

Monitoring River Levels Using LoRaWAN



Monitoring River Levels Using LoRaWAN: Implementation Guide

Copyright © 2025 Amazon Web Services, Inc. and/or its affiliates. All rights reserved.

Amazon's trademarks and trade dress may not be used in connection with any product or service that is not Amazon's, in any manner that is likely to cause confusion among customers, or in any manner that disparages or discredits Amazon. All other trademarks not owned by Amazon are the property of their respective owners, who may or may not be affiliated with, connected to, or sponsored by Amazon.

Table of Contents

.....	iv
Abstract and overview	i
Overview	1
Are you Well-Architected?	1
Before you begin	3
Cost	4
Architecture overview	5
LoRaWAN gateway	5
LoRaWAN device	5
AWS IoT Core for LoRaWAN	5
AWS Lambda decoder	6
Walkthrough	7
Configuring a LoRaWAN gateway	7
Simulating a river level sensor using ESP32 and MicroPython	13
Registering the wireless device and integration with AWS IoT Core	15
Creating a decoder Lambda function	20
Security	24
Source code	25
MicroPython application example	25
Lambda decoder function example	29
Conclusion	31
Contributors	32
Additional resources	33
Document history	34
Notices	35
AWS Glossary	36

This whitepaper is for historical reference only. Some content might be outdated and some links might not be available.

Monitoring River Levels Using LoRaWAN

Publication date: **August 10, 2021** ([Document history](#))

Authorities around the world have the important responsibility of monitoring river and sea levels, so both public institutions and private citizens can be better informed of flood risks. This implementation guide demonstrates how [AWS IoT Core for LoRaWAN](#) can be used in conjunction with a qualified gateway device from AWS Advanced Technology Partner Laird Connectivity to install a private [long range wide-area network](#) (LoRaWAN) capable of collecting environmental monitoring data, such as river levels.

Overview

Both public and private sector organizations play a crucial role in managing the risk to life and property from flooding. To illustrate the size of the task faced by such authorities, [Flooding in England: national assessment of flood risk](#), published by the Environment Agency, identified that one in six properties in England is at risk of flooding. Furthermore, it reported that rising sea levels and increasingly severe and frequent rainstorms caused by climate change mean that the risk of flooding will only increase.

As part of a comprehensive approach, authorities commonly undertake monitoring of river and sea levels at a finite number of fixed monitoring stations, providing both immediate and longer-term profiling of risk from rising water levels. To facilitate even greater geographical coverage, low-power wide-area networks (LPWAN) technologies such as LoRaWAN give organizations additional flexibility to deploy low-cost, low-power sensors without depending on existing power or telecoms infrastructure.

This implementation guide demonstrates how [AWS IoT Core for LoRaWAN](#) can be leveraged alongside the [Laird Connectivity Sentrius RG1xx LoRaWAN Gateway](#) to deploy a private LoRaWAN network capable of collecting environmental sensor readings from a fleet of geographically distributed microcontrollers.

Are you Well-Architected?

The [AWS Well-Architected Framework](#) helps you understand the pros and cons of the decisions you make when building systems in the cloud. The six pillars of the Framework allow you to learn architectural best practices for designing and operating reliable, secure, efficient, cost-effective,

and sustainable systems. Using the [AWS Well-Architected Tool](#), available at no charge in the [AWS Management Console](#), you can review your workloads against these best practices by answering a set of questions for each pillar.

In the [IoT Lens](#) and [IoT Lens Checklist](#), we focus on best practices for architecting your IoT applications on AWS.

For more expert guidance and best practices for your cloud architecture—reference architecture deployments, diagrams, and whitepapers—refer to the [AWS Architecture Center](#).

Before you begin

- **Long range (LoRa)** is a wireless radio communication technology which operates in the license-free, sub-gigahertz radio frequency band. Due to the technology's focus on achieving longer range and lower power consumption compared to other wireless connectivity standards such as Bluetooth, Wi-Fi or mobile broadband, LoRa has found widespread use to meet a variety of Internet of Things (IoT) use cases where there is a compelling requirement to implement an LPWAN. In such deployments, the LPWAN is often used to facilitate communication between geographically distributed, low-cost, power-constrained devices such as battery-operated sensor units that are positioned in remote locations with challenges in access.
- **Long range wide-area network (LoRaWAN)** provides the protocols for the upper layers of the LPWAN. It builds on the lower physical foundations provided by LoRa technology, including its hardware, to manage end-to-end communication between devices that participate in the overall network. Additionally, LoRaWAN allows data payloads to wirelessly flow between devices participating in the network and centralized gateways responsible for routing the traffic.
- **[AWS IoT Core for LoRaWAN](#)** is a fully managed feature that removes the undifferentiated heavy-lifting of instantiating and operating a private LoRaWAN network by enabling customers to build a fully serverless, scalable, and secure LoRaWAN-based application that tightly integrates with AWS services, including [AWS IoT Core](#).

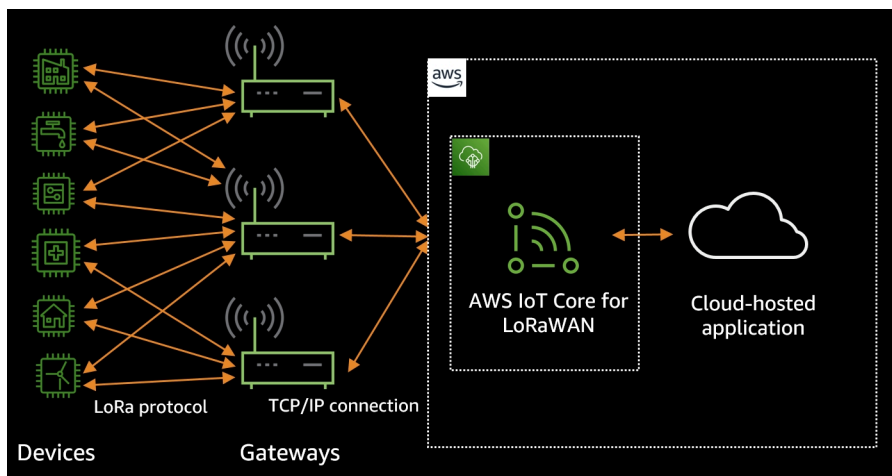


Figure 1 – AWS IoT Core for LoRaWAN overview

Cost

All AWS services included in this implementation guide have a pay-as-you-go pricing model, which scales relative to the demand placed on the application. There are no upfront or monthly commitments.

There are no additional charges for using AWS IoT Core for LoRaWAN, beyond AWS IoT Core charges incurred from messaging. However, if additional AWS IoT Core features are used in conjunction with the deployment, connectivity, [device shadow](#), registry, and [rules engine](#), charges may apply.

The cost of [AWS Lambda](#), used in the solution to decode LoRaWAN payloads, is based on the number of times the function is run, and the duration it runs for.

Architecture overview

LoRaWAN gateway

To facilitate LoRaWAN connectivity, deploy a Laird Connectivity Sentrius RG1xx LoRaWAN gateway, which is a gateway device qualified for use with AWS IoT Core, and provides a range of up to ten miles for connecting devices. This gateway is registered in AWS IoT Core for LoRaWAN, and configured to receive data payloads from the remote IoT device wirelessly.

Once configured and registered, the gateway communicates with AWS IoT Core for LoRaWAN over a fixed internet connection using two distinct protocols: Configuration and Update Service (CUPS) and WebSocket Secure (WSS).

The CUPS protocol allows a supported LoRaWAN gateway to periodically retrieve configuration and software updates from a remote CUPS server. Although optional, its use is highly recommended, as it simplifies the management of LoRaWAN gateways. The LoRaWAN Basics Station software running on the gateway leverages CUPS to securely communicate with the managed CUPS server running in AWS over HTTPS, and retrieve endpoint information and certificates for the data plane.

Thereafter, actual data transfer is facilitated over the data plane using the LoRaWAN Network Server (LNS) protocol based on WebSocket Secure (WSS).

LoRaWAN device

To simulate a low-cost, low-power microcontroller monitoring a designated river level, a Pycom LoPy4 ESP32 development board equipped with a built-in Semtech SX1276 LoRa transceiver is used to upload data to the gateway. You will use an HC-SR04 ultrasonic distance sensor to approximate the distance to the water surface, and to send this measurement as a lean, two-byte payload. This paper provides a [MicroPython application example](#) which illustrates the distance capture and its subsequent transmission as a valid LoRaWAN payload.

AWS IoT Core for LoRaWAN

With a goal of building an end-to-end application, service and wireless device profiles are configured in AWS IoT Core for LoRaWAN, and the microcontroller registered as a wireless device. To facilitate onward connectivity to additional AWS services, a destination accompanied by an AWS IoT rule is configured.

AWS Lambda decoder

Payloads received by LoRaWAN are base64 encoded. As such, you will use a decoder function deployed to AWS Lambda to decode the payload, construct a meaningful JSON payload, and republish this back to AWS IoT Core from where it can be forwarded to AWS services. This paper provides a [Lambda decoder function example](#) to demonstrate this conversion.

The Lambda decoder function allows other applications and devices to subscribe to messages arriving via LoRaWAN through the use of the [MQTT](#) protocol and a designated topic. Depending on the precise use case, the Lambda function could be modified to undertake alternative tasks, such as directly invoking an [AWS SDK API](#) call to forward data to other AWS services, or updating the device shadow.

