

## **Problem Definition and Objectives**

A lot of small businesses, households and group offices do not find it easy or cost-efficient to keep a log of who has come to their front door. Many people use normal security cameras that watch events continuously. So, people either need to use hard drive space for the videos by saving them on their local DVR or use bandwidth by sending them to the cloud. What's more, it takes a lot of time and gets quite frustrating to search through many hours of footage for one scene. For this reason, a lot of these places give up on real security and use only basic security cameras that trigger when something moves.

Our solution is to build a Smart Door Security & Visitor Logging System which has these main purposes:

### **Face detection at the edge (on devices)**

We set up Raspberry Pi and a Pi Camera to recognize faces without using cloud services. It will only save a few images when it detects a face instead of making entire video recordings. Format conversions help reduce the need for a lot of bandwidth and storage space.

### **Cloud Logging and Cloud Analytics**

All the snapshots are sent to Google Cloud Storage. Also, we pass along the person's name, image file name and timestamp using HTTP to a Cloud Function. The information is stored in BigQuery by that function. Since everything is saved as events and not as full videos, we save storage expenses and make data analysis faster through SQL.

### **Real-Time Alerts**

If there is someone at the gate the system does not recognize, it instantly sends you an alert with a picture and the time through Discord. Using this, admins are alerted if anything needs attention without examining things manually.

### **The secure Web Dashboard.**

We made a web dashboard using Flask and added security with Firebase Authentication. Only those who are admin can use these features when logged in. See the last 10 dogs recorded by time and name (including a photo). Get a chart each day with the number of detections made. Control whether the Raspberry Pi is detecting faces or not over the internet. Inspect the full log file to find any problems.

## **Research Existing Solutions and Methodologies**

Systems like Ring, Nest Cam and Arlo which are popular, often use a non-stop feed of video to their cloud service. These cameras film when they see motion and keep the videos for people who pay for the service. Even though some of them can identify faces, they need a lot of the internet and space for storage which costs a lot. Besides, most of their important tools, for instance improved analytics, require paid memberships.

Many developers also choose to run everything straight on a Raspberry Pi using OpenCV. Photographs of people are detected and stored on a USB drive or a shared folder in the network with this configuration. All connections are local which makes the music cheaper to download. The biggest risk is that if the Pi is taken, stops working or loses power, everything saved will be lost. It's difficult to put together or analyze information from several devices without much extra effort.

A few bigger companies rely on a more advanced approach by using Google Cloud Pub/Sub. Detected events are sent to a Pub/Sub topic by edge devices and then they are pushed into BigQuery by a Cloud Function or Dataflow job. Doing this gives the device more stability and keeps it out of BigQuery, but it is also more complex to get ready. You need to set up service-account keys just to make it work which is not necessary for a basic prototype with only one device.

So we opted for a basic setup with the Raspberry Pi, an authorized service-account JSON key and an HTTP POST carrying a JSON message (containing timestamp, person's name and image name). It saves the information in BigQuery and also lets Discord know right away if a face is not recognized. It allows us to put our network together quickly, the delay is almost nonexistent and we can handle more Pis without needing to redesign much.

## **Explanation of Why an IoT-Cloud Approach Is Suitable**

This approach of combining IoT with the cloud delivers the best results for us since we get advanced computing locally and access to special tools from the cloud.

### **Edge Processing Reduces the Amount of Data That is Stored**

Because the process of face detection and recognition runs on the device itself, we can send only a small amount of information (approximately 20–50 KB) and basic data to the cloud. It's much more efficient than playing a video every time which can use up a lot of data in no time.

Also, instead of storing all the data, we keep only summaries of events which saves money on cloud storage.

### **Leaning on Cloud Services Allows for Scaling While Keeping Things Easy**

If we attach many Raspberry Pis, Google Cloud Functions automatically increase resources so we do not have to handle extra servers. Each second, BigQuery can process thousands of pieces of data, so running quick queries for daily visitor counts or appearances is possible. Firebase Authentication takes care of user login security in the web dashboard without us needing to develop our own login system. Also, with Google Cloud IAM roles, each part of our system can access only the resources it requires which makes our system safer.

## **System Design**

### **Hardware and Cloud Services Selection**

The system uses a Raspberry Pi 4B and a Pi Camera Module 3 NoIR for face detection. Real-time facial recognition on the Pi 4B (4 GB RAM) is possible with the face\_recognition library. It is budget-friendly, at only \$85 NZD and its energy-efficient design is good for space heaters.

We opted for the Camera Module 3 NoIR because it does well in low-light conditions when compared to earlier versions. Since this camera has no IR-cut filter, it's good for use with infrared lighting at dawn or dusk. Video can be captured at 640×480 resolution (or more), with up to 90 frames appearing each second on the screen. This is enough to detect faces with every image without causing the Pi to slow down.

### **Cloud Services – Google Cloud**

By using Google Cloud, we can count on storage, processing and alerts, without needing to take care of our own IT infrastructure.

### **Google's cloud storage (GCS)**

If the Pi detects a face, it puts the photo in a storage bucket called doorlogger-images. To allow direct uploads, we gave the Pi the "Storage Object Creator" role. We also made the folder

public, so we can add links to images to the web dashboard and messages without needing to create temporary signed URLs.

## BigQuery

A data set called `doorlogger_logs` was built, as well as a table called `face_events` with three columns—name, image file and timestamp. When detection is identified, Pi is able to use the `log_face_entry` function to push it straightaway into BigQuery. We design the tables to be divided by date which makes looking up daily data easier and quicker.

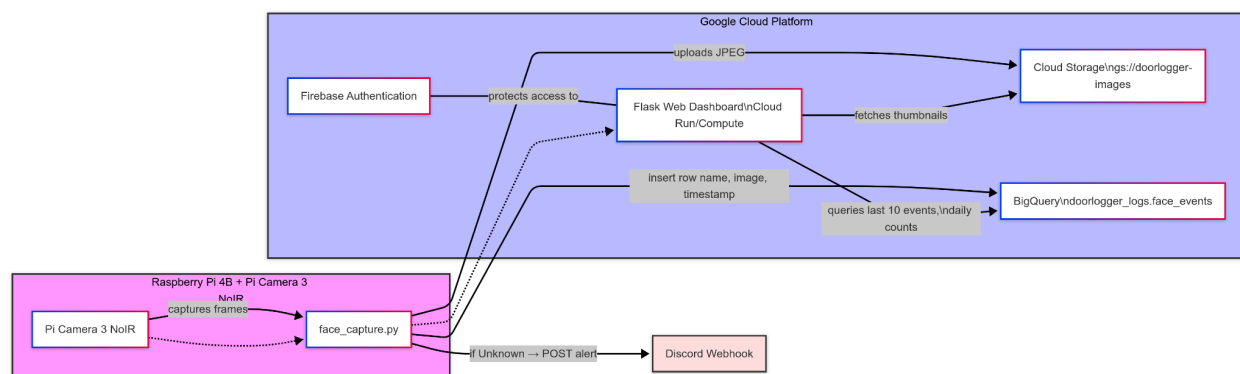
## Firebase Authentication

Flask dashboard security is handled through Firebase Authentication. Users have to use their email and password to log in as an admin. Making use of Firebase Admin SDK on the backend, the server authenticates a user with the help of `auth.verify_id_token` on the Firebase server. Because of this, we don't have to create a login and only the approved admins can get to the dashboard or control the Pi.

## Discord Webhook

A Discord channel is notified in real time by the Pi when it spies unfamiliar people (i.e., those marked as “Unknown”). Cloud Storage provides the image link in the message, so it is easy for administrators to know who sent it without relying on email or SMS.

## System Block Diagram Illustrating All Components and Their Interactions



## Data Flow and Network/Communication Protocols

All links between the Raspberry Pi and Google Cloud are done securely with encrypted HTTPS (TLS). After the Pi's face capture program spots a face, it creates a local JPEG copy and then uploads the image over HTTPS to `gs://doorlogger-images/captures/` in the Cloud. After this, the

script uses gRPC (over TLS) or HTTPS to transmit a JSON row (person\_name, image\_name, timestamp) into the BigQuery table by calling `bq_client.insert_rows_json(...)`. If the detected person's name is "Unknown", the Pi creates an HTTPS POST to Discord's webhook with the image file included in the request, so that Discord displays the photo in the channel. All of these processes depend on the permissions given through the Pi's Service account (based on the Google Application Credentials), allowing only the specified account to make uploads or inserts to the GCS bucket and BigQuery table.

Administrators reach the Flask web interface at the administration side using HTTPS. Every request has a Firebase ID token (in an `Authorization: Bearer <token>` header or secure cookie) and Flask makes sure the token is genuine using the Firebase Admin SDK on a secure connection with Google's token verification endpoint. Once the user is authenticated, Flask sends gRPC (or HTTPS) requests to BigQuery to get the recent events and daily counts. The information is sent back to the browser over a TLS connection and displayed as HTML and JavaScript (Chart.js). It adds basic links to each file in GCS (such as `https://storage.googleapis.com/doorlogger-images/captures/<filename>`) and when your browser displays these links, they get downloaded using HTTPS. Files are protected during transfer with encryption and IAM roles and Firebase Auth restrict unauthorized access all along the process.

## Prototype Implementation (Cloud Backend)

### Data Ingestion

When the Raspberry Pi's camera spots a face, data ingestion gets underway at once. This is all laid out in the script which is at:

***/home/amaz/doorlogger/face\_capture.py***

When the Python script moves forward, it turns on the Pi Camera 3 NoIR to film at 640×480 and saves the video frames. The frame is given to the `face_recognition` library which detects all faces using `face_locations()` and then makes a 128-value code for each face using `face_encodings()`.

If there's a face match, the script checks if it is among known faces in `/home/amaz/doorlogger/known_faces/`. Should the face match the picture, marks it with "Amaz," but if no match is found, it writes "Unknown" on the sticker.

As soon as a label is picked, the script makes a new filename with the name and the UTC date—such as `Amaz_20250602_134512.jpg` or `Unknown_20250602_134515.jpg`—and saves the image into the `captures/` folder:

**`/home/amaz/doorlogger/captures/`**

With `cv2.imwrite()` from OpenCV, the script can write the image to the Pi's hard drive. If a network problem appears, the snapshot will still be saved locally.

As soon as the JPEG is in `/home/amaz/doorlogger/captures/`, the script sends it to Google Cloud Storage. In the `face_capture.py` file (starting at line 50 and going up to line 58), user credentials are obtained from a special environment variable.

**`GOOGLE_APPLICATION_CREDENTIALS="/home/amaz/doorlogger/doorlogger-auth-firebase-adminsdk-fbsvc-0763fc5a09.json"`**

This variable is loaded from the file kept in the directory `/home/amaz/doorlogger/.env`. The service account has a key file called

**`doorlogger-auth-firebase-adminsdk-fbsvc-0763fc5a09.json`** and this key only gets the “Storage Object Creator” role for the bucket `doorlogger-images`.

After everything is ready, the script transfers the image securely over HTTPS (TLS) with the help of the credentials in the key file.

```
storage_client = storage.Client.from_service_account_json(os.getenv("GOOGLE_APPLICATION_CREDENTIALS"))
bucket = storage_client.bucket(GCS_BUCKET_NAME)
```

```
blob = bucket.blob(f"captures/{filename}")
blob.upload_from_filename(filepath)
print(f"[CLOUD] Uploaded {filename} to GCS.")
```

As soon as the code runs, you see the JPEG image:

**`gs://doorlogger-images/captures`**

Thanks to the `captures/` prefix, an accessible URL can be used by any other service (including the Flask dashboard and Discord) to fetch the image without requiring signed URLs.

