

TTN vs ChirpStack: A LoRaWAN Integration Journey

BME280 Sensor Use Case, Network Feasibility Analysis, and Integration for the Gladiolen Project



Uche Nwogbo

Thomas More University
Elektronika Intern, Spring 2025
4th April 2025

Table of Content

1. Introduction to LoRaWAN	2
2. Step-by-Step Integration Guide Using BME280 Sensors	2
2.1. Introduction to BME280 Sensor Test Case	2
2.2. Hardware and Software Requirements	3
2.2.1. Hardware	3
2.2.2. Software	3
2.3. Preparing the Sensor and Microcontroller	3
2.3.1. Enabling Interfaces on Raspberry Pi	4
2.3.2. Installing CircuitPython Environment and Libraries	5
2.3.3. Initial Testing and Verification of Sensors	5
2.4. Writing the Python Script	6
2.4.1. Importing Required Libraries	6
2.4.2. Sensor Protocols Initializations	6
2.4.3. LoRa Module Initialization	7
2.4.4. LoRaWAN ABP Keys and Network Setup	7
2.4.5. Data Encoding	8
2.4.6. Reading Sensor Data	8
2.5. Setting up and Connecting to TTN	9
2.5.1. Creating Application and End Device	9
2.5.2. Payload Formatter	10
2.5.3. Running the Python Script	11
2.6. Setting up and Connecting to ChirpStack	11
2.6.1. Configuring Gateway for ChirpStack	12
2.6.2. Open ChirpStack Console	13
2.6.2. Creating Application and End Device	13
2.6.3. Starting the Python Script	15
2.7. Results	16
2.7.1. TTN	16
2.7.2. ChirpStack	16
2.8. Common Issues and Troubleshooting Tips	17
3. Feasibility Study and Solution Selection for the Gladiolen Project	18
3.1. Introduction to the Gladiolen Project	18
3.2. Defining Requirements for the Gladiolen Project	19
3.3. Feasibility Analysis of TTN Approach	19
3.3.1. Advantages	19
3.3.2. Disadvantages	20
3.4. Feasibility Analysis of ChirpStack Approach	21
3.4.1. Advantages	21
3.4.2. Disadvantages	21
3.5. Comparison Summary: TTN vs. ChirpStack	22
3.6. Recommendations for Implementing the Gladiolen Project	23
4. Conclusion	24
5. References	26

1. Introduction to LoRaWAN

LoRa (Long Range) is a wireless modulation technique designed for low-power, long-range communication, making it ideal for Internet of Things (IoT) applications such as environmental monitoring, smart agriculture, and asset tracking. Built on top of LoRa, LoRaWAN (Long Range Wide Area Network) is an open standard network protocol that defines how devices communicate with gateways and network servers over long distances while optimizing battery life and data security [1]. LoRaWAN networks operate in unlicensed radio frequencies, enabling affordable and scalable deployment. LoRaWAN is a Media Access Control (MAC) layer protocol built on top of LoRa modulation. It is a software layer that defines how devices use the LoRa hardware, for example when they transmit, and the format of messages[2]. Two common types of LoRaWAN network implementations are The Things Network (TTN) and ChirpStack. TTN is a global, community-driven LoRaWAN network that provides free access to a shared infrastructure, making it accessible for prototyping and small-scale projects [1]. In contrast, ChirpStack is an open-source, private LoRaWAN network server stack that gives users full control over their infrastructure, offering greater flexibility, data privacy, and scalability, especially for enterprise or research-focused deployments [3]. All of these LoRaWAN networks are made up of certain basic components, including:

1. **End Devices:** These are sensors or actuators that send uplink messages (payloads) to gateways and receive downlink messages from gateways when required.
2. **Gateways:** Acting as intermediaries, gateways relay data between end devices and the Network Server. They receive uplink messages from end devices forward them to the Network Server, and transmit downlink messages from the server back to the devices.
3. **Network Server:** Also known as the LoRaWAN Network Server (LNS), this software manages the connectivity, authentication, and monitoring of devices and gateways. It ensures proper routing and handling of LoRaWAN messages [4].
4. **Application Server:** This server handles data encryption and decryption, processes uplink and downlink payloads and manages integrations such as MQTT, HTTP webhooks, AWS SNS, and others for forwarding data to user applications.

2. Step-by-Step Integration Guide Using BME280 Sensors

2.1. Introduction to BME280 Sensor Test Case

The BME280 sensor test case was chosen to demonstrate a simple yet practical example of transmitting multiple sensor readings — including temperature, pressure, and humidity — over LoRaWAN networks. Its compact size, low power consumption, and ability to provide three types of environmental data make it an ideal candidate for testing LoRaWAN integration. By using the BME280 sensor as a reference, this test case illustrates how real-world sensor data can be encoded, transmitted, and received through both The Things Network (TTN) and ChirpStack, providing a basis for comparing their setup complexity, data handling, and performance in a typical IoT deployment.

