Drones gone wild: Using drones to aid wildlife monitoring

Team 12
Github

Thomas van der Sterren
thvan23@student.sdu.dk
University of Southern Denmark

Pol Blancafort Figueras
pobla23@student.sdu.dk
University of Southern Denmark

Theodor Paal Põlluste
thpul23@student.sdu.dk
University of Southern Denmark

Milosz Dresler
midre23@student.sdu.dk
University of Southern Denmark

January 18, 2026

1 Solution Approach

This section will describe the general approach to the problem as defined. It describes the high-level solution approach, the overall architecture and design.

1.1 Mindset

When building the system these tenets of the Unix philosophy were kept in mind where possible.

1. Compact, one-purposed program that does well one thing.
2. Use shell scripts to increase leverage and portability.
3. Incorporate security measures when applicable.
4. Use software leverage to your advantage.
5. Account for scalability in the project.
6. Choose portability over efficiency.

During the build process each module was built to work as standalone processes, that was connected afterwards through integration.

1.2 Architecture

The system consists of 5 different systems: Wildlife Camera (Raspberry Pi4), RaspberryPi Pico, ESP8266, the Drone (laptop) and the Cloud (laptop). The Raspberry Pi4 is used as the Wildlife camera, that is constantly connected to the ESP8266 through Wi-Fi and the RaspberryPi Pico with an USB cable. The Wildlife camera takes care of image acquisition and runs a local website. The ESP8266 works as a external trigger that through MQTT tells the camera to take an image, when a button is pressed on the ESP8266. The RaspberryPi Pico works as a rain detection sensor and wiper combo. The communication between the Wildlife camera and Pico is done through serial bus, but the decision making of when to wipe works through MQTT due to requirements. The drone communicates with the Wildlife camera through Wi-Fi and is used to update the internal clock of the Pi4, and to download all the images from the Wildlife camera when passing by. The cloud runs the images through an offline Large Language Model to describe the images and uploads the information about the images to GitHub.

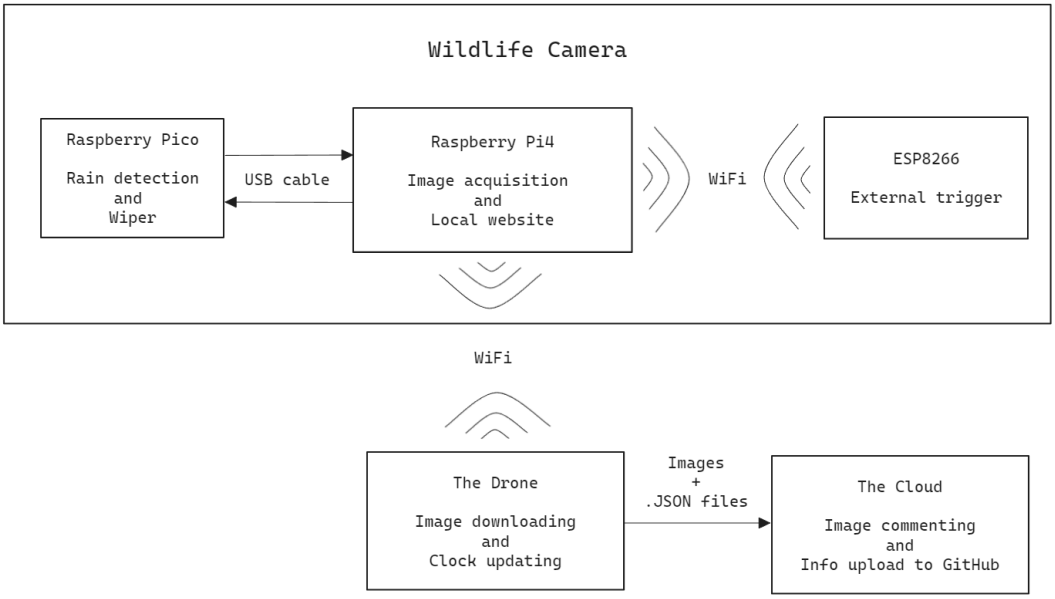

Figure 1: High-level architecture

1.3 Wildlife Camera

The Wildlife camera will have a start-up bash script that runs when the Pi4 boots up. Start-up script starts 4 indefinitely running scripts: motion detection, MQTT listener for external trigger, rain detection, and wiper control. Motion detection works by taking two images, one that was taken last run and an image that is taken this run and feeds them through a motion detection Python script. The Python script compares the two images, and outputs if motion was detected. The last image is then saved into the correct directory.
Listener for external trigger works by subscribing to a MQTT-broker (in this case Mosquitto) remote trigger topic. If the trigger word is read, the Wildlife camera takes an image and saves into the correct directory.
Rain detection reads the output of the Pico over USB and when rain is detected by the Pico it is published over MQTT in the Wildlife camera. It also listens to a different MQTT topic to receive the wipe command, after which it tells the Pico to wipe the servo motor connected to it.
Wiper control is a simple script that listens to one MQTT topic and publishes to another topic. When it reads that rain is detected it gives the command to wipe over MQTT.