**`https://storage.googleapis.com/doorlogger-images/captures/amaz_20250530_042028.jpg`**

Directly following the Cloud Storage upload, the program runs `log_face_entry(filename, name)` which is a helper function in `/home/amaz/doorlogger/bq_logger.py`. That helper also authenticates to BigQuery using the same JSON file to perform a streaming insert into the table

Filter Enter property name or value								
<input type="checkbox"/>	Field name	Type	Mode	Key	Collation	Default Value	Policy Tags ?	Description
<input type="checkbox"/>	image_name	STRING	NULLABLE	-	-	-	-	-
<input type="checkbox"/>	person_name	STRING	NULLABLE	-	-	-	-	-
<input type="checkbox"/>	timestamp	TIMESTAMP	NULLABLE	-	-	-	-	-

Because `DATE(timestamp)` is used as a partition, BigQuery knows where to put every new row. `Bq_logger.py` contains the following insertion code:

```
import os
from google.cloud import bigquery
from dotenv import load_dotenv
from datetime import datetime

load_dotenv()

project_id = os.getenv("BIGQUERY_PROJECT_ID")
dataset = os.getenv("BIGQUERY_DATASET")
table = os.getenv("BIGQUERY_TABLE")

table_id = f"{project_id}.{dataset}.{table}"

bq_client = bigquery.Client(project=project_id)

def log_face_entry(filename, name):
    row = {
        "image_name": filename,
        "person_name": name,
        "timestamp": datetime.utcnow().isoformat()
    }

    errors = bq_client.insert_rows_json(table_id, [row])
    if errors:
        print(f"[BQ ERROR] {errors}")
    else:
        print("[BQ] Logged face event.")
```

An error message starting with `[BQ ERROR]` is displayed in the console when inserting data into BigQuery, if something doesn't go as expected. Messages can also be sent to `capture.log` which allows you to identify the problem later. It's important for the service account (`doorlogger-auth-firebase-adminsdk-fbsvc@doorlogger-auth.iam.gserviceaccount.com`) to have

the BigQuery Data Editor role for the BigQuery table  
door-logger-pi:doorlogger\_logs.face\_events for the processes to function properly.

The instant a script detects an “Unknown Face”, details are delivered to a Discord group. The code inside face\_capture.py (lines 60–67), once the JPEG image is saved and uploaded, accesses it and posts it over a secure connection to the Discord webhook URL which is configured in the environment file at /home/amaz/doorlogger/.env.:

```
if name == "Unknown":
    with open(filepath, "rb") as f:
        response = requests.post(
            DISCORD_WEBHOOK_URL,
            files={"file": (filename, f)},
            data={"content": f"Unknown face detected at {timestamp}"}
        )
    print(f"[DISCORD] Notification sent, status: {response.status_code}")
```

Because Google Cloud Storage already has the JPEG, the image is loaded into Discord quickly without lags. Everything on Discord is transmitted through HTTPS, so everything remains secure as it is sent.

During the process—scanning for a face, storing the image, uploading it to GCS, logging it in BigQuery and sending an alert (should it be required)—all steps are protected by TLS. The system runs on one service account key file (doorlogger-auth-firebase-adminsdk-fbsvc-0763fc5a09.json) which gives access just to upload images to the doorlogger-images bucket and write to the doorlogger\_logs.face\_events table in BigQuery. It ensures safety and also makes things straightforward and fast.

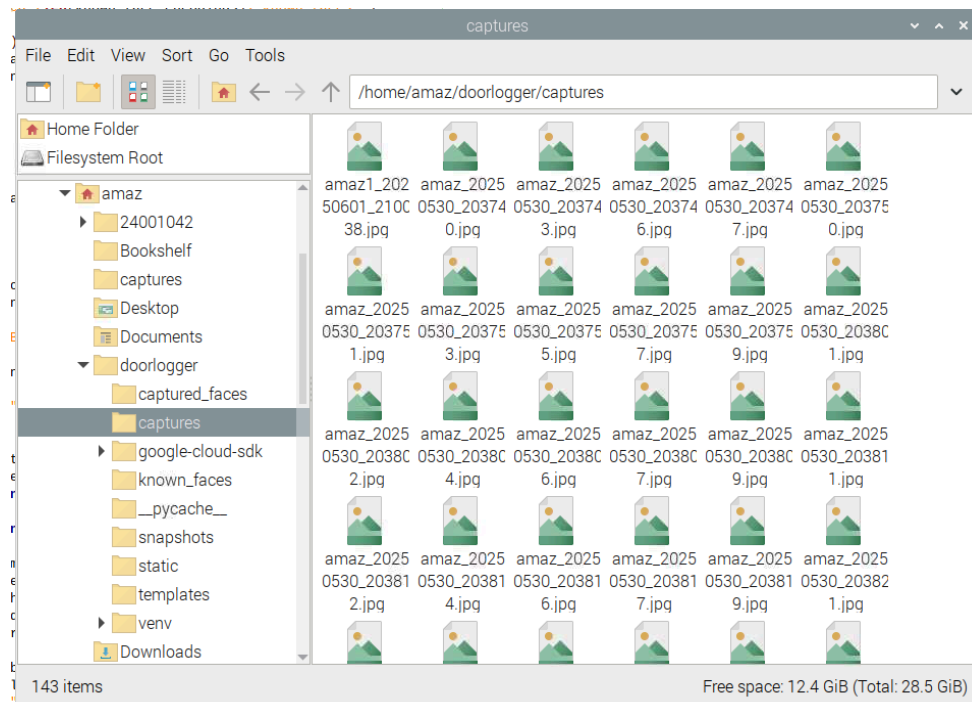
## Screenshots:

Terminal Running for face\_capture.py



```
amaz@raspberrypi: ~/doorlogger
File Edit Tabs Help
[DEBUG] Detected 1 face(s)
[INFO] Welcome back, amaz
[CLOUD] Uploaded amaz_20250602_212413.jpg to GCS.
[BQ] Logged face event.
[DEBUG] Detected 1 face(s)
[INFO] Welcome back, amaz
[CLOUD] Uploaded amaz_20250602_212415.jpg to GCS.
[BQ] Logged face event.
[DEBUG] Detected 1 face(s)
[INFO] Welcome back, amaz
[CLOUD] Uploaded amaz_20250602_212416.jpg to GCS.
[BQ] Logged face event.
[DEBUG] Detected 1 face(s)
[INFO] Welcome back, amaz
[CLOUD] Uploaded amaz_20250602_212418.jpg to GCS.
[BQ] Logged face event.
[DEBUG] Detected 1 face(s)
[INFO] Welcome back, amaz
[CLOUD] Uploaded amaz_20250602_212420.jpg to GCS.
[BQ] Logged face event.
[DEBUG] Detected 0 face(s)
[DEBUG] Detected 0 face(s)
[DEBUG] Detected 0 face(s)
```

## Local Captures Folder:



## Cloud Storage Bucket in GCP Console

Google Cloud | door-lagger-pl | Search (/) for resources, docs, products, and more

Cloud Storage | Bucket details | us (multiple regions in United States) | Standard | Public to internet | Soft Delete

Overview | Buckets | Monitoring | Settings | Storage Intelligence | Insights datasets | Configuration

Folder browser | Objects | Configuration | Permissions | Protection | Lifecycle | Observability | Inventory Reports | Operations

doorlogger-images | captures/ | Create folder | Upload | Transfer data | Other services

Filter by name prefix only | Filter objects and folders

Name	Size	Type	Created	Storage class	Last modified
Unknown_20250530_044249.jpg	74.2 KB	image/jpeg	May 30, 2025, 4:42:50 AM	Standard	May 30, 2025, 4:42:50
Unknown_20250530_044252.jpg	76.2 KB	image/jpeg	May 30, 2025, 4:42:52 AM	Standard	May 30, 2025, 4:42:52
Unknown_20250530_044254.jpg	76.1 KB	image/jpeg	May 30, 2025, 4:42:55 AM	Standard	May 30, 2025, 4:42:55
Unknown_20250530_044257.jpg	74.3 KB	image/jpeg	May 30, 2025, 4:42:57 AM	Standard	May 30, 2025, 4:42:57
Unknown_20250530_175809.jpg	79.5 KB	image/jpeg	May 30, 2025, 5:58:09 PM	Standard	May 30, 2025, 5:58:09
Unknown_20250530_195335.jpg	66.3 KB	image/jpeg	May 30, 2025, 7:53:35 PM	Standard	May 30, 2025, 7:53:35
Unknown_20250530_195347.jpg	66.3 KB	image/jpeg	May 30, 2025, 7:53:48 PM	Standard	May 30, 2025, 7:53:48
Unknown_20250530_195412.jpg	61.7 KB	image/jpeg	May 30, 2025, 7:54:12 PM	Standard	May 30, 2025, 7:54:12
Unknown_20250530_195416.jpg	62.6 KB	image/jpeg	May 30, 2025, 7:54:17 PM	Standard	May 30, 2025, 7:54:17
Unknown_20250530_195419.jpg	61.9 KB	image/jpeg	May 30, 2025, 7:54:19 PM	Standard	May 30, 2025, 7:54:19
Unknown_20250530_200142.jpg	73 KB	image/jpeg	May 30, 2025, 8:01:43 PM	Standard	May 30, 2025, 8:01:43
Unknown_20250530_200147.jpg	64.5 KB	image/jpeg	May 30, 2025, 8:01:47 PM	Standard	May 30, 2025, 8:01:47
Unknown_20250530_200227.jpg	77.7 KB	image/jpeg	May 30, 2025, 8:02:27 PM	Standard	May 30, 2025, 8:02:27
Unknown_20250530_200231.jpg	56.3 KB	image/jpeg	May 30, 2025, 8:02:32 PM	Standard	May 30, 2025, 8:02:32
Unknown_20250530_200238.jpg	55.9 KB	image/jpeg	May 30, 2025, 8:02:38 PM	Standard	May 30, 2025, 8:02:38
Unknown_20250530_200240.jpg	51.8 KB	image/jpeg	May 30, 2025, 8:02:41 PM	Standard	May 30, 2025, 8:02:41
Unknown_20250530_200306.jpg	51.6 KB	image/jpeg	May 30, 2025, 8:03:06 PM	Standard	May 30, 2025, 8:03:06
Unknown_20250530_200308.jpg	51.2 KB	image/jpeg	May 30, 2025, 8:03:08 PM	Standard	May 30, 2025, 8:03:08
Unknown_20250530_200310.jpg	51.1 KB	image/jpeg	May 30, 2025, 8:03:11 PM	Standard	May 30, 2025, 8:03:11
Unknown_20250530_200312.jpg	50.7 KB	image/jpeg	May 30, 2025, 8:03:13 PM	Standard	May 30, 2025, 8:03:13
Unknown_20250530_200315.jpg	51.3 KB	image/jpeg	May 30, 2025, 8:03:15 PM	Standard	May 30, 2025, 8:03:15
Unknown_20250530_200317.jpg	50.5 KB	image/jpeg	May 30, 2025, 8:03:18 PM	Standard	May 30, 2025, 8:03:18
Unknown_20250530_200323.jpg	63.1 KB	image/jpeg	May 30, 2025, 8:03:23 PM	Standard	May 30, 2025, 8:03:23
Unknown_20250530_200325.jpg	51.8 KB	image/jpeg	May 30, 2025, 8:03:26 PM	Standard	May 30, 2025, 8:03:26
Unknown_20250530_200327.jpg	52.2 KB	image/jpeg	May 30, 2025, 8:03:28 PM	Standard	May 30, 2025, 8:03:28
Unknown_20250530_200329.jpg	54.4 KB	image/jpeg	May 30, 2025, 8:03:30 PM	Standard	May 30, 2025, 8:03:30
Unknown_20250530_200332.jpg	52.1 KB	image/jpeg	May 30, 2025, 8:03:33 PM	Standard	May 30, 2025, 8:03:33
Unknown_20250530_200348.jpg	51.2 KB	image/jpeg	May 30, 2025, 8:03:49 PM	Standard	May 30, 2025, 8:03:49
Unknown_20250530_200351.jpg	51.8 KB	image/jpeg	May 30, 2025, 8:03:51 PM	Standard	May 30, 2025, 8:03:51
Unknown_20250530_200410.jpg	87.6 KB	image/jpeg	May 30, 2025, 8:04:11 PM	Standard	May 30, 2025, 8:04:11
Unknown_20250531_074510.jpg	72.8 KB	image/jpeg	May 31, 2025, 7:45:10 AM	Standard	May 31, 2025, 7:45:10

# BigQuery Table Preview

Google Cloud | door-lagger-pl | Search (/) for resources, docs, products, and more

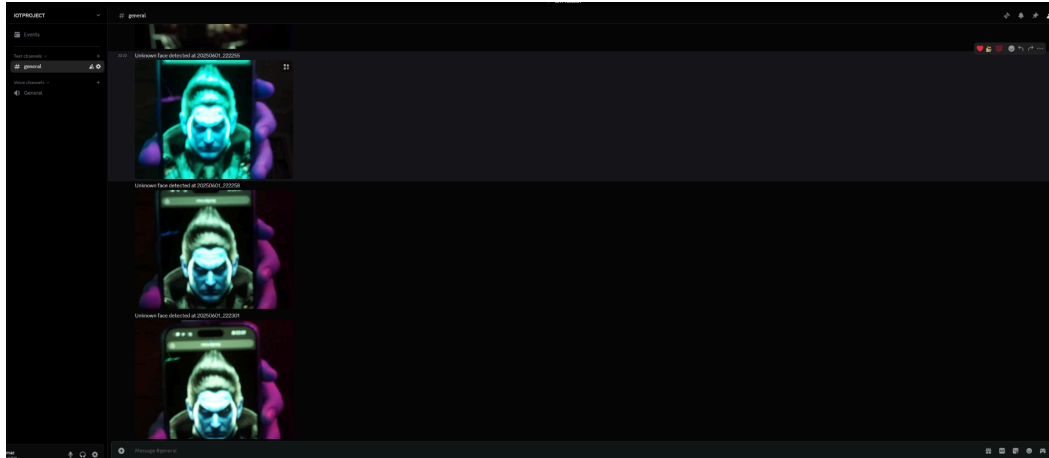
BigQuery | face\_events | Query | Open in | Share | Copy | Snapshot | Delete | Export

Schema | Details | Preview | Table Explorer | Insights | Lineage | Data Profile | Data Quality

Preview may not show all data being streamed to BigQuery. If you want to check for such data, run a SELECT statement over the table.

