



University of
Zurich^{UZH}

Optimizing the Collection and Processing of Wi-Fi Probe Requests

Jonas Brunner
Zurich, Switzerland
Student ID: 17-709-650

Supervisor: Bruno Rodrigues, Eder Scheid, Dr. Thomas Bocek
Date of Submission: June 16, 2020

Abstract

Smartphones and other Wi-Fi devices are ubiquitous nowadays and almost everyone carries such a device with them throughout the daily routine. As Wi-Fi devices emit different signals, even when not connected to a Wi-Fi Access Point, lots of data can be collected by passively listening to the ongoing network traffic. The collected data can be used to gain insights into the behavior patterns of the people carrying Wi-Fi-capable devices. This so-called passive Wi-Fi tracking has found different practical applications, for example in the context of marketing.

This study presents a solution to optimize the collection and the processing of Wi-Fi probe requests. The proposed solution reduces the bandwidth requirements of the capturing devices by introducing an aggregation scheme. A scalable and extendable cloud architecture is presented to store and process the collected data. Additionally, the cloud architecture is used to manage the devices remotely, thereby enabling flexible deployment and adaption to changing environments.

Contents

Abstract	i
1 Introduction	1
1.1 Motivation	1
1.2 Description of Work	2
1.3 Thesis Outline	2
2 Related Work	3
3 Design	5
3.1 Collecting Probe Requests	6
3.1.1 Fetching Configuration	6
3.1.2 Capturing Data	6
3.1.3 Filtering and Aggregating Data	8
3.1.4 Preparing for transmission	9
3.2 Data transmission	10
3.3 Storing and Processing Data	12
3.3.1 Storage Requirements	12
3.3.2 Optimizing costs	15

4	Implementation	17
4.1	Sensor	17
4.1.1	Python	17
4.1.2	AWS Integration	18
4.1.3	Capturing Probe Requests	20
4.1.4	Parsing PCAP files using Python	21
4.1.5	Serializing Data	24
4.1.6	Installation & Start	25
4.2	AWS Implementation	26
4.2.1	IoT Core	26
4.2.2	Kinesis Data Firehose	27
4.2.3	AWS Athena	28
4.2.4	Process measurements	29
4.2.5	Real-Time Analytics	32
5	Evaluation	33
5.1	Optimization of Bandwidth Usage	34
5.1.1	Aggregation	34
5.1.2	Serialization	35
5.2	Costs	35
6	Final Considerations	39
	Abbreviations	47
	List of Figures	47
	List of Tables	49
A	Further code examples	53
A.1	Sensor scripts	53
A.2	AWS implementation	57

Chapter 1

Introduction

1.1 Motivation

The usage of mobile devices such as smartphones has grown rapidly during the past years. The global mobile consumer survey [1] conducted by Deloitte shows, that 92% of the Swiss population own a smartphone, and 97% of them use it on a daily basis. With Wi-Fi enabled, smartphones periodically [2] send requests to detect nearby Wi-Fi Access Points (AP). Therefore, considering the broad usage of smartphones, large amounts of data are produced and sent to any AP within the range of the smartphone. By passively listening to wireless signals emitted by smartphones, specifically probe request frames, it is possible to gain insights about the number of smartphones around, without having installed any software on the device itself. This approach has already been used in the field of live marketing [3] to gain usage and interaction insights for activities like trade shows or tastings.

In web analytics, KPI's (Key Performance Indicators) like the number of visits or the number of returning visitors are straightforward to collect with tools such as Google Analytics [4]. In contrast to passive Wi-Fi tracking, web analytics tools have access to the user's device in the form of cookies that can be stored on the device itself. Those cookies often contain a unique identifier that allows for a re-identification of users at a later point in time to measure, for example, the number of returning visitors.

In passive Wi-Fi tracking, the MAC address, as a unique identifier, is frequently affected by MAC address randomization and it requires further processing of the data to gain insights similar to web analytics. However, the processing of this data exceeds the computational power of most IoT devices which can be used to collect probe requests. Furthermore, to aggregate data over multiple sensors, processing of the data on a central server is required. While solutions such as [5, 6] exist, they often require a significant amount of effort to configure and are not fully optimized towards data transmission. This makes them inflexible for temporary measurements without existing network infrastructure as they need to transmit a significant amount of data which is often expensive or not even possible when relying on a cellular connection. Therefore, alternative solutions are needed

to make passive Wi-Fi tracking with IoT devices easy to deploy and flexible to configure while minimizing the bandwidth requirements.

Other existing setups [7] do only transmit their data at the end of the day or process the data overnight [3]. Using such an approach, insights are always delayed and no information about the current number of devices in sensor reach is available. To enable real-time insights, it is therefore needed, to align the processing of the collected probe requests with current best practices considering streaming data [8] to make passive Wi-Fi tracking close the gap to the widely spread tracking in the web.

1.2 Description of Work

The aim of this Independent Study is, at an initial stage, to understand the business impact of collecting data in a cloud architecture. Although cloud applications are designed to scale, collecting lots of data has an impact on the costs. Thus, a key analysis is to define the design of the architecture in a scalable, yet cost-efficient way.

A custom solution to collect probe requests using IoT devices needs to be developed. The solution needs to optimize data transmission. It is therefore needed to research which data format and protocol integrates best with the cloud architecture. A further task is to filter and aggregate incoming sensor data in a scalable way, based on the Amazon Web Services [9] (AWS) infrastructure. AWS is one of the key technologies for scalability. Thus, it needs to be evaluated as well with respect to scalability and pricing.

Another aspect is the MAC randomization often present in mobile devices. Newer devices perform Wi-Fi probing with random MAC addresses to make tracking more difficult. However, each vendor implements the randomization differently. Thus, this study includes research on how different manufacturers perform MAC address randomization, in order to ensure that enough data is collected to derandomize the MAC addresses at a later point.

1.3 Thesis Outline

Chapter 2 describes different approaches for passive Wi-Fi tracking presented in recent literature. **Chapter 3** presents the design decisions of the proposed solution. This includes the aggregation scheme and the cloud architecture. **Chapter 4** shows the implementation details of the presented design and **Chapter 5** evaluates the implementation in terms of costs and bandwidth reduction based on two example scenarios. **Chapter 6** concludes this independent study with the final considerations.

Chapter 2

Related Work

Position tracking has become ubiquitous with positioning systems such as Global Positioning Systems (GPS) or Galileo. While GPS-based systems need a clear line of sight to provide accurate results, Indoor Positioning Systems (IPS) often use technologies such as Wi-Fi, Bluetooth, or cameras. In the area of Wi-Fi position tracking, there are many approaches based on active tracking, where devices need to be associated with an AP [10, 11, 12].

Passive Wi-Fi tracking methods are often based on probe request capture. The amount of probe requests emitted by different smartphones has been measured in [2]. In an experimental setup, they measured the number of emitted probe requests with different device configurations. The authors discovered that Android devices send probe requests with the highest frequency and concluded that this can be used to track the devices with high precision. It was also found that certain devices send probe requests, even if they are connected to an AP or are in flight mode. However, it has to be noted that the probing behavior can vary significantly based on the operating system of the given device.

[5] and [7] measured the occupancy of university campuses using passive Wi-Fi monitoring. Both works used sensors that collect and send data to a centralized server and provided useful results considering room and building occupancy. However, they were both based on manual setup. [7] utilized manually configured Wi-Fi Pineapples to run a `Tcpdump` over their web interface. Still, the data was transmitted to the server only once a day, which did not allow real-time measurements.

For our solution, [6] presented a more useful approach to collect and analyze probe requests. The proposed solution consists of two subsystems, one for device administration (Probr-Core) and one for Wi-Fi data analysis (Probr-Analysis). Probr-Core allows remote access to the sensors to configure measurements. Similar to [7], Probr[6] also collects data using `Tcpdump`, however, the generated PCAP files are sent to the server in real-time. Still, because of redundant data in the PCAP file, bandwidth requirements are high compared to solutions [13] that only forward certain elements of a probe request to a central server.

Musa and Eriksson [13] present a system to passively track uninstrumented smartphones using off-the-shelf Access Points (AP). The collected data is forwarded to a central server

on a per-second basis using either a Wi-Fi or Ethernet uplink. The authors propose different approaches to estimate the location of devices on the central server based on statistical analysis and the received signal strength of collected packets. To send the data in real-time, they only forward the maximum received signal strength and the hashed mac address. This reduces the used bandwidth but does not take MAC address randomization into account, as the collected data does not allow a deanonymisation. Still, [13] shows that passive Wi-Fi tracking can be used for relative accurate positioning with comparatively low costs.

Most solutions presented in literature strive to optimize algorithms to reach higher positioning accuracy while taking the underlying infrastructure such as ethernet or Wi-Fi connection for granted. This Independent Study focuses on optimizing the passive collection of Wi-Fi probe requests, to make it scalable and flexible to deploy.

Chapter 3

Design

The main goal of this study is to present an approach that optimizes the collection and processing of Wi-Fi probe requests. Such an approach involves a series of steps toward the collection of data and because capturing devices often have restricted hardware, it needs to optimize the transmission of data as well. As discussed in Chapter 2, existing solutions are typically based on fixed infrastructure or require manual setup. We hereby propose a solution that optimizes the whole process of collecting and processing Wi-Fi probe requests by providing the following benefits.

- **Flexible Deployment:** The sensors which collect data are adaptable to changing environments such as location or type of measurement. This means that the configuration on the sensor can be easily managed, without having direct access (*e.g.* an SSH connection) to the device itself.
- **Allow scalability in the future:** The proposed solution scales over multiple measurements with several sensors without introducing bottlenecks.
- **Reliability:** Using constrained IoT devices, the solution ensures that the bandwidth used lies within the bandwidth capacity of the device.
- **Cost-efficiency:** While providing scalability, the solution uses hardware and server infrastructure in a cost-efficient way.

Collecting and aggregating Wi-Fi probe requests is a multi-step process. It can essentially be divided into three main parts which are addressed in the following sections.

1. The first part consists of passively collecting Wi-Fi data using IoT devices. This also includes the management and the configuration of the devices to allow flexible deployment and adaption to different environments.
2. The second part consists of the preparation process, including filtering unnecessary data (*i.e.*, layer 2 fields), and aggregating data over an adaptable period. Then, such data can be sent from the IoT device to a server in a bandwidth preserving way.

3. Storing and processing the collected data on a server is the last step and is done by relying on cloud infrastructure in a cost-efficient way.

3.1 Collecting Probe Requests

As defined in the IEEE 802.11 norm [14], Wi-Fi devices probe for nearby Wi-Fi networks by sending out probe request management frames. While most Wi-Fi interfaces are used to connect to an Access Point (AP), they can also be set to monitor mode, to listen to the ongoing Wi-Fi traffic including the probe requests. This means that collecting probe requests passively is possible with off-the-shelf hardware. However, when capturing probe requests, not all data is relevant and useful for further analysis. To capture and extract the most relevant elements of probe requests efficiently, this study proposes a simple model based on the following steps:

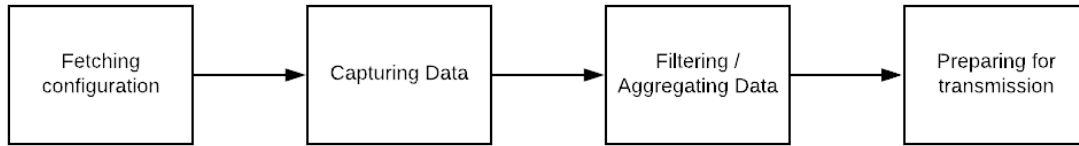


Figure 3.1: Steps to collect Wi-Fi Probe Requests

3.1.1 Fetching Configuration

The proposed solution relies on a device configuration that is stored on an external server. This allows remote configuration of the capturing device and the flexible adaption to changing measurement environments. Therefore, the capturing device always requests its configuration from the central server at startup. The configuration itself contains device-specific parameters like the ID of the device but also measurement specific parameters, which are addressed in section 3.1.2. If there is no Internet connection available at startup, the sensor falls back to the locally stored configuration and updates the configuration at a later point in time.