1.4 The Drone

A laptop capable of Wi-Fi communication will act as the drone. It connects to the Wildlife camera when in range by actively checking for the SSID of the Wildlife camera over Wi-Fi. It must work without human input, apart from the script start, even after connection is lost temporarily. In terms of bandwidth, the setup is limited to 2.4GHz due to the usage of an ESP8266, which only supports this frequency. After the drone connects to the Wildlife camera, it will update the system time of the Wildlife camera. While connected, the signal quality and strength are logged. The main purpose of the the drone is to download the images and `.JSON` files from the Wildlife camera.

1.5 The Cloud

Once the images and `.JSON` files are collected by the drone, they are offloaded to a specified directory on the PC, acting as the cloud. A script, running continuously on the PC, checks the directory for new images and using a local Large Language Model (LLM Ollama) [1] adds annotations to `.JSON` files describing each of the photos. Then after performing those annotations on a batch of images, the `.JSON` files are moved to the local Git repository and then pushed to GitHub.

2 Solution description

The whole system is divided into 4 main sections: Image acquisition, Wiping process, Downloading to drone and the Cloud. The image acquisition part takes care of taking the images at the right time and generating the `.JSON` files as required. A log file of all the images taken is also kept. Wiping process takes care of detecting rain, communicating it through MQTT and keeping images from being taken while wiping is in process. The downloading to drone part checks if any Wildlife cameras are with-in range and connects to the one it finds. It then downloads the image log file and starts downloading the images found in that log. Drone log is only updated when download of the file is successful. This way, when connection is lost and the drone re-connects, the process restarts from the last pending image. It will iterate through the log file trying to download all new pending images. Finally the cloud part runs the images through an Ollama `LLaVA` 7b [1] model to give a description of what is present in the images. It adds the comment to the `.JSON` file of the image. After a batch of images has been ran through the describing model the `.JSON` files are committed to a team git repository.

2.1 Image acquisition

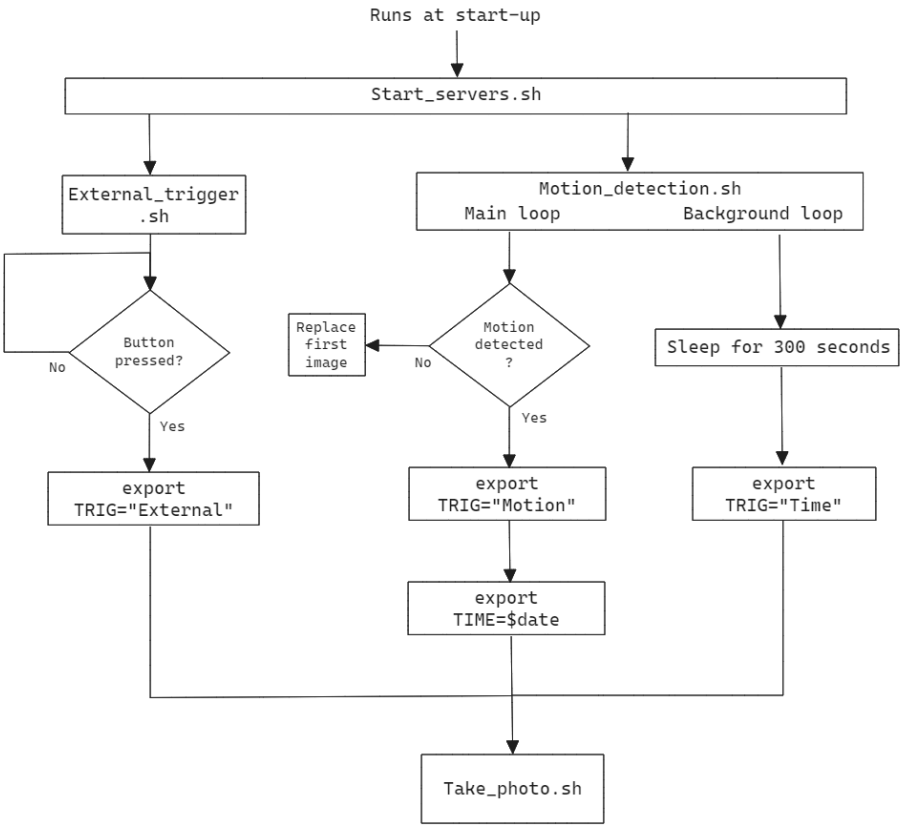

Figure 2: Image acquisition part UML

Upon boot a start-up script `start_servers.sh` runs, which starts all the other scripts of the Wildlife camera. One is the motion detection script `motion_detection.sh` which contains two loops. The main loop runs every 3 seconds. `Motion_detection.sh` works by checking if any images have already been taken by the `motion_detection.sh` script. If not then the initial image is taken using `rpicam-still` command [2] and saved to a temp folder. After the initial run another image is taken using `rpicam-still` command [2] and the image is saved to the same temp folder. The script then sends paths of both of these images as arguments to the `motion_detection.py` program, that calculates the difference between the two images using the OpenCV library [3]. The python program outputs "motion" if there is significant enough changes in the two images. If motion is detected the directory of the last image taken by `motion_detection.py` is passed with the trigger word "Motion" to the `take_photo.sh` bash script. Also the current date is exported to the `take_photo.sh` due to the time difference between when the image was taken and when the `motion_detection.py` finishes processing images. If there is no motion found, the first image

is deleted and the last image is renamed to be the next first image. Within the `motion_detection.sh` script there is also another loop running in the background which will run the `take_photo.sh` every 5 minutes with the trigger word "Time". The third trigger is actuated by a bash script `external_trigger.sh`, which is running continuously in the background and also initiated upon boot. It is continuously listening if a new message is published on a topic "remote-trigger/1/out", where the number "1" could be changed for different numbers in case of multiple external triggers. This makes the system scalable and it would be possible to know, which trigger was activated. If the message is found it exports the trigger word "External" to the `take_photo.sh` script. The external trigger will be further explained in section 2.1.2.

2.1.1 Taking Photos

In the `take_photo.sh` script a couple of different tasks are performed, the flow of actions is visualized in figure 3. Firstly the script tries to make a directory with the current date as the name of the directory. It then enters that directory and checks if the wiper is in the middle of wiping the lens by looking for a flag file in the temp folder of the system. If there is wiping in process it waits until the flag file is removed. This will be later discussed in section 2.2. After the flag file is gone, it continues with normal function by checking the trigger word. If the trigger word is "Motion" the function copies the image from the temp folder and renames it according to the timestamp exported by the `motion_detection.sh`. In the two other cases the script uses the `rpicam-still` command [2] to make an image and rename it as the current timestamp. A `.JSON` file is generated using ExifTool [4], accompanying the image, containing all the necessary information with the same name as the image. Finally the file name and directory are added to the `imagelog1.txt`. This log will later be used by the drone and described in section 2.4. The log file is numerated to keep track of different Wildlife cameras. It serves it's main purpose in the downloading images process.

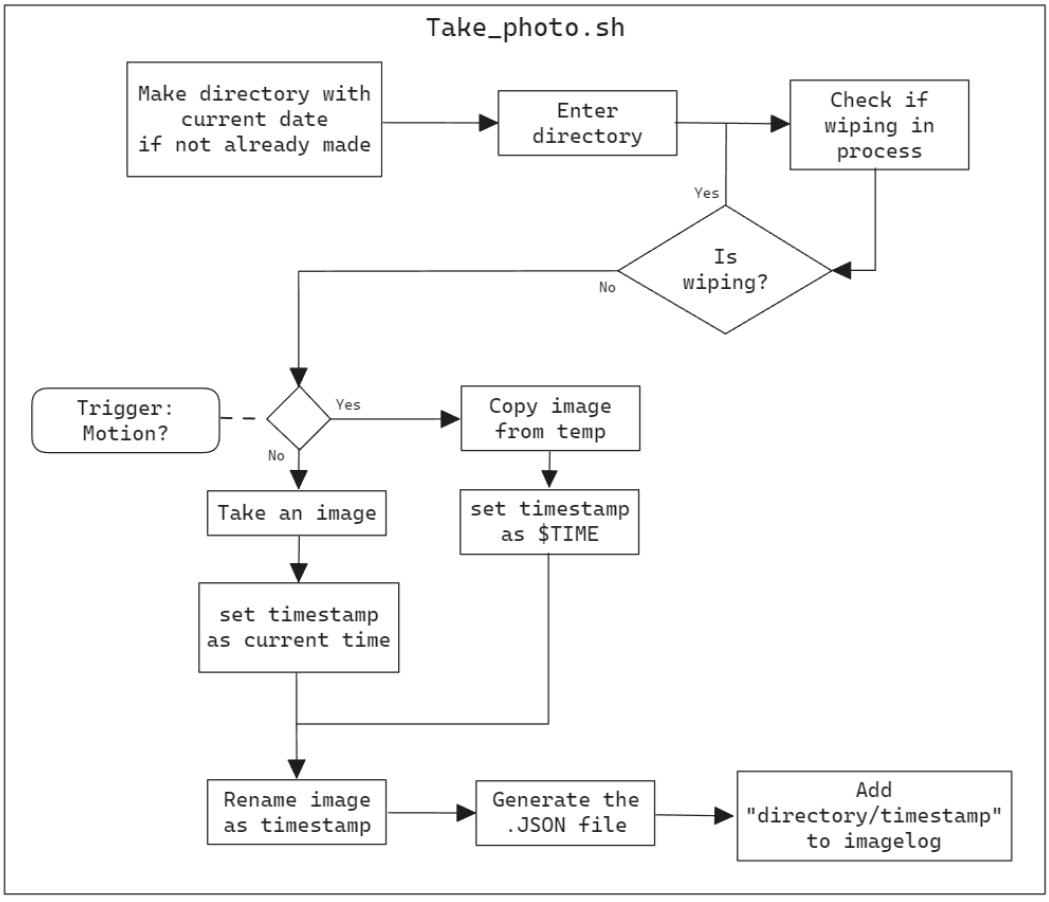

Figure 3: Take photo UML

2.1.2 External Trigger

As mentioned in the previous section, one of the triggers that can be handled is the External Trigger. This is developed using a ESP8266 which connects to the MQTT network over Wi-Fi. To ensure the ESP8266 can connect to the Wi-Fi, the Wildlife camera must be set to 2.4GHz as the ESP8266 is not able to connect to a 5GHz network. In the future, an ESP32 could be used to allow it to connect to a higher frequency network. This would mean that transferring data to the drone would also be able to go faster.

The code for the ESP8266 was created using the Arduino IDE. Within the code a couple of different function fire. Upon boot of the ESP8266, the pins are set to the correct settings, being able to take the button (simulating the pressure-plate or external sensor) as an input, and the onboard LED as output. It then connects to the Wi-Fi of the Wildlife camera. Once connection is successful, it will connect to the MQTT broker and register to the correct topic. Although the External Trigger is currently setup to only output when it has a trigger, the code is build in such a way that it also subscribes to an input topic. Allowing later developments of more complex triggers to be made. Once the button is a pressed, a trigger message is sent to the Wildlife camera, and will start the imaging process as mentioned earlier.

2.2 The Rain Wiper

To ensure good photos, even in the worst conditions, the Wildlife camera is equipped with a rain detection and wiping system. This system runs on a RaspberryPi Pico board, connected to the serial bus using a USB cable. The way the Wildlife camera and the Pico board handle data communication is portrayed in the figure 4 below.

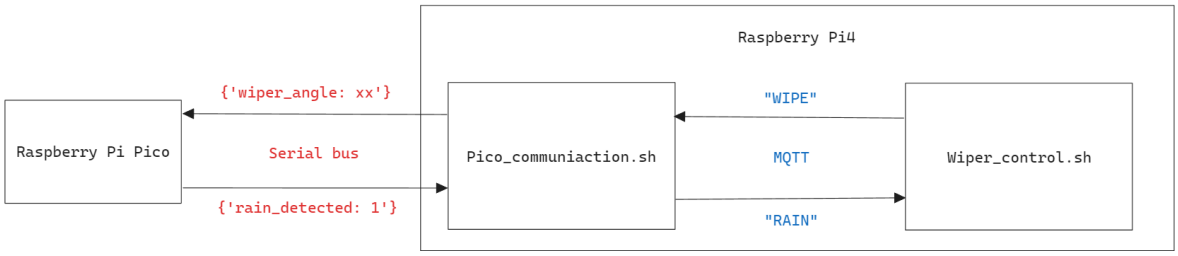

Figure 4: Flowchart to show the communication between the Wildlife Camera and the Pico Rain detector

As can be seen in the figure, the Pico receives an input from a rain sensor (emulated by a button press) and coordinates this to the Wildlife camera via Serial in `JSON` format. Additionally the Pico also continuously shares the position of the wiper to the Wildlife camera. When the Wildlife camera receives information about rain through the `pico_communication.sh`, it transfers this to its internal MQTT network to handle a response. The `wiper_control.sh` script will receive this MQTT message and tell the `pico_communication.sh` to start wiping, additionally it also creates a flag file that is used by the `take_photo.sh` script to ensure no photo gets captured while wiping. As it is not favorable to make an image of a wiper. When wiping is completed, `pico_communication.sh` will notify the `wiper_control.sh` that it finished. This allows `wiper_control.sh` to remove the flag file, allowing the images to be taken again.