Row	image_name	person_name	timestamp
1	Unknown_20250530_044249.jpg	Unknown	2025-05-30 16:42:50.379965 UTC
2	Unknown_20250530_044252.jpg	Unknown	2025-05-30 16:42:52.062739 UTC
3	Unknown_20250530_044254.jpg	Unknown	2025-05-30 16:42:55.638929 UTC
4	Unknown_20250530_044257.jpg	Unknown	2025-05-30 16:42:56.133781 UTC
5	Unknown_20250530_195335.jpg	Unknown	2025-05-30 07:53:36.134351 UTC
6	Unknown_20250530_195347.jpg	Unknown	2025-05-30 07:53:48.655847 UTC
7	Unknown_20250530_195412.jpg	Unknown	2025-05-30 07:54:12.789118 UTC
8	Unknown_20250530_195416.jpg	Unknown	2025-05-30 07:54:17.294881 UTC
9	Unknown_20250530_195419.jpg	Unknown	2025-05-30 07:54:19.762483 UTC
10	Unknown_20250530_200142.jpg	Unknown	2025-05-30 08:01:43.622856 UTC
11	Unknown_20250530_200147.jpg	Unknown	2025-05-30 08:01:47.746696 UTC
12	Unknown_20250530_200227.jpg	Unknown	2025-05-30 08:02:28.033786 UTC
13	Unknown_20250530_200231.jpg	Unknown	2025-05-30 08:02:32.509395 UTC
14	Unknown_20250530_200238.jpg	Unknown	2025-05-30 08:02:38.870159 UTC
15	Unknown_20250530_200306.jpg	Unknown	2025-05-30 08:02:41.410518 UTC
16	Unknown_20250530_200308.jpg	Unknown	2025-05-30 08:03:06.769693 UTC
17	Unknown_20250530_200308.jpg	Unknown	2025-05-30 08:03:09.067819 UTC
18	Unknown_20250530_200310.jpg	Unknown	2025-05-30 08:03:11.320448 UTC
19	Unknown_20250530_200312.jpg	Unknown	2025-05-30 08:03:13.119443 UTC
20	Unknown_20250530_200315.jpg	Unknown	2025-05-30 08:03:16.551268 UTC
21	Unknown_20250530_200317.jpg	Unknown	2025-05-30 08:03:18.487019 UTC
22	Unknown_20250530_200323.jpg	Unknown	2025-05-30 08:03:23.653881 UTC
23	Unknown_20250530_200325.jpg	Unknown	2025-05-30 08:03:26.131882 UTC
24	Unknown_20250530_200327.jpg	Unknown	2025-05-30 08:03:28.606609 UTC
25	Unknown_20250530_200330.jpg	Unknown	2025-05-30 08:03:31.031378 UTC
26	Unknown_20250530_200332.jpg	Unknown	2025-05-30 08:03:33.615282 UTC
27	Unknown_20250530_200348.jpg	Unknown	2025-05-30 08:03:49.484708 UTC
28	Unknown_20250530_200351.jpg	Unknown	2025-05-30 08:03:51.862804 UTC
29	Unknown_20250530_200410.jpg	Unknown	2025-05-30 08:04:11.589225 UTC
30	Unknown_20250531_074510.jpg	Unknown	2025-05-30 19:45:10.847142 UTC
31	Unknown_20250530_175809.jpg	Unknown	2025-05-30 05:58:09.551728 UTC

# Discord Alert Message



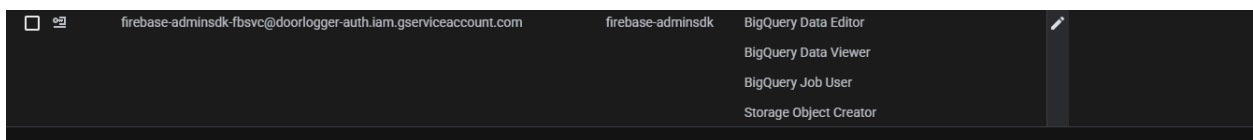
## .env file Configuration

```

amaz@raspberrypi: ~/doorlogger
File Edit Tabs Help
GNU nano 7.2 .env *
REDIRECT_URI=http://localhost:5000/login/google/authorized
GOOGLE_APPLICATION_CREDENTIALS=/home/amaz/doorlogger/doorlogger-auth-firebase-a>
BIGQUERY_PROJECT_ID=door-logger-pi
BIGQUERY_DATASET=doorlogger_logs
BIGQUERY_TABLE=face_events
BUCKET_NAME=doorlogger-images
DISCORD_WEBHOOK_URL=https://discord.com/api/webhooks/1377657524857208872/08HN-e>
ADMIN_USERNAME=amazsalman2@gmail.com
SECRET_KEY=supersecretkey
^G Help      ^O Write Out ^W Where Is  ^K Cut       ^T Execute   ^C Location
^X Exit      ^R Read File ^\ Replace   ^U Paste     ^J Justify   ^_ Go To Line

```

## IAM Policy Binding in GCP Console:



## Data Storage

Images are placed in Google Cloud Storage (GCS), while results of the detection are added to BigQuery in this prototype. Every time Raspberry Pi's screen captures a face, no matter if it is "Amaz" or not, it first creates a JPEG in `/home/amaz/doorlogger/captures`.

Immediately after, it places the image into a specific folder in the GCS called `doorlogger-images` and the file name contains "captures." By using Python, I work with GCS using GCS client under the line `storage.Client.from_service_account_json(os.getenv("GOOGLE_APPLICATION_CREDENTIALS"))`. Doorlogger transmits the uploaded data over HTTPS and checks its authenticity with the service account key file named `doorlogger-auth-firebase-adminsdk-fbsvc-0763fc5a09.json`. In other words, this service account can only perform file uploads (using Storage Object Creator) to that particular bucket.

Since we wanted to view the images from the dashboard and Discord, we provided read access (`allUsers:objectViewer`) only to the `captures/` folder. So, as soon as the image is uploaded, anyone can see it by using its public URL, for example:

***<https://storage.googleapis.com/doorlogger-images/captures/<filename>.jpg>***

The storage in GCS is extremely cheap—each month, it is about \$0.02 NZD per GB. And, since it easily scales, we can put hundreds or many thousands of JPEGs on our drive without having to arrange them independently.

To save storage space, we might use a lifecycle rule in the real system to remove older images after a specific period (for example, 30 days). Yet, for this prototype, images are being kept forever.

The information collected for each detection is put into BigQuery. The table door-logger-pi:doorlogger\_logs.face\_events has the following structure

***person\_name* STRING,**  
***image\_name* STRING,**  
***timestamp* TIMESTAMP**

BigQuery has set up the table so that data is organized by date in the timestamp field. A helper function named helper from the /home/amaz/doorlogger/bq\_logger.py file is called after the JPEG upload is complete. insert\_rows\_json(table\_id, [row]) from bq\_client is what this function uses to put one row into BigQuery. Every row shows the visitor's name (or "Unknown"), the image file name and the time when the event occurred (UTC).

The dates separate the table and each fact gets saved in the proper day of the week section. A query asking for the COUNT for a specific date, like "COUNT(\*) GROUP BY DATE(timestamp)," is quicker and less expensive, since BigQuery only looks through the needed data instead of reading everything in the whole table.

Only the BigQuery Data Editor role is assigned to the service account used. It only allows new rows to be inserted, not any kind of table change or deletion, so your data remains safe and managed.

People using BigQuery only need to pay about \$0.02 NZD per GB scanned and around \$0.000002 per row inserted. And because there is a large number of detections, the price for the service remains low. Using partitions helps cut the costs even more.

When transferring information, it is safe: both uploads to GCS and communication with BigQuery happen over HTTPS/TLS and gRPC over TLS. Least privilege is practiced, so permissions are limited as much as possible.

Ultimately, having GCS keep the files and BigQuery track the events is a very good combination. The system is cost-effective, can be used by many at once and ensures we have strong, speedy data access.

**Screenshots:**

## GCS Bucket Contents:

		<a href="#">Unknown_20250530_200410.jpg</a>	87.5 KB	image/jpeg	May 30, 2025, 8:04:11 PM	Standard	May 30, 2025, 8:04:11		
		<a href="#">Unknown_20250531_074510.jpg</a>	72.8 KB	image/jpeg	May 31, 2025, 7:45:10 AM	Standard	May 31, 2025, 7:45:10		
		<a href="#">Unknown_20250601_031207.jpg</a>	68.9 KB	image/jpeg	Jun 1, 2025, 3:12:08 AM	Standard	Jun 1, 2025, 3:12:08 A		
		<a href="#">Unknown_20250601_210039.jpg</a>	72.7 KB	image/jpeg	Jun 1, 2025, 9:00:40 PM	Standard	Jun 1, 2025, 9:00:40 P		
		<a href="#">Unknown_20250601_210135.jpg</a>	72.7 KB	image/jpeg	Jun 1, 2025, 9:01:35 PM	Standard	Jun 1, 2025, 9:01:35 P		
		<a href="#">Unknown_20250601_210137.jpg</a>	66.3 KB	image/jpeg	Jun 1, 2025, 9:01:38 PM	Standard	Jun 1, 2025, 9:01:38 P		
		<a href="#">Unknown_20250601_210140.jpg</a>	65.6 KB	image/jpeg	Jun 1, 2025, 9:01:40 PM	Standard	Jun 1, 2025, 9:01:40 P		
		<a href="#">Unknown_20250601_210202.jpg</a>	67 KB	image/jpeg	Jun 1, 2025, 9:02:03 PM	Standard	Jun 1, 2025, 9:02:03 P		
		<a href="#">Unknown_20250601_215605.jpg</a>	70.1 KB	image/jpeg	Jun 1, 2025, 9:56:06 PM	Standard	Jun 1, 2025, 9:56:06 P		
		<a href="#">Unknown_20250601_222231.jpg</a>	64.1 KB	image/jpeg	Jun 1, 2025, 10:22:32 PM	Standard	Jun 1, 2025, 10:22:32		
		<a href="#">Unknown_20250601_222255.jpg</a>	84.9 KB	image/jpeg	Jun 1, 2025, 10:22:56 PM	Standard	Jun 1, 2025, 10:22:56		
		<a href="#">Unknown_20250601_222258.jpg</a>	69.2 KB	image/jpeg	Jun 1, 2025, 10:22:58 PM	Standard	Jun 1, 2025, 10:22:58		
		<a href="#">Unknown_20250601_222301.jpg</a>	72.6 KB	image/jpeg	Jun 1, 2025, 10:23:01 PM	Standard	Jun 1, 2025, 10:23:01		
		<a href="#">Unknown_20250601_225950.jpg</a>	67.4 KB	image/jpeg	Jun 1, 2025, 10:59:51 PM	Standard	Jun 1, 2025, 10:59:51		
		<a href="#">Unknown_20250601_230156.jpg</a>	64 KB	image/jpeg	Jun 1, 2025, 11:01:57 PM	Standard	Jun 1, 2025, 11:01:57		
		<a href="#">Unknown_20250601_230200.jpg</a>	65.2 KB	image/jpeg	Jun 1, 2025, 11:02:01 PM	Standard	Jun 1, 2025, 11:02:01		
		<a href="#">amaz_20250601_210038.jpg</a>	72.7 KB	image/jpeg	Jun 1, 2025, 9:00:39 PM	Standard	Jun 1, 2025, 9:00:39 P		
		<a href="#">amaz_20250530_042028.jpg</a>	65.3 KB	image/jpeg	May 30, 2025, 4:20:30 AM	Standard	May 30, 2025, 4:20:30		
		<a href="#">amaz_20250530_042032.jpg</a>	77.8 KB	image/jpeg	May 30, 2025, 4:20:32 AM	Standard	May 30, 2025, 4:20:32		
		<a href="#">amaz_20250530_042034.jpg</a>	73.7 KB	image/jpeg	May 30, 2025, 4:20:34 AM	Standard	May 30, 2025, 4:20:34		
		<a href="#">amaz_20250530_042035.jpg</a>	73.2 KB	image/jpeg	May 30, 2025, 4:20:36 AM	Standard	May 30, 2025, 4:20:36		
		<a href="#">amaz_20250530_042037.jpg</a>	71.3 KB	image/jpeg	May 30, 2025, 4:20:37 AM	Standard	May 30, 2025, 4:20:37		
		<a href="#">amaz_20250530_042954.jpg</a>	69.9 KB	image/jpeg	May 30, 2025, 4:29:55 AM	Standard	May 30, 2025, 4:29:55		
		<a href="#">amaz_20250530_043621.jpg</a>	67.8 KB	image/jpeg	May 30, 2025, 4:36:22 AM	Standard	May 30, 2025, 4:36:22		
		<a href="#">amaz_20250530_043624.jpg</a>	73.4 KB	image/jpeg	May 30, 2025, 4:36:25 AM	Standard	May 30, 2025, 4:36:25		
		<a href="#">amaz_20250530_043626.jpg</a>	74.3 KB	image/jpeg	May 30, 2025, 4:36:26 AM	Standard	May 30, 2025, 4:36:26		
		<a href="#">amaz_20250530_043628.jpg</a>	78.2 KB	image/jpeg	May 30, 2025, 4:36:28 AM	Standard	May 30, 2025, 4:36:28		
		<a href="#">amaz_20250530_043631.jpg</a>	80.1 KB	image/jpeg	May 30, 2025, 4:36:32 AM	Standard	May 30, 2025, 4:36:32		
		<a href="#">amaz_20250530_044209.jpg</a>	74.8 KB	image/jpeg	May 30, 2025, 4:42:11 AM	Standard	May 30, 2025, 4:42:11		
		<a href="#">amaz_20250530_044217.jpg</a>	73.5 KB	image/jpeg	May 30, 2025, 4:42:17 AM	Standard	May 30, 2025, 4:42:17		
		<a href="#">amaz_20250530_044218.jpg</a>	79.3 KB	image/jpeg	May 30, 2025, 4:42:19 AM	Standard	May 30, 2025, 4:42:19		
		<a href="#">amaz_20250530_044243.jpg</a>	77.2 KB	image/jpeg	May 30, 2025, 4:42:44 AM	Standard	May 30, 2025, 4:42:44		
		<a href="#">amaz_20250530_175440.jpg</a>	72.4 KB	image/jpeg	May 30, 2025, 5:54:41 PM	Standard	May 30, 2025, 5:54:41		
		<a href="#">amaz_20250530_175755.jpg</a>	72.6 KB	image/jpeg	May 30, 2025, 5:57:56 PM	Standard	May 30, 2025, 5:57:56		
		<a href="#">amaz_20250530_175758.jpg</a>	82.8 KB	image/jpeg	May 30, 2025, 5:57:59 PM	Standard	May 30, 2025, 5:57:59		
		<a href="#">amaz_20250530_175800.jpg</a>	83.1 KB	image/jpeg	May 30, 2025, 5:58:01 PM	Standard	May 30, 2025, 5:58:01		
		<a href="#">amaz_20250530_175812.jpg</a>	72.8 KB	image/jpeg	May 30, 2025, 5:58:13 PM	Standard	May 30, 2025, 5:58:13		
		<a href="#">amaz_20250530_175814.jpg</a>	72.7 KB	image/jpeg	May 30, 2025, 5:58:14 PM	Standard	May 30, 2025, 5:58:14		

## BigQuery Table Schema:

face\_events

Filter Enter property name or value

Field name	Type	Mode	Key	Collation	Default Value	Policy Tags	Description
image_name	STRING	NULLABLE	-	-	-	-	-
person_name	STRING	NULLABLE	-	-	-	-	-
timestamp	TIMESTAMP	NULLABLE	-	-	-	-	-

Edit schema View row access policies

# BigQuery Table Preview:

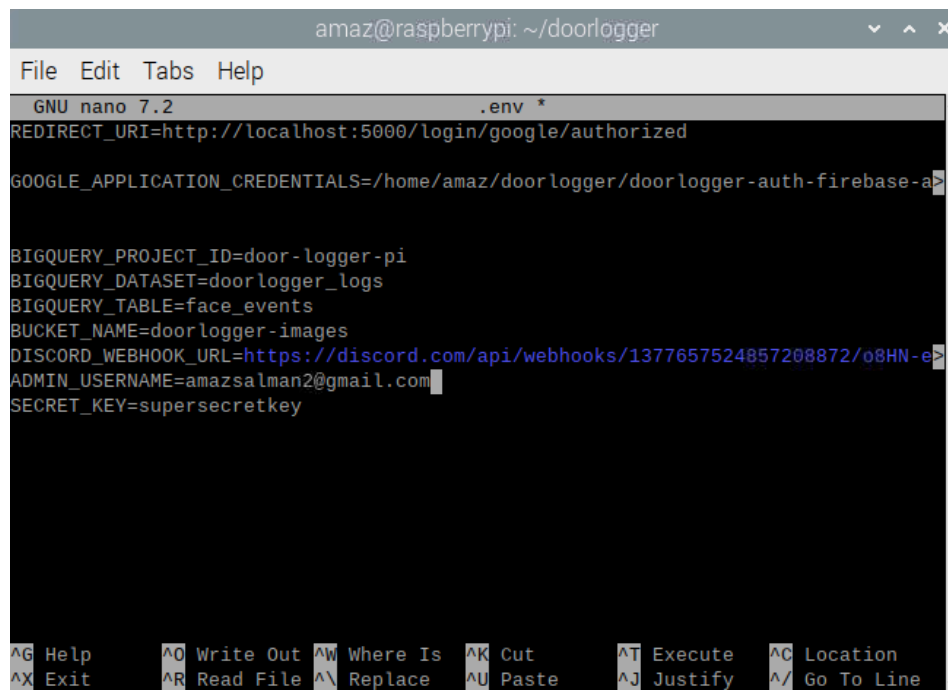
Google Cloud door-lagger-pi

face\_events

Schema Details Preview Table Explorer Insights Lineage Data Profile Data Quality

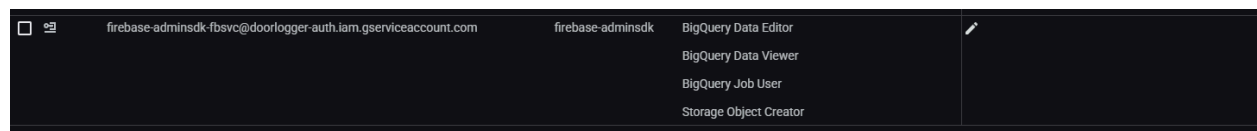
Row	image_name	person_name	timestamp
1	Unknown_20250530_044249.jpg	Unknown	2025-05-29 16:42:50.377965 UTC
2	Unknown_20250530_044252.jpg	Unknown	2025-05-29 16:42:53.062739 UTC
3	Unknown_20250530_044254.jpg	Unknown	2025-05-29 16:42:55.638929 UTC
4	Unknown_20250530_044257.jpg	Unknown	2025-05-29 16:42:58.113761 UTC
5	Unknown_20250530_193258.jpg	Unknown	2025-05-30 07:53:36.162451 UTC
6	Unknown_20250530_193547.jpg	Unknown	2025-05-30 07:53:48.850349 UTC
7	Unknown_20250530_195417.jpg	Unknown	2025-05-30 07:54:12.789118 UTC
8	Unknown_20250530_195418.jpg	Unknown	2025-05-30 07:54:17.294801 UTC
9	Unknown_20250530_195419.jpg	Unknown	2025-05-30 07:54:19.763483 UTC
10	Unknown_20250530_200142.jpg	Unknown	2025-05-30 08:01:43.622856 UTC
11	Unknown_20250530_200147.jpg	Unknown	2025-05-30 08:01:47.740690 UTC
12	Unknown_20250530_200227.jpg	Unknown	2025-05-30 08:02:28.033786 UTC
13	Unknown_20250530_200231.jpg	Unknown	2025-05-30 08:02:32.509395 UTC
14	Unknown_20250530_200258.jpg	Unknown	2025-05-30 08:02:38.970759 UTC
15	Unknown_20250530_200348.jpg	Unknown	2025-05-30 08:03:41.410318 UTC
16	Unknown_20250530_200308.jpg	Unknown	2025-05-30 08:03:06.793051 UTC
17	Unknown_20250530_200308.jpg	Unknown	2025-05-30 08:03:09.057819 UTC
18	Unknown_20250530_200310.jpg	Unknown	2025-05-30 08:03:11.323446 UTC
19	Unknown_20250530_200312.jpg	Unknown	2025-05-30 08:03:13.619643 UTC
20	Unknown_20250530_200315.jpg	Unknown	2025-05-30 08:03:16.051206 UTC
21	Unknown_20250530_200317.jpg	Unknown	2025-05-30 08:03:18.487019 UTC
22	Unknown_20250530_200323.jpg	Unknown	2025-05-30 08:03:23.623881 UTC
23	Unknown_20250530_200325.jpg	Unknown	2025-05-30 08:03:26.101982 UTC
24	Unknown_20250530_200327.jpg	Unknown	2025-05-30 08:03:28.600091 UTC
25	Unknown_20250530_200330.jpg	Unknown	2025-05-30 08:03:31.091378 UTC
26	Unknown_20250530_200332.jpg	Unknown	2025-05-30 08:03:33.615282 UTC
27	Unknown_20250530_200348.jpg	Unknown	2025-05-30 08:03:49.484706 UTC
28	Unknown_20250530_200351.jpg	Unknown	2025-05-30 08:03:51.862804 UTC
29	Unknown_20250530_200410.jpg	Unknown	2025-05-30 08:04:11.559225 UTC
30	Unknown_20250531_074310.jpg	Unknown	2025-05-30 19:45:10.847142 UTC
31	Unknown_20250530_074958.jpg	Unknown	2025-05-30 09:49:59.927728 UTC
32	Unknown_20250601_091207.jpg	Unknown	2025-05-31 11:19:08.203741 UTC
33	amaz_20250530_043621.jpg	amaz	2025-05-29 16:36:22.638604 UTC
34	amaz_20250530_043624.jpg	amaz	2025-05-29 16:36:25.419413 UTC
35	amaz_20250530_043626.jpg	amaz	2025-05-29 16:36:27.107039 UTC
36	amaz_20250530_043628.jpg	amaz	2025-05-29 16:36:28.833311 UTC

.env file Configuration:



```
amaz@raspberrypi: ~/doorlogger
File Edit Tabs Help
GNU nano 7.2 .env *
REDIRECT_URI=http://localhost:5000/login/google/authorized
GOOGLE_APPLICATION_CREDENTIALS=/home/amaz/doorlogger/doorlogger-auth-firebase-a>
BIGQUERY_PROJECT_ID=door-logger-pi
BIGQUERY_DATASET=doorlogger_logs
BIGQUERY_TABLE=face_events
BUCKET_NAME=doorlogger-images
DISCORD_WEBHOOK_URL=https://discord.com/api/webhooks/1377657524857208872/08HN-e>
ADMIN_USERNAME=amazsalman2@gmail.com
SECRET_KEY=supersecretkey
^G Help ^O Write Out ^W Where Is ^K Cut ^T Execute ^C Location
^X Exit ^R Read File ^\ Replace ^U Paste ^J Justify ^_ Go To Line
```

IAM Policy Bindings:



firebase-adminsdk-fbsvc@doorlogger-auth.iam.gserviceaccount.com	firebase-adminsdk	BigQuery Data Editor
		BigQuery Data Viewer
		BigQuery Job User
		Storage Object Creator

## Data Processing

The prototype processes data after the Raspberry Pi uploads an image and logs the event. When the `face_capture.py` script saves a face snapshot to Google Cloud Storage, it calls a helper function. This function, `log_face_entry(...)`, comes from the `bq_logger.py` file.

