Cloud-Based IoT Management with SIM Card

Harrison Cassar
University of California, Los Angeles
Los Angeles, CA, U.S.
harrisoncassar@cs.ucla.edu

Ricky Guo
University of California, Los Angeles
Los Angeles, CA, U.S.
rickyrguo@cs.ucla.edu

Disha Zambani
University of California, Los Angeles
Los Angeles, CA, U.S.
dmzambani@cs.ucla.edu

Albert Stanley
University of California, Los Angeles
Los Angeles, CA, U.S.
albertstanley@ucla.edu

ABSTRACT

The growing functionality of SIM cards in 5G IoT networks brings challenges in managing its many features. Because of the multi-carrier capabilities of 5G SIM cards, in addition to their functionality as a secure bridge between IoT devices and the cloud, there is a growing need to develop a cloud service for SIM card management. We developed a cloud service to manage these capabilities and a two way transmission scheme that receives instructions from the cloud and transmits the IoT data to the cloud. The cloud service is integrated with Kafka to enable message queues for the incoming IoT data for downstream tasks. The user of the service has access to an easy to use UI to visualize the IoT data as well as submit carrier change requests.

CCS CONCEPTS

• **Networks → Cloud computing**; **Network management**; **Mobile networks**.

KEYWORDS

multi-carrier SIM, 5G, cellular IoT, cloud, modem

1 INTRODUCTION

The convergence of 5G (fifth-generation) networks and the Internet of Things (IoT) has ushered in a new era of connectivity, offering unprecedented speed, reliability, and scalability. With this new era, new features and technologies emerge. In this section, we focus on two key features of 5G IoT and how we develop a service to manage them as the basis for our project.

1.1 Motivation

With the growing interest in 5G IoT, there also comes a need to develop technologies to manage its many features. A key feature of 5G IoT is the multi-carrier capabilities of 5G SIM cards. These cards can also transmit data packets, which can be used for transmitting IoT data to the cloud from IoT devices connected to the SIM card device. Creating a cloud service to handle the implications of these two features would be useful for efficient management of SIM cards and associated IoT data for 5G IoT users.

1.2 Problem Statement and Project Goals

To realize this cloud service, we first identify a few key features needed for its implementation. First, we require a 2-way message service that is able to send IoT data to and receive messages from the cloud. We also want to allow users to visualize the IoT data being sent over. Second, we use the mentioned cloud service to enable message queues for incoming IoT data for future downstream tasks such as data analytics or machine learning tasks. Third, we include a way to control and change the SIM card carrier via a user-friendly UI for SIM card owners where users can also view the SIM card carrier status. Finally, we want to achieve these components by having little-to-no modifications on the device SIM card device, and by having the SIM card communicate directly to the cloud we can avoid possible modifications.

2 METHODS

In this section, we outline our developed system, including its implementation and some challenges we faced along the way.

2.1 System Overview

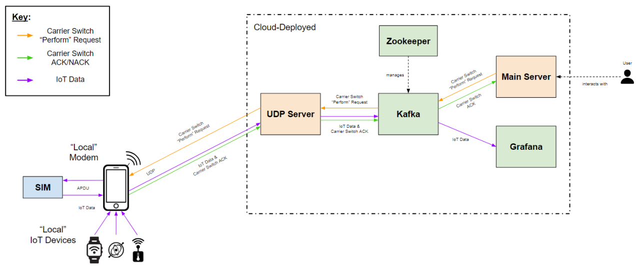


Figure 1: High-level system diagram with main data flows labeled.

As seen in Figure 1, our developed system is composed of two main subsystems: the cloud-deployed server subsystem, and the "local" modem subsystem. Effectively, the cloud server provides the main carrier switch functionality and IoT data visualization to the user while bridging the (communication) gap to the local modem and SIM subsystem. In order to limit this developed system to that of the original project scope, we replace the local modem and SIM subsystem with a mocked modem client and a Javacard-eSIM applet (as seen in Figure 2), allowing us to focus purely on the actual project requirements instead of dealing with an actual SIM and modem, which would prove more difficult—or expensive—to maintain or make any modifications to.

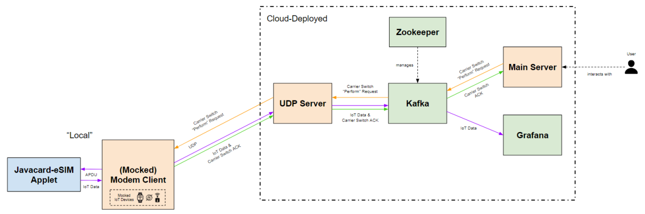


Figure 2: High-level system diagram with main data flows labeled and actual depiction of mocked local subsystem.

Our system provides 3 main data flows, including the 1 downstream flow of a Carrier Switch "Perform" Request from the user down to the mocked modem client and SIM card, and the 2 upstream flows of a Carrier Switch ACK/NACK and (mocked, generated) IoT data for visualization for the user. Each of the subsystems' components help to jointly achieve support for these data flows.

The cloud subsystem is composed of 5 main components who ultimately provide this end-to-end data flows that are key to this project's requirements. Apache Kafka is a distributed event streaming platform, and, along with its sister component Zookeeper who manages the Kafka instance, is used mostly out-of-the-box with some specific configurations to effectively provide a platform to pass messages and data to and from the front-end and back-end of our cloud server. The Main Server provides the direct interface with the user, serving our web application to the user that exposes the carrier switch functionality and links to Grafana, which provides the visualization of any up-streamed IoT data to the user. The last component of our cloud system is that of the UDP Server, who effectively manages all communication with the local modem client. Each of these 5 main components are deployed in their own Docker container and connected over the default Docker internal "bridge" network setup by Docker Compose, easing up the process of deployment of our system (as covered in the below "Cloud Deployment" section).

The local subsystem is composed of 2 main components: the Modem Client and the Javacard-eSIM Applet. The Modem Client serves to mock the modem and IoT devices, managing all communication with the SIM card and the UDP Server, while the Javacard-eSIM Applet mocks the multi-carrier SIM card.

2.2 Subsystem Implementations

In this section, we provide a deeper description of the various individual components of both the local and cloud subsystems.

2.2.1 SIM. The SIM card's primary purpose is to run the JavaCard applet that will contain our "carrier switch" and message parsing/forming logic. In the real world, the SIM would also provide cellular service to the device (cellular phone/modem) that it is plugged into. Our setup is a SIM card connected to a SIM reader (shown in 3) that we can connect through the USB port on a computer. The JavaCard applet developed handles logic to parse incoming carrier switch and IoT data packets, execute the corresponding logic, and return the acknowledgments or simply mirror back the IoT data packets. The code for the applet developed can be found here: https://github.com/JinghaoZhao/eSIM-Applet-dev/tree/dev-5G-security.