3.1.2 Capturing Data

Probe requests essentially contain two elements: An SSID to probe for, which can also be replaced by a wildcard, and the supported rates of the device sending the probe request. When capturing probe requests, additional information such as radiotap headers [15] gets appended by the wireless adapter. Radiotap is a de facto standard and includes information such as Received Signal Strength Intensity (RSSI), timestamp, or channel

Element	Location	Size
Source MAC Address	Probe Request MAC Header	6 B
RSSI	Radiotap Header	1 B
Timestamp	Radiotap Header	8 B
Sequence Number	Probe Request MAC Header	2 B
Information Elements	Probe Request Frame Body	variable

Table 3.1: Fields of a Probe Request to be captured

frequency. However, it is not efficient to store or send the raw captured data to a central server, as only following elements are interesting for passive Wi-Fi tracking:

The Source MAC Address (when not randomized) is used as a unique identifier, while the RSSI can be used to estimate the distance. Due to privacy reasons, the MAC address is hashed on the capturing device itself and never transmitted as clear text. Sequence Number and Information Elements are captured due to MAC Address Randomization.

Mac Address Randomization

Knowing that the MAC address as a unique identifier poses a privacy risk when sent with every probe request, different manufacturers started to randomize MAC addresses when probing for available Wi-Fi networks. While the type of randomization differs from manufacturer to manufacturer [16], the randomization itself can be detected relatively easily.

A MAC address consists of 6 bytes (octet) and is usually displayed in the following format, with a 2-digit hexadecimal number for each octet:

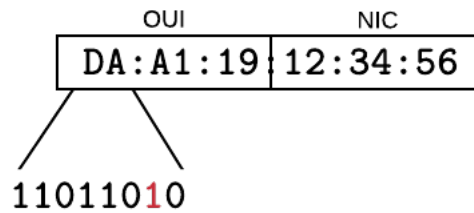


Figure 3.2: Example MAC Address using a CID registered by Google Inc.

The first 3 bytes build the Organizationally Unique Identifier (OUI), which is a vendor-specific identifier. OUIs are managed and assigned by the IEEE Registration Authority [17]. They prevent MAC address collisions to ensure that the MAC address is a unique identifier across devices of different manufacturers. The last 3 bytes are Network Interface Controller (NIC) specific and assigned by the manufacturer.

Each Mac address contains a local flag (second least-significant bit of the first byte), which determines whether the MAC address is used globally as a unique identifier or

locally. Similar to OUI prefixes, the IEEE also sells Company Identifiers (CID) [18], which always have the local bit set and cannot be used for global MAC addresses. Due to the fact that randomized MAC addresses cannot be unique, the local flag always has to be set. However, locally administered MAC addresses are not only used for randomization purposes, but also for services like P2P or Wi-Fi-direct. Nonetheless, the vast majority of locally administered MAC addresses can be assigned to MAC randomization [16].

Derandomization techniques often rely on different information elements of management frames, such as fields for Wi-Fi Protected Setup (WPS) [19]. However, most techniques require extensive computational power, *e.g.*, large hash table lookups [16], which is not feasible for common IoT devices. On the capturing device itself, we assume local MAC addresses as randomized and leave the further processing to be done on a more powerful server than the capturing device. To derandomize MAC addresses at a later point, all information elements of probe requests and the sequence number are also captured and forwarded to the central server.

3.1.3 Filtering and Aggregating Data

Based on the type of measurement, collected data needs to be fine-grained to a greater or lesser extent. In other words, to create a heatmap of the devices nearby, the data must be collected in shorter time intervals. Conversely, when only collecting basic usage statistics such as the number of devices around, the data can be aggregated over a few seconds.

It also has to be noted, that smartphones often send probe requests in a bursty way *i.e.*, multiple requests at once followed by none for a while. Such bursts can contain up to 50 probe requests in one second [2] and are used to probe for different SSID's. Therefore, each probe request in a burst differs only slightly from one other. Fields that change within bursts include the SSID (if no wildcard is used), the sequence number, and, sometimes due to MAC address randomization, the MAC address of the sender. This also implies that for more precise measurements an aggregation period is preferred, to avoid the transmission of multiple almost similar probe requests to a server for further processing.

Measurement Type	Aggregation Duration
Heatmap	1 second
Traffic	10 seconds

Table 3.2: Aggregation duration for different measurement types

Within an aggregation period, for each unique MAC address, only the first probe request is fully stored including the fields listed in Table 3.1. For additional MAC addresses, only the new RSSI is considered. This means, that within an aggregation period multiple RSSI values for a single MAC address are possible. Table 3.1 is therefore extended by an *Average RSSI* value and a *Maximum RSSI* value. Furthermore, we count the number of received packets per MAC address within an aggregation period. For the timestamp, we use the smallest value, *i.e.*, the first received frame within the aggregation period. Due to the fact that the sequence number is only used to derandomize MAC addresses at a later

Element	Frame
MAC Address (hashed)	first processed frame
OUI	first processed frame
Local Flag	first processed frame
Maximum RSSI	variable
Average RSSI	all
Number of frames	all
Timestamp	first received frame
Sequence Number	first processed frame
Information Elements	first processed frame

Table 3.3: Elements to be captured and calculated for a single MAC address within an aggregation period. **Bold** values are aggregated over multiple frames.

point, additional sequence numbers from the same MAC address within an aggregation period are discarded. The bold values in Table 3.3 show the values that are calculated over an aggregation period.

Furthermore, by extracting values like the OUI or the local bit from the MAC address, it is possible to already filter data on the capturing device. This allows us to exclude, for example, frames with OUI from manufacturers that are known to produce more infrastructure-related hardware, such as Cisco. By extracting those values from the MAC address directly on the capturing device, the MAC address can be hashed before getting transmitted to an external source to preserve privacy.

3.1.4 Preparing for transmission

Preparing the data for transmission is important because of two reasons. First, to reduce the number of transmitted packets (and therefore, communication protocol overhead), the capturing device has to send. Secondly, to reduce the bandwidth used for transmitting the captured data by using an aggregation approach. In this approach, we propose to buffer the data within an aggregation period and send one packet containing all aggregated MAC addresses. The scheme is shown in Figure 3.3 with each MAC address containing all the fields from Table 3.3.

Furthermore, data about the sensor which captured the data is included, to be able to map the data to different measurements afterwards. Using this scheme, the number of sent packets does not depend on the number of capture probe requests but on the aggregation duration.

To reduce the used bandwidth, the data is serialized before getting transmitted to a server for further processing.

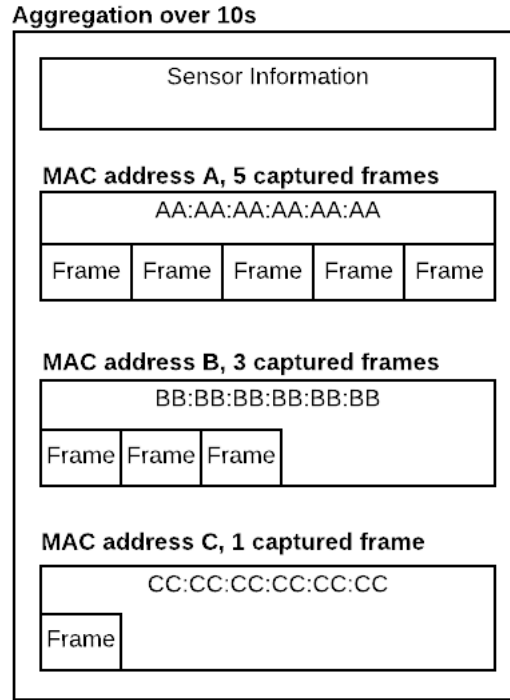


Figure 3.3: Structure of an aggregated packet that is transmitted after an aggregation period.

3.2 Data transmission

There are different approaches to transmit data from an IoT device to a server. This study considers the two approaches displayed in Figure 3.4.

- In the first approach (a), all IoT devices directly connect to a central server. This means, that each device needs to have a working uplink connection whenever it wants to send data. While for permanent installed long-term measurements a device can be connected to the internet via LAN, a cellular connection is needed to enable flexible placement of the devices, without relying on existing infrastructure. This also makes devices independent from one other and allows interchangeability of devices between measurement without additional configuration.
- In the second approach (b), multiple IoT devices share the same uplink using a sink. This omits the requirement of a cellular connection for each IoT device and enables further aggregation/filtering locally when using a computationally more capable device as a sink *i.e.*, a Raspberry Pi. However, since the IoT devices need to know, how to connect to their sink, it requires reconfiguration when interchanging IoT devices between measurements.

Due to the fact, that the aggregation and filtering of our proposed solution do not require much computational power, we use the first approach (a) to transmit the data without

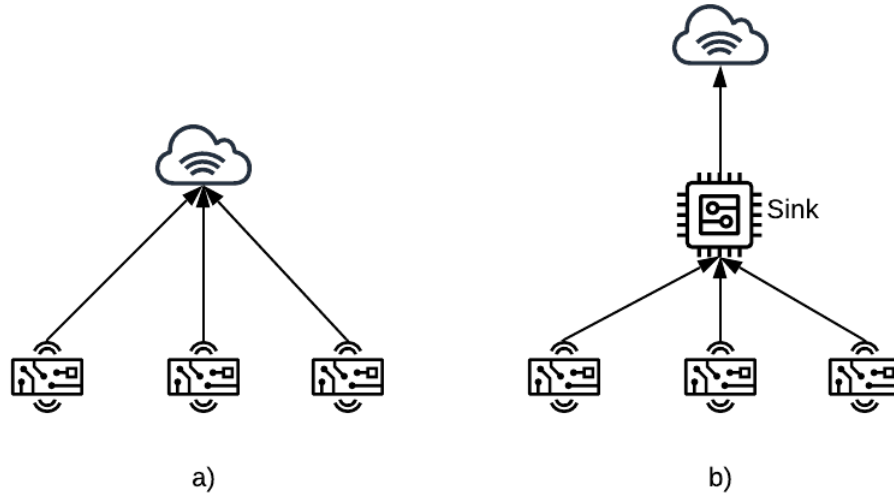


Figure 3.4: Different approaches to transmit data from an IoT device to a server.

a sink and with more flexibility. Nevertheless, since off the shelf hardware boards often don't provide support for cellular connection using a SIM card, we also consider a hybrid approach of using a sink only for sharing the uplink connection.

IoT devices are often constrained with respect to their uplink capacity, therefore the transmission of the data needs to be optimized. As mentioned in Subsection 3.1.4, the captured data is serialized to be sent in a binary format which reduces the overhead for each packet. In contrast, readable data formats like JSON or XML produce a large overhead by always including field names. The structure of a JSON file including the fields (*c.f.*, Table 3.3) for one MAC address already takes up 142 Bytes (*c.f.*, Listing 3.1). Comparing data formats, using a sample of 11 unique MAC addresses, revealed, that the JSON file format takes up 4'113 bytes while the binary format only takes up 2'680 bytes. While the reduction in percentage depends on the number of information elements included in each captured probe request, we can conclude that a binary format reduces bandwidth by about a third.

```

1 {
2   "sensorId":
3   "mac": [
4     {
5       "timestamp":
6       "mac":
7       "rssiMax":
8       "rssiAvg":
9       "numberOfPackets":
10      "sequenceNumber":
11      "localMac":
12      "oui":
13      "informationElements":
14    }
15  ]
16 }
```

Listing 3.1: Overhead produced through readable data formats like JSON

3.3 Storing and Processing Data

Based on the restrictions of IoT devices and to allow measurements with multiple capturing devices, a central server to store and process the data is used. Specifically, this study relies on Amazon Web Services (AWS)[9], as they provide a large palette of computing and storage services including solutions for IoT. Henceforth, a serverless architecture is defined to allow for scalable processing of data, in which resources can be dynamically allocated based on the usage. By using a serverless approach, fixed costs for pre-purchased capacity units can be omitted which keeps costs low at an initial stage. To keep costs reasonable with increasing usage, each used service needs to support the usage patterns of the application in a cost-efficient way.

3.3.1 Storage Requirements

Following usage patterns need to be supported for storing and processing Wi-Fi probe requests:

- **Continuous Input:** Multiple measurements with many capturing devices produce a large number of small data records that are simultaneously sent to a server. This is also known as streaming data [20].
- **Long term storage:** Large amounts of data need to be stored long-term in a cost-efficient way to allow further analysis *e.g.*, using Machine Learning (ML) at a later point in time.
- **Bursty Access:** The data is not often accessed but on-demand in large queries that can span multiple days of sensor data.
- **Real-time insights:** Basic real-time insights during a measurement must be available.

While AWS provides different database options [21], the above requirements conflict with each other in a way that they cannot be fulfilled by a single storage option. This is due to the fact, that cheaper storage has slower access time. Following AWS database options have been considered:

- **AWS Relational Database Service (RDS)**[22] is a scalable cloud service for database engines like MySQL or MariaDB. It provides fast access times and database instances with burst capacity. However, for large amounts of data, the database can get expensive very quickly.
- **AWS Dynamo DB** [23] is a document database with very fast access for single documents. However, the database costs per read/write capacity unit. This makes the database unsuitable for bursty access patterns, as either cost for unused capacity units is caused, or the capacity units have to be scaled up before each large request and scaled down afterward.