The helper function gathers a simple JSON-style row - this row holds three pieces of information - it contains the UTC timestamp of the detection, the person's name along with the filename of the image that was just saved. The name is "Amaz" if the system recognizes the face, but it reads "Unknown" otherwise.



After the function builds the row, it sends it to the BigQuery table. The table is `door-logger-pi:doorlogger_logs.face_events`, and the function uses BigQuery's streaming insert method.

The table has a schema that expects these fields, so the data fits. It is ready for querying and analysis. No extra work or formatting is necessary.

```
def log_face_entry(filename, name):  
    row = {  
        "image_name": filename,  
        "person_name": name,  
        "timestamp": datetime.utcnow().isoformat()  
    }
```

The BigQuery table separates data by date - it uses a timestamp field, so each new row finds its daily section by itself. This arrangement keeps queries quick. As the Raspberry Pi moves data into the table, it does more than just log. It checks the data's form, sets timestamps to UTC, and allows queries to access the data right away.

When an insert fails, perhaps a partition misses or permissions are wrong, a helper function writes an error message. The message starts with [BQ ERROR] and appears in the console or in `capture.log` - this helps someone see and solve problems quickly, before much data accumulates.

Once data enters BigQuery, all other processing happens when requested through the Flask web backend. For instance, when a user opens the dashboard, the `/api/data` endpoint starts. That path runs a SQL query against BigQuery to get the newest face detection events.

The `/api/data` path brings back only entries from the last hour; this provides users a live view of recent actions. The Flask app puts the query together as needed and runs it immediately. This keeps the system current and responds well without needing to hold more data on the local machine.

```

@app.route("/api/daily-data")
def api_daily_data():
    if "user" not in session:
        return redirect("/login")

    query = f"""
        SELECT
            DATE(timestamp) AS date,
            COUNT(*) AS count
        FROM `{table_id}`
        GROUP BY date
        ORDER BY date ASC
    """

    print("DEBUG: Running daily-data BigQuery query:", query.strip().split("\n")[0] + " ...")
    query_job = bq_client.query(query)
    daily = [{"date": str(row["date"]), "count": row["count"]} for row in query_job]
    return jsonify(daily)

```

A Flask route first checks if a user shows proof of identity. The system looks for "user" within the session, and if it is not present, the user goes to the /login page. This check means only users who log in can enter the dashboard and view detection data.

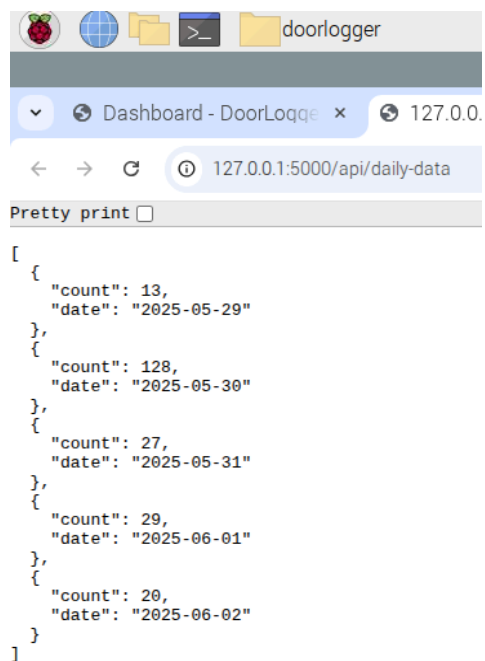
After a user proves identity, the route creates a SQL query to get daily statistics from BigQuery. The query changes each detection event's timestamp into a date, groups all events by that date, and then counts the events that occur on each day - it also places the results in order, so older dates come first.

The BigQuery table divides by DATE(timestamp), so the query runs well. BigQuery skips partitions that do not contain data - this lowers the query time and the cost.

For debugging, the Flask app writes the start of the SQL query to the console. In the terminal, one sees a message like  
 DEBUG - Running daily data BigQuery query - SELECT DATE(timestamp) AS date, ...

When BigQuery sends back the results, the code uses a list comprehension to change each row into a dictionary. Every dictionary contains the date as text, for instance, "2025-06-02" along with the number of face detection events on that date as a whole number.

At last, the Flask route uses jsonify(daily) to send the whole list of daily event counts back to the frontend as a JSON array. The dashboard can then use this data to show a bar chart or a summary.



The screenshot shows a web browser window with the title 'Dashboard - DoorLogge' and the address bar displaying '127.0.0.1:5000/api/daily-data'. The page content shows a JSON array of objects, each containing a 'count' and a 'date'. The data is as follows:

```
[
  {
    "count": 13,
    "date": "2025-05-29"
  },
  {
    "count": 128,
    "date": "2025-05-30"
  },
  {
    "count": 27,
    "date": "2025-05-31"
  },
  {
    "count": 29,
    "date": "2025-06-01"
  },
  {
    "count": 20,
    "date": "2025-06-02"
  }
]
```

The data sent back via the Flask route is then used to populate a bar or line chart on the dashboard, producing a visual representation of how many individuals were detected on each day. This makes it easier to see patterns, or anomalies, in visitor behaviour.

Much of the heavy lifting, filtering, grouping by date, sorting is done directly in BigQuery, which is optimized for efficient large-scale data processing. Flask's job is simple: it only constructs the SQL query, ensures the user is logged in, transforms the results into JSON and sends them to the frontend.

On the frontend, JavaScript handles the last part. It will parse the JSON response, map timestamp strings into local time, and then pass the data to Chart.js to update the

graph. This arrangement maintains the entire system light, responsive, and cheap by employing fitting tools for each step.

## Screenshot:

Flask route code for /api/daily-data

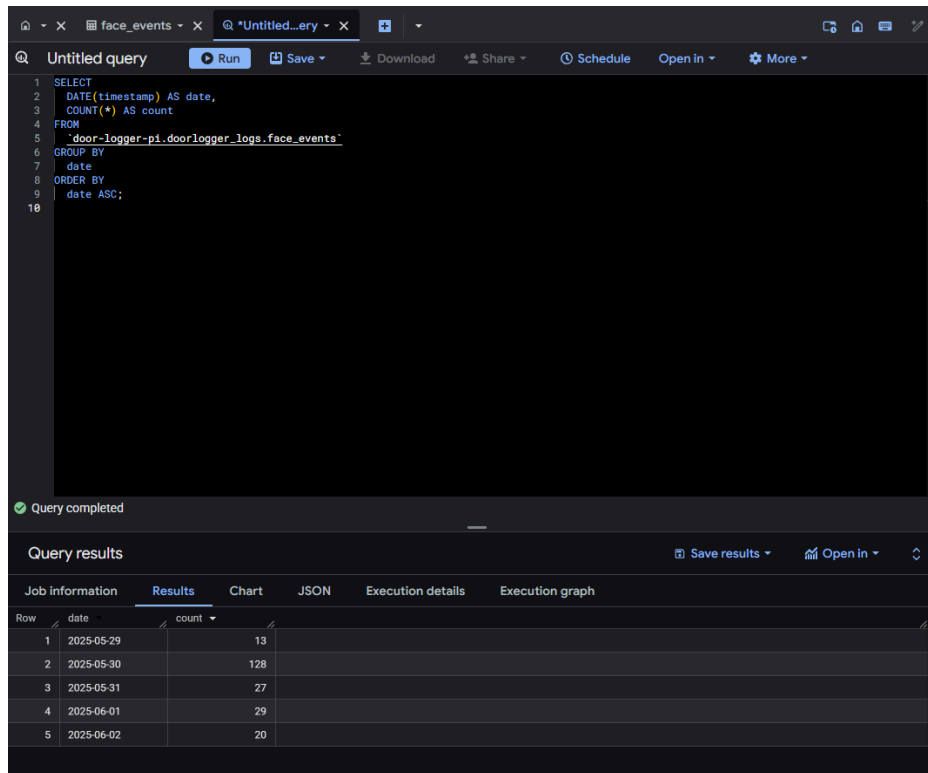
```
@app.route("/api/daily-data")
def api_daily_data():
    if "user" not in session:
        return redirect("/login")

    query = f"""
        SELECT
            DATE(timestamp) AS date,
            COUNT(*) AS count
        FROM `{table_id}`
        GROUP BY date
        ORDER BY date ASC
    """
    print("DEBUG: Running daily-data BigQuery query:", query.strip().split("\n")[0] + " ...")
    query_job = bq_client.query(query)
    daily = [{"date": str(row["date"]), "count": row["count"]} for row in query_job]
    return jsonify(daily)
```

Flask console showing Debug prints:

```
DEBUG: Received ID token: eyJhbGciOi...
DEBUG: Decoded token 'aud': doorlogger-auth
127.0.0.1 - - [03/Jun/2025 00:34:49] "POST /sessionLogin HTTP/1.1" 200 -
DEBUG: Running BigQuery query: SELECT person_name, image_name, timestamp ...
127.0.0.1 - - [03/Jun/2025 00:34:53] "GET /dashboard HTTP/1.1" 200 -
DEBUG: Running daily-data BigQuery query: SELECT ...
127.0.0.1 - - [03/Jun/2025 00:34:55] "GET /api/daily-data HTTP/1.1" 200 -
127.0.0.1 - - [03/Jun/2025 00:35:03] "GET /.well-known/appspecific/com.chrome.de
```

BigQuery Query Editor with Daily-Counts SQL



## Chart.js Rendering of Daily Counts on Dashboard

Pretty print ☐

```
[
  {
    "count": 13,
    "date": "2025-05-29"
  },
  {
    "count": 128,
    "date": "2025-05-30"
  },
  {
    "count": 27,
    "date": "2025-05-31"
  },
  {
    "count": 29,
    "date": "2025-06-01"
  },
  {
    "count": 20,
    "date": "2025-06-02"
  }
]
```

## Security & Access Control

For security and access control, this system uses three key points: guarding the secrets in service-accounts, using less privileged roles on IAM and defending the endpoints of the Flask application.

Service-account credentials are never added to source code. The JSON file keeps track of the keys needed for signing in.

***/home/amaz/doorlogger/doorlogger-auth-firebase-adminsdk-fbsvc-0763fc5a09.json***

Its is only referenced via the environment variable

***GOOGLE\_APPLICATION\_CREDENTIALS="/home/amaz/doorlogger/doorlogger-auth-firebase-adminsdk-fbsvc-0763fc5a09.json"***

The location of the service account key path is /home/amaz/doorlogger/.env inside a .env file. When running, the app uses load\_dotenv() to bring in the environment variable found in .env. Both app.py and bq\_logger.py can get the key file path securely, using the environment, instead of including it in the function. Because the .env file is omitted from tracking by .gitignore, the key file is left out of all public repositories.

We reduce the amount of permissions the service account has, keeping them to only the key ones needed. For Google Cloud Storage, go to the doorlogger-images bucket and add the role of Storage Object Creator to the service account doorlogger-auth-firebase-adminsdk-fbsvc@doorlogger-auth.iam.gserviceaccount.com. So, the Pi can add JPEG pictures but needs permission to remove or replace already stored files.

The BigQuery Data Editor role is given to the same service account in BigQuery, but only on the dataset doorlogger\_logs. With this role, new rows can enter through insert\_rows\_json(...), but the schema cannot be edited and nothing can be deleted. The approach is based on limiting access, so that possession of the key will not allow anyone to seriously affect the data.