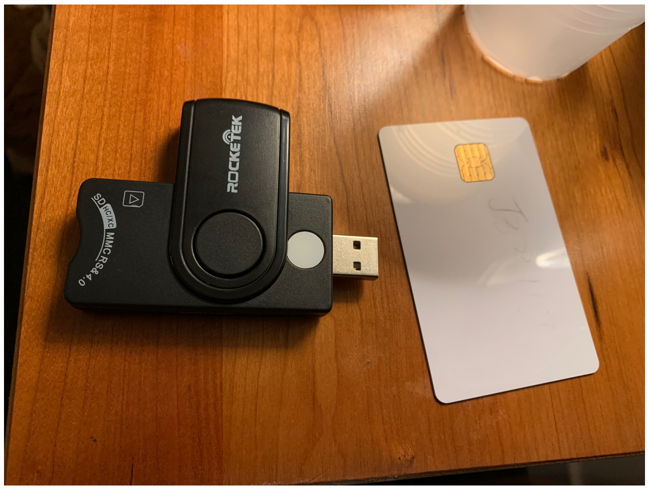


Figure 3: The SIM card and SIM reader used for testing.

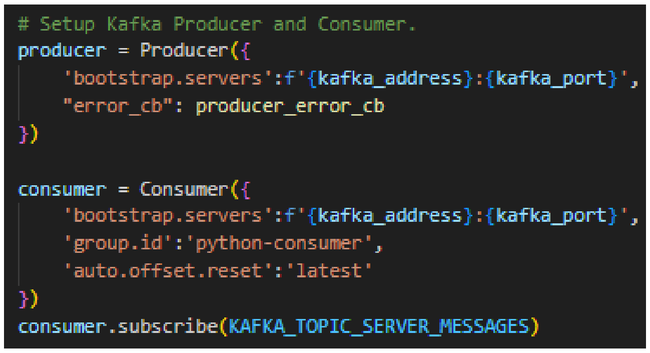
2.2.2 Mocked Modem Client. The mocked modem client is used for two primary purposes. For one, it will act as an actual modem used to communicate with the SIM card through APDU commands. The second functionality the modem has is to serve as a simulated device that is generating some IoT data that needs to be eventually passed to the cloud. This modem client works as a Python script with dependencies on 'pytlv' and 'pyscard' to interface with the SIM card. The 'SIM_Reader' class from the eSIM-loader repository provided by the UCLA Wing lab[2] was used to interface with the card itself when connected though the SIM card reader.[1]

The mocked modem client uses five separate threads to achieve the needed functionality. One thread listens from the server to incoming UDP packets and places it a local buffer. A second thread takes packets from this local buffer and determines whether it is a packet that the SIM card is able to handle. In our current design, we only expect packets containing a carrier switch request to come from the server. Another thread serves as our simulation by polling simulated IoT sensors to produce values at a given rate and placing these data packets into a local buffer. A fourth thread continuously pings the SIM card to determine whether the SIM would like to send or receive data so that any data packets can either be transmitted or received from the SIM. A final thread was then used to route packets received from the SIM to the server.

The mocked modem client runs as an independent script within a python virtual environment. The mocked modem client sets a specific port at which it listens to UDP packets from the server, and interacts with the main server by sending packets to a specified port on the server.

2.2.3 *UDP Server.*

The UDP Server, as mentioned briefly before, facilitates all communication between the cloud server and the modem, providing an "entrypoint" or "point-of-contact" for all modem-to-cloud communication. This UDP Server is fully developed in Python, utilizing multithreading with Python's built-in ‘threading‘ module to run each of its three main tasks concurrently in a (mostly) non-blocking state. These tasks include: 1) listening for UDP packets sent from the modem, and pushing them into a thread-safe Queue for processing; 2) processing UDP packets from this Queue, decoding them, and running their respective handlers that serve to push the packet's data or information to Kafka; and 3) listening—and processing—any Carrier Switch Requests coming downstream from the Main Server via consuming from Kafka.



```python
# Setup Kafka Producer and Consumer.
producer = Producer({
    'bootstrap.servers':f'{kafka_address}:{kafka_port}',
    "error_cb": producer_error_cb
})

consumer = Consumer({
    'bootstrap.servers':f'{kafka_address}:{kafka_port}',
    'group.id':'python-consumer',
    'auto.offset.reset':'latest'
})
consumer.subscribe(KAFKA_TOPIC_SERVER_MESSAGES)
```

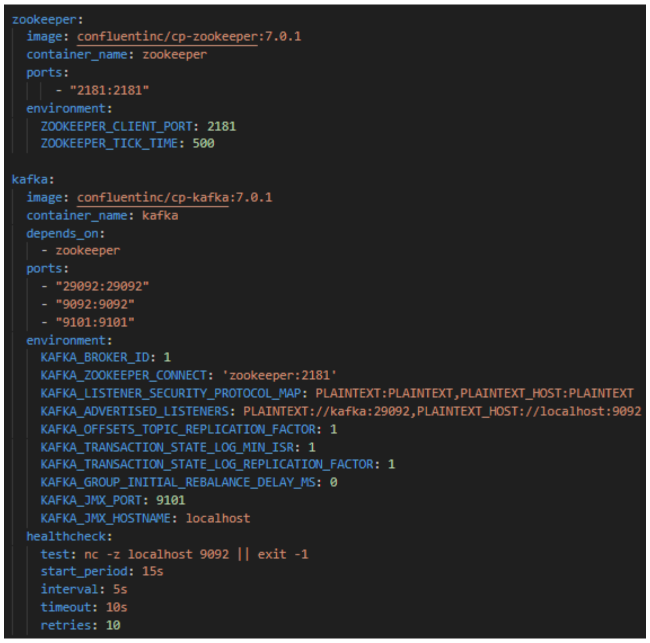
Figure 4: Screenshot of UDP Server source code that sets up a Consumer and Producer instance for interacting with the Kafka server component. This utilizes the Confluent's Apache Kafka Python Client API.

To enable all Kafka communication, the UDP server utilizes the ‘confluent_kafka‘ module, Confluent's Python Client for Apache Kafka [6][4]. This involves instantiating a (configured) Consumer and Producer object, working in a very similar fashion to normal uni-directional queues or sockets. As described in the following "Kafka and Zookeper" section, all messages and data are produced to and pulled from Kafka "Topics", where this project configures a specific set of pre-defined Kafka topics to cover all data flows (as covered in the later "Kakfa Topic Configuration" section). The UDP Server obeys this pre-defined configuration, pushing/pulling data from these topics based on the direction and type of message/packet received.

