a new project to life. For this model to be used correctly, we need to make our infrastructure smart. If we think in fact in the world of work or even in your own home it is understandable that implementing this model on different devices would be very complex. For this reason, as we will see later through the use of Amazon Web Service, you can design, implement an infrastructure from CloudOps to EdgeOps. This will allow us to recognize and study the strengths or weaknesses in both cases, providing a complete view of what is the right choice to make, based on your needs. Our aim, for this reason, was to make the entire project usable and general, so that it would be possible to implement our infrastructure for similar purposes, both for those that use object detection as the main focus and those in a broader spectrum. In the next chapters, we will illustrate in detail the analysis, the design, and the implementation carried out at the code level to allow the modification of these parameters.

1.3 Structure of the thesis

This thesis is structured in several chapters in which various aspects of the project will be analyzed not only to show how to effectively implement the system to use it properly, but we will also deepen the topics and technologies covered in this paper since we find them really interesting but also crucial in the technological world as it is right now. To allow for this thesis to be available to the widest possible audience, we will explain all the technicalities we run into to have a paper as clear as possible. To be specific, in Chapter 2 we will give an introduction to the two main arguments around which this thesis revolves and all the tools and technologies used in order to implement them as smoothly as possible. In Chapter 3 we will present our Use Case for this project which, while being only one of the possible applications, we keep close at heart. We discuss of why of this choice and present the steps necessary to reach what we have reached while developing this project. Chapter 4 goes more

in-depth in the Cloud-Centric System and shows how to take a deployable machine learning model and use it to make predictions and store them in a database. On the contrary, Chapter 5 takes what we created in the previous chapter and adapts it to make it deployable in an Edge Environment and all the additional steps and requirements needed to have a fully working smart device deploying an ML model. Finally, in chapter 6 we address what is one of the main goals of the thesis, creating a Continuous Integration and Continuous Delivery pipeline to ease an otherwise obnoxious and time-wasting implementation. We will show all the tools and how to use them to create a system that is regularly kept up to date. In the conclusions of this paper, we will discuss the results reached, some evaluation parameters for both implementations, and what we can do in the future to improve our project to better suit the need of the customers and society.

Chapter 2

Model Deployment at the Edge

In the following chapters we will analyze in detail not only the implementation, the code, and the development of the application, but above all the technologies used, the methods of implementation of them, and the various ways in which they can be used according to the service we want to offer. All this allows us to better learn the potential of the technology that we are going to use. Specifically, we will start with the creation of a model, to do this we will get from our device with an integrated camera several frames that will be added to our dataset. Then we will focus on the use of two infrastructures:

- Cloud-centric
- Edge-based

with their relative advantages and disadvantages for each of them. Once development is complete, it could be very time-consuming to create a smart environment on many devices. We will analyze how through the use of a CI/CD pipeline we will automate the whole process and what benefits we can derive from it; automated pipelines remove manual errors, provide standardized feedback loops to developers, and enable fast product iterations. Finally, we will carry out a cost assessment analysis for the creation of the entire infrastructure and time analysis for both cases. To make our project more accessible to anyone that wants to implement it, we have used a lot of different services that also helped us a great deal during the Design and Implementation phases. In particular, we used several tools from the Amazon Web Services console, AWS is one if not the most used platform offering services for cloud computing, storage, training, and much more. Obviously, there are more realities out there and they all work very well with competitive costs, but for clarity and simplicity, we decided to stick with Amazon's services for all our needs. Amazon Web Services, Inc. (AWS) is a subsidiary of Amazon providing on-demand cloud computing platforms and APIs to individuals, companies, and governments, on a metered pay-as-you-go basis. These cloud computing web services provide a variety of basic abstract technical infrastructure and distributed computing building blocks and tools. AWS's virtual computers emulate most of the attributes of a real computer, including hardware central processing units (CPUs) and graphics processing units (GPUs) for processing; local/RAM; hard-disk/SSD storage; a choice of operating systems; networking; and pre-loaded application software such as

web servers, databases, and customer relationship management (CRM). The AWS technology is implemented at server farms throughout the world and maintained by the Amazon subsidiary. Fees are based on a combination of usage (known as a "Pay-as-you-go" model), hardware, operating system, software, or networking features chosen by the subscriber required availability, redundancy, security, and service options. These are paid services, but AWS provides a free tier you can use to experiment and even implement without the fear of "wasting" money. More on the costs of these services will be studied later in this thesis showing some use cases. Following here we will give a brief description of all the services used for the sake of fluidity later in the paper.

2.1 Architectures

2.1.1 Cloud-Centric

Cloud computing is the on-demand availability of computer system resources, especially data storage (cloud storage) and computing power, without direct active management by the user. Large clouds often have functions distributed over multiple locations, each location being a data center. Cloud computing relies on sharing of resources to achieve coherence and economies of scale, typically using a "pay-as-you-go" model which can help in reducing capital expenses but may also lead to unexpected operating expenses for unaware users. This obviously means that much more bandwidth will be used and that some latency of transfer on the prediction might be present.

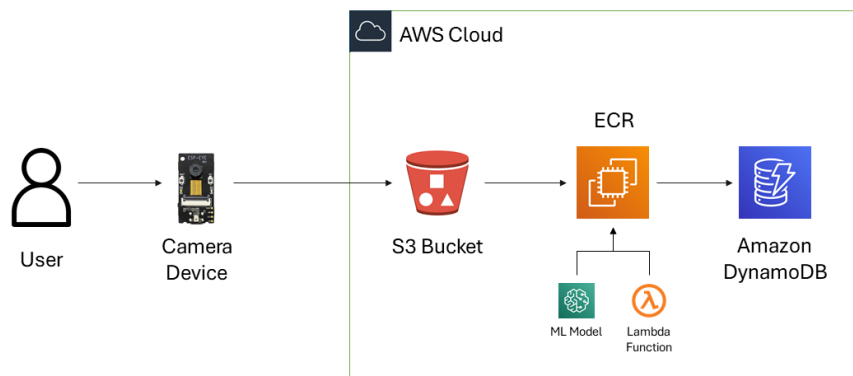


Figure 2.1. Cloud-Centric Architecture

2.1.2 Edge-Based

Edge computing is a distributed computing paradigm that brings computation and data storage closer to the sources of data. This is expected to improve response times and save bandwidth. The idea behind our interest in designing an implementation in an Edge System comes from the fact that to perform machine learning tasks the data exchanged is typically relatively big in size. This could generate bandwidth and delay problems when the system will perform real-time execution and invocation of the ML model. The solution to this could be loading the model on a device that will acquire all the information needed by the model, manipulate them locally as to send only the data necessary for the execution of the function which would obviously be smaller.

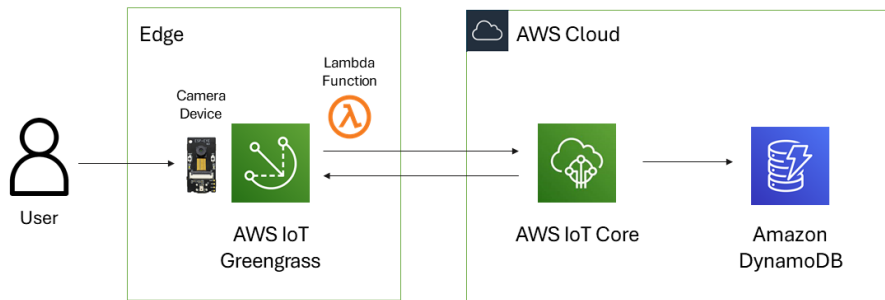


Figure 2.2. Edge-Centric Architecture

2.2 Cloud-based ML Technologies

2.2.1 Machine Learning

The practice of assisting software in performing a task without explicit programming or rules is known as machine learning. A programmer specifies rules for the computer to follow in traditional computer programming. However, ML necessitates a different mindset. Real-world ML is more concerned with data analysis than with coding. Programmers provide a set of examples, and the computer uses the data to learn patterns. Machine learning can be thought of as "data programming." Machine learning is a field that is constantly evolving. As a result, there are a few things to think about when working with machine learning methodologies or analyzing the impact of machine learning processes. Tasks are generally classified into broad categories in machine learning. These classifications are based on how learning is received and how the system is given feedback on the learning. Supervised learning

and unsupervised learning are two of the most widely used machine learning methods. Unsupervised learning, which provides the algorithm with no labeled data in order to allow it to find structure within its input data, and supervised learning, which trains algorithms based on example input and output data that is labeled by humans, are two of the most widely used machine learning methods. Deep learning tries to mimic how the human brain converts light and sound into vision and hearing. A deep learning architecture is inspired by biological neural networks and consists of multiple layers in an artificial neural network made up of hardware and GPUs. Deep learning extracts or transforms data features using a cascade of processing unit layers. One layer's output is used as the input for the next layer. Deep learning, the most widely used and developed machine learning algorithm, absorbs the most data and has been shown to outperform humans in various cognitive tasks. Deep learning has become the approach with the most potential in the artificial intelligence area as a result of these characteristics. Deep learning algorithms have made substantial advances in computer vision and speech recognition. A neural network is a pattern-recognition model that can be trained. It has layers that include input and output layers, as well as at least one concealed layer. Each layer's neurons learn increasingly abstract data representations. For example, we can see neurons identifying lines, shapes, and textures in this visual representation. These representations (or learned characteristics) allow the data to be classified. In our case, machine learning will allow us to create a model that recognizes the Sign Language gesture shown to our device's camera, which brings us to our use of IoT. Combining services from AWS in the Machine Learning (ML) and Internet of Things (IoT) space, training a custom computer vision model, and running it at the edge has become easier than ever. In computer vision, image classification tells you what type of objects are in the image. Object detection, in addition to defining objects, also tells you where the objects are by producing bounding boxes that mark the location of each object being detected.

2.2.2 Amazon SageMaker

Amazon SageMaker is a fully managed service that provides every developer and data scientist with the ability to build, train, and deploy machine learning (ML) models quickly. SageMaker removes the heavy lifting from each step of the machine learning process to make it easier to develop high-quality models. Amazon SageMaker helps data scientists and developers to prepare data and build, train, and deploy machine learning models quickly by bringing together purpose-built capabilities. These capabilities allow you to build highly accurate models that improve over time without all the undifferentiated heavy lifting of managing ML environments and infrastructure. As for any model training, you need lots of data. SageMaker lets you connect and load your data from sources such as Amazon S3 so that you can use this data to train your model. Models learn complex and subtle patterns to let you map inputs to predicted outputs. You can retrieve an entire data set for training. Once your model is deployed, you can retrieve individual features to make low latency predictions, such as predicting in real-time. Next, great models can be used in different situations if they are trained on a balanced set of features and data. You can use SageMaker Clarify to identify potential bias in your training data. This will help you ensure your model is trained across a range of genres,

leading to more accurate predictions. You can also use SageMaker Clarify to inspect individual predictions to understand how each feature plays a role in the prediction. This allows you to check that the model isn't overly reliant on features that are underrepresented in the data. One of the great things about machine learning is that models can improve over time, not just based on new data as it becomes available, but also by incorporating the learnings from tools like SageMaker Clarify and SageMaker Debugger to systematically identify sources of error or slowness and remove them from your model. With this approach, you can condense hundreds of thousands of hours of real-world experience into just a few retraining iterations, so your models can improve quickly. And since you want to continually improve the model by rebuilding it regularly, you can take advantage of Amazon SageMaker Pipelines, which provides continuous integration. This decreases the time between model improvements and delivers better models more quickly. Amazon SageMaker provides tools which every developer is familiar with, visual editors, debuggers, profilers, and CI/CD, all wrapped into the Amazon SageMaker Studio integrated development environment for machine learning.

2.2.3 Amazon SageMaker Ground Truth

In 2018, Amazon Sagemaker Ground Truth was launched to fully manage data labeling services for generating high-quality ground truth datasets to be trained into machine learning models. Ground Truth can integrate Amazon Mechanical Turk (the crowdsourcing platform) or internal data labeling team or external 3rd party vendors to get the labeling job done. Workflows can be customized or made use of built-in. This labeled dataset output from Ground Truth can be used to train their models or as a training dataset for an Amazon SageMaker model. Sagemaker Ground truth offers a wide range of services in image, audio, video, and text having features such as removal of distortion in images, automatic 3D cuboid snapping, and auto-segment tools to reduce the labeling time. Auto Labelling is possible using semi-supervised learning, where it learns to label the data. Varied pricing for each labeled object (image/video frame, audio recording, a section of the text, etc.) whether it's labeled automatically by Ground Truth or by a human labeler. If you use a vendor or Mechanical Turk to provide labels, you pay an additional cost per labeled object. If you use your employees for labeling, there is no additional cost per labeled object. The workforce type can be public or private mode.

2.2.4 Apache MXNet in AWS

Model Server for Apache MXNet (MMS) is an open-source component that is designed to simplify the task of deploying deep learning models for inference at scale. Deploying models for inference is not a trivial task. It requires collecting the various model artifacts, setting up a serving stack, initializing and configuring the deep learning framework, exposing an endpoint, emitting real-time metrics, and running custom pre-process and post-process code, to mention just a few of the engineering tasks. While each task might not be overly complex, the overall effort involved in deploying models is significant enough to make the deployment process slow and cumbersome. With MMS, AWS contributes an open-source engineering toolset for

Apache MXNet that drastically simplifies the process of deploying deep learning models. The key features that we can achieve through the use of MXNet are different, as tooling to package and export all model artifacts into a single “model archive” file that encapsulates everything required for serving an MXNet model or the ability to customize every step in the inference execution pipeline, from model initialization, through pre-processing and inference, up to post-processing the model’s output.

2.2.5 OpenCV

OpenCV (Open Source Computer Vision Library) is a library of programming functions mainly aimed at real-time computer vision. Originally developed by Intel, it was later supported by Willow Garage then Itseez. The library is cross-platform and free for use under the open-source Apache 2 License. Starting with 2011, OpenCV features GPU acceleration for real-time operations. OpenCV is a great tool for image processing and performing computer vision tasks. It is an open-source library that can be used to perform tasks like face detection, objection tracking, landmark detection, and much more. The library is equipped with hundreds of useful functions and algorithms, which are all freely available to us. Some of these functions are really common and are used in almost every computer vision task. Whereas many of the functions are still unexplored and haven’t received much attention yet. OpenCV’s application areas include:

- 2D and 3D feature toolkits
- Facial recognition system
- Gesture recognition
- Human–computer interaction (HCI)
- Mobile robotics
- Object detection
- Segmentation and recognition
- Motion tracking
- Augmented reality

2.3 Cloud-based Storage Services

2.3.1 Amazon S3

Amazon S3 (Simple Storage Service) provides object storage, which is built for storing and recovering any amount of information or data from anywhere over the internet. It provides this storage through a web services interface. While designed for developers for easier web-scale computing, it provides 99.99999999 percent durability and 99.99 percent availability of objects. It can also store computer files up to 5 terabytes in size.

It is probably the most commonly used, go-to storage service for AWS users given the features like extremely high availability, security, and simple connection to other AWS Services. AWS S3 can be used by people with all kinds of use cases like mobile/web applications, big data, machine learning, and many more. An object consists of data, key (assigned name), and metadata. A bucket is used to store objects. When data is added to a bucket, Amazon S3 creates a unique version ID and allocates it to the object.

2.3.2 Amazon DynamoDB

Amazon DynamoDB is a fully managed, serverless, key-value NoSQL database designed to run high-performance applications at any scale. DynamoDB offers built-in security, continuous backups, automated multi-region replication, in-memory caching, and data export tools. With DynamoDB, there are no servers to provision, patch, or manage, and no software to install, maintain or operate. DynamoDB automatically scales tables to adjust for capacity and maintains performance with zero administration. Availability and fault tolerance are built-in, eliminating the need to architect your applications for these capabilities. It provides capacity modes for each table: on-demand and provisioned. For workloads that are less predictable for which you are unsure that you will have high utilization, on-demand capacity mode takes care of managing capacity for you, and you only pay for what you consume. Tables using provisioned capacity mode require you to set read and write capacity. Provisioned capacity mode is more cost-effective when you're confident you'll have decent utilization of the provisioned capacity you specify. For tables using the on-demand capacity mode, DynamoDB instantly accommodates your workloads as they ramp up or down to any previously reached traffic level. If a workload's traffic level hits a new peak, DynamoDB adapts rapidly to accommodate the workload. You can use on-demand capacity mode for both new and existing tables, and you can continue using the existing DynamoDB APIs without changing the code.

2.4 Cloud-based Access Control and Accounting

2.4.1 AWS Identity and Access Management (IAM)

Amazon Web Services (AWS) cloud provides a secure virtual platform where users can deploy their applications. Compared to an on-premises environment, AWS security provides a high level of data protection at a lower cost to its users. There are many types of security services, but Identity and Access Management (IAM) is one the most widely used. AWS IAM enables you to securely control access to AWS services and resources for your users. Using IAM, you can create and manage AWS users and groups, and use permissions to allow and deny their access to AWS resources.

2.4.2 Amazon CloudWatch

Amazon CloudWatch is a component of Amazon Web Services that provides monitoring for AWS resources and the customer applications running on the Amazon infrastructure.

CloudWatch enables real-time monitoring of AWS resources such as Amazon Elastic Compute Cloud (EC2) instances, Amazon Elastic Block Store (EBS) volumes, Elastic Load Balancing, and Amazon DynamoDB tables. The application automatically collects and provides metrics for CPU utilization, latency, and request count. Users can also stipulate additional metrics to be monitored, such as memory usage, transaction volumes, or error rates. Users can access CloudWatch functions through an application programming interface (API), command-line tools, one of the AWS software development kits, or the AWS Management Console. The CloudWatch interface provides current statistics that users can view in graph format. Users can set notification alarms to be sent when something being monitored surpasses a specified threshold. The app can also detect and shut down unused or underused EC2 instances.

2.5 Compute

2.5.1 AWS Lambda

AWS Lambda is a serverless compute service that lets you run your code without worrying about provisioning or managing any server. You can run your application or back-end service using AWS Lambda with zero administration. Just upload your code on Lambda, and it will run your code, even scale the infrastructure with high availability. The code which you run on AWS Lambda is called a lambda function. Currently, it supports many languages but what we will use will be Python. AWS Lambda easily scales the infrastructure without any additional configuration. It reduces the operational work involved and it offers multiple options like AWS S3, CloudWatch, DynamoDB, API Gateway, Kinesis, CodeCommit, and many more to trigger an event. An important feature is you don't need to invest upfront. You pay only for the memory used by the lambda function and minimal cost on the number of requests hence cost-efficient and also AWS Lambda is secure, it uses AWS IAM to define all the roles and security policies.

2.5.2 Amazon Elastic Container Registry(ECR)

Amazon ECR is a fully managed container registry offering high-performance hosting, so you can reliably deploy application images and artifacts anywhere. Amazon ECR stores your container images and artifacts in Amazon S3. This means that your data is available when needed and protected against failures, errors, and threats. Amazon ECR can also automatically replicate your data to multiple AWS Regions for your high availability applications. Amazon ECR stores both the containers you create and any container software you buy through AWS Marketplace. AWS Marketplace for Containers offers verified container software for high-performance computing, security, and developer tools, as well as SaaS products that manage, analyze, and protect container applications. In our case, ECR is used to contain the function and

model to implement the Cloud Environment System since AWS Lambda does not allow files of size bigger than 250MB while the libraries needed by the model are way heavier than that.