There is one extra defense provided by the Flask web app. The login control is done with sessions and also makes use of Firebase Authentication. In app.py, each protected route starts by looking at whether the user is already logged in. If the "user" is not present in the session, the app sends the user to the login page directly. For this reason, administrators who are verified users can manage sensitive things like watching the log files or manually running the Pi's detection script.

```
@app.route("/api/daily-data")
def api_daily_data():
    if "user" not in session:
        return redirect("/login")
```

Only after verifying a firebase ID token:

```
@app.route("/sessionLogin", methods=["POST"])
def session_login():
    id_token = request.json.get("idToken")
    print("DEBUG: Received ID token:", id_token[:10] + "...")
    decoded_token = auth.verify_id_token(id_token)
    print("DEBUG: Decoded token 'aud':", decoded_token.get("aud"))
    session["user"] = {"email": decoded_token["email"], "uid": decoded_token["uid"]}
    return jsonify({"status": "success"})
```

For /sessionLogin, once the login is successful, the Flask app sets session["user"] = {"email": user\_email}. This allows the user to be authenticated so they can view and use files and folders restricted by role such as /dashboard, /api/data and /api/daily-data. Only people verified by Firebase are given access to the site.

The .env file also contains a secret key (SECRET\_KEY) that the Flask app makes use of. It helps sign session cookies, so if someone tries changing them, the system will catch and block the threat.

All the data sent between different parts of the system is encrypted.

The Raspberry Pi uploads images to Google Cloud Storage via blob.upload\_from\_filename() method using HTTPS and gRPC over TLS and streams data into BigQuery using insert\_rows\_json(...) method too. Thanks to SSL, data sent during a session is properly encrypted.

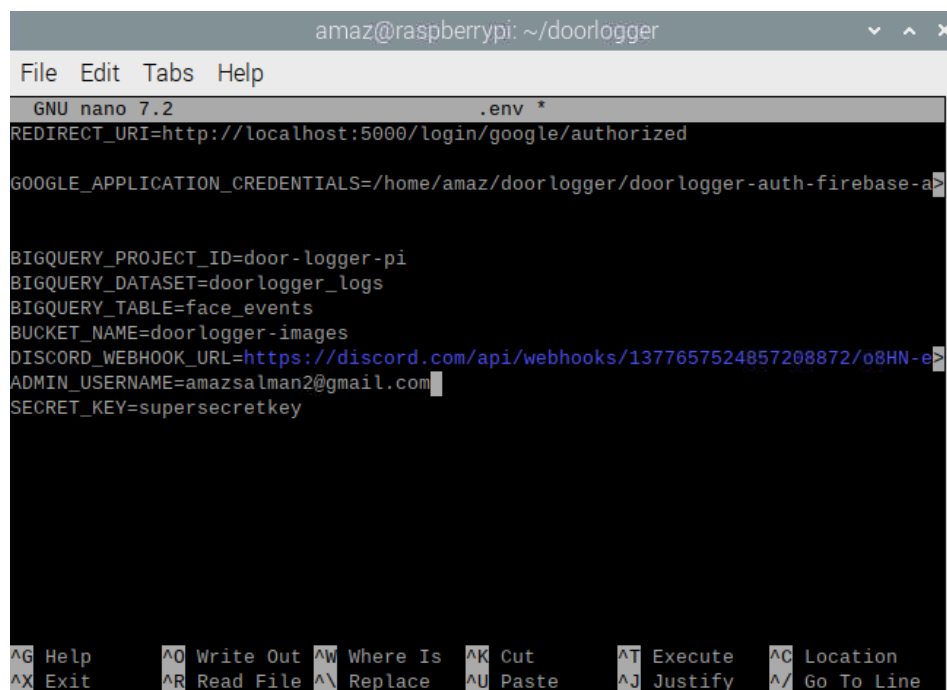
When a new face appears, Discord is notified right away by the system. It happens over HTTPS by relying on these methods:

***DISCORD\_WEBHOOK\_URL=https://discord.com/api/webhooks/1377657524857208872/o8HN-e>***

A webhook is used by sending a POST request such as `requests.post(DISCORD_WEBHOOK_URL, ...)`, to ensure the alert message and link to the image are delivered safely.

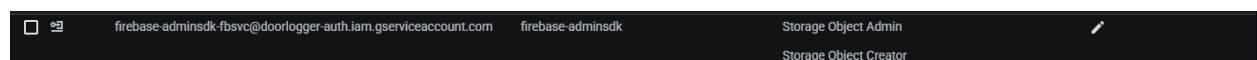
### Screenshots:

.env file with GOOGLE\_APPLICATION\_CREDENTIALS



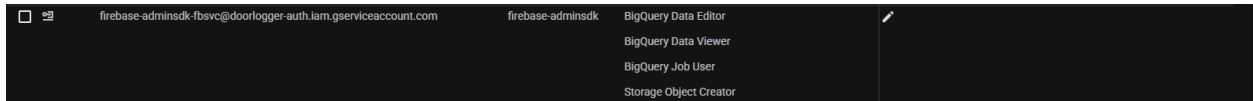
```
amaz@raspberrypi: ~/doorlogger
File Edit Tabs Help
GNU nano 7.2 .env *
REDIRECT_URI=http://localhost:5000/login/google/authorized
GOOGLE_APPLICATION_CREDENTIALS=/home/amaz/doorlogger/doorlogger-auth-firebase-a>
BIGQUERY_PROJECT_ID=door-logger-pi
BIGQUERY_DATASET=doorlogger_logs
BIGQUERY_TABLE=face_events
BUCKET_NAME=doorlogger-images
DISCORD_WEBHOOK_URL=https://discord.com/api/webhooks/1377657524857208872/o8HN-e>
ADMIN_USERNAME=amazsalman2@gmail.com
SECRET_KEY=supersecretkey
^G Help      ^O Write Out ^W Where Is  ^K Cut       ^T Execute   ^C Location
^X Exit      ^R Read File ^\ Replace   ^U Paste     ^J Justify   ^_ Go To Line
```

### GSC Bucket Permissions:



### BigQuery IAM Permissions:





Flask Route Code Showing Session Check:

```
@app.route("/dashboard")
def dashboard():
    if "user" not in session:
        return redirect("/login")

    query = f"""
        SELECT person_name, image_name, timestamp
        FROM `{table_id}`
        ORDER BY timestamp DESC
        LIMIT 10
    """

    return render_template("graph.html")

130
131
132
133 @app.route("/api/data")
134 def api_data():
135     if "user" not in session:
136         return redirect("/login")
137
```

Flask /sessionlogin Endpoint Verifying Firebase Token

```
@app.route("/sessionLogin", methods=["POST"])
def session_login():
    id_token = request.json.get("idToken")
    print("DEBUG: Received ID token:", id_token[:10] + "...")
    decoded_token = auth.verify_id_token(id_token)
    print("DEBUG: Decoded token 'aud':", decoded_token.get("aud"))
    session["user"] = {"email": decoded_token["email"], "uid": decoded_token["uid"]}
    return jsonify({"status": "success"})
```

Flask Console Showing Invalid Token Error

```

AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
File "/home/amaz/doorlogger/venv/lib/python3.11/site-packages/firebase_admin/_
token_gen.py", line 418, in _decode_unverified
    raise self._invalid_token_error(str(error), cause=error)
firebase_admin._auth_utils.InvalidIdTokenError: Wrong number of segments in token: b'this_is_an_invalid_token'

```

## User Interface

### Real-Time Data Visualization

A real-time video chart is included in the dashboard which displays every face detection event over the last hour. It does this by contacting a Flask API every five seconds to maintain its live update. Chart.js is used by the frontend to show the data as a line graph that updates itself regularly.

Handling this is the role of the `/api/data` endpoint in the Flask backend (`app.py`). When needed, it builds a query for BigQuery that gets all rows from the `face_events` table where the timestamp is from one hour in the past up to the current time.

This lets the dashboard respond quickly, while all the data filtering takes place in BigQuery which keeps the Pi and Flask server going fast and smoothly.

```

131
132
133 @app.route("/api/data")
134 def api_data():
135     if "user" not in session:
136         return redirect("/login")
137
138     now = datetime.datetime.utcnow()
139     past = now - datetime.timedelta(hours=1)
140     query = f"""
141         SELECT person_name, timestamp
142         FROM `{table_id}`
143         WHERE timestamp BETWEEN TIMESTAMP_SUB(CURRENT_TIMESTAMP(), INTERVAL 1 HOUR) AND CURRENT_TIMESTAMP()
144         ORDER BY timestamp ASC
145     """
146     print("DEBUG: Running API data BigQuery query:", query.strip().split("\n")[0] + " ...")
147     query_job = bq_client.query(query)
148     data = [{"time": str(row["timestamp"]), "name": row["person_name"]} for row in query_job]
149     return jsonify(data)
150

```

The `graph.html` file on the client side has a `<canvas>` element with the ID `logChart`. Here, the real-time line chart will show up on the screen. The page loads and JavaScript uses Chart.js to create a blank line chart using empty labels and data.

A request to `/api/data` in the Flask backend happens on the backend every five seconds. Once the data arrives, JavaScript separates the detections by their local time strings (like “14:00”, “14:05”, etc.) and counts how many detections took place in each time slot, after which it updates the chart with the updated data.

In having the most recent activity shown automatically, users feel like the chart is always fresh, without needing to refresh it.

```

<h2>Detections per Day</h2>
<div class="chart-container">
  <canvas id="dailyChart"></canvas>
</div>

<script>
  const ctx = document.getElementById('dailyChart').getContext('2d');
  const dailyChart = new Chart(ctx, {
    type: 'bar',
    data: {
      labels: [],
      datasets: [{
        label: 'Number of Detections',
        data: [],
        borderColor: 'blue',
        backgroundColor: 'rgba(0, 123, 255, 0.5)',
        borderWidth: 1
      }]
    },
    options: {
      scales: {
        x: {
          title: { display: true, text: 'Date' }
        },
        y: {
          title: { display: true, text: 'Count' },
          beginAtZero: true,
          precision: 0
        }
      }
    }
  });

  async function updateDailyChart() {
    const response = await fetch("/api/daily-data");
    const json = await response.json();

    const dates = json.map(item => item.date);
    const counts = json.map(item => item.count);

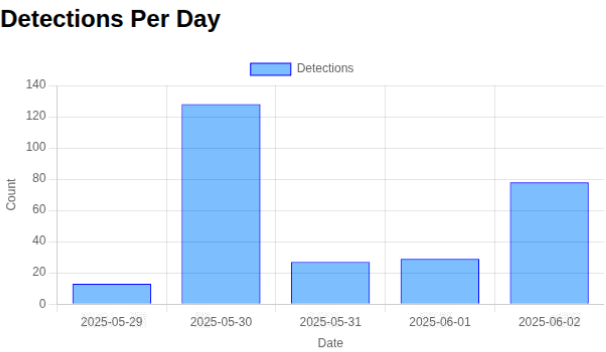
    dailyChart.data.labels = dates;
    dailyChart.data.datasets[0].data = counts;
    dailyChart.update();
  }

  updateDailyChart();
</script>

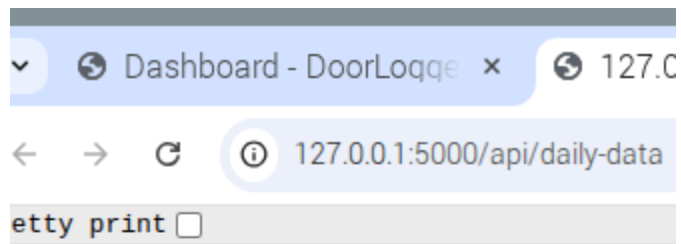
```

**Screenshot:**

Dashboard page:



Browser Network Tab showing .JSON response.



```
{
  "count": 13,
  "date": "2025-05-29"
},
{
  "count": 128,
  "date": "2025-05-30"
},
{
  "count": 27,
  "date": "2025-05-31"
},
{
  "count": 29,
  "date": "2025-06-01"
},
{
  "count": 78,
  "date": "2025-06-02"
}
```

Terminal running app.py showing the debug print of the BigQuery query.

```
vttools.json HTTP/1.1" 404 -
DEBUG: Running daily-data BigQuery query: SELECT ...
127.0.0.1 - - [03/Jun/2025 04:32:15] "GET /api/daily-data HTTP/1.1" 200 -
DEBUG: Running daily-data BigQuery query: SELECT ...
127.0.0.1 - - [03/Jun/2025 04:33:19] "GET /api/daily-data HTTP/1.1" 200 -
DEBUG: Running BigQuery query: SELECT person_name, image_name, timestamp ...
127.0.0.1 - - [03/Jun/2025 04:35:02] "GET /dashboard HTTP/1.1" 200 -
DEBUG: Running daily-data BigQuery query: SELECT ...
127.0.0.1 - - [03/Jun/2025 04:35:03] "GET /.well-known/appspecific/com.chrome.de
vttools.json HTTP/1.1" 404 -
127.0.0.1 - - [03/Jun/2025 04:35:05] "GET /api/daily-data HTTP/1.1" 200 -
DEBUG: Running daily-data BigQuery query: SELECT ...
127.0.0.1 - - [03/Jun/2025 04:38:40] "GET /api/daily-data HTTP/1.1" 200 -
```

## Interactive Controls

There are interactive buttons on the dashboard so that administrators can control the face detection feature from the web. You have to manually start and stop the Raspberry Pi's camera by clicking on the buttons labeled "Start Capture" and "Stop Capture", rather than having it run all the time. All buttons are within their own HTML `<form>` tags so that when pressed, they make a POST request to the Flask server.