We note that the UDP Server maintains the important role of essentially "routing" all upstream data to the correct Kafka topic(s) to enable further tasks (downstream from the user perspective, upstream from the local/modem perspective) on this data. Currently, as outlined before, this project is limited to the task of basic visualization for the user using Grafana. We push all IoT data to Grafana in 2 different ways: via specific Kafka topics (to meet the requirements of our project) and via streaming directly through Grafana Live by simple HTTP POST requests (to achieve a faster livestream of data for our current implementation). We describe these further in the later "Grafana" subsection.

2.2.4 *Kafka and Zookeeper.*

Apache Kafka—along with its sister manager module Zookeeper—is an open-source distributed event streaming platform used by our system in order to effectively pass messages and data around between our various cloud subsystem components. For our system, we utilize the Confluent Platform for Apache Kafka [4] along with its sister Zookeeper module virtually almost completely out-of-the-box, performing a few slight configurations to properly enable our system for its nominal behavior. Being even more specific, we use version 7.0.1 Docker images of both Confluent Kafka and Confluent Zookeeper.



```yaml
zookeeper:
  image: confluentinc/cp-zookeeper:7.0.1
  container_name: zookeeper
  ports:
      - "2181:2181"
  environment:
    ZOOKEEPER_CLIENT_PORT: 2181
    ZOOKEEPER_TICK_TIME: 500

kafka:
  image: confluentinc/cp-kafka:7.0.1
  container_name: kafka
  depends_on:
    - zookeeper
  ports:
    - "29092:29092"
    - "9092:9092"
    - "9101:9101"
  environment:
    KAFKA_BROKER_ID: 1
    KAFKA_ZOOKEEPER_CONNECT: 'zookeeper:2181'
    KAFKA_LISTENER_SECURITY_PROTOCOL_MAP: PLAINTEXT:PLAINTEXT,PLAINTEXT_HOST:PLAINTEXT
    KAFKA_ADVERTISED_LISTENERS: PLAINTEXT://kafka:29092,PLAINTEXT_HOST://localhost:9092
    KAFKA_OFFSETS_TOPIC_REPLICATION_FACTOR: 1
    KAFKA_TRANSACTION_STATE_LOG_MIN_ISR: 1
    KAFKA_TRANSACTION_STATE_LOG_REPLICATION_FACTOR: 1
    KAFKA_GROUP_INITIAL_REBALANCE_DELAY_MS: 0
    KAFKA_JMX_PORT: 9101
    KAFKA_JMX_HOSTNAME: localhost
  healthcheck:
    test: nc -z localhost 9092 || exit -1
    start_period: 15s
    interval: 5s
    timeout: 10s
    retries: 10
```

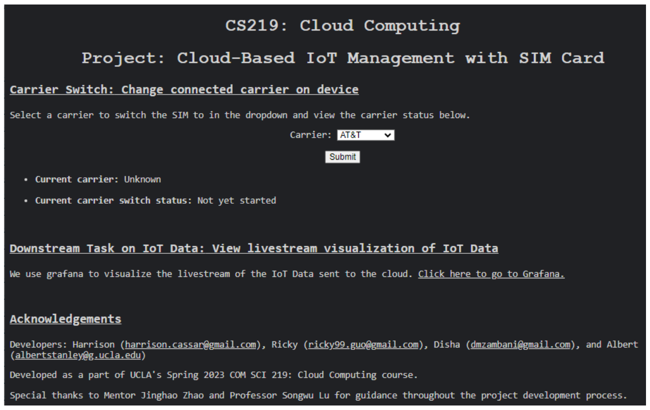
Figure 5: Screenshot of the Zookeper and Kafka Docker image configurations used in our cloud system's ‘docker-compose.yml‘ file.

To speak a bit about the actual operation, all of our cloud components speak directly to the Kafka instance deployed on its own Docker container. Different clients can either be Consumers or Producers of data to Kafka, pushing or pulling data to specific Kafka Topics (indicated by distinct names), which effectively act like message platform blackboards. Kafka Brokers effectively handle these client consume or produce requests as a server, replicating the messages in Topics that they are in charge of to create redundancy. Messages in a specific Topic are split across Partitions to fit the large capacity-distributed context that Kafka aims to support. For our purposes, we keep our system design relatively simple and limit ourselves to a singular Broker (replication factor of 1) and a few specific Topics. It is trivial, with access to more compute power, to then scale horizontally and add more Kafka Brokers to increase Kafka's processing abilities. See Figure 5 for the full list of configurations.

2.2.5 *Main Server.*

The Main Server component of the cloud subsystem, as mentioned before, stands as the direct interface and entry-point for the user to the system, serving our web application to the user and exposing all carrier-switch functionality. The Main

Server is built using Python-Flask [10]. Similar to the UDP Server component, it interacts with the Kafka component utilizing the ‘confluent_kafka‘ module, Confluent’s Python Client for Apache Kafka [6][4]. Needing to both produce and consume messages from the Kafka instance for carrier switch downstream requests and carrier switch ACKs respectively, the Main Server instantiates a (configured) Consumer and Producer object, and pushes/pulls from a pre-defined set of Kafka Topics (as covered in the later "Kakfa Topic Configuration" section).



CS219: Cloud Computing

Project: Cloud-Based IoT Management with SIM Card

Carrier Switch: Change connected carrier on device

Select a carrier to switch the SIM to in the dropdown and view the carrier status below.

Carrier: AT&T

Submit

- Current carrier: Unknown
- Current carrier switch status: Not yet started

Downstream Task on IoT Data: View livestream visualization of IoT Data

We use grafana to visualize the livestream of the IoT Data sent to the cloud. Click here to go to Grafana.

Acknowledgements

Developers: Harrison (harrison.cassar@gmail.com), Ricky (ricky99.guo@gmail.com), Disha (dmzambani@gmail.com), and Albert (albertstanley@g.ucla.edu)

Developed as a part of UCLA's Spring 2023 COM SCI 219: Cloud Computing course.

Special thanks to Mentor Jinghao Zhao and Professor Songwu Lu for guidance throughout the project development process.

Figure 6: Screenshot of the current state front-end of the cloud server web application.

In its current state, the front-end application served by the Main Server is a simple one-page HTML/CSS webpage. A screenshot of this page can be seen in Figure 6.

Additionally, this Flask server utilizes the "Flask-WTF" (or "Flask What-The-Forms") plugin[13] to enable easy setup and configuration of a basic form object to take in and validate user input for the carrier switch functionality.

