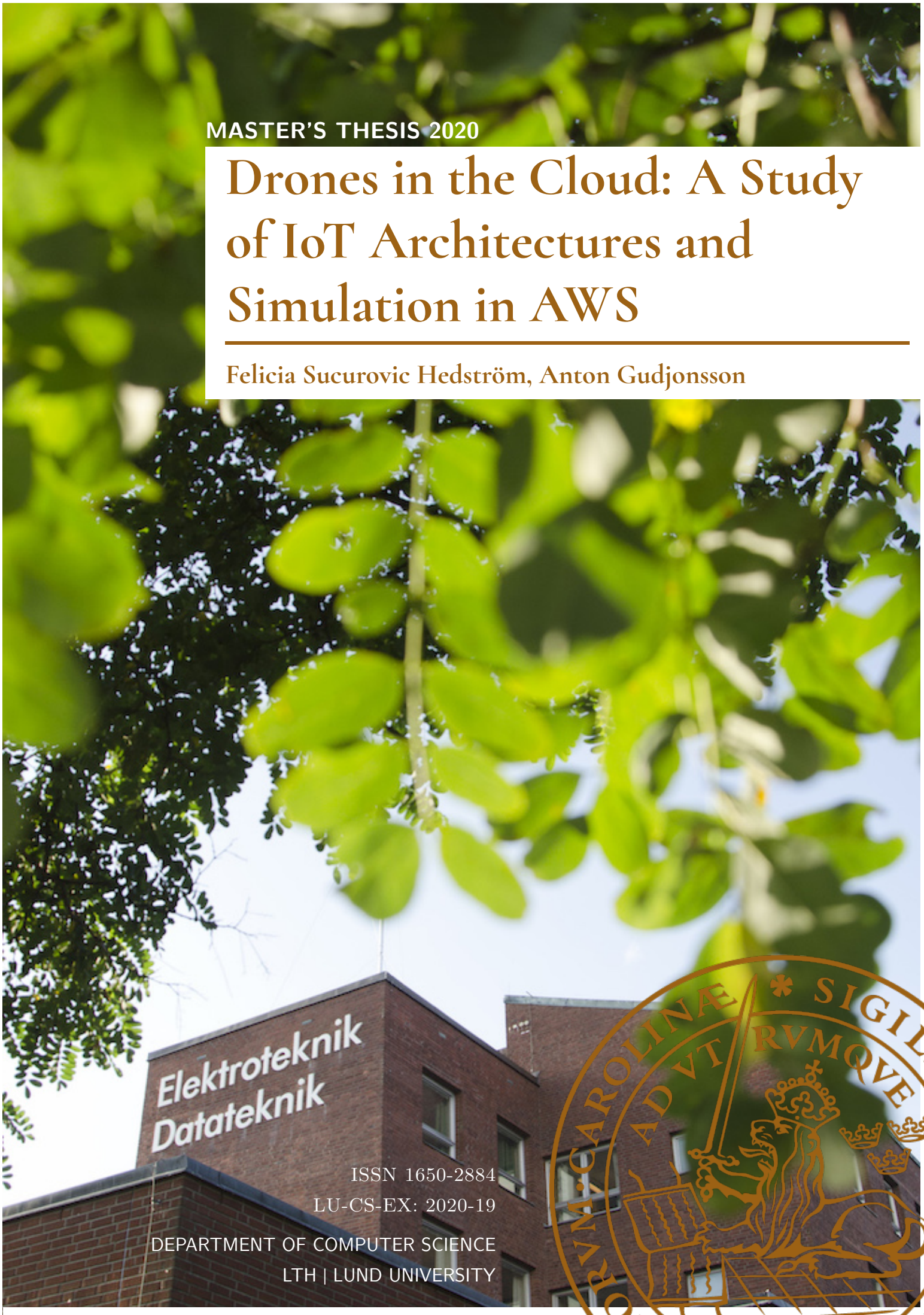
MASTER'S THESIS 2020

# Drones in the Cloud: A Study of IoT Architectures and Simulation in AWS

Felicia Sucurovic Hedström, Anton Gudjonsson




ISSN 1650-2884
LU-CS-EX: 2020-19

DEPARTMENT OF COMPUTER SCIENCE
LTH | LUND UNIVERSITY

EXAMENSARBETE
Datavetenskap

LU-CS-EX: 2020-19

# Drones in the Cloud: A Study of IoT Architectures and Simulation in AWS

**Felicia Sucurovic Hedström, Anton Gudjonsson**

# Drones in the Cloud: A Study of IoT Architectures and Simulation in AWS

Felicia Sucurovic Hedström
dat14fhe@student.lu.se

Anton Gudjonsson
dat14agu@student.lu.se

June 17, 2020

Master's thesis work carried out at Dewire Consultants AB.

Supervisors: Christian Eriksson, christian.h.eriksson@knightec.se
Ulf Asklund, ulf.asklund@cs.lth.se

Examiner: Flavius Gruian, flavius.gruian@cs.lth.se

## Abstract

With IoT devices becoming both smaller and more computationally powerful, new cloud computing architectures have evolved. These new architectures have presented an opportunity to use IoT for more purposes, such as autonomous drones which is what will be focused on in this thesis. As a result of this, many cloud computing platforms now offer support for hosting IoT networks. Amazon Web Services will be used for developing and hosting the IoT networks. The three IoT architectures which will be investigated in this thesis are Cloud, Fog, and Edge computing. These architectures will be evaluated using metrics relevant to a real-time IoT system. The result of the evaluation is a recommendation of an architecture that is the most suitable for our use case. This architecture is a hybrid solution that performs time-critical computations onboard the device and all other operations in the cloud.

Since IoT networks can consist of many devices it might not be feasible to create large networks of physical devices for testing. Therefore there is a need to investigate whether the simulation of devices is something that can be used for testing, especially smart devices such as drones. As a solution to this, this thesis will explore the possibilities of simulating drones using AWS. The AWS simulation solution offers the possibility to generate large amounts of data from simple devices. However, this is not enough to be able to simulate a drone system completely since the simulated devices lack the ability to receive any input.



**Keywords**: MSc, IoT Architecture, AWS, Simulation, Cloud computing, Fog computing, Edge computing

# Acknowledgements

We would like to thank everyone involved in the process of this master thesis. A special thanks to our supervisor at Dewire Christian Eriksson whom has provided us with knowledge and valuable input throughout this thesis.

We would also give a special thanks to our supervisor at LTH Ulf Asklund whom has guided us trough this process, and provided us with valuable feedback in regard to our report and how to structure our work.

A big thanks to everyone at the Dewire office, for making our time there educative and helping us with AWS related problems.

# Contents

# Chapter 1
# Introduction

The Internet of Things (IoT) is the concept of connecting everyday physical "things" to the internet. These "things" are devices such as microcontrollers, sensors, actuators, or other smart appliances. Since the IoT devices are connected to the internet, it allows them to communicate with each other or a central computer without human interaction, but can also be monitored or controlled remotely by a human. As IoT has gotten bigger, it has been introduced into many areas of everyday life. Some of these areas are Smart Cities, Healthcare, Wearables, and Smart Homes, and by 2025 it has been predicted that the amount of IoT connections will reach at least 24.9 billion [1].

With IoT devices becoming more popular, several cloud service providers are offering solutions for building IoT networks. Cloud computing is the concept of on-demand computer resources, usually storage and computing power. The user simply requests access to certain resources as the needs appear, needing not to worry about any physical hardware or infrastructure. Amazon, Microsoft, and Google have, among others, established themselves as cloud service providers with their services Amazon Web Services, Microsoft Azure, and Google Cloud. They utilize big data centers across the continents to provide resources to their users. Since these services are mostly pay-per-use or pay-as-you-go it has allowed companies to become flexible and expand seamlessly as they grow and smaller companies can handle great computational loads without the need of their own infrastructure.[2] The use of cloud computing has allowed IoT devices to become both smaller and more lightweight since the computing power can be outsourced. This has led to the emergence of a new IoT domain called Flying IoT. These flying IoT devices are drones utilizing the computational power of the cloud. This thesis will explore how flying IoT can be constructed using a cloud service provider. The focus of the thesis will be on how the drones communicate and where the computing power resides. Since IoT networks can consist of many devices it might not be feasible to create large networks of physical devices for testing. It would be especially helpful to simulate more computationally powerful devices such as drones since they can be expensive. Therefore this thesis will also investigate the possibilities of testing flying IoT systems using simulation.

## 1.1 About Dewire

Dewire is a consulting company with offices in Sundsvall, Stockholm, and Gothenburg. They are a certified APN Consulting Partner (a member of the Amazon Partner Network) and specialize in digitalization solutions using Amazon Web Services. With future customers showing an interest in autonomous drones and IoT, they have asked us to further explore the AWS IoT services.

To narrow down the area of research Dewire provided us with a use case, detailing a system for controlling drones. The use case identifies and summarizes the needs of future projects and will be presented in the next section. The use case will be analyzed and discussed further in chapter 3.

### 1.1.1 Use Case

With the landscape of autonomous drones quickly emerging, Dewire foresees that future commissions may involve systems for controlling fleets of such devices. The system should be able to control and set targets for drones out of sight from a central controlling system. In turn, the drones should report back to the central system.

The envisioned network could be deployed and used for the following scenarios:

- supervision and inspection of equipment or assets in remote location, eg.
  - electric power transmission networks
  - firebreaks
  - spread of forest fires
  - ordnance surveying
- transportation of packages and equipment
- provide coverage (eg. internet) in remote areas.

The drones should:

- report data back to control. The reported data should include:
  - coordinates (3D)
  - conditions (wind, temperature)
  - speed
  - direction of travel
  - system health (battery status, power level and status of motors, signal strength (GSM/Cellular, GPS, Wifi))
  - planned route
  - sensor data, if available
- reports should be sent every second

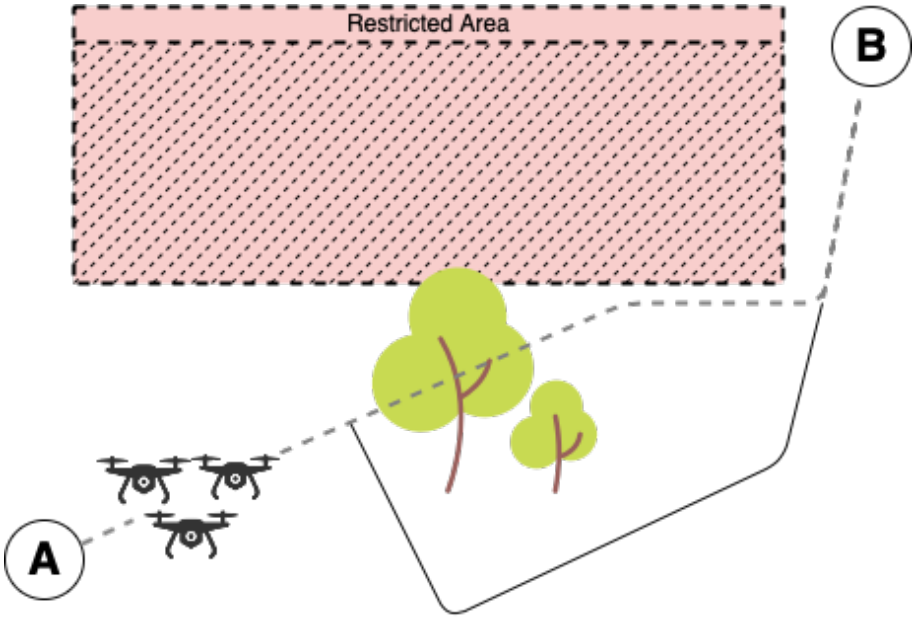



**Figure 1.1:** A drone flying from point A to point B, while needing to recalculate its route because of an object in its way.

- respond to new goals from the control and adjust route accordingly
- respect geo-fencing restrictions (3D) set up by the controller
- detect obstacles and avoid collision
- take emergency action if:
  - lost connection to controller
  - system failure
  - battery levels under limit
- emergency actions could be:
  - return to base (drone specific)
  - hover
  - continue to destination

In figure 1.1, it is illustrated how the drones will operate in the field. The drones will receive an initial command from a user. This command will consist of instructions on how to get from point A to point B with respect to areas restricted by geo-fencing. This route is illustrated by the dotted line in the figure. Along the way to point B, the drones will report data back to the controller. This controller will be located differently depending on the architecture. Eventually, the drones will detect a tree and will need to take avoidance action. This action will need to respect both the tree and the restricted area when calculating the new route. This route is represented by the line. These calculations are made by the controller and can differ depending on the capacity of the drone.

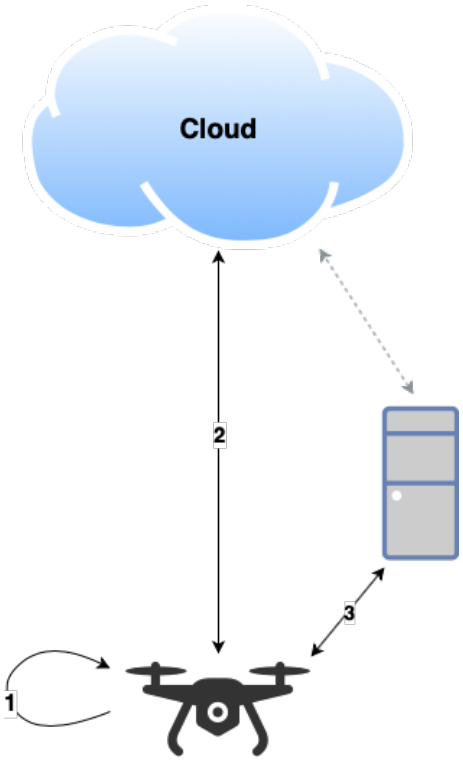



**Figure 1.2:** Communication paths for 1) Edge, 2) Cloud, 3) Fog

## 1.2 Problem definition

As IoT devices become smaller and computationally more powerful there are greater opportunities to distribute the workload, instead of performing all the computations in the cloud. This has led to several different IoT architectures emerging on the market. These architectures take advantage of the increased computational power of the devices by moving computations away from the cloud. We have narrowed it down to three architectures which we will investigate further to see which one we believe fits our requirements the best.

These three architectures will have different communication paths from the drone to where the computations are made. These paths are illustrated in figure 1.2. Number one in the figure represents the edge architecture where the computations are made onboard the drone itself. Number two is the Cloud architecture where all computations are made in the cloud. Number three represents the Fog architecture where a local node, an intermediate device, performs most computations. Both the edge and fog architectures still have a connection to the cloud to be able to perform some computations in case the device cannot handle it by itself.

With these architectures in mind, we aim to answer the following question:

- Which IoT architecture is the most suitable for a fleet of drones in regard to the use case?

The suitability of the architectures will be determined by looking at how the different architectures perform in regards to a few metrics. These will be presented in section 3.5.