As soon as a user clicks "Start Capture", the browser calls the `/start_capture` endpoint in Flask. That route first ensures the face detection process is not currently active. Otherwise, Flask makes sure to start the `face_capture.py` script on the Pi concurrently using Python's `subprocess.Popen`. After the script is executed, Flask takes the user to the `/dashboard` page.

Whenever the capture script starts, any new faces detected by the dashboard are added right away to the log table or appear on the graph, since BigQuery is always refreshed.

The job of the "Stop Capture" button is also very similar. The method creates a POST request to the `/stop_capture` path. It looks to see if the face detection script is currently running. When that's true, terminal will run the call `capture_process.terminate()` to shut down the program in a clean manner. When you stop the application, Flask takes you back to the main dashboard.

Even after reloading, the logs showing the most recent faces will still be shown and users must click "Start Capture" again for new images to display.

## Screenshots:

Dashboard with Buttons Visible:

### Welcome to DoorLogger

User: amazsalman2@gmail.com | [Logout](#)

Start Capture

Stop Capture

Network Tab Showing POST to /start\_capture:

start_capture	▼ General
dashboard	Request URL
chart.js	Request Method
amaz_20250603_042308.jpg	Status Code
amaz_20250603_042303.jpg	Remote Address
amaz_20250603_042301.jpg	Referrer Policy
amaz_20250603_042259.jpg	▼ Response Headers
amaz_20250603_042256.jpg	Connection
amaz_20250603_042253.jpg	Content-Length
amaz_20250603_042251.jpg	Content-Type
amaz_20250603_042249.jpg	Date
amaz_20250603_042247.jpg	Location
amaz_20250603_042243.jpg	Server
daily-data	▼ Request Headers
	Accept
	Accept-Encoding
	Accept-Language
	Cache-Control
	Connection
	Content-Length
	Content-Type
	Cookie
	Host
	Origin
	Referer
14 requests   8.7 kB transferred	

## Flask Console Showing Process Launch

```
127.0.0.1 - - [03/Jun/2025 04:52:02] "GET /api/daily-data HTTP/1.1" 200 -
127.0.0.1 - - [03/Jun/2025 04:52:12] "GET /start_capture HTTP/1.1" 405 -
```

Contents of capture.log Updating:



```

[DEBUG] Detected 0 Face(s)
[DEBUG] Detected 0 Face(s)
[DEBUG] Detected 0 Face(s)
[DEBUG] Detected 0 Face(s)
[DEBUG] Detected 0 Face(s)
[DEBUG] Detected 0 Face(s)
[DEBUG] Detected 0 Face(s)
[DEBUG] Detected 0 Face(s)
/home/alex/doorlogger/venv/lib/python3.11/site-packages/face_recognition/models/_load.py:7: UserWarning: pkg_resources is deprecated as an API. See https://setuptools.pypa.io/en/latest/pkg_resources.html. The pkg_resources package is slated for removal as early as 2025-11-30. Refrain from using this package or pin to setuptools<68.
  from pkg_resources import resource_filename

[52:04:42.056899325] [INFO] [[1:32w 2M0D [[1:32wCamera [[1:32wCamera_manager.cpp:328 [[@libcamera v6.5.0-0-037f664
[52:04:42.072987407] [INFO] [[1:32w WMN [[1:32wCameraSensorProperties [[1:32wCamera_sensor_properties.cpp:473 [[@libcamera static properties available for 'imx708_msr'
[52:04:42.073067322] [INFO] [[1:32w WMN [[1:32wCameraSensorProperties [[1:32wCamera_sensor_properties.cpp:475 [[@libcamera consider selecting the camera sensor properties database
[52:04:42.085756365] [INFO] [[1:32w WMN [[1:32wMipiSh [[1:32wMipiSh.cpp:48 [[@libcamera legacy SDR tuning - please consider moving SDR inside rpi device
[52:04:42.087847382] [INFO] [[1:32w WMN [[1:32wCameraSensor [[1:32wCamera_sensor_legacy.cpp:582 [[@libcamera delays found in static properties. Assuming unverified defaults.
[52:04:42.087847382] [INFO] [[1:32w 2M0D [[1:32wMipiSh [[1:32wMipiSh.cpp:487 [[@libcamera camera /home/alex/220mcu/220f/lan7808a to Union device /dev/media0 and ISP device /dev/media1
[52:04:42.087908792] [INFO] [[1:32w 2M0D [[1:32wMipiSh [[1:32wMipiSh.cpp:1323 [[@libcamera configuration file '/usr/share/libcamera/pipeline/rpi/v3d/rpi_apps.yaml'
[52:04:42.087908792] [INFO] [[1:32w 2M0D [[1:32wCamera [[1:32wCamera.cpp:1280 [[@libcamera (New/fingerprint stream): (0) Sublime-MANAGE (1) Sublime-UNMANAGE (2) STOP
[52:04:42.103364742] [INFO] [[1:32w 2M0D [[1:32wMipiSh [[1:32wMipiSh.cpp:622 [[@libcamera /home/alex/220mcu/220f/lan7808a - Selected sensor Format: 1536x864-506080x1536 - Selected union Format: 1536x864-pJMA
/home/alex/doorlogger/venv/lib/python3.11/site-packages/face_recognition/models/_load.py:7: UserWarning: pkg_resources is deprecated as an API. See https://setuptools.pypa.io/en/latest/pkg_resources.html. The pkg_resources package is slated for removal as early as 2025-11-30. Refrain from using this package or pin to setuptools<68.
  from pkg_resources import resource_filename

[52:04:42.085882225] [INFO] [[1:32w 2M0D [[1:32wCamera [[1:32wCamera_manager.cpp:328 [[@libcamera v6.5.0-0-037f664
[52:04:42.072987407] [INFO] [[1:32w WMN [[1:32wCameraSensorProperties [[1:32wCamera_sensor_properties.cpp:473 [[@libcamera static properties available for 'imx708_msr'
[52:04:42.073067322] [INFO] [[1:32w WMN [[1:32wCameraSensorProperties [[1:32wCamera_sensor_properties.cpp:475 [[@libcamera consider selecting the camera sensor properties database
[52:04:42.085756365] [INFO] [[1:32w WMN [[1:32wMipiSh [[1:32wMipiSh.cpp:48 [[@libcamera legacy SDR tuning - please consider moving SDR inside rpi device
[52:04:42.087847382] [INFO] [[1:32w WMN [[1:32wCameraSensor [[1:32wCamera_sensor_legacy.cpp:582 [[@libcamera delays found in static properties. Assuming unverified defaults.
[52:04:42.087847382] [INFO] [[1:32w 2M0D [[1:32wMipiSh [[1:32wMipiSh.cpp:487 [[@libcamera camera /home/alex/220mcu/220f/lan7808a to Union device /dev/media0 and ISP device /dev/media1
[52:04:42.087908792] [INFO] [[1:32w 2M0D [[1:32wMipiSh [[1:32wMipiSh.cpp:1323 [[@libcamera configuration file '/usr/share/libcamera/pipeline/rpi/v3d/rpi_apps.yaml'
[52:04:42.087908792] [INFO] [[1:32w 2M0D [[1:32wCamera [[1:32wCamera.cpp:1280 [[@libcamera (New/fingerprint stream): (0) Sublime-MANAGE (1) Sublime-UNMANAGE (2) STOP
[52:04:42.103364742] [INFO] [[1:32w 2M0D [[1:32wMipiSh [[1:32wMipiSh.cpp:622 [[@libcamera /home/alex/220mcu/220f/lan7808a - Selected sensor Format: 1536x864-506080x1536 - Selected union Format: 1536x864-pJMA
[52:04:42.103364742] [INFO] [[1:32w 2M0D [[1:32wMipiSh [[1:32wMipiSh.cpp:622 [[@libcamera /home/alex/220mcu/220f/lan7808a - Selected sensor Format: 1536x864-506080x1536 - Selected union Format: 1536x864-pJMA

```

## Network Tab Showing POST to /stop\_capture:

Name	X	Headers	Preview	Response	Initiator	>>
stop_capture		General				
dashboard						
chart.js						
amaz_20250603_042308.jpg						
amaz_20250603_042303.jpg						
amaz_20250603_042301.jpg						
amaz_20250603_042259.jpg						
amaz_20250603_042256.jpg						
amaz_20250603_042253.jpg						
amaz_20250603_042251.jpg						
amaz_20250603_042249.jpg						
amaz_20250603_042247.jpg						
amaz_20250603_042243.jpg						
daily-data						
14 requests	8.7 kB transferred					
Console						

Flask Console Showing Process Termination:

```
127.0.0.1 - - [03/Jun/2025 04:58:09] "GET /api/daily-data HTTP/1.1" 200 -  
127.0.0.1 - - [03/Jun/2025 04:58:12] "GET /stop_capture HTTP/1.1" 405 -
```

## Secure Login and Authentication:

Only after logging in via Firebase Authentication can users use the dashboard or access other API endpoints. An easy email/password form is shown to users when they visit the /login page.

When the user completes the form and clicks “Login,” the client-side JavaScript (from static/main.js) uses Firebase’s SDK to run a function.

You can authenticate users with the auth object by using their email and password via `signInWithEmailAndPassword(auth, email, pass)`.

When login is successful, Firebase sends a user credential back. The script passes on to `userCredential.user.getIdToken(true)` to retrieve the secure ID token for the user. This record demonstrates who the person is.

The token is delivered in a POST request to the /sessionLogin endpoint on the Flask server which checks it and creates a session for the user to access restricted features.

```
const app = initializeApp(firebaseConfig);  
const auth = getAuth(app);  
  
async function handleAuth(action) {  
  const email = document.getElementById("email").value;  
  const pass = document.getElementById("password").value;  
  
  try {  
    let userCredential;  
    if (action === "login") {  
      userCredential = await signInWithEmailAndPassword(auth, email, pass);  
    } else {  
      userCredential = await createUserWithEmailAndPassword(auth, email, pass);  
    }  
    const idToken = await userCredential.user.getIdToken();  
  
    await fetch("/sessionLogin", {  
      method: "POST",  
      headers: { "Content-Type": "application/json" },  
      body: JSON.stringify({ idToken })  
    });  
  
    window.location.href = "/dashboard";  
  } catch (e) {  
    document.getElementById("error-msg").innerText = `Firebase: ${e.code}`;  
  }  
}
```

Inside app.py on the server side, the /sessionLogin route manages the login process. Once a user logs in, the frontend sends these types of data as a JSON payload:

The idToken is: "..."

Flask calls the Firebase Admin SDK to check the token with `auth.verify_id_token(idToken)`. Flask will form a session for the user if the token is valid and add their email to the session like this:

```
session["user"] = { "email": decoded_token["email"] }
```

At that point, it responds `{ "status": "success" }` to let the user know they are now logged in.

Flask refuses the login if the token is not valid or has expired and replies with a HTTP 401 response and this message.