2.2. Hardware and Software Requirements

This section outlines the essential hardware components and software tools needed to set up the BME280 sensor and connect it to both TTN and ChirpStack networks.

2.2.1. Hardware

1. **Raspberry Pi 3 MCU:** Used as the main processing unit to collect sensor data and manage LoRa communication.
2. **BME280 Sensor:** Environmental sensor for measuring temperature, humidity, and pressure.
3. **RFM95W LoRa Module:** LoRa transceiver module for sending data over LoRaWAN.
4. **MCP3008:** Used to interface analog sensors with the Raspberry Pi's digital GPIO.
5. **LoRa Gateway:** Acts as the communication bridge between LoRa nodes (end devices) and the network server (TTN/ChirpStack).
6. **SD Card:** Storage for Raspberry Pi OS and project files.
7. **Breadboard and Jumper Wires:** For easy prototyping and wiring connections between components.

2.2.2. Software

1. **Raspberry Pi OS:** Operating system for running the Raspberry Pi.
2. **Thonny:** Integrated Development Environment (IDE) for writing, editing, and managing code.
3. **Git Bash:** Command-line tool for version control and accessing Git repositories.
4. **Python 3.x:** A programming language for interfacing sensors, handling data, and managing LoRa communications.
5. **ChirpStack LoRaWAN Stack (for private network):** LoRaWAN network server for managing private LoRaWAN infrastructure.
6. **The Things Network (TTN) Console (for public network):** Platform for managing LoRaWAN devices and applications in TTN.
7. **LoRaWAN Python libraries:** Libraries to handle LoRa communication protocol.

2.3. Preparing the Sensor and Microcontroller

Figure 1 shows the wiring schematic for this setup, including the Raspberry Pi 3, BME280 sensor (for temperature, humidity, and pressure), RFM95W LoRa module, and MCP3008 ADC (for additional analog sensor integration, such as an LM35 temperature sensor).

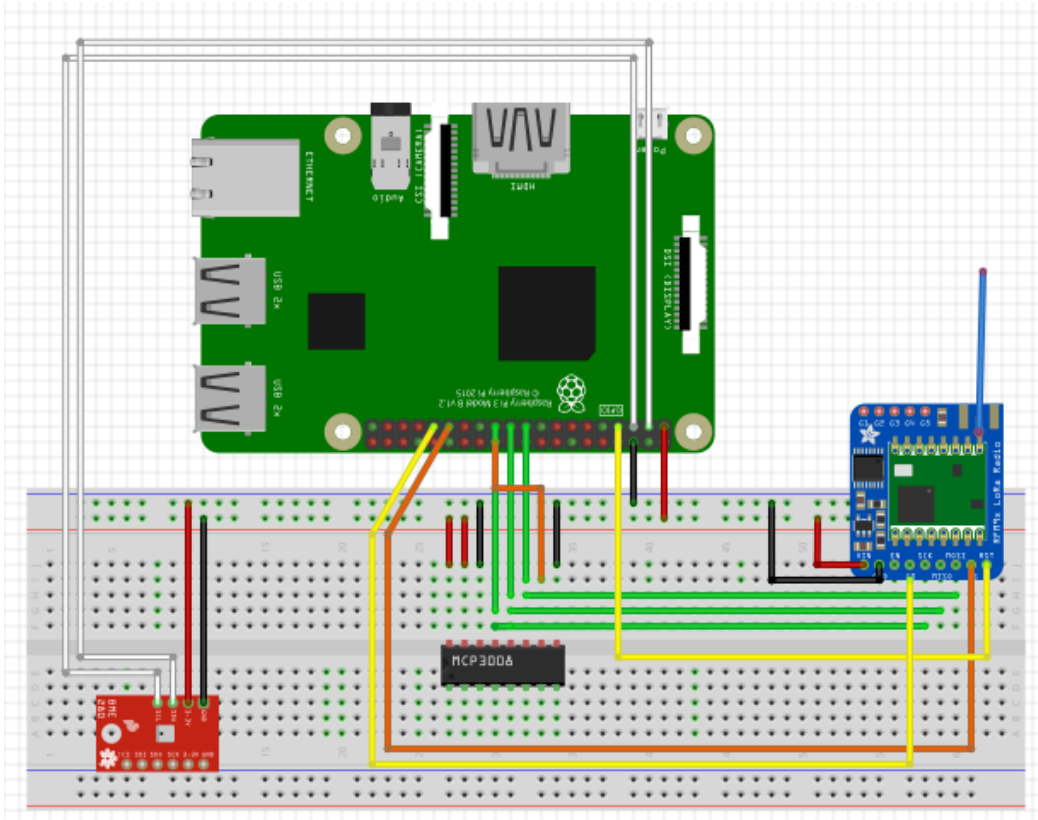


Figure 1: Connection schematic of Raspberry Pi, BME280, MCP3008, and RFM95 LoRa module

2.3.1. Enabling Interfaces on Raspberry Pi

This [first step](#) is to download the Raspberry Pi OS Imager onto the SD Card if not done already. After doing this, we can use a terminal emulator like PuTTY to simulate our Raspberry Pi OS on the PC. After typing the correct IP address of the Raspberry Pi device, click “open” and the terminal opens up. Enter your username and password chosen during the OS configuration step.

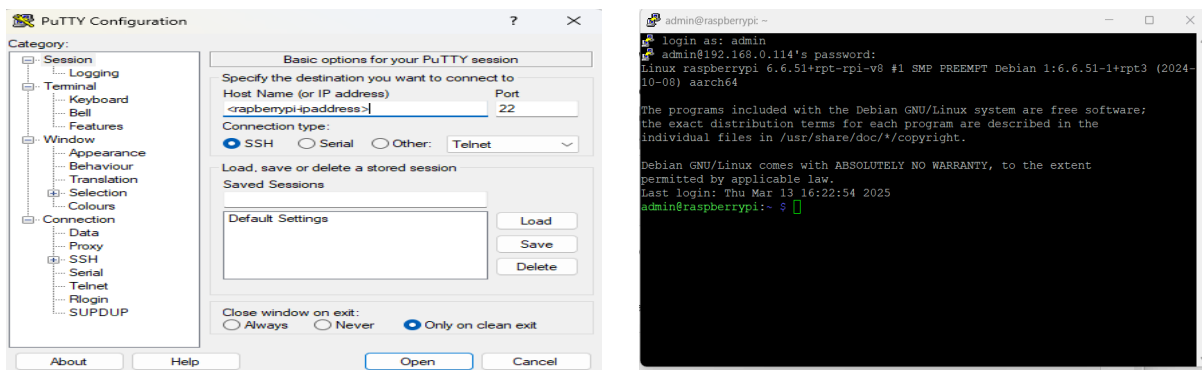


Figure 2: PyTTY configuration using SSH for Raspberry Pi Console

Before connecting and running the sensor modules, enable SPI and I2C on the Raspberry Pi to allow communication with the RFM95W and BME280 sensors:

☒ `sudo raspi-config`

Go to Interfacing Options, Enable SPI and I2C. Then reboot to apply changes:

☒ `sudo reboot`

To confirm that SPI and I2C are enabled, run:

☒ `ls /dev/spi*`

☒ `ls /dev/i2c*`

2.3.2. Installing CircuitPython Environment and Libraries

Following [Adafruit's CircuitPython on Raspberry Pi guide](#), the following commands were used to set up the environment needed for sensor communication.

Update system and install pip3:

☒ `sudo apt-get update`

☒ `sudo apt-get upgrade`

☒ `sudo apt-get install python3-pip`

Install and upgrade essential Python tools:

☒ `sudo pip3 install --upgrade setuptools`

Install Adafruit Blinka (CircuitPython compatibility layer for Raspberry Pi):

☒ `pip3 install adafruit-blinka`

Install TinyLoRa for RFM95W module communication:

☒ `sudo pip3 install adafruit-circuitpython-tinylora==2.2.6`

Install BME280 library for environmental data reading:

☒ `pip3 install adafruit-circuitpython-bme280`

2.3.3. Initial Testing and Verification of Sensors

Run the following to verify that the BME280 sensor is detected: Expected output should display the sensor address (commonly `0x76` or `0x77`).

☒ `i2cdetect -y 1`

Testing SPI Devices: Expected output → `/dev/spidev0.0 /dev/spidev0.1`

☒ `ls /dev/spi*`

After completing this section, the Raspberry Pi is fully connected to the BME280 sensor and RFM95W LoRa module, with the necessary communication interfaces (SPI and I2C) enabled. CircuitPython libraries for both sensors and LoRa communication are installed and ready for use. In the next section, we will focus on configuring TTN and ChirpStack to receive sensor data over LoRaWAN.

2.4. Writing the Python Script

Both TTN and ChirpStack python scripts are designed to read data from a BME280 environmental sensor (temperature, humidity, pressure) and transmit this data over a LoRaWAN network using an RFM95W LoRa module. The main difference between them is the network server target — The Things Network (TTN) or ChirpStack. Although they share a similar structure, each script contains unique network credentials tailored for the specific LoRaWAN network they communicate with.

2.4.1. Importing Required Libraries

```
# --- Imports ---
import time
import board
import busio
import digitalio
from adafruit_bme280.basic import Adafruit_BME280_I2C
from adafruit_tinyloara.adafruit_tinyloara import TTN, TinyLoRa
```

Figure 3: Import section from the Python script

The import section includes all the necessary Python libraries required to communicate with the BME280 sensor and the RFM95 LoRa module. Libraries like *time*, *board*, *busio*, and *digitalio* are used for time management and interfacing with Raspberry Pi's GPIO pins. The *adafruit_bme280* library enