

# Lab 1: Thing-to-cloud End-to-end Setup

Handout: 10 Feb, Hand-in: 3 March (Thursday)

## Overview

In this lab, your group will work together and test the **two** different approaches below to setup end-to-end thing-to-cloud systems.

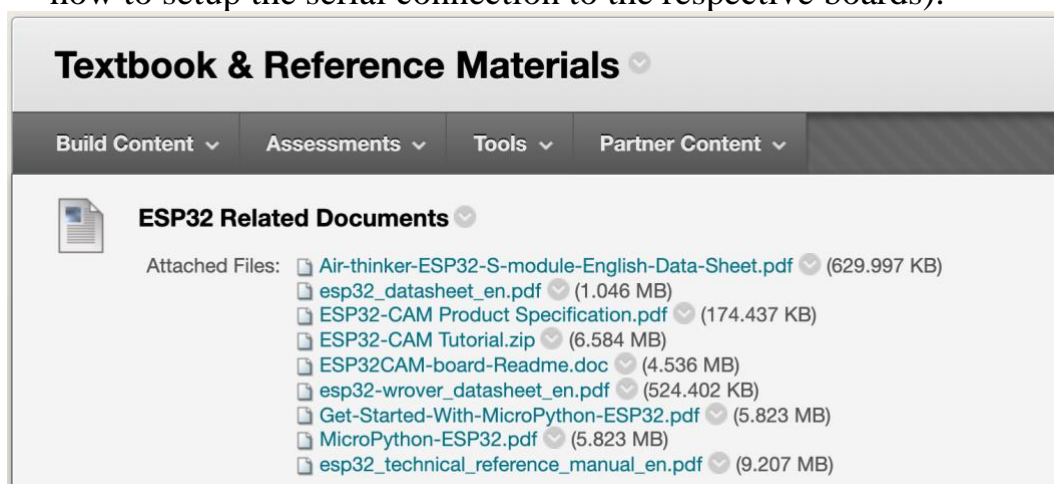
- 1) **An object recognition system using the ESP32-CAM and AWS Services**, by following the instructions in the Annex A (original post from <https://aws.amazon.com/blogs/iot/creating-object-recognition-with-espessif-esp32/> We have worked with CCIoT Batch 2021 students to provide an edited version with errata and hints in the Annex A.)

**Your task** is to let your ESP32-CAM light up different LED when pointing to different subject:

- Red LED: a pet
- Green LED: a person

Hints:

- As pets are not allowed in our classroom, you can point the ESP32-CAM camera to your laptop screen displaying some photo instead. 😊
- Refer to the resources in eDimension (e.g., ESP32-CAM Tutorial.zip, and MicroPython-ESP32.pdf) for startup guides (e.g., how to setup the serial connection to the respective boards).



The screenshot shows a web interface titled "Textbook & Reference Materials" with a dropdown arrow. Below the title is a navigation bar with four tabs: "Build Content", "Assessments", "Tools", and "Partner Content", each with a dropdown arrow. Under the "Assessments" tab, there is a section titled "ESP32 Related Documents" with a document icon and a dropdown arrow. Below this title, it says "Attached Files:" followed by a list of files with their sizes in parentheses:

- Air-thinker-ESP32-S-module-English-Data-Sheet.pdf (629.997 KB)
- esp32\_datasheet\_en.pdf (1.046 MB)
- ESP32-CAM Product Specification.pdf (174.437 KB)
- ESP32-CAM Tutorial.zip (6.584 MB)
- ESP32CAM-board-Readme.doc (4.536 MB)
- esp32-wrover\_datasheet\_en.pdf (524.402 KB)
- Get-Started-With-MicroPython-ESP32.pdf (5.823 MB)
- MicroPython-ESP32.pdf (5.823 MB)
- esp32\_technical\_reference\_manual\_en.pdf (9.207 MB)

- When setting up the ESP32-CAM, if after connecting the board to your PC with the USB it fails to detect or read the COM port (or

you get error 10), please check your device manager under Ports (COM & LPT) if the below appears:

Ports (COM & LPT)

PL2303HXA PHASED OUT SINCE 2012. PLEASE CONTACT YOUR SUPPLIER.

please install the older version of the driver

(<http://wp.brodzinski.net/2014/10/01/fake-pl2303-how-to-install/>, find a link entitled

IO-Cable PL-2303 Drivers-Generic Windows PL2303 Prolific

change which driver the USB is using (Right click -> Update driver -> Browse Computer -> Let me pick from drivers on my computer -> pick version 3.3.2.105)

- When uploading your code, remember to hold the reset button for a while.
- The hardware may not be very stable (e.g., the camera), you can switch to another one, and ask for a replacement from our lab officer, if you found it faulty.

## 2) ESP Rainmaker platform based setup

<https://rainmaker.espressif.com/>

<https://rainmaker.espressif.com/docs/get-started.html>

*“ ESP RainMaker is an end-to-end platform that enables Makers to realize their IoT ideas faster with Espressif's ESP32-S2/ESP32 SoC without hassle of managing any infrastructure. It provides a device SDK, self-adapting phone apps, transparent cloud service and host utilities to reduce complexity in development. ”*

**Your task** is to use Rainmaker's phone app to interact with your ESP32 board (e.g., toggle an LED, or do something more fun).

Hints:

- The provided code is based on a different board, so its GPIO pin to LED may not work on your chosen board. Change that accordingly.
- Avoid space in your file path when downloading the code to your local computer.

## What to hand-in?

- 1) **Demonstrate** to Prof. Binbin your two working systems **by 3 March (Thur)**. You can show us earlier (e.g., on 17 Feb, or after sessions in Week 6, or other slots by appointment), when your group completes the

tasks. You can show us the two systems in two different time slots (before the deadline). We will take note of what you have demonstrated.

**Regarding the hardware:** For the first setup, you need to use ESP32-CAM. For the second setup, you can use either the ESP32-CAM board or the MakePython development board.

**Regarding the development environment:** We recommend you to follow the instructions from the provided guides (e.g., instead of Arduino IDE or uPython, in this lab, we recommend you to use the command line ESP idf.py tool directly and the PlatformIO IDE). You need to study and understand the overall structure of the provided code for the respective setup, especially how the application code uses the corresponding device SDK from AWS and Espressif respectively. On the other hand, you should be able to accomplish your tasks without much need to edit the code. [Note: for your course project, you can consider continuing to use the lab's development environment, or you can switch to other IDEs and software stack, e.g., Arduino, uPython, if you prefer.]

- 2) Write a brief **lab report** documenting the following:
- Did you encounter any issue(s) during the setup?
    - For each issue you encountered, briefly describe the issue and how you solve it.
  - Draw one diagram to explain the architecture of the Rainmaker and its provisioning process.
  - Reflection on the difference of these two approaches. Comment on some of their pros and cons.

Submit your lab report to eDimension before 3 March 23:59. This is a group-based lab and each group just need to submit one lab report. Indicate your group ID and all the members' names and student IDs (including audit students, if any) in your lab report.

Total marks for this lab: 30pt

- 15pt: ESP32cam object recognition demo
- 7pt: Rainmaker demo
- 8pt: Lab report

# Annex A: Creating Object Recognition with Espressif ESP32

**An edited version based on the following original post**

by Nikolai Danylchyk | on 02 OCT 2020 | [Original-link](#)

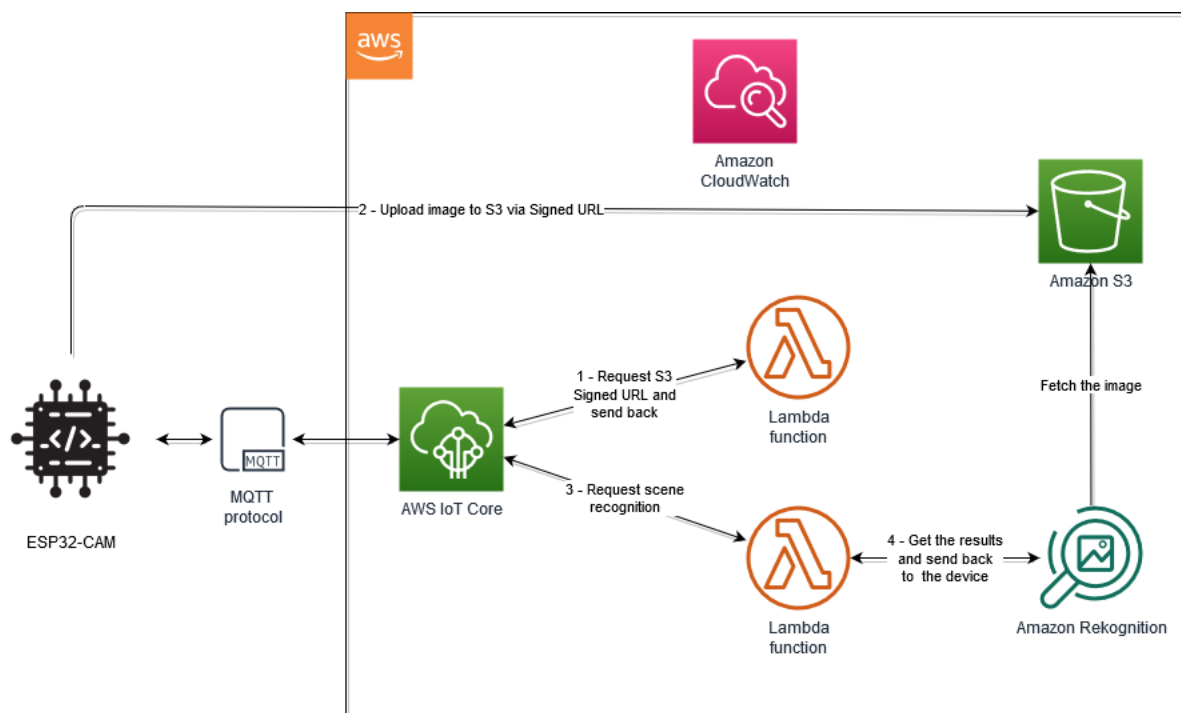
**Thanks to SUTD CCIoT course students (the pioneer batch of 2021) for their contributions to the errata and hints**

By using low-cost embedded devices like the [Espressif ESP32 family](#), and the breadth of AWS services, you can create an advanced object recognition system.