The service responded with, "Invalid ID token."

verified Firebase credentials is required to access the dashboard and all protected API routes in this system.

```
@app.route("/capture_logs")
def capture_logs():
    if "user" not in session:
        return redirect("/login")

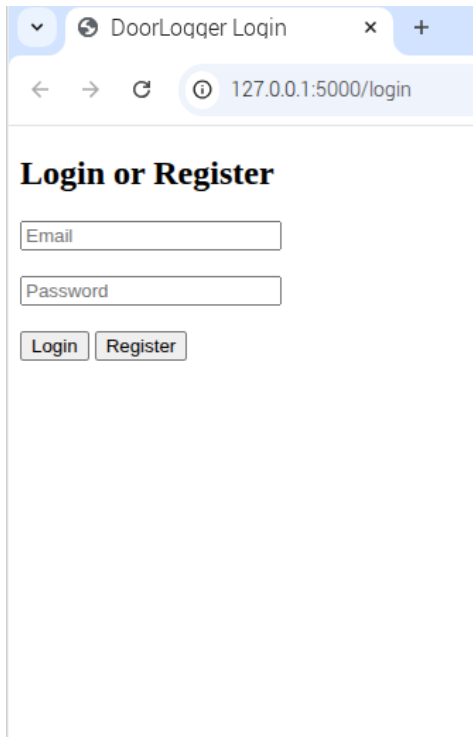
    log_path = "/home/amaz/doorlogger/capture.log"
    if not os.path.exists(log_path):
        logs = []
    else:
        with open(log_path, "r") as f:
            logs = f.readlines()
    return render_template("capture_logs.html", logs=logs)
```

Any time someone tries to visit /dashboard, /api/data or /api/daily-data without a valid login session, it will automatically send them to the login page. It gives important routes an extra level of security.

Flask keeps things secure by using a `secret_key` which comes from the .env file as `SECRET_KEY`. As it is used to sign session cookies, the key makes sure that no client can alter them. This way, just people who are verified can use and interact with the system.

## Screenshots:

Login Page UI:

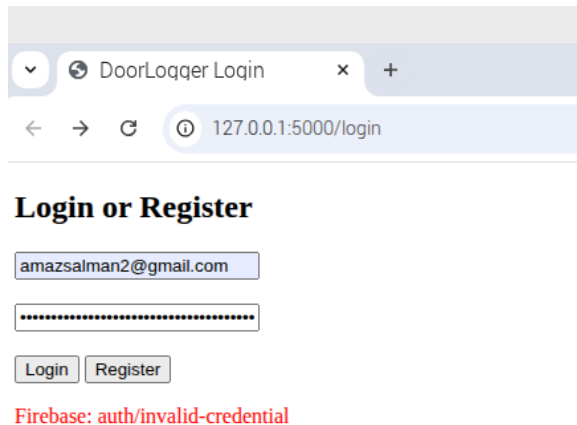


A screenshot of a web browser window showing the 'DoorLogger Login' page. The browser's address bar displays '127.0.0.1:5000/login'. The page has a title 'DoorLogger Login' and a main heading 'Login or Register'. Below the heading, there are two input fields: 'Email' and 'Password'. At the bottom of the form, there are two buttons: 'Login' and 'Register'.

Flask Console - Successful Token Verification

```
127.0.0.1 - - [03/Jun/2023 03:14:21] "GET /favicon.ico HTTP/1.1" 404  
DEBUG: Received ID token: eyJhbGciOi...  
DEBUG: Decoded token 'aud': doorlogger-auth
```

Unsuccessful Login:



The screenshot shows a web browser window with the title 'DoorLoggger Login'. The address bar displays '127.0.0.1:5000/login'. The page content includes a heading 'Login or Register', a text input field containing 'amazsalman2@gmail.com', a password input field with masked characters, and two buttons labeled 'Login' and 'Register'. Below the buttons, a red error message reads 'Firebase: auth/invalid-credential'.

## Notifications and Alerts

When a face is labeled “Unknown” by `face_capture.py`, the alert system notifies the administrators. As soon as the image is put into Cloud Storage and the result is logged in BigQuery, the script looks at whether the detected name is “Unknown”.

If an alert is triggered, a Discord webhook message is sent, showing the image and the time when it happened. This helps admins know right away if an intruder or an unfamiliar person has come on the property.

```
if name == "Unknown":
    with open(filepath, "rb") as f:
        response = requests.post(
            DISCORD_WEBHOOK_URL,
            files={"file": (filename, f)},
            data={"content": f"Unknown face detected at {timestamp}"}
        )
    print(f"[DISCORD] Notification sent, status: {response.status_code}")
```

The URL for the webhook is safely kept in .env and is read and used by the script when it is running. When the system finds an unknown face and saves the image, it sends a POST request to this webhook URL right away.

If you are approved, your linked Discord channel will visibility show:

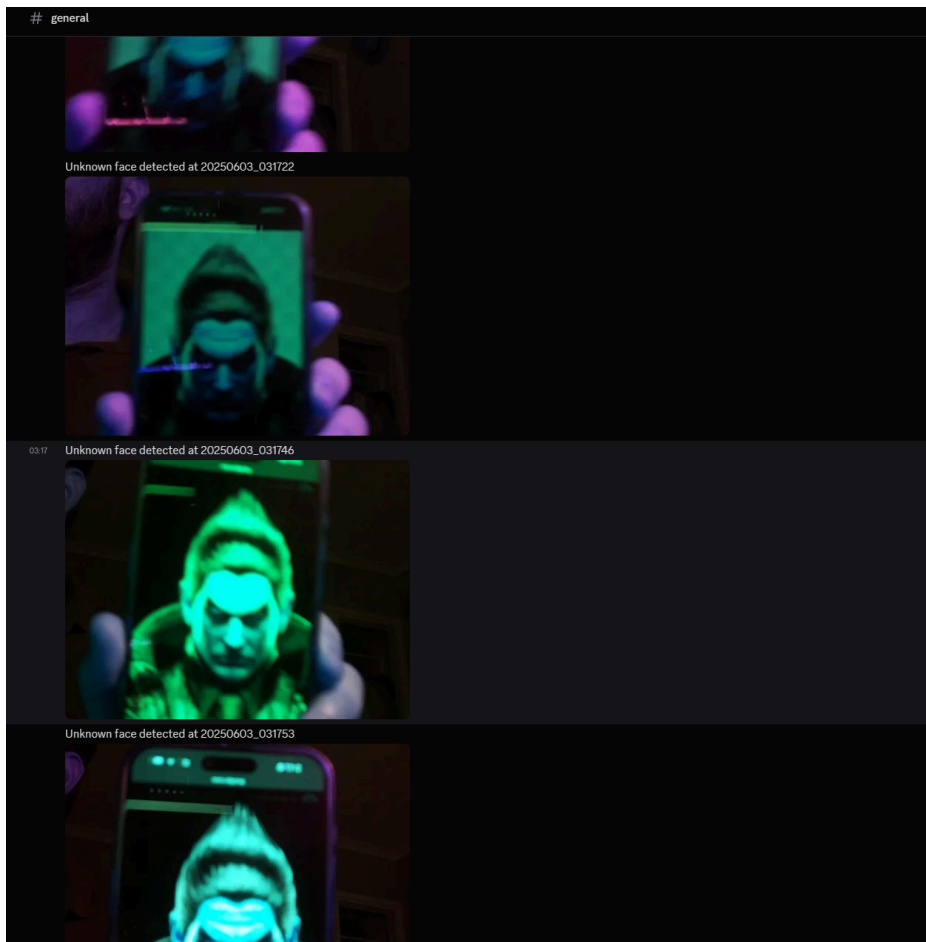
There was a face in the camera at 2025-06-02\_14:22:10 that was not identified.

The message also comes with the attached image which was pulled from Google Cloud Storage.

When an alert happens, the administrator watches the channel and gets immediate notification which eliminates the need to constantly monitor the dashboard or logs.

### Screenshot:

Discord channel notification:



Capture.log Showing Discord POST:

```
[DEBUG] Detected 1 face(s)
[DEBUG] Detected 1 face(s)
[ALERT] Unknown person detected!
[CLOUD] Uploaded Unknown_20250601_222231.jpg to GCS.
[BQ] Logged face event.
[DISCORD] Notification sent, status: 200
[DEBUG] Detected 0 face(s)
[DEBUG] Detected 0 face(s)
[DEBUG] Detected 0 face(s)
```

## Well-structured setup instructions and user manual

### Setup Instructions

Check that you have Python 3.11 or a later version and the Pi Camera Module 3 NoIR connected and set through the raspi-config menu before anything else. Create a directory, for example **/home/amaz/doorlogger** and clone the entire project into it. Make sure you are inside the folder and then run

```
python3 -m venv venv
```

after finishing the project, put it into operation

```
source venv/bin/activate
```

Run the command to install the necessary Python dependencies

**Install flask, python-dotenv, google-cloud-bigquery, google-cloud-storage, firebase-admin, face\_recognition, opencv-python, picamera2 and requests by running pip install [module name].**

Following this, generate a file named .env in the project's main folder filled with the below contents (changing placeholder values as needed).

```
REDIRECT_URI=http://localhost:5000/login/google/authorized
```

```
GOOGLE_APPLICATION_CREDENTIALS="/home/amaz/doorlogger/doorlogger-auth-firebase-adminsdk-fbsvc-0763fc5a09.json"
```

```
BIGQUERY_PROJECT_ID=door-logger-pi
```

**BIGQUERY\_DATASET=doorlogger\_logs**

**BIGQUERY\_TABLE=face\_events**

**BUCKET\_NAME=doorlogger-images**

**DISCORD\_WEBHOOK\_URL=https://discord.com/api/webhooks/1377657524857208872/o8HN-e**

**SECRET\_KEY=supersecretkey**

Move the JSON file, doorlogger-auth-firebase-adminsdk-fbsvc-0763fc5a09.json, into /home/amaz/doorlogger and change .env so that it points correctly to the file. From the Google Cloud Console, give the service account Storage Object Creator rights on the doorlogger-images bucket and Data Editor rights on the doorlogger\_logs dataset in project door-logger-pi. Confirm in BigQuery that a table called face\_events is present under door-logger-pi.doorlogger\_logs, with columns named person\_name (STRING), image\_name (STRING), timestamp (TIMESTAMP) and columns are partitioned by the DATE function applied to timestamp. Within Firebase Console, ensure Email/Password authentication is set up and make at least one user account. Put the JSON for the Firebase Admin service account in the project root as doorlogger-auth-firebase-adminsdk-fbsvc-0763fc5a09.json. A final step is to launch the Flask server by using

**python3 app.py**

inside the virtual environment after it has been turned on. Flask should show debug statements letting you know that Firebase Admin is working and BigQuery is ready for use. Open a web browser and go to <http://<Raspberry-Pi-IP>:5000/login> to start.

## **User Manual**

When your Flask application is active, go to the login page and input both a registered Firebase email and password. The Dashboard will appear after a successful login. Your user email is shown at the top of the Dashboard together with a Logout link. Start Capture and Stop Capture are the labels for the two buttons found beneath your user information. Pressing the Start icon sends a request to /start\_capture and the program face\_capture.py is launched in the background. The process is automatic and the program begins taking pictures from the Pi Camera, checking for faces, transferring the pictures to Google Cloud Storage, logging everything in BigQuery and sending alerts on unknown faces to Discord. The Dashboard's face detection logs include the last ten events, with the person's name, an image link from GCS and the time each event happened. Choosing the Stop button communicates with /stop\_capture which causes the face\_capture.py process to end. The Dashboard refreshes and you won't see any more face events until you click Start Capture again. Go to the Live Graph page by clicking on View Graph in the trend section. The Live Visitor Activity (Past Hour) line chart is labeled



there and it updates every five seconds, receiving its data from `/api/data`. It shows the tally of face-detected minutes for the last hour. Under it, a second bar chart labeled Daily Detections (Last 7 Days) changes every hour by using the `/api/daily-data` and displays the daily total of face events. If the face is not recognized, `face_capture.py` reports this by sending a message and an image to the webhook and a red banner reading "Unknown face detected!" appears briefly on the Live Graph page to inform admins right away. To leave your session, use the Logout button at the top of the Dashboard; you will go back to the login page. Once you end using the system, go back to the Flask terminal and press `Ctrl+C` to finish the server and then use the command `deactivate` to disable the virtual environment. Run `pskill -f face_capture.py` to end the running of `face_capture.py` if it is still working.