2.3 Local Website

A local website hosted on the Wildlife camera, was created to provide easy access to images, `.JSON` files, and the log file. The website is served using the Apache2 web server hosted locally. The webserver user was set as the owner of the `/var/www/wildlife_camera/` data directory. The directories containing the images and `.JSON` files, as well as the log file, were symbolically linked with the corresponding directory/file in the `/var/www/wildlife_camera/`, named `images` (for images and `.JSON` files) and `logs/main_log` (for the log file).

The local website uses an HTML file, `index.html`, as the main page to provide hyperlinks to the images folder and the log file. The view of the page can be seen in Figure 5.

Figure 5: Local Website

2.4 The Drone

The final drone code is a shell script with 197 lines of code that can run on a standard Ubuntu laptop. Upon script startup, the code turns on a background process that takes care of two jobs: checks connection status every 1 second and logs LAN status. This *monitor_and_log_wifi* function performs a ping to the RPI IP address and depending on the output it updates a flag on a status file. This flag is later used by the main code to trigger the download process. Additionally, such function also performs the task of logging the level and strength of the Wi-Fi signal and inserts the results into a SQL database.

The main code runs indefinitely in a loop unless the code is stopped. The first thing the main code does is check for the SSID of the Wildlife camera. If not in range, it waits until the SSID is present. The main code accesses the Status File mentioned before with the "Connection Status" flag. This means that the drone always knows if it is connected to the Wildlife camera (i.e. ping successful). When in range, the main code encounters a while loop that is going to run only while connected. If the code does not enter this loop, it will echo "Not connected" state and try to reconnect (because at this point, the SSID has been found to be in range). When the Wi-Fi connection is successful, the flag will allow the code to start the sync-and-download process.

The sync process starts by retrieving the date/time of the drone. Then, the Wildlife camera is synced up with the laptop using a ssh command. Next up, a first connection check is performed. The whole idea is to ping the Wildlife camera in critical parts of the code to make sure connection is still up. In case the ping fails mid-code, a break forces the exit of the inner while loop and the entering of "Not connected" state. But, if everything works correctly, the drone will start the download process. This process works by comparing two files: the drone image log file and the Wildlife camera image log file. It is summarized in the next step list:

1. After retrieving the timestamp of both log files (drone and Wildlife camera) a comparison is made. If the Wildlife camera's log is newer, then the Wildlife camera's image log file is downloaded via ssh.

2. Having both files in the drone (local) the logs are compared for differences. Every new line that the Wildlife camera's file has, is an image that is needed to download. All differences (new lines) are stored in a dynamic variable, *differences*.

3. The code iterates through *differences* going line by line. For each line, the location and the name of the image are separated to download.

4. Just before downloading, a ping check is performed to confirm connection between continuing.

5. If the drone is still connected, the files of the current image (both `.jpg` and `.JSON` files) are downloaded. After the ssh download command, an evaluation of the outcome of the step is performed. Only when the download has been completely successful, the current line (in differences) is added to the drone image log. Additionally, after copying is successful, the original `.JSON` file from the Wildlife camera is edited by adding two keys (Drone number and epoch time).

6. The code will keep iterating through differences trying to download all new images until differences is empty. In that case, both log files are compared again. If the drone image log file is equal to the Wildlife camera's local log file, the loop is broken and the following massage is sent to terminal: "Step 3 completed. Drone has all new RPI images". Here the timestamp of the drone log file gets updated as well.

7. While connected, the code keeps checking if the Wildlife camera has a newer image log file to begin the difference comparison from step 1.

In case the drone is still connected, the time has been synced, and the comparison between files outputs have no difference, the code will echo "No new data to copy. Waiting..." and the script starts the loop again.