Since IoT networks can consist of many devices it might not be feasible to create large networks of physical devices for testing. Simulating these IoT devices could solve this problem. It would be especially helpful to simulate more computationally powerful and complex devices such as drones since they can be expensive. To simulate IoT devices AWS offers a solution called IoT Device simulator. This solution claims to be used for the purpose of testing IoT device integration and IoT back-end services. Therefore we aim to answer the following question:

- Can the AWS IoT Device Simulator be used to simulate drones for the purpose of testing?

The Device Simulator will be evaluated by its ability to emulate a drone as described by the use case. With several different potential IoT architectures for the fleet to use, it is also important that the IoT Device Simulator can be used with any of them.

## 1.3 Related work

The research article "Distributed Measurement Data Gathering about Moving Objects" describes two different approaches for measurement data gathering about moving objects [3]. The first approach, and the one most relevant to this thesis, is Fog computing. It presents the idea of moving the assessment of the quality of the data to nodes located closer to the moving objects. The second approach is predicting telemetry quality using mining models. As a result of implementing these approaches, load balancing between the cloud and other edge nodes became possible. This leads to a reduction of network traffic and thereby lowering the requirements for the bandwidth of communication channels and the possibility to gather measurement data over a wireless network.

With an increasing demand for autonomous drones, Devos et. al recognized that an important challenge in the field was for drones to avoid obstacles in a complex environment [4]. They developed an inexpensive obstacle avoidance algorithm capable of being run directly on the drone itself. The obstacle detection was done using two LiDAR sensors attached to the front of the drone and the algorithm was a simple two-neuron neural. The algorithm was run on a Raspberry Pi Zero, connected to the controller by a serial link. In their tests, the drones achieved a 100% success rate of navigating without crashing or getting stuck in an environment with a low density of obstacles and a 90% success rate in a high-density environment with corners and deadlocks.

Alwateer et. al has studied the trade-off between maximizing revenue and client satisfaction for a drone renting service [5]. To do this they created a model for the drones' onboard decision making and performed a simulation in which clients could request a drone for a task and the drone had to make a decision of whether it is able to satisfy the request. The simulation was run with AnyLogic Simulation Software.

A lot of IoT systems are using 4G networks today, and these networks are still developing in order to meet the requirement for new IoT applications that are developed. However, there are a lot of challenges when it comes to IoT and 4G, such as connecting a large number of devices, security, and new standards. The paper "5G Internet of Things: A survey" brings up these challenges and reviews the current research on how 5G will affect IoT and these challenges [6]. It also presents some of the performance requirements of new IoT applications

such as massive connectivity, security, trustworthiness, coverage of wireless communication, ultra-low latency, throughput, ultra-reliable, for a huge number of IoT devices.

## 1.4 Contributions

This thesis will contribute to the understanding of IoT architectures and how they can be used for different types of data. It will also give an insight into how simulation can be used for testing IoT systems.

Both Felicia and Anton contributed to every part of this thesis to some extent. Felicia focused more on the implementation in AWS, while Anton wrote the scripts for connecting and communicating. Anton also put in more work during research regarding the IoT architectures while Felicia focused on developing the test setup. In regard to the report, Felicia did more writing on the simulation part while Anton focused on the IoT architectures.

# Chapter 2
# Theory

In this chapter, we will provide background concepts and practices used during this thesis. It will explain the basics of IoT and present information about the AWS component that will be used. This theory will be used for evaluating the IoT architectures and the IoT Device Simulator.

## 2.1 MQTT

Message Queuing Telemetry Transport (MQTT) is a lightweight messaging protocol that is used for communication in IoT and Machine to Machine (M2M). This is the protocol that our devices will use. It is easy to implement and design for the situation where the setting might be constrained in some way, which could be limitations of bandwidth and/or memory. Most commonly MQTT runs over TCP/IP, but it is supported by any network protocol which can provide ordered, lossless and bi-directional connections [7]. For clients to send messages to each other, an MQTT broker is required. For a client to send a message, it needs to connect to the broker and then publish the message to a topic. The clients that want a copy of this message also needs to connect to the broker and then subscribe to the same topic. For every publish/subscribe to a topic, quality of service (QoS) needs to be specified [8].

There are three levels of QoS:

- QoS 0 - "At most once", i.e the message will be sent only once without any acknowledgment (ACK) from the receiver.
- QoS 1 - "At least once", i.e the message will be received at least once depending on if the ACK from the receiver reaches the sender or not.
- QoS 2 - "Exactly once", i.e the clients will ensure that the message is sent exactly once by using a four-step handshake.

Higher QoS will increase the guaranties for the message to be received, but will also lead to more data traffic. These levels will be discussed for different data types presented in section 3.3. How messages are sent between clients, via the MQTT broker, is illustrated in figure 2.1.

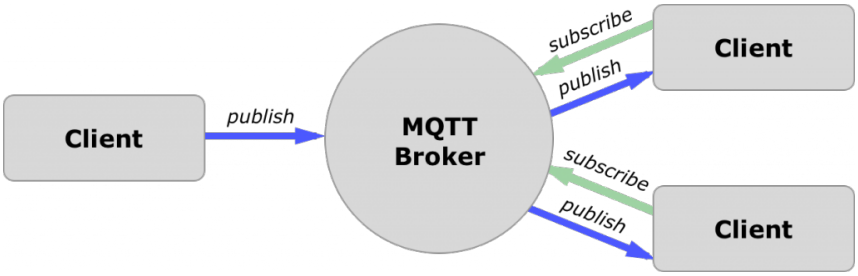



**Figure 2.1:** Clients communicating via an MQTT broker [9]

## 2.2 Drones

As mentioned in the previous chapter, the IoT devices we are mainly interested in are drones. A drone also knows as an unmanned aerial vehicle, is an aircraft without a pilot. Depending on their intended use, they come in varying sizes. A drone can either be controlled by a pilot in another remote location or by having software that makes it fully autonomous [10]. There are many areas in which drones could be useful: package delivery, performing inspections, and collecting data from agricultural sensors among others [11].

The control unit of a drone will vary depending on several things such as application area, model, or size. Therefore it is not certain that all drones can be integrated with AWS without modification. A common way to solve this problem is to attach a Raspberry Pi, or some other compatible microcontroller, to the existing control unit [4].

## 2.3 Raspberry Pie Zero W

In this project, we do not have access to actual drones. To represent drones and local nodes in our test setups we instead use a Raspberry Pi Zero W. It is a complete computer built on a small circuit board. Raspberry Pis uses a system on a chip (SoC) developed by Broadcom for all their products. For this model, the SoC used includes the processor ARM1176JZF-S 1 GHz. It comes with 512MB RAM and connection possibilities via wireless LAN (2.4GHz) and Bluetooth 4.1 [12]. The operating system installed on the SD card is Raspbian, which is officially supported by the Raspberry Pi Foundation.

## 2.4 Metrics

In this section, we will present some metrics which we will use when evaluating the suitability of our chosen IoT architectures. These metrics were chosen because we consider them important for controlling drones.

### 2.4.1 Latency

In the IoT architectures, we will evaluate, data will be sent between edge devices, data centers, and local nodes. The transmission time of this data is called latency. Latency can be measured in two ways, where one is the time it takes for a packet to be sent by one end-point and then received by another, this is also called one-way latency (OWL). The other one is called round-trip latency (RTL), this is the time for OWL plus the time it takes for an acknowledgment to be received by the source end-point [13]. The most common one to use is RTL, since measuring time in only one end-point is not as complicated as syncing two end-points time-wise. The latency will be measured to decide if all three architectures can be used for all types of data presented in the use case

### 2.4.2 Packet error rate

Packet error rate measures the ratio between how many packets reach their destination versus how many were sent. IoT systems often send a lot of data back and forth and since IoT devices mostly use QoS 0, lost packets will not be present. This could result in even a small error rate leading to a big loss of real-time data, which will not be possible to retrieve at a later date. For our specific use case, errors can affect the ability to control the drones. Important messages, such as flight instructions, need to be guaranteed to reach its targets.

### 2.4.3 Data Usage

Data usage is the amount of data that needs to be sent over a network. It measures how much a unit is dependent on an internet connection and what requirements such a connection needs to fulfill [14]. IoT networks can produce enormous amounts of data and can therefore be heavily dependent on a network connection. This dependency can be regulated by the implementation of the IoT network which is what we will look at when using different architectures.

## 2.5 IoT Architectures

There are several different ways to distribute computing and logic in IoT networks. These all have different benefits and drawbacks for flying IoT networks. The three architectures presented in this section are the ones we will evaluate in this thesis. More architectures can be created by combining the three, but we are not interested in those since no new communication paths should arise in them. Therefore we consider it enough to look at each architecture independently.

### 2.5.1 Cloud computing

Cloud computing is defined by having on-demand computing resources over the internet. This can be anything from servers and databases to software and analytics. By having the resources centralized in the cloud they become easy to modify and develop based on one's needs.

In terms of IoT, this means the IoT devices themselves can be very computationally lightweight and have very low energy consumption. This will give them a longer lifetime in the field since they only need a network connection and some sensors or actuators. The computational power available can also be much greater than what could have been on-board the device itself, allowing for faster and more complex tasks to be performed. The downside would be that the devices become reliant on a network connection, which might not always be possible with sensors in remote places, and subject to latency issues meaning they should not rely on any time-critical tasks to be performed by the cloud [15].

### 2.5.2 Fog computing

The term "Fog computing" was created by an employee at Cisco in 2014 with the idea of creating an extension of the cloud somewhere in between the center and the "things". Fog computing tries to move parts of the computation and logic closer to the end devices without burdening the devices themselves. It does this by utilizing a central node, close to the edge devices, with which the devices can communicate [13]. This allows the edge devices to remain lightweight yet have low response times since the central node is close by and handles computation. As shown in figure 2.2 the fog nodes are implemented at geographically different places, where every fog node operates as a lightweight version of the cloud. This solution provides the edge devices with resources located closer to them. It also allows the edge devices to communicate with each other over a fog node. The fog nodes together with the edge devices form a network that provides storage and can perform computations in real-time without the need to go via the cloud. The fog node can be any type of device as long as it can offer the resources the IoT devices requires. This could be storage, computing, or other network resources. A fog node could be anything ranging from a PC to a drone. The purpose of using Fog computing is not to replace the cloud entirely, but to take some of the workloads of the cloud. This will also lead to less traffic between the cloud and the edge devices which will reduce the use of network bandwidth and energy consumption.

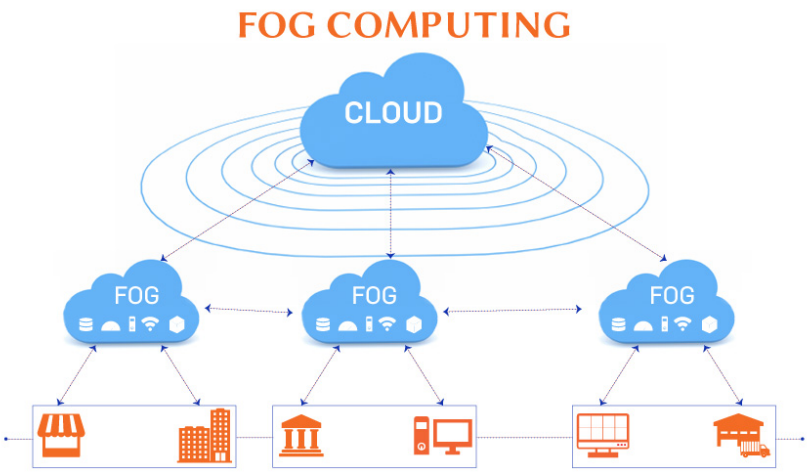



**Figure 2.2:** The concept of Fog computing[16]

### 2.5.3 Edge computing

The increasing number of IoT devices in today's society not only puts an enormous load on the cloud servers processing the data but also the network transporting it [17]. To alleviate this, edge computing can be used. Edge computing moves computation and logic to the edge device itself. By performing computations and making decisions directly on the device, significantly less data needs to be sent to the cloud and the load on central servers decreased. This architecture is also suitable for time-critical tasks. Delay can be split into two parts, computational delay, and transmission delay. Since the edge device performs the computations in direct proximity to the sensors, the transmission delay is eliminated. The downsides of running an edge architecture are the limitations of the device. A cloud server can have virtually infinite resources, while the edge device often is small and have very limited computational power. There can also be some load balancing issues with edge computing since each device takes care of its data separately and some can have a heavy load, while others are idle [18]. If the device is running on a battery, the life expectancy of the device may take a severe hit since computations require a lot of energy. From a software perspective, each device will have to be updated individually if there are any changes made.

## 2.6 AWS Components

Amazon Web Services are a central part of the work we do in this thesis since the IoT systems we build are built using AWS services. In this section, we will provide an introduction to the AWS services which will be relevant to this work.

### 2.6.1 AWS IoT Core

The IoT Core provides Core functionality for enabling IoT devices to connect to both the cloud and each other. The IoT Core can be seen in figure 2.3 and it consists of several subcomponents: Device Gateway, Message Broker, Authentication, Device Registry, Device Shadow, and Rules Engine [19]. The Device Gateway manages all device connections to the IoT Core and supports MQTT, HTTP, and WebSocket messaging protocols. The Message Broker allows devices to easily subscribe and publish data to different topics. The broker keeps track of which devices are subscribed to a certain topic and forwards messages to them when there is something published to it. The broker uses MQTT version 3.1.1. However, the IoT Core only supports sending och receiving messages with QoS 0 and 1, not QoS 2[20]. For a client to connect to the AWS IoT Core it must use an AWS IoT SDK.

Using the Authenticator, the IoT Core also makes sure that all devices connected to the IoT Core are authenticated which means one can always be certain of the identity of the device one is communicating with. All communication is always done using end-to-end encryption between all points of connection. The Device Registry provides devices with a virtual identity that stores the attributes and metadata of the device. The Device Shadow is a virtual copy of the device and keeps track of its state. This allows applications to change the state of a device even if it is currently offline. An example of this would be changing the color of a light bulb from yellow to red while it is turned off. By using a device shadow the bulb will know that it should be red the next time it is turned on. The Rules Engine will filter and

evaluate any messages reaching the IoT Core and can, based on those evaluations, transform and forward them to other AWS services. This feature allows devices to connect to AWS services easily.

These components of the IoT Core are all relevant to understand how central the IoT Core is in the AWS IoT ecosystem but our work will mainly include the IoT Rules Engine, the MQTT Message Broker, the Device Registry, and the Authenticator. In addition to this, it is also the home of AWS Greengrass which will be presented shortly.