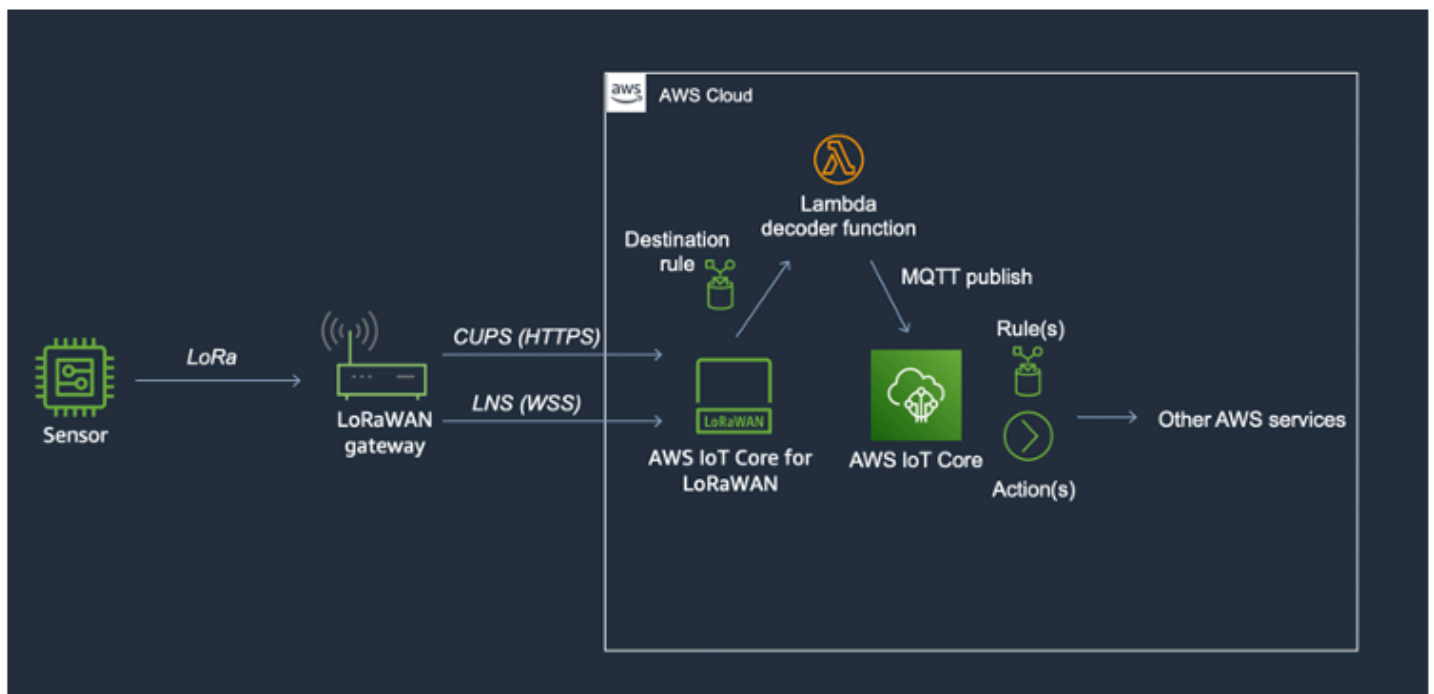


Figure 2 – Solution overview

Walkthrough

Topics

- [Configuring a LoRaWAN gateway](#)
- [Simulating a river level sensor using ESP32 and MicroPython](#)
- [Registering the wireless device and integration with AWS IoT Core](#)
- [Creating a decoder Lambda function](#)

Configuring a LoRaWAN gateway

Laird Connectivity Sentries RG1xx is an 8-channel LoRaWAN gateway with +27 dBm maximum transmit power, with support for multiple wireless and wired interfaces such as LoRaWAN, 802.11a/b/g/n, Bluetooth v4.0 and Ethernet. The gateway has been qualified by AWS, and is [available from the AWS Partner Device Catalog](#). It runs the LoRaWAN Basics Station software, allowing it to use both the CUPS protocol for the management plane, and the LNS protocol for the data plane.



Figure 3 – Laird Connectivity Sentrius RG1xx

Note

This guide uses firmware version **93.8.5.25**. For gateway firmware versions and update procedures, refer to the [Laird Connectivity Sentrius RG1xx User Guide](#).

Initial setup and configuration of the Sentrius RG1xx is not in scope of this document. For the purposes of this guide, it is expected that you have followed the [Quick Start Guide](#) from Laird Connectivity, and that the device is connected to the internet through either 802.11a/b/g/n wireless, or wired Ethernet networking.

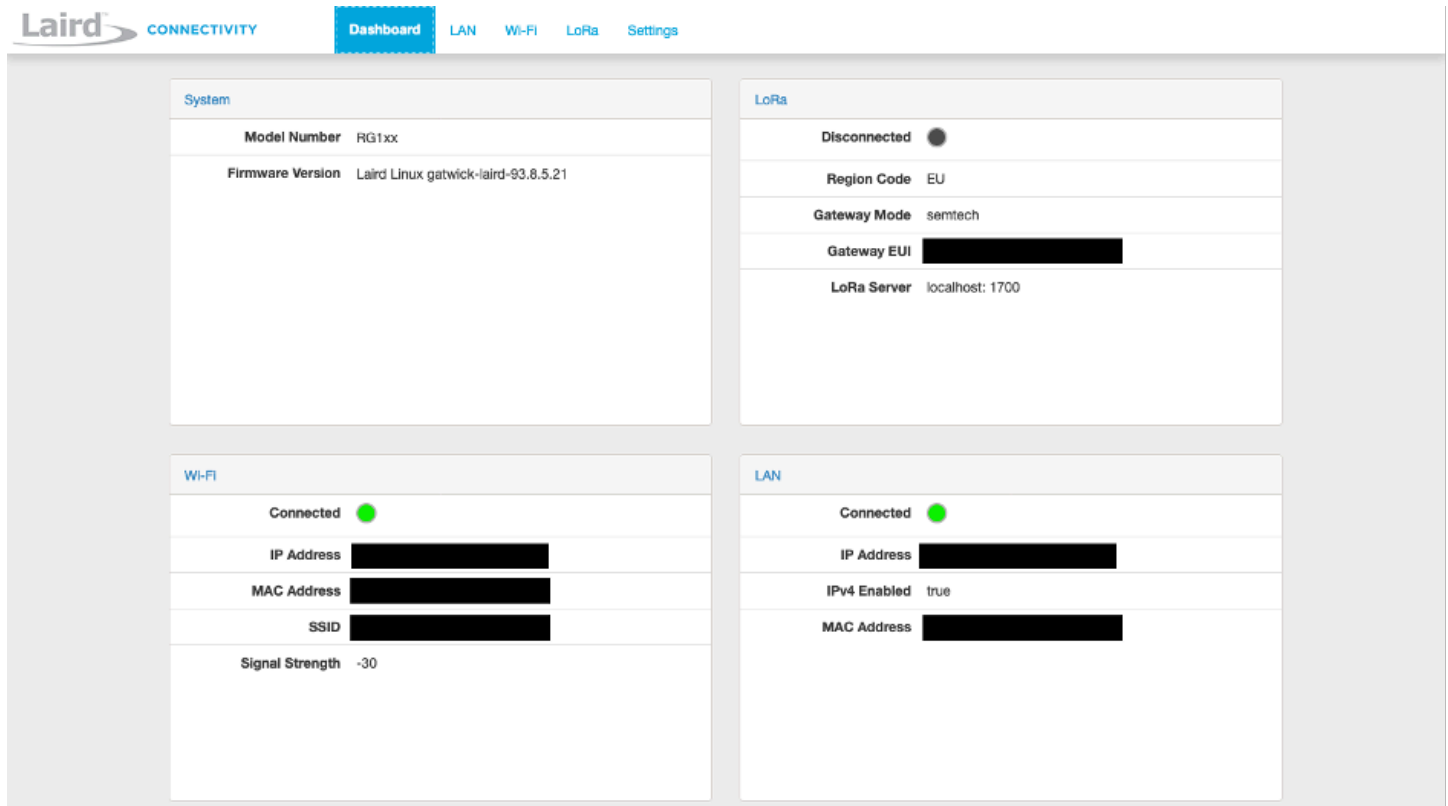


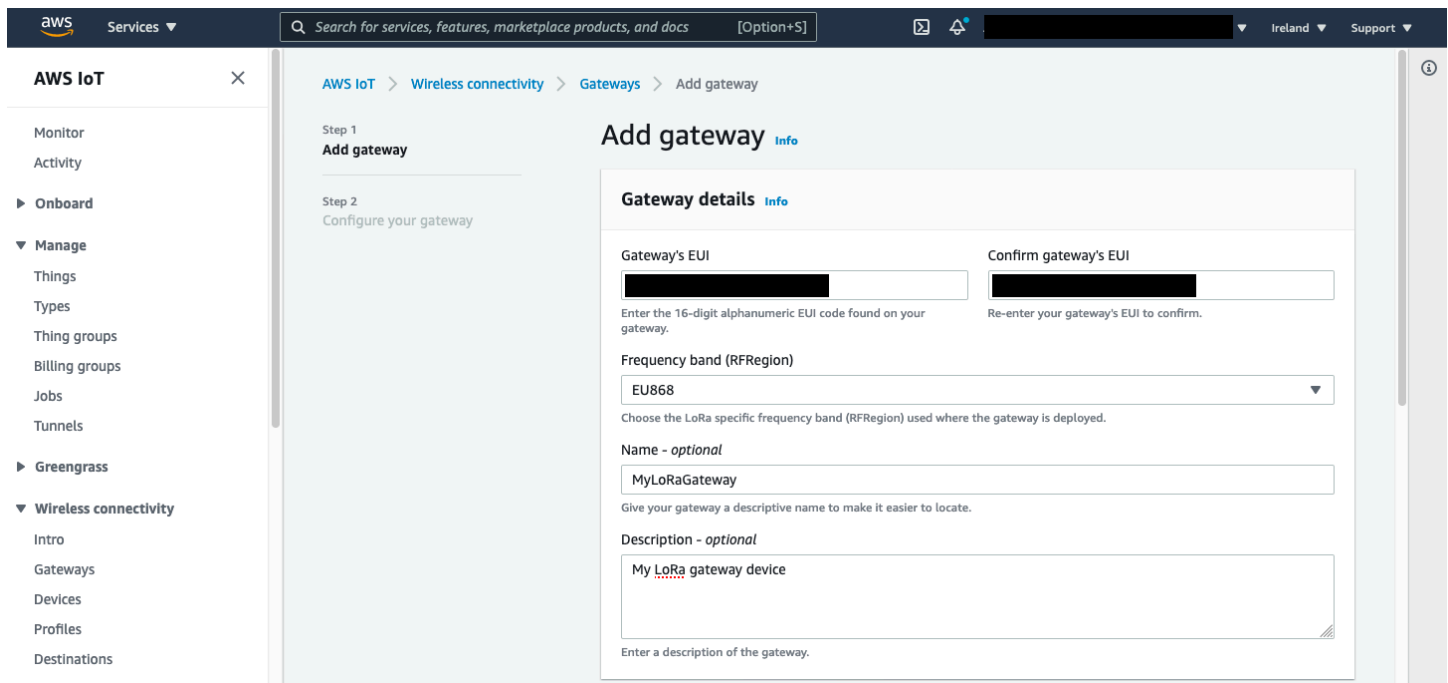
Figure 4 – Viewing the RG1xx dashboard after setup

With the gateway operational, you are now able to register it with the AWS IoT Core console, using the Extended Unique Identifier (EUI) assigned to the hardware.

Note

Before proceeding, follow the steps outlined in the [AWS IoT Developer Guide](#) to configure an IAM role that will allow the Configuration and Update Server (CUPS) to manage gateway credentials.

Once the role exists, you can successfully add the gateway using the console under **Wireless connectivity > Gateways**.



Step 1
Add gateway

Step 2
Configure your gateway

Gateway details

Gateway's EUI
[Redacted]

Confirm gateway's EUI
[Redacted]

Enter the 16-digit alphanumeric EUI code found on your gateway.
Re-enter your gateway's EUI to confirm.

Frequency band (RFRegion)
EU868

Choose the LoRa specific frequency band (RFRegion) used where the gateway is deployed.

Name - optional
MyLoRaGateway

Give your gateway a descriptive name to make it easier to locate.

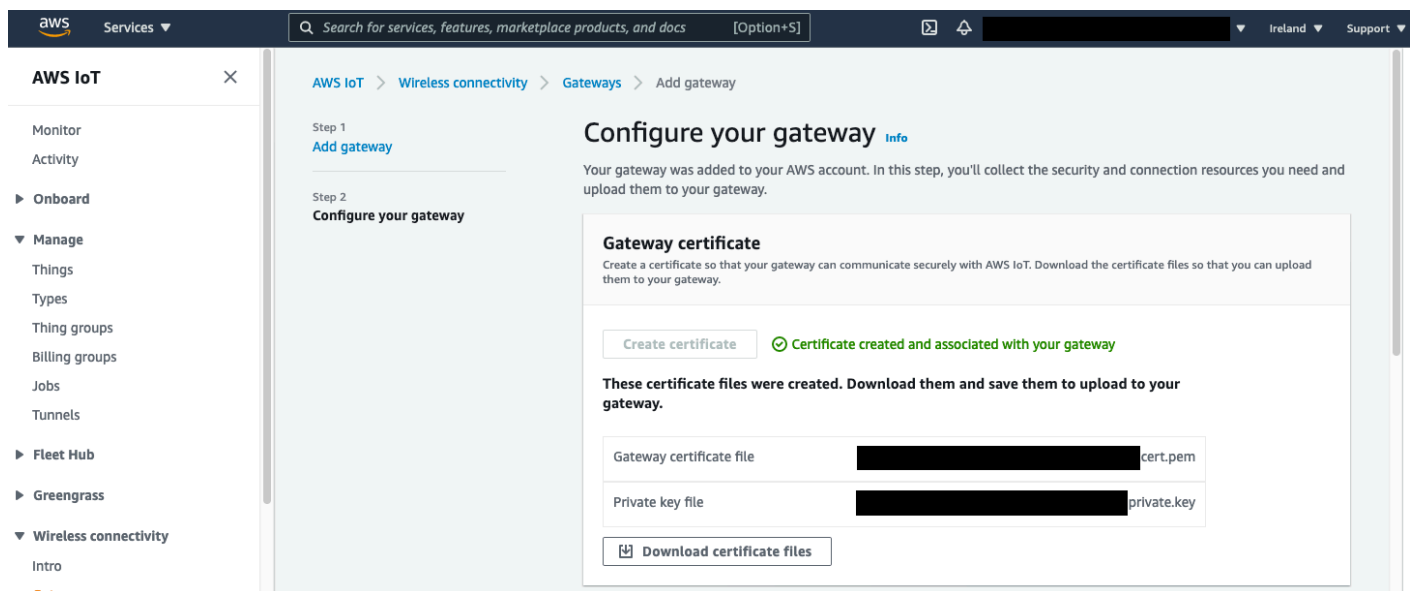
Description - optional
My LoRa gateway device

Enter a description of the gateway.

Figure 5 – Adding a gateway in the AWS IoT console

During the next step in the gateway registration, you will generate and download certificates that allow the gateway to securely communicate with the CUPS server, and to authenticate itself.

1. First, generate and download the *.cert.pem personal certificate and *.private.key personal private key files required to authenticate the gateway with the CUPS server running in AWS.



Step 1
[Add gateway](#)

Step 2
Configure your gateway

Configure your gateway

Your gateway was added to your AWS account. In this step, you'll collect the security and connection resources you need and upload them to your gateway.

Gateway certificate

Create a certificate so that your gateway can communicate securely with AWS IoT. Download the certificate files so that you can upload them to your gateway.

[Create certificate](#) ✔ Certificate created and associated with your gateway

These certificate files were created. Download them and save them to upload to your gateway.

Gateway certificate file [Redacted] cert.pem

Private key file [Redacted] private.key

[Download certificate files](#)

Figure 6 – Generating CUPS certificates

2. Download the certificate of the trusted certificate authority (CA) – cups . trust – and note the assigned CUPS server endpoint for the gateway.

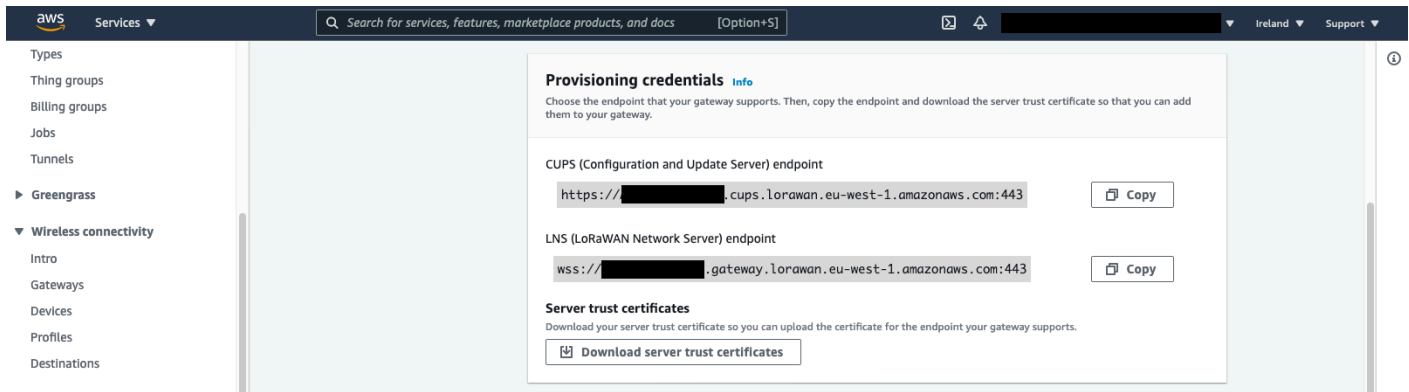


Figure 7 – Noting the CUPS endpoint

3. Back on the RG1xx gateway, choose the Semtech Basics Station mode under **LoRa > Forwarder**, and enter the HTTPS endpoint of the CUPS server endpoint which you noted earlier.

You do not require any LoRaWAN Network Server (LNS) configuration, as the LNS WebSocket Secure endpoint details and the certificates required to secure the data plane are downloaded automatically by the gateway using the CUPS protocol.

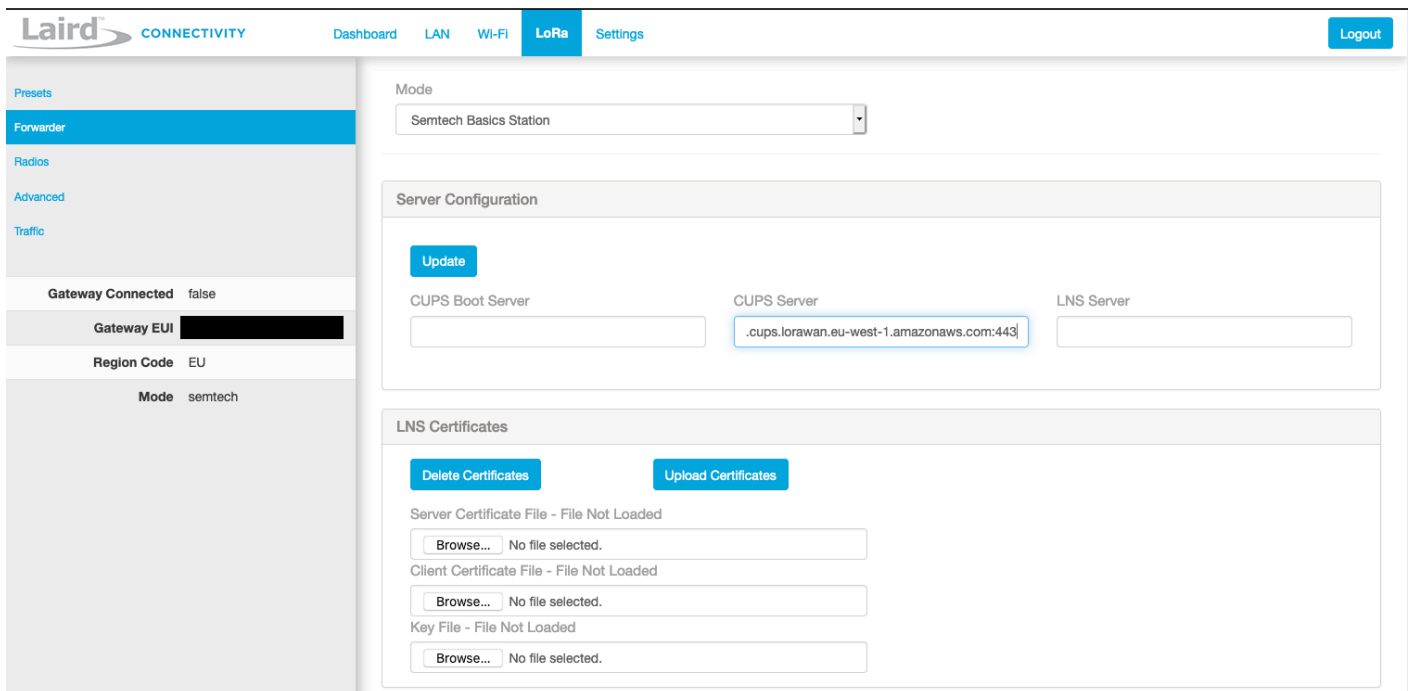


Figure 8 – Configuring the gateway with a CUPS server

4. Upload the previously downloaded *.cert.pem personal certificate and *.private.key personal private key files, together with the cups.trust server trust certificate, to the gateway to secure the CUPS communication.

The screenshot displays the 'LoRa' settings page in the Laird CONNECTIVITY web interface. On the left, a sidebar contains 'Presets' and 'Forwarder' sections. The 'Forwarder' section shows 'Gateway Connected' as false, 'Gateway EUI' as a redacted black box, 'Region Code' as EU, and 'Mode' as sbs. The main content area is titled 'CUPS Certificates' and includes 'Delete Certificates' and 'Upload Certificates' buttons. Below these, there are three rows for certificate uploads: 'Server Certificate File - File Not Loaded' with a 'Browse...' button showing 'cups.trust', 'Client Certificate File - File Not Loaded' with a 'Browse...' button showing a redacted filename and '.cert.pem', and 'Key File - File Not Loaded' with a 'Browse...' button showing a redacted filename and '.private.key'. A second section titled 'CUPS-Boot Certificates' also has 'Delete Certificates' and 'Upload Certificates' buttons, but its upload fields show 'No file selected.' for all three categories.

Figure 9 – Configuring the gateway with CUPS certificates

After the gateway successfully communicates with the CUPS server running in AWS, it can retrieve LNS configurations and establish a secure WebSocket connection. This can be confirmed on the gateway under **Dashboard**.

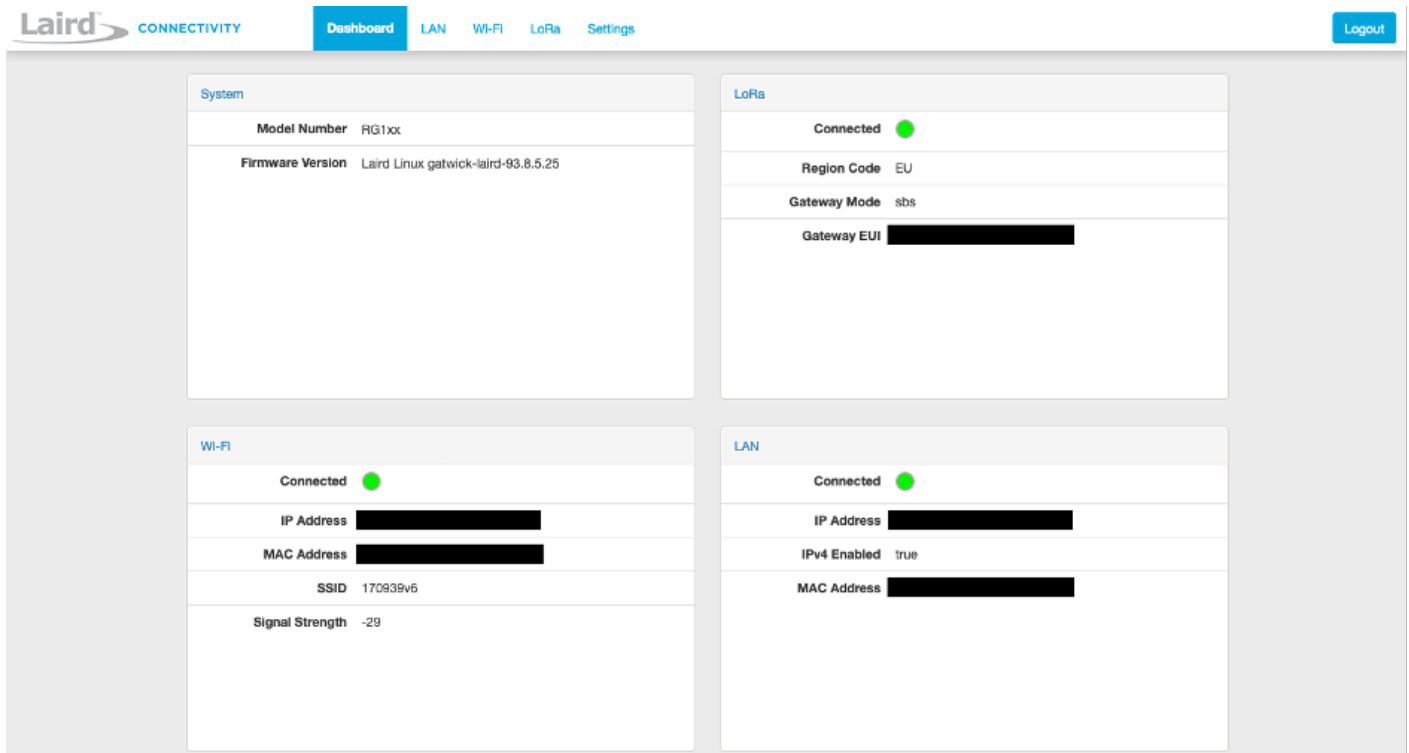


Figure 10 – Confirming gateway connectivity to AWS IoT

In the AWS IoT Core for LoRaWAN console, the gateway now registers recent uplink activity from the gateway under **Wireless connectivity > Gateways**.

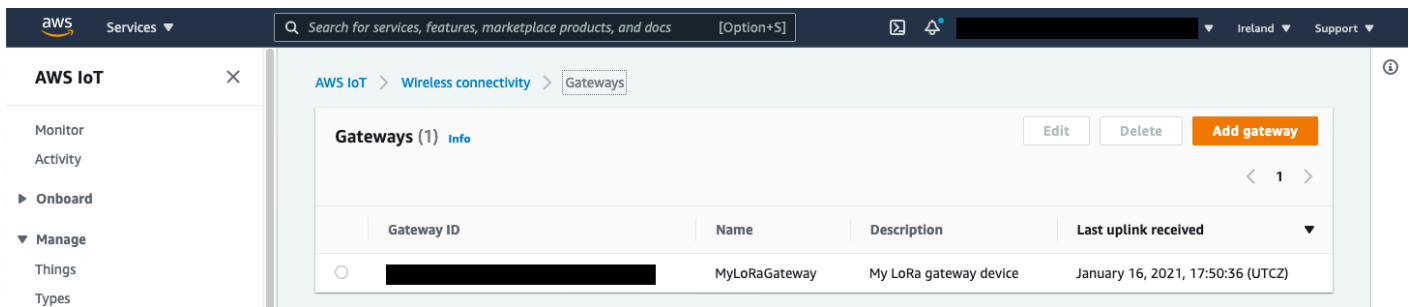


Figure 11 – Validating gateway connectivity in AWS IoT console

You are now ready to send LoRaWAN payloads to the gateway using the microprocessor.

Simulating a river level sensor using ESP32 and MicroPython

Use a Pycom LoPy4 ESP32 development board equipped with a Semtech SX1276 LoRa transceiver to simulate the river level sensor.

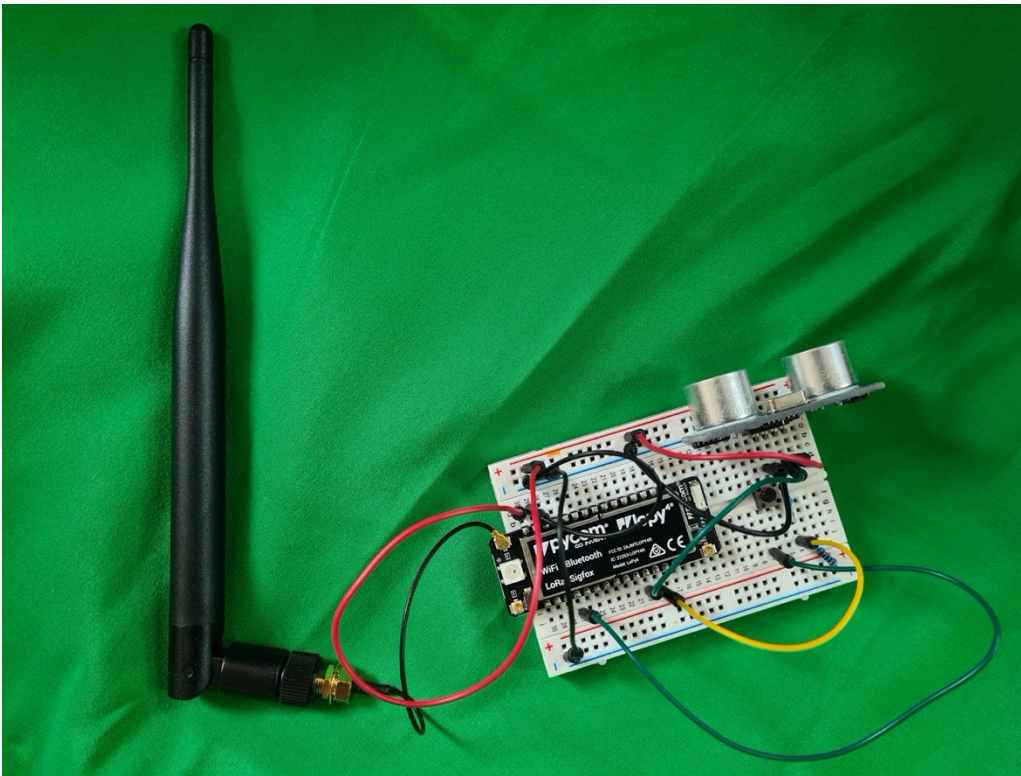


Figure 12 –

The microcontroller is awakened every ten minutes from deep sleep using its real-time clock (RTC), and configured to run a short MicroPython application which retrieves a distance reading to the water surface using the HC-SR04 ultrasonic distance sensor. The captured distance is thereafter reorganized into two bytes – the first byte houses the distance in meters, and the second byte houses the remainder in centimeters.

This data is then broadcast out using LoRaWAN.

This paper provides a [MicroPython application example](#) which demonstrates the application running on generic MicroPython for ESP32 firmware. When running, the console displays the distance recorded by the HC-SR04 ultrasonic distance sensor, LoRaWAN OTAA join status, and the bytes sent.

The following is a running sample MicroPython application:

```
Distance recorded (0.1722414m)
Waiting to join LoRaWAN using OTAA...
Waiting to join LoRaWAN using OTAA...
Waiting to join LoRaWAN using OTAA...
```

```
Joined LoRaWAN
Bytes sent (bytearray(b'\x00\x11'))
Sleeping... (600000ms)
```

Registering the wireless device and integration with AWS IoT Core

AWS IoT Core for LoRaWAN service profiles describes the parameters the device needs to communicate with the LoRaWAN application server.

1. For the purposes of this demonstration, create a service profile with default configuration parameters under **Wireless connectivity > Profiles**.

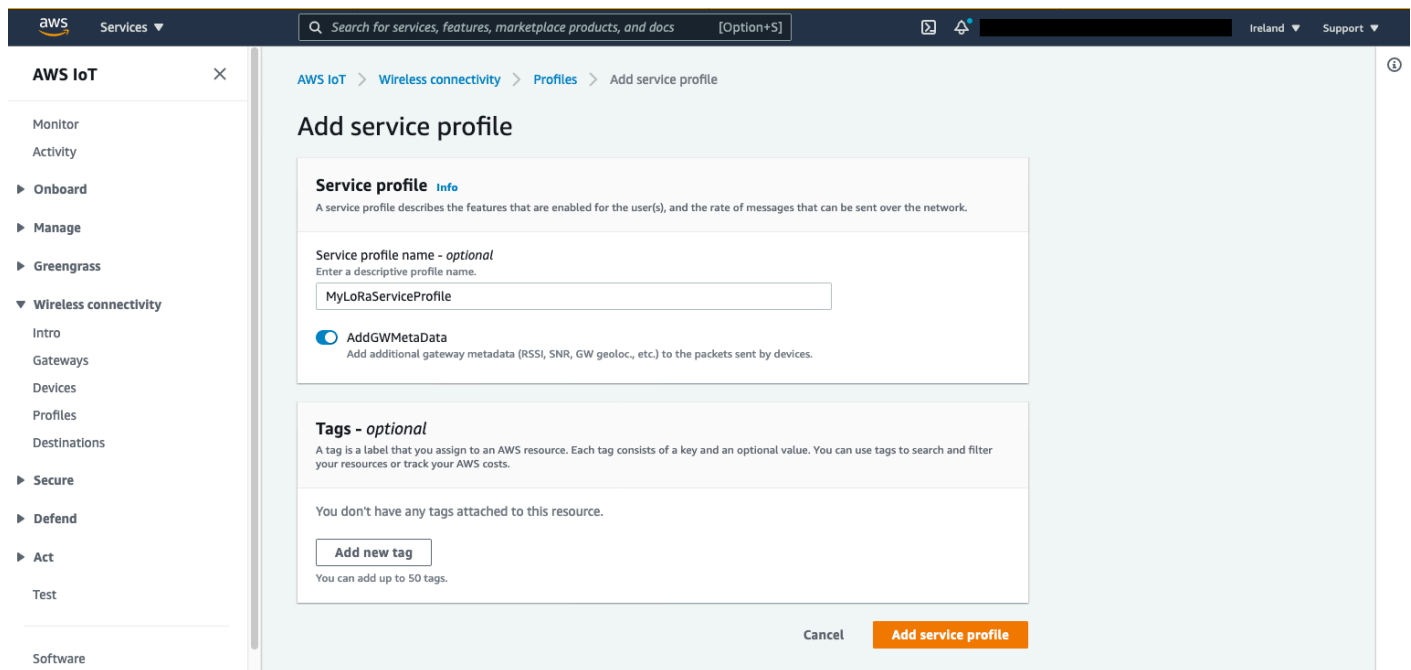


Figure 13 – Adding a service profile

Device profiles contain the parameters the device needs to communicate with the LoRaWAN network server.

2. Select the LoRa frequency band in your Region, the desired media access control (MAC) version, the Regional parameters version, and the maximum equivalent isotropic radiated power (EIRP) value.

The screenshot shows the AWS IoT console interface. On the left is a navigation menu with options like Monitor, Activity, Onboard, Manage, Greengrass, Wireless connectivity, Secure, Defend, Act, Test, Software, Settings, Learn, and Documentation. The main content area is titled 'Add device profile' and contains a form. The form has the following fields and options:

- Device profile name:** A text input field containing 'MyLoRaDeviceProfile'.
- Frequency band (RFRegion):** A dropdown menu showing 'EU868'.
- MAC version:** A dropdown menu showing '1.0.3'.
- Regional parameters version:** A dropdown menu showing 'RP002-1.0.1'.
- MaxERP:** A text input field containing '15'.
- Supports Class B:** A checkbox that is unchecked.
- Supports Class C:** A checkbox that is unchecked.
- Supports Join:** A checkbox that is unchecked.
- Optional settings:** A link at the bottom of the form.

Figure 14 – Adding a device profile

AWS IoT Core for LoRaWAN destinations found in **Wireless connectivity > Destinations** describe the AWS IoT rule that processes a device's data for use by other AWS services.

3. Provision a destination that identifies the AWS IoT rule that will be created later, and also associate this destination with an IAM role.
4. Attach this IAM role to an IAM policy that has the ability to send messages to the rule.

Note

Before proceeding with the creation of the destination, follow the steps described in the AWS IoT Core Developer Guide to [create an IAM role](#) that gives AWS IoT Core for LoRaWAN the permissions necessary to send data to the rule.