The final drone_script.sh code is summarized in the following diagram:

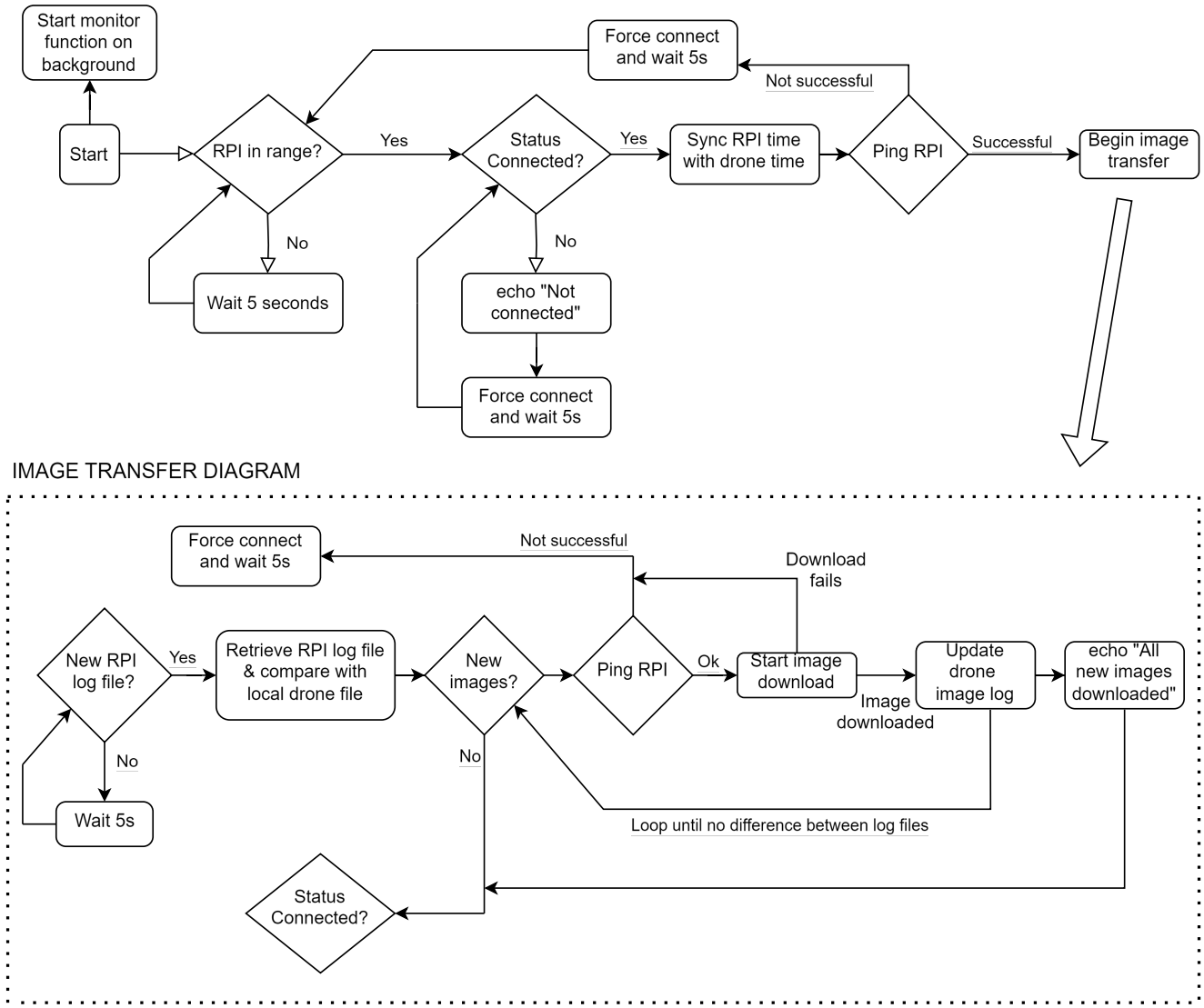

Figure 6: Flowchart of the drone steps to connect and download images.

2.5 The Cloud

First, to successfully use git, an SSH key was generated on the PC, which is then added in the GitHub for authentication. Then the repository for uploading `.JSON` files was cloned to that PC, using SSH URL.

The whole annotation and pushing to git process happens inside `annotations.sh` script that runs continuously on the PC. First the script checks for images in the specified offload directory, using `find` function to check for all existing `.jpg` files. Then it proceeds to handle each image by using the `process_image` function. Within this function there is a new directory made for storing all the `.JSON` files in their respected sub-directories (named from the date).

Next using `jq -e` [5], which sets an exit status based on the result of the filter, it is determined whether the file has been already annotated. The filter is set up to check if the output of `Source` in the current `.JSON` is `Ollama:'$MODEL_VERSION'`, with the current version of Ollama.

```
if jq -e '.Annotation."Source" | contains("Ollama:'$MODEL_VERSION'")' "$json_file" > /dev/null 2>&1
```

When this is true, the `.JSON` file is copied to the local Git directory. If it is not, then the process of creating an annotation is started.

First, the script runs the `LLaVA` model with the predefined version (7b) on the image and directs the output to `ollama_output`. Then the operations on the `.JSON` file start. First, the version name is assigned and then the variable `$text` is populated with the value of `ollama_output`. Both of these values are then assigned to the correct fields, `Source` and `Test` respectively.

Next that modification is added to the original `.JSON` file, with its contents redirected to a temporary file. If the `jq` [5] command succeeds, the original file is replaced with the updated temp file.

```
jq --arg version "Ollama:$MODEL_VERSION" --arg text "$ollama_output" \
          '.Annotation["Source"] = $version | .Annotation.Test = $text' "$json_file" > tmp.$$ && mv tmp.
    $$ "$json_file"
```

Next the `.JSON` file is copied to the local Git directory in the correct main directory and subdirectory. Then the file is staged, after this is done for all detected pairs of files (`.jpg` and `.JSON`), the `.JSON` files are pushed to the remote GitHub repository. After that, the whole procedure is repeated for the new batch of offloaded files, the pipeline for this operation can be seen on Figure 7.

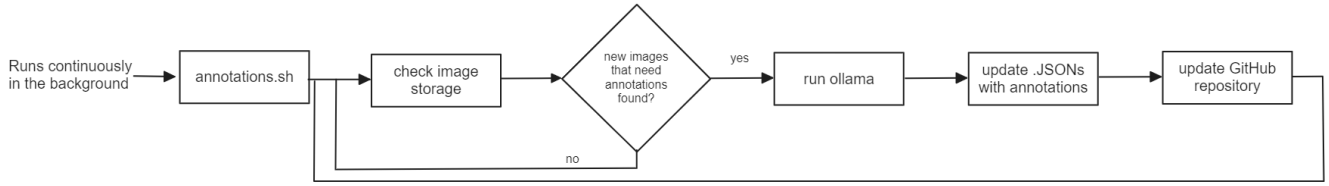

Figure 7: Cloud Operations UML

2.6 Security

During development, a basic level of security was implemented. For starters, the default passwords were changed to a more secure, uncommon password. Decreasing the chance of being brute-forced using common passcodes. Additionally the connection from the drone to the Wildlife camera, and the PC to the GitHub, make use of SSH Keys. These SSH Keys are a safer alternative to log in without having to enter a password. The public SSH keys is shared between the Drone and the Wildlife camera. In addition, the SSH private key can be further protected via password.

As stated earlier, a Wi-Fi connection is the primary method of communicating with the Wildlife camera. This could also be a vulnerability. Though for the offloading of images, it is essential to have this connection available. To mitigate risk, the Wi-Fi Protected Access second generation (WPA2) standard was used.

Within the network, MQTT is a communication protocol running. To ensure safety within this, a user account is created. Meaning that, to be able to communicate on the MQTT network, one must log in to the registered account with the correct password.

While working on the internal processes it is possible to log in to the Wildlife camera using SSH with password. This can be turned off once the product is going to be used. To create a current safer system, fail2ban is used. This system will temporally and eventually permanently ban an IP if it has too many failed login attempts. Preventing unwanted individuals from trying different passwords.

For the sake of current development, there was no implementation of safety regarding hardware.

3 Tests and Results

Each part was tested individually several times throughout the development process. The individual tests had the objective of verifying that each solution was getting the expected results and implemented all functional requirements. After obtaining optimal results from every part of the project separately, the team began integration of those parts into the complete project solution. After combining everything, the project was being tested as a whole with the constrains mentioned throughout the project.

To be sure that the parts all keep working with synergy, a log is kept within the Wildlife camera. This log will output important actions from the different scripts. Thus giving an operator the chance to look back in the rare case that anything failed.

3.1 Individual parts of the project

This section will focus on describing the methodology for testing and the gathered results for individual sections of the project:

Motion Detection
The motion detection was tested by running the motion detection script with the camera facing a natural scene, with mild wind blowing. The threshold was gradually enlarged to the point where the movement of the leaves did not trigger the motion detection. The threshold was left low enough that having medium sized objects enter the scene would trigger it. The script was also tested for false file type insertion, where it would just reject the output and re-run the script. This would then produce correct file formats and the script would fix itself.

External Trigger
The external trigger was tested using a tool called MQTT Explorer [6]. This tool allowed an in depth view of what messages were being transferred over MQTT. Additionally it also gives the opportunity to publish messages onto an MQTT topic from the tool. Giving easy debugging opportunities. With the tool at hand, the External trigger was tested by developing the ESP8266 part first, and then creating a listening code afterwards. Finally, the combination of the both was tested before implementing it in the boot script.