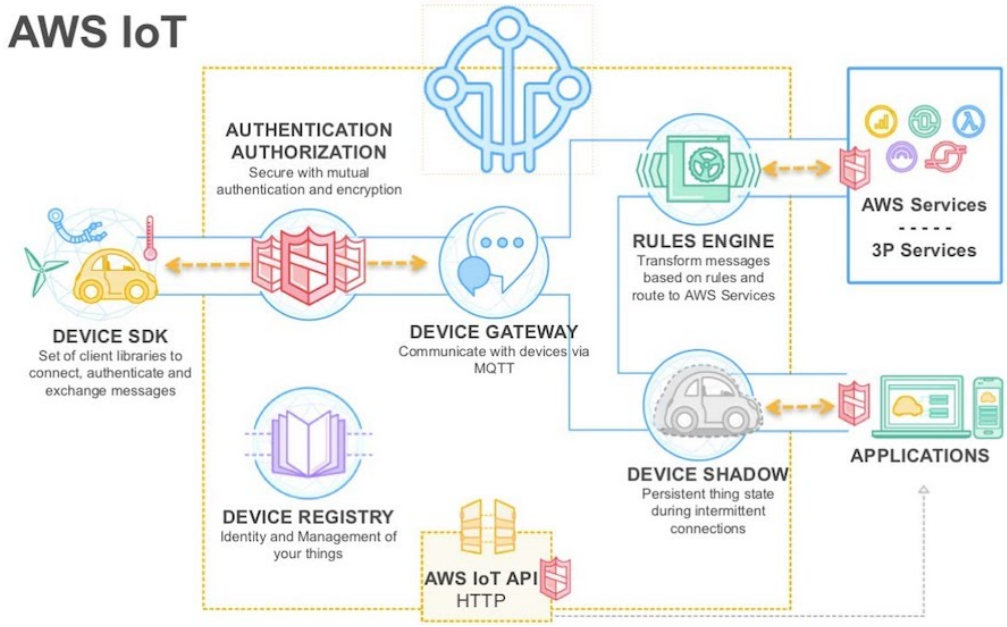



**Figure 2.3:** IoT Core components[19]

### 2.6.2 AWS Greengrass

Greengrass extends parts of the AWS cloud toward the edge of the IoT system, allowing a remote Greengrass Core to run a subset of AWS Services outside of the cloud [21]. The Greengrass Core performs some of the tasks which the IoT Core normally would handle, allowing local IoT devices to send data and perform computing tasks on the Core instead of in the cloud. Since the devices and the Greengrass Core share a local network, the devices can message each other and use AWS services even if the connection with the cloud is offline. Greengrass also supports AWS IoT Greengrass Connectors which makes it possible to connect third-party applications and local resources directly to the Greengrass Core. What device the Greengrass Core needs to be run on will depend on your use case but a minimum of 1GHz (ARM or x86) CPU and 128MB of RAM is required which means it can be run on very light devices.

By using lambdas or Docker containers you can develop code for Greengrass directly in the cloud. Greengrass supports lambdas written in Python, Node JS, Java, C, and C++ and can be deployed directly to the Greengrass Core using over-the-air updates. These deployments contain everything the Greengrass Core needs from device configurations to settings and lambdas. An example of how Greengrass can be utilized is illustrated in figure 2.4.

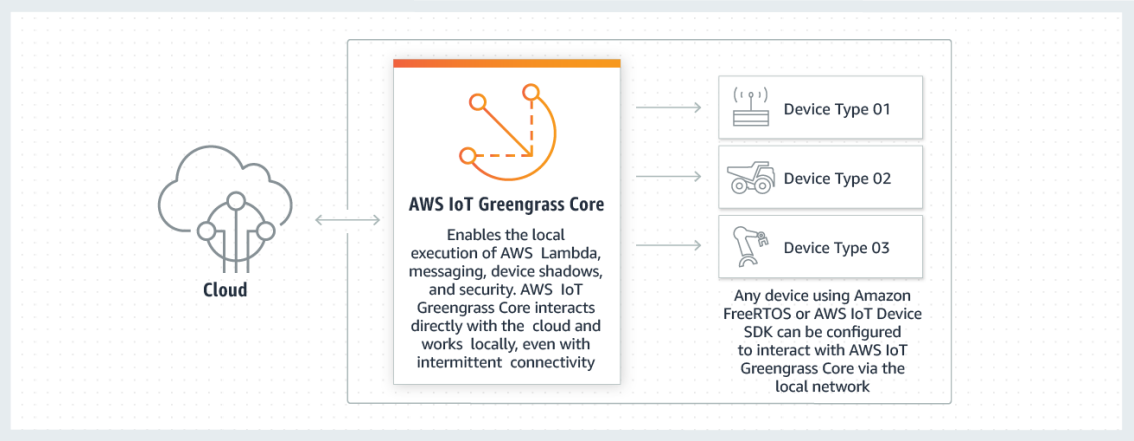



**Figure 2.4:** An example of hoe the AWS Greengrass Core can operate [21]

### 2.6.3 AWS Lambda

AWS Lambda is a versatile tool that allows users to execute code in the cloud without the need to set up hardware or a working environment[22]. It will take care of all requirements for running the code as well as scaling it to meet the rate of requests. This is done using containers, virtual sandboxes which can be tailored to suit the specific needs of the code. If a single lambda is invoked multiple times in short succession it will automatically scale up. More containers are temporarily started to meet the demand and when the number of invocations is fewer the containers are shut down.

Lambda can be used by other services to make them more complex, adding filters and logic to processes that are otherwise simple. By using lambdas one can chain together a number of AWS services and thus create a more full solution. Two of the services which make use of AWS Lambda is the IoT Core and Greengrass. They can invoke a lambda based on the inputs of the IoT devices connected to them, allowing the IoT devices themselves to make use of computing power that might not otherwise be available to them. Greengrass can start two types of lambdas, long-lived and on-demand. An on-demand lambda will create a container when the lambda is invoked and then shut down a while after it has stopped executing, this is the type of Lambda used in non-Greengrass contexts. A long-lived lambda will start when the Greengrass Core is deployed and keep running indefinitely, this is useful for continuous tasks or tasks that require you to save a state. Both types will be used in our test setups.

### 2.6.4 AWS EC2

Amazon Elastic Compute Cloud is a service that provides users with computing power in the cloud. A user can create virtual containers with customizable specifications ranging from CPU power and memory to which libraries and operating systems should be run[23]. The containers will scale based on criteria set by the user, e.g a large number of requests to the application running in the containers. The containers can be started on-demand or be kept alive if an application needs to be online at all times. This service could be interesting when discussing alternative simulation solutions but is also used by the AWS IoT Device Simulator.

## 2.7 Simulation

For testing the IoT network we will investigate the possibility to simulate the devices instead of using physical ones. A simulation is the prediction of the behavior or outcome of a certain system. Given a set of initial parameters, the simulation will calculate the actions of the system and how different actions and parameters might affect each other. To be able to do this one has to create a model of the system. A model is a representation of the system, often a more simple one. Given a specific model and specific parameters a simulation should always return the same outcome (assuming there are no random elements). There are several uses of simulations. By using it before starting a new project or making changes to an existing system one can use the results of the simulation to optimize system performance, find potential bottlenecks, and reduce the chances of failure to meet requirements[24]. We will look to AWS for simulation since it is the cloud platform we have chosen to use throughout this thesis. The simulation solution they offer is IoT Device Simulator, which is introduced in the section below.

### 2.7.1 IoT Device Simulator

The purpose of this solution is to facilitate testing of device integration and other back-end services built in AWS for the purpose of an IoT network. In the graphical user interface created for this solution, a customer can easily build a network of simulated devices connected to each other. These devices send user-defined data to the AWS IoT platform, where the behavior of the back-end can be observed.

The IoT Device Simulator consist of two main parts; a web-based graphical user interface (GUI) and a simulation engine. The Out of the box solution comes with a GUI and a simple random data generator, but these can both be customized to fit one's needs.

By default, there are two categories of devices; general types which can be created and customized by the user, and vehicles. Vehicles are a pre-defined device type that is configured to generate data mimicking a car. At most 100 devices can be created at once but there is no set limit to how many devices one can have. In any case, there can not be more than 1000 simulated devices running at once. Each device simulation has to run for a minimum of 60 seconds and the transmission interval of data can be no shorter than 1 second[25]. Since each MQTT message has a maximum size of around 260MB this means that one can, in theory, simulate around 260GB of data per second.

The source code for the IoT Device Simulator is provided by AWS along with a deployment guide. This allows the user to make any desired changes to the simulator.

# Chapter 3

# Approach

In this chapter, the method for our thesis will be presented. Then we will go through the work process of how we design the IoT architectures and tested them.

## 3.1 Method

First of all, we conducted a literature study to determine which IoT architectures we would research further during this thesis work. The only requirements we had for the architectures were that they should fit the scenarios described in our use case. In order to pinpoint the most important criteria for our system, we categorized the data that would be sent.

Once we had decided on which architectures to investigate we found what metrics we thought were suitable for evaluating these architectures. After we established which architectures and metrics to use, we made a design for each architecture using AWS resources.

Given the designs, we implemented a simplified version to use for testing the latency of the different IoT architectures. Since we did not have access to drones we used Raspberry Pis instead. The tests were then conducted and the results evaluated.

When we were done with testing the latency, we set up the AWS IoT Device Simulator. First, we investigated the IoT Device Simulator's ability to simulate a drone by customizing the source code. Then we looked into the possibilities of adapting the IoT Device Simulator to simulate devices for the architectures we had evaluated. First of all, we conducted a literature study to determine which IoT architectures we would research further during this thesis work. The only requirements we had for the architectures were that they should fit the scenarios described in our use case. In order to pinpoint the most important criteria for our system, we categorized the data that would be sent.

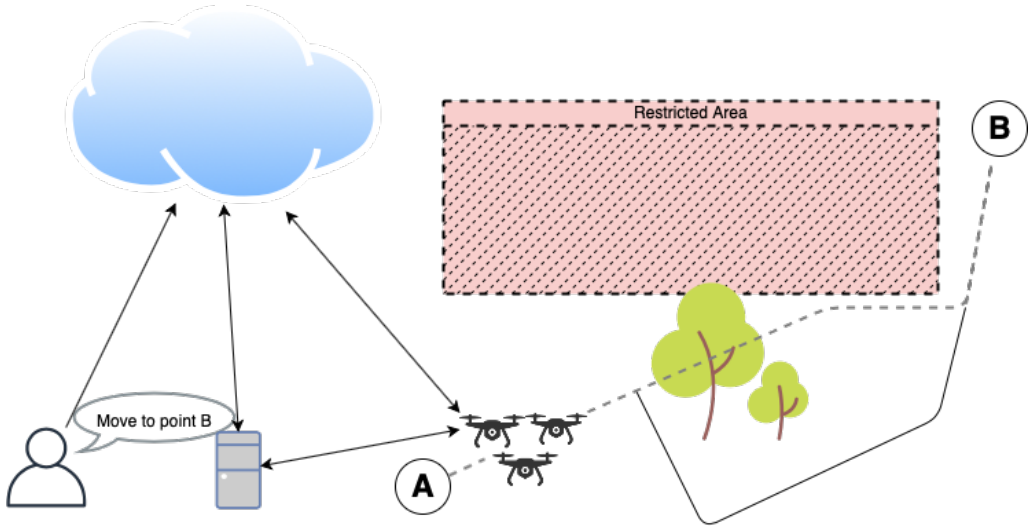



**Figure 3.1:** An illustration of how the system will operate

## 3.2 Description of the system

In figure 3.1 it is illustrated how the system will operate. The user will command the drones in point A to move to point B. This command will be received by the cloud and depending on the implementation and architecture used, the route can be calculated in a number of different ways. It can either be calculated directly in the cloud, on a local node (the grey box), or on the drone itself. This route calculation will take restricted areas into consideration. If a drone encounters an object in its way, it will need to perform an avoidance action and recalculate its route based on this new information. These calculations will also take place at different locations in the system based on what architecture is used.

## 3.3 Categorization of data in use case

Data is a very general term and to make it easier to refer to the different kinds of data in the use case we have chosen to categorize it. We have chosen to divide them into three different groups based on what requirements they impose on the system.

Since the capabilities of the drones can vary, our focus is not on how the data is formulated but rather how it is sent to the drone. An example of this is a drone receiving a command to fly to a location. Some drones might only be able to process simple instructions such as "Turn left" while others can process "Go to X, Y, Z". How these instructions are sent to the drones can be impacted by implementing different architectures.

### 3.3.1 Group A

This group is a data report including data from sensors such as condition, speed, and direction. This data will be stored in a database and used for analyses and other non-time-critical tasks. The data reports are sent often and usually with a low QoS since each individual message is not important for the autonomous functionality of the drone. The latency of the data

is not crucial for the system. It is more important that the messages arrive in the correct order. This can easily be solved by adding a timestamp to every message, enabling the system to sort the data.

### 3.3.2 Group B

Instructions for the drone. These instructions could be for collision avoidance or simply coordinates for a new goal. It is important that the messages are received by the drone once and only once, which can be solved by using a higher QoS. However, AWS IoT does not support QoS 2, meaning the system needs to be able to handle if a message is received twice. For this group, latency is something we need to take into account since some instructions, such as avoiding collisions, needs to be executed quickly.

### 3.3.3 Group C

Data reports of critical data such as system health, coordinates, and obstacle detection sensor data. This is data that the drone might have to take action upon such as respecting geo-fencing and changing routes based on coordinates or take emergency action based on system health. Therefore it is important that the data is guaranteed to reach its target, but this can be implemented in a number of ways. The messages can either be sent with a higher QoS or with a higher frequency to ensure it is acted upon in time. In this case, QoS 1 is enough since it does not matter if duplicate data reaches its target. This data might trigger the instructions in Group B.

## 3.4 IoT Architectures

We conducted a literature study to determine which IoT architectures would be suitable to investigate further in the context of the use case. We found that the most common and accepted architectures were *Cloud, Fog, and Edge computing*. It seemed natural to choose these three architectures since Cloud and Edge computing are polar opposite concepts and Fog computing is something in between.

Based on the requirements presented in section 1.1.1, we designed the three architectures in AWS. To make it easier to understand how the architectures function and how they differ from each other we will present them in the context of a real-world scenario, this can be seen in figure 3.3, 3.5 and 3.7.

For this scenario, we will assume the drones have some basic autonomous functionality. Drones can have varying computational capacity and different pre-installed functions. Collision avoidance is not something that needs to be implemented on all drones since some models will have it pre-installed. We have still chosen to use this as our scenario in section 3.4 since a lot of drones do not have this pre-installed and it is a crucial part of the system. However, the obstacle detection scenario can be replaced with any scenario similar to this e.g. respecting geo-fencing.

*The scenario is as follows:*

Somewhere along the route to the given destination, while flying autonomously, the drone encounters a building in its path. This is detected by sensors attached to the drone. The

sensor data is used to make a decision on whether or not to perform an avoidance action. It performs the potential action and then continues to its destination.

We have chosen to handle this scenario by using two lambdas, the Filter lambda, and the Action lambda. The Filter lambda filters the data sent from the obstacle detection sensor and decides if the drone needs to take avoidance action. If action is needed, the Action lambda is invoked. The Action lambda will calculate an appropriate avoidance action based on the data available at the time and give instructions to the drone.

### 3.4.1 Cloud Architecture

In figure 3.2 it is illustrated how the cloud architecture will operate in regards to the data groups. As we can see, all group A and C data are being sent to the cloud and all the group B data is being sent from the cloud. The thickness of the arrows represent the amount of data sent, a thicker arrow means more data.