2.5.3 Docker

Docker is an open platform for developing, shipping, and running applications. Docker enables you to separate your applications from your infrastructure so you can deliver software quickly. With Docker, you can manage your infrastructure in the same ways you manage your applications. By taking advantage of Docker's methodologies for shipping, testing, and deploying code quickly, you can significantly reduce the delay between writing code and running it in production. Docker simplifies and accelerates your workflow while giving developers the freedom to innovate with their choice of tools, application stacks, and deployment environments for each project. Docker is the de facto standard to build and share containerized apps - from desktops to the cloud. In the scope of our project, Docker is crucial since we need to create a container image of our function in order for it to be deployed in the AWS Lambda environment. An image is a read-only template with instructions for creating a Docker container. Often, an image is based on another image, with some additional customization. You might create your own images or you might only use those created by others and published in a registry. To build your own image, you create a Dockerfile with a simple syntax for defining the steps needed to create the image and run it. Each instruction in a Dockerfile creates a layer in the image. When you change the Dockerfile and rebuild the image, only those layers which have changed are rebuilt. This is part of what makes images so lightweight, small, and fast when compared to other virtualization technologies.

2.6 Edge technologies

2.6.1 Internet of Things

The Internet of Things (IoT) describes the network of physical objects—"things"—that are embedded with sensors, software, and other technologies for the purpose of connecting and exchanging data with other devices and systems over the internet. These devices range from ordinary household objects to sophisticated industrial tools; kitchen appliances, thermostats, weather stations but also cars and the greatly discussed industry 4.0. Over the past few years, IoT has become one of the most important technologies of the 21st century. Now that we can connect everyday objects to the internet via embedded devices, seamless communication is possible between people, processes, and things. By means of low-cost computing, the cloud, big data, analytics, and mobile technologies, physical things can share and collect data with minimal human intervention. In this hyperconnected world, digital systems can record, monitor, and adjust each interaction between connected things. While the idea of IoT has been in existence for a long time, a collection of recent advances in a number of different technologies has made it practical.

- **Access to low-cost, low-power sensor technology.** Affordable and reliable sensors are making IoT technology possible for more manufacturers.

- **Connectivity.** A host of network protocols for the internet has made it easy to connect sensors to the cloud and other “things” for efficient data transfer.
- **Cloud computing platforms.** The increase in the availability of cloud platforms enables both businesses and consumers to access the infrastructure they need to scale up without actually having to manage it all.
- **Machine learning and analytics.** With advances in machine learning and analytics, along with access to varied and vast amounts of data stored in the cloud, businesses can gather insights faster and more easily. The emergence of these allied technologies continues to push the boundaries of IoT and the data produced by IoT also feeds these technologies.
- **Conversational artificial intelligence (AI).** Advances in neural networks have brought natural-language processing (NLP) to IoT devices (such as digital personal assistants Alexa, Cortana, and Siri) and made them appealing, affordable, and viable for home use.

When something is connected to the internet, that means that it can send information or receive information, or both. This ability to send and/or receive information makes things “smart.” To be smart, a thing doesn’t need to have super storage or a supercomputer inside of it - it just needs access to it. In the Internet of Things, all the things that are being connected to the internet can be put into three categories: 1. Collecting and Sending Information

Imagine a smart farm that uses sensors to collect information useful to the farmer. Sensors could be temperature sensors, motion sensors, moisture sensors, air quality sensors, or many more. These sensors, along with a connection, allow us to automatically collect information from the environment which, in turn, allows us to make more intelligent decisions. This could give relevant information to who it may concern so that he may act accordingly to the information received. 2. Receiving and Acting on Information

It’s not a surprise that we can send commands to devices to trigger their functions; we can send a signal to a printer to receive our pdf, or with the help of a remote controller we can change the channel without getting up from our couch. So what’s so incredible about this? The real power of the Internet of Things arises when things can do both of the above. Things that collect information and send it, but also receive information and act on it. The real power of the Internet of Things arises when things can do both of the above. Things that collect information and send it, but also receive information and act on it. 3. Doing Both: The Goal of an IoT System Let’s quickly go back to the farming example. The sensors can collect information about the soil moisture to tell the farmer how much to water the crops, but you don’t need the farmer. Instead, the irrigation system can automatically turn on as needed, based on how much moisture is in the soil. Or maybe if the irrigation system receives information about the weather from its internet connection, it can also know when it’s going to rain and decide not to water the crops today because they’ll be watered by the rain anyways. What if all this information about the soil moisture, how much the irrigation system is watering the crops, and how well the crops grow can be collected and sent to supercomputers that run amazing algorithms

that can make sense of all this information? The applications are limitless and can be easily implemented without great costs thanks to the platform available on the web. Our application of IoT is to use smart camera devices that will record their surroundings to collect and send images displaying gestures from the ISL alphabet so that they can be predicted and stored. Running computer vision algorithms at the edge unlocks many industry use cases that have low or limited internet connectivity.

2.6.2 AWS IoT Greengrass

AWS IoT Greengrass is an open-source edge runtime and cloud service for building, deploying, and managing device software. AWS IoT Greengrass provides pre-built components so you can easily extend edge device functionality without writing code. AWS IoT Greengrass components enable you to add features and quickly connect to AWS services or third-party applications at the edge. IoT devices can vary in size, ranging from smaller microcontroller-based devices to large appliances. AWS IoT Greengrass Core devices, AWS IoT Device SDK-enabled devices, and FreeRTOS devices can be configured to communicate with one another.

If the AWS IoT Greengrass Core device loses connectivity to the cloud, connected devices can continue to communicate with each other over the local network. You can deploy, run, and manage Docker containers on AWS IoT Greengrass devices. Your Docker images can be stored in Docker container registries, such as Amazon Elastic Container Registry (Amazon ECR), Docker Hub, and then used on AWS Lambda for the device's logic. Greengrass includes support for AWS Lambda. You can run AWS Lambda functions on the device to respond quickly to local events, interact with local resources, and process data to minimize the cost of transmitting data to the cloud. AWS Lambda functions deployed on an AWS IoT Greengrass Core can access local resources that are attached to the device. This allows you to use serial ports, peripherals such as the board camera, or the local file system to quickly access and process local data as you will see later. AWS IoT Greengrass ML Inference is a feature of AWS IoT Greengrass that makes it easy to perform machine learning inference locally on AWS IoT Greengrass devices using models that are built and trained in the cloud. This means you won't incur data transfer costs or increased latency for applications that use machine learning inference.

2.7 CI/CD

2.7.1 GitHub Repository

Through all the exposition of our work, we will often reference several scripts, files, or configurations that we used in different parts of the project. These are part of our approach on the implementation, we will make them all available to the public so as to ease the reproduction or possible changes to better suit your need. Our GitHub repository containing all the files separated in folders for each "phase" is published at the following link:

<https://github.com/alessandromigliore/TesiObjectDetection>

2.7.2 AWS CodeCommit

AWS CodeCommit is a secure, highly scalable, managed source control service that hosts private Git repositories. It makes it easy for teams to securely collaborate on code with contributions encrypted in transit and at rest. CodeCommit eliminates the need for you to manage your own source control system or worry about scaling its infrastructure. You can use CodeCommit to store anything from code to binaries. It supports the standard functionality of Git, so it works seamlessly with your existing Git-based tools. AWS CodeCommit stores your repositories in Amazon S3 and Amazon DynamoDB. Your encrypted data is redundantly stored across multiple facilities. This architecture increases the availability and durability of your repository data. You can transfer your files to and from AWS CodeCommit using HTTPS or SSH, as you prefer. Your repositories are also automatically encrypted at rest through AWS Key Management Service (AWS KMS) using customer-specific keys.

2.7.3 AWS CodePipeline

AWS CodePipeline is a continuous delivery service that helps you automate your build and deploy stages, which leads to faster product delivery and early mitigation of issues. AWS CodePipeline sits in a CI/CD setup, it lets you integrate your source code from multiple sources, such as GitHub, Amazon S3, AWS CodeCommit, etc, monitor it continuously for any code change, and generate builds when any change is detected. Just like any continuous delivery service, AWS CodePipeline helps teams deliver features and updates rapidly.

2.7.4 AWS CloudFormation

AWS CloudFormation lets you model, provision, and manage AWS and third-party resources by treating infrastructure as code. With CloudFormation you can automate, test, and deploy infrastructure templates with continuous integration and delivery (CI/CD) automation. It gives you an easy way to model a collection of related AWS and third-party resources, provision them quickly and consistently, and manage them throughout their lifecycles, by treating infrastructure as code. A CloudFormation template describes your desired resources and their dependencies so you can launch and configure them together as a stack. You can use a template to create, update, and delete an entire stack as a single unit, as often as you need to, instead of managing resources individually. AWS also provides a visual designer, that can be used if you don't have familiarity with YAML files, to have a clear view of how all the different services are connected and interact with each other.

Chapter 3

Use Case

3.1 Background

We find that in today's society, inclusion is one of the most important topics discussed daily, people live, work and generally feel better when they are not abandoned, when society thinks about their needs and does everything in its power to help them. With the aid of Machine Learning and IoT we think we can help, even if just a little, people to live in a world where they feel they could belong. For this reason, we decided to create an easily implementable project to convince public and private businesses to adopt our product in favor of the inclusion of speech-impaired people since, unfortunately, the vast majority of people don't know Sign Language as they do not deem it necessary. In this paper we will show every necessary step in order to implement completely the project; remember that this is just an application and is not by any means at its final stage and can be modified to respond to other needs. We would be happy if we have helped in any way.

3.2 International Sign Language

Sign languages are natural languages that have the same linguistic properties as spoken languages. They have evolved over years in the different Deaf Communities across the world and Europe. Despite widespread opinions, there is not one single universal sign language in the world or even in Europe. Just as spoken languages, sign languages vary greatly between countries and ethnic groups. Some countries have more than one sign language or dialect. Countries that have the same spoken language do not necessarily have the same signed language (see for example Germany and Austria). In recent years Deaf people have been traveling extensively, taking part in international events, which increased the need for a "lingua franca", much like English is widely used today. Experience has shown that it is difficult to teach IS to anyone not knowing at least one or more national sign languages. For this reason, we find useful a way to help deaf people to be understood in everyday life in a society where not many non-deaf people have any knowledge of Sign Language. As a simple example, nowadays workers that are required to interface with a lot of people, like hotel or airport staff, need to have a good knowledge of the English Language as it is the one most used in a commercial environment but could be

caught off guard if a speech-impaired person tries to communicate with them. It is important to say that with our project we do not want to force in any way a global spread of a single Sign Language because it is "better this way". We find it beautiful and crucial to have differences in all the SLs since they are part of the country's roots and traditions. Hopefully, our desire to just help is clear.



Figure 3.1. International Sign Language

3.3 Thesis Objectives

As we mentioned earlier, the thesis aims to design, implement and deploy a machine learning model for AI-powered Smart Cameras connected using the IoT service AWS bringing what once was possible almost exclusively in a cloud environment closer to the edge to take advantage of its benefits.

Moreover, since this transformation brings its consequences in several fields such as the project's integration and delivery, we will make sure to reach a state where it will be easy to implement and scale our system to new devices added to the group. In fact, in the cloud environment, it is easy to maintain the code and scale the implementation since all the logic is being kept in one place and uploaded to hosts that can be accessed virtually infinite times. On the other hand in an edge environment, the topic gets more complicated since any new device must be provided with all the logic it needs and this might be time-consuming. Thankfully, using services like CloudFormation and CodeCommit we are able to create a repository where all the functions are stored and we can create the whole group with a single click assuring a CI/CD approach to the whole system. To be more precise, in our case we want to develop and create through these new technologies, a device that is able to help and involve in society all people with speech impairment, making it possible for them to collaborate or socialize quickly with anyone.

In the beginning, it will be important to focus on the construction of the model, we must in fact run the train of the model using a dataset of images that allows us to accurately recognize the sign language. Since very similar gestures can often be present, a precise distinction is required for an excellent prediction. For this model to be used correctly, we need to make our infrastructure smart. If we think in fact in the world of work or even in your own home it is understandable that implementing

this model on different devices would be very complex. For this reason, as we will see later through the use of Amazon Web Service, you can design, implement an infrastructure from CloudOps to EdgeOps. This will allow us to recognize and study the strengths or weaknesses in both cases, providing a complete view of what is the right choice to make, based on your needs. Our aim, for this reason, was to make the entire project usable and general, so that it would be possible to implement our infrastructure for similar purposes that use object detection as the main focus. In the next chapters, we will illustrate in detail the analysis, the design, and the implementation carried out at the code level to allow the modification of these parameters.

3.4 Model Training

With these premises, it is important to create and appropriately train a model to reach our goal in the best way possible. In fact, the project also wants to highlight the differences and analogies of the implementation of the same Machine Learning Object Detection Model in two different environments, namely in Edge and Cloud systems. As we mentioned already, our Object Detection (OD) model will be used to recognize the gestures shown to our smart camera from a set of gestures coming from the International Sign Language (ISL) alphabet. So the first step is to train an OD model since it is the main part of both implementations. Once we have our deployable model we can use it in a Cloud Environment to infer the gesture from an image or to create our Internet Of Things (IoT) Device that does all it needs to perform the prediction.

3.4.1 Data Preparation

The first step, and also a crucial one, in the creation of any ML model is the collection, preparation, and selection of the data that will be used for the training of the model. The quality of the data chosen will influence greatly the performance of the model, the time necessary for its training, and the reliability of the prediction. As we said several times now, our model is an Object Detection Model that will analyze images to recognize gestures from the ISL alphabet, so in order to train a model that is capable of doing this, we will have to prepare a proper dataset. To create our images that will form the set we used OpenCV through a Python script to record a video using our device's webcam and from the said video we retrieve all the frames as different images. Keep in mind that this is just a way as any other to retrieve a group of images with the same characteristics, you can use the method you prefer like just capturing the images one by one and separating them by category. Obviously, by using this method, we will have several junk frames that cannot be used cause they, are blurred, bring no value to the model cause it is identical to other frames, or are just transition frames where no gesture is shown. We have to get rid of them by performing a selection of the frames that will better carry out the final job. We repeated this process for each gesture we wanted to have in the project and ended up having 40 different images for the 30 different signals we agreed upon. A good practice to use when recording the videos is to have the final images slightly different from the previous one so that the model will work better even if

you are not holding your hand up in the exact same way you did while training; we addressed this by slowly rotating and moving our hands during the recording to show the hand in different angles. Now that we have completed this part, we can upload all the selected images in a folder of our S3 bucket so that we can create a Labeling Manifest that will be needed in the next part by GroundTruth to retrieve the images for labeling.

3.4.2 Labeling with AWS GroundTruth

With all the images in our Bucket, all we need to do to create our Labeling Manifest is to execute the corresponding script specifying the location of the frames in S3. This way a new JSON file containing the URI to the images will be created and can later be uploaded in our Bucket and retrieved by the GroundTruth console. The JSON manifest will look somewhat like this.

```
{"source-ref": "s3://signlanguagebucket/frames/frameA (1).jpg"}
{"source-ref": "s3://signlanguagebucket/frames/frameA (10).jpg"}
{"source-ref": "s3://signlanguagebucket/frames/frameA (11).jpg"}
{"source-ref": "s3://signlanguagebucket/frames/frameA (12).jpg"}
{"source-ref": "s3://signlanguagebucket/frames/frameA (13).jpg"}
...
...
{"source-ref": "s3://signlanguagebucket/frames/frameZ (5).jpg"}
{"source-ref": "s3://signlanguagebucket/frames/frameZ (6).jpg"}
{"source-ref": "s3://signlanguagebucket/frames/frameZ (7).jpg"}
{"source-ref": "s3://signlanguagebucket/frames/frameZ (8).jpg"}
{"source-ref": "s3://signlanguagebucket/frames/frameZ (9).jpg"}
```

Now we can start with the labeling job. The labeling is a crucial phase of our model training because we will be saying to our model that a definite image corresponds to a definite gesture so that our algorithm will learn to discern and classify different signs. AWS GroundTruth comes to our help during this phase by retrieving the images referenced in the manifest and providing us with a tool that allows us to draw a bounding box around the hand making the gesture; let's see how to set this task. From the AWS SageMaker Console access the GroundTruth section and select Labeling Jobs, here you can specify an Input Dataset Location that is the path to your manifest, and an Output Dataset Location that indicates where you want your final output to be stored. Then you have to specify the type of task you want to label; there are several with different characteristics depending on what you wish to accomplish. In our case, we will select Bounding Box as the Task Type since we want to classify and annotate the position of the gesture.

Task type
[Info](#)


Task category
 Select the type of data being labeled to view available task templates for it or select 'Custom' to create your own.

Image ▼

Task selection
 Select the task that a human worker will perform to label objects in your dataset.


☐ **Image Classification (Single Label)**
 Get workers to categorize images into individual classes. [Info](#)


☒ **Basketball**
☐ **Soccer**



☐ **Image Classification (Multi-label)**
 Get workers to categorize images into one or more classes. [Info](#)

☒ **Human**
☒ **Vehicle**
☐ **Animal**



☒ **Bounding box**
 Get workers to draw bounding boxes around specified objects in your images. [Info](#)



☐ **Semantic segmentation**
 Get workers to draw pixel level labels around specific objects and segments in your images. [Info](#)


Figure 3.2. GroundTruth Task Types

The next step is to select the workforce that will label your images. There are different options available, each with its costs, pros, and cons. You can select an external workforce that will label your data for you, this will considerably speed up your job but it is obviously paid and need you to give clear instruction to the workers if you want a job well done. Alternatively, you can select a private workforce (which can consist even of you and your friends) and work on your data yourself. Keep in mind though that labeling is a time-consuming job, and depending on the number of images that need to be labeled can become obnoxious and can take some time. With this said, you can add the classes of your images and add some good and bad examples in case you use an external workforce; when the labeling job is completed, you will have an output file containing the image name, the box position, the corresponding class and other metadata used later in the training. A good practice with this kind of labeling is to draw the box as close as possible to the object and not to include parts of the object that are overlapping or that cannot be seen, even though you think you can interpolate the whole shape. Also, you might

want to check the labeling job from the console to see if some misclassifications are present.

3.4.3 Training with AWS SageMaker

Process SageMaker Ground Truth labeling job outputs for training

The output of the SageMaker Ground Truth bounding box labeling job is in a format called augmented manifest file. Using an augmented file allows us to gain numerous benefits from our training input. For example, there is no need for a format conversion if we use Sagemaker Ground truth to generate data labels, or instead of the traditional approach of providing paths to the input images separately from its labels, augmented manifest file already combines both into one entry for each input image, reducing complexity in algorithm code for matching each image with labels. Also after splitting our dataset into train/validation instead of re-uploading our file images into the respective folders in the S3. Once you upload your image files to S3, you never need to move them again, you can just place pointers to these images in your augmented manifest file for training and validation. A last important note about the augmented file is that with its use the training input images are loaded onto the training instance in Pipe mode, which means the input data is streamed directly to the training algorithm while it is running. This results in faster training performance and less disk resource utilization. In our thesis project although our augmented file is supported by Amazon Sagemaker it may be appropriate to perform some small processing:

- Join outputs from multiple jobs: To be able to iterate on Ground Truth jobs, we created several smaller labeling jobs for our dataset instead of a single large job containing the full dataset.
- Discard any bad labels from visual inspection: this step is optional, you may manually review the labeled bounding boxes on the Ground Truth console and mark the image IDs that didn't pass a quality bar.
- Inject the correct class labels: This step is useful in case the labeler has not been asked, during the labeling work done on Ground Truth, to choose the right class when drawing the delimiting boxes. This means that in our manifest there is only one class_id: 0 because you only specified one class of objects when you submitted the labeling job.
- Split dataset between train and validation: As mentioned above, Amazon Sagemaker requires you to split the dataset during training, a train, and a validation dataset. The training set consists of all frames and all records used to train the model, while validation is used to validate that the model can accurately make predictions on previously unseen data, and is also used to compare accuracy between different training jobs during hyperparameter tuning.
- Data Augmentation (optional): This simple technique allows us to reach a higher accuracy quickly and cheaply, through two python scripts in fact we

create a copy for each frame rotated on the axis of x and y of 90 degrees. We found that even simple data augmentation like this can make a significant difference in accuracy.

Amazon SageMaker Object Detection using the augmented manifest file format

Now that we have finished with all the required setups, we can start our training jobs. Using boto3 SDK, by Amazon Sagemaker, we can define input train and validation.

```
{
  "source-ref": "s3://signlanguagebucket/frames/frameN (33).jpg",
  "SignLanguageDetectionTesi": {
    "image_size": [{"width": 640, "height": 480, "depth": 3}], "annotations": [{"class_id": 13, "top": 151, "left": 399, "height": 94, "width": 70}],
    "SignLanguageDetectionTesi-metadata": {"objects": [{"confidence": 0}],
    "class-map": {"0": "A", "1": "B", "2": "C", "3": "D", "4": "E", "5": "F", "6": "G", "7": "H", "8": "I", "9": "J",
    "10": "K", "11": "L", "12": "M", "13": "N", "14": "O", "15": "P", "16": "R", "17": "S", "18": "T", "19": "U",
    "20": "V", "21": "W", "22": "X", "23": "Y", "24": "Z", "25": "Hello", "26": "NO", "27": "OK", "28": "ILoveYou", "29": "ThankYou"},
    "type": "groundtruth/object-detection", "human-annotated": "yes", "creation-date": "2021-11-26T09:47:09.295936", "job-name": "labeling-job/signlanguagedetectiontesi"}}
  }
},
{
  "source-ref": "s3://signlanguagebucket/frames/frameA (16).jpg",
  "SignLanguageDetectionTesi": {
    "image_size": [{"width": 640, "height": 480, "depth": 3}], "annotations": [{"class_id": 0, "top": 181, "left": 144, "height": 78, "width": 83}],
    "SignLanguageDetectionTesi-metadata": {"objects": [{"confidence": 0}],
    "class-map": {"0": "A", "1": "B", "2": "C", "3": "D", "4": "E", "5": "F", "6": "G", "7": "H", "8": "I", "9": "J",
    "10": "K", "11": "L", "12": "M", "13": "N", "14": "O", "15": "P", "16": "R", "17": "S", "18": "T", "19": "U",
    "20": "V", "21": "W", "22": "X", "23": "Y", "24": "Z", "25": "Hello", "26": "NO", "27": "OK", "28": "ILoveYou", "29": "ThankYou"},
    "type": "groundtruth/object-detection", "human-annotated": "yes", "creation-date": "2021-11-26T10:36:18.658595", "job-name": "labeling-job/signlanguagedetectiontesi"}}
  }
},
...
{
  "source-ref": "s3://signlanguagebucket/frames/frame0 (17).jpg",
  "SignLanguageDetectionTesi": {
    "image_size": [{"width": 640, "height": 480, "depth": 3}], "annotations": [{"class_id": 14, "top": 188, "left": 490, "height": 91, "width": 91}],
    "SignLanguageDetectionTesi-metadata": {"objects": [{"confidence": 0}],
    "class-map": {"0": "A", "1": "B", "2": "C", "3": "D", "4": "E", "5": "F", "6": "G", "7": "H", "8": "I", "9": "J",
    "10": "K", "11": "L", "12": "M", "13": "N", "14": "O", "15": "P", "16": "R", "17": "S", "18": "T", "19": "U",
    "20": "V", "21": "W", "22": "X", "23": "Y", "24": "Z", "25": "Hello", "26": "NO", "27": "OK", "28": "ILoveYou", "29": "ThankYou"},
    "type": "groundtruth/object-detection", "human-annotated": "yes", "creation-date": "2021-11-26T09:58:33.485575", "job-name": "labeling-job/signlanguagedetectiontesi"}}
  }
},
{
  "source-ref": "s3://signlanguagebucket/frames/frameX (34).jpg",
  "SignLanguageDetectionTesi": {
    "image_size": [{"width": 640, "height": 480, "depth": 3}], "annotations": [{"class_id": 22, "top": 139, "left": 381, "height": 159, "width": 112}],
    "SignLanguageDetectionTesi-metadata": {"objects": [{"confidence": 0}],
    "class-map": {"0": "A", "1": "B", "2": "C", "3": "D", "4": "E", "5": "F", "6": "G", "7": "H", "8": "I", "9": "J",
    "10": "K", "11": "L", "12": "M", "13": "N", "14": "O", "15": "P", "16": "R", "17": "S", "18": "T", "19": "U",
    "20": "V", "21": "W", "22": "X", "23": "Y", "24": "Z", "25": "Hello", "26": "NO", "27": "OK", "28": "ILoveYou", "29": "ThankYou"},
    "type": "groundtruth/object-detection", "human-annotated": "yes", "creation-date": "2021-11-26T10:54:22.754448", "job-name": "labeling-job/signlanguagedetectiontesi"}}
  }
}
```

Figure 3.3. Output JSON File

Inserting in S3 URI the link to our bucket contain the input images, and attribute name for the bounding box annotations. Next, set the hyperparameters. You can find documentation for all the supported hyperparameters in the Amazon SageMaker documentation. In this case, we recommend leaving the recommended parameters, knowing that our algorithm uses a base_network, which is typically a VGG or a ResNet, and changing only the number of classes, according to your choice.

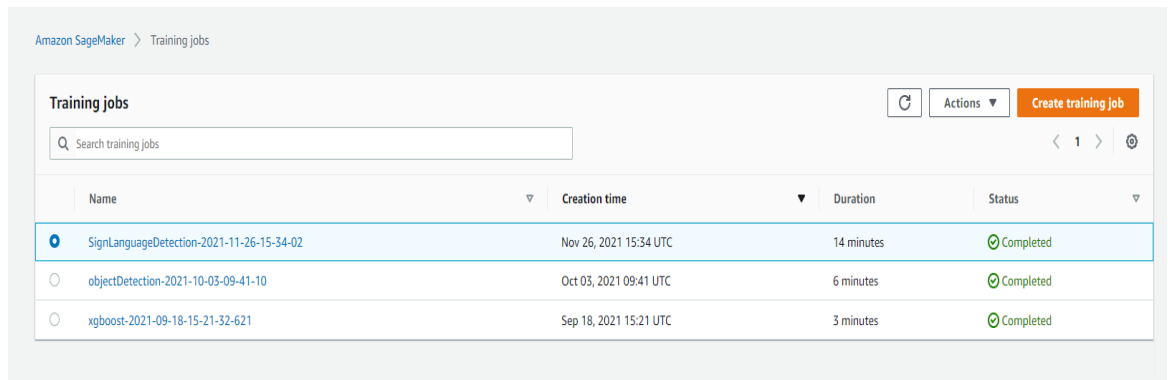
```
{
  "base_network": "resnet-50",
  "use_pretrained_model": "1",
  "num_classes": "30",
  "mini_batch_size": "10",
  "epochs": "10",
  "learning_rate": "0.001",
  "lr_scheduler_step": "10,20",
  "lr_scheduler_factor": "0.25",
  "optimizer": "sgd",
  "momentum": "0.9",
}
```

```

"weight_decay": "0.0005",
"overlap_threshold": "0.5",
"nms_threshold": "0.45",
"num_training_samples": "5400",
"image_shape": "512",
"_tuning_objective_metric": "",
"_kvstore": "device",
"kv_store": "device",
"_num_kv_servers": "auto",
"label_width": "150",
"freeze_layer_pattern": "",
"nms_topk": "400",
"early_stopping": "False",
"early_stopping_min_epochs": "10",
"early_stopping_patience": "5",
"early_stopping_tolerance": "0.0",
"_begin_epoch": "0"}

```

Now we just have to create our own SageMaker training jobs. To carry out this last part you need special permissions from the Amazon platform, you can get the necessary resources by contacting the support and indicating our development region and the notebook instance that requires the resource. Once the necessary resources have been obtained we create the SageMaker training job.



Amazon SageMaker > Training jobs

Training jobs 🔄 Actions ▼ Create training job

🔍 Search training jobs < 1 > ⌂

Name	Creation time	Duration	Status
SignLanguageDetection-2021-11-26-15-34-02	Nov 26, 2021 15:34 UTC	14 minutes	🟢 Completed
objectDetection-2021-10-03-09-41-10	Oct 03, 2021 09:41 UTC	6 minutes	🟢 Completed
xgboost-2021-09-18-15-21-32-621	Sep 18, 2021 15:21 UTC	3 minutes	🟢 Completed

Figure 3.4. SageMaker Training Jobs

Once the training job completes, move on to the next notebook to convert the trained model to a deployable format and run local inference.

3.5 Creating Deployable Model

Now we have completed the training of our model, but in order to check its functioning, we need to adjust our model artifact so that it can be deployed.

The trained model parameters along with its network definition are stored in a tar.gz

file in the output path for the training job. We need to download and unzip it to a local disk in order to work on it. Inside the tar file you will find 3 files stored that together form our trained model :

- `model_algo_1-symbol.json`: neural network definition
- `hyperparams.json`: hyperparameters
- `model_algo_1-0000.params`: trained weights for the neural network

The model output produced by the built-in object detection model leaves the loss layer in place and does not include a non-max suppression (NMS) layer. To make it ready for inference on our machine, we need to remove the loss layer and add the NMS layer. We will be using a script from this GitHub repo: <https://github.com/zhrshold/mxnet-ssd> Make sure to execute the script just as we do in the provided ipynb Notebook because the parameters passed in the function must correspond to the hyperparameters you used in the training phase or it will not work. This will create two files that will be uploaded in our S3 Bucket.

- `deploy_model_algo_1-0000.params`
- `deploy_model_algo_1-symbol.json`

Now, if we want to, we can perform a test inference selecting an image, deploying the model and checking if the classification is coherent with our training. You can do it by following the mentioned notebook that will guide you in the creation of a PDF file with the predictions of the selected images of a batch. This is our result.

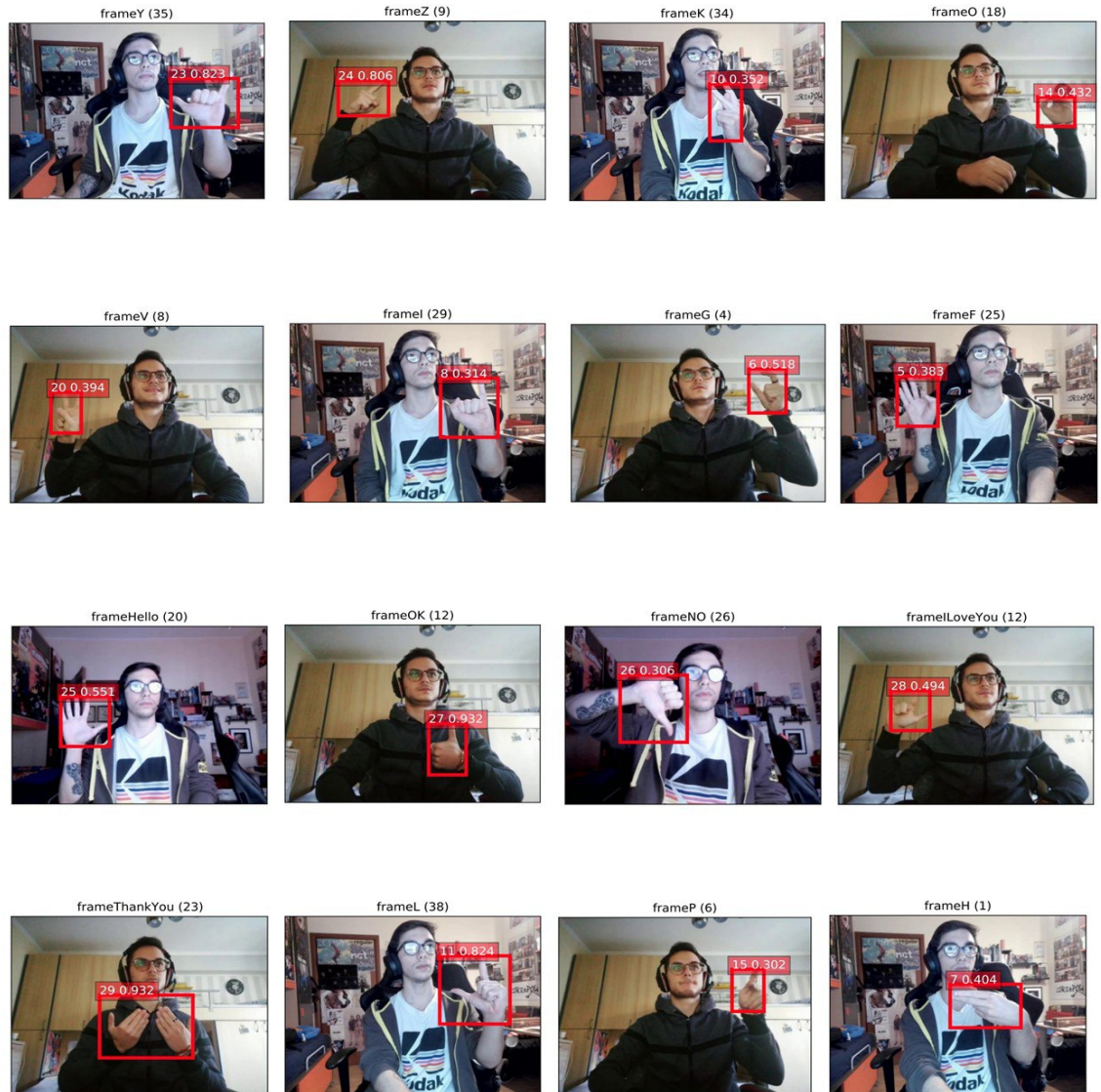


Figure 3.5. Report Visualization

Now we are ready for the real implementation of the model; we will start with the Edge Environment where we will create a GreenGrass device to perform the prediction.

Chapter 4

Cloud-Based Services

4.1 Introduction

Our main interest in creating a Cloud Environment implementation is the fact that you don't need any personal infrastructure since you will be using services already present on the internet. In fact, here obviously we don't have any external device that can perform the prediction so all the separate images need to be sent to the function in charge of loading the model, which will later provide the upload of the recognized gesture on our Database.

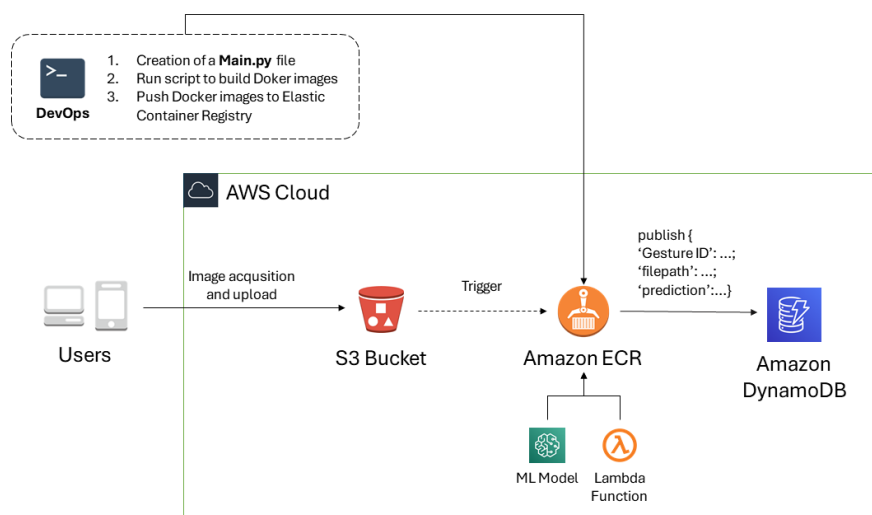


Figure 4.1. Cloud Architecture

The system is composed of a local device, external to the environment, that is in charge of capturing the images. Those images will be first uploaded to an S3 Bucket and then retrieved by our Lambda that will perform the prediction and will finally create an item in our DynamoDB. Let's see all these steps one by one.

4.2 Images collection and upload

Since we don't have a smart device with a camera, we need a computer external to our system that with a webcam captures the images so that they can be sent for prediction to our function. The script used to achieve this task will be pretty similar to the one used in the edge system; the main difference is that the images retrieved from the video will be uploaded since we need a place where to store them for later. To do so we will use the boto3 library to connect to our bucket in the specified region.

```

1 import cv2
2 import time
3 import logging
4 import json
5 import os
6 import time
7 import boto3
8 from botocore.client import Config
9 from botocore.exceptions import ClientError
10 import numpy as np
11 import urllib
12
13
14 def main():
15     video_fname = '#PATH FOLDER/VIDEO NAME'
16     camera_id = 0
17
18     cap = cv2.VideoCapture(camera_id)
19
20     # Define the codec and create VideoWriter object. For mac mp4v or
21     # av1 is the best option.
22     # You can also use: 0x00000021 if this codec doesn't work for you
23     fourcc = cv2.VideoWriter_fourcc(*'MJPG')
24
25     # Create a video writer, specify the codec as well as the image
26     # widge and height.
27     # cap.get(3) is the width, and cap.get(4) is the height of the
28     # camera in cap.
29     out = cv2.VideoWriter(video_fname, fourcc, 5.0, (int(cap.get(3)),
30     int(cap.get(4))))
31
32     start_time = time.time()
33
34     while( int(time.time() - start_time) < 10 ):
35         ret, frame = cap.read()
36
37         if ret==True:
38             out.write(frame)
39
40             #if cv2.waitKey(1) & 0xFF == ord('q'):
41             # print("\nstop signal received.")
42             # break
43
44         else:
45             break

```

```

43
44 # When everything done, release the capture
45 cap.release()
46 out.release()
47 cv2.destroyAllWindows()
48
49
50 # Opens the Video file
51 cap2= cv2.VideoCapture('#PATH FOLDER /VIDEO NAME')
52 i=0
53 while(cap2.isOpened()):
54     ret, frame = cap2.read()
55     if ret == False:
56         break
57
58     if i % 10 == 0:
59         cv2.imwrite('#PATH FOLDER/frame'+str(i)+'.jpg',frame)
60         client = boto3.client('s3', region_name='eu-west-1')
61         client.upload_file('#PATH FOLDER/frame'+str(i)+'.jpg', '#BUCKET
        ', 'images/frame'+str(i)+'.jpg')
62         i+=1
63
64 cap2.release()
65 cv2.destroyAllWindows()
66
67 main()

```

Listing 4.1. Video Recording Function

Each image uploaded in the bucket will later trigger the Lambda function that is the hearth of our Cloud implementation.

4.3 The Lambda Function

The main part of the Cloud Environment is without a doubt the creation of Lambda function that by itself needs to receive the images, load the model, perform the prediction, create an item and finally upload that item to the DynamoDB. All of this obviously requires a relevant amount of space that the "classical" configuration of AWS Lambda does not support; in fact, Lambda Functions can only be created from zip files if the total size between the main file plus any additional layer does not exceed 250MBs of space. This amount is way too strict since we need libraries like OpenCV, MXNET, and the model itself that are more than 100MBs each. The way around this limitation is the fact that Amazon allows the creation of Lambdas with container images with a maximum size of 10GBs which is more than enough. This means that we need a container image holding all our files and libraries, let's see how to do that.

4.3.1 Lambda with Container Image

Using container images for your lambdas you can easily build and deploy larger workloads that rely on sizable dependencies, such as machine learning or data-intensive workloads. Just like functions packaged as ZIP archives, functions deployed as container images benefit from the same operational simplicity, automatic scaling,

high availability, and native integration with many services. Amazon provides base images for all the supported Lambda runtimes (Python, Node.js, Java, .NET, Go, Ruby) so that you can easily add your code and dependencies. So we will use the Python base image as a start for our container where model and libraries will be added. The base image for Python can be found at the following link:

<https://docs.aws.amazon.com/lambda/latest/dg/python-image.html> Before creating the Dockerfile, we need the main Python script that does what we described earlier; it will retrieve the item uploaded to the S3 bucket specified, will create an image from the raw data retrieved, it then calls the script in charge of loading the model and the prediction, from the prediction result it creates an item that will be put inside our database. Notice that to access your bucket and database from script you need to provide your Amazon credentials that were removed from this snippet for security. In case you want to reproduce this script functioning replace it with your credentials.

```

1 #
2 # Copyright 2010-2017 Amazon.com, Inc. or its affiliates. All Rights
  Reserved.
3 #
4
5 # Lambda entry point
6 from model_loader import MLModel
7 import logging
8 import os
9 import time
10 import json
11 import boto3
12 from botocore.client import Config
13 from botocore.exceptions import ClientError
14 import os
15 import numpy as np
16 import urllib
17 import cv2
18
19 dynamodb = boto3.resource("dynamodb", region_name="eu-west-1")
20 tableName = "#DATABASENAME"
21 predicted_gesture = "none"
22 seven_days_as_seconds = 604800
23
24 s3_signature = {
25     'v4': 's3v4',
26     'v2': 's3'
27 }
28
29 ML_MODEL_BASE_PATH = 'model/'
30 ML_MODEL_PREFIX = 'deploy_model_algo_1'
31 ML_MODEL_PATH = os.path.join(ML_MODEL_BASE_PATH, ML_MODEL_PREFIX)
32
33
34 #client=boto3.client('iot-data', endpoint_url='-----.iot.eu-
  west-1.amazonaws.com')
35
36 model = None
37
38 # Load the model at startup

```



```

39 def initialize(param_path=ML_MODEL_PATH):
40     global model
41     model = MLModel(param_path)
42
43
44 def lambda_handler(event, context):
45     bucket_name = event['Records'][0]['s3']['bucket']['name']
46     key = event['Records'][0]['s3']['object']['key']
47
48     generated_signed_url = create_presigned_url(bucket_name, key,
49         seven_days_as_seconds, s3_signature['v4'])
50     print(generated_signed_url)
51     image_complete = url_to_image(generated_signed_url)
52
53     start = int(round(time.time() * 1000))
54     prediction = model.predict_from_file(image_complete)
55     end = int(round(time.time() * 1000))
56
57     response = {
58         'prediction': prediction,
59         'timestamp': time.time()
60     }
61
62     if prediction[0][0] == 0:
63         predicted_gesture = "A"
64     elif prediction[0][0] == 1:
65         predicted_gesture = "B"
66     elif prediction[0][0] == 2:
67         predicted_gesture = "C"
68     ...
69     ...
70     ...
71     elif prediction[0][0] == 25:
72         predicted_gesture = "Hello"
73     elif prediction[0][0] == 26:
74         predicted_gesture = "NO"
75     elif prediction[0][0] == 27:
76         predicted_gesture = "YES"
77     elif prediction[0][0] == 28:
78         predicted_gesture = "ILoveYou"
79     elif prediction[0][0] == 29:
80         predicted_gesture = "ThankYou"
81
82     global tableName
83     table = dynamodb.Table(tableName)
84     table.put_item(
85         Item={
86             "Gesture ID": str(time.time()),
87             "prediction": predicted_gesture,
88             "filepath": key
89         }
90     )
91
92     return response
93
94
95 def url_to_image(URL):

```

```

96     resp = urllib.request.urlopen(URL)
97     image = np.asarray(bytearray(resp.read()), dtype="uint8")
98     image = cv2.imdecode(image, cv2.IMREAD_COLOR)
99
100     return image
101
102 def create_presigned_url(bucket_name, bucket_key, expiration=3600,
103     signature_version=s3_signature['v4']):
104
105     s3_client = boto3.client('s3',
106         aws_access_key_id=#YOUR ACCESS KEY ID",
107         aws_secret_access_key=#YOUR SECRET ACCESS
108     KEY,
109         config=Config(signature_version=
110     signature_version),
111         region_name='eu-west-1'
112     )
113
114     try:
115         response = s3_client.generate_presigned_url('get_object',
116     Params={'Bucket': bucket_name, 'Key': bucket_key}, ExpiresIn=
117     expiration)
118         print(s3_client.list_buckets()['Owner'])
119         for key in s3_client.list_objects(Bucket=bucket_name, Prefix=
120     bucket_key)['Contents']:
121             print(key['Key'])
122     except ClientError as e:
123         logging.error(e)
124         return None
125
126     # The response contains the presigned URL
127
128     return response
129
130 # If this path exists then this code is running on the greengrass
131 # core and has the ML resources it needs to initialize.
132 if os.path.exists(ML_MODEL_BASE_PATH):
133     initialize()
134 else:
135     logging.info('{} does not exist and we cannot initialize this lambda
136     function.'.format(ML_MODEL_BASE_PATH))

```

Listing 4.2. Container Image Main File

Once our main function file is completed, we can change the example Dockerfile to fit our needs. As you can see, we are using the public lambda Python 3.8 image offered by Amazon Web Services so that we have the basic libraries and configurations already prepared. First, we load both `main.py` and `model_loader.py` that we need for the prediction, then we copy the model artifact in a new folder that we called `model`, we install libraries and dependencies and finally, we set the CMD to your handler.

```

1 FROM public.ecr.aws/lambda/python:3.8
2
3 # Copy function code
4 COPY main.py ${LAMBDA_TASK_ROOT}
5
6 COPY model_loader.py ${LAMBDA_TASK_ROOT}
7

```

```

8 RUN mkdir model
9
10 COPY deploy_model_algo_1-symbol.json ${LAMBDA_TASK_ROOT}/model
11
12 COPY deploy_model_algo_1-0000.params ${LAMBDA_TASK_ROOT}/model
13
14 COPY hyperparams.json ${LAMBDA_TASK_ROOT}/model
15
16 # Install the function's dependencies using file requirements.txt
17 # from your project folder.
18
19 COPY requirements.txt .
20 RUN pip3 install -r requirements.txt --target "${LAMBDA_TASK_ROOT}"
21
22 RUN yum -y install tar gzip zlib freetype-devel \
23     gcc \
24     ghostscript \
25     lcms2-devel \
26     libffi-devel \
27     libimagequant-devel \
28     libjpeg-devel \
29     libraqm-devel \
30     libtiff-devel \
31     libwebp-devel \
32     make \
33     openjpeg2-devel \
34     rh-python36 \
35     rh-python36-python-virtualenv \
36     sudo \
37     tcl-devel \
38     tk-devel \
39     tkinter \
40     which \
41     xorg-x11-server-Xvfb \
42     zlib-devel \
43     && yum clean all
44
45 RUN yum -y install libgomp
46
47 RUN yum -y install libquadmath
48
49 RUN yum -y install mesa-libGL
50
51 # Set the CMD to your handler (could also be done as a parameter
52   # override outside of the Dockerfile)
53 CMD [ "main.lambda_handler" ]

```

Listing 4.3. Dockerfile

Before building the container image, make sure your function folder configuration looks like this to ensure the building will go through fine and to have a perfectly working container since this is a time consuming process and re-building many times cause something went wrong can be annoying.

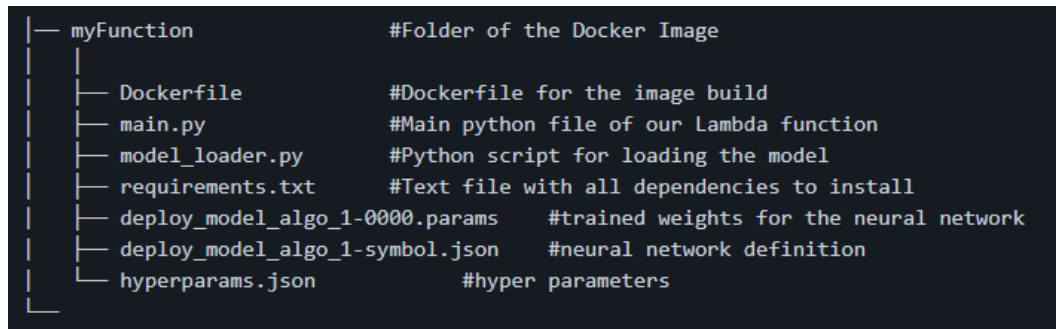


Figure 4.2. Lambda Function Folder

Now that our Dockerfile is complete we can use the Docker CLI to build the lambda container image locally. When this step runs without any error we can continue. To upload the container image, I create a new ECR repository in my account and tag the local image to push it to ECR. To help me identify software vulnerabilities in my container images, I enable ECR image scanning.

```
$ aws ecr create-repository --repository-name lambda
--image-scanning-configuration scanOnPush=true
$ docker tag lambda:latest
123412341234.dkr.ecr.eu-west-1.amazonaws.com/lambda:latest
$ aws ecr get-login-password --region eu-west-1 | sudo docker login
--username AWS --password-stdin
123412341234.dkr.ecr.eu-west-1.amazonaws.com
$ docker push 123412341234.dkr.ecr.eu-east-1.amazonaws.com/lambda:latest
```

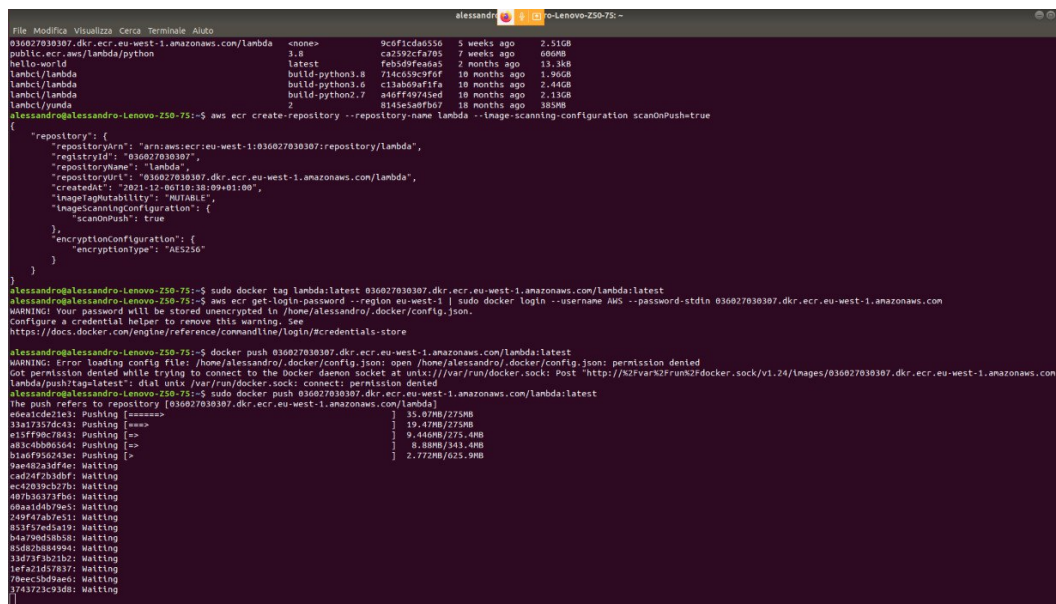


Figure 4.3. Pushing Container Image in ECR

In order to execute some of these commands, make sure you have correctly configured your AWS CLI. By completing this passage we have successfully uploaded

our container image that holds all is necessary to execute our lambda function on Cloud. So we now need to create the function itself. From the Lambda console in AWS select the "Container Image" option and browse to find the right image in your ECR repositories. After I select the repository, I use the latest image I uploaded. When I select the image, the Lambda is translating that to the underlying image digest. You can see the digest of your images locally with the `docker images --digests` command. In this way, the function is using the same image even if the latest tag is passed to a newer one, and you are protected from unintentional deployments. You can update the image to use in the function code. Updating the function configuration has no impact on the image used, even if the tag was reassigned to another image in the meantime.

The screenshot shows the AWS Lambda 'Crea funzione' (Create function) page. At the top, there are four tabs: 'Crea da zero', 'Usa un piano', 'Immagine del container' (which is selected), and 'Sfoglia il repository dell'app serverless'. Below the tabs, there is a section titled 'Informazioni di base' (Basic information). It contains a text input for 'Nome funzione' (Function name) with the value 'Cloudloginfer'. Below that, there is a text input for 'URI dell'immagine del container' (Container image URI) with the value '056027030307.dkr.ecr.eu-west-1.amazonaws.com/lambda/sha256:fceeb3a1bc93f89b6155eb51a9846909e332ae0635c144c5c025b080720a7'. There is a button 'Sfoglia le immagini' (Browse images) next to the URI input. Below the URI input, there is a section titled 'Architettura' (Architecture) with two radio buttons: 'x86_64' (selected) and 'arm64'. At the bottom, there is a section titled 'Autorizzazioni' (Permissions) with a link 'Modifica del ruolo di esecuzione predefinito' (Modify default execution role).

Figure 4.4. Creating Lambda with Container Image

Then we can leave all other options untouched. We can now create our Lambda, it will take some time. As an important note if you are using AWS Free Tier during the implementation of this project, uploading a function of this size in an Amazon Elastic Registry (ECR) will rapidly consume all your free use for this service. We noticed that after completing these steps, two days of execution had already exceeded our monthly quota and it started using standard pricing. So, if you want to avoid unwanted expenses we suggest you give an eye out on the billing section every now and then to have a clear vision of the cost management.

4.3.2 Lambda Configuration and Trigger

The function is now ready, but we still need to make some adjustments to make it work properly. First of all, since the function is clearly pretty heavy, we need to grant more memory and a greater timeout interval or it won't be able to complete the work in time. This can be easily achieved by working on the function configuration in the specific section. Now, what we need is a way for the function to be invoked on each frame that has been sent from our local Python script. This is the reason

why we decided to upload those images in our bucket so that we could add a trigger to our function to be executed every time a new file with an image extension is loaded in a specific folder of our S3. To create this just click add a new trigger, select S3 for the trigger configuration, and set other parameters like shown in the figure.

Add trigger

Trigger configuration

S3
aws storage

Bucket
Please select the S3 bucket that serves as the event source. The bucket must be in the same region as the function.
07081996

Event type
Select the events that you want to have trigger the Lambda function. You can optionally set up a prefix or suffix for an event. However, for each bucket, individual events cannot have multiple configurations with overlapping prefixes or suffixes that could match the same object key.
All object create events

Prefix - optional
Enter a single optional prefix to limit the notifications to objects with keys that start with matching characters.
images/

Suffix - optional
Enter a single optional suffix to limit the notifications to objects with keys that end with matching characters.
.jpg

Lambda will add the necessary permissions for Amazon S3 to invoke your Lambda function from this trigger. [Learn more about the Lambda permissions model.](#)

Recursive invocation
If your function writes objects to an S3 bucket, ensure that you are using different S3 buckets for input and output. Writing to the same bucket increases the risk of creating a recursive invocation, which can result in increased Lambda usage and increased costs. [Learn more](#)

☒ I acknowledge that using the same S3 bucket for both input and output is not recommended and that this configuration can cause recursive invocations, increased Lambda usage, and increased costs.

Cancel Add

Figure 4.5. Trigger Configuration

4.4 Conclusions and Tests

Now we have configured all the necessary services to test out the function. We can now upload a JPG image to the created S3 bucket by opening the bucket in the AWS management console and clicking Upload. This will invoke the function on the uploaded frame and its prediction will be uploaded to the DynamoDB specified in the main.py file we pushed in the container image. If this test works correctly, we can even test the function as a whole by executing our local video recording script that should upload images on the bucket for them to be retrieved by the function. The function will then load the model, recognize the gesture and create a new item to put in our DB table. With this, we have completed our Cloud Environment Implementation.

images/ Copia URI S3

Oggetti Proprietà

Oggetti (15)

Gli oggetti sono le entità fondamentali archiviate in Amazon S3. Per ottenere un elenco di tutti gli oggetti nel bucket, puoi utilizzare [l'inventario di Amazon S3](#). Per consentire ad altri utenti di accedere ai tuoi oggetti, è necessario concedere loro le autorizzazioni esplicitamente. [Ulteriori informazioni](#)

Copia URI S3 Copia URL Scarica Apri Elimina Operazioni Crea cartella Carica

Q Trova oggetti per prefisso

<input type="checkbox"/>	Nome	Tipo	Ultima modifica	Dimensioni	Classe di storage
<input type="checkbox"/>	frame0.jpg	jpg	06 Dec 2021 11:19:47 AM CET	39.2 KB	Standard
<input type="checkbox"/>	frame10.jpg	jpg	06 Dec 2021 11:19:48 AM CET	48.2 KB	Standard
<input type="checkbox"/>	frame100.jpg	jpg	06 Dec 2021 11:19:53 AM CET	46.1 KB	Standard
<input type="checkbox"/>	frame110.jpg	jpg	06 Dec 2021 11:19:53 AM CET	46.3 KB	Standard
<input type="checkbox"/>	frame120.jpg	jpg	06 Dec 2021 11:19:54 AM CET	47.0 KB	Standard
<input type="checkbox"/>	frame130.jpg	jpg	06 Dec 2021 11:19:55 AM CET	46.6 KB	Standard
<input type="checkbox"/>	frame140.jpg	jpg	06 Dec 2021 11:19:55 AM CET	47.0 KB	Standard
<input type="checkbox"/>	frame20.jpg	jpg	06 Dec 2021 11:19:48 AM CET	47.9 KB	Standard
<input type="checkbox"/>	frame30.jpg	jpg	06 Dec 2021 11:19:49 AM CET	48.5 KB	Standard
<input type="checkbox"/>	frame40.jpg	jpg	06 Dec 2021 11:19:49 AM CET	48.9 KB	Standard
<input type="checkbox"/>	frame50.jpg	jpg	06 Dec 2021 11:19:50 AM CET	49.1 KB	Standard
<input type="checkbox"/>	frame60.jpg	jpg	06 Dec 2021 11:19:51 AM CET	48.9 KB	Standard
<input type="checkbox"/>	frame70.jpg	jpg	06 Dec 2021 11:19:51 AM CET	48.4 KB	Standard
<input type="checkbox"/>	frame80.jpg	jpg	06 Dec 2021 11:19:52 AM CET	47.2 KB	Standard
<input type="checkbox"/>	frame90.jpg	jpg	06 Dec 2021 11:19:52 AM CET	46.8 KB	Standard

(a) Images Loaded to the Bucket

Voci restituite (28) Copia Operazioni Crea voce

< 1 ... > Impostazioni Stampa

<input type="checkbox"/>	Gesture ID	filepath	prediction
<input type="checkbox"/>	1635258535.2870028	images/frame90.jpg	I Love You
<input type="checkbox"/>	1635864834.9030282	images/frame60.jpg	I Love You
<input type="checkbox"/>	1635855026.175429	images/frame20.jpg	NO
<input type="checkbox"/>	1635866307.0172663	images/frame130.jpg	Thank You
<input type="checkbox"/>	1635866301.1642087	images/frame120.jpg	Hello
<input type="checkbox"/>	1637315789.3692136	images/frame100.jpg	OK
<input type="checkbox"/>	1635866317.389197	images/frame10.jpg	NO
<input type="checkbox"/>	1635866300.5409946	images/frame0.jpg	NO
<input type="checkbox"/>	1635258288.8984847	/shared/greengrass/buffer/frame90.jpg	Hello
<input type="checkbox"/>	1635870467.6998	/shared/greengrass/buffer/frame80.jpg	Thank You
<input type="checkbox"/>	1637230426.0475419	/shared/greengrass/buffer/frame70.jpg	I Love You
<input type="checkbox"/>	1635258522.2904282	/shared/greengrass/buffer/frame50.jpg	I Love You
<input type="checkbox"/>	1637230614.6049523	/shared/greengrass/buffer/frame40.jpg	Hello
<input type="checkbox"/>	1637230610.9726896	/shared/greengrass/buffer/frame30.jpg	I Love You
<input type="checkbox"/>	1635258301.5290308	/shared/greengrass/buffer/frame20.jpg	NO
<input type="checkbox"/>	1637252656.3158529	/shared/greengrass/buffer/frame110.jpg	Hello

(b) Prediction stored in the Database

Figure 4.6. Results of Execution of the Cloud Function

Chapter 5

Edge-Based Services

5.1 Introduction

The next step in the building of this project is the actual transposition of the whole system from a Cloud-Centric implementation to the Edge-Based one. Obviously the most obvious difference between the two at first sight is that in Edge we need a smart device that is in charge of the computation closer to the source of data in order to improve response times. In Edge the device will take care of recording the video, retrieving the frames, invoking the function to receive the prediction based on the frame sent and finally sending that same prediction to the DynamoDB through the web.

5.2 AWS IoT Greengrass Core software and Greengo Deployment

The first step will be to install and configure the AWS IoT Greengrass Core software on a Linux device, such as a Raspberry Pi, or a Windows device. This device is a Greengrass core device. For the sake simplicity, we will use a VirtualBox VM on a Windows host that mounts an Ubuntu distribution. The instructions on how to do that can be easily found on Amazon documentation at the following link : <https://docs.aws.amazon.com/greengrass/v1/developerguide/what-is-gg.html>

If you followed the tutorial at the end you will have installed the GreenGrass Core software and will have provided the AWS credential to execute commands through the AWS CLI. Then we will create the GreenGrass Group and Core Device. there are several ways to do so all analogous; you can use the AWS IoT Core console to use a Graphical Interface to do all the work, you can execute ad hoc commands using AWS CLI, or you use Greengo. Greengo is an open-source project that uses a YAML file to describe a GreenGrass group, device, lambdas, subscription, and resources to have everything created automatically with a simple command. The only problem is that it works on Python2.7 that was recently removed by Amazon for their Lambdas Runtime, so you will need to work around that. In the greengo.yaml file, which defines configurations and lambda functions for an IoT Greengrass Group the top portion of the file defines the name of the IoT Greengrass Group and IoT Greengrass Cores:

```

1
2 Group:
3   name: GG_Object_Detection
4 Cores:
5   - name: GG_Object_Detection_Core
6     key_path: ./certs
7     config_path: ./config
8     SyncShadow: True
9
10 Resources:
11   - Name: MyObjectDetectionModel
12     Id: MyObjectDetectionModel
13     S3MachineLearningModelResourceData:
14       DestinationPath: /ml/od/
15       S3Uri: s3://bucketmigliore/deploy_model.tar.gz

```

Listing 5.1. Greengo YAML

Then by executing the corresponding greengo command we create all Greengrass group artifacts in AWS and we place the certificates and config.json for IoT Greengrass Core in ./certs/ and ./config/. In order to perform the deployment to the IoT Core console, we need to move the group certificates and configurations created by greengo to the /greengrass/certs/ and /greengrass/config/ directories on the device. Then we need to download from the web a root CA certificate compatible with the certificates Greengo generated to the /greengrass/certs/ folder. Finally, we can start the IoT Greengrass Core daemon on the edge device so that we are able to navigate back to the greengo folder and deploy. This will deploy the configurations you define in greengo.yaml to the IoT Greengrass Core on the edge device.

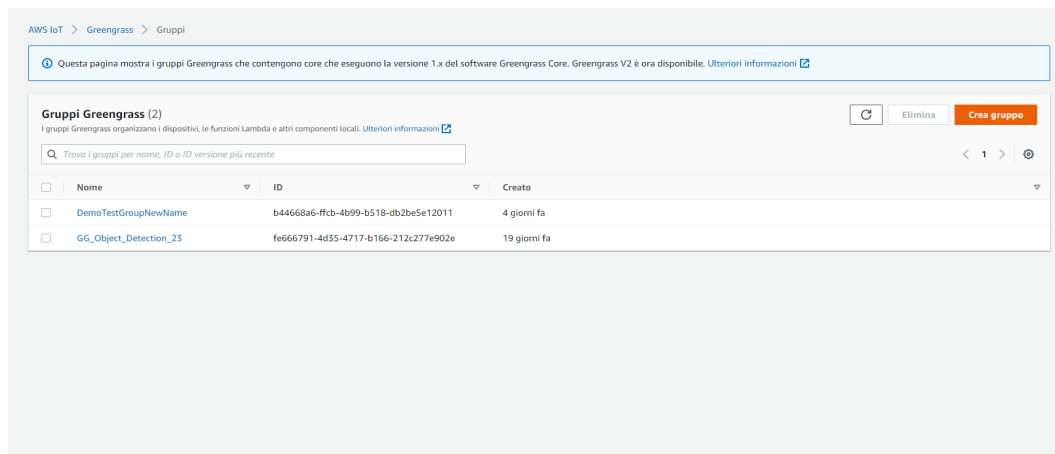


Figure 5.1. GreenGrass Groups

So far we haven't defined any Lambda functions yet in our Greengo configuration, so this deployment just initializes the IoT Greengrass Core and the Machine Learning Resource that takes the deploy_model tar.gz from our S3 Bucket. We will create and add our Lambda Functions from scratch in the next section.

5.3 Creating your inference pipeline in AWS IoT Greengrass Core

We're ready to put it all together now that we've started IoT Greengrass and tested our inference code on the edge device: create a Greengrass Lambda feature that starts recording a video by collecting frames from it, a lambda that executes the inference code inside of Greengrass Core, and a lambda that inserts the value of the prediction into a Database. We'll build the following pipeline to test the Greengrass Lambda IoT functions for inference:

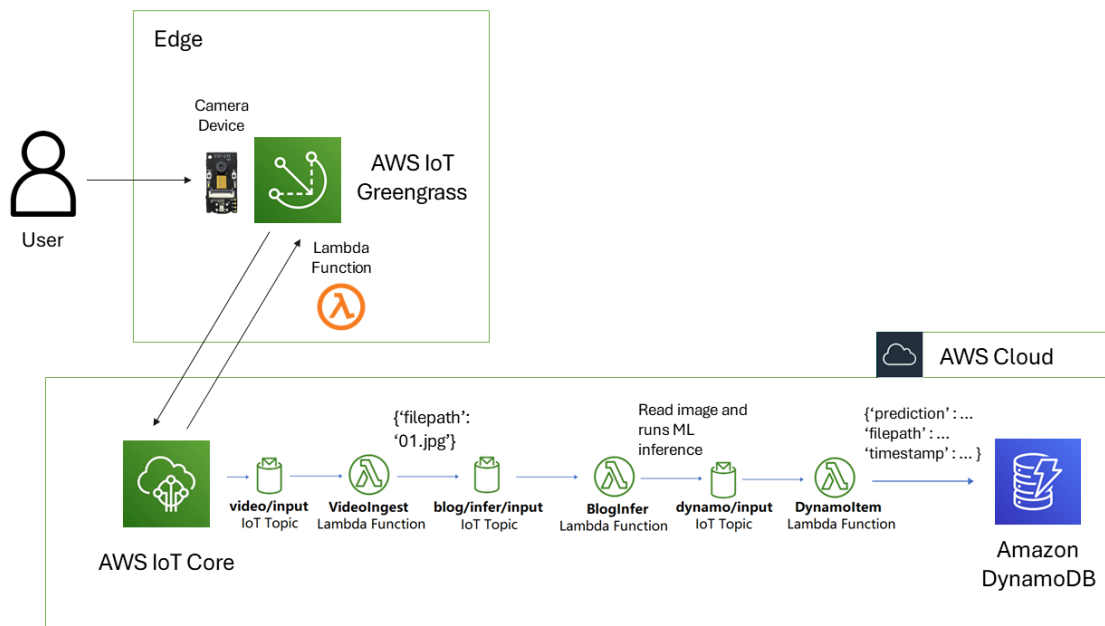


Figure 5.2. Edge Architecture

- When the AWS video/input topic receives a boot message, the Lambda function VideoIngest is called.
- The frame capture code is contained in the Lambda function Videoingest, which starts recording a 10-second video and saves each frame per second in the buffer.
- The AWS IoT topic blog/infer/input will provide the location of the image file on the edge device for the BlogInfer Lambda function to make inference on.
- The object detection inference logic is included in a Lambda function executing in IoT Greengrass Core BlogInfer.
- The IoT dynamo/input topic delivers the Dynamoitem Lambda function the prediction for each frame evaluated in JSON format.

- Finally, the `DynamoItem` method connects to your DynamoDB and creates a new frame entry with the received message's timestamp and a prediction of the gesture made in the image.

5.4 Lambda Functions

AWS Lambda, as previously mentioned, is a serverless computing service that allows you to run your code without having to worry about deploying or managing any servers. AWS Lambda allows you to execute your application or backend service with zero administration. Simply submit your code to Lambda, and it will execute it as well as grow the infrastructure for high availability. As you can see from the pipeline, we'll be using three lambda functions, each of which will serve a distinct purpose.

5.4.1 VideoIngest

Image acquisition will be of interest to the first Lambda. In actuality, we connect to our smart cameras using OpenCV, supplying our camera id as an input resource, which is set to 0 by default (this parameter varies according to the USB port of our device). Sending a message to start the Lambda from our IoT Core on the cloud will start recording a video, which will then be split down into frames (for convenience we have chosen as the dividing threshold every 5 frames). Each frame is saved in a buffer and sent to the following lambda via the MQTT protocol. The message's payload includes the filepath for each frame stored in the buffer, as well as the Lambda topic to which the message should be sent.

```

1 # The cloud system does not need to access the device, so the import
   for greengrasssdk is not present
2 import greengrasssdk
3
4 client = greengrasssdk.client('iot-data')
5
6 ...
7
8 if i % 10 == 0:
9     cv2.imwrite('/buffer/frame'+str(i)+'.jpg',frame)
10
11     """
12
13     The Cloud Local Script needs to store the images in a bucket
   to trigger the main Lambda Function
14
15     client = boto3.client('s3', region_name='eu-west-1')
16     client.upload_file('#PATH FOLDER/frame'+str(i)+'.jpg', '#
   BUCKET ', 'images/frame'+str(i)+'.jpg')
17
18     """
19
20     msg2 = json.dumps({'filepath': '/shared/greengrass/buffer/frame'
   +str(i)+'.jpg'})
21     client.publish(topic='blog/infer/input', payload=msg2)
22     msg = json.dumps('video ingest is sending frame')
```

```

23     msg3 = json.dumps('filepath/shared/greengrass/buffer/frame'+str
24       (i)+'.jpg')
25     client.publish(topic=OUTPUT_TOPIC, payload= msg + ' ' + msg3)
26     logging.info(msg)
27     i+=1
28 cap2.release()
29 cv2.destroyAllWindows()

```

Listing 5.2. VideoIngest Function

The main difference with the cloud counterpart is the fact that the created frames are not uploaded in the S3 bucket, instead they are stored locally and the image local path is provided to the BlogInfer Lambda for the next step.

5.4.2 BlogInfer

The object detection inference algorithm is contained in a lambda function that runs in IoT Greengrass Core BlogInfer. The act of passing live data points through a machine learning algorithm (or "ML model") to compute an output such as a single numerical score is known as machine learning inference. "Operating an ML model" or "bringing an ML model into production" are other terms for this procedure. When a machine learning (ML) model is used in production, it is sometimes referred to as artificial intelligence (AI) since it performs functions akin to human reasoning and analysis. The BlogInfer Lambda function will receive input from the AWS IoT topic blog/infer/input for the location of the picture file on the edge device to conduct inference on. Once the procedure is complete, the prediction output of the BlogInfer Lambda function will be sent via MQTT message to the dynamo/input input topic of the next DynamoItem lambda. The message in JSON format comprises, in addition to the prediction, the timestamp of when the frame was received and the filepath of the same frame, since it is important to know which frame the prediction refers to when saving the message in the database.

```

1  #
2  # Copyright 2010-2017 Amazon.com, Inc. or its affiliates. All Rights
3  #   Reserved.
4  #
5  # Lambda entry point
6  import greengrasssdk
7
8  client = greengrasssdk.client('iot-data')
9
10 OUTPUT_TOPIC = 'dynamo/input'
11
12 # Load the model at startup
13 def initialize(param_path=ML_MODEL_PATH):
14     global model
15     model = MLModel(param_path)
16
17
18 def lambda_handler(event, context):
19     """

```

```

20     While the cloud implementation needs a reference to the uploaded
    S3 object to download the stream of data and subsequently
    reconstruct the image from it, the Edge System only need the path
    to the image stored locally in order to load it and perform
    prediction.

21
22     bucket_name = event['Records'][0]['s3']['bucket']['name']
23     key = event['Records'][0]['s3']['object']['key']
24
25     generated_signed_url = create_presigned_url(bucket_name, key,
26     seven_days_as_seconds, s3_signature['v4'])
27     print(generated_signed_url)
28     image_complete = url_to_image(generated_signed_url)
29
30     """
31
32
33     """
34     Gets called each time the function gets invoked.
35     """
36     ...
37
38     if 'filepath' not in event:
39         msg = 'filepath is not in input event. nothing to do.
    returning.'
40         logging.info(msg)
41         client.publish(topic=OUTPUT_TOPIC, payload=msg)
42         return None
43
44     filepath = event['filepath']
45
46     if not os.path.exists(filepath):
47         msg = 'filepath does not exist. make sure \'{}\'' exists on
    the device'.format(filepath)
48         logging.info(msg)
49         client.publish(topic=OUTPUT_TOPIC, payload=msg)
50         return None
51
52     ...
53
54     prediction = model.predict_from_file(filepath)
55
56     """
57
58     Then, BlogInfer creates a message to send via MQTT to the third
    and final Lambda DynamoItem that will put a new item in our table
    in DynamoDB.
59
60     """
61
62     client.publish(topic=OUTPUT_TOPIC, payload=json.dumps(response))
63     client.publish(topic='blog/infer/output', payload=json.dumps(
    response))

```

Listing 5.3. BlogInfer Function

As you can see, the Edge Implementation is somewhat easier respect to the whole image retrieval since we have no middle step, represented by the upload in S3 in

the Cloud. Moreover, since we are already logged to our account in AWS and we are not accessing it from an external device we don't need any authentication and security measures that are also often cause of unwanted errors.

5.4.3 DynamoItem

DynamoItem via boto3 SDK connects to our Dynamodb, this is because every time an event from the previous Lambda starts the creation of a new item within the Database. This item comprises a key called Gesture ID for each value, which corresponds to the date of the received frame, a second value that provides the frame prediction, and lastly the filepath of the same. Furthermore, the IoT topic dynamo/output will broadcast the DynamoItem Lambda function's prediction output to the AWS IoT message broker in the cloud.

```

1 import greengrasssdk
2 import boto3
3 import logging
4 import os
5 import time
6 import json
7
8 dynamodb = boto3.resource("dynamodb", region_name="eu-west-1")
9 tableName = #DYNAMODBNAME
10 client = greengrasssdk.client('iot-data')
11
12 OUTPUT_TOPIC = 'dynamo/output'
13
14
15 def lambda_handler(event, context):
16
17     logging.info(event["prediction"])
18     logging.info(event["filepath"])
19     global tableName
20
21     """
22
23     global tableName
24     table = dynamodb.Table(tableName)
25     table.put_item(
26         Item={
27             "Gesture ID": str(time.time()),
28             "prediction": predicted_gesture,
29             "filepath": key
30         }
31     )
32
33     """
34
35     table = dynamodb.Table(tableName)
36     table.put_item(
37         Item={
38             "Gesture ID": str(time.time()),
39             "prediction": event["prediction"],
40             "filepath": event["filepath"]
41         }
42     )

```

```

43 client.publish(topic=OUTPUT_TOPIC, payload=json.dumps("publish
44 prediction on DynamoDB" + ':' + event["prediction"] + event["
    filepath"]))
45

```

Listing 5.4. DynamoItem Function

This Lambda is almost identical to the use we make in the previous chapter, the only noticeable difference is the ever-present "greengrasssdk" import necessary for any lambda for Greengrass devices.

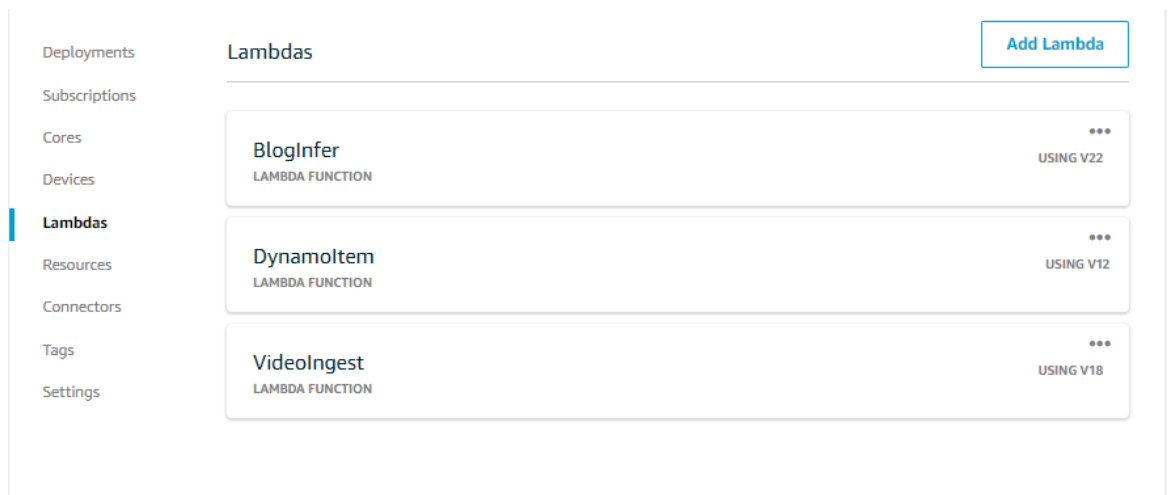


Figure 5.3. Group Lambda Function

5.4.4 Configure Lambda functions

Lambda natively supports a variety of common runtimes, including Python, Node.js, Java, .NET, and others. If you prefer to use any other runtime, such as PHP or Perl, you can use a custom runtime. As we've seen before, we'll use Python 3.8 as our runtime because all of the essential dependencies are already installed. We are now ready to configure your Lambda function for AWS IoT Greengrass. Choose Greengrass, Classic (V1), Groups in the AWS IoT console's navigation pane. Click the previously formed group under Greengrass groups, then choose Lambdas on the group setup page, and then Add Lambda. Look up the name of the Lambda you made in the previous step and copy it. Make the following modifications to the Group-specific Lambda configuration page: Set the Timeout to 60 seconds to ensure that this Lambda function rests for 5 seconds before each call. Select the Lambda lifecycle option, make this function long-lived by allowing read access to the /sys directory and making it run indefinitely. All other fields should be left unchanged, and you should pick Update to save your changes.

The screenshot shows the configuration page for an AWS Lambda function named "VideoIngest". At the top, there is a link to "View function in AWS Lambda". Below this, the "Version 18" is displayed with a "Remove version" link. The "Run as" section has two options: "Use group default (currently: ggc_user/ggc_group)" which is selected, and "Another user ID/group ID". The "Containerization" section has three options: "Use group default (currently: Greengrass container)" which is selected, "Greengrass container (always)", and "No container (always)". The "Memory limit" is set to "900" MB. The "Timeout" is set to "1" Minute. The "Lambda lifecycle" section has two options: "On-demand function" and "Make this function long-lived and keep it running indefinitely" which is selected. The "Read access to /sys directory" has two options: "Disable" and "Enable" which is selected. The "Input payload data type" has two options: "JSON" which is selected, and "Binary".

Figure 5.4. Lambda Configuration

5.5 Install machine learning dependencies on the device

If your device has a GPU, check sure you have the appropriate GPU drivers loaded, such as CUDA. You can still execute the inference if your device only has a CPU, but it will be slower. MXNet is used to develop the SageMaker object detection model. You'll need to install the mxnet library on your device to perform inference on the edge. The MXNet version that corresponds to the CUDA driver may be found in the MXNet install manual. For example, on the Ubuntu instance, we installed CUDA 10.1, thus we installed mxnet-cu101.

```
$ sudo pip install mxnet-cu101 # on a GPU enabled device with CUDA 10.1
```

Install mxnet if you're using a CPU-only device.

```
$ sudo pip install mxnet # on a CPU only device
```

Because OpenCV is a huge requirement, we'll also install it on our device:

```
$ sudo pip2 install opencv-python
```

Be extremely cautious throughout these steps, since the dependencies required for inference execution on the device must be installed within the python directory that you need to use for Lambda execution, in our instance, python 3.8, so all packages must be present.

5.6 Local and Machine Learning Resources

5.6.1 Local Resources

To reduce the expense of data transmission to the cloud, you can use AWS Lambda functions on the device to respond fast to local events, interact with local resources, and process data. But, to allow the Lambda Function to access shared folders or even some of the device's peripherals we need to create through AWS Console the corresponding resources. With AWS IoT Greengrass, you can author AWS Lambda functions and configure connectors in the cloud and deploy them to core devices for local execution. On Greengrass cores running Linux, these locally deployed Lambda functions and connectors can access local resources that are physically present on the Greengrass core device. You can access two types of local resources: volume resources and device resources.

- Volume resources
 - Files or directories on the root file system (except under /sys, /dev, or /var)
- Device resources
 - Files under /dev. Only character devices or block devices under /dev are allowed for device resources.

This is extremely important for our implementation because for the execution of the first Lambda VideoIngest we need to grant permission to access both a shared folder where the function will save and later retrieve the frames and also the device's camera that will record the video of a person making gestures. Let's start with the shared folder: to allow the function to access any file in the device's root file system we need to declare a Local Resource of Volume type. In this case, we create a shared folder with path source /greengrass/shared/buffer and destination path /buffer as shown in the image.

Local resource

Local resources can be used with Greengrass to make filesystem volumes or physical devices accessible to Greengrass Lambdas while offline.

Resource name

Resource type

☐ Device

☒ Volume

Source path

Destination path

Group owner file access permission

An AWS IoT Greengrass Lambda function process normally runs without an OS Group. However, you can give additional file access permissions to the Lambda function process.

☐ No OS group

☒ Automatically add OS group permissions of the Linux group that owns the resource

☐ Specify another OS group to add permission

Choose permissions for this Lambda function

☐ Read-only access

☒ Read and write access

Figure 5.5. Volume Resource

Then we just need to let it automatically add OS group permissions of the Linux group that owns the resource and attach it to the interested lambda. Instead, to give access to the camera, we need to create a different type of Local Resource. In fact, we need to create a Device Resource specifying the path to the camera in the /dev path. /dev is the location of special or device files. It is a very interesting directory that highlights one important aspect of the Linux filesystem - everything is a file or a directory. Inside this path look for a device that might correspond to your camera (typically video0 or video1) and create a resource with that path. Then attach to the interested Lambda, in our case, it's still the VideoIngest function.

Local resource

Local resources can be used with Greengrass to make filesystem volumes or physical devices accessible to Greengrass Lambdas while offline.

Resource name

video0

Resource type

☒ **Device**

☐ **Volume**

Device path

/dev/video0

Group owner file access permission

An AWS IoT Greengrass Lambda function process normally runs without an OS Group. However, you can give additional file access permissions to the Lambda function process.

☐ **No OS group**

☒ **Automatically add OS group permissions of the Linux group that owns the resource**

☐ **Specify another OS group to add permission**

Choose permissions for this Lambda function

☐ **Read-only access**

☒ **Read and write access**

[Cancel](#) [Update](#)

Figure 5.6. Device Resource

5.6.2 Machine Learning Resources

User-defined Lambda functions can access machine learning resources to run local inference on the AWS IoT Greengrass core. A machine learning resource consists of the trained model and other artifacts that are downloaded to the core device. Local Lambda functions can directly engage with machine learning models that are deployed to your Greengrass Core. This is where you specify where to deploy the model locally and how Lambda functions can access it. To allow a Lambda function to access a machine learning resource on the core, you must attach the resource to the Lambda function and define access permissions. The containerization mode of the affiliated (or attached) Lambda function determines how you do this. First, we need to select from our S3 bucket the tar.gz file of our deployable model so that the function has access to the model, then we can specify the path where the model artifacts will be stored. When you need to troubleshoot an ML resource deployment, it's helpful to remember that IoT Greengrass has a containerized architecture and uses filesystem overlay when it deploys resources such as ML model

artifacts. In the example above, even though we configured the ML model artifacts to be extracted to `/ml/od/`, IoT Greengrass actually downloads it to something like `/greengrass/ggc/deployment/mlmodel/<uuid>/`. However, to your IoT Greengrass local Lambda function that you declare to use this artifact, the extracted files will appear to your lambda code to be stored in `/ml/od` due to the filesystem overlay. The OS group and permissions are used by Lambda functions to access downloaded resource artifacts. You must specify an OS group to attach the resource to non-containerized Lambda functions. If this resource is attached to both containerized and non-containerized Lambda functions, containerized Lambda functions should define read or write permissions that are the same or more restrictive. Notice that, if you followed the greengrass example we did earlier, you will already have a machine learning resource created (you can check that in the IoT Core console in the resources tab) identical to the one we just created manually.

Machine learning resource

Local Lambda functions can directly engage with machine learning models that are deployed to your Greengrass Core. This is where you specify where to deploy the model locally and how Lambda functions can access it.

Resource name

MyObjectDetectionModel

Model source

☒ Upload a model in S3 (including models optimized through Deep Learning Compiler)

☐ Use a model trained in AWS SageMaker

Model from S3

deploy_model.tar.gz [Change](#)

Local path

/ml/od/

Identify resource owner and set access permissions

The OS group and permissions are used by Lambda functions to access downloaded resource artifacts. You must specify an OS group to attach the resource to non-containerized Lambda functions. If this resource is attached to both containerized and non-containerized Lambda functions, containerized Lambda functions should define read or write permissions that are the same or more restrictive.

☒ No OS group

☐ Specify OS group and permissions

Lambda function affiliations

BlogInfer	READ AND WRITE ACCESS	Remove	Edit
-----------	-----------------------	------------------------	----------------------

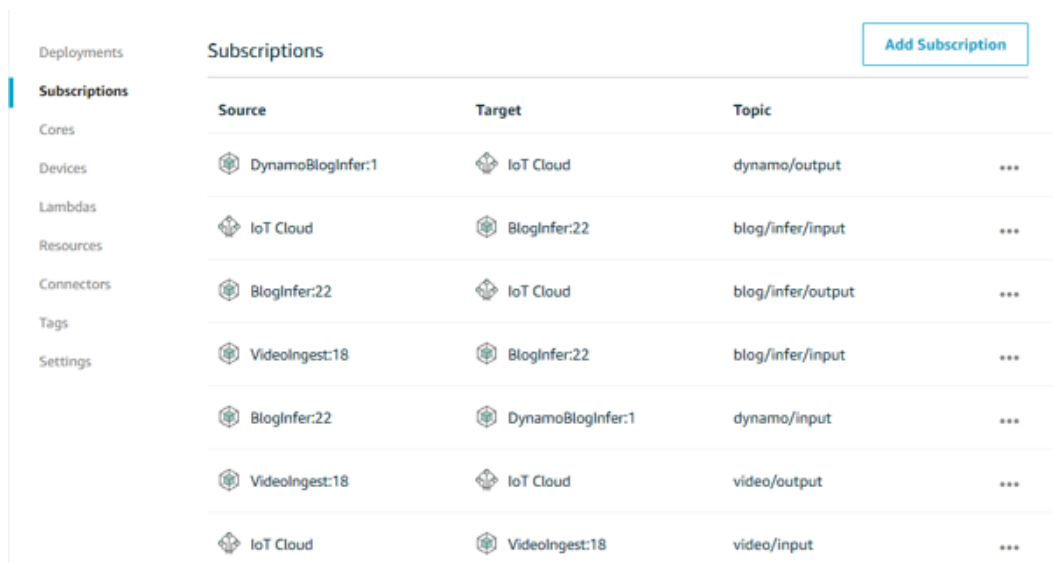
[Select another Lambda function to attach](#) [Select](#)

Figure 5.7. Machine Learning Resource

We just wanted to show you that usually there are different ways to achieve the same objective.

5.7 Subscriptions

A Subscription consists of a source, target, and topic. The source is the originator of the message. The target is the destination of the message. The first step is selecting your source and target. We use subscriptions to make the functions communicate and as a trigger to start the acquisition. We will need to create a connection between each one of the lambda functions and also a link between to IoT Core system to the first lambda to start the whole system. When creating the subscription between two agents, we need to set the topic that must be exactly the one we specified inside the lambda function to make it listen for new messages in that topic. Here you will see the whole picture of the subscriptions for our group.



The screenshot shows the AWS IoT Subscriptions console. On the left is a navigation menu with options: Deployments, Subscriptions (selected), Cores, Devices, Lambdas, Resources, Connectors, Tags, and Settings. The main area is titled 'Subscriptions' and contains a table with columns: Source, Target, Topic, and an ellipsis menu. There are eight subscriptions listed. An 'Add Subscription' button is in the top right corner.

Source	Target	Topic	
DynamoBlogInfer:1	IoT Cloud	dynamo/output	...
IoT Cloud	BlogInfer:22	blog/infer/input	...
BlogInfer:22	IoT Cloud	blog/infer/output	...
VideoIngest:18	BlogInfer:22	blog/infer/input	...
BlogInfer:22	DynamoBlogInfer:1	dynamo/input	...
VideoIngest:18	IoT Cloud	video/output	...
IoT Cloud	VideoIngest:18	video/input	...

Figure 5.8. Group Subscriptions

5.8 MQTT Test Client

You can use the AWS IoT MQTT client to better understand the bidirectional communication that occurs between your devices and AWS IoT over the MQTT protocol. The AWS IoT message broker uses topics to route messages from publishing clients to subscribing clients. The forward-slash (/) is used to separate topics into a hierarchy. Devices publish messages on topics to which AWS IoT or your applications can respond. You can use the AWS IoT MQTT client to subscribe to these topics to see the content of these messages or publish messages on these topics to update the device state. In our case the MQTT Test Client also allows us to start our functions. In fact, just by sending an example message to the topic specified in the subscriptions step we can trigger the VideoIngest function that will subsequently start all the others. Also thanks to the messages sent to the IoT Core target we can see the products of execution of each step of the system.

The screenshot shows the MQTT client interface. The 'Publish to a topic' tab is active. The 'Topic name' is 'video/output'. The 'Message payload' is a JSON object: `{ "message": "Start" }`. The 'Subscriptions' list on the left includes 'video/output', 'dynamo/output', and 'blog/infer/output'. The main panel displays the output for the 'video/output' topic, showing three messages: 'video ingest is sending frame' with filepaths for frame160.jpg, frame150.jpg, and frame140.jpg, each with a timestamp of December 06, 2021, 16:29:08 (UTC+0100), 16:29:07 (UTC+0100), and 16:29:07 (UTC+0100) respectively.

(a) Output of VideoIngest

The screenshot shows the MQTT client interface. The 'Publish to a topic' tab is active. The 'Topic name' is 'video/output'. The 'Message payload' is a JSON object: `{ "message": "Start" }`. The 'Subscriptions' list on the left includes 'video/output', 'dynamo/output', and 'blog/infer/output'. The main panel displays the output for the 'blog/infer/output' topic, showing two messages: 'prediction: "hello"' with timestamps of December 06, 2021, 16:29:42 (UTC+0100) and 16:29:40 (UTC+0100), and filepaths for frame160.jpg and frame150.jpg.

(b) Output of BlogInfer

The screenshot shows the MQTT client interface. The 'Publish to a topic' tab is active. The 'Topic name' is 'video/output'. The 'Message payload' is a JSON object: `{ "message": "Start" }`. The 'Subscriptions' list on the left includes 'video/output', 'dynamo/output', and 'blog/infer/output'. The main panel displays the output for the 'dynamo/output' topic, showing three messages: 'publish prediction on DynamoDB:hello/shared/greengrass/buffer/frame160.jpg', 'publish prediction on DynamoDB:hello/shared/greengrass/buffer/frame150.jpg', and 'publish prediction on DynamoDB:hello/shared/greengrass/buffer/frame140.jpg', each with a timestamp of December 06, 2021, 16:29:42 (UTC+0100), 16:29:40 (UTC+0100), and 16:29:38 (UTC+0100) respectively.

(c) Output of DynamoItem

Figure 5.9. Results of Execution of the Edge Function

Chapter 6

Continuous Integration and Delivery

6.1 Introduction

The methods for building, packaging, and deploying your application are known as continuous integration (CI) and continuous delivery (CD). It essentially lays out certain best practices to follow in order for the code you build to reach your consumers more efficiently and safely, thus generating value. Continuous Integration (CI) is a software development method that ensures that software components communicate with one another. Continuous integration (CI) allows you to integrate code into a single, shared, and easily accessible repository on a regular basis. The ability to release your integrated code into production without the requirement for human intervention is known as continuous delivery (CD). CD lets you take code from the repository and deliver it to production on a regular basis. A continuous integration/continuous delivery (CI/CD) pipeline is a method for delivering a unit of change from development to deployment. It allows application development teams to deploy software fast by automating processes in the software delivery process. One of our main goals for this whole project is to create a system that can be easily reproduced in order to boost its diffusion by anyone that could be interested in what we did. One important step to reach this goal is to find a way to implement our project even in a case where we want to push our system to hundreds of different GreenGrass devices. In a naive world, we would have to create a group, a core, all the lambdas, resources, and subscriptions for every single device we want in the network. This is obviously very tedious and time-consuming, while also introducing more possible points of failure where things might go wrong cause of a human error or any bug in the system. To resolve this problem and also to have a way to achieve CI/CD we take advantage of the Amazon Web Services CodeCommit and CodePipeline. Finally, with AWS CloudFormation we will create what is called a "Stack" that if executed through the mentioned service will create all parts necessary for the Edge System, as you will see later.

6.2 AWS CodeCommit

CodeCommit is a secure, highly scalable, managed source control service that hosts private Git repositories. You can use CodeCommit to store anything from code to binaries. It supports the standard functionality of Git, so it works seamlessly with your existing Git-based tools. We will use CodeCommit to store all three of the Lambdas created for the Edge Environment since they are the parts more prone to changes over time which means we need a way to make changes on a regular basis. The first step when using CodeCommit is to create a repository, you can easily do this through Amazon Console. On creation, we can leave empty said repository since we will synchronize it with a local repository where we will actually work. To set up a local repo on our machine to connect to, we select a directory on our local machine that represents the local repository. We use Git to clone and initialize a copy of your empty CodeCommit repository inside of that directory. Then we specify the user name and email address used to annotate our commits. To clone the CodeCommit repo, we need the Clone URL that can be obtained with the specific button on the main page. There are three different URLs used in different cases:

- Copy the HTTPS URL if you are using either Git credentials with your IAM user or the credential helper included with the AWS CLI.
- Copy the HTTPS (GRC) URL if you are using the `git-remote-codecommit` command on your local computer.
- Copy the SSH URL if you are using an SSH public/private key pair with your IAM user.

On the command prompt, clone the repository with the `git clone` command and provide the clone URL you copied earlier. For example, we created a repository for our first Lambda `VideoIngest` and want it in a local repo inside a folder called `VideoLambda`:

```
git clone https://git-codecommit.eu-west-1.-  
-amazonaws.com/v1/repos/LambdaVideo VideoLambda
```

We can now drag and drop all the files needed for our Lambda (these are exactly the same to the one used in our Edge Implementation) inside the local folder and commit to the CodeCommit Repo after configuring User name and password like this.

```
git config --local user.name "your-user-name"  
git config --local user.email your-email-address  
  
git add *  
  
git commit -m "Added VideoIngest Lambda"  
  
git push -u origin main
```

This will push all the files we moved into our local repo to the CodeCommit repository that will be the place used to store and change the code to keep it up to date over time after the final implementation of the project.

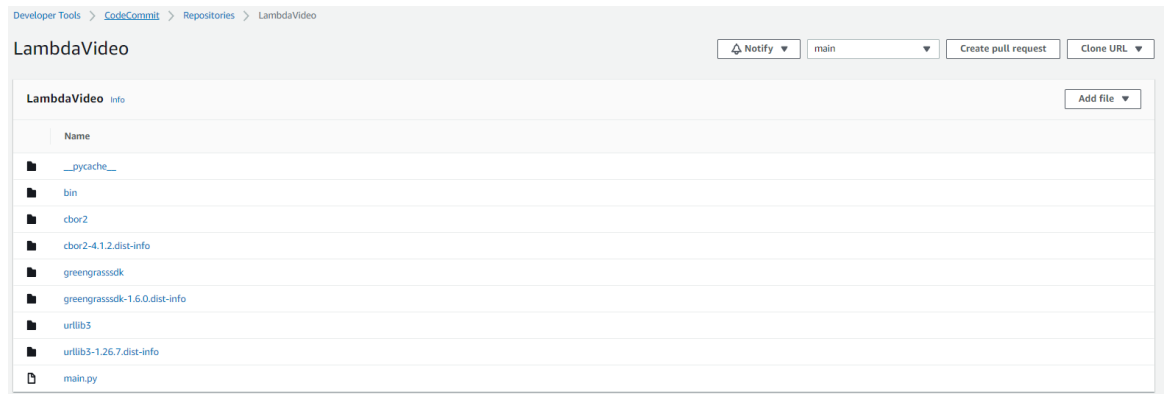


Figure 6.1. VideoIngest Repository

Once we repeat this step for each Lambda we will have three different repositories and changes in any of these will trigger the AWS CodePipeline that we will create in the next section.