- **AWS Simple Storage Service (S3)** [24] is a cheap storage bucket. In combination with AWS Glue[25], it can be used to store data in columnar format, which then can be queried using AWS Athena [26]. While it is cheap to store large amounts of data, queries take multiple seconds and are therefore not usable for time-critical tasks like real-time insights.

As none of the above solutions fulfills all the requirements, we use an architecture which combines the advantages of S3 as cheap storage, and the advantage of RDS with its fast access times by consuming a data stream at different points (*cf.*, Figure 3.5). Per default, the incoming streaming data flows through a Kinesis Data Firehose [27] stream which is used to transform the data back from the binary format and to store it in S3. For performance reasons, Kinesis Data Firehose buffers the data before writing it to S3. This introduces a delay, which is not acceptable for displaying real-time insights. Therefore, Kinesis Data Analytics [28] is used to analyze the streaming data on the fly before the data is written to S3. The aggregated results are then stored in RDS, where they can be queried with fast access times.

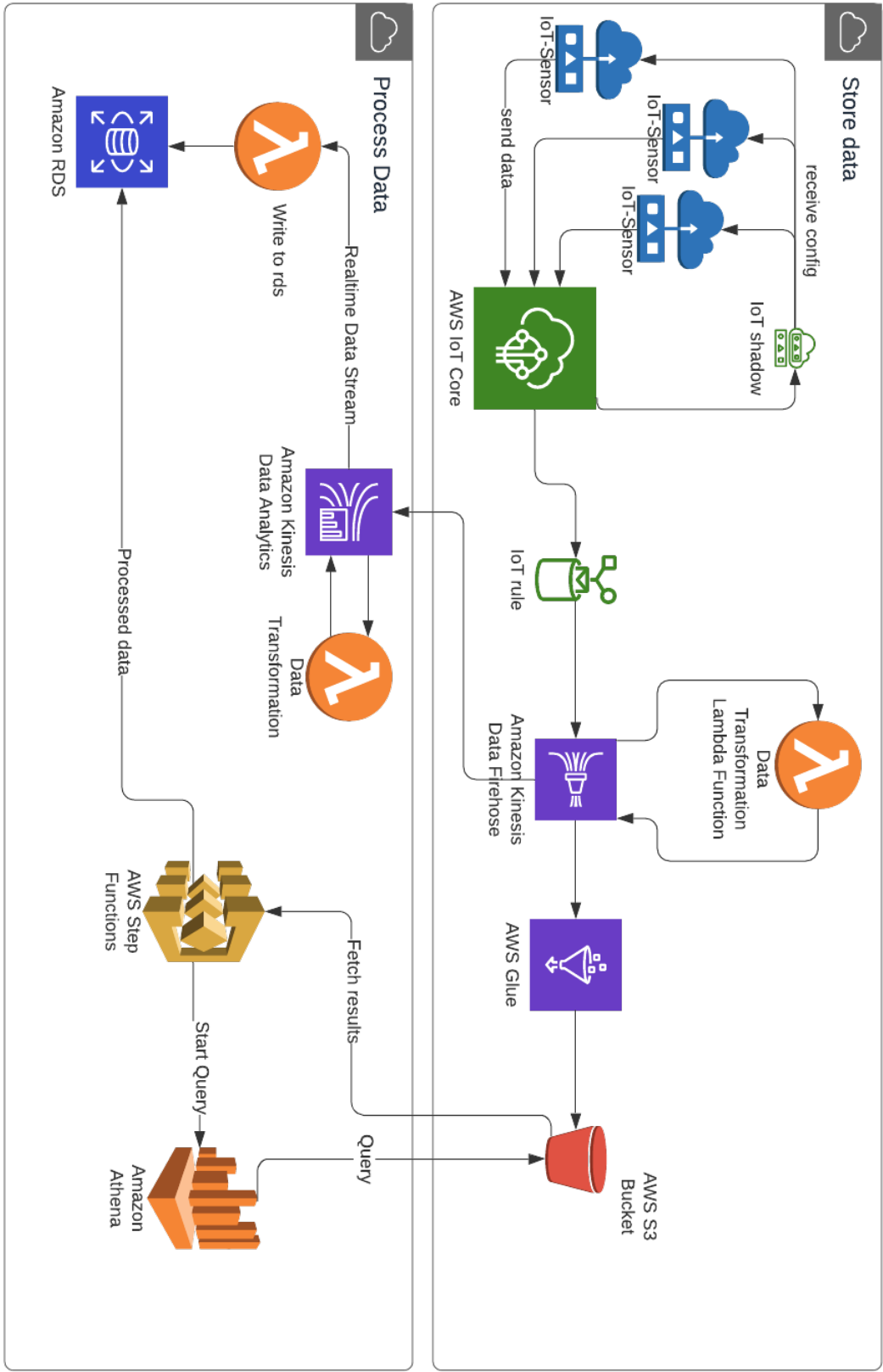


Figure 3.5: AWS Architecture to store and process Wi-Fi probe request data.

3.3.2 Optimizing costs

The above-mentioned solution based on RDS [22] and S3 [24] optimizes the storage of data in a way that raw and large data sets are stored in cheap and slow buckets while the aggregated data can be queried fast from a more expensive RDS database. However, to further optimize costs, we take the exact billing model of following services into account:

- **AWS S3** is billed per GB of storage that is used [29]. We rely on an S3 standard bucket that costs between 0.023\$ and 0.021\$ per GB depending on the amount of storage used.
- **AWS RDS** database instances are billed per hour with additional costs per stored GB [30]. While the price per GB is almost 5 times as expensive as S3, costs are optimized by only storing aggregated results in RDS which take up significantly less space than the raw data in S3.
- **AWS Athena and AWS Glue** are used to be able to query the data from S3. Queries in AWS Athena are billed by the amount of data that needs to be scanned to return the query results [31]. To reduce the data to be scanned, we partition the data catalog stored in AWS Glue by date and time. Furthermore, using columnar storage, we can retrieve only a subset of fields of a row, without scanning all the other fields.
- **AWS Lambda** [32] is used to trigger queries in Athena and to process the query results so they can be stored in RDS. However, as Lambda functions are billed by execution time [33], it is not efficient to have a Lambda function waiting multiple seconds or even minutes for query results from Athena. To keep the execution time, and therefore costs low, AWS Step Functions [34] are used to query and process data stored in S3.
- **AWS Kinesis Data Firehose** is billed by the volume of data that goes through the stream [35]. This means that no further optimization in the architecture itself is possible, but the costs can be reduced by aggregating and filtering the incoming Wi-Fi data on the capturing device itself.

Chapter 4

Implementation

This chapter shows the implementation of the proposed model to optimize the capturing and processing of Wi-Fi probe requests. The implementation consists out of two major parts: (a) The firmware and scripts used on the capturing device to collect data, and (b) the cloud architecture to handle and process the incoming data.

4.1 Sensor

As the design is not dependent on specific hardware, we use a general solution that can be installed on any UNIX-based device with tcpdump [36] and Python [37] installed. For our implementation, we use an Odroid-C1 [38] running Ubuntu 18.04 minimal [39].

It has to be noted, that not all Wi-Fi chipsets support monitor mode, which is needed to capture Wi-Fi data. Therefore, we use the external Odroid Wi-Fi Module 4 [40], which is connected via USB. The external Wi-Fi Module supports monitor mode for both 2.4 and 5GHz. Since the Odroid-C1 does not provide support for cellular connection, we use a second external Wi-Fi Module which is used to connect to the internet.

4.1.1 Python

Python [37] is an interpreted high-level programming language. This means, that the programming language abstracts underlying details such as memory management and allows rapid development. However, Python does not perform as fast as, for example, software written in C or Go, since it is an interpreted language rather than compiled. However, developing embedded software would have exceeded the scope of this study. Furthermore, the filtering and aggregation on the device are not as computationally intensive, so that it can be achieved using Python. For this solution we use version 3.6 of Python.

PIP

Python includes a package-management system called *pip* [41], which allows the installation of additional python packages from the command line. For the sensor implementation,

we use pip to manage the installation of all required python packages by creating a *requirements.txt* file. The file contains a list of all required packets and their corresponding version. When working in a Python virtual environment (VENV), the *requirements.txt* file can be automatically generated by running the following command:

```
python -m pip freeze > requirements.txt
```

This command collects all the installed packages in the VENV and creates an output file as shown in Listing 4.1.

```
1 AWSIoTPythonSDK==1.4.8
2 pathtools==0.1.2
3 protobuf==3.11.3
4 scapy==2.4.3
5 six==1.14.0
6 watchdog==0.10.2
```

Listing 4.1: Automatically generated requirements.txt file.

The *requirements.txt* file allows easy installation of all required packages by running the command `python -m pip install -r requirements.txt`

We use this command in the installation script shown in subsection 4.1.6.

4.1.2 AWS Integration

To connect capturing devices to the cloud architecture hosted on AWS, we use the AWS IoT Core [42] service. IoT Core can be used to manage and authenticate devices. While IoT Core can also be used to distribute incoming data to other cloud services, the focus in this section lies in establishing a connection and retrieving the configuration from IoT Core. Therefore, we use the *AWS IoT Device SDK* [43] for Python to access IoT Core services from our capturing-device. The SDK can be installed using pip and is thus included as a dependency in the *requirements.txt* file.

We use the following two services of the SDK:

- **MQTT Client** is used to publish messages to the IoT Core. It handles the authentication and allows the management of Quality of Service (QoS).
- **MQTT Shadow Client** is used to receive and acknowledge configuration updates from the Device Shadow service of IoT Core. The client automatically checks for differences between the reported configuration state of the device and the desired configuration state managed in the cloud.

For both services, we use a certificate-based connection [44] with certificates generated in AWS IoT core. Three certificates are used to establish an authenticated connection:

- **Root Certificate:** A public-key certificate that allows a device to verify, that it connects to the desired server.

- **Device private-key:** Used to sign messages locally on the device.
- **Device certificate:** A certificate signed by the Root Certificate private key that contains the public key of the device. This certificate is sent to the server and allows the server to verify signed messages by using the included public key.

Fetching the configuration

With all required certificates on the device, we can connect to the Device Shadow [45] service using the AWS IoT Device SDK. At line 9 of Listing 4.2, we define the function to be called whenever the device has established a connection. In our case, the *onConnect* method of the *shadowCallbackManger* instance downloads and writes the configuration to a JSON file and additionally creates a *measurement.conf* file containing bash variables used for the measurement.

Similar to the callback on line 9, we register a callback (*cf.*, line 11) for updates of the configuration while the device is online. The *customShadowCallbackDelta* method updates both configuration files. Furthermore, if the configuration of the ongoing measurement is changed, a reboot of the device is triggered, to start a new measurement with updated configurations. The full code of the *ShadowCallbackContainer* class, which handles all the updates of the Device Shadow, can be found in the appendix (*cf.*, A.1).

```

1 shadowClient = AWSIoTMQTTShadowClient(str(uuid.uuid4()))
2 shadowClient.configureEndpoint(config["host"], config["port"])
3 shadowClient.configureCredentials(rootCa, privateKey, cert)
4 shadowClient.configureConnectDisconnectTimeout(10)
5 shadowClient.configureMQTTOperationTimeout(5)
6
7 deviceShadowHandler = shadowClient.createShadowHandlerWithName(str(
8     config["sensorId"]), True)
9 shadowCallbackManager = ShadowCallbackContainer(deviceShadowHandler)
10 shadowClient.onOnline = shadowCallbackManager.onConnect
11 shadowClient.connect()
12 deviceShadowHandler.shadowRegisterDeltaCallback(
13     shadowCallbackManager.customShadowCallbackDelta)

```

Listing 4.2: Connect to Shadow Client and register callbacks to handle updates

Sending Data

The setup to send data to the IoT Core is almost similar to the Device Shadow connection. Using the certificates stored on the device, a connection is established. For each message to be sent, a QoS can be set which determines the reliability of delivery. The MQTT protocol [46] includes three different QoS levels:

- **QoS 0** delivers the message at most once.
- **QoS 1** delivers the message at least once.
- **QoS 2** delivers the message exactly once.

To guarantee the delivery of messages, we rely on QoS 1. Furthermore, each MQTT message is sent to a specific topic. MQTT topics are UTF-8 strings to filter and organize

incoming data. In the example in Listing 4.3, we send an MQTT message to the topic specified in the configuration file using QoS 1 (*cf.*, line 4).

```

1 mqttClient = AWSIoTMQTTClient(str(uuid.uuid4()))
2 mqttClient.configureEndpoint(config["host"], config["port"])
3 mqttClient.configureCredentials(rootCa, privateKey, cert)
4 mqttClient.publish(config["topic"], bytearray(data), 1)

```

Listing 4.3: Connect to the MQTT broker and publish a message

4.1.3 Capturing Probe Requests

Tcpdump

Tcpdump [36] is a command-line packet analyzer that can be used to capture network traffic. While it is already preinstalled on many UNIX systems, we include the installation command in our installation script to allow compatibility with lightweight Linux distributions.

Tcpdump can not only print the network traffic directly to the console, but it can also write the data to PCAP (Packet capture) files. We use the parameter `-G` to rotate the dump file in order to get a new PCAP file for every aggregation window. The *\$aggregation* variable in listing 4.4 is sourced (*cf.*, line 1) from the configuration file and adapts to remote changes of the configuration. However, as the `tcpdump` command is executed at startup, a reboot of the device is necessary whenever the aggregation duration is updated during a measurement. The reboot is automatically triggered by the python script described in subsection 4.1.2.

The code in Listing 4.4 stores the captured data to the `/capture` folder. It has to be noted, that the capture of probe request is independent of aggregation or filtering and can continue even if a python script crashes.

```

1 . /pySniff/measurement.conf
2 cd capture
3 tcpdump -i wlan0 -e -vv -n type mgt subtype probe-req -G $aggregation -w
   capture-%s.pcap

```

Listing 4.4: Tcpdump to collect and store probe requests to the `/capture` folder

Inotifywait

To allow real-time insights, it is necessary to parse, filter, and aggregate the PCAP files immediately after capture. However, the `tcpdump` process captures probe requests independently of any python script. To inform the python script, we use a file watcher that observes the `/capture` folder and reports file changes *i.e.* new PCAP files that are generated by the `tcpdump` process. For our solution, we use the *inotifywait* command which is included in the *inotify-tools* [47] package.

Listing 4.5 shows the watcher listening for *close_write* events in the `/capture` directory. The *close_write* event is fired each time a file is closed after being opened in writable mode. The `-m` option specifies that the file watcher is executed indefinitely, and thus,

does not exit after the first received event. As a result, the command will immediately return the path and filename whenever a new PCAP file is generated in the */capture* folder.

```

1 inotifywait -m -e close_write --format '%w%f' "${MONITORDIR}" | while
   read NEWFILE
2 do
3     echo -n $NEWFILE > /pySniff/pcapNotify
4 done

```

Listing 4.5: Watching for file changes and writing to a named pipe.

While it would be possible to start the Python script for each generated PCAP file, the startup time of the python script would introduce an overhead, which made measurements with short aggregation periods infeasible. This is due to the fact that importing libraries and establishing a certificate-based connection can take multiple seconds on constrained devices. Therefore, we use a named pipe for Inter Process Communication (IPC) between the file watcher and the Python script. For each generated file, we write the path and filename to the *pcapNotify* pipe (cf., line 3). The pipe can then be consumed by the Python script. The interplay between tcpdump, inotifywait, and the Python script is shown in Figure 4.1.

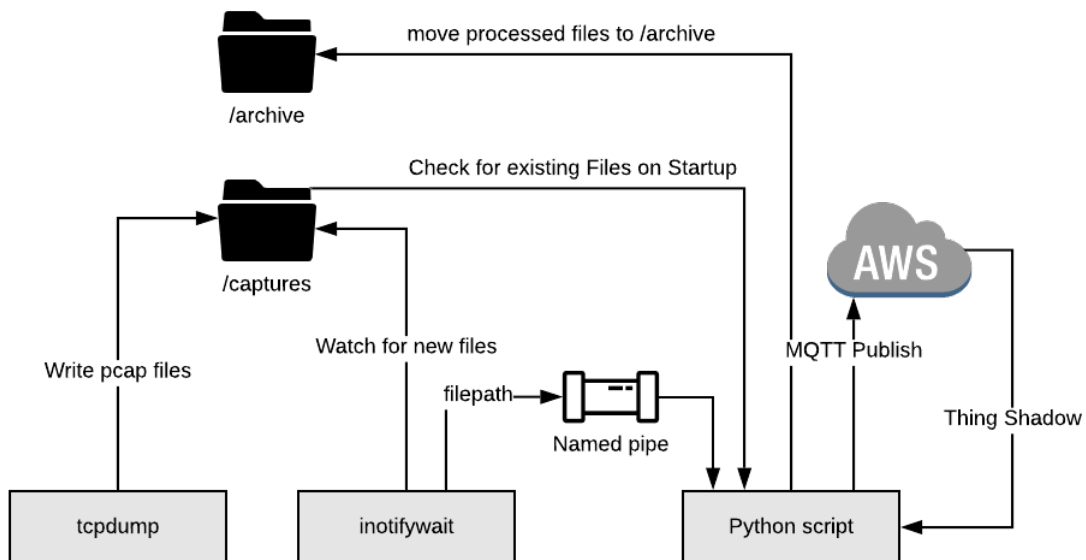


Figure 4.1: Interplay between tcpdump, inotifywait, and the Python script to parse the data.

4.1.4 Parsing PCAP files using Python

While the usage of Python to connect to AWS has already been discussed in subsection 4.1.2, this section focuses on parsing the PCAP files that are generated by the tcpdump, as described in subsection 4.1.3. Using the architecture shown in figure 4.1, we can pipe file paths into a running Python script. This allows us to extend the existing Python script

and run a single Python process which, at the start, connects to AWS and afterwards listens for new PCAP files to parse. The full python script can be found in the appendix (*cf.*, A.1).

Named Pipes (FIFO)

We use the *mkfifo* command to create a First in - First out (FIFO) pipe. The *mkfifo* command is called from within the Python script using the *os* module which provides access to system-dependent functionality. The try block ensures, that the script does not break if the pipe already exists.

Reading the data from the pipe is done by using two while loops. The innermost while loop (*cf.*, Listing 4.6, line 11) reads all data as long as the connection is open. When a message with length 0 is received, the connection is closed and we break out of the inner while loop. The outer while loop (*cf.*, line 9) reopens the connection immediately after it has been closed. Based on the fact, that reading a pipe is a blocking action if there are no writers, the Python script runs indefinitely without using much CPU to continuously read from the named pipe.

```

1  import os
2
3  try:
4      os.mkfifo(FIFO)
5  except OSError as oe:
6      if oe.errno != errno.EEXIST:
7          raise
8
9  while True:
10     with open(FIFO) as fifo:
11         while True:
12             data = fifo.read()
13             print(data)
14             if len(data) == 0:
15                 break
16             parse_file(data)

```

Listing 4.6: Listening to FIFO messages

Scapy

Scapy[48] is a Python library that can be used to send, sniff, and dissect network packets. The library can be installed using pip and is thus included in the *requirements.txt* file. As we collect the data using tcpdump, we mainly use Scapy to parse the PCAP files and to dissect probe request frames in order to easily access different fields. Listing 4.7 shows how Scapy is used to read and parse PCAP files. The *rdpcap* (*cf.*, line 7) method is used to get an iterable of all packets inside a PCAP file. Fields like the sender address (*addr2*) or the RSSI (*dBm_AntSignal*) are automatically dissected by Scapy. The different information elements are located in the *Dot11Elt* layer and accessed using the *getLayer* method.

```

1 from scapy.all import rdpcap
2 from scapy.layers.dot11 import Dot11Elt
3 from capture import Capture
4
5 def parse_from_path(path, sensor_id):
6     try:
7         packets = rdpcap(path)
8         capture = Capture(sensor_id)
9         for packet in packets:
10             mac = packet.addr2
11             rssi = packet.dBm_AntSignal
12             timestamp = packet.time
13             sequence_nr = packet.SC
14             information_elements = bytes(packet.getlayer(Dot11Elt))
15             capture.add_packet(timestamp, mac, rssi, sequence_nr,
16                               information_elements)
17         return capture.serialize()
18     except Exception as e:
19         print("could not parse pcap file" + str(e))
20         return False

```

Listing 4.7: Reading a PCAP file and dissecting the probe request frames with scapy

Aggregating Data

The *parse_from_path* method in Listing 4.7 takes a PCAP file path and a sensor ID as input and returns aggregated data according to the structure shown in Figure 3.3. We follow an object-oriented approach to aggregate the data. For each PCAP file, a *Capture* instance is created. While iterating over the captured frames inside a file, we use the *add_packet* method of the *Capture* class. If the MAC address has not been seen yet, a new *SingleMac* instance will be created, otherwise, the *add_packet* method will increment the *number_of_packets* and recalculate the average RSSI value of an existing *SingleMac* instance.

```

1 from singleMac import SingleMac
2
3 class Capture(object):
4     def __init__(self, sensor_id):
5         self.packets = {}
6         self.sensor_id = sensor_id
7
8     def add_packet(self, timestamp, mac, rssi, sequence_nr,
9                  information_elements):
10         if mac in self.packets:
11             self.packets.get(mac).add_packet(timestamp, rssi)
12         else:
13             self.packets[mac] = SingleMac(timestamp, mac, rssi,
14                                           sequence_nr, information_elements)

```

Listing 4.8: Capture class

As described in the Design chapter, for each unique MAC address in an aggregation period (one PCAP file), we extract information elements, RSSI, sequence number, and MAC specific parts such as the local flag or the OUI. For additional packets of the same MAC address, we only consider the RSSI and the timestamp.

```

1  class SingleMac(object):
2      def __init__(self, timestamp, mac, rssi, sequence_nr,
        information_elements):
3          macString = mac.replace(":", "")
4          self.mac = mac
5          self.rssi = [rssi]
6          self.number_of_packets = 1
7          self.rssi_avg = rssi
8          self.timestamp = int(timestamp)
9          self.sequence_nr = sequence_nr
10         self.ie = information_elements
11         self.localMac = macString[1].lower() in ["2", "3", "6", "7", "a",
        "b", "e", "f"]
12         self.oui = macString[:6]
13
14     def add_packet(self, timestamp, rssi):
15         self.number_of_packets = self.number_of_packets + 1
16         self.rssi.append(rssi)
17         self.rssi_avg = sum(self.rssi)/self.number_of_packets
18         if int(timestamp) < self.timestamp:
19             self.timestamp = int(timestamp)

```

Listing 4.9: SingleMac class

4.1.5 Serializing Data

To optimize the usage of bandwidth, we serialize the data and send it in a binary format. We use Google’s Protocol Buffers (Protobuf) [49], which allow the serialization of structured data. We define the basic structure in the *capture.proto* file (Listing 4.10) and align it to the Python classes used in subsection 4.1.4. Using the Protocol Buffer compiler, data access classes for different languages can be generated. We use the Python data access class to set and access different fields and to serialize and parse the data.

An extract of the usage of the *capture_pb2* data access class is shown in Listing 4.11. The full code for the *Capture* class can be found in the appendix (*cf.*, A.1) First, we populate the *ser_capture* object and use the *SerializeToString* method of the Protobuf data access class to serialize the data. Even though the method is called *SerializeToString*, the string is only used as a container to hold binary bytes, not text.

```

1  syntax = "proto2";
2
3  message Capture {
4      repeated SingleMac mac = 1;
5      required int32 sensor_id = 2;
6  };
7
8  message SingleMac {
9      required int64 timestamp = 1;
10     required string mac = 2;
11     required sint32 rssi_max = 3;
12     required sint32 rssi_avg = 4;
13     required int32 number_of_packets = 5;

```

```

14     required int32 sequence_number = 6;
15     required bool localMac = 7;
16     required string oui = 8;
17     required string information_elements = 9;
18 };

```

Listing 4.10: *.proto* file which defines the structure of the serialized data

```

1 class Capture(object):
2     [...]
3
4     def serialize(self):
5         ser_capture = capture_pb2.Capture()
6         ser_capture.sensor_id = self.sensor_id
7         for key in self.packets.keys():
8             ser_capture.mac.extend([self.packets.get(key).serialize()])
9         return ser_capture.SerializeToString()

```

Listing 4.11: Serializing data using Protobuf

4.1.6 Installation & Start

To install the required packages we use the Advanced Packaging Tool (APT) for Linux based packages and pip for Python packages. Using the two package managers, the installation script can be written with 2 lines of code.

```

1 sudo apt update && sudo apt install python3-pip tcpdump inotify-tools
2 python3 -m pip install -r /pySniff/requirements.txt

```

Listing 4.12: Installation script

The *start.sh* (Listing 4.13) script is executed at startup and orchestrates the start of all required processes to capture probe requests. In our solution, we place the script inside the *rc.local* file. By using the & sign, the processes start in the background and keep running after the *start.sh* script has finished its execution.

```

1 #!/bin/sh
2 cd /pySniff;
3 mkdir -p capture;
4
5 # Start Python script
6 (python3 main.py) &
7 sleep 10;
8
9 #Set up Filewatcher
10 sh /pySniff/scripts/filewatcher.sh &
11
12 #Set Wi-Fi interface to monitor mode and start capturing
13 ifconfig wlan0 down
14 iwconfig wlan0 mode Monitor
15 ifconfig wlan0 up
16 . /pySniff/measurement.conf
17 (cd capture && tcpdump -i wlan0 -e -vv -n type mgt subtype probe-req -G
    $aggregation -w capture-%s.pcap) &

```

Listing 4.13: Start script

4.2 AWS Implementation

The cloud implementation is based on AWS [9] and uses different services for computing and storage. For services like Lambda [32] or Step Functions [34], we specify the needed infrastructure as code using a YAML file. This allows for replication across different regions. However, as this is not the primary goal of this study, we also configure several services directly using the AWS console. All configurations that are done in the console can also be reflected in YAML code in the future. The following subchapters describe the configuration and usage of the different services.

4.2.1 IoT Core

AWS IoT Core [42] is the counterpart of the AWS integration shown in subsection 4.1.2. In this study, we use it mainly for three purposes:

- **Device Management** includes the issuance of certificates to establish a secure connection but also the policies which define the privileges of each device. We grant our devices the right to publish but do not allow them to subscribe to any topic but their device shadow. This prevents security leaks when someone exploits the physical access to a device.
- **Device Shadow** is used to store certain configuration parameters of the device in the cloud. This also means that the configuration can be changed using a REST API. The device shadow consists of a desired and a reported state. To change the configuration of a device, we change the desired state and wait for the IoT device to store the changes locally and report the updated state back to IoT core.

```

1      {
2          "desired": {
3              "sensorId": 1000,
4              "topic": "1000",
5              "aggregationDuration": 10
6          },
7          "reported": {
8              "sensorId": 1000,
9              "port": 8883,
10             "host": "ar55523qdivf5-ats.iot.us-east-2.amazonaws.com",
11             "topic": "1000",
12             "aggregationDuration": 10
13         }
14     }
```

Listing 4.14: Device Shadow state of a capturing device

- **IoT Rules** are used to distribute incoming data to different services. Rules consist of a SQL query statement which selects the data to be forwarded and an action which defines where the data is sent. As the incoming data is in a binary format, we can not select single fields and therefore use `*` as a placeholder to select all data from all (+) topics. This results in the following query: `SELECT * FROM '+'` We forward

the selected data to a Kinesis Data Firehose Stream [27] using an IoT rule action. For our solution, the query is rather general. However, for a future extension with *i.e.*, Bluetooth, the query could be adapted to select from '*{connection}*/' + '. The topic of the existing devices could then be changed to '*wifi/{sensorId}*' using the device shadow.

4.2.2 Kinesis Data Firehose

Kinesis Data Firehose [27] is a service to deliver streaming data to different storage services. We use the Data Firehose to deliver data from IoT Core to an S3 storage bucket and to parse the binary data. As Kinesis Data Firehose is optimized for streaming data, it automatically buffers the data to reduce write and data transform operations.

Data Transformation

For the data transformation from the binary Protobuf format to a readable JSON format, we use a Lambda function with Python 3 as runtime. The data is buffered up to 3MiB for a maximum of 60s. This reduces the number of Lambda invocations while still allowing a reasonable delivery time. Whenever a buffer threshold is reached, a lambda function with an array of records, *i.e.*, a list of MQTT messages, is invoked. Listing 4.15 shows how a single record is parsed. As the data between AWS services is sent in base64 encoded format, we first need to decode the data. The Protobuf data access class and its *ParseFromString* method are used to parse the data back. To omit indefinite retries of empty or bad records, we handle all error cases and return corresponding status codes, *i.e.*, *ProcessingFailed* or *Dropped*.

```

1 def parse_record(record):
2     try:
3         data = base64.b64decode(record["data"])
4         capture = capture_pb2.Capture()
5         capture.ParseFromString(data)
6         result_data = ""
7         for SingleMac in capture.mac:
8             parsed = {
9                 'mac': SingleMac.mac,
10                [...]
11                'information_elements': SingleMac.information_elements
12            }
13            result_data = result_data + json.dumps(parsed) + "\n"
14
15        # Empty pcap
16        if len(capture.mac) == 0:
17            return {"recordId": record["recordId"], "result": "Dropped",
18                    "data": ""}
19
20        # Successfully converted
21        return {"recordId": record["recordId"], "result": "Ok",
22                "data": base64.b64encode(result_data.encode('utf-8')).
23                    decode('utf-8')}

```

```

24     except Exception as e:
25         print("Could not parse record:", e)
26         return {"recordId": record["recordId"],
27                "result": "ProcessingFailed",
28                "data": ""}

```

Listing 4.15: Lambda Data Transformation

Writing to S3

Before writing the data to S3, the data gets buffered again. The data takes up more space after transformation, thus we use a higher buffer size of 64MiB with a maximum buffer time of 60 seconds.

To make the data in S3 queryable, we convert the data records into Apache Parquet [50] format. For automatic data conversion in Firehose, we create a data catalog table (*cf.*, Table 4.1) in AWS Glue which defines the database scheme. Based on the Glue data catalog table, the records are converted from JSON to Apache Parquet using the OpenX JSON SerDe [51].

Column Name	Data Type
mac	string
sensor_id	smallint
timestamp	timestamp
type	string
rss_avg	decimal
rss_max	decimal
sequence_nr	int
number_of_packets	smallint
oui	string
localmac	boolean
information_elements	string

Table 4.1: Glue Data Catalog Table

4.2.3 AWS Athena

Athena [26] is a query service to analyze data in S3 using SQL. With its serverless approach, the pricing is based on a per-query basis [31], respectively the data that needs to be scanned to return query results. To query data, Athena relies on the data catalog defined in AWS Glue [25].

Athena is well suited for large queries as it stores the query results automatically as a CSV in an S3 bucket. Using the CSV files, we can consume the query results at any later point in time without causing additional costs. We, therefore, store the query ID of each executed Athena query in the RDS database, to map query result CSV files to measurement IDs.

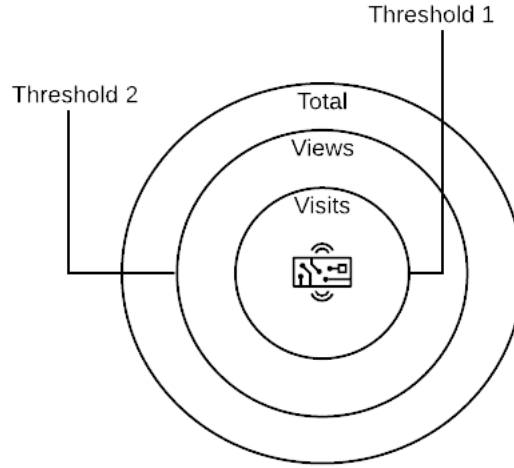


Figure 4.2: Categorization of probe requests into different zones based on the RSSI value.

Compared to other database services, Athena has one drawback: Query results are not returned in a matter of milliseconds but can take up multiple seconds. This problem is addressed by using AWS Step Functions [34].

4.2.4 Process measurements

To gain insights into the collected data, we process the records based on measurements. A measurement is defined by a list of sensors, a time interval, and an ID. It has to be noted, that measurements can also have overlapping sensor lists, *i.e.*, a sensor can be included in multiple measurements at the same time.

For each measurement, we define two RSSI thresholds and categorize the measured probe requests accordingly. Based on the type of measurement, these thresholds can vary and thus need to be adaptable. We denote the captured MAC addresses with RSSI values above threshold 1 as *Visits* and RSSI values above threshold 2 as *Views*. MAC addresses with RSSI values below both thresholds are just counted towards the total. These three categories are calculated over different time intervals, *i.e.*, hourly, daily and weekly.

We use AWS Step Functions[34] to process and aggregate data stored in S3. This includes following steps:

- **Querying data** from S3 based on a list of sensors and a time interval.
- **Aggregating data** over different time intervals, *i.e.* hourly, daily and weekly.
- **Writing data to RDS** where it can be queried with fast access times.

Serverless Framework

The Serverless Framework [52] simplifies the development and deployment of serverless functions. In this study, the Serverless Framework is used to create an AWS CloudFormation Stack [53] which includes all Lambda- and Step functions. The Serverless Framework

thereby handles the packaging and deployment of functions to include all dependencies. For all Lambda functions we use Python 3.8 as runtime. The whole configuration is captured in a *serverless.yml* file and contains not only the function configurations but also environment variables and Virtual Private Cloud (VPC) [54] configurations.

```

1 provider:
2   name: aws
3   runtime: python3.8
4   stage: dev
5   region: us-east-2
6   environment:
7     DB_USER: 'admin'
8     DB_PASSWORD: '###'
9     DB_ENDPOINT: 'is-jonas.cluster-###.us-east-2.rds.amazonaws.com'
10  vpc:
11    securityGroupIds:
12      - sg-9776a6f8
13    subnetIds:
14      - subnet-01c6a33cccc3c8fc9
15      - subnet-0a39ce658086b48d3
16      - subnet-0f89a2a5f847b45a2
17
18 functions:
19   parse-sensor-data:
20     handler: parseData/handler.parse_sensor_data
21     timeout: 10
22     memorySize: 128

```

Listing 4.16: Serverless Framework configuration file

State Machine

As AWS Athena queries run multiple seconds, we use Step Functions to query the data efficiently. Using the state machine (*cf.*, Figure 4.3) we can reduce the billed execution time by introducing wait states. We use the wait state to await query results. Periodically, *i.e.*, every 10 seconds, we check the query status to see if the query has already finished. Depending on the query status, we wait again, abort, or start the data processing.

While the logic of this state machine is simple, it is easy to extend in the future. It would be possible, for example, to implement an additional state to derandomize MAC addresses without changing the existing the implementation of the existing steps. The state machine is also defined in the *serverless.yml* configuration file (*cf.*, A.2) and can be deployed using the Serverless Framework [52].

The individual steps of the state machine consist out of Lambda functions. To read, process, and write data we use the following Python packages:

Boto3

Boto3 [55] is the AWS SDK for Python and provides APIs for various services. We use Boto3 to connect to the AWS Athena service. After starting a query, the returned query ID is passed to subsequent Lambda functions of the State Machine and used to check the query status. It has to be noted, that the query results are not retrieved using the Boto3 Athena client but by directly accessing the CSV file stored in S3.

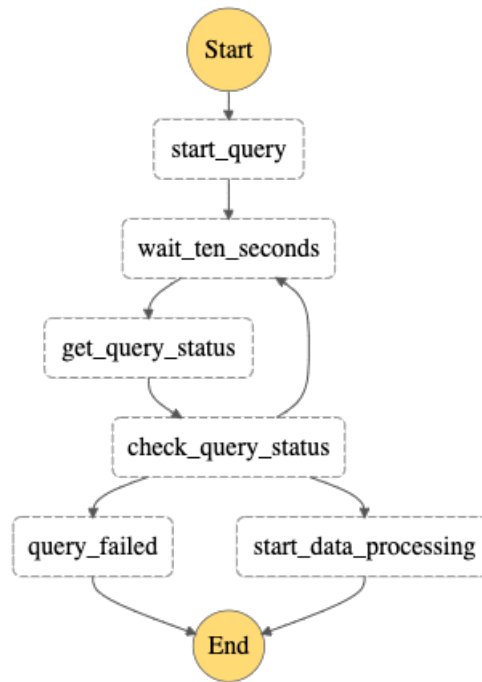


Figure 4.3: Step functions state machine

Pandas

Pandas[56] is a data analysis library for Python which provides various useful functions for our solution. Using the `pd.read_csv()` method, we can directly read the query results from S3 into a Pandas dataframe. The basic aggregation including the categorization into different threshold zones can be done with few lines of code (Listing 4.17). Additionally, we use the Pandas `to_sql()` method which generates and executes a SQL statement to write a Pandas DataFrame to a database.

SQLAlchemy

SQLAlchemy [57] is a SQL toolkit for Python which provides efficient and high-performing database access. We use it to create an engine to access the database. In combination with the Pandas `to_sql()` method, we can directly write DataFrames to the RDS database, which is used for fast access to the data.

```

1 #aggregate hourly data
2 hourly = df.groupby(pd.Grouper(key='timestamp', freq='1h'))
3 hourly_agg = hourly.agg({'mac': ['count', 'nunique'],
4                           'rssi_max': [checkThreshold(VISIT_THRESHOLD
5                                                   , "Visits"),
6                                           checkThreshold(VIEW_THRESHOLD,
7                                                         "Views")])})
6 hourly_agg['measurement_id'] = event["measurementId"]
7 hourly_agg.columns = ['count', 'unique_count', 'visits', 'views', '
  measurement_id']
8 hourly_agg.to_sql('hours', con=engine, if_exists='append', chunksize
  =1000)

```

Listing 4.17: Aggregating sensor data with pandas

4.2.5 Real-Time Analytics

For real-time insights into the data, we use the Kinesis Data Analytics [28] service which allows analysis of streaming data using SQL. The Kinesis Data Analytics service is connected to the existing Kinesis Data Firehose [27] stream and receives the raw data before any buffering happens. This minimizes the delay, however, additional parsing of the data is needed, as binary data can't be analyzed using SQL.

When analyzing streaming data, a window over which the data is aggregated has to be defined. The following window types supported by AWS have been considered:

- **Tumbling Windows**[58] aggregate data over fixed and non-overlapping time intervals. Each data record is only part of one window.
- **Stagger Window**[59] aggregate the data similar to tumbling windows. However, the window does not open until the first record matching the partition key arrives. Stagger windows are often used for time-related records that should not fall into different windows.
- **Sliding Windows**[60] aggregate data continuously over a fixed interval. While they can be used to *i.e.*, count all incoming records, they do not support *Group By* and are therefore not suitable to aggregate data per sensor.

For our solution, we are specifically interested in the number of MAC addresses per time interval. As it does not matter if a specific record falls in one window or another, we use a 10 seconds tumbling window.

Listing 4.18 shows the SQL query for the tumbling window which groups by the sensor ID. As a result, we get the count of MAC addresses per sensor every 10 seconds. The output of the SQL query is sent to a Lambda function (*cf.*, A.2), which writes the data to the RDS database.

```

1 CREATE OR REPLACE STREAM "DESTINATION_SQL_STREAM" (
2     "COL_timestamp" TIMESTAMP,
3     "sensor_id"      INTEGER,
4     "COL_count"      INTEGER);
5
6 CREATE OR REPLACE PUMP "STREAM_PUMP" AS
7 INSERT INTO "DESTINATION_SQL_STREAM"
8 SELECT STREAM
9     MIN(TO_TIMESTAMP("COL_timestamp")) as "COL_timestamp",
10     "sensor_id",
11     COUNT("mac") AS "COL_count"
12 FROM "SOURCE_SQL_STREAM_001"
13 GROUP BY "sensor_id", STEP("SOURCE_SQL_STREAM_001".ROWTIME BY
    INTERVAL '10' SECOND);

```

Listing 4.18: Aggregating sensor data with pandas

Chapter 5

Evaluation

This chapter evaluates the optimizations of collecting and processing Wi-Fi probe requests. Based on two different scenarios, we quantify the reduction of bandwidth usage and show in which environments the solution performs best. Furthermore, we present a cost estimation of the cloud infrastructure.

The number of probe requests a sensor can capture is strongly dependent on the surrounding environment and the usage behavior of the nearby smartphone users. In this chapter, we consider two typical scenarios of measurements as they are conducted by live marketing companies such as Livealytics [3].

- **Scenario 1** is a measurement in a retail store. The primary goal is to count the number of people and to assign them into categories as shown in Figure 4.2.
- **Scenario 2** is a measurement in a more crowded environment than scenario 1 *i.e.*, a tradeshow. The primary goal of the measurement is to collect a detailed heatmap.

Platform	Device Status	probe requests / h
Android	Sleep	1750
Android	Active	2250
iOS	Sleep	100
iOS	Active	600

Table 5.1: Probe request frequency estimation based on [2]

Based on the experimental setup of [2], we estimate the number of emitted probe requests per device (*cf.*, Table 5.1). Considering the number of devices in reach, the percentage of active devices, and the percentage of iOS devices, we estimate the number of probe requests that are captured by a sensor during one hour.

Table 5.2 shows the two scenarios and their different environments. The total probe requests per hour are calculated using data from Table 5.1. Furthermore, we calculate

the emitted probe requests per device within an aggregation period. For multiple probe requests from the same device within an aggregation period we assume a MAC randomization probability of 50%. In this way, we estimate the number of unique MAC addresses a sensor captures during an aggregation period.

Due to the fact that probe requests are bursty and the frequency varies according to a manufacturer, such calculations cannot reproduce reality exactly. Still, they can be used to evaluate the optimization and to estimate costs.

	Scenario 1	Scenario 2
Measurement duration	9 hours	4 hours
Aggregation duration (A)	10 seconds	1 second
Avg. number of devices in reach	25	40
Active devices	20%	30%
iOS devices	40%	40%
Total probe requests per hour	29750	49600
Probe requests per device and hour	1190	1240
Probe requests during A / device	3.3	0.34
Total probe requests / A	83	14
Total unique Mac Addresses /A	54	14

Table 5.2: Measurement Scenarios

5.1 Optimization of Bandwidth Usage

The proposed solution reduces the bandwidth in two ways. The following two subchapters quantify the reduction of bandwidth based on the above-presented scenario.

5.1.1 Aggregation

The aggregation reduces the number of packets a sensor needs to send. In contrast to solutions such as Livealytics [3], which send one packet per captured MAC address, the number of packets using aggregation solely depends on the aggregation duration, not on the amount of captured MAC addresses.

In scenario 1, we observe a reduction from 29750 to 360 packets/hour. Even with the short aggregation period in scenario 2, a reduction from 47600 to 3600 packets/hour can be achieved. The reduction of packets enables the use of a higher QoS for the MQTT messages and, therefore, increases reliability as well.

Furthermore, the aggregation also reduces the data to be sent, as multiple probe requests from the same MAC address within an aggregation period are combined. This omits

the transmission of duplicate data, such as identical MAC addresses or information elements, within an aggregation period. By only transmitting the unique MAC addresses, a reduction of 35% can be achieved in scenario 1. The aggregation performs best with an aggregation period of multiple seconds and with devices that do not randomize their MAC address for each probe request.

5.1.2 Serialization

As the structure of the sent data remains the same during measurements, we serialize the data to omit additional overhead caused by *i.e.* field names of readable data formats. Table 5.3 shows the size of the capture overhead and the size for each additional MAC. The capture overhead includes information about the sensor (Figure 3.3) and data format specific overhead. For each additional MAC address, we assume an information element size of 50 Bytes.

Comparing the 54 unique MAC addresses per aggregation period in scenario 1, the JSON data format takes up 11320 Bytes. On the other hand, the serialized Protobuf format takes up only 6645 Bytes, which is a reduction of over 40%. Similar results are observed for scenario 2.

	Protobuf	JSON
Capture overhead	3B	34B
Additional MAC	123B	209B

Table 5.3: Size comparison between the binary Protobuf format and JSON

In contrast to solutions which neither perform aggregation nor serialization [3], we can reduce the data to be transmitted by more than 60% in scenario 1 and still by more than 40% in scenario 2. The number of packets that is sent from a sensor to a server is reduced by 98% in scenario 1 and by 92% in scenario 2. Other solutions such as Probr [6] transmit entire PCAP files. While PCAP files are already in a binary format, they still contain redundant data inside and do not preserve privacy, as the unmodified MAC address is transmitted. Our proposed solution, therefore, provides a privacy-preserving collection of probe requests, while still reducing the data and packets to be transmitted.

5.2 Costs

For most services we use, the pricing model of AWS is based on usage *i.e.*, execution time or used storage. Only two services we use, namely Kinesis Data Analytics[28] and RDS[22], are billed a fixed price per hour. Generally, with increased usage, the rates decline due to volume discounts. Table 5.4 shows the pricing for the different services in the US East (Ohio) region as of June 2020 [61, 35, 33, 29, 31, 30, 62].

AWS Service	Price (monthly)
IoT Core	$\frac{1.00\$}{1 \text{ million messages}}$
Kinesis Data Firehose	$\frac{0.029\$}{\text{GB input}} + \frac{0.018\$}{\text{GB conversion}}$
Lambda	$\frac{0.0000008333\$}{100\text{ms running time}}$
S3	$\frac{0.023\$}{\text{GB}}$
Athena	$\frac{5.00\$}{\text{TB scanned}}$
RDS	$\frac{0.06\$}{\text{hour}} + \frac{0.10\$}{\text{GB}}$
Kinesis Data Analytics	$\frac{0.11\$}{\text{hour}} * 24 * 30 = 79.2\$$

Table 5.4: AWS Pricing for the US East (Ohio) region

To estimate the monthly cost we calculate the price for 1000 *scenario 1* measurements and 500 *scenario 2* measurements.

The total transmitted data and the number of transmitted packets per month would therefore be:

$$\textbf{Data: } 1000 \cdot 9\text{h} \cdot \frac{360}{h} \cdot 6645B + 500 \cdot 4\text{h} \cdot \frac{3600}{h} \cdot 1725B = 32'377MB$$

$$\textbf{Packets: } 1000 \cdot 9\text{h} \cdot \frac{360}{h} + 500 \cdot 4\text{h} \cdot \frac{3600}{h} = 10'440'000$$

IoT Core is billed by the number of exchanged messages. To account for Device Shadow updates we round the number of messages up to the next million. This results in monthly costs of:

$$\textbf{IoT Core: } (10.4M + 0.6M) \cdot 1\$ = 11.00\$$$

Kinesis Data Firehose is billed per GB that flows through the stream and results in costs of:

$$\textbf{Kinesis Data Firehose: } 32.3GB \cdot \frac{(0.029\$+0.018\$)}{GB} = 1.52\$$$

Kinesis Data Analytics is billed per Kinesis Processing Unit (KPU) per hour. As we only perform a simple SQL query, we only assume the usage of 1 KPU.

$$\textbf{Kinesis Data Analytics: } 1KPU \cdot \frac{79.2\$}{KPU} = 79.20\$$$

S3 is used as primary data storage. With the data increasing by 32 GB each month, we generously calculate the costs for 500 GB storage.

$$\textbf{S3: } 500GB \cdot \frac{0.023\$}{GB} = 11.50\$$$

Athena is billed per TB of data that needs to be scanned to return query results. As the query results are stored as CSV files, we only need to query the newly added data. To run queries for all measurements, we assume that all of the monthly added data is scanned 25 times.

$$\textbf{Athena: } 25 \cdot 0.032GB \cdot \frac{5\$}{TB} = 4\$$$

RDS only stores aggregated data and real-time results from Kinesis Analytics. We use an Aurora serverless RDS instance with 2 Aurora Capacity Units (ACU) assigned and 50 GB storage.

$$\textbf{RDS: } 2ACU \cdot \frac{43.2\$}{ACU} + 50GB \cdot \frac{0.10\$}{GB} = 91.4\$$$

Lambda functions are used at multiple locations. For the Kinesis Data Firehose, we assume that every 30 seconds a Lambda function to parse the data is invoked. For Kinesis Data Analytics, a Lambda function is invoked every second for incoming data and every 10 seconds to write the output to RDS. As simple data transformations can be computed very fast, we assume the minimal billable execution time of 100ms.

For the processing of the measurements, we assume a total execution time of 10 seconds for each measurement. This results in the following costs for the Lambda execution.

$$\begin{array}{c} \textbf{Lambda} \\ (2 + 60 + 6) \cdot 60 \cdot 30 \cdot 24 \cdot 100ms \cdot \frac{0.0000008333\$}{100ms} + 1500 \cdot 10'000ms \cdot \frac{0.0000008333\$}{100ms} = 2.57\$ \end{array}$$

This results in total costs of **201.19\$/month** for 1500 measurements according to the above-described scenarios. It has to be noted, that RDS and Kinesis Data Analytics account for more than 80% of the total cost due to their fixed hourly rate. This means that with even more measurements, a better cost/measurement rate could be achieved.

Even though the costs per measurement decrease the more measurements are collected and processed, it might at some point become cheaper to use a traditional server-based approach to process the data. However, this introduces additional work to configure and update the whole infrastructure. A tradeoff between the ease of configuration of the serverless approach and the cheaper computational power of the traditional server approach remains. Still, one of the key drivers for minimizing the costs is the presented aggregation and serialization which is independent of the cloud architecture.

Chapter 6

Final Considerations

The main goal of this study was to optimize the collection and processing of Wi-Fi probe requests. This included two main parts, one being the sensor firmware and the other being the cloud architecture, which handles and processes incoming data.

The general objective of a solution to collect Wi-Fi probe requests efficiently has been achieved in a way, that an aggregation duration in combination with a bandwidth preserving data format has been introduced. However, a large amount of time has been invested, to gain insight on different hardware boards towards an optimal setup which runs stable and is extendable to other connection types in the future. Hence, there are practical aspects typically revealed during the development of the study which, due to their experimental nature, prevent expressive optimizations from being possible due to the natural time of familiarization with different devices. Following three hardware boards were experimented in this independent study.

- GL-MiFi 4G Smart Router [63] running OpenWRT
- Odroid C1 [38] running Ubuntu minimal
- Digi ConnectCore 6UL SBC Pro [64] running Digi Embedded Yocto

As all platforms run different types of firmware and use different Wi-Fi interfaces a more general solution had to be implemented in order to achieve the goal in the given time frame. The GL-MiFi Router did not work because of its very constrained computational performance and memory (DDR2 64MB / FLASH 16MB) in contrast with the others. Furthermore, the default firmware image of the Digi ConnectCore contained a Wi-Fi driver which did not support monitor mode.

The presented Odroid C1 solution based on Python works but is not optimal in terms of performance. While the Digi ConnectCore hardware board is the best in terms of performance and connectivity, developing an embedded system providing the same functionality is left for further research.

The second objective of a scalable cloud architecture has been achieved. The use of AWS with its broad range of IoT and computing services contributed to the success. The cost

estimations show, that the presented solution is scalable up to thousands of measurements while still being reasonably cheap. The costs have been optimized by studying different use cases and best practices that AWS provides in its whitepapers [65]. By testing and comparing different services, solid knowledge of the whole AWS landscape has been gained, including services that did not end up in the presented solution. It has to be noted, that the used AWS services allow scalability in theory but no load tests to verify scalability have been run.

Because of the difficulties with the different hardware boards, the goal of scalability and reliability has been adapted to a more general optimization as there was no time left, to extensively test and evaluate the reliability. Furthermore, the written documentation took more time than originally anticipated.

Nonetheless, the presented solution provides a solid basis for future work. This includes the derandomization of MAC addresses which can be included in the existing cloud architecture. Furthermore, future work can use and adapt the aggregation scheme to other connection types such as Bluetooth.

Bibliography

- [1] Deloitte, “Smartphones - bane or boon? Opportunities and risks for customers, consumers and companies, Global Mobile Consumer Survey 2018: Results for Switzerland,” 2018. <https://www2.deloitte.com/content/dam/Deloitte/ch/Documents/technology-media-telecommunications/ch-deloitte-en-global-mobile-consumer-survey-2018.pdf> (Last accessed June 2020).
- [2] J. Freudiger, “How talkative is your mobile device? an experimental study of wi-fi probe requests,” in *Proceedings of the 8th ACM Conference on Security & Privacy in Wireless and Mobile Networks*, pp. 1–6, 2015.
- [3] Livealytics, “We make Live Experience measurable.” <https://www.livealytics.com/> (Last accessed June 2020).
- [4] “Google Analytics.” <https://analytics.google.com/analytics/web/> (Last accessed June 2020).
- [5] E. Kalogianni, R. Sileryte, M. Lam, K. Zhou, M. Van der Ham, S. Van der Spek, and E. Verbree, “Passive wifi monitoring of the rhythm of the campus,” in *Proceedings of The 18th AGILE International Conference on Geographic Information Science*, pp. 1–4, 2015.
- [6] J. Scheuner, G. Mazlami, D. Schöni, S. Stephan, A. De Carli, T. Bocek, and B. Stiller, “Probr-a generic and passive wifi tracking system,” in *2016 IEEE 41st Conference on Local Computer Networks (LCN)*, pp. 495–502, IEEE, 2016.
- [7] E. Vattapparamban, B. S. Çiftler, I. Güvenç, K. Akkaya, and A. Kadri, “Indoor occupancy tracking in smart buildings using passive sniffing of probe requests,” in *2016 IEEE International Conference on Communications Workshops (ICC)*, pp. 38–44, IEEE, 2016.
- [8] AWS, “Serverless Streaming Architectures and Best Practices.” https://d1.awsstatic.com/whitepapers/Serverless_Streaming_Architecture_Best_Practices.pdf (Last accessed June 2020).
- [9] AWS, “Amazon Web Services (AWS) - Cloud Computing Services.” <https://aws.amazon.com> (Last accessed June 2020).

- [10] S. He and S.-H. G. Chan, “Wi-fi fingerprint-based indoor positioning: Recent advances and comparisons,” *IEEE Communications Surveys & Tutorials*, vol. 18, no. 1, pp. 466–490, 2015.
- [11] Y. Kim, H. Shin, and H. Cha, “Smartphone-based wi-fi pedestrian-tracking system tolerating the rss variance problem,” in *2012 IEEE International Conference on Pervasive Computing and Communications*, pp. 11–19, IEEE, 2012.
- [12] H. Wang, S. Sen, A. Elgohary, M. Farid, M. Youssef, and R. R. Choudhury, “No need to war-drive: Unsupervised indoor localization,” in *Proceedings of the 10th international conference on Mobile systems, applications, and services*, pp. 197–210, 2012.
- [13] A. Musa and J. Eriksson, “Tracking unmodified smartphones using wi-fi monitors,” in *Proceedings of the 10th ACM conference on embedded network sensor systems*, pp. 281–294, 2012.
- [14] “IEEE Standard for Information technology-Telecommunications and information exchange between systems Local and metropolitan area networks-Specific requirements - Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications,” *IEEE Std 802.11-2016 (Revision of IEEE Std 802.11-2012)*, pp. 1–3534, 2016.
- [15] The Tcpdump Group, “Tcpdump.” <https://www.tcpdump.org/> (Last accessed June 2020).
- [16] J. Martin, T. Mayberry, C. Donahue, L. Foppe, L. Brown, C. Riggins, E. C. Rye, and D. Brown, “A study of mac address randomization in mobile devices and when it fails,” *Proceedings on Privacy Enhancing Technologies*, vol. 2017, no. 4, pp. 365–383, 2017.
- [17] “IEEE Registration Authority.” <https://standards.ieee.org/products-services/regauth/index.html> (Last accessed June 2020).
- [18] IEEE Registration Authority, “Company ID (CID).” <https://standards.ieee.org/products-services/regauth/cid/index.html> (Last accessed June 2020).
- [19] M. Vanhoef, C. Matte, M. Cunche, L. S. Cardoso, and F. Piessens, “Why mac address randomization is not enough: An analysis of wi-fi network discovery mechanisms,” in *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, pp. 413–424, 2016.
- [20] AWS, “What is Streaming Data?.” <https://aws.amazon.com/streaming-data/> (Last accessed June 2020).
- [21] AWS, “Databases on AWS.” <https://aws.amazon.com/products/databases/> (Last accessed June 2020).
- [22] AWS, “Amazon Relational Database Service (RDS).” <https://aws.amazon.com/rds/> (Last accessed June 2020).

- [23] AWS, “Amazon DynamoDB.” <https://aws.amazon.com/dynamodb/> (Last accessed June 2020).
- [24] AWS, “Amazon Simple Storage Service S3.” <https://aws.amazon.com/s3/> (Last accessed June 2020).
- [25] AWS, “AWS Glue.” <https://aws.amazon.com/glue/> (Last accessed June 2020).
- [26] AWS, “Amazon Athena.” <https://aws.amazon.com/athena/> (Last accessed June 2020).
- [27] AWS, “Amazon Kinesis Data Firehose.” <https://aws.amazon.com/kinesis/data-firehose/> (Last accessed June 2020).
- [28] AWS, “Amazon Kinesis Data Analytics.” <https://aws.amazon.com/kinesis/data-analytics/> (Last accessed June 2020).
- [29] AWS, “Amazon S3 pricing.” <https://aws.amazon.com/s3/pricing> (Last accessed June 2020).
- [30] AWS, “Amazon RDS for MySQL Pricing.” <https://aws.amazon.com/rds/mysql/pricing/> (Last accessed June 2020).
- [31] AWS, “Amazon Athena pricing.” <https://aws.amazon.com/athena/pricing/> (Last accessed June 2020).
- [32] AWS, “AWS Lambda.” <https://aws.amazon.com/lambda/> (Last accessed June 2020).
- [33] AWS, “AWS Lambda pricing.” <https://aws.amazon.com/lambda/pricing/> (Last accessed June 2020).
- [34] AWS, “AWS Step Functions.” <https://aws.amazon.com/step-functions/> (Last accessed June 2020).
- [35] AWS, “Amazon Kinesis Data Firehose pricing.” <https://aws.amazon.com/kinesis/data-firehose/pricing> (Last accessed June 2020).
- [36] “Radiotap.” <https://www.radiotap.org/> (Last accessed June 2020).
- [37] “Python.” <https://www.python.org/> (Last accessed June 2020).
- [38] Hardkernel, “ODROID-C1.” <https://www.hardkernel.com/shop/odroid-c1-2/> (Last accessed June 2020).
- [39] “Odroid C1 Ubuntu 18.04 images.” https://odroid.in/ubuntu_18.04lts/C0_C1/ (Last accessed June 2020).
- [40] Hardkernel, “WiFi Module 4 - Odroid.” <https://www.hardkernel.com/shop/wifi-module-4/> (Last accessed June 2020).
- [41] “pip - The Python Package Installer.” <https://pip.pypa.io/en/stable/> (Last accessed June 2020).

- [42] AWS, “AWS IoT Core.” <https://aws.amazon.com/iot-core/> (Last accessed June 2020).
- [43] AWS, “AWS IoT Device SDK for Python.” <https://github.com/aws/aws-iot-device-sdk-python> (Last accessed June 2020).
- [44] N. Corbett, “Understanding the AWS IoT Security Model.” <https://aws.amazon.com/blogs/iot/understanding-the-aws-iot-security-model/> (Last accessed June 2020).
- [45] AWS Docs, “Device shadow service for AWS IoT.” <https://docs.aws.amazon.com/iot/latest/developerguide/iot-device-shadows.html> (Last accessed June 2020).
- [46] Oasis Open, “MQTT Version 3.1.1.” <http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.html> (Last accessed June 2020).
- [47] R. McGovern, “Inotify-Tools.” <https://github.com/inotify-tools/inotify-tools/> (Last accessed June 2020).
- [48] Philippe Biondi and the Scapy community, “Scapy.” <https://scapy.net/> (Last accessed June 2020).
- [49] Google, “Protocol Buffers.” <https://developers.google.com/protocol-buffers> (Last accessed June 2020).
- [50] Apache Software Foundation, “Apache Parquet.” <https://parquet.apache.org/> (Last accessed June 2020).
- [51] AWS Docs, “JSON SerDe Libraries.” <https://docs.aws.amazon.com/athena/latest/ug/json.html#openxjson> (Last accessed June 2020).
- [52] Serverless, Inc., “The Serverless Application Framework.” <https://www.serverless.com/> (Last accessed June 2020).
- [53] AWS, “AWS Cloud Formation.” <https://aws.amazon.com/cloudformation/> (Last accessed June 2020).
- [54] AWS, “Amazon Virtual Private Cloud.” <https://aws.amazon.com/vpc/> (Last accessed June 2020).
- [55] AWS, “AWS SDK for Python (Boto3).” <https://aws.amazon.com/sdk-for-python/> (Last accessed June 2020).
- [56] The pandas development team, “pandas - Python Data Analysis Library,” 2020. <https://pandas.pydata.org/>.
- [57] SQLAlchemy, “The Database Toolkit for Python.” <https://www.sqlalchemy.org/> (Last accessed June 2020).
- [58] AWS Docs, “Tumbling Windows (Aggregations Using GROUP BY).” <https://docs.aws.amazon.com/kinesisanalytics/latest/dev/tumbling-window-concepts.html> (Last accessed June 2020).

- [59] AWS Docs, “Stagger Windows.” <https://docs.aws.amazon.com/kinesisanalytics/latest/dev/stagger-window-concepts.html> (Last accessed June 2020).
- [60] AWS Docs, “Sliding Windows.” <https://docs.aws.amazon.com/kinesisanalytics/latest/dev/sliding-window-concepts.html> (Last accessed June 2020).
- [61] AWS, “AWS IoT Core pricing.” <https://aws.amazon.com/iot-core/pricing/> (Last accessed June 2020).
- [62] AWS, “Amazon Kinesis Data Analytics pricing.” <https://aws.amazon.com/kinesis/data-analytics/pricing/> (Last accessed June 2020).
- [63] GL iNet, “GL-MiFi 4G Smart Router.” <https://store.gl-inet.com/products/gl-mifi-4g-smart-router> (Last accessed June 2020).
- [64] Digi, “Digi ConnectCore 6UL SBC Pro.” <https://www.digi.com/products/embedded-systems/single-board-computers/connectcore-for-i-mx6ul-sbc-pro> (Last accessed June 2020).
- [65] AWS, “AWS Whitepapers Guides.” <https://aws.amazon.com/whitepapers/> (Last accessed June 2020).

Abbreviations

AP	Access Point
AWS	Amazon Web Services
CID	Company Identifiers
GPS	Global Positioning System
IPC	Inter Process Communication
IPS	Indoor Positioning System
KPI	Key Performance Indicator
ML	Machine Learning
NIC	Network Interface Controller
OUI	Organizationally Unique Identifier
QoS	Quality of Service
RSSI	Received Signal Strength Intensity
SSID	Service Set Identifier
VENV	Python Virtual Environment
VPC	Virtual Private Cloud
WPS	Wi-Fi Protected Setup

List of Figures

3.1	Steps to collect Wi-Fi Probe Requests	6
3.2	Example MAC Address using a CID registered by Google Inc.	7
3.3	Structure of an aggregated packet that is transmitted after an aggregation period.	10
3.4	Different approaches to transmit data from an IoT device to a server. . . .	11
3.5	AWS Architecture to store and process Wi-Fi probe request data.	14
4.1	Interplay between tcpdump, inotifywait, and the Python script to parse the data.	21
4.2	Categorization of probe requests into different zones based on the RSSI value.	29
4.3	Step functions state machine	31

List of Tables

3.1	Fields of a Probe Request to be captured	7
3.2	Aggregation duration for different measurement types	8
3.3	Elements to be captured and calculated for a single MAC address within an aggregation period. Bold values are aggregated over multiple frames. .	9
4.1	Glue Data Catalog Table	28
5.1	Probe request frequency estimation based on [2]	33
5.2	Measurement Scenarios	34
5.3	Size comparison between the binary Protobuf format and JSON	35
5.4	AWS Pricing for the US East (Ohio) region	36

Appendix A

Further code examples

This chapter contains additional code examples. The full source code is available on Github¹.

A.1 Sensor scripts

Main Python script

```
1 import errno
2 import json
3 import os
4 import shutil
5 import uuid
6 import pcapParser
7
8 from AWSIoTPythonSDK.MQTTLib import AWSIoTMQTTClient,
   AWSIoTMQTTShadowClient
9 from shadowCallbackHandler import ShadowCallbackContainer
10
11 with open('config.json', 'r') as configFile:
12     config = json.load(configFile)
13
14 rootCa = ".ssh/root-CA.crt"
15 privateKey = ".ssh/" + str(config["sensorId"]) + ".private.key"
16 cert = ".ssh/" + str(config["sensorId"]) + ".cert.pem"
17
18 # MQTT Client
19 mqttClient = AWSIoTMQTTClient(str(uuid.uuid4()))
20 mqttClient.configureEndpoint(config["host"], config["port"])
21 mqttClient.configureCredentials(rootCa, privateKey, cert)
22 mqttClient.configureOfflinePublishQueueing(-1)
23 mqttClient.configureConnectDisconnectTimeout(10)
24 mqttClient.configureMQTTOperationTimeout(10)
25
26 mqttClient.connect()
```

¹<https://github.com/j0nezz/pySniff>

```

27
28 # Device Shadow
29 shadowClient = AWSIoTMQTTShadowClient(str(uuid.uuid4()))
30 shadowClient.configureEndpoint(config["host"], config["port"])
31 shadowClient.configureCredentials(rootCa, privateKey, cert)
32 shadowClient.configureAutoReconnectBackoffTime(1, 32, 20)
33 shadowClient.configureConnectDisconnectTimeout(10)
34 shadowClient.configureMQTTOperationTimeout(5)
35
36 deviceShadowHandler = shadowClient.createShadowHandlerWithName(str(
    config["sensorId"]), True)
37 shadowCallbackContainer_Bot = ShadowCallbackContainer(
    deviceShadowHandler)
38 shadowClient.onOnline = shadowCallbackContainer_Bot.onConnect
39 shadowClient.connect()
40
41 deviceShadowHandler.shadowRegisterDeltaCallback(
    shadowCallbackContainer_Bot.customShadowCallback_Delta)
42
43 # paths
44 FIFO = "pcapNotify"
45 CAPTURE = "capture/"
46 ARCHIVE = "archive/"
47
48 def parse_file(path):
49     # Always read from config file so Shadow changes are applied without
        restart.
50     with open('config.json', 'r') as tmp_config_file:
51         tmp_config = json.load(tmp_config_file)
52         result = pcapParser.parse_from_path(path, tmp_config["sensorId"])
53         with open("status.log", "a") as log:
54             log.write("Publishing file: " + path + "\n")
55         mqttClient.publish(tmp_config["topic"], bytearray(result), 1)
56     try:
57         shutil.move(path, 'archive/'+path[path.rfind('/')+1:])
58     except FileNotFoundError as e:
59         with open("status.log", "a") as log:
60             log.write("Could not move file:" + str(e))
61
62
63 def parse_existing():
64     for filename in os.listdir(CAPTURE):
65         result = pcapParser.parse_from_path(CAPTURE + filename, config['
            sensorId'])
66         mqttClient.publish(config["topic"], bytearray(result), 1)
67         print("published", filename)
68         shutil.move(CAPTURE + filename, ARCHIVE + filename)
69
70
71 try:
72     parse_existing()
73 except Exception as e:
74     print(e)
75
76 # listen for messages in named pipe
77 try:

```

```

78     os.mkfifo(FIFO)
79 except OSError as oe:
80     if oe.errno != errno.EEXIST:
81         raise
82
83 while True:
84     with open(FIFO) as fifo:
85         while True:
86             data = fifo.read()
87             print(data)
88             if len(data) == 0:
89                 break
90             parse_file(data)

```

Capture Class

```

1 import capture_pb2
2 from singleMac import SingleMac
3
4
5 class Capture(object):
6     def __init__(self, sensor_id):
7         self.packets = {}
8         self.sensor_id = sensor_id
9
10    def add_packet(self, timestamp, mac, rssi, sequence_nr,
11                  information_elements):
12        if mac in self.packets:
13            self.packets.get(mac).add_packet(timestamp, rssi)
14        else:
15            self.packets[mac] = SingleMac(timestamp, mac, rssi,
16                                           sequence_nr, information_elements)
17
18    def serialize(self):
19        ser_capture = capture_pb2.Capture()
20        ser_capture.sensor_id = self.sensor_id
21        for key in self.packets.keys():
22            ser_capture.mac.extend([self.packets.get(key).serialize()])
23        return ser_capture.SerializeToString()

```

Device Shadow Callback Handler

```

1 import json
2 import os
3 from threading import Thread
4
5
6 class ShadowCallbackContainer:
7     def __init__(self, deviceShadowInstance):
8         self.deviceShadowInstance = deviceShadowInstance
9
10    def onConnect(self):
11        thread = Thread(target=self.shadowGet)
12        thread.start()
13
14    def shadowGet(self):
15        try:

```

```

16         self.deviceShadowInstance.shadowGet(self.updateLocalShadow,
17         10)
18     except Exception as e:
19         print("could not get shadow", str(e))
20
21 def updateLocalShadow(self, state, status, id):
22     try:
23         print(state)
24         stateJson = json.loads(state)
25         with open("config.json", "w") as config:
26             print(json.dumps(state, indent=2))
27             json.dump(stateJson["state"]["reported"], config, indent
28             =2)
29             if stateJson["state"]["reported"]["aggregationDuration"]:
30                 with open("measurement.conf", "w") as bashConfig:
31                     bashConfig.write("aggregation=" + str(stateJson["
32                     state"]["reported"]["aggregationDuration"]))
33     except Exception as e:
34         print("could not write config.json", str(e))
35
36 # Handle shadow changes
37 def customShadowCallback_Delta(self, payload, responseStatus, token)
38 :
39     reboot = False
40     print("Received a delta message:")
41     payloadDict = json.loads(payload)
42     deltaMessage = json.dumps(payloadDict["state"])
43     print(deltaMessage)
44
45 # write to file
46 try:
47     with open("config.json", "r") as config:
48         stateJson = json.load(config)
49         for key, value in payloadDict["state"].items():
50             stateJson[key] = value
51         with open("config.json", "w") as config:
52             print("data to write", stateJson)
53             json.dump(stateJson, config, indent=2)
54             if "aggregationDuration" in payloadDict["state"]:
55                 reboot = True
56                 with open("measurement.conf", "w") as bashConfig:
57                     bashConfig.write("aggregation="+str(payloadDict["
58                     state"]["aggregationDuration"]))
59
60 # acknowledge changes
61 newPayload = '{"state":{"reported":"' + deltaMessage + '}}'
62 self.deviceShadowInstance.shadowUpdate(newPayload, None, 5)
63
64 # trigger reboot if necessary
65 if reboot:
66     os.system('reboot')
67
68 except Exception as e:
69     print("Could not parse payload delta", str(e))

```

A.2 AWS implementation

State Machine Logic (serverless.yml)

```

1  stepFunctions:
2    stateMachines:
3      query_and_process_data:
4        name: "Query-and-process-data"
5        definition:
6          StartAt: start_query
7          States:
8            start_query:
9              Type: Task
10             Resource:
11               Fn::GetAtt: [start-athena-query, Arn]
12             Next: wait_ten_seconds
13            wait_ten_seconds:
14              Type: Wait
15              Seconds: 10
16              Next: get_query_status
17            get_query_status:
18              Type: Task
19              Resource:
20                Fn::GetAtt: [get-athena-query-status, Arn]
21              Next: check_query_status
22            query_failed:
23              Type: Fail
24              Cause: Cause unclear, investigate the cloudwatch logs
25              Error: The query failed, check the logs.
26            check_query_status:
27              Type: Choice
28              Choices:
29                - Or:
30                  - Variable: "$.QueryState"
31                    StringEquals: FAILED
32                  - Variable: "$.QueryState"
33                    StringEquals: CANCELED
34                  Next: query_failed
35                - Or:
36                  - Variable: "$.QueryState"
37                    StringEquals: RUNNING
38                  - Variable: "$.QueryState"
39                    StringEquals: QUEUED
40                  Next: wait_ten_seconds
41                - Variable: "$.QueryState"
42                  StringEquals: SUCCEEDED
43                  Next: start_data_processing
44            start_data_processing:
45              Type: Task
46              Resource:
47                Fn::GetAtt: [process-sensor-data, Arn]
48            End: true

```

Kinesis Analytics to RDS Lambda Function

```

1     import base64
2     import os
3     import sys
4     import pymysql
5     import json
6
7     # environment variables
8     rds_host = os.environ['DB_ENDPOINT']
9     name = os.environ['DB_USER']
10    password = os.environ['DB_PASSWORD']
11    db_name = 'wifi'
12
13    # get pymysql connection to rds
14    try:
15        conn = pymysql.connect(rds_host, user=name, passwd=password, db=
            db_name, connect_timeout=5, autocommit=True)
16    except pymysql.MySQLError as e:
17        print("ERROR: Unexpected error: Could not connect to MySQL instance.
            ")
18        sys.exit()
19
20    print("SUCCESS: Connection to RDS MySQL instance succeeded")
21
22
23    def kinesis_analytics_to_rds(event, context):
24        payload = event['records']
25        output = []
26        success = 0
27        failure = 0
28        with conn.cursor() as cur:
29            for record in payload:
30                try:
31                    # parse record and write it to RDS
32                    print("b64decode data", record['data'])
33                    payload = base64.b64decode(record['data'])
34                    print("payload", payload)
35                    data_item = json.loads(payload)
36                    query = "INSERT INTO wifi.live (timestamp,sensor_id,
                        count) values (%s, %s, %s)"
37                    cur.execute(query, (str(data_item["COL_timestamp"]),
                        data_item["sensor_id"], data_item["COL_count"]))
38                    success += 1
39                    output.append({'recordId': record['recordId'], 'result':
                        'Ok'})
40
41                except Exception as e:
42                    print("Error", str(e))
43                    failure += 1
44                    output.append({'recordId': record['recordId'], 'result':
                        'Ok'})
45
46    print('Successfully delivered {0} records, failed to deliver {1}
        records'.format(success, failure))
47    return {'records': output}

```