For the scenario, the sequence of events is illustrated in figure 3.3. The obstacle detection sensor, attached to the drone, sends collected group C data to an IoT Topic in the cloud. An IoT rule is set up to listen to that same IoT Topic. When the IoT Rule receives data, it will invoke the filter lambda and forward the message there. The filter lambda decides if the drone needs to perform an avoidance action or not based on the data. If action is needed i.e the building is too close, the Action lambda is invoked. It will calculate what action needs to be taken based on the data and publish these instructions (group B data) to an IoT Topic. The drone is subscribed to that same topic and will perform any instructions published to it.

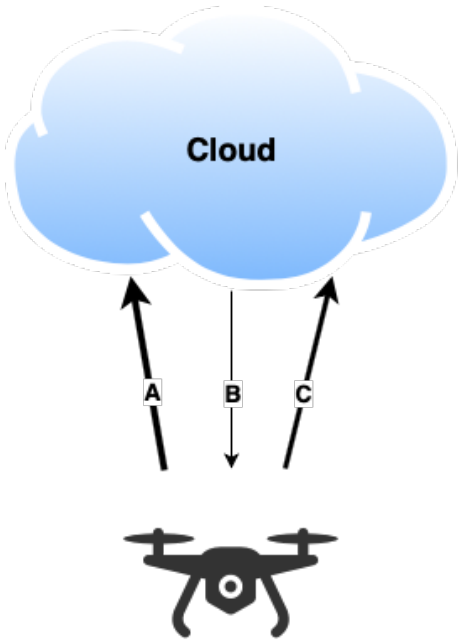



**Figure 3.2:** Data flow for each data group in the Cloud architecture

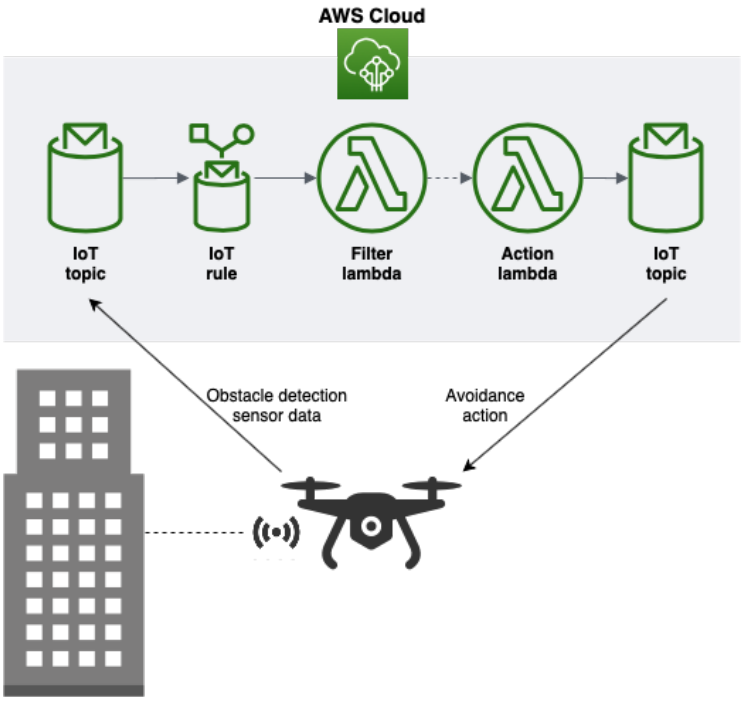



**Figure 3.3:** Obstacle detection using Cloud computing

### 3.4.2 Fog Architecture

The Fog architecture and how its data is sent is illustrated in figure 3.4. Unlike the cloud architecture, the fog architecture has a local node, a fog node, which receives the data. Since the fog node has processing power it can handle some of the computations, resulting in group B and C data only going back and forth to the fog node. Group A still has to be sent to the cloud, but the fog node can filter it which decreases the amount of data being sent to the cloud.

In figure 3.5 the scenario is illustrated for the Fog architecture. As in the cloud architecture, the obstacle detection sensor, attached to the drone, sends collected group C data to an IoT Topic. However, in this case, the message broker is not located in the cloud but rather on the Greengrass Core acting as the fog node. Instead of having an IoT rule, the Greengrass Core can set subscriptions between AWS resources via IoT topics. A subscription is set from the drone to the filter lambda. The filter lambda is invoked by the sensor data and decides if the drone needs to perform avoidance action. If action is needed i.e the building is too close, the Action lambda is invoked. It will calculate what action needs to be taken based on the data and publish these instructions (group B data) to an IoT Topic. The drone will receive these instructions since a Greengrass subscription is set up between the action lambda and the drone via that specific topic.

This architecture differs from the Cloud architecture by having a local fog node, the Greengrass Core. This fog node is located much closer to the drone and acts as an extension of the cloud by hosting the message broker, executing the lambdas, and handling the subscriptions. As long as there is an internet connection, both the drone and Greengrass Core is still connected to the cloud. This connection can be used for syncing and logging or other AWS resources that are not available on the Greengrass Core.

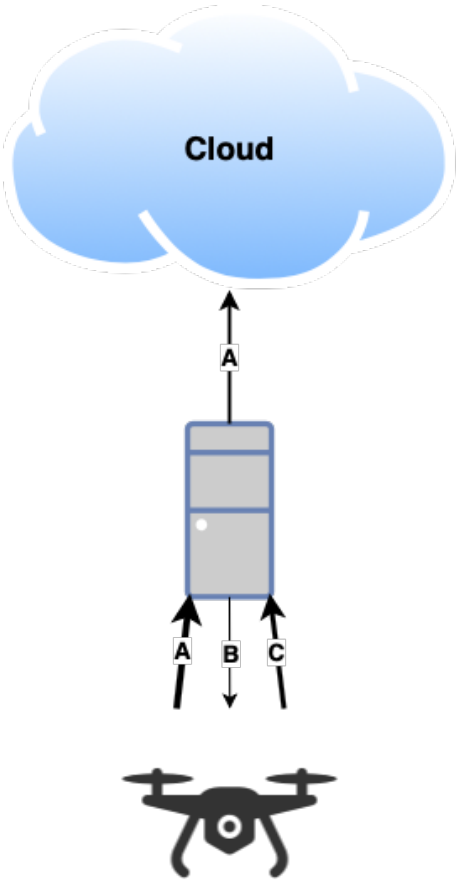



**Figure 3.4:** Data flow for each data group in the Fog architecture

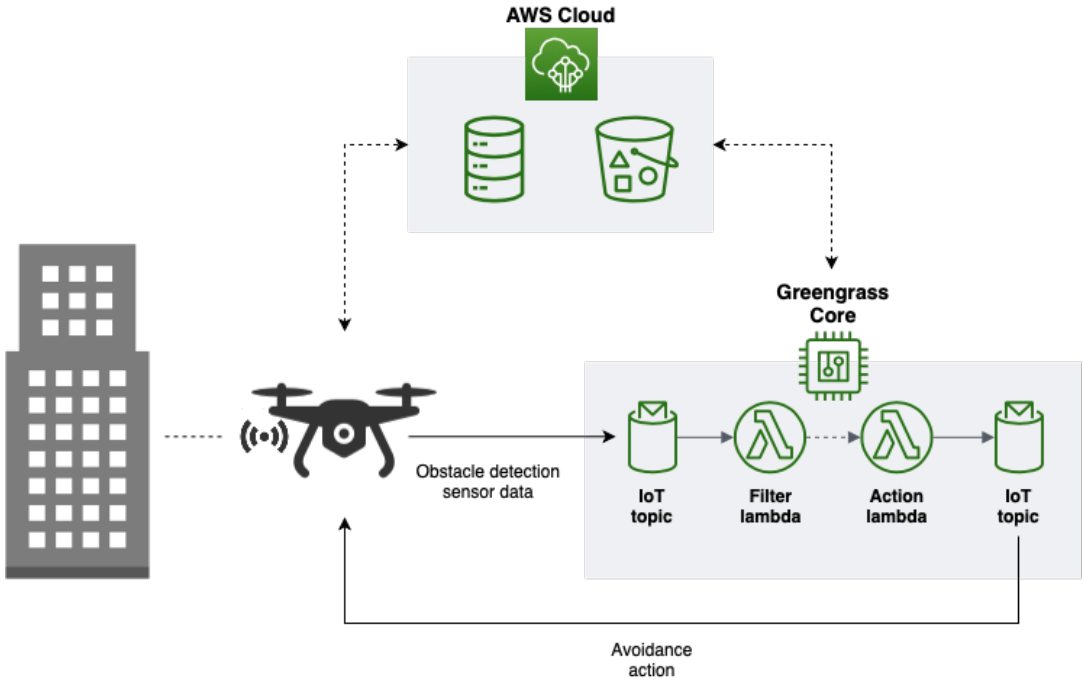



**Figure 3.5:** Obstacle detection using Fog computing

### 3.4.3 Edge Architecture

In figure 3.6 it is illustrated how the Edge architecture will handle the data groups. The computations for group B data are made by the drone itself, meaning group C data does not need to be transmitted over a network. Data from group A is still sent to the cloud but can be filtered by the drone.

The scenario for this architecture, illustrated in figure 3.7, differs the most from both the Cloud and Fog architectures. Since the sensor is attached to the drone, which in this case also is the Greengrass Core, the filter lambda has direct access to its local resources such as sensors. The filter lambda continually accesses the new sensor data and decide if action needs to be taken. If action is needed, the action lambda is invoked and calculates what actions will be performed by the drone. These instructions are sent to the drone's navigation system.

As in the Fog architecture, the Greengrass Core performs all computations in this scenario. However, there is no network traffic in this case, since the drone is the Greengrass Core and can access local resources directly. The drone will still be connected to the cloud to perform logging, syncing, and Greengrass incompatible services.

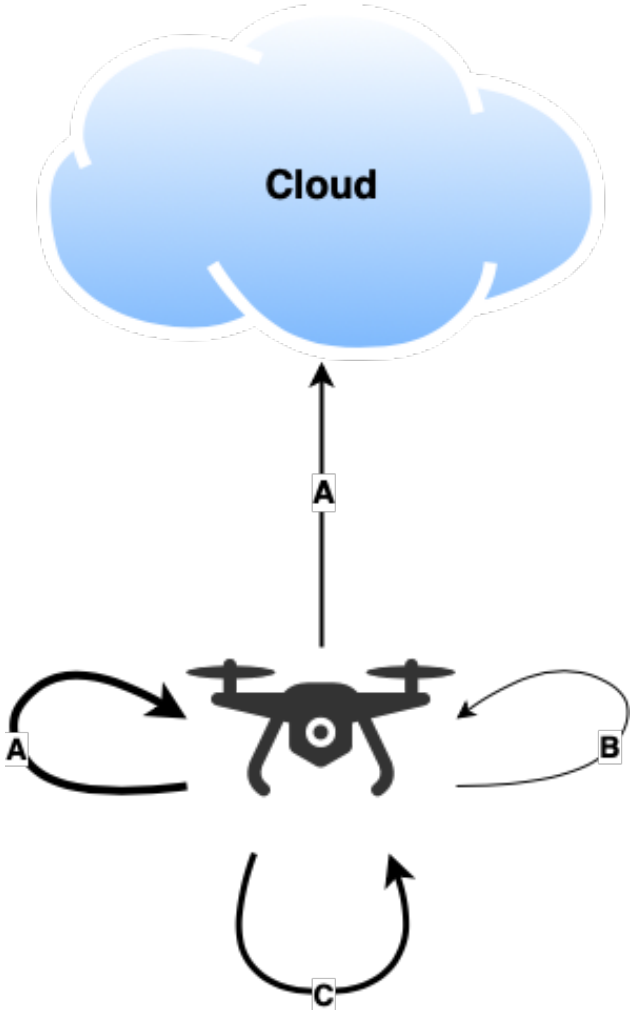



**Figure 3.6:** Data flow for each data group in the Edge architecture

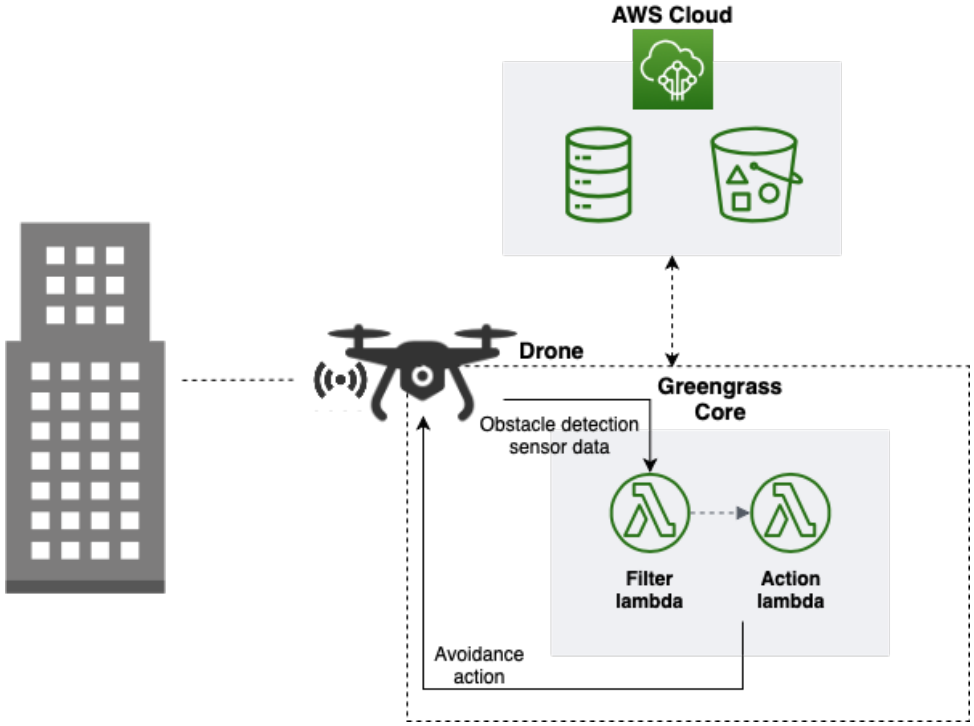



**Figure 3.7:** Obstacle detection using Edge computing

## 3.5 Evaluation method

When evaluating the IoT architectures we have chosen to focus on these three aspects: *latency, error rate, and data usage*. We consider these to be the most relevant when evaluating a system for controlling drones, as we have it described in the use case. In this section, we will present how each of these relates to the use case and how we will use them for the evaluation.

### 3.5.1 Latency

Latency was chosen since there is a significant difference in where the computing power is located in the different architectures. This means the data has to be sent to points at different lengths from the device.

The drones described in our use case are autonomous real-time systems, and the latency for real-time control should at most be 100ms for one-way latency [26]. Several requirements outlined in the use case in section 1.1.1 can be considered as real-time control and will, therefore, be affected by high latency. These requirements are related to unforeseen events that the drone will have to react to, such as emergency actions, respecting geo-fencing restrictions, and obstacle detection.