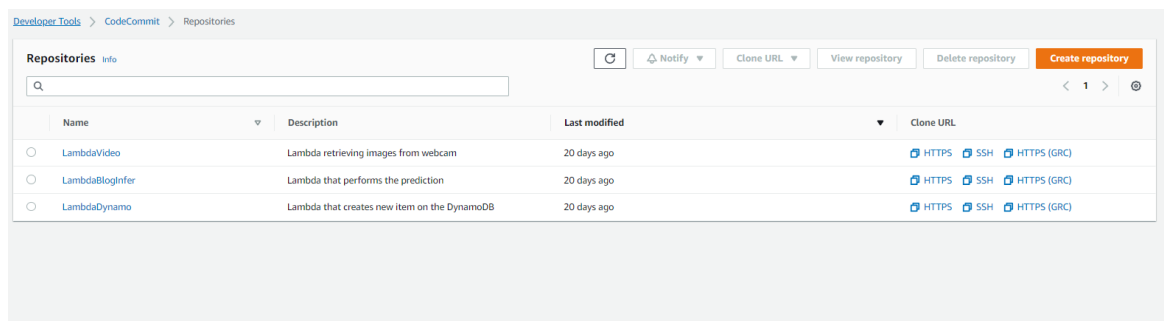


Figure 6.2. CodeCommit Repositories

6.3 AWS CodePipeline

You'll learn how to develop an automated software release pipeline that releases a live sample app in this article. AWS CodePipeline, a service that creates, tests, and deploys your code every time it changes, will be used to build the pipeline. The source location for the sample app's code will be your GitHub account, an Amazon Simple Storage Service (S3) bucket, or an AWS CodeCommit repository. When your pipeline is finished, it will be able to detect changes to the source repository containing the sample app and update your live sample app accordingly. Continuous deployment automates the software release process by allowing you to push modifications to a production environment without requiring explicit permission

from a developer. In this section, you create a CodePipeline automated release process with the following actions: Log in to the AWS Management Console and go to <http://console.aws.amazon.com/codesuite/codepipeline/home> to access the CodePipeline console. Choose Create pipeline from the Welcome page, Getting started page or Pipelines page. In the first step, select Pipeline settings and input the name of your pipeline in Pipeline name. Perform one of the following tasks in your Service role: to authorize CodePipeline to establish a new service role in IAM, choose New service role, or to utilize a service role that has already been created in IAM, choose Existing service role. After that, pick Next and leave the Advanced options at their defaults. In the second step, add a source stage and choose AWS CodeCommit as the source provider. Choose the name of the CodeCommit repository you generated in the previous step under Repository name. Choose the name of the branch that contains your most recent code change in Branch name, only main is available until you build a distinct branch on your own. The Amazon CloudWatch Events rule to be established for this pipeline is presented after you pick the repository name and branch, and then you choose Next. Next, add a build stage, choose Skip construction stage, and accept the warning notice by selecting Skip once again. Then Add a stage for deployment:

- Select Amazon S3 from the Deploy provider drop-down menu.
- Fill in the name of your public bucket in Bucket.
- If you do not choose Extract file before deploy, the deployment will fail. This is because the AWS CodeCommit activity in your pipeline zips source artifacts, and your file is a ZIP file. Deployment path is displayed when Extract file before deploy is selected. Put the name of the path you'd want to utilize here. This creates a folder structure in Amazon S3 to which the files are extracted, but we can leave this field blank.

Add source stage [Info](#)

Source

Source provider
This is where you stored your input artifacts for your pipeline. Choose the provider and then provide the connection details.

AWS CodeCommit ▼

Repository name
Choose a repository that you have already created where you have pushed your source code.

Q LambdaDynamo X

Branch name
Choose a branch of the repository

Q main X

Change detection options
Choose a detection mode to automatically start your pipeline when a change occurs in the source code.

☒ **Amazon CloudWatch Events (recommended)**
Use Amazon CloudWatch Events to automatically start my pipeline when a change occurs

☐ **AWS CodePipeline**
Use AWS CodePipeline to check periodically for changes

Output artifact format
Choose the output artifact format.

☒ **CodePipeline default**
AWS CodePipeline uses the default zip format for artifacts in the pipeline. Does not include git metadata about the repository.

☐ **Full clone**
AWS CodePipeline passes metadata about the repository that allows subsequent actions to do a full git clone. Only supported for AWS CodeBuild actions.

[Cancel](#) [Previous](#) [Next](#)

Figure 6.3. Pipeline Source Setting

Now, all we have to do is review the information and select Create pipeline. We saw in the previous paragraph that we utilized three distinct Codecommit repositories for each lambda function for more effective lambda management, therefore we'll establish the same amount of pipelines repeating the same procedures.

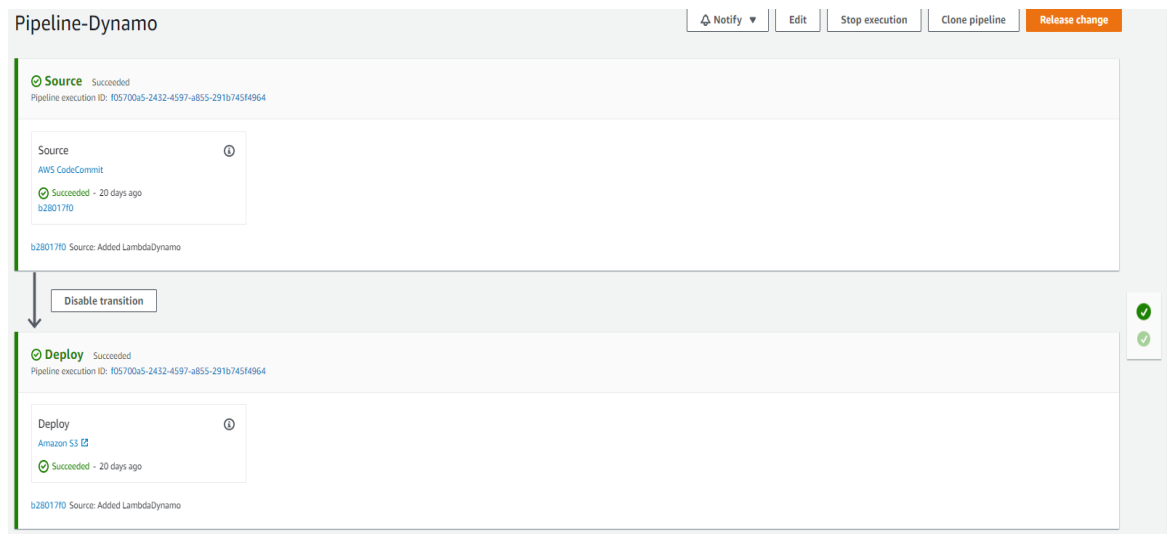


Figure 6.4. Pipeline Example

We try to make a modification to your source files and then push the change to your repository when we want to make sure the pipeline is working properly. This triggers your pipeline to run, and after your pipeline runs successfully, open the Amazon S3 console and verify that your files appear in your public bucket as shown:

DynamoItem.zip
 LambdaBlogInfer.zip
 LambdaVideoIngest.zip

These three zip files containing the lambda functions represent the most updated version of each function. This means that a newly created group can and must use these as their source files to work properly. In fact, in the next section, we will show how to create a Cloud Formation Stack that, when creating the Lambda Function from YAML, will refer to these archives that can be publicly available inside the enterprise working on the project. This allows for a small group in charge of maintaining the code to make proper changes while anyone can deploy the stack and create a new working system using that same code.

6.4 AWS CloudFormation and Stack

6.4.1 Introduction

AWS CloudFormation is an AWS resource creation service that allows developers to generate resources in an orderly and predictable manner. Text files in the JavaScript Object Notation (JSON) or Yet Another Markup Language (YAML) format are used to store resources. The templates have a specific syntax and structure that is determined by the resources being produced and maintained. You write your resources in JSON or YAML with any code editor, save them to a version control system, and CloudFormation constructs the services you specify in a secure, repeatable manner. As a stack, a CloudFormation template is deployed into the AWS environment. The AWS Management Console, AWS Command Line Interface,

and AWS CloudFormation APIs may all be used to manage stacks. If you need to make changes to a stack's running resources, you must update it. You can create a change set, which is a summary of your proposed modifications, before making changes to your resources. Before adopting changes, you may use change sets to evaluate how they will affect your operating resources, especially key resources. You may construct and update a complete environment with a single template, or you can use distinct templates to handle multiple levels inside an environment. This allows for the modularization of templates while also adding a degree of control that many businesses value. When you build or change a stack in the console, events appear that reflect the configuration's state. When an error occurs, the stack is rolled back to its prior state by default. The Amazon Simple Notification Service (Amazon SNS) sends out event alerts. You may, for example, utilize Amazon SNS to track stack generation and deletion progress through email and programmatically interact with other operations. When you configure an AWS stack, you may use AWS CloudFormation to organize and deploy a set of AWS resources, as well as define any dependencies or pass in additional parameters.

You can work with a wide range of AWS services using CloudFormation templates, including Amazon S3, Amazon DynamoDB, Amazon EC2, and Amazon VPC. See the AWS resource and property types reference for the most up-to-date list of supported resources.

6.4.2 Design and implementation of the template in YAML language

In this section, we'll look at how to create a template that defines a group for Edge development that is equal to the group set up in previous chapters, a core and subscriptions for the group, the necessary resources, and three lambda, one for frame acquisition, one for inference, and one for storing the prediction in DynamoDB.

Parameters

The template provides options that allow you to define the core's certificate ARNs. It references IDs, ARNs, and other characteristics necessary to generate Greengrass resources using the Ref and GetAtt intrinsic methods. The parameters object in a template is where you define parameters. A parameter has a set of properties that specify its value as well as limitations on it. The Type element is the only one that must be specified, and it might be String, Number, or an AWS-specific type.

```
1 AWSTemplateFormatVersion: 2010-09-09
2 Parameters:
3   CoreCertificateArn:
4     Type: String
```

Listing 6.1. Parameters

Resources

Declare an AWS IoT object using the AWS::IoT::Thing resource, which in this case is our Core. Then we make two resources: a core definition and a core definition

version that includes a core definition. The `CoreDefinitionId` for the core definition version is provided by the template using the `Ref` function (which associates the version with the core definition). The core definition name, as well as the ID, Thing ARN, and certificate ARN to use for the core are represented as arguments in the template.

```

1 ObjectDetectionCore:
2   Type: 'AWS::IoT::Thing'
3   Properties:
4     ThingName: ObjectDetectionCore
5   TestCoreDefinition:
6     Type: 'AWS::Greengrass::CoreDefinition'
7     Properties:
8       Name: DemoTestCoreDefinition
9   TestCoreDefinitionVersion:
10    Type: 'AWS::Greengrass::CoreDefinitionVersion'
11    Properties:
12      CoreDefinitionId: !Ref TestCoreDefinition
13      Cores:
14        - Id: ObjectDetectionCore
15          CertificateArn: !Ref CoreCertificateArn
16          SyncShadow: 'false'
17          ThingArn: !Join
18            - ':'
19            - - 'arn:aws:iot'
20              - !Ref 'AWS::Region'
21              - !Ref 'AWS::AccountId'
22              - thing/ObjectDetectionCore

```

Listing 6.2. Core

A Lambda function is created using the `AWS::Lambda::Function` resource. A deployment package and an execution role are required to construct a function. Your function code is contained in the deployment package, which is a .zip file archive or container image. The function's execution role allows it to leverage AWS services like Amazon CloudWatch Logs for log streaming. If the deployment package is a container image, the package type is `Image`. The code attribute for a container image must provide the URI of a container image in the Amazon ECR registry. The handler and runtime attributes do not need to be specified. If the deployment package is a .zip file archive, the package type is set to `Zip`. The code attribute of a .zip file archive defines the location of the .zip file. The handler and runtime attributes must also be specified. We can refer to the .zip file accessing our bucket, with `S3key` set to `lambda path`, and keep the function continually updated because we already built a CI/CD pipeline through AWS Code Pipeline. Lambda allows a function to continue for a certain period of time (in seconds) before stopping it. The default value is 3 seconds, and the highest allowable value is 900 seconds, however, we standardized it to 1 minute in our situation.

```

1 BlogInferCloudFormation:
2   Type: AWS::Lambda::Function
3   Properties:
4     Code:

```



```

5      S3Bucket: bucketmigliore
6      S3Key: lambdas/LambdaBlogInfer.zip
7      Runtime: python3.7
8      Role: arn:aws:iam::036027030307:role/
GG_Object_Detection_23_Lambda_Role
9      Handler: main.lambda_handler
10     FunctionName: BlogInferCloudFormation
11     MemorySize: 1024
12     Timeout: 60
13 VideoIngestCloudFormation:
14   Type: AWS::Lambda::Function
15   Properties:
16     Code:
17       S3Bucket: bucketmigliore
18       S3Key: lambdas/LambdaVideoIngest.zip
19       Runtime: python3.7
20       Role: arn:aws:iam::036027030307:role/
GG_Object_Detection_23_Lambda_Role
21     Handler: main.lambda_handler
22     FunctionName: VideoIngestCloudFormation
23     MemorySize: 1024
24     Timeout: 60
25 DynamoBlogInferCloudFormation:
26   Type: AWS::Lambda::Function
27   Properties:
28     Code:
29       S3Bucket: bucketmigliore
30       S3Key: lambdas/DynamoBlogInfer.zip
31       Runtime: python3.7
32       Role: arn:aws:iam::036027030307:role/
GG_Object_Detection_23_Lambda_Role
33     Handler: main.lambda_handler
34     FunctionName: DynamoCloudFormation
35     MemorySize: 1024
36     Timeout: 60

```

Listing 6.3. Lambda Functions

A DynamoDB table is created with the `AWS::DynamoDB::Table` resource. The attributes that make up a table's primary key, or the key attributes of an index, are specified by a key schema. A single attribute of the primary key is represented by a `KeySchemaElement`. A basic primary key, for example, might be represented by a single `KeySchemaElement` (for the partition key). A scalar, top-level attribute must be a `KeySchemaElement` (not a nested attribute). String, Number, or Binary must be the data type. Within a List or a Map, the attribute cannot be nested. In our case we create a Table with a single primary key called Gesture ID.

```

1 myDynamoDBTable:
2   Type: AWS::DynamoDB::Table
3   Properties:
4     KeySchema:
5       -
6         AttributeName: "GestureID"
7         KeyType: "HASH"
8     AttributeDefinitions:
9       -
10      AttributeName: "GestureID"

```

```

11     AttributeType: "S"
12     ProvisionedThroughput:
13       ReadCapacityUnits: "5"
14       WriteCapacityUnits: "5"
15     TableName: "DynamoDB"

```

Listing 6.4. Dynamo Table

For AWS IoT Greengrass, the `FunctionDefinitionVersion` resource indicates a function definition version. A list of functions is included in the function definition version. The resources for function definition and function definition version are defined in the following snippet. The function definition version includes a function as well as a reference to the function definition. In this example, we refer to the lambda created in the previous step through `!GetAtt`. We must distribute the appropriate resources for each lambda function, keeping in mind that `VideoIngest` receives the resources to record frames from the device and put them in a buffer, whereas `BlogInfer` holds the model for inference.

```

1  TestFunctionDefinition:
2    Type: 'AWS::Greengrass::FunctionDefinition'
3    Properties:
4      Name: DemoTestFunctionDefinition
5  TestFunctionDefinitionVersion:
6    Type: 'AWS::Greengrass::FunctionDefinitionVersion'
7    Properties:
8      FunctionDefinitionId: !GetAtt
9        - TestFunctionDefinition
10       - Id
11    DefaultConfig:
12      Execution:
13        IsolationMode: GreengrassContainer
14    Functions:
15      - Id: VideoIngest
16        FunctionArn: !Join
17          - ':'
18          - !GetAtt 'VideoIngestCloudFormation.Arn'
19          - '1'
20      FunctionConfiguration:
21        Pinned: 'true'
22        MemorySize: '900000'
23        Timeout: '60'
24        EncodingType: json
25        Environment:
26          ResourceAccessPolicies:
27            - ResourceId: ResourceId1
28              Permission: rw
29            - ResourceId: ResourceId2
30              Permission: rw
31          AccessSysfs: 'true'
32          Execution:
33            IsolationMode: GreengrassContainer
34      - Id: BlogInfer
35        FunctionArn: !Join
36          - ':'
37          - !GetAtt 'BlogInferCloudFormation.Arn'
38          - '1'

```

```

39     FunctionConfiguration:
40       Pinned: 'true'
41       MemorySize: '900000'
42       Timeout: '60'
43       EncodingType: json
44       Environment:
45         ResourceAccessPolicies:
46           - ResourceId: ResourceId3
47             Permission: rw
48         AccessSysfs: 'true'
49         Execution:
50           IsolationMode: GreengrassContainer
51 - Id: DynamoBlogInfer
52   FunctionArn: !Join
53     - ':'
54     - - !GetAtt 'DynamoBlogInferCloudFormation.Arn'
55       - '1'
56   FunctionConfiguration:
57     Pinned: 'true'
58     MemorySize: '900000'
59     Timeout: '60'
60     EncodingType: json
61     Environment:
62       AccessSysfs: 'true'
63       Execution:
64         IsolationMode: GreengrassContainer

```

Listing 6.5. Functions Definition

A resource definition version for AWS IoT Greengrass is represented by the AWS Greengrass ResourceDefinitionVersion resource. A list of resources is included in the resource definition version. You must give the ID of the resource definition you wish to associate with the version when creating a resource definition version. To learn how to create a resource definition, go [here](#). You must add a resource definition version to your group version when you generate a resource definition version that contains the resources you intend to deploy. We'll utilize two local device resources as resources: a local volume resource that represents a file or directory on the root file system, and a local device resource that represents a file under /dev. While we utilize the model as a new resource from our Bucket S3.

```

1 TestResourceDefinition:
2   Type: 'AWS::Greengrass::ResourceDefinition'
3   Properties:
4     Name: DemoTestResourceDefinition
5   TestResourceDefinitionVersion:
6     Type: 'AWS::Greengrass::ResourceDefinitionVersion'
7     Properties:
8       ResourceDefinitionId: !Ref TestResourceDefinition
9     Resources:
10      - Id: ResourceId1
11        Name: LocalDeviceResource
12        ResourceDataContainer:
13          LocalDeviceResourceData:
14            SourcePath: /dev/video0
15            GroupOwnerSetting:
16              AutoAddGroupOwner: 'true'
17      - Id: ResourceId2

```

```

18     Name: LocalVolumeResourceData
19     ResourceDataContainer:
20       LocalVolumeResourceData:
21         SourcePath: /shared/greengrass/buffer
22         DestinationPath: /buffer
23         GroupOwnerSetting:
24           AutoAddGroupOwner: 'true'
25   - Id: ResourceId3
26     Name: S3MachineLearningModelResourceData
27     ResourceDataContainer:
28       S3MachineLearningModelResourceData:
29         S3Uri: 's3://bucketmigliore/deploy_model.tar.gz'
30         DestinationPath: /ml/od/

```

Listing 6.6. Resources Definition

MQTT messages are exchanged between devices, functions, and connectors in the group, as well as with AWS IoT or the local shadow service, using subscriptions. A subscription specifies a message source, message destination, and topic (or subject) for routing messages from the source to the destination. A subscription specifies the flow of messages from the source to the target in one direction. Set up two subscriptions, one for each side, for two-way conversation. The themes will be the same as those developed for the edge architecture part.

```

1 TestSubscriptionDefinition:
2   Type: 'AWS::Greengrass::SubscriptionDefinition'
3   Properties:
4     Name: DemoTestSubscriptionDefinition
5 TestSubscriptionDefinitionVersion:
6   Type: 'AWS::Greengrass::SubscriptionDefinitionVersion'
7   Properties:
8     SubscriptionDefinitionId: !Ref TestSubscriptionDefinition
9     Subscriptions:
10      - Id: VideoIngest1
11        Source: cloud
12        Subject: video/input
13        Target: !Join
14          - ':'
15          - !GetAtt 'VideoIngestCloudFormation.Arn'
16          - '1'
17      - Id: VideoIngest2
18        Source: !Join
19          - ':'
20          - !GetAtt 'VideoIngestCloudFormation.Arn'
21          - '1'
22        Subject: video/output
23        Target: cloud
24      - Id: BlogInfer1
25        Source: cloud
26        Subject: blog/infer/input
27        Target: !Join
28          - ':'
29          - !GetAtt 'BlogInferCloudFormation.Arn'
30          - '1'
31      - Id: BlogInfer2
32        Source: !Join
33          - ':'

```

```

34         - - !GetAtt 'BlogInferCloudFormation.Arn'
35         - '1'
36         Subject: blog/infer/output
37         Target: cloud
38     - Id: Dynamo1
39       Source: !Join
40         - ':'
41         - - !GetAtt 'DynamoBlogInferCloudFormation.Arn'
42         - '1'
43       Subject: dynamo/output
44       Target: cloud
45     - Id: VideoBlog
46       Source: !Join
47         - ':'
48         - - !GetAtt 'VideoIngestCloudFormation.Arn'
49         - '1'
50       Subject: blog/infer/input
51       Target: !Join
52         - ':'
53         - - !GetAtt 'BlogInferCloudFormation.Arn'
54         - '1'
55     - Id: BlogDynamo
56       Source: !Join
57         - ':'
58         - - !GetAtt 'BlogInferCloudFormation.Arn'
59         - '1'
60       Subject: dynamo/input
61       Target: !Join
62         - ':'
63         - - !GetAtt 'DynamoBlogInferCloudFormation.Arn'
64         - '1'

```

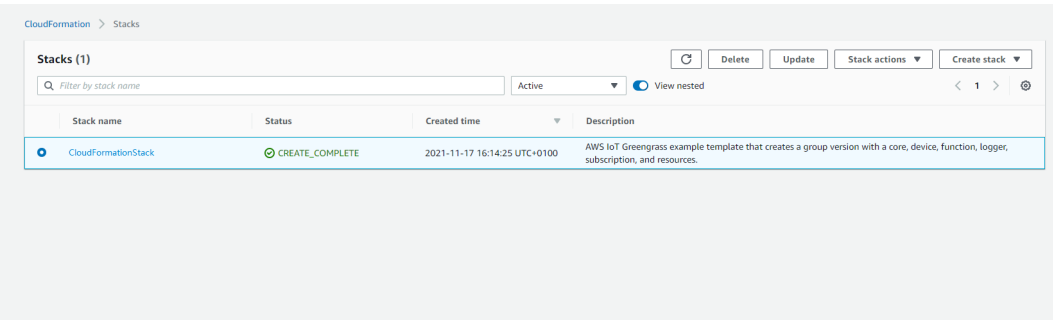
Listing 6.7. Subscriptions Definition

In AWS IoT Greengrass, the AWS Greengrass Group resource represents a group. Groups are used in the AWS IoT Greengrass API to arrange your group versions. Multiple group versions can be referenced by a single group. A group must be assigned to all group versions. A group version refers to the components you intend to deploy to a Greengrass core device, such as the device definition version, subscription definition version, and other version classes. A group version must relate to a core definition version that contains one core in order to be deployed.