The screenshot shows the AWS IoT console interface for adding a destination. The left sidebar contains navigation links for AWS IoT, Monitor, Activity, Onboard, Manage, Greengrass, Wireless connectivity, Secure, Defend, Act, Test, Software, Settings, Learn, and Documentation. The main content area is titled 'Add destination' and includes sections for 'Permissions' (IAM Role: IoTWirelessLoRa) and 'Destination details' (Destination name: MyLoRaDestination, Destination description: My LoRa destination). At the bottom, there are two radio buttons for 'Enter a rule name or topic': 'Enter a rule name' (selected) and 'Publish to AWS IoT Core message broker'. The 'Enter a rule name' option has a text box containing 'MyLoRaRule' and a 'Copy' button.

Figure 15 – Adding a destination

Note

This architecture can be modified to take advantage of [basic ingest](#) to optimize messaging costs. By creating a destination with the option Publish to AWS IoT Core message broker, with a destination MQTT topic of `$aws/rules/rule-name` (where *rule-name* is the name of an AWS IoT Core rule with accompanying actions), received payloads can be dispatched directly to an AWS IoT Core rule, bypassing the MQTT broker. This may be desirable, for example, if the inline Lambda decoder function or MQTT publishing steps can be removed altogether.

5. Register the microcontroller you are using in the console as a wireless device under **Wireless connectivity > Devices**. You will use the OTAA v1.1 wireless device specification.
6. Enter further unique identifiers and security keys related to the wireless device, as specified in the dialog. These parameters must match the actual configurations set on the microcontroller,

as this is the method through which the arriving payload is mapped to your registered wireless device.

Note

Although AWS IoT Core for LoRaWAN supports both Activation by Personalization (ABP) and Over the Air Activation (OTAA), it is recommended that you use OTAA for better security posture.

The screenshot shows the AWS IoT console interface for adding a new device. The left sidebar contains navigation links: Monitor, Activity, Onboard, Manage, Greengrass, Wireless connectivity (selected), Secure, Defend, Act, Test, Software, Settings, and Learn. The main content area is titled 'Add device' and shows a breadcrumb trail: AWS IoT > Wireless connectivity > Wireless devices > Add device. It indicates 'Step 1: Add device' and 'Step 2: Choose destination'. The 'LoRaWAN specification and wireless device configuration' section includes a dropdown for 'Wireless device specification' set to 'OTAA v1.1'. Below this are five pairs of input fields for DevEUI, AppKey, NwkKey, and JoinEUI, each with a 'Confirm' field and a description of the required value (e.g., 'The 16-digit hexadecimal DevEUI value found on your wireless device.').

Figure 16 – Adding a wireless device

- When registering the wireless device, link it to the previously created wireless device and service profiles.

Wireless device name - optional
MyLoRaDevice
A descriptive name to make the wireless device easier to locate.

Wireless device description - optional
My LoRa device
A helpful description of your wireless device.

Thing association [Info](#)

☒ Associate a thing with your wireless device
We'll create a thing in AWS IoT for you and associate it with this device. Things in AWS IoT can make it easier to search for and manage your devices.

Profiles

Wireless device profile
Choose a wireless device profile so your device can pass the correct messages to your gateway.
MyLoRaDeviceProfile

Service profile
Choose a service profile.
MyLoRaServiceProfile

Figure 17 – Associating a wireless device with profiles

8. Assign the device to the destination you created earlier.

Choose destination

Choose destination

Destination name
Destinations route LoRaWAN messages from your wireless device to other AWS services.
MyLoRaDestination

Cancel Previous **Add device**

Figure 18 – Choosing a destination for the wireless device

If the microcontroller is already broadcasting OTAA join requests over LoRaWAN, the newly registered device is now able to join the network, and its status reflects the last uplink data received.

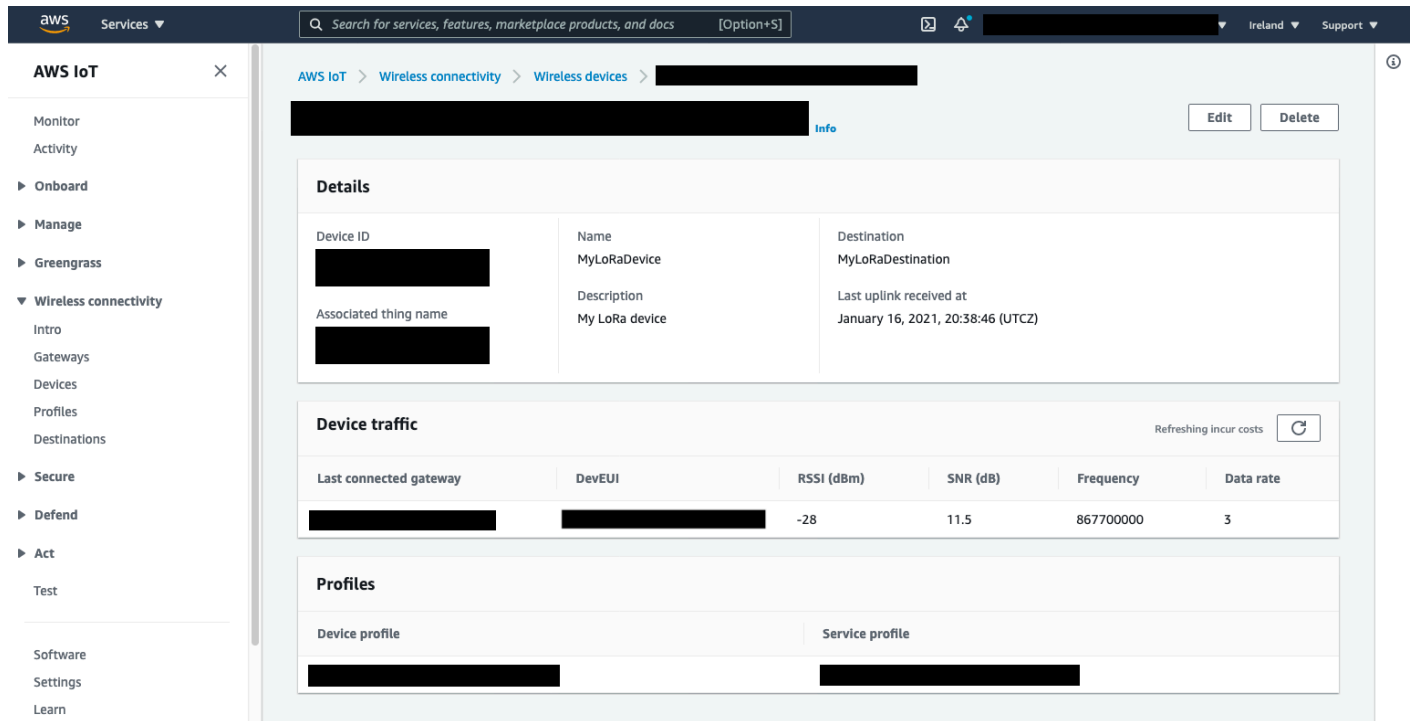


Figure 19 – Confirming uplink from wireless device

You are now ready to complete the final step required for your LoRaWAN payloads to be forwarded to other AWS services.

Creating a decoder Lambda function

The primary purpose of the Lambda decoder function is to intercept the incoming LoRaWAN bytes encoded in base64, and to convert them into a data format that is meaningful to downstream applications. In this solution, the Lambda function performs this conversion, reconstructs the river level reading from each of the respective bytes, and publishes this datapoint along with a timestamp as a JSON document. This data is published to the chosen topic in AWS IoT Core using MQTT, from where any number of rules and actions can be configured to forward the data to the required AWS service.

1. First, create an AWS rule for the destination under **Act > Rules**.

aws Services ▾ Search for services, features, marketplace products, and docs [Option+S] Ireland ▾ Support ▾

AWS IoT ×

Monitor
Activity
► Onboard
► Manage
► Greengrass
► Wireless connectivity
► Secure
► Defend
▼ Act
Rules
Destinations
Test

Software
Settings
Learn
Documentation ↗

AWS IoT > Rules > Create a rule

Create a rule

Create a rule to evaluate messages sent by your things and specify what to do when a message is received (for example, write data to a DynamoDB table or invoke a Lambda function).

Name
MyLoRaRule

Description
My LoRa destination rule

Rule query statement
Indicate the source of the messages you want to process with this rule.

Using SQL version
2016-03-23

Rule query statement
SELECT <Attribute> FROM <Topic Filter> WHERE <Condition>. For example: SELECT temperature FROM 'iot/topic' WHERE temperature > 50. To learn more, see [AWS IoT SQL Reference](#).

```
1 SELECT * FROM 'iot/topic'
```

Figure 20 – Creating an AWS IoT rule

This rule has an action to invoke the Lambda function, called (for the purpose of this paper) `myLoRaDecoderFunction`.

The Lambda function uses the `base64` Python module to handle the conversion of the payload, before repackaging the necessary data as a JSON object. This is then published by the function to an MQTT topic using the `boto3` module.

2. Configure the Lambda function with an [execution role](#) with permissions to publish to the designated MQTT topic. The AWS IoT Core Developer Guide contains [example permissions](#) required to interact with the service.

For example, the following IAM policy, when attached to the execution role of the Lambda function, allows the message to be published to the `myTopic/*` topic.

JSON

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
```

```

        "Effect": "Allow",
        "Action": "logs:CreateLogGroup",
        "Resource": "arn:aws:logs:eu-west-1:111111111111:*"
    },
    {
        "Effect": "Allow",
        "Action": [
            "logs:CreateLogStream",
            "logs:PutLogEvents"
        ],
        "Resource": [
            "arn:aws:logs:eu-west-1:111111111111:log-group:/aws/lambda/myLoRaDecoderFunction:*"
        ]
    },
    {
        "Effect": "Allow",
        "Action": [
            "iot:Publish"
        ],
        "Resource": [
            "arn:aws:iot:eu-west-1:111111111111:topic/myTopic/*"
        ]
    }
]
}

```

Note

AWS account number *111111111111* is used for demonstration purposes only. This value must be replaced with the account number of the actual AWS account in use. It is also necessary to modify the Region specified in this example policy, if the deployment is not in *eu-west-1*.

By monitoring this topic using the AWS IoT console under **Test**, you can see the expected data being published in JSON format.

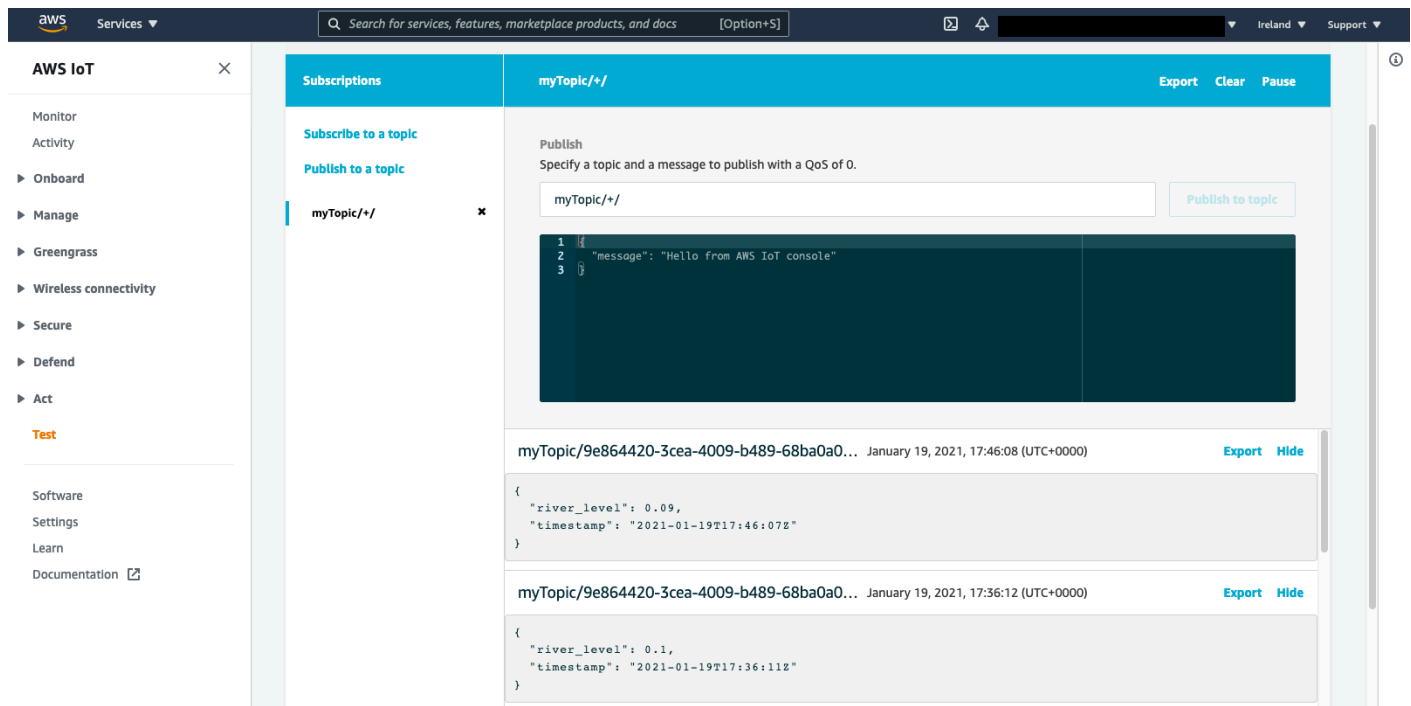


Figure 21 – Confirming the receipt of decoded data

From here, you can configure rules to implement any number of out-of-the-box AWS IoT actions to forward this data to the intended AWS service.

For example, for a river level monitoring system, it might be beneficial for this data to be considered an input to a detector model in AWS IoT Events, so that the measurements can be interpreted and actioned in the context of predefined states. Alternatively, this data could populate a data lake in [Amazon Simple Storage Service](#) (Amazon S3), a time-series database in [Amazon Timestream](#), or a key-value table in [Amazon DynamoDB](#). All of this is possible through built-in integrations via an AWS IoT rule and action.

Security

LoRaWAN devices encrypt their binary messages using [AES128 CTR](#) mode before they are transmitted over the air. Transport Layer Security (TLS) encryption is used further upstream between the LoRaWAN gateway and AWS IoT Core.

Refer to [Data Security with AWS IoT Core for LoRaWAN](#) for details on how data security is addressed between each component.

Source code

MicroPython application example

The Pycom LoPy4 ESP32 development board used for this demonstration has the following physical connectivity, as outlined in this schematics diagram:

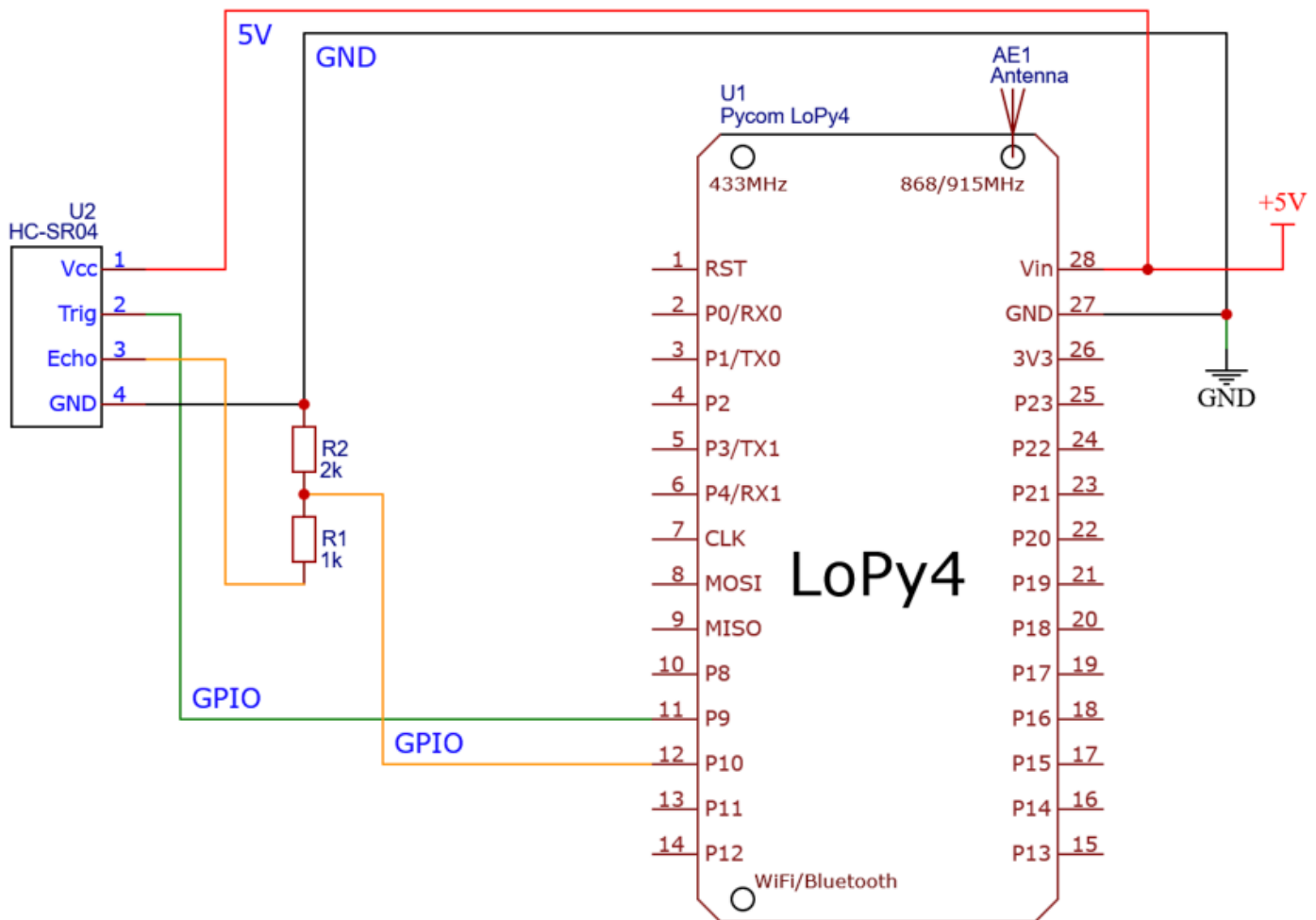


Figure 22 – ESP32 development board schematics

Note

In this example, a voltage divider is used on the output of the HC-SR04's echo pin to convert the 5V to 3.3V.

Note that this diagram does not show receive/transmit (RX/TX) serial connections required to program the board, nor the recommended button between P2 and ground (GND) to

place the device into bootloader mode. Refer to the [Pycom LoPy4 Product Info, Datasheets](#) webpage for additional information on the connections required to program the device.

Table 1 – ESP32 development board physical connectivity

Module	ESP32 (LoPy4) pin	Description
HC-SR04	P9	Trigger
HC-SR04	P10	Echo
5V	Vin	5V
GND	GND	Ground

```
"""
Sample MicroPython application for the Pycom LoPy4 development board. Demonstrates
unconfirmed data uplink of HC-SR04 ultrasonic distance sensor readings using
LoRaWAN. Uses LoRaWAN OTAA support of Pycom's network driver. Spends time in
deep sleep between readings to conserve power.
"""
```

```
# Required MicroPython libraries
# pylint: disable=E0401
import socket
import utime
import ubinascii
from machine import Pin, deepsleep
from network import LoRa

# HC-SR04 ultrasonic distance sensor configurations
HCSR04_TRIGGER_PIN = "P9"
HCSR04_ECHO_PIN = "P10"
HCSR04_ECHO_TIMEOUT_MS = const(50)          # pylint: disable=E0602

# LoRaWAN OTAA connection details. Replace with own settings.
LORAWAN_APP_EUI = ubinascii.unhexlify("REPLACE")
LORAWAN_APP_KEY = ubinascii.unhexlify("REPLACE")
LORAWAN_OTAA_TIMEOUT_MS = const(30000)      # pylint: disable=E0602
```

```

# Additional program configurations
PROGRAM_LOOP_MS = const(600000)          # pylint: disable=E0602
PROGRAM_WAIT_MS = const(3000)             # pylint: disable=E0602

# pylint: disable=R0903
class HCSR04():
    """ Driver for HC-SR04 ultrasonic distance sensor """

    # HC-SR04 fixed parameters
    HCSR04_US_TO_CM_CONST = const(58)      # pylint: disable=E0602
    HCSR04_MAX_RANGE_CM = const(400)       # pylint: disable=E0602

    def __init__(self, trigger_pin, echo_pin, echo_timeout_ms):
        """ Initialises HC-SR04 ultrasonic distance sensor pins """
        self.trigger_pin = Pin(trigger_pin, mode=Pin.OUT, pull=None)
        self.echo_pin = Pin(echo_pin, mode=Pin.IN, pull=None)
        self.echo_timeout_ms = echo_timeout_ms
        self.distance_cm = None

    def get_distance_cm(self):
        """ Retrieves distance to nearest surface in m (decimal).
        Raises exception if range is unsupported, or if echo response
        times out. This is a blocking method. """
        echo_detected = False
        self.trigger_pin(True)
        utime.sleep_us(10)
        self.trigger_pin(False)
        echo_timeout_start_ms = utime.ticks_ms()
        while (utime.ticks_ms() - echo_timeout_start_ms) < self.echo_timeout_ms:
            if self.echo_pin():
                # If high is detected on echo pin, start echo timer
                echo_detected = True
                echo_timer_start_us = utime.ticks_us()
                break
        if echo_detected:
            while self.echo_pin():
                pass
            # If echo pin goes low, stop echo timer
            duration_us = utime.ticks_us() - echo_timer_start_us
            self.distance_cm = duration_us / self.HCSR04_US_TO_CM_CONST
            if self.distance_cm > self.HCSR04_MAX_RANGE_CM:
                raise OSError(
                    "Unsupported HC-SR04 range (>" +
                    str(self.HCSR04_MAX_RANGE_CM) +

```

```

        "cm)"
    )
else:
    # If no error signal is detected, time out
    raise OSError(
        "Failed to detect echo signal (>" +
        str(self.echo_timeout_ms) +
        "ms)"
    )
return self.distance_cm

def main():
    """ Runs the ultrasonic distance sensor check and dispatches results as
    unconfirmed data upload LoRaWAN payload. Payload is 2 bytes (first byte is
    distance in m, second byte is remaining distance in cm). Programs enters
    deep sleep in between checks. """
    start_time = utime.ticks_ms()
    data = bytearray(2)
    sensor = HCSR04(
        trigger_pin=HCSR04_TRIGGER_PIN,
        echo_pin=HCSR04_ECHO_PIN,
        echo_timeout_ms=HCSR04_ECHO_TIMEOUT_MS
    )
    try:
        distance_m = sensor.get_distance_cm() / 100
        print(
            "Distance recorded (" +
            str(distance_m) +
            "m)"
        )
    except OSError as exception:
        print(
            "Sensor fault (" +
            str(exception) +
            ")"
        )
    else:
        # First byte of output is distance in m
        data[0] = int(str(distance_m).split(".")[0])
        # Second byte of output is remainder of the distance in cm
        data[1] = int(str(distance_m).split(".")[1][:2])
        # LoRaWAN OTAA data upload. Region is EU868 (change as required).
        lora = LoRa(mode=LoRa.LORAWAN, region=LoRa.EU868)

```

```

    lora.join(activation=LoRa.OTAA, auth=(LORAWAN_APP_EUI, LORAWAN_APP_KEY),
timeout=0)
    otaa_timeout_start_ms = utime.ticks_ms()
    while (utime.ticks_ms() - otaa_timeout_start_ms) < LORAWAN_OTAA_TIMEOUT_MS:
        if lora.has_joined():
            print("Joined LoRaWAN")
            break
        print("Waiting to join LoRaWAN using OTAA...")
        utime.sleep(2.5)
    if lora.has_joined():
        lora_socket = socket.socket(socket.AF_LORA, socket.SOCK_RAW)      # pylint:
disable=E1101
        lora_socket.setsockopt(socket.SOL_LORA, socket.SO_DR, 5)          # pylint:
disable=E1101
        lora_socket.setblocking(True)
        lora_socket.send(data)
        print(
            "Bytes sent (" +
            str(data) +
            ")"
        )
        lora_socket.setblocking(False)
    else:
        print("Failed to join LoRaWAN using OTAA")
    finally:
        utime.sleep_ms(PROGRAM_WAIT_MS)
        print(
            "Sleeping... (" +
            str(PROGRAM_LOOP_MS) +
            "ms)"
        )
        deepsleep(PROGRAM_LOOP_MS - (utime.ticks_ms() - start_time))

if __name__ == "__main__":
    main()

```

Lambda decoder function example

```

import json
import base64
import boto3

```

```
import boto3

client = boto3.client('iot-data')
mqtt_topic = 'myTopic/'

def lambda_handler(event, context):
    """ Decode LoRa payload and republish back to AWS IoT as a transformed event """

    river_level_bytes = base64.b64decode(event['PayloadData'])
    # First byte of payload is meters, second byte centimeters
    river_level = river_level_bytes[0] + (river_level_bytes[1] / 100)
    event_transformed = {
        'river_level': river_level,
        'timestamp': event['WirelessMetadata']['LoRaWAN']['Timestamp']
    }
    try:
        response = client.publish(
            topic=mqtt_topic+event['WirelessDeviceId']+'/',
            payload=json.dumps(event_transformed)
        )
    except boto3.exceptions.ClientError as error:
        print('Operataion failed! ' + str(error))
    else:
        print('Event successfully transformed and republished!')
```

Conclusion

Increased urgency to minimize risk to lives and property from extreme weather events such as flooding requires organizations around the world to adopt innovative solutions to undertake extensive monitoring of the environment around us. LoRa and LoRaWAN provide purpose-built, tried-and-tested radio communication technology to connect a fleet of geographically dispersed sensors, without reliance on pre-existing telecoms or power infrastructure.

With AWS IoT Core for LoRaWAN, it is now possible for organizations to securely gather meaningful data from the field cost-effectively, and leverage it to gain better insights using AWS services to drive decision making.

Contributors

Contributors to this document include:

- Alan Peaty, Partner Solutions Architect, Amazon Web Services
- Ali Benfattoum, Senior Specialty Solutions Architect (IoT), Amazon Web Services

Additional resources

For additional information, see:

- [AWS IoT Core for LoRaWAN Developer Guide](#)
- [AWS IoT Core Developer Guide](#)
- [Connecting Gateways to AWS IoT Core for LoRaWAN](#)
- [Pycom LoPy4 Product Info, Datasheets](#)

Document history

To be notified about updates to this whitepaper, subscribe to the RSS feed.

Change	Description	Date
Initial publication	Whitepaper first published.	August 10, 2021

Note

To subscribe to RSS updates, you must have an RSS plug-in enabled for the browser that you are using.

Notices

Customers are responsible for making their own independent assessment of the information in this document. This document: (a) is for informational purposes only, (b) represents current AWS product offerings and practices, which are subject to change without notice, and (c) does not create any commitments or assurances from AWS and its affiliates, suppliers or licensors. AWS products or services are provided “as is” without warranties, representations, or conditions of any kind, whether express or implied. The responsibilities and liabilities of AWS to its customers are controlled by AWS agreements, and this document is not part of, nor does it modify, any agreement between AWS and its customers.

© 2021 Amazon Web Services, Inc. or its affiliates. All rights reserved.

AWS Glossary

For the latest AWS terminology, see the [AWS glossary](#) in the *AWS Glossary Reference*.