The measurements will be performed using round-trip-latency with a starting and end-point in the drone. It will measure the time that it takes for the data to be sent from the drone, be received and processed by the two lambdas, and then sent back to the drone.

### 3.5.2 Packet Error Rate

Error rate was chosen because there is a significant risk that some messages may be lost since IoT devices cause a great amount of network traffic. Most IoT devices implement QoS 0 which is a "fire and forget" type of message policy, meaning a lost message will not be resentful. Losing messages will have varying consequences depending on what type of data the message contained. These consequences range from losing a data point from a sensor to not receiving critical instructions.

While designing how the architectures were to be tested, we realized that tests for error rate would not give any meaningful results. Since the conditions of our testing environment are more or less perfect there would be no packet loss to speak of. Instead of focusing on the number of packets lost, we chose to discuss how the different architectures can handle lost messages. We will look at the loss of messages when data is transmitted over a network. In the Cloud architecture, we are interested in the communication between the drone and the cloud. In the Fog architecture, we are instead looking at the communication between the drone and the fog node. Lastly, in the edge architecture, there will be no network communication between the drone and the Greengrass Core and therefore no error rate that we are interested in.

Errors might occur in other places than on the network, such as hardware related errors, but this is not something that can be impacted by the IoT architectures and therefore not something we have chosen to focus on.

### 3.5.3 Data Usage

We chose to include data usage as one of the metrics for evaluating the architectures. The reason for this is because the amount of data sent over a network is one of the major differences between the different architectures. An architecture that sends a large amount of data over the network will be more reliant on a stable network connection. The number of devices in an IoT network will be limited by the network's capacity.

One of the major differences between the three IoT architectures is where computing power and logic lies. This means the architectures will handle data transmissions differently, where some will send more data over the internet than others. The amount of data sent will also have an impact on the other two metrics since they are directly related to network traffic.

As with the case regarding the error rate, we decided not to measure data usage since it would not give us any meaningful results. Data usage is not something that is impacted by external factors but simply a product of our implementation. Since it is our implementation, we will know how much data is being sent in the different architectures and is, therefore, nothing we need to measure. It is not the exact amount of data but rather the impact on the system that we will discuss.

## 3.6 Test setup

In order to test the latency, we built a test setup for each architecture. The test setups were built to resemble the IoT architectures that were designed for the obstacle detection scenario. However, since we did not have access to drones, we used Raspberry Pi Zero W instead. To get

the most accurate latency measurements we tried to simplify the scenario by implementing the lambdas without any real logic. By simplifying the lambdas we get a latency that is not dependent on the implementation or programming language of the lambdas.

The Raspberry Pi Zeros had to be configured in the right way in order for them to be compatible with AWS. We flashed the SD-cards with Raspbian since it is a requirement for installing the Greengrass software. Then we created a virtual device in AWS for each Raspberry Pi Zero. Each virtual device is associated with a private key, a public key, and a public key certificate. All these are, together with a root CA, downloaded on the Raspberry Pi Zero and used for authenticating the device to AWS IoT Core. Both the Fog and Edge architecture are using Greengrass as a part of the solution. To set up a Greengrass Group, a Greengrass Core needs to be created. This is also a type of virtual device with the same requirements for keys and certificates. However, the Greengrass cores will also be required to install the Greengrass software and start the Greengrass daemon. The devices which are going to communicate with the Greengrass Core needs to be added to the same Greengrass Group. Other AWS resources, such as lambdas and connectors, that will be used by the Core also needs to be added to the Group. To enable communication between these resources or devices, subscriptions between them have to defined.

For each architecture, two tests were performed. In both tests, the Raspberry Pi Zero was connected via Wi-Fi. However, in one test the Wi-Fi was a cellphone hotspot connected to the internet via 4G, while the other was a router connected to the internet via a fiber connection.

The reason for having the two setups with different internet connections was because the use cases can differ. In some cases, the drones will have to fly great distances, and then it is reasonable to assume they will be connected using a 4G connection. If, on the other hand, the drones will be more stationary or operate in a limited area such as a large depot then it could be reasonable to assume there could be a fiber connection. The main purpose of these latency tests was to see if the architectures pass the latency limit for real-time control. We assumed the fiber connection would be much faster and wanted to investigate if the connection type would have an impact on if the architecture passes the latency limit or not.

### 3.6.1 Cloud

The test setup we built for measuring latency in the Cloud architecture consisted of a Raspberry Pi Zero, two AWS Lambdas, the AWS IoT Core, and the IoT Rule Engine. This setup and data flow are illustrated in figure 3.8.

We wrote a python script, running on the Raspberry Pi Zero, which utilized the AWSIoTPythonSDK to establish an MQTT connection to the AWS IoT Core message broker with the device keys and certificates. After the connection was established the script published timestamps to the topic latencyTest/cloud/start. For the device to receive timestamps the script also subscribed to latencyTest/cloud/stop. We set up an IoT Rule to be subscribed to the latencyTest/cloud/start and invoke the Filter lambda with the messages published to that topic. The Filter lambda in turn repackaged the message and invoked the Action lambda. The Action lambda repackaged and published the message to the topic latencyTest/cloud/stop. When the Raspberry Pi received the message it saved a timestamp which was then used to calculate the latency. 150 messages were sent to get a mean value of the latency. The first 50 messages were ignored because the lambda containers need a few seconds to be initial-

ized which results in high latency for the first few messages. The messages were sent with one-second intervals as specified in the use case.

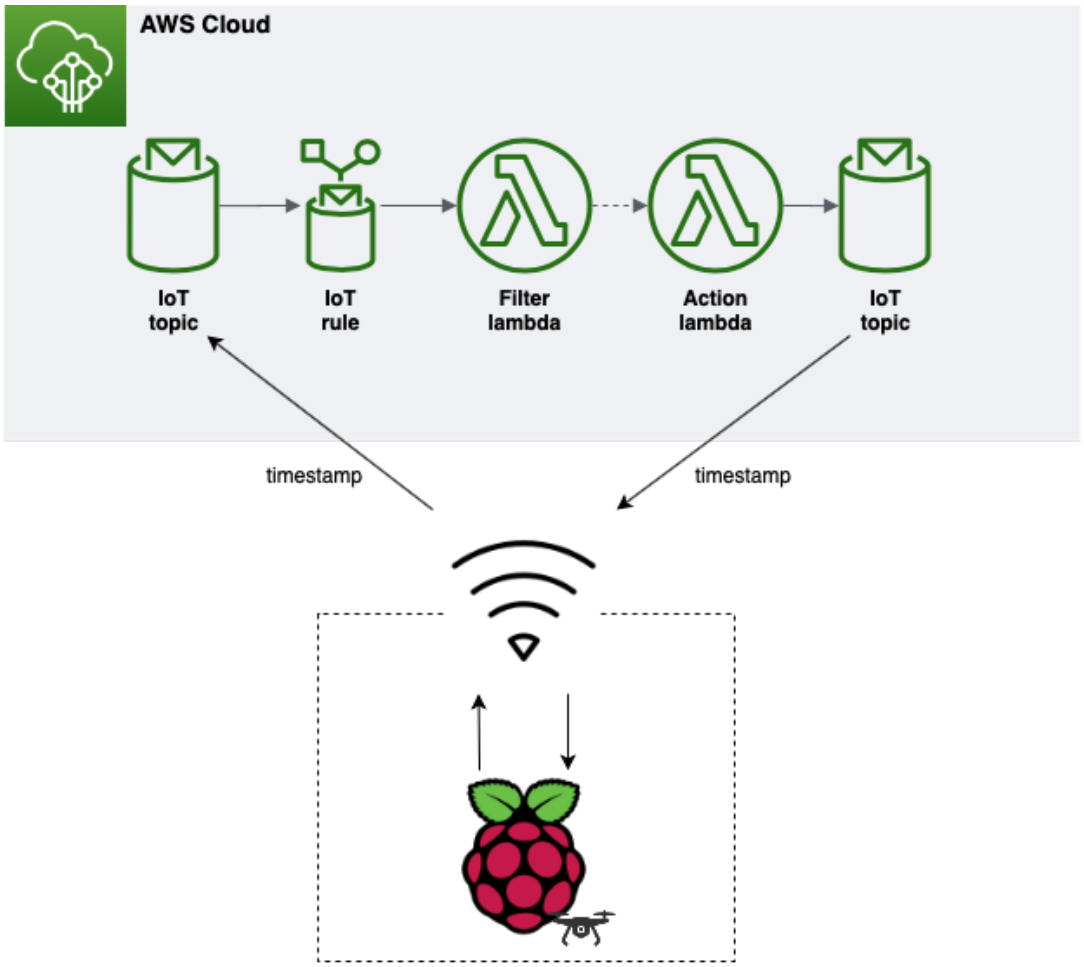



**Figure 3.8:** Test setup for a latency test using Cloud computing

For both tests, the data between the Raspberry Pi Zero to the routing device was transmitted via a Wi-Fi connection. When we used a mobile hotspot, the messages were routed by a cellphone to the cloud via a 4G connection. In the other scenario, the messages will be routed by a router via a fiber connection.

### 3.6.2 Fog

The test setup for the Fog architecture consisted of two Raspberry Pi Zeros (one referred to as Greengrass Core and fog node) and two AWS Lambdas. In a real-world scenario, the fog node would be much more powerful than the drones. However, we decided to represent the fog node using a Raspberry Pi Zero as well since the lambdas only forward the messages and the difference in execution time will be insignificant in relation to the latency.

To enable communication between devices and resources, subscriptions needed to be set in the Greengrass Group. These subscriptions were from the Raspberry pi to the Filter lambda via topiclatencyTest/fog/start and from the Action lambda to the Raspberry pi via topic latencyTest/fog/stop. This setup and data flow is illustrated in figure 3.9.

As in the previous setup, we wrote a Python script that utilized the AWSIoTPythonSDK to establish an initial connection to the message broker, this time locally in the Greengrass Core. By running the script on the Raspberry Pi, it subscribed to the topic latencyTest/fog/stop and then published timestamps to the topic latencyTest/fog/start. Due to having a subscription set up in the Greengrass Group, the filter lambda received the timestamp. The lambda then repackaged the message and invoked the Action lambda with it. In the Action lambda, the message was repackaged again and published to the topic latencyTest/fog/stop. By having a subscription via that topic in the Greengrass Group, and also subscribing to it in the script, the Raspberry pi received the message. When the message is received the script saves a new timestamp which was then used to calculate the latency. As before this was done with 150 messages with one-second intervals, where the 50 first were ignored.

In this setup, the data transmitted between the Raspberry pi and the Greengrass Core is through a Wi-Fi connection. As in the cloud setup, the routing device is either a cellphone with a hotspot or a regular router. In our test case, there was no need to use the cloud and all our communication was therefore via Wi-Fi. The only time a connection to the cloud is needed is when we need to deploy the Greengrass Group to the Core or establish the initial connection to the message broker.

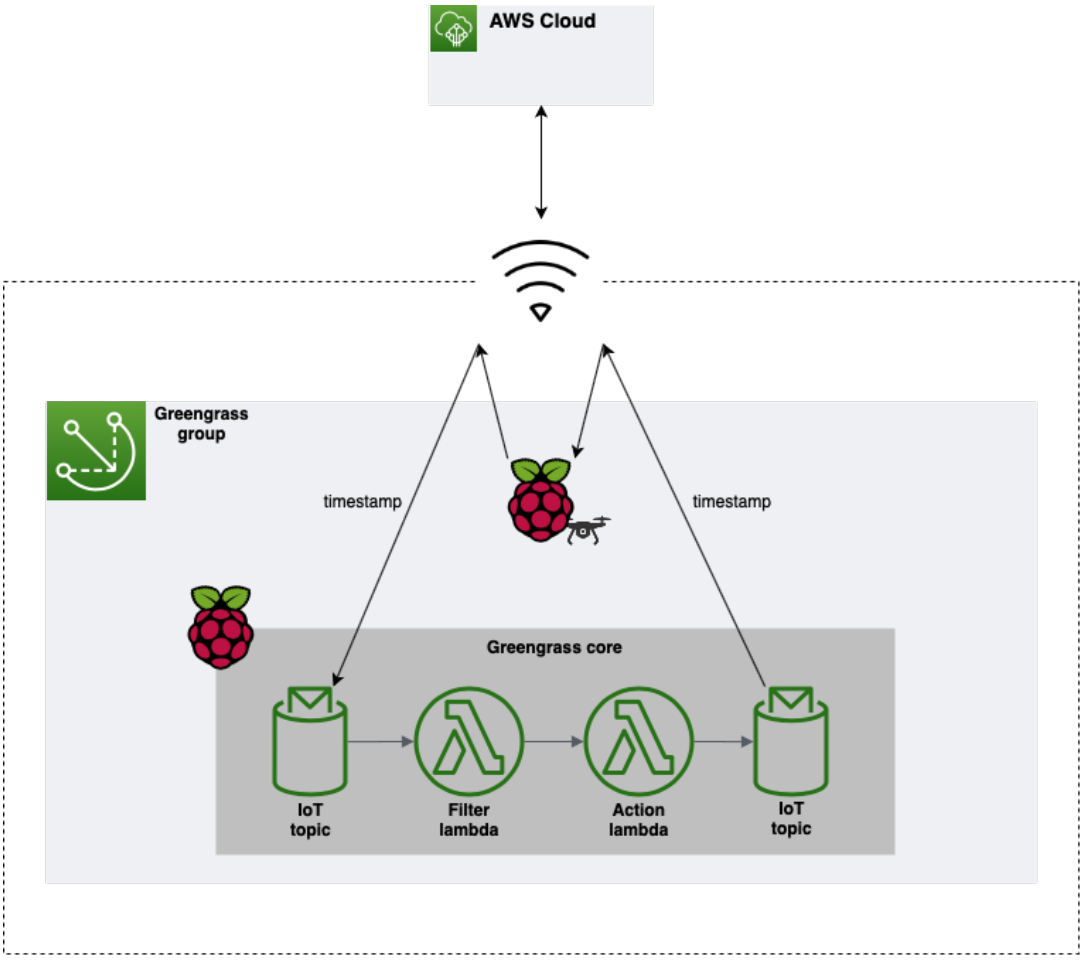



**Figure 3.9:** Test setup for a latency test using Fog computing

### 3.6.3 Edge

The test setup for the edge architecture consisted of a Raspberry Pi Zero running the Greengrass Core software and two AWS Lambdas. By designating the Pi as a Greengrass Core it could be given local access to AWS services. The Filter lambda was initialized as a long-lived lambda, meaning it would run from the point the Greengrass daemon is started. The Filter lambda created messages containing a timestamp and then invoked the Action lambda with the message. This can be done since both lambdas are deployed to the Core and can be run without cloud interaction. When the Action lambda received a message it also saved a timestamp and calculated the latency. This setup and data flow are illustrated in figure 3.10.