[ESP32 microcontroller](#) is a highly integrated solution for Wi-Fi and Bluetooth IoT applications, with around 20 external components. In this example, we use AI Thinker ESP32-CAM variant that comes with an OV2640 camera module. This module is a low voltage CMOS image sensor providing full functionality of a single-chip UXGA (1632×1232) camera and image processor in a small footprint package.

As a development platform, we will be using [PlatformIO](#). It is a cross-platform, cross-architecture, multi-framework, professional tool for embedded systems engineers and for software developers who write applications for embedded products. We will be using it to program our ESP32-CAM microcontroller.

This following diagram demonstrates a connection between an ESP32-CAM and [AWS IoT Core](#). It allows publishing and subscribing to MQTT topics. This means that the device can send any arbitrary information to AWS IoT Core while also being able to receive commands back.



## Solution overview

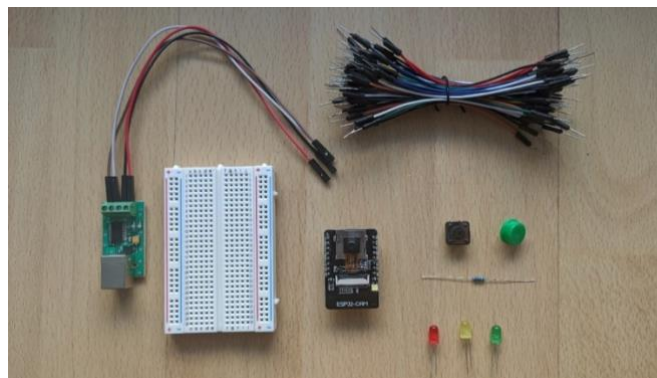
This post will walk you through building a complete object recognition solution, starting with deploying a serverless project on AWS that handles the communication between the cloud and ESP32-CAM device and then setup of AWS IoT Device SDK for Embedded C inside PlatformIO project.

The list of services that we will be using is as follows:

- [AWS IoT Core](#)
- [Amazon Rekognition](#)
- [Amazon S3](#)
- [AWS Lambda](#)

Required equipment:

- AI Thinker ESP32-CAM
- USB – TTL Serial Adapter
- Breadboard
- 3x LEDs
- 3x 330 Ohm resistors
- Jumper Wires
- 1x button



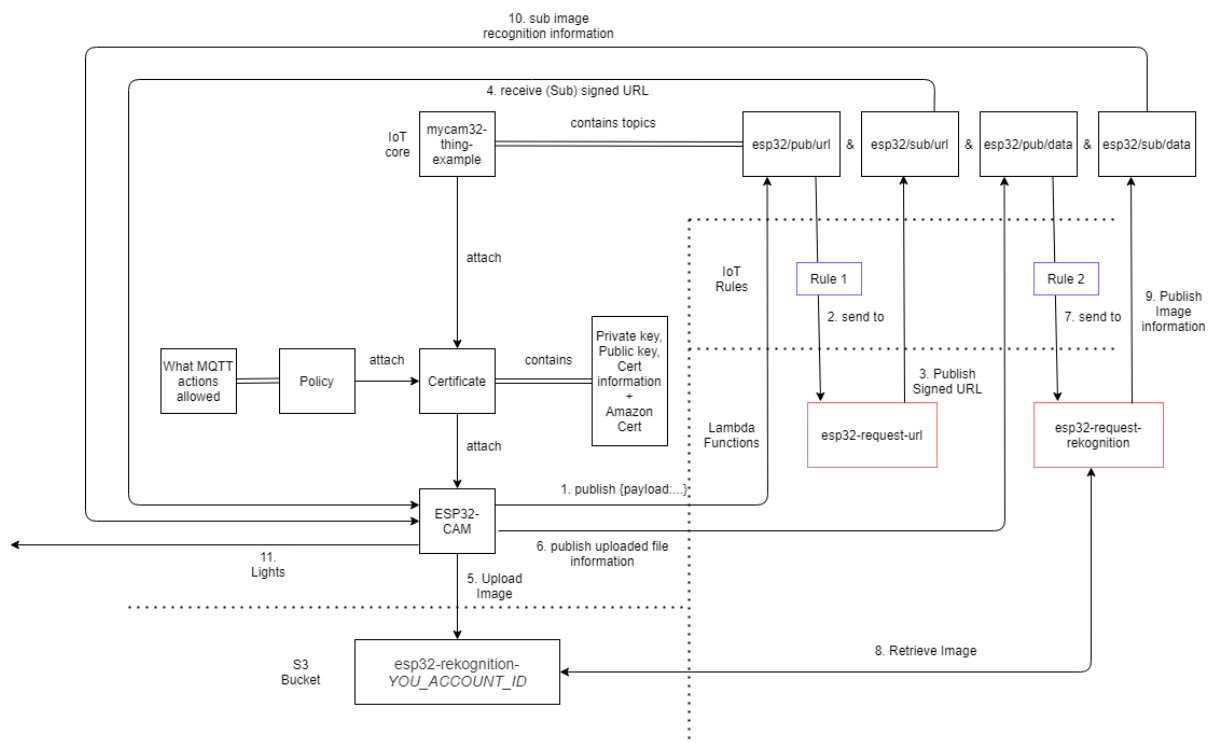
Hint:

Before you can upload the code to the ESP32- cam board, you need to ensure that the GND and GPIO 0 pin is connected. After a successful build and flash process, the wire connecting GND and GPIO has to be removed. After the removal of the wire, you need to press the button on the ESP32- cam board, after which the program will run on the ESP32- cam board.

In addition, the USB-TTL serial adapter that connects the ESP32- cam board to the laptop uses an older version of the driver which needs to be downloaded, if not the port would not be recognized by the laptop.

High-level overview of the steps involved in this demo:

- [Create an AWS IoT device](#) and export the certificates
- [Create an S3 bucket](#)
- [Create two Lambda functions](#)
- [Create two AWS IoT Core rules](#)
- Install and configure the [Visual Studio Code](#) with [PlatformIO](#) extension
- Install Espressif 32 platform inside PlatformIO
- Assemble all the components on the breadboard
- Download the project source code, build and flash it to ESP32-CAM device
- Press the button and point the camera towards any subject
- Observe the LEDs (Red: animal, Green: person, Yellow: everything else)
- Check the serial port and [Amazon CloudWatch](#) for logs



**Credit: Jo-Shen, ISTD Class 2021**

This solution relies on the [MQTT protocol](#) to communicate with the cloud. The dataflow starts at the ESP32 embedded device that sends an MQTT message once the button is pressed. An IoT rule forwards this message to a Lambda function that generates an S3 signed URL and sends it back. Once the device receives the URL, it takes the framebuffer data from the camera module and does an HTTPS request to S3 to POST the image. On successful upload, ESP32 sends another MQTT message to a Lambda function and provides the name of the uploaded file. This Lambda function then calls Amazon Rekognition API, identifies the scene on the image and sends the results back to the embedded device that then decides which LED to turn on.

# Creating an AWS IoT device

To communicate with the ESP32 device, it must connect to AWS IoT Core with device credentials. You must specify the MQTT topics it has permissions to publish and subscribe on.

1. In the [AWS IoT console](#), choose Manage, Things and click [Create](#).
2. Name the new thing myesp32-cam-example. Leave the remaining fields set to their defaults. Choose Next.
3. Choose Create certificate. Only the thing cert, private key, and Amazon Root CA 1 downloads are necessary for the ESP32 to connect. Download and save them somewhere secure, as they are used when programming the ESP32 device.
4. Click on Activate and then click on Attach a policy.
5. Click on Register Thing without attaching any policies at this step.
6. In the AWS IoT console side menu, choose Secure, Policies, Create a policy.
7. Name the policy Esp32Policy. Choose the Advanced tab.
8. Paste in the following policy template swapping out **REGION** and **ACCOUNT\_ID**.
9. Remember to attach your policy.

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": "iot:Connect",
      "Resource": "arn:aws:iot:REGION:ACCOUNT_ID:client/myesp32-cam-example"
    },
    {
      "Effect": "Allow",
      "Action": "iot:Subscribe",
      "Resource": "arn:aws:iot:REGION:ACCOUNT_ID:topicfilter/esp32/sub/+"
    },
    {
      "Effect": "Allow",
      "Action": "iot:Receive",
      "Resource": "arn:aws:iot:REGION:ACCOUNT_ID:topic/esp32/sub/url"
    },
    {
      "Effect": "Allow",
      "Action": "iot:Receive",
      "Resource": "arn:aws:iot:REGION:ACCOUNT_ID:topic/esp32/sub/data"
    },
    {
      "Effect": "Allow",
      "Action": "iot:Publish",
      "Resource": "arn:aws:iot:REGION:ACCOUNT_ID:topic/esp32/pub/data"
    },
    {
      "Effect": "Allow",
      "Action": "iot:Publish",
      "Resource": "arn:aws:iot:REGION:ACCOUNT_ID:topic/esp32/pub/url"
    }
  ]
}
```

### ERRATA:

Original post gives subscription permission to

```
{
  "Effect": "Allow",
  "Action": "iot:Subscribe",
  "Resource":
"arn:aws:iot:REGION:ACCOUNT_ID:topicfilter/esp32/sub/data"
},
{
  "Effect": "Allow",
  "Action": "iot:Subscribe",
  "Resource":
"arn:aws:iot:REGION:ACCOUNT_ID:topicfilter/esp32/sub/url"
},
```

While in the provided ESP32 code, it tries to subscribe to .../sub/+, which will be denied.

The step 9 was not mentioned in the original post.

## Creating an S3 bucket

To store the images from ESP32-CAM we need to [create an S3 bucket](#). Create a bucket called “esp32-rekognition-*YOUR\_ACCOUNT\_ID*”. Please use your account ID as a suffix to the bucket name. S3 bucket names are unique and this technique simplifies the process of choosing a non-taken name. Next, we make sure that your bucket is private.

In the permissions, set the bucket policy to the following. Modify account ID and username placeholders respectively:

```
{
  "Version": "2012-10-17",
  "Id": "Policy1547200240036",
  "Statement": [
    {
      "Sid": "Stmt1547200205482",
      "Effect": "Allow",
      "Principal": {
        "AWS": "arn:aws:iam::YOUR_ACCOUNT_ID:user/YOUR_USERNAME"
      },
      "Action": [
        "s3:GetObject",
        "s3:PutObject"
      ],
      "Resource": "arn:aws:s3:::esp32-rekognition-YOUR_ACCOUNT_ID/*"
    }
  ]
}
```



Also, paste the following CORS configuration (in JSON format) to allow cross account requests:

```
[
  {
    "AllowedHeaders": [
      "*"
    ],
    "AllowedMethods": [
      "GET",
      "PUT",
      "POST",
      "DELETE"
    ],
    "AllowedOrigins": [
      "*"
    ],
    "ExposeHeaders": []
  }
]
```

To learn more about Cross-Origin Resource Sharing (CORS), see:

<https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS>

#### ERRATA:

Note that the original post uses the following XML format configuration, which is not supported.

```
<?xml version="1.0" encoding="UTF-8"?>
<CORSConfiguration xmlns="http://s3.amazonaws.com/doc/2006-03-01/">
  <CORSRule>
    <AllowedOrigin>*</AllowedOrigin>
    <AllowedMethod>GET</AllowedMethod>
    <AllowedMethod>POST</AllowedMethod>
    <AllowedMethod>PUT</AllowedMethod>
    <AllowedMethod>DELETE</AllowedMethod>
    <AllowedHeader>*</AllowedHeader>
  </CORSRule>
</CORSConfiguration>
```

# Creating Lambda functions

Lambda functions are responsible for generating [S3 Signed URLs](#) as well as interacting with Amazon Rekognition API.

First, we create the Lambda function responsible for getting S3 Signed URLs. Name it 'esp32-request-url', choose the 'Python 3.8' runtime, and add the following code, swapping out your REGION and account ID:

```
import boto3
from botocore.client import Config
import json
import uuid

def lambda_handler(event, context):
    bucket_name = 'esp32-rekognition-YOUR_ACCOUNT_ID'
    file_name = str(uuid.uuid4()) + '.jpg'

    mqtt = boto3.client('iot-data', region_name='REGION')
    s3 = boto3.client('s3', config=Config(signature_version='s3v4'))

    url = s3.generate_presigned_url('put_object',
    Params={'Bucket':bucket_name, 'Key':file_name}, ExpiresIn=600,
    HttpMethod='PUT')
    # command = "curl --request PUT --upload-file {} '{}'.format(file_name,
    url)
    # print(command) # for local testing purpose
    # print(file_name + '/' + url[8:]) # for local testing purposes

    response = mqtt.publish(
        topic='esp32/sub/url',
        qos=0,
        payload=file_name + '/' + url[8:]
    )
```

## ERRATA:

note that there is an error in the bucket name in the original post (We should define bucket name as 'esp32-rekognition-YOUR\_ACCOUNT\_ID' NOT 'esp32-rekognition\_YOUR\_ACCOUNT\_ID' as in the original post)

Also, the code was changed from

```
s3 = boto3.client('s3')      to
s3 = boto3.client('s3', config=Config(signature_version= 's_3_v_4_'))
```

<https://aws.amazon.com/blogs/aws/amazon-s3-update-sigv2-deprecation-period-extended-modified/>

### Services deprecating Signature Version 2

- [Amazon Simple Storage Service \(Amazon S3\) - Amazon S3 Update - SigV2 Deprecation](#)

Now we modify your new Lambda functions execution role, adding the following permissions, so that it can generate S3 Signed URLs, and publish them to the MQTT topic:

```
{
  "Sid": "VisualEditor3",
  "Effect": "Allow",
  "Action": [
    "iot:Publish"
  ],
  "Resource": "*"
},
{
  "Sid": "VisualEditor2",
  "Effect": "Allow",
  "Action": [
    "s3:*"
  ],
  "Resource": [
    "arn:aws:s3:::esp32-rekognition-YOUR_ACCOUNT_ID",
    "arn:aws:s3:::esp32-rekognition-YOUR_ACCOUNT_ID/*"
  ]
}
```

#### Hint:

Set up permissions for Lambda functions at configuration->permissions->execution role->role name and click role name. This will bring you to the Roles management page where you have to click on the policy name ->permissions->edit policy and add the given JSON text.

Now create the second Lambda function. Name it “esp32-request-rekognition” and follow the same structure as the first, with Python 3.8 runtime. Paste the following code:

```
import boto3
import json

def detect_labels(bucket, key, max_labels=10, min_confidence=90,
region="REGION"):
    rekognition = boto3.client("rekognition", region)
    response = rekognition.detect_labels(
        Image={
            "S3Object": {
                "Bucket": bucket,
                "Name": key,
            }
        },
        MaxLabels=max_labels,
        MinConfidence=min_confidence,
    )
    return response['Labels']

def lambda_handler(event, context):
    results = ''
    mqtt = boto3.client('iot-data', region_name='REGION')
```

```

bucket_name = 'esp32-rekognition-YOUR_ACCOUNT_ID'
file_name = str(event['payload'])

for label in detect_labels(bucket_name, file_name):
    if (float(label['Confidence']) > 90):
        results += (label['Name'] + ';')

response = mqtt.publish(
    topic='esp32/sub/data',
    qos=0,
    payload=results
)

```

Again, we need to modify function's execution role, adding the following permissions, so that it can invoke Rekognition detection service and publish the results to the MQTT topic:

```

{
    "Sid": "VisualEditor0",
    "Effect": "Allow",
    "Action": [
        "rekognition:DetectLabels",
        "iot:Publish"
    ],
    "Resource": "*"
},
{
    "Sid": "VisualEditor1",
    "Effect": "Allow",
    "Action": [
        "s3:GetObject"
    ],
    "Resource": [
        "arn:aws:s3:::esp32-rekognition-YOUR_ACCOUNT_ID",
        "arn:aws:s3:::esp32-rekognition-YOUR_ACCOUNT_ID/*"
    ]
}

```

#### ERRATA:

In the original post, there is an extra , in

```

"Action":[
    "s3:GetObject",
]

```

#### Hint:

Remember to add an extra , between the {}s

These permissions allow Lambda function to talk to Amazon Rekognition API and publish the results to another MQTT topic.

# Creating IoT Core rules

To allow AWS IoT Core to Invoke our Lambda functions when MQTT messages arrive, we need to create two AWS IoT Core rules.

The first rule will forward the request esp32-request-url Lambda function. For that, you need to specify the following query statement:

```
SELECT * FROM 'esp32/pub/url'
```

Select “Send a message to a Lambda function” as an Action and point it to “esp32-request-url” Lambda function. Optionally you can add error action and forward the results to [CloudWatch logs](#).

The second rule will forward the request to the esp32-request-rekognition Lambda function. For that you need to specify the following query statement:

```
SELECT * FROM 'esp32/pub/data'
```

## ERRATA:

The original post misses an ' at the end.

Select “Send a message to a Lambda function” as an Action and point it to “esp32-request-rekognition” Lambda function. Also, here you can add an error action and forward the results to [CloudWatch logs](#).

## Testing the cloud solution.

Before we move on to the embedded device, let us test this solution.

First, let’s test if our Lambda function returns signed URLs for our ESP32-Cam to use as an upload location. Open AWS IoT core and go to “Test” section.

Subscribe to esp32/sub/url MQTT topic

Publish the following json to esp32/pub/url MQTT topic:

```
{"payload": "virtual-esp32-device"}
```

If you check on the topic that you subscribed to, you should receive a response with the filename/signed URL.

You can test this signed URL with tools like [Postman](#). Paste the URL, select the PUT method, and set the body to 'binary', which allows you to select the file you want to upload. Once you submit the request, check your S3 bucket, a new file should be there.

Second, let's upload an image called *test.jpg* to the S3 bucket that you created earlier. Then open AWS IoT core and go to "Test" section again.

Subscribe to `esp32/sub/data` MQTT topic

Publish the following JSON to `esp32/pub/data` MQTT topic:

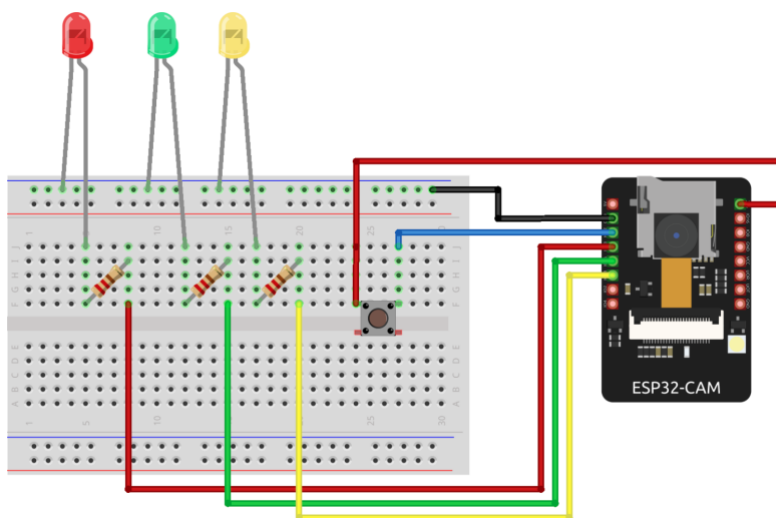
```
{"payload": "test.jpg"}
```

Check the response in the `esp32/sub/data` topic. You should see keywords associated to what Amazon Rekognition service identified on your test image.

## Setting up ESP32-CAM

Before we start, make sure you have the right components at hand.

Use the following diagram to assemble this solution:



Download the source of this [demo project](#).

Hints:

### Enabling the following:

Component Config -> Driver Configurations -> RTCIO configuration -> Support array 'rtc\_gpio\_desc'

Otherwise, PlatformIO throws the following error when trying to flash the ESP32:  
error: 'rtc\_gpio\_desc' undeclared (first use in this function); did you mean 'rtc\_io\_desc'?

Modified the code to reset the led every time a button press is detected.

```
if (buttonTrigger) {  
    ESP_LOGI(TAG, "Button Pressed");  
    gpio_set_level(RED_LED_GPIO,0);  
    gpio_set_level(GREEN_LED_GPIO,0);  
}
```

This project has the following structure:

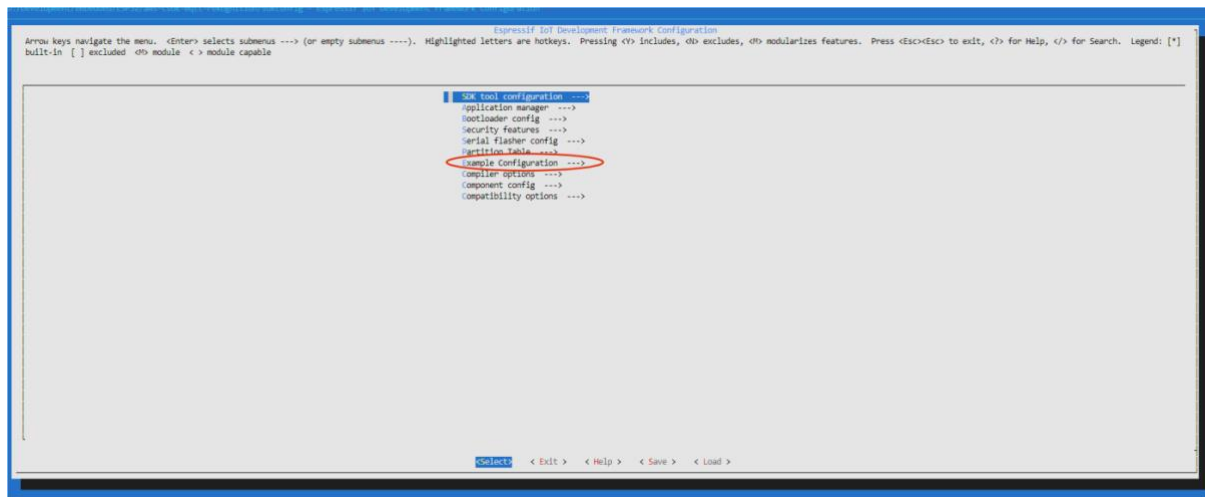
- components:
  - [esp32-camera/](#)
  - [aws-iot-device-sdk-embedded-C/](#)
- include
  - main project include files
- src
  - certs/
  - main project source files
  - CMakeLists.txt
  - Kconfig.projbuild
- platformio.ini
- sdkconfig
- CMakeLists.txt

Extract it into your working directory, open src/certs and substitute private.pem.key and certificate.pem.crt with the certificates that you downloaded during IoT device creation (maintaining the same filenames).

Open PlatformIO IDE (part of Visual Studio Code), make sure Espressif 32 framework is installed. Then inside PlatformIO terminal run the following command:

```
platformio run -t menuconfig
```

Select Example Configuration option and configure your WiFi details as well as IoT client ID (myesp32-cam-example).



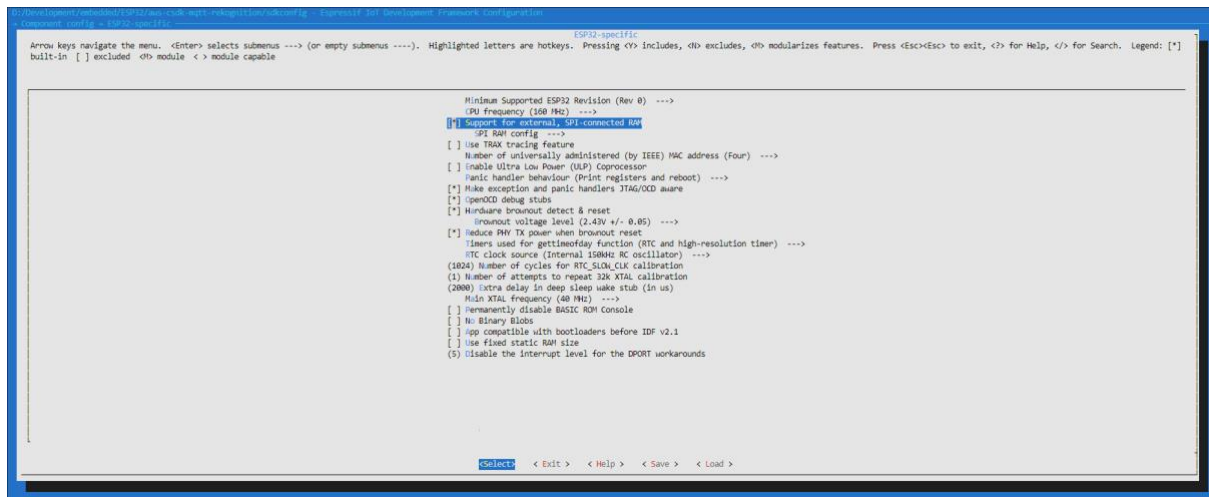
### Hint:

Instead of using the school Wi-Fi (which uses WPA2-Enterprise), you may need to use your laptop or Android phone to create a personal WiFi hotspot (using WPA2-Personal). You need to use 2.4GHz, as ESP32 does not support 5GHz.

There may be issue with an Apple hotspot. See the note from 2021 batch students below: "ESP32 unable to connect to Apple Hotspot. After flashing our code to ESP32, there was a loading bar that showed "....." That continued to go on without any progress. A quick search online revealed that it might be the internet connection, it looks like the issue is a result of Apple introducing Smart Punctuation which changes the standard ASCII quote into a right-facing quote (U+2019, or UTF-8 E2 80 99). One solution is to remap the character. Another is for the user to turn off iOS Smart Punctuation, which results in the standard 0x27 quote character being used by the iOS device. We tried using the hotspot from an android phone instead as a workaround."

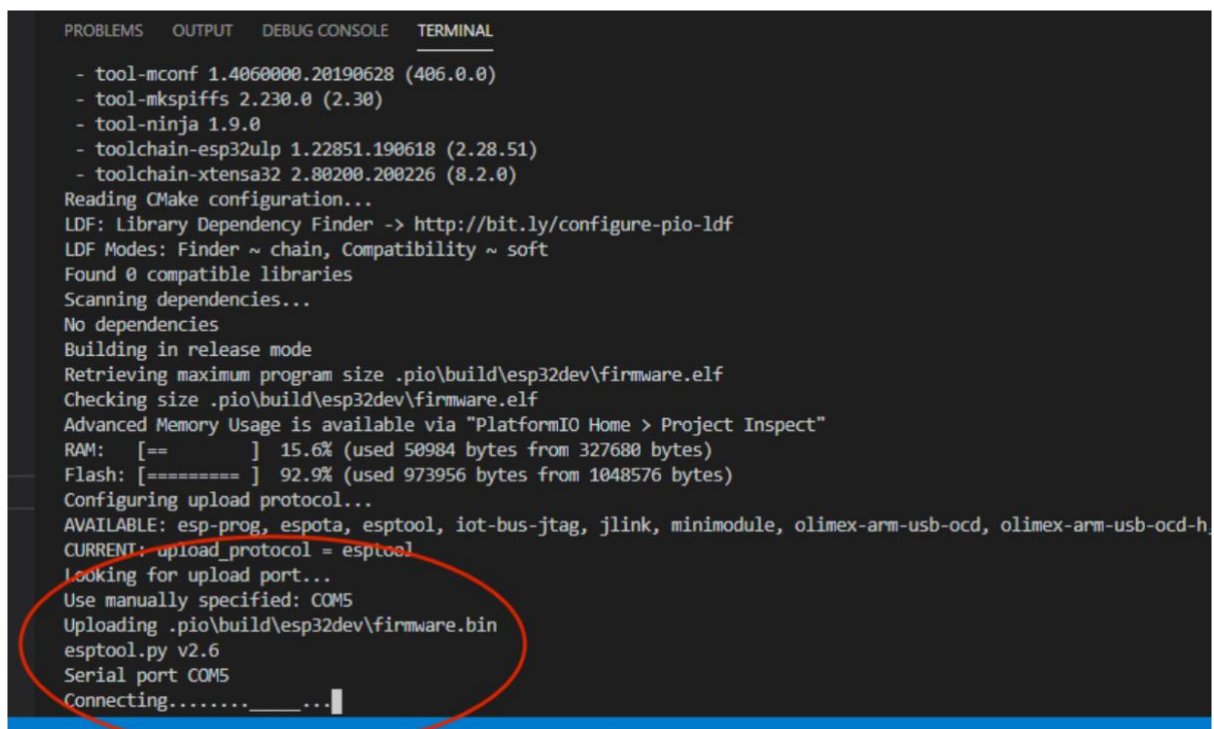
Also, make sure to select "Support for external, SPI-connected RAM" under Component config → ESP32-specific option.





Open sdkconfig file in the root folder and modify CONFIG\_AWS\_IOT\_MQTT\_HOST variable to the one that is under IoT Core – Manage – Things – myesp32-cam-example – Interact (HTTPS).

After completion of these steps, you are ready to build and flash to your device.



After a successful build and flash process, you can point the camera at a subject, press the button and wait for an LED to light up based on the results from Amazon Rekognition.

Now let us test it on a real subject knowing the following:

- Red LED: animal
- Green LED: person
- Yellow LED: everything else



As you can see, my cat was correctly identified as an animal.

To debug any issues, connect a serial monitor to the serial port associated with your device. In addition, you can see logs associated to this solution inside [CloudWatch Logs](#) in the AWS Management Console.

## Clean up

To avoid incurring future charges, delete the IoT Things together with both Lambda functions.

## Conclusion

AWS supports an array of widely available IoT embedded devices to connect your workloads to the cloud. This post showed that with a sub 20 USD IoT board, you could implement an object identification. Lighting up three LEDs is a very basic example, but this can be easily modified to trigger some other physical device that can solve your business problem.

With the use of AWS serverless services, advanced functionality can be added to an IoT device. This is done by offloading work to the cloud to maintain and use devices in a more power- efficient way. The next step is to explore what you can do with [FreeRTOS](#) and the capabilities of [AWS serverless](#).