2.2.6 *Grafana.*

Grafana stands as a stand-alone open-source analytics, monitoring, visualization web application, providing a very convenient and easy-to-customize interface for setting up custom dashboards to visualize data. In our system, we use Grafana for the sake of performing this "downstream task" of visualization on the IoT data we save and collect through the Kafka instance. We use Grafana almost entirely out-of-the-box, only deploying it with a custom configuration file that simply disables the login page (which otherwise normally just had the default login information of username "admin" and password "admin"). To be more specific, we utilize the Grafana Enterprise v8.2 Docker image for our system, utilizing this non-latest version to allow us access to a stable version that exposes Grafana Live and also has support for an existing Kafka data source plugin [8], which we describe more later in this section.

Our system sets up Grafana to obtain data via two methods: 1. through specific Kafka topics (to meet the requirements of our project), and 2. through streaming directly with Grafana Live.

Ideally, Grafana is able to pull data as a Consumer directly from the Kafka instance, and visualize it live, not needing any other alternative. We are using Grafana v8.2, which has a semi-working Kafka data source plugin[8] that essentially connects to Kafka, consumes messages from specified topics, and streams that data live for display with Grafana. Our system utilizes this plugin, meeting our project requirements. We note, however, that this plugin lacks some critical features that makes the live display not ideal for our purposes. This includes no support for selection of which data rows (in the Kafka message) to visualize (and which to simply ignore), instead opting for ALL data rows to visualize. This wouldn’t work for our purposes, as we desire to have a "timestamp" field in our Kafka messages. Additionally, it only supports to have data graphed with timestamp either determined by the message timestamp (when the Kafka message was pushed to Kafka) or when it was consumed from Kafka. This will not work for our purposes, as we desire to have the data graphed at the time that the IoT data was actually collected from the device, which is represented by the "timestamp" field that we’ve tagged our data with (in the Kafka message itself). In practice, we expect that the message timestamp is no more than a few seconds after the actual data timestamp, however this is fundamentally not ideal. We could upgrade the plugin, but this is out of the scope of this project. Therefore, as described further "Kafka Topic Configuration" section, we just setup a specific set of Topics meant specifically for this Grafana consumer that has a slightly-different format than the ideal message format (strips the "timestamp" field from our Kafka message, and having the plugin just graph with the message timestamp).

As mentioned previously, just to explore more options, we can also avoid Kafka altogether, simply streaming data directly to Grafana via Grafana Live. Grafana v8.0 introduced Grafana Live [9], a new streaming capability that allows us to push data to the UI in "near real-time". This is extremely efficient and lightweight (not needing any plugin and/or assisting backend), and can be achieved by simply performing an HTTP POST request to a specific URL in a specific format. For our project purposes, this does not need Kafka, and therefore does not satisfy project requirements. However, we still do this anyway just to demonstrate the functionality.

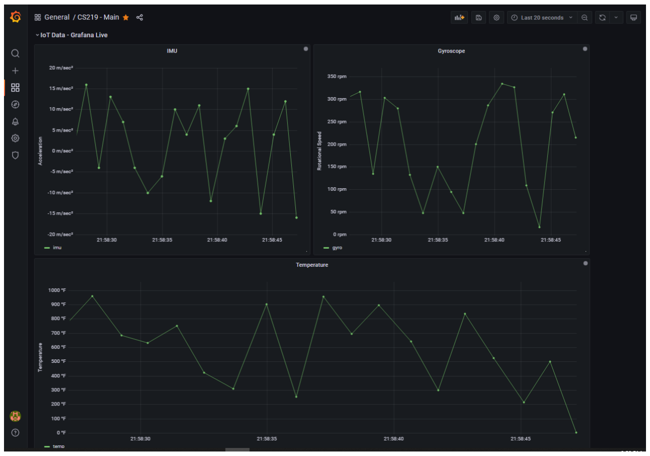


Figure 7: Screenshot of main Grafana Dashboard used to visualize our current-state mocked IoT Data.

For a screenshot of the dashboard setup for the current state of this project (with three mocked sensors supported), see Figure 7.

2.3 Subsystem Interfacing

In this section, we provide a description of the various communication methods that the systems' components utilize to interface with one-another into the full system.

2.3.1 Port Assignments. In order to present an easy and static means of inter-component communications, we hard-code assign the ports that each of the cloud components use in order to interact with each other. In our cloud-deployed context, we also expose a subset of these ports to allow remote (or "local" and "external", if we're considering this from the modem or user's perspective) access to these server components. This assignment is as follows:

- **Main (Flask) Server:** 8000
- **Zookeeper:** 2181
- **Kafka:** 29092 (Docker containers), 9092 (host), 9101
- **Grafana:** 3000
- **UDP Server:** 6001
- **Modem Client:** 6002 (locally hosted, not in Docker container)

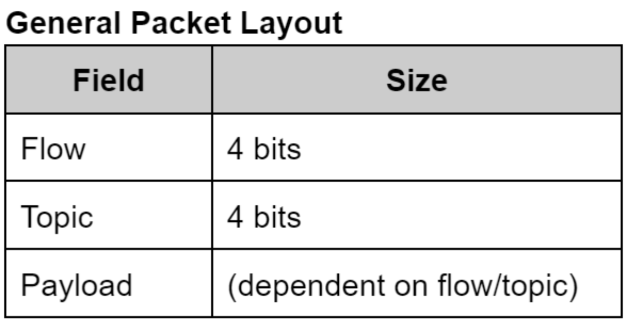
For the cloud-deployed context, we also expose the Grafana and Main Server ports to the external world to allow for remote connections.

2.3.2 SIM to Modem Communication. The modem and SIM communicate with each other through APDU commands. The typical communication flow begins with the modem sending an APDU request to the SIM card, and the SIM card responding with an APDU response. In proactive commands however, the SIM can initiate commands itself such as the OPEN CHANNEL, SEND DATA, and RECEIVE DATA commands. These proactive commands are presented to the modem every time a modem makes a FETCH command to the SIM card.

The OPEN CHANNEL command instructs the modem to setup the channel for future communication of future data packets. The RECEIVE DATA command tells the modem that the SIM card is ready for future communication. The SEND DATA command is used to send bytes from the SIM card to the modem. Finally, the TERMINAL RESPONSE command, which is not a proactive command, is used by the modem to send a response back to the SIM when these proactive commands are received.

2.3.3 UDP Protocol Definitions. In order to support the 3 data flows we desire to maintain in this project, we define a custom protocol with its own defined packet structure to be respected during data transmission over the UDP link connection between the cloud and local subsystems—with the UDP Server and Modem Client as the components-of-interest, respectively. Respective packet decoders and parsers are subsequently based off of these definitions, and used within both the Modem Client code and UDP Server code. Refer back to Figure 2 for a diagram of the overall system layout.