Rain Wiper
The Rain Wiper, just like the external trigger, was also tested hardware first. The hardware was tested by establishing the serial connection. This resulted in the returning of the `JSON` formatted data. After which the writing to the Serial was tested. Once this was successful, the MQTT connection was tested in similarly to the method described in the external trigger. After successfully passing the tests, it was also added to the boot script.

Drone
For the drone part, preliminary tests were performed online using 8.8.8.8 address. This allowed the testing of the drone script without having to rely on local RPi IP address. Further tests were then performed with the Wildlife Camera (RPi address) in a local network with the laptop acting as a drone to test the transfer of images. All final tests were performed offline to replicate the real-world scenario. The evaluation included checking if the new images where appearing, the status of the log file (that was being updated as the downloads occur) and the overall time it took to transfer the data. If both devices (drone and Wildlife camera) where in the same room, the transfer took less than one second. The test also included disconnection scenarios, where the two devices were being separated to check unconnected state. A check was performed to see how much time it took for the Wildlife camera to disappear from the drone Wifi SSID search and how long it took for the connection to be reestablished. The drone code successfully handles connection and re-connection between the drone and the Wildlife camera while monitoring the Wi-Fi status. Figure 8 shows a screenshot of the connection status log:

```
1716562209 | WiFi Quality: 70/70, Signal Level: level=-35
1716562209 | WiFi Quality: 70/70, Signal Level: level=-34
1716562210 | WiFi Quality: 70/70, Signal Level: level=-35
1716562210 | WiFi Quality: 70/70, Signal Level: level=-35
1716562211 | WiFi Quality: 70/70, Signal Level: level=-34
1716562211 | WiFi Quality: 70/70, Signal Level: level=-35
1716562212 | WiFi Quality: 70/70, Signal Level: level=-34
1716562212 | WiFi Quality: 70/70, Signal Level: level=-34
1716562213 | WiFi Quality: 70/70, Signal Level: level=-33
1716562213 | WiFi Quality: 70/70, Signal Level: level=-34
1716562214 | WiFi Quality: 70/70, Signal Level: level=-33
1716562214 | WiFi Quality: 70/70, Signal Level: level=-33
1716562215 | WiFi Quality: 70/70, Signal Level: level=-34
1716562215 | WiFi Quality: 70/70, Signal Level: level=-34
1716562216 | WiFi Quality: 70/70, Signal Level: level=-35
1716562216 | WiFi Quality: 70/70, Signal Level: level=-35
```

Figure 8: Log of Wi-Fi status while connected

Ollama image description

Describing images with Ollama and adding annotations to `.JSON` files was first tested by feeding the model images of animals found online. Next the annotation part was tested on artificially created `.JSON` files, with the specified structure. After obtaining promising results, the system was tested on images from the Wildlife Camera. The tests also involved detecting if an image is already described and does the algorithm find images in all subdirectories. The implementation of transferring the `.JSON` files into the correct subdirectories on GitHub was tested by performing several runs of the script with different directories and files. The results from the Wildlife camera images can be seen in the `annotations` directory on `Github`.

4 Conclusion

This project has proved challenging due to the amount of small parts that need to work together in order to solve the proposed task. With that said, the approach that the team took was modular and flexible, allowing each member to work independently in a specific part without relying on others code. Nonetheless, communication has been key for the integration part to assure all parts work correctly together.

The final result is a solution that covers all required parts with good results in our tests. In a constrained test scenario, our solution detects motion, handles external trigger and moves the servomotor when rain is detected. Addtionally, a second device can download such images during multiple runs and reconnect when needed. This second device has been successful in annotating the downloaded images before being pushed to a Github. Although the tests have been performed in a controlled situation with access to power supply, no adverse climatic conditions and full Wi-Fi range, the team is confident that our solution could be scalable and ready for the real-world after some modifications in power supply and automatic script startup. Our solutions features a basic but strong layer of security. However, this particular field could be improved further in future redesigns of the solution.

References

[1] "Ollama: Large language model platform." Accessed: 2024-06-02.

[2] R. P. Foundation, "Camera software documentation." `https://www.raspberrypi.com/documentation/computers/camera_software.html`, 2024. Accessed: 2024-06-01.

[3] G. Bradski, "The OpenCV Library," *Dr. Dobb's Journal of Software Tools*, 2000.

[4] Phil Harvey, "Exiftool," 2016.

[5] jqlang, "jq: Command-line json processor." Accessed: 2024-06-02.

[6] T. Nordquist, "Mqtt explorer," 2024. Accessed: 2024-05-02.