As in the fog setup, a connection to the cloud is needed for deploying the configurations and resources needed to the Greengrass Core. Other than that the edge setup does not need any connections, to the cloud or other devices, for it to be able to run the lambdas.

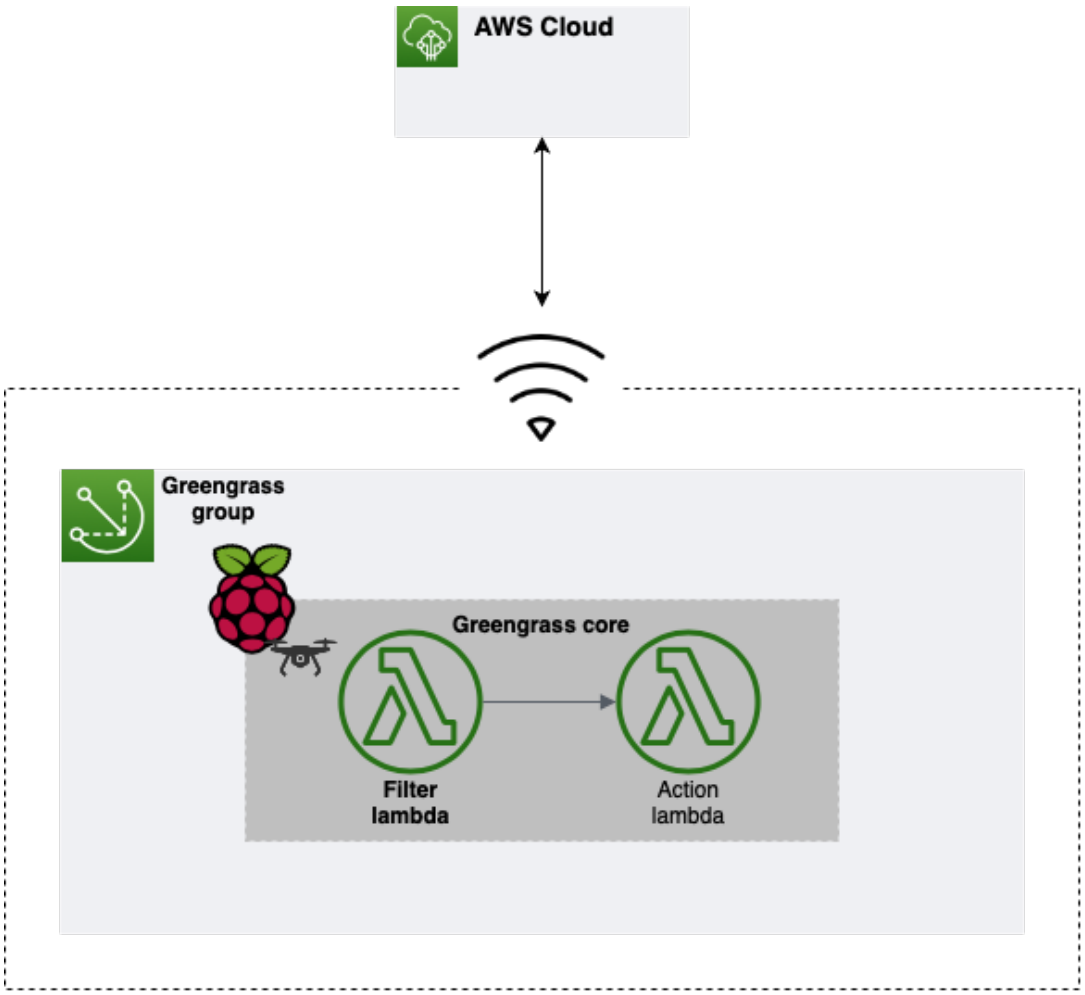



**Figure 3.10:** Test setup for a latency test using Edge computing

# Chapter 4
# Latency Results

In this chapter, the results of the measurements for latency will be presented.

## 4.1 Cloud

The results of the latency tests can be seen in figure 4.1. The plots are very similar and there is no major difference in latency between the two, only an 8.5% increase of the mean latency when using a cellular connection. Both graphs show a consistent oscillation of 0.1s which is about 25% of the mean value. This is the architecture with the highest average latency.

## 4.2 Fog

This architecture has the greatest difference between running on fiber compared to 4G. Using a 4G network resulted in an increase in latency of 38%, the largest difference out of all the architectures. As can be seen in figure 4.2, the fiber plot show very little oscillation with only a few peaks. The peaks are very large however, up to three times the average. The 4G plot shows a higher number of and much more severe peaks, up to six times the average latency.

## 4.3 Edge

In this architecture, there was basically no difference between fiber and 4G. The latency is very low since it is not dependant on a network but rather on the computational power of the device. The peaks at the start of the plots in figure 4.3 is due to how AWS Lambda works with containerization. The oscillation is quite large in relation to the mean latency but is still low compared to the other architectures and is consistent.

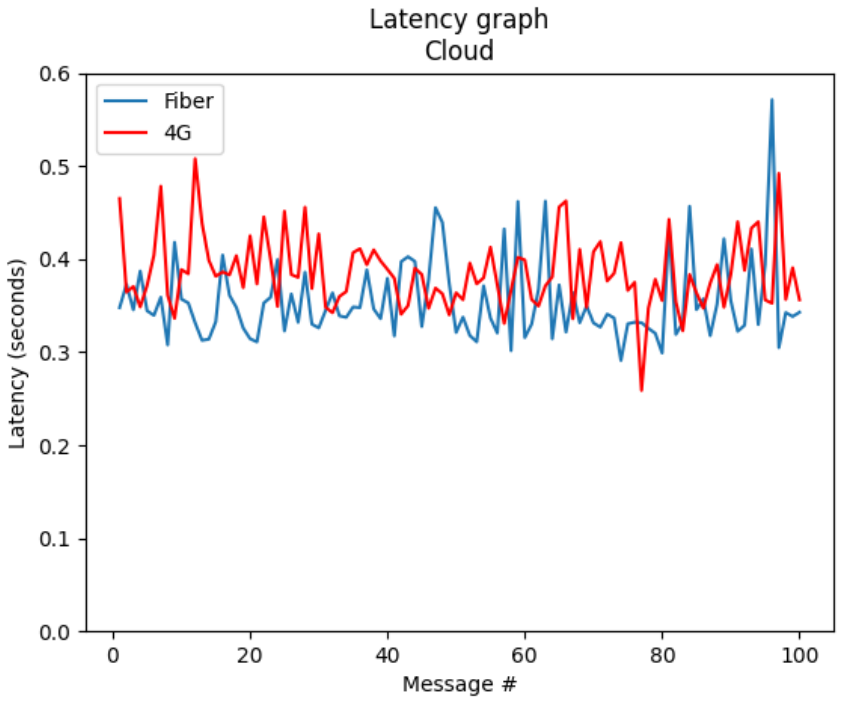



**Figure 4.1:** Round-trip latency on Wi-Fi network, connected via fiber, for Cloud computing

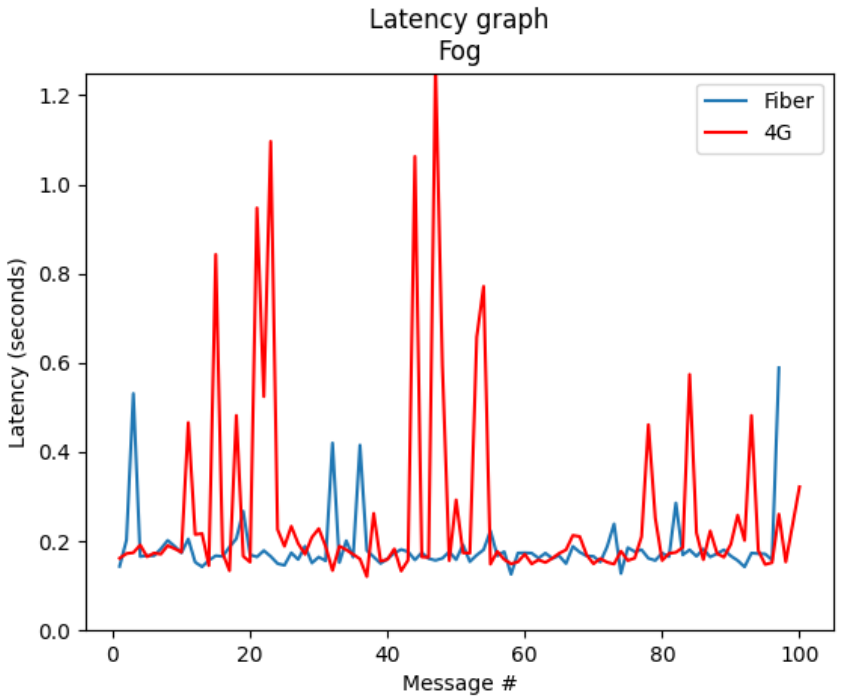



**Figure 4.2:** Round-trip latency on Wi-Fi network, connected via fiber, for Fog computing

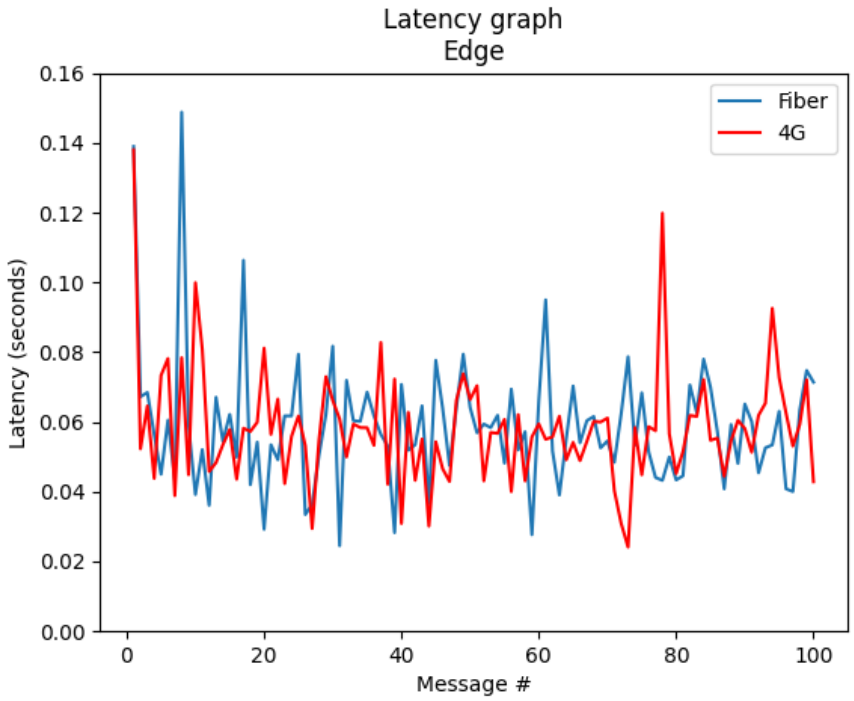



**Figure 4.3:** Round-trip latency on Wi-Fi network, connected via fiber, for Edge computing

| | Cloud | Fog | Edge |
|---|---|---|---|
| **Fiber** | 0.35459372520446775 | 0.18535657027333052 | 0.05864422798156738 |
| **4G** | 0.38516143798828123 | 0.25782711505889894 | 0.05820315837860107 |

**Table 4.1:** Mean values for latency in seconds

# Chapter 5
# Discussion

This chapter will present the discussion regarding the three IoT architectures we have evaluated in this thesis. The discussion of the architectures will be based on the metrics and data categorizations made earlier in the thesis.

## 5.1 Cloud

As familiar, the Cloud architecture performs all its computations in the cloud. Therefore the system relies on a stable connection between the drones and AWS since all data and instructions need to be sent in between these end-points. This makes the Cloud architecture heavily dependent on the performance of all three metrics; latency, error rate, and data usage.

In figure 3.2 it is illustrated how the data will travel. Each arrow represents a data group and the amount of data transmitted. A thicker arrow implies a greater amount of data. As can be seen, all data goes to the cloud which makes the drone dependent on an internet connection. With all of the data being sent to the cloud, it will set higher requirements for the bandwidth of the network. Another consequence of sending all data to the cloud, it that is will increase the risk of latency for all data groups.

This architecture has the highest latency of the three, both for 4G and fiber. As shown in Table 4.1 there is no major difference in latency between the connection types, where 4G has a latency of 0.38s while the fiber connection has 0.35s. Both results are significantly higher than the latency limit of 0.2s for real-time control. This means the cloud architecture should not be used for time-critical operations. Both graphs are oscillating fairly consistently with an amplitude of 0.05s, which is the greatest of the three architectures. Even though the mean value and oscillation are the greatest it is the most predictable latency which could be a benefit. The higher latency for the cloud solution might depend on your location and which AWS region you are using. In our case we were located in Sweden, using the Ireland AWS region (since this was the closest region with support for Greengrass) meaning all data needed to be sent to an Amazon data center in Ireland to be processed.

A longer distance to the data center will likely result in more hops for each data packet in order for it to reach its destination. Since each hop is a potential source of error, a longer distance will increase the likelihood of an error occurring. With the computations in the cloud, all data is vulnerable to errors since it needs to be sent over a network. The impact of an error will differ depending on which group the data belongs to. The data in group A is not critical to the system and a loss of such a message would not affect it to any greater extent than a few missing data points. Losing data from group B on the other hand would have a major impact on the system since these instructions are essential for the autonomous functionalities of the drone. Data from group C can often be what triggers the system to send instructions to the drone and is therefore as essential to the drone as the data in group B. However, it is important that the instructions are executed exactly once while data from group C has to be received at least once. To make sure instructions are received and executed exactly once, QoS 2 should be used. Unfortunately, AWS only supports up to QoS 1, which guarantees message delivery but duplicates can occur. To detect duplicate instructions the drone has to contain logic, which is not possible in a pure cloud architecture. To guarantee that the group C data arrive at the controller, QoS 1 can be used but is not necessary since data reporting from the sensors are done with a high frequency. Group A generates the largest amount of data in the system. This is because the drone report sensor data to the cloud each second. This also applies to group C, but this group reports data from a subset of sensors which reduces the amount of data sent in comparison to group A. Group B is instructions which are based on sensor data from group C. However, not all data point from group C will require action in the form of a group B instruction. Therefore the amount of group B data will always be proportionally smaller than the amount of group C data.

By centralizing all logic in the cloud, the drones themselves can be smaller and less powerful at the cost of being entirely dependent on an internet connection. By using less powerful drones one could reduce the cost of a large system of drones significantly. Due to not having any logic on the drone, there will be no maintenance of duplicate code since all drones use the same functions. The only requirement for the devices is to be able to run any of the AWS IoT device or mobile SDKs, which makes it possible for the devices to connect to AWS IoT Core. The SDKs are available in a number of different programming languages and can thus be run by any device that supports the common high-level programming languages [27].