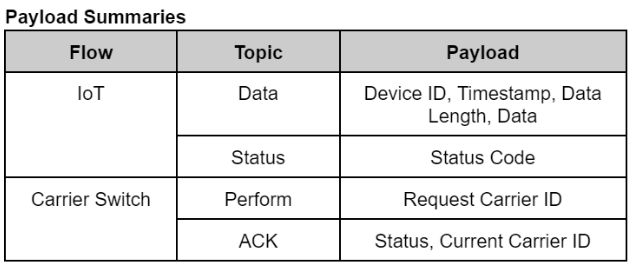
The general packet type can be seen in Figure 8, where specific packet types are determined by the values present in the "Flow" and "Topic" header fields. There currently exists 4 packet types: IoT Data, IoT Status, Carrier Switch Perform, and Carrier Switch ACK. These, along with a summary of their respective Payload field, can be seen enumerated in Figure 9.



General Packet Layout

Field	Size
Flow	4 bits
Topic	4 bits
Payload	(dependent on flow/topic)

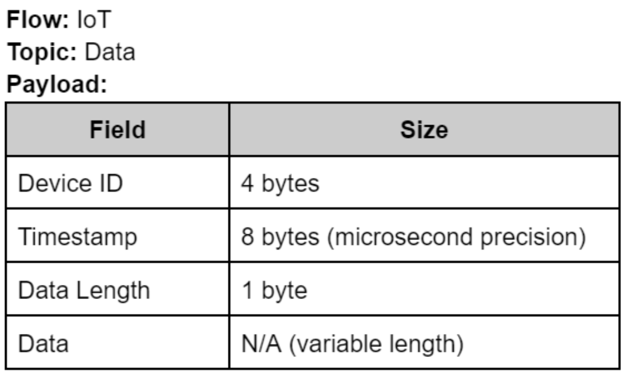
Figure 8: General packet structure for our custom UDP Protocol.



Payload Summaries

Flow	Topic	Payload
IoT	Data	Device ID, Timestamp, Data Length, Data
	Status	Status Code
Carrier Switch	Perform	Request Carrier ID
	ACK	Status, Current Carrier ID

Figure 9: Summary of Payload fields for all packets supported in our custom UDP Protocol.

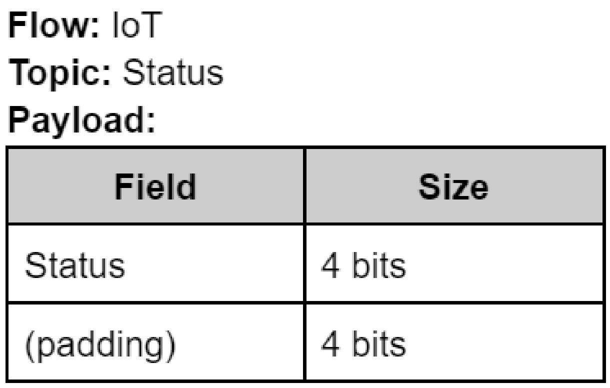


Flow: IoT
Topic: Data
Payload:

Field	Size
Device ID	4 bytes
Timestamp	8 bytes (microsecond precision)
Data Length	1 byte
Data	N/A (variable length)

Figure 10: Payload definition for the IoT Data Packet.

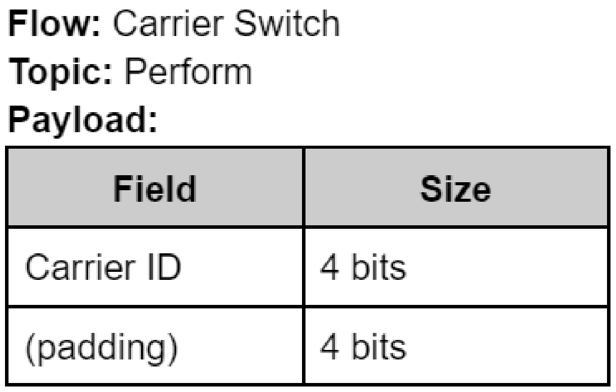
The IoT Data packets (as seen in Figure 10) contain data collected from IoT sensors (or, in our case, mocked and generated) to be sent upstream for storage and use by downstream tasks (e.g. visualization). The "Device ID" field represents a unique ID for the specific device in question, being used to distinguish the source (and type) of the data specified within the packet, while the "Timestamp" field represents the exact time (with support for microsecond precision, being packed in accordance to the Temporenc format [3]) that data was collected/polled from the IoT device. The "Data" field has variable length, where the structure is defined per-device



Flow: IoT
Topic: Status
Payload:

Field	Size
Status	4 bits
(padding)	4 bits

Figure 11: Payload definition for the IoT Status Packet (currently unused).



Flow: Carrier Switch
Topic: Perform
Payload:

Field	Size
Carrier ID	4 bits
(padding)	4 bits

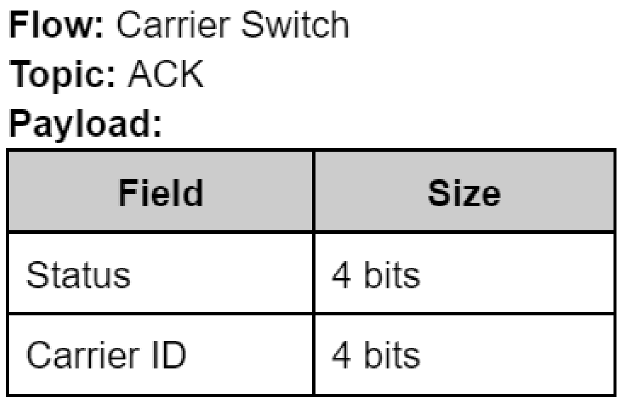
Figure 12: Payload definition for the Carrier Switch Perform Packet.

(for our project purposes, as a proof-of-concept, we assume that all devices send a singular integer of data of 4 bytes).

The IoT Status packets (as seen in Figure 11) contain status-related information for the managed IoT devices, where the "Status" field encodes some specific opcode that describes the current status of the device (i.e. "Nominal", "Idle", or "Off-Nominal"). For the current state of this project, this packet is not actively used, but still supported by the protocol and the implemented parsers.

The Carrier Switch Perform packets (as seen in Figure 12) represent the downstream request from the user to perform a carrier switch on the SIM card integrated with the local modem device. The "Carrier ID" field represents an integer ID for a number of supported carriers (i.e. "Verizon", "T-Mobile", and "AT&T").

The Carrier Switch ACK packets (as seen in Figure 13) represent the upstream ACK or NACK (as represented by the "Status" field) of the user's downstream Carrier Switch Perform Request. The "Carrier ID" field represents an integer ID for the now currently-connected carrier following the possible successful carrier switch.