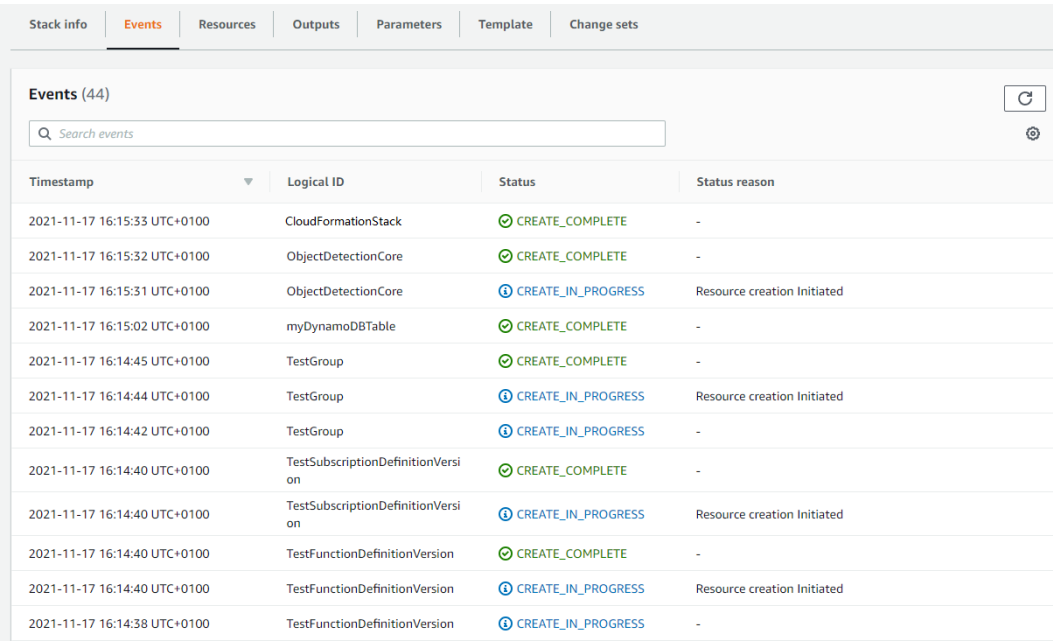
```

1 TestGroup:
2   Type: 'AWS::Greengrass::Group'
3   Properties:
4     Name: DemoTestGroupNewName
5     RoleArn: 'arn:aws:iam::036027030307:role/
6       GreengrassServiceRole_IgHUL'
7     InitialVersion:
8       CoreDefinitionVersionArn: !Ref TestCoreDefinitionVersion
9       FunctionDefinitionVersionArn: !Ref
10      TestFunctionDefinitionVersion
11      SubscriptionDefinitionVersionArn: !Ref
12      TestSubscriptionDefinitionVersion
13      ResourceDefinitionVersionArn: !Ref

```

(a) Stack Creation



(b) Events of creation with Stack

Figure 6.6. Stack Creation and Execution



Chapter 7

Conclusions and Future Works

During this whole thesis, we have shown you how to design, develop and pilot Continuous Integration and Continuous Delivery Pipelines as a way to transform a Cloud Implementation of a Machine Learning Model to an Edge Environment to exploit its advantages thanks to the latest advancements in technologies and devices. Also, we wanted to show how simple it is for anyone wishing to go one step further when talking about inclusion in the everyday life to improve many people's life. We showed different methods of implementation and all the services that as of now represent the go-to for most of the task they were created for and also provided you the code we used to prompt anyone with close to no knowledge about code and scripts to make his attempt in the creation of a system analogue to ours. On the other hand, we also highlighted the main differences of the two macro-sections of our work. Nevertheless, Edge and Cloud Environments, while they obviously present differences in implementation and execution, should not be treated as opposite worlds that are exclusive in a project; in fact, we showed in our case studies how they can be used to work together to comply to different needs that may arise in a real world scenario. In the implementation of the Edge and Cloud environment for our project the differences showed themselves very clearly since, after all, they are very different ways of building a project. In our study, we denoted that the Edge Implementation is generally faster during execution and limit the use of bandwidth given the fact that the computation is done on the device and only its output is sent through the internet connection. In this implementation, while you can use several services and tools available on the internet, you still need to have a physical device suited for the task. This implies some cost that might not be for everyone; on the other hand, keeping the computation closer to the edge allows to manipulate and change the data before sending it through the internet connection. The Cloud implementation instead has no need for you to prepare a group of devices since all of our project logic is uploaded to a virtual machine hosted by some service. This lets you avoid the sunk cost of buying the boards but brings more expenses on the hosting. Moreover, having to upload each image in order to perform the prediction takes a considerable amount of time and bandwidth. Talking about Total Execution Time, for the Edge System it is the time needed for the acquisition and detection of gestures on the edge device, plus the time elapsed for the gestures to be sent via API to the DynamoDB and finally the time needed to store those gestures

on the database. The results are shown here:

```
10 seconds and 270 milliseconds time between starting point and last frame
1 second and 560 milliseconds between getting first frame and last frame
5 seconds and 693 milliseconds between first frame and first prediction
47 seconds and 643 milliseconds between first prediction and last prediction
326 milliseconds between first prediction and first put table in DynamoDB
47 seconds and 392 milliseconds between first put in DynamoDB and last put
```

When we talk about the Cloud implementation instead, things are similar but with some important differences. In fact, in the Total Execution Time for the Cloud System it is still present a step for image acquisition, but the detection is not performed on the device anymore, instead we send the whole image via API call to predict the gesture in the Lambda function. Finally we compute the time for storing said gesture in our Database.

```
1 minute 47 seconds and 940 from first upload on S3 and last put on DynamoDB
32 seconds and 552 from first to last put on DynamoDB
3 seconds and 228 from first to last upload on S3
```

With this said though, even when using apposite services and tools, its setting is somewhat more difficult, specially when a lot of devices need to be added to the group. For this reason, one of the main goal of this paper was to study, and eventually provide, a pipeline that would allow users to scale existing infrastructures or building one anew without great efforts and excessive costs. We addressed this issue by creating, with the help of several Amazon Services, a repository where the lambdas code can be stored and updated for all the existing devices and a CloudFormation Stack that when executed will create a new Edge system inside the AWS console. Same goes for the ML model that will be stored inside a bucket and only the latest one will be retrieved. Cost-wise, Edge Implementation present the need of acquiring all the different smart devices that, depending on the number of them, might be expensive. The Cloud Implementation has an easier set up and does not need a smart board for its execution, but at the cost of a "slower" prediction and the increased costs that present themselves cause of the cloud service used to host our function. Moreover for the costs we created a case study in order to have an estimation of the expenses to give a clearer idea of what to expect once anyone adopts this project and wants to use it. In a case we may have an entrepreneur that, caring deeply for the inclusion of all her customers, decided to implement our project in her smart clothing store. In this case, being her store relatively small and the percentage of her hearing-impaired customer probably on a single digit, she does not need a lot of edge devices in order to satisfyingly implement our project. We think that even a single device should be enough. This obviously means that the usage of each service will be limited, and possibly even covered by the free tier offered by AWS. Even if this wasn't the case, considering the she does not need to train her own model, she doesn't need to have a cloud infrastructure and considering that the Database usage will be pretty low, the total costs that she would have to pay in a 1-year usage of our project are about 30 US Dollars, low enough to be affordable by a single private individual. The only upfront cost not considered in the

pricing calculator is the price that needs to be paid to acquire a camera equipped device. This is a *una tantum* cost that we approximated to 30€. Moreover, while as we showed the source code for Edge devices can be modified without much effort creating a new zip file containing the Lambda, bringing any update or change to the only function for the Cloud system is not that simple. In fact, the Cloud Function operates as an image container thanks to the help of Docker CLI; what this means is that any change, debugging or update will need you to create, build and push a new docker image for the Lambda service to retrieve.

7.1 Future works

In the future, more in-depth studies might be conducted in order to improve our model that, as of right now, is only able to recognize static gestures such as the ISL alphabet whereas the "real" sign language is not limited to static hand gestures, but is actually much more composed of movements of the hands and arms. In order to reach this degree of recognition a more complex analysis of the movements composing the gesture would be needed. Given the amount of data to be analyzed, this would almost surely result in a big improvement in the model, but it would also result in large increase in costs including, but not limited to, the training of the model. Finally, we have already planned to integrate an LCD and a speaker in the embedded system for future deployments. By combining the two devices, the end user will be able to see the system's reaction in real time and interact with it directly. We built the entire project on a virtual Linux machine with camera, but we want to use an AIoT-compatible board in the future. The ESP-EYE is an image recognition and audio processing development board that may be utilized in a variety of AIoT applications. It has an ESP32 processor, as well as a 2-megapixel camera and a microphone. With an 8 Mbyte PSRAM and a 4 Mbyte flash, the ESP-EYE has plenty of storage. It has a Micro-USB connector for debugging and Wi-Fi for image transfer and it also provides out-of-the-box connectivity with AWS-IoT and other AWS services.

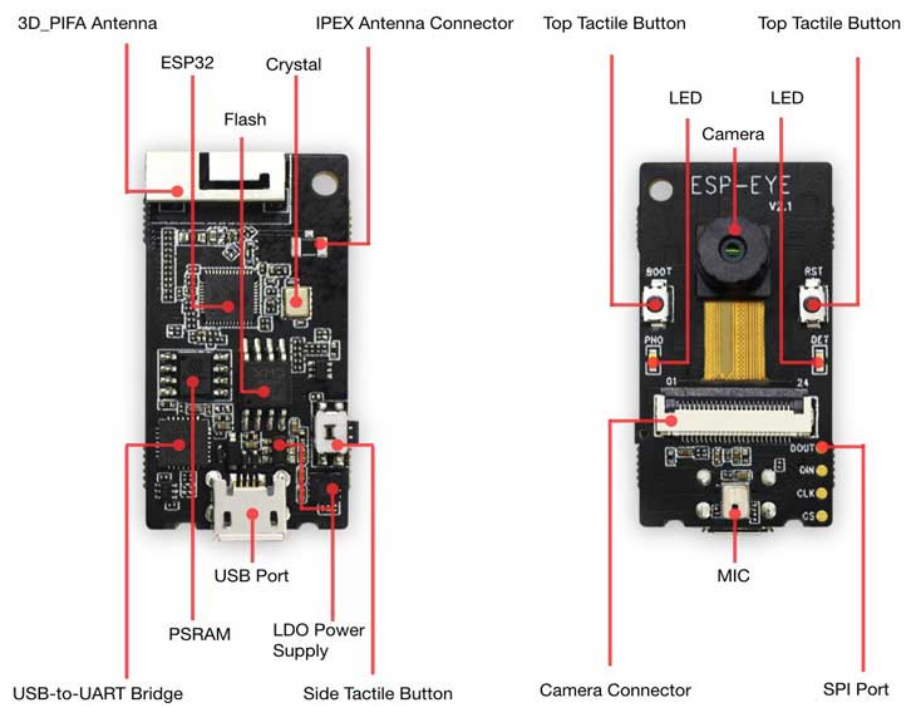


Figure 7.1. ESP-EYE Board Detail

List of Figures

2.1	Cloud-Centric Architecture	6
2.2	Edge-Centric Architecture	7
3.1	International Sign Language	18
3.2	GroundTruth Task Types	21
3.3	Output JSON File	23
3.4	SageMaker Training Jobs	24
3.5	Report Visualization	26
4.1	Cloud Architecture	27
4.2	Lambda Function Folder	34
4.3	Pushing Container Image in ECR	34
4.4	Creating Lambda with Container Image	35
4.5	Trigger Configuration	36
4.6	Results of Execution of the Cloud Function	37
5.1	GreenGrass Groups	40
5.2	Edge Architecture	41
5.3	Group Lambda Function	46
5.4	Lambda Configuration	47
5.5	Volume Resource	49
5.6	Device Resource	50
5.7	Machine Learning Resource	51
5.8	Group Subscriptions	52
5.9	Results of Execution of the Edge Function	53
6.1	VideoIngest Repository	57
6.2	CodeCommit Repositories	57
6.3	Pipeline Source Setting	59
6.4	Pipeline Example	60
6.5	CloudFormation Stack Design	68
6.6	Stack Creation and Execution	69
6.7	Pipeline of Stack Events	70
7.1	ESP-EYE Board Detail	74

Listings

4.1	Video Recording Function	28
4.2	Container Image Main File	30
4.3	Dockerfile	32
5.1	Greengo YAML	40
5.2	VideoIngest Function	42
5.3	BlogInfer Function	43
5.4	DynamoItem Function	45
6.1	Parameters	61
6.2	Core	62
6.3	Lambda Functions	62
6.4	Dynamo Table	63
6.5	Functions Definition	64
6.6	Resources Definition	65
6.7	Subscriptions Definition	66
6.8	Greengrass Group	67

Bibliography

- [1] International Sign | *European Union Of the Deaf*. (2012, May 25). URL: <https://www.eud.eu/about-us/eud-position-paper/international-sign-guidelines/>.
- [2] Oracle | *What Is the Internet Of Things (IoT)?*. URL: <https://www.oracle.com/internet-of-things/what-is-iot/>.
- [3] Introduction to IoT | *What is IoT*. URL: <https://www.leverage.com/iot-ebook/what-is-iot>.
- [4] Introduction To Internet Of Things (IoT) | Set 1 - GeeksforGeeks. (2018, August 14). *GeeksforGeeks*. URL: <https://www.geeksforgeeks.org/introduction-to-internet-of-things-iot-set-1/>.
- [5] Amazon Web Services - *Wikipedia*. (2002, July 1). URL: https://en.wikipedia.org/wiki/Amazon_Web_Services.
- [6] Cloud Products. *Amazon Web Services, Inc.*. URL: <https://aws.amazon.com/products/>.
- [7] Introducing Model Server for Apache MXNet | Amazon Web Services. (2017, December 8). *Amazon Web Services*. URL: <https://aws.amazon.com/blogs/machine-learning/introducing-model-server-for-apache-mxnet/>.
- [8] Why Docker? | Docker. (n.d.). *Docker*. URL: <https://www.docker.com/why-docker>.
- [9] OpenCV - *Wikipedia*. (2012, November 1). *OpenCV - Wikipedia*. URL: <https://en.wikipedia.org/wiki/OpenCV>.
- [10] Training the Amazon SageMaker Object Detection Model And Running It On AWS IoT Greengrass – Part 1 Of 3: Preparing Training Data | *Amazon Web Services*. (2019, November 27). *Amazon Web Services*. URL : <https://aws.amazon.com/blogs/iot/sagemaker-object-detection-greengrass-part-1-of-3/>.
- [11] Edge Computing - *Wikipedia*. (2019, May 14). URL: https://en.wikipedia.org/wiki/Edge_computing.
- [12] Hazelcast. *What Is Machine Learning Inference?*. (2020, September 5). URL: <https://hazelcast.com/glossary/machine-learning-inference/>.

- [13] AWS Lambda. (n.d.). *Choosing and managing runtimes in Lambda functions - AWS Lambda*. URL: <https://docs.aws.amazon.com/lambda/latest/operatorguide/runtimes-functions.html>.
- [14] AWS IoT Greengrass. (n.d.). *Access local resources with Lambda functions and connectors*. URL: <https://docs.aws.amazon.com/greengrass/v1/developerguide/access-local-resources.html>.
- [15] AWS IoT Greengrass. (n.d.). *machine learning resources from Lambda functions*. URL: <https://docs.aws.amazon.com/greengrass/v1/developerguide/access-ml-resources.html>.
- [16] What Is AWS CloudFormation? - AWS CloudFormation. (n.d.). *What is AWS CloudFormation? - AWS CloudFormation*. URL: <https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/Welcome.html>.
- [17] Infrastructure As Code – AWS CloudFormation – Amazon Web Services. (n.d.). *Amazon Web Services, Inc.*. URL: <https://aws.amazon.com/cloudformation/>.
- [18] AWS CodePipeline | Continuous Integration & Continuous Delivery. (n.d.). *Amazon Web Services, Inc.*. URL: <https://aws.amazon.com/codepipeline/>.
- [19] Getting Started With Git And AWS CodeCommit - AWS CodeCommit. (n.d.). *Getting started with Git and AWS CodeCommit - AWS CodeCommit*. URL: <https://docs.aws.amazon.com/codecommit/latest/userguide/getting-started.html>.
- [20] AWS CodeCommit | Managed Source Control Service. (n.d.). *Amazon Web Services, Inc.*. URL: <https://aws.amazon.com/codecommit/>.
- [21] What Is the CI/CD Pipeline? | IBM. (2021, September 27). *What Is the CI/CD Pipeline? | IBM*. URL: <https://www.ibm.com/cloud/blog/ci-cd-pipeline>.
- [22] View MQTT Messages With the AWS IoT MQTT Client - AWS IoT Core. (n.d.). *View MQTT messages with the AWS IoT MQTT client - AWS IoT Core*. URL: <https://docs.aws.amazon.com/iot/latest/developerguide/view-mqtt-messages.html>.
- [23] What Is AWS IoT? - AWS IoT Core. (n.d.). *What is AWS IoT? - AWS IoT Core*. URL: <https://docs.aws.amazon.com/iot/latest/developerguide/what-is-aws-iot.html>.