We found this architecture to be the most straight forward to set up in AWS. It was not difficult to add or remove devices from the system, making it highly scalable. Overall the Cloud architecture is a simple system and easy to use. The administrative tasks become easier by collecting them all in one place but by locating the logic in the cloud the drone will have an increased dependency on the performance of the network. A Cloud architecture could be used for an autonomous drone system but the degree of suitability will vary depending on the scenario for which it will be used. For example, it is not suitable for a scenario where the system requires time-critical operations but is perfectly usable for collecting and storing data.

## 5.2 Fog

Instead of performing all computations in the cloud, the Fog architecture has a local node that can be seen as an extension of the cloud. This node is located closer on the same net-

work and therefore closer to the device. As with the Cloud architecture, the distance to the computations will have an impact on both latency and error rate. By having a local fog node perform some of the computations, the distance to the computing platform will decrease, and therefore the latency and error rate as well. Our measurements in table 4.1 show that the latency in a Fog architecture is significantly lower than for a Cloud architecture. The Fog architecture has the greatest difference between running on fiber and 4G. This most likely because our 4G hotspot is not as good as the fiber router at forwarding messages. The latency tests for this architecture show mixed results. The fiber connection has lower latency than the limit for real-time control while the 4G connection is slightly above. This means a fog architecture could be used for controlling a real-time drone system but it will be sensitive to environmental factors.

Since the system operates on a local network, we can assume that the error rate will be close to zero. Overall, the error rate of this architecture will be lower than the Cloud architecture for the data in Group B and C since it only has to travel to the local node. Data from Group A will have to be sent to the cloud for storage since Greengrass does not support databases. This means the error rate of data from Group A stays the same as in the Cloud architecture. The impact of an error, for all data groups, is the same as we discussed previously in our Cloud architecture discussion.

In a Fog architecture, we can decrease the amount of data that is being sent to the cloud by using a filter. If a data point has not changed much, the local node can decide whether or not to forward it to the cloud. With a smaller amount of messages being sent one could prioritize a higher QoS to ensure message delivery. This is illustrated in figure 3.4, where it can be seen that the amount of group A data is reduced before it is sent to the cloud. The amount of group B and C data follows the same logic as discussed in the previous section but in this architecture, the data is transmitted between the drone and the fog node. As a result of using a filter, there will be less bandwidth consumption between the fog node and the cloud. The local network still has to be able to handle large quantities of data.

The Greengrass software does not provide support for a full Fog architecture since some computation has to be made in the cloud, e.g database storage. As mentioned previously, a Greengrass Group can be run offline as long as there is a local network. Any messages to the cloud will then be queued and sent when a new connection to the cloud is established. If we weren't using Greengrass, we could have connected a database to the fog node and avoided having to send the data to the cloud for database storage.

Updates of the logic will be made in the cloud but will then have to be deployed to the relevant Greengrass Groups for it to be used. If we have several Greengrass Groups using the logic, the changes have to be deployed to each Group manually which is not very efficient when having a large number of fog nodes. The Fog architecture allows the drones to be lightweight but requires a more powerful fog node. The fog node can be any type of device as long as it has support for the Greengrass software. The biggest downside when implementing a Fog architecture in AWS is that the drone and fog node needs to be on the same network limiting the range of the drones to the reach of the Wi-Fi. This severely limits the area of application of the IoT network. One possibility which could be interesting to explore further would be to let a more powerful drone be the fog node, making the node portable. Having a local fog node will still outweigh strictly cloud solutions in many situations since it will reduce overall latency.

## 5.3 Edge

An Edge architecture performs most of its computations on the device. By not having logic at another location there is no need to send the data over the network which will result in lower latency, no error rate, and no sent data. However, this only applies to data from Group B and C while data from group A still has to be sent to the cloud since Greengrass does not support database storage. This is illustrated in figure 3.6 where group A is the only data going to the cloud. The left-most arrow marked with an A represents the data collected by sensors. This is then filtered and sent to the cloud, which results in the thinner A-arrow. The principles are the same as in the Fog architecture, but in this case, the data only has to be transmitted within the drone.

As seen in table 4.1 there is no difference between running on fiber or 4G since it does not communicate over the network. However, it is also significantly lower than the latency of the other architectures. Both the connections have significantly lower latency than the limit for real-time control, even the highest peaks are below the limit. This makes the Edge architecture highly suitable for controlling real-time drone operations. The latency in this architecture is only the time it takes for the lambdas to execute since there is no transmission time. Because the other architectures were running the same lambdas their latency can be seen as transmission time and edge latency added together. As in the Fog architecture, the data from Group A can be pre-processed on the device to reduce the amount of data sent to the cloud. The data in groups B and C are not at risk for any errors since this data will not leave the device. Errors can still occur but those will not be related to the architecture but rather to the hardware or software.

The edge architecture can be run completely offline, and while it has available memory it can queue messages that will be sent to the cloud when a connection is re-established. Since all computations are made on the device itself, it does not require a local network as the Fog architecture does. Running an Edge architecture would require more powerful hardware on each drone since it needs to meet the requirements for running the Greengrass software. By having a Greengrass Core on each drone, any updates to the software will have to be deployed to each Core manually.

An edge solution is the most suitable for real-time operations in our use case because of the extremely low latency and error rate. However, this will lead to more maintenance since each drone will be its own Greengrass Group. Therefore it is beneficial to host as many non-time-critical functions as possible in the cloud.

# Chapter 6
# Simulation

In this chapter, the IoT Device Simulator will be evaluated based on its ability to simulate drones. First of all, a description of how to use the simulator will be provided. Then we will evaluate how well the simulator can adapt to the different architectures and explore the possibilities of customization. Lastly, we will present our findings and discuss potential solutions to some of the problems that we encountered.

## 6.1 IoT Device Simulator

The first step was to explore the functionality of the IoT Device Simulator. To do this we set up the out-of-the-box solution by launching the IoT Device Simulator stack. This stack was created by using a default template, provided by the Amazon Deployment Guide. A stack is a collection of AWS resources that can be handled as a single unit. In figure 6.1 we can see all the resources, and their relationships, which the IoT Device Simulator template defines.

One can access the simulator through a custom website hosted by Amazon CloudFront, using code from the S3 Bucket (IoT Device Simulator Console). AWS Fargate is running the containers containing the Simulation Engine. The rest of the services are utilities that are required to set up the simulator.

### 6.1.1 Using the simulator

To use the Device Simulator one first need to define a Device Type. As can be seen in figure 6.2 one gets to configure Name, Visibility (to other users), Data Topic (IoT Topic), Transmission Duration, and Transmission Interval. Then one adds their desired attributes, which could be an Integer, String, or Timestamp. For each attribute, you set some kind of MIN and MAX value from which it will pick a random value.

Once the device type is created one can create Widgets of that type. These Widgets represent the IoT devices and one can create at most 100 Widgets at once and have 1000

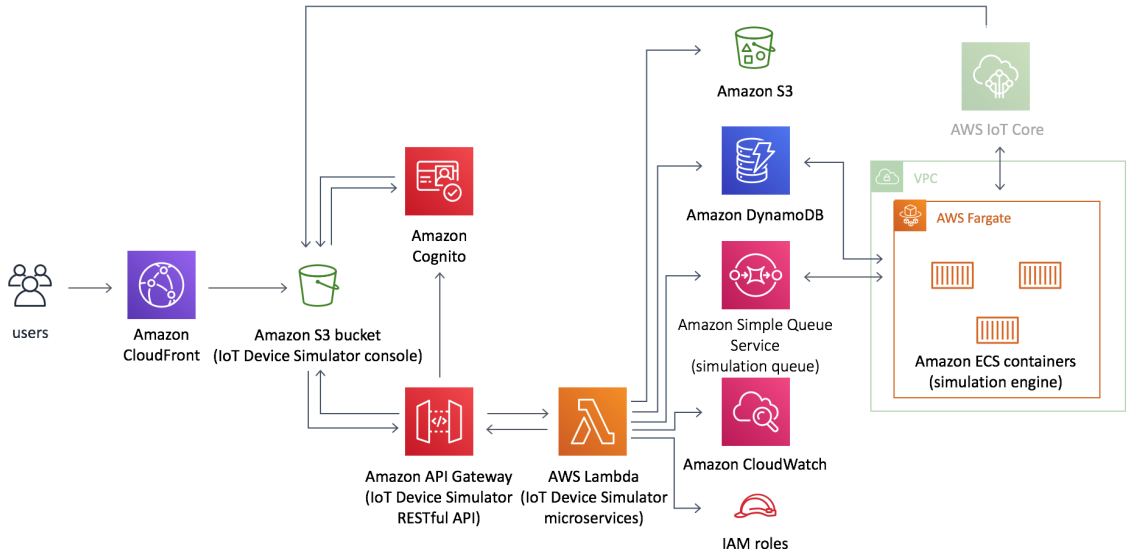



**Figure 6.1:** The IoT device simulator components and their relationships[25]

simulations running at the same time. When simulating a widget, data is published with a given interval and duration, to the Data Topic in the IoT Core. AWS services and other devices can subscribe to the topic in the IoT Core to receive the data published by the widget.

### 6.1.2 Customization

After exploring the possibilities of the IoT Device Simulator we looked into how the simulator could adapt to the different architectures. We realized that the simulator would not be able to simulate the drone described in our use case. To do this, the simulated drone would need to be able to both send messages and receive input. Since the out-of-the-box solution does not provide any support for device input we needed to take a look at the possibilities to customize the simulator. This was a possibility since Amazon provides the source code and some instructions on how to deploy the custom solution[28].

The IoT Device Simulator consists of two parts, the simulation engine, and the console. The simulation engine is the part of the IoT Device simulator which manages the virtual devices and the data they generate. Any changes concerning the data generation itself, e.g attributes, data transmission, or visibility, should be made here. A Docker container is hosting the simulation engine. To make an update to the engine, the changes need to be applied to a new docker image which is then uploaded to AWS. The console consists of a distributable and a template. The template is used when launching the stack, which will create all the required Amazon resources. If you want to add, remove, or change any Amazon resources it should be made to this template. However, this is not something we have looked into since it is outside of our scope. The distributable is the website, both front-end and some back-end. Any changes to how the GUI looks or functions have to be made here.

We were not able to customize the simulator to the extent that the devices could receive input and change the state. What we managed to do was some changes to the simulation engine and console. Specifically, we changed how data was generated for some attributes and

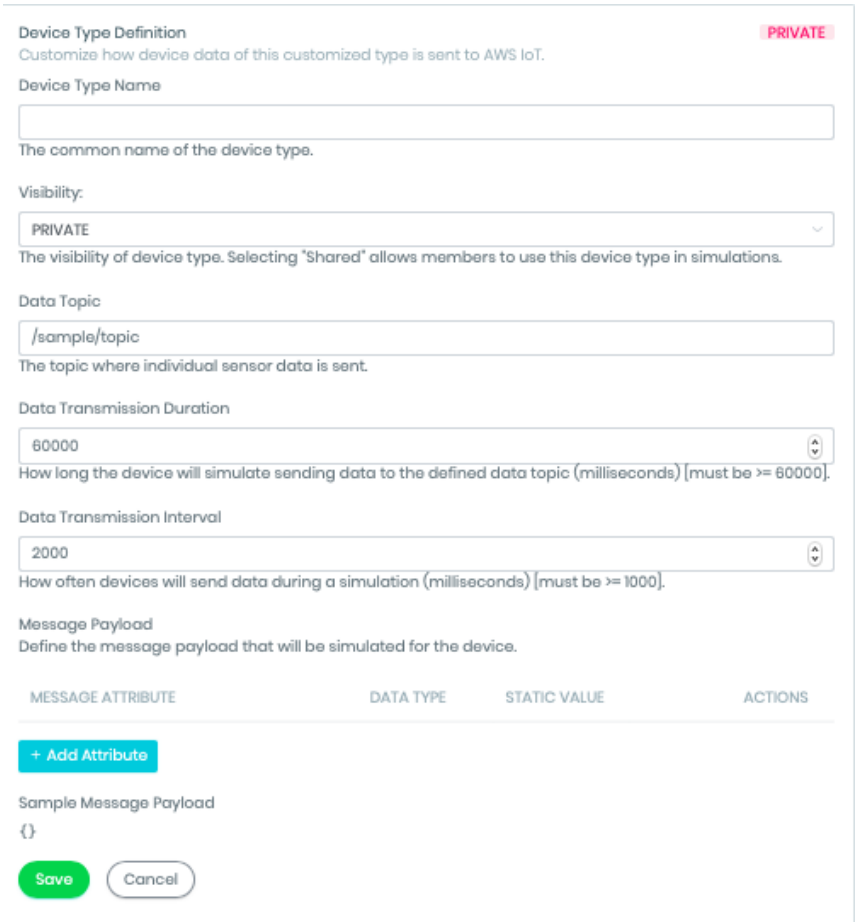



**Figure 6.2:** Screenshot of the IoT device simulator data type configuration page.

some small changes in the GUI. However, it was not enough to simulate a drone from our use case. To test the changes we made, we tried following the guide provided by AWS. We found the guide lacking in some regards but we managed to upload the code and test our changes.

### 6.1.3 Adaptability to different architectures

We were not able to customize the simulator in regards to the previously stated requirements, but we still had to investigate the simulators' ability to adapt to different architectures. What the device simulator offers is a solution to generate a big amount of data without any physical devices. This is a good solution when you want to test out your back-end in the cloud.

When trying to adapt the IoT Device Simulator to the Fog and Edge architectures, we ran into some issues. There was no way to include a simulated device in a Greengrass Group or allow one of them to act as a Greengrass Core since we could not attach a Device Certificate to a widget.

## 6.2 Discussion

Setting up an initial simulation environment was not a particularly difficult task. The Device Simulator was easy to understand and user friendly. There were some easy examples of how to create a custom device type, but it was barely needed since the GUI was self-explanatory.

Customizing the Device Simulator was far more complicated. The source code for the Device Simulator, written in javascript, was not that well documented and it could be difficult at times to find where a change should be made or how the different classes and modules were related to each other. When we were to deploy the changes we tried following the `README` which was included with the source code. We found the instructions lacking in many areas and had to spend a lot of time trying to figure out how it all was connected in AWS. As can be seen in figure 6.1, the Device Simulator consists of many interlocking parts. Once the customized solution is deployed it is easy to update it again. One can upload new simulation engines or distributables and then easily add a changeset to the stack template to make it point to the new versions. This makes it easy if one wants to swap a simulation engine without having to create a whole new Device Simulator stack.

We have previously discussed three different IoT Architectures which could be used for setting up a drone system. The compatibility of these architectures and the Device Simulator varied somewhat. Simulating simple sensors (e.g a thermometer) can easily be done without modification using a Cloud architecture. What the simulator cannot do is keep track of the state of a device or provide the device with input. The input we would like to give to the devices are commands that will change its state, this could be a new route which the device then needs to follow. In a Fog or Edge architecture, the logic is run on a Greengrass Core. To publish data to a Greengrass Core you need to set up a subscription between an endpoint and a resource in the Greengrass Group. Preferably we would want to be able to add the simulated devices to the Greengrass Group and set up a subscription between them directly, but this is not possible. An alternative solution is to publish the data from the simulator to the IoT Core and set up a subscription between the IoT Core and the endpoint. However, this is only relevant for the Fog architecture because there are no other devices to simulate other than Greengrass Cores in an Edge architecture.

In our architectures, we used Greengrass Cores to run logic outside of the cloud. Therefore to fully simulate a Fog or Edge architecture we would need to simulate a Greengrass Core. As mentioned in section 2.6.2 a Greengrass Core needs to contain a device certificate to connect to AWS. For the Core to function it needs to have the Greengrass software installed and running. However, the device simulator does not support adding certificates or installing software on its devices which makes fully simulating a Fog or Edge architecture impossible.

By customizing the Device Simulator we hoped to make it more compatible with our three IoT Architectures. To improve the Cloud architecture simulation we wanted to add functionality for the devices to receive input. Unfortunately, we discovered that the SDK used for the IoT connection did not support a subscription function. So for the devices to be able to receive input we would need to use a new SDK. This would mean we'll have multiple connections to the IoT Core which would work but would not be ideal. Preferably, all communication would go through a single connection but this would mean a lot of code would need to be replaced. Even if you choose to not replace the other SDK, adding input functionality is not simple. We tried to add functionality for input but found it too complicated

for us to implement. We are no javascript experts and without rigorous documentation, the code was simply too messy and had too many dependencies for us to get a grip on it.

An important part of the Fog and Edge architectures is the Greengrass Core, but since the Device Simulator does not provide any support for simulating a Greengrass Core we need to look at alternative solutions. Such a solution could be running a Greengrass Core in an EC2 instance or another arbitrary virtual machine. This would not simulate the whole system but would eliminate the need for physical devices while testing.

One thing we missed while working with the Device Simulator was a smooth integration with the IoT Core. We would have liked to see the ability to add simulated devices to the device list in the Core. Right now the simulated devices are contained in the Device Simulator and only use the IoT Core to publish its data generation. Adding them to the IoT Core would mean we could add them to Greengrass Groups and possibly have hybrid Groups containing both physical and simulated devices. The device simulator felt very detached from the rest of AWS and we would like to see it integrated with the rest of the IoT services.

# Chapter 7
# Conclusions

In this chapter, we will answer the research questions stated in chapter 1 and also suggest some topics for future work. Research questions one will be discussed in regards to the data groups presented in section 3.3 and the scenarios provided in the use case in section 1.2. The second research question will conclude the findings and discussion for the IoT device Simulation in regards to the drones functionality and IoT architectures discussed in this report.

## 7.1 RQ1: Which IoT architecture is the most suitable for a fleet of drones in regards to the use case?

We have concluded that the most suitable solution will be a hybrid architecture. This is because the data groups prioritize metrics differently. The data in group A does not require the drone to take action and is therefore not time-critical. Using Edge computing would in this case be unnecessary and letting the data go to the cloud directly would be a better solution. However, when the number of devices increases the network traffic will increase as well, this will over time lead to network congestion and overloaded cloud severs. To solve this and offload the cloud, fog nodes can be implemented to take some of the workloads. Some of the instructions sent as part of group B will be a direct response to data in group C, which means they need to have the same computing model. Since the data in both of these groups can be considered time-critical, the distance to the computations should be minimized to reduce latency. The best solution to this is to use Edge computing which eliminates transmission latency.

To sum up, a hybrid solution combining Fog and Edge computing would be optimal for handling the data in our use case. This solution would provide computational power to the device to handle time-critical autonomous functions while reducing the communication to the cloud by filtering and aggregating data in a fog node.

Unfortunately, implementing this in AWS is not possible due to the limitations of Greengrass. In our implementation, the Fog architecture is a Greengrass Group consisting of a Greengrass Core acting as a fog node and a device acting as a drone. The Edge architecture consists of a single Greengrass Core acting as a drone. To combine the two solutions we would like to add an Edge Core to the same Greengrass Group as a Fog Core. However, this is not possible since AWS only allows one Greengrass Core per Greengrass Group.

Instead, what we can do is use Edge computing for the most time-critical computations and utilize the cloud for the rest. The reason why not all computations are made in the edge devices is that we want to keep the devices fairly lightweight even though network traffic will increase. This will both ease maintenance but also keep the costs of the drones down.

Another limitation of Greengrass is that the Core and devices in the same Group have to be connected to the same local network. Since the scenarios presented in section 1.2 involves covering great distances, having the drones on the same network as a fog node would likely not be possible. This makes the cloud and edge combination more suitable for these scenarios.

## 7.2 RQ2: Can the AWS IoT Device Simulator be used to simulate drones for the purpose of testing?

To simulate a drone, the AWS IoT Device Simulator is required to provide support for creating a virtual device that can generate data and receive input. Based on this input the device should be able to change its behavior and output accordingly. However, the IoT device simulator can only generate large quantities of data and does therefore not provide enough features to simulate a drone completely. We do not consider the IoT Device Simulator to be sufficient to simulate a complete system for the purpose of testing.

Not all architectures are compatible with the IoT Device Simulator, not even for generating data. A Cloud architecture will be the most compatible since it has the best integration with IoT Device Simulator. Because of Greengrass, neither the Fog nor Edge solutions can be integrated with IoT Device Simulator. Instead, an EC2 instance can be used to simulate a Greengrass Core. In a Fog solution, it is not enough to only simulate the Greengrass Core, you also need devices connected to the Core. This does not apply to the Edge solution since the Greengrass Core is the drone, and thereby making it easier to simulate with an EC2 instance.

There are ways to customize the simulator, but we only managed to change small parts of the data generation and GUI. Making a greater change such as adding input functionality would be very complicated and time-consuming since the simulator source code is complex and not very well documented.

## 7.3 Ethical and Societal Impact

One of the goals of this thesis was to make the process of implementing a system of drones easier by utilizing a cloud platform. This could lead to drones becoming more accessible to the public and expand their area of application. Drones are highly debated when it comes to privacy and espionage and by making it more accessible there is a risk for drones being used

for nefarious purposes. Individuals could use drones to stalk other individuals. Companies and governmental institutions could utilize drones to track citizens.

Using simulation for the purpose of testing could on the other hand lead to companies not buying drones for the sole purpose of testing their IoT systems. With reduced demand for drones, fewer drones could be manufactured which will decrease the impact on the environment.

## 7.4 Future work

This section will bring up areas of research that could be interesting to look into as a continuation of this thesis work.

### 7.4.1 Simulation

Since we did not succeed in simulating drones in the IoT Device Simulator, we would suggest conducting further research into both the IoT Device Simulator and other potential simulation solutions. We believe there is greater potential in customizing the IoT Device Simulator that we were not able to investigate because of a lack of experience and time.

With IoT devices becoming more complex, there is a great need for a simulator that can perform these simulations. It is not only the devices that become more complex but also the IoT architectures and systems. They would also benefit from the opportunity to be able to be tested with the help of simulation.

### 7.4.2 IoT Architectures

IoT architectures are interesting concepts and could be explored further in the context of smart devices and drones. There were some major limitations regarding AWS Greengrass which limited the possibility to set up a Fog architecture. Further research could be conducted into other Cloud platforms and their IoT solutions for Fog computing.

Although Greengrass was not the best for our Fog computing needs, it offers a lot more functionality than we had time to explore such as support for local machine learning inference. Some further research into the possibilities of using Greengrass to make a drone autonomous could, therefore, be interesting.

# References


[1] Ericsson. *Ericsson Mobility Report November 2019*. May 18, 2020. URL: https://www.ericsson.com/4acd7e/assets/local/mobility-report/documents/2019/emr-november-2019.pdf.

[2] S. Ranger. *What is cloud computing? Everything you need to know about the cloud, explained*. Mar. 11, 2020. URL: https://www.zdnet.com/article/what-is-cloud-computing-everything-you-need-to-know-from-public-and-private-cloud-to-software-as-a/.

[3] I. Kholod, N. Plokhoy, and A. Shorov. "Distributed Measurement Data Gathering about Moving Objects". In: *Wireless Communications and Mobile Computing* (2017).

[4] A. Devos, E. Ebeid, and P. Manoonpong. "Development of Autonomous Drones for Adaptive Obstacle Avoidance in Real World Environments". In: *21st Euromicro Conference on Digital System Design* (2018).

[5] M. Alwateer and S. W. Loke. "On-Drone Decision Making For Service Delivery: Concept And Simulation". In: *IEEE International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops)* (2019).

[6] S. Li, L. Da Xu, and S. Zhao. "5G Internet of Things: A survey". In: *Journal of Industrial Information Integration* 10 (2018), pp. 1–9.

[7] OASIS Technical Committee. *MQTT Version 5.0*. June 3, 2020. URL: https://docs.oasis-open.org/mqtt/mqtt/v5.0/mqtt-v5.0.pdf.

[8] OASIS Technical Committee. *MQTT Version 3.1.1*. Apr. 7, 2020. URL: http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.html.

[9] openHAB. *MQTT Things and Channels Binding*. June 12, 2020. URL: https://www.openhab.org/addons/bindings/mqtt.generic/.

[10] International Civil Aviation Organization. "Unmanned Aircraft Systems (UAS)". In: Circular 328 (2011).

[11] R. Nouacer, H. Espinosa Ortiz, Y. Ouhammou, and R. Castiñeira González. "Framework of key enabling technologies for safe and autonomous drones' applications". In: *22nd Euromicro Conference on Digital System Design (DSD)* (2019).

[12] Raspberry Pi Foundation. *Raspberry Pi Zero W*. Mar. 11, 2020. URL: https://www.raspberrypi.org/products/raspberry-pi-zero-w/.

[13] G. Caiza, M. Saeteros, W. Oñatea, and M. V.Garcia. "Fog computing at industrial level, architecture, latency, energy, and security: A review". In: *Heliyon* 6.4 (2020).

[14] X. Liu, T. Zhang, N. Hu, and P. Zhang Yu Zhang. "The method of Internet of Things access and network communication based on MQTT". In: *Computer Communications* 153 (2020).

[15] H. Hong. "From Cloud Computing to Fog Computing: Unleash the Power of Edge and End Devices". In: *IEEE International Conference on Cloud Computing Technology and Science (CloudCom)* (2017).

[16] TipsMake. *What is fog computing?* June 12, 2020. URL: https://tipsmake.com/fog-computing-what-is-fog-computing.

[17] W. Yu, F. Liang, X. He, W. Grant Hatcher, C. Lu, J. Lin, and X. Yang. "A Survey on the Edge Computing for the Internet of Things". In: *IEEE Access* 6 (2017).

[18] N. Ky Giang, R. Lea, M. Blackstock, and V. C.M. Leung. "Fog at the Edge: Experiences Building an Edge Computing Platform". In: *IEEE International Conference on Edge Computing (EDGE)* (2018).

[19] Amazon Web Services. *IoT Core*. May 16, 2020. URL: https://aws.amazon.com/iot-core/.

[20] Amazon Web Services. *MQTT*. Mar. 11, 2020. URL: https://docs.aws.amazon.com/iot/latest/developerguide/mqtt.html.

[21] Amazon Web Services. *IoT Core*. May 16, 2020. URL: https://aws.amazon.com/greengrass/.

[22] Amazon Web Services. *IoT Core*. May 16, 2020. URL: https://aws.amazon.com/lambda/.

[23] Amazon Web Services. *IoT Core*. May 16, 2020. URL: https://aws.amazon.com/ec2/.

[24] A. Maria. "Introduction to Modeling and Simulation". In: *Proceedings of the 1997 Winter Simulation Conference* (1997).

[25] Amazon Web Services. *IoT Core*. May 16, 2020. URL: https://aws.amazon.com/solutions/implementations/iot-device-simulator/.

[26] G. Yang, X. Lin, Y. Li, H. Cui, M. Xu, D. Wu, H. Rydén, and S. Bin Redhwan. "A Telecom Perspective on the Internet of Drones: From LTE-Advanced to 5G". In: (2018).

[27] Amazon Web Services. *AWS IoT device and mobile SDKs*. June 16, 2020. URL: https://docs.aws.amazon.com/iot/latest/developerguide/iot-sdks.html.

[28] AWSlabs. *IoT Device Simulator source code*. June 15, 2020. URL: https://github.com/awslabs/iot-device-simulator.

**EXAMENSARBETE** Drones in the Cloud: A Study of IoT Architectures and Simulation in AWS
**STUDENT** Felicia Hedström och Anton Gudjonsson
**HANDLEDARE** Ulf Asklund (LTH), Christian Eriksson (Dewire)
**EXAMINATOR** Flavius Gruian (LTH)

# Är det en fågel? Är det ett flygplan? Nej, det är en drönare!

POPULÄRVETENSKAPLIG SAMMANFATTNING **Felicia Hedström och Anton Gudjonsson**

Drönare har kommit att bli alltmer populära i samhället. För att göra det mer tillgängligt för företag och privatpersoner att nyttja drönare måste det bli enklare. Detta examensarbete tar fram den bästa arkitekturen för bygga nätverk av drönare.

Samtidigt som IoT enheter blir mindre och mindre ökar även deras kapacitet och beräkningsförmåga. Genom att placera logiken för dess system på olika ställen så kan man öka prestandan. Genom att till exempel placera all logik i molnet så minskar underhåll och konfiguration av enheter, men till ett pris av sämre prestanda så som fördröjningar och fel.

Under vårt arbete har vi fokuserat på självstyrande drönare och var logiken bör placeras för att systemet ska fungera så bra som möjligt. Idag finns ingen best practice för hur ett sådant system ska se ut och därför har vi utforskat olika arkitekturer för dessa IoT nätverk. Resultatet av vårt arbete är en rekommendation för hur ett nätverk med drönare bör utformas med fokus på enkelhet utan att behöva göra avkall på prestandan. För att göra rekommendationen var vi först tvungna att dela upp det data en drönare kan tänkas skicka, i olika grupper. Dessa grupper är baserade på vilka krav som ställs på data, exempelvis att ett avvärjningskommando måste nå drönaren snabbt samtidigt som systemet måste kunna garantera att det når den. För att göra rekommendationen tittade vi på var och en av datagrupperna och tilldelade varje grupp den arkitektur som passade bäst. Detta resulterade i att den slutgiltiga rekommendationen för helhetslösningen blev en kombination av arkitekturer beroende på vilket data som skulle skickas.

Vi hoppas att fler företag, med hjälp av vårt arbete, enkelt ska kunna bygga system för självstyrande drönare och att de därigenom kommer kunna nyttjas i samhället. Några användningsområden skulle kunna vara paketleveranser, temporära Wi-Fi hotspots och sökinsatser.

2016 lanserade Amazon den första versionen av sin helt autonoma leveransdrönare Amazon Prime Air. Under sommaren 2019 kom den andra versionen, men fungerar dock bara på ett antal utvalda platser. Om du är intresserad av att se mer hur detta fungerar så kan du kolla på detta YouTube-klipp: `https://youtu.be/3HJtmx5f1Fc`