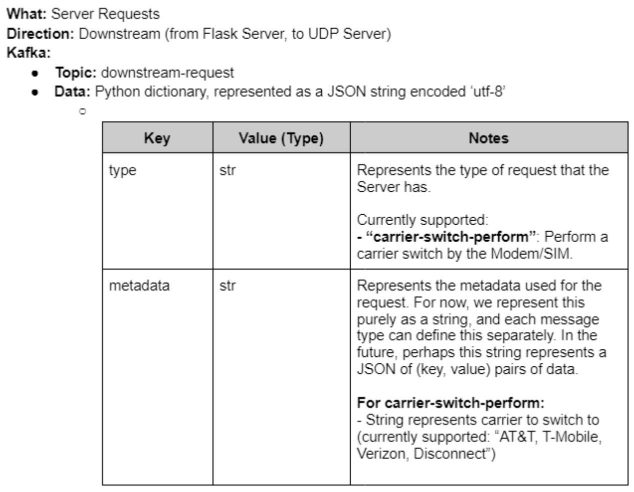


Flow: Carrier Switch
Topic: ACK
Payload:

Field	Size
Status	4 bits
Carrier ID	4 bits

Figure 13: Payload definition for the Carrier Switch ACK Packet.

2.3.4 Kafka Topic Configuration. Moving a bit further upstream, we further support the 3 data flows we desire to maintain in this project by defining a specific set of Kafka Topics that our various cloud components can become Producers/Consumers to/from. As expected, these interface definition is subsequently used and employed in the source code and configurations for most of the cloud components, including the Main Server, UDP Server, and Grafana. Refer back to Figure 2 for a diagram of the overall system layout.



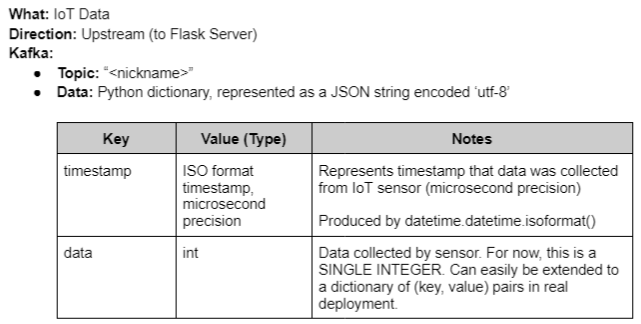
What: Server Requests
Direction: Downstream (from Flask Server, to UDP Server)
Kafka:
- **Topic:** downstream-request
- **Data:** Python dictionary, represented as a JSON string encoded 'utf-8'
    -

Key	Value (Type)	Notes
type	str	Represents the type of request that the Server has. Currently supported: **- "carrier-switch-perform"**: Perform a carrier switch by the Modem/SIM.
metadata	str	Represents the metadata used for the request. For now, we represent this purely as a string, and each message type can define this separately. In the future, perhaps this string represents a JSON of (key, value) pairs of data. **For carrier-switch-perform:** - String represents carrier to switch to (currently supported: "AT&T, T-Mobile, Verizon, Disconnect")

Figure 14: Message format definition for the "downstream-request" Kafka topic.

For our downstream data flow, we have the "downstream-request" topic, which contains all downstream requests from the user and Main Server to be sent down to the local Modem/SIM Client. The message format can be seen in Figure 14.

For our upstream data flows, we have the "carrier-switch-ack" topic, which contains all the ACK messages for carrier switch requests sent previously downstream to the Modem/SIM client. Additionally, we have a number of topics for all IoT data, separated

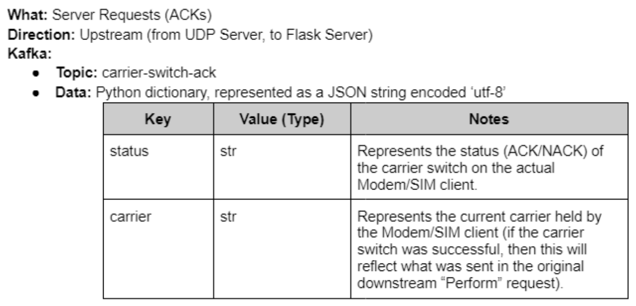


What: IoT Data
Direction: Upstream (to Flask Server)
Kafka:

- **Topic:** "<nickname>"
- **Data:** Python dictionary, represented as a JSON string encoded 'utf-8'

Key	Value (Type)	Notes
timestamp	ISO format timestamp, microsecond precision	Represents timestamp that data was collected from IoT sensor (microsecond precision) Produced by datetime.datetime.isoformat()
data	int	Data collected by sensor. For now, this is a SINGLE INTEGER. Can easily be extended to a dictionary of (key, value) pairs in real deployment.

Figure 15: Message format definition for the IoT Data-related Kafka topics.



What: Server Requests (ACKs)
Direction: Upstream (from UDP Server, to Flask Server)
Kafka:

- **Topic:** carrier-switch-ack
- **Data:** Python dictionary, represented as a JSON string encoded 'utf-8'

Key	Value (Type)	Notes
status	str	Represents the status (ACK/NACK) of the carrier switch on the actual Modem/SIM client.
carrier	str	Represents the current carrier held by the Modem/SIM client (if the carrier switch was successful, then this will reflect what was sent in the original downstream "Perform" request).

Figure 16: Message format definition for the "carrier-switch-ack" topic.

and named by their device type/nickname (i.e. "imu", "gyro", and "temp"). The message formats for these topics can be seen in Figures 16 and 15, respectively. Due to the specific issues with the Kafka data source Grafana plugin we use in this project (as described earlier in the "Grafana" section), we also have a special extra set of Grafana-specific topics named as "<nickname>-grafana", which achieve the same format as the regular IoT data-related Kafka topics but without the "timestamp" field. In an ideal implementation, these will be scrapped and not used, where we serve to simply fix and adjust the Kafka Grafana plugin for our project's purposes.

We provide a bit more motivation for this organization of Kafka Topics. First of all, why do we split our IoT data among different topics? We do this by a convention that is recommended for a Kafka Cluster: We don't care about the order of processing one sensor's data over the other (we have timestamps in the actual data we Produce to a Kafka Topic), so we just want it processed as fast as we can. Additionally, separate message types should, by convention, be in separate topics, as these may have different encoding and message formats, which changes how we decode from a certain Topic stream. This thus allows a nicer form of horizontal scalability.

Second of all, why do we keep all server requests (from our Main Server to the UDP Server) in one topic? We again follow a convention for Kafka Clusters: We want to ultimately provide the guarentee to the Main Server that the UDP Server will handle one request that was sent BEFORE a later request that was sent. In order words, we do care about the ordering of requests/handling of them. Although we do not currently support any other requests other than a "Carrier Switch Perform", perhaps we could imagine the Main Server wants to change the polling rate of an IoT sensor, but we only want to do this AFTER a "Carrier Switch Perform" because the IoT sensor depends on a certain carrier being selected. These different request types can be easily encoded in some "type" parameter in the message pushed to this Kafka "downstream-request" topic.

2.4 Cloud Deployment

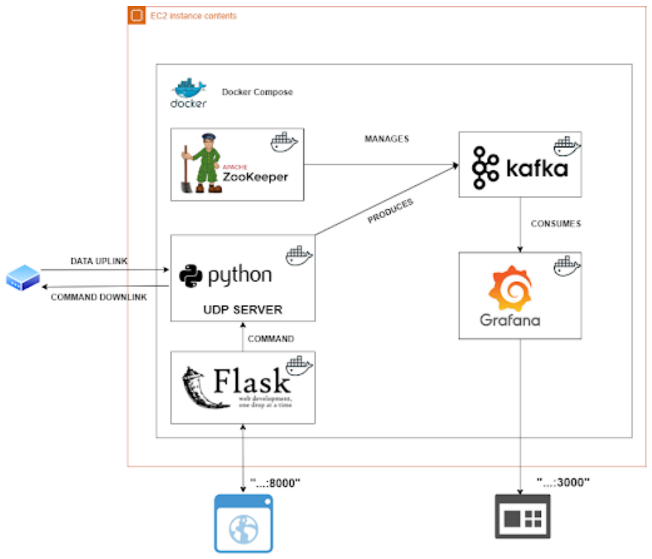


Figure 17: Cloud deployment diagram.

We deployed our Dockerized services onto the AWS public cloud on a single t2.large EC2 Instance equipped with 2 vCPUs 16 GiB RAM[11]. Previously, we attempted to use the weaker (but free-tier-eligible t2.micro instance) but found that it had insufficient memory to support Kafka. Although our deployment was to a single instance on the cloud, the Dockerized containers can be easily separated into their own instance in the future. To spin up the instances, we run `docker compose up` which individually instantiates each service, ordering the dependents after the dependencies have been successfully started. We utilize a Docker bridge network to interlink all the communication between the services which, on a single instance, still communicate through HTTP/TCP/UDP. Although this adds a non-trivial load to the CPU, this form of communication scales out naturally with partitioning our services to their own separate instances and communication details are once again abstracted away by Docker's bridge network.

2.4.1 Avoiding Vendor Lock-in. For future cloud deployments, we have made our system independent of the exact cloud vendor we use due to our contained system. For instance, we could easily migrate to Google Cloud Platform (GCP) or Microsoft Azure should we choose to do so, spinning up our cloud services on their respective instance classes. This was a conscious decision we made. By making our design as generic as possible, running on an instance instead of a managed service (e.g. Managed Kafka, Managed Grafana, etc.) we avoid locking ourselves into a specific vendor and make our implementation completely open. However, one trade-off in this design is that cost may be higher. Managed instances often come with a pay-as-you go model[12] as compared to a flat hourly use cost. Given our low usage in our proof-of-concept deployment, the pay-as-you-go model would seem to be favored from an economical perspective, but this difference is likely diminished with greater usage. Further, the managed services offer performance tuning and some auto-scaling. We think that we can achieve similar performance and by cleverly selecting our choice of instances after partitioning out our services to different and relying on Docker Swarm Mode[7] to scale on our behalf.

2.4.2 A Simple Scale Out Strategy. Our scale out strategy is simple which is a testament to the flexibility of fully Dockerized system components. We will simply partition out our services to their own instances/clusters and let Docker to handle all the finer-grained details. The Docker Bridge Network will continue to automatically link efficient inter-communication across instances allowing us to avoid any manual configuration of networking. While we haven't deployed Docker Swarm in our single instance deployment proof-of-concept, doing so will abstract away the deployment of our services across multiple instances. While conventional wisdom in scaling is to attempt scaling up before scaling out, our system is designed in such a way to favor a scale out strategy. For instance, Kafka favors being deployed over clusters as compared to a single instance to achieve better resilience and performance. Consequently, scaling out is the natural direction of our work moving forward.

2.5 Implementation Challenges and Quirks

2.5.1 Docker Local Development Environment Issues. One minor pain-point we faced for our team members developing on Windows OS machines was a known memory leak problem which, as of writing this, still has not been resolved. This led to some surprising crashes on their systems while developing and forced them to restart every so often to prevent memory leaks from becoming too severe. We note that this bug does not affect our cloud deployment.

2.5.2 Thorough Testing of Subsystems. One major hurdle encountered in development was the testing of some subsystems required the development of others (which may or may not be completed yet). While unit tests make sense in some contexts, the majority of our system required integration tests to make sure our services were communicating with each other appropriately. To resolve this, we initially worked top-down to craft well-defined interfaces and mock out dependent systems. For instance, we mocked out the modem to be able to test the cloud-deployed services in isolation. However, though we believe this approach was correct, it was not without its difficulties as it added a lot of additional work for us.

2.5.3 UDP Downlink Communication due to CGNAT. One surprising problem that we encountered while deploying to the remote system was the CGNAT apparent in our ISP. This made downlink communication from the cloud to the modem non-trivial as we noticed our UDP packets dropping. We initially attempted to open two separate UDP sockets to listen and send data through on both our UDP server and our modem. However, UDP packets from the UDP server to the modem kept dropping because we relied on a static IP address/port of our modem. With CGNAT, the IP address seemed to be fixed but the port was constantly changing within a minute. To resolve this, we switched to using the same socket on both ends to handle both listening and sending packets and (at the UDP server) caching and using the most recent modem IP/port that the server received data from. This allowed the downlink packets to traverse the NAT successfully and avoid being dropped.

2.5.4 Interactive UI. For our demo, we served a simple HTML page from our Flask server that was able to perform the basic tasks (e.g. carrier switch) at hand. Still, we ran into some hurdles while attempting to develop with ReactJS and Kafka. Initially, we had hoped to have some tabular view of the modems using ReactJS and allowing for more interactivity with the Kafka data. However, in our experience, Kafka did not integrate very well with our attempted ReactJS frontend. We attempted to connect our ReactJS with the remote server using two methods: using a NPM module to interface with our cloud Kafka service and using Flask as a proxy to fetch data from Kafka. However, both methods did not end up working so we elected to continue using our simple HTML page with some added CSS.

3 CURRENT STATUS AND LIMITATIONS

3.1 Current Status

We have completely implemented our system to specification according to our original goals and have even deployed to cloud. Our project mentor saw the cloud deployment as a reach goal so we are glad to have accomplished this. Specifically, our non-modem systems exist in the cloud and we can concurrently run modems locally to interface with the cloud. The UI that we serve, whether through Grafana or Flask, accomplishes the goals we aimed for. Grafana serves as a visualization framework for our data and our Flask-served UI can successfully execute the carrier switch command through the cloud.

3.2 Limitations

However, our deployment is not without its limitations. We still only support a single modem carrier switch. We note here that we were limited to a single SIM card so we did not test beyond that. Furthermore, it would be seemingly trivial to extend to more modems. Another reach goal which we did not accomplish was performing more downstream tasks beyond visualization. This could take the form of perhaps some Machine Learning pipeline connected to the data streamed in from Kafka. As our data is currently randomly generated (as was the original intent) and due to our time limitations, we did not pursue this goal further. Given real data, this could be a direction of future work. Another limitation we encountered was the 1Hz bottleneck on receiving IoT data for Kafka. We were not completely certain what the bottleneck was but it appeared to be due to Kafka. This would not be surprising given that we only deployed Kafka to a single instance, and the single instance was severely under the recommended specification of the open-source Confluent Kafka that we used[5]. We presume that this bottleneck could be easily lifted if this Kafka service were to be productionized and partitioned across many powerful instances as compared to our single EC2 t2.large instance.

4 EVALUATION AND DEMO

Our demo worked with a local modem client and with the remaining components deployed on the cloud. These remaining components included the Kafka service, the main Flask server, the USP server and Grafana. We began by making a carrier switch request on the frontend to Verizon. The frontend displayed that a carrier change was in progress. In showing the logs on the modem-client we showed that the SIM received a carrier switch request packet and passed back an carrier switch ACK packet. When returning to the frontend, we showed that the current carrier had swapped to Verizon.

The next component of the demo involved showing that data could be sent from the modem-client to the cloud. The modem-client continuously streams data from the three simulated sensors. We navigated to the Grafana section of the frontend to display that the data for the three different sensors could be visualized on three separate graphs.

A demo of the local deployment can be found here: https://www.youtube.com/watch?v=eUHK2VMmx1w&ab_channel=AlbertStanley

5 DISCUSSION AND CONCLUSIONS

We found that Apache Kafka is extremely flexible and fits the distributed context very well. Practically, Kafka is great for event streaming and for tasks with real time requirements. For example, live monitoring of critical systems are great with Kafka. An early focus on dockerizing the components made the deployment process much simpler later on in the project. This made managing dependencies easier for local development environments in addition to the cloud. During the course of the project, maintaining documents outlining the interfaces between the different components enabled parallel work that sped up the development of the entire pipeline. Overall, the project highlighted the benefits of a stream-processing platform like Kafka in addition to the importance of following good practices to speed up the development of cloud services.

6 FUTURE WORK

In this section, we propose a number of additional possible experimentations, explorations, and expansions to this project, all aiming to even further support our original task of enabling management of SIM-enabled IoT devices via a cloud-based system.

6.1 Extensions to Support

We would like to support multiple modem clients per user as well as multiple users to be able to manage their own suite of devices. This could require an extension of possibly expanding the number of Kafka topics produced to and having a set of Kafka topics subscribed to per user. On the main Flask server, we would additionally create a user management system where users can login to view and manage their devices and SIM cards. We can extend current metrics to include SIM and modem metrics such as power and speed as well as add a metrics management feature on the main Flask server to allow for configuring, adding, and deleting of tracked modems and/or IoT devices.

6.2 Cleaner System

In our current implementation, we have many components that could be unified to minimize slower over-network traffic. To achieve this integration, further inspection of the current system would be necessary. We would also like to develop a more functional and interactive UI. Our main Flask server has limited capability to develop such a UI and, so, we would have to consider moving our main server to a more capable front-end server such as React with Nodejs or building a front-end with React and Nodejs for UI uses and having it communicate with the current Flask server as a proxy. We would also extend the front-end to reflect features mentioned in the Extensions to Support subsection.

6.3 Closer to a Real-world Scenario

Our current implementation mocks IoT device data and a simulates a modem. In the future, we would like to implement this system using an actual modem with our 5G multi-carrier SIM card and connect actual IoT devices to the device. We would also like to implement more relevant downstream tasks on Kafka such as data analytics for our IoT devices.

ACKNOWLEDGMENTS

A special thanks to our project mentor, Jinghao Zhao, for not only making us excited about this project's content, but also assisting us endlessly throughout the project development in many different ways. This work could not have been done without him.

REFERENCES

[1] [n.d.]. eSIM-loader. https://github.com/JinghaoZhao/eSIM-Loader
[2] [n.d.]. Wireless Networking Group. http://metro.cs.ucla.edu/
[3] Wouter Bolsterlee. [n.d.]. Temporenc. https://temporenc.org/
[4] Inc. Confluent. [n.d.]. Confluent Kafka Documentation. https://developer.confluent.io/
[5] Inc. Confluent. [n.d.]. Confluent Platform System Requirements. https://docs.confluent.io/platform/current/installation/system-requirements.html
[6] Inc. Confluent. [n.d.]. Confluent's Python Client for Apache Kafka. https://pypi.org/project/confluent-kafka/
[7] Docker Inc. 2023. Swarm mode overview. https://docs.docker.com/engine/swarm/
[8] Hamed Karbasi. [n.d.]. Kafka Datasource for Grafana. https://grafana.com/grafana/plugins/hamedkarbasi93-kafka-datasource/
[9] Grafana Labs. [n.d.]. Grafana Live Docs. https://grafana.com/docs/grafana/latest/setup-grafana/set-up-grafana-live/
[10] Pallets. [n.d.]. Flask Documentation. https://flask.palletsprojects.com/en/2.3.x/
[11] Amazon Web Services. 2023. Amazon EC2 T2 Instances. https://aws.amazon.com/ec2/instance-types/t2/
[12] Amazon Web Services. 2023. Amazon Managed Streaming for Apache Kafka (MSK). https://aws.amazon.com/msk/
[13] WTForms. [n.d.]. Flask-WTF Documentation. https://flask-wtf.readthedocs.io/en/1.0.x/

A TECHNICAL DOCUMENTATION

- APDU Reference Guide: https://docs.yubico.com/yesdk/users-manual/yubikey-reference/apdu.html

B SOURCE CODE

For all source code developed and utilized as a part of this project, please refer to the following git repository: https://github.com/harrisonCassar/CS219-Cloud-IoT-Management-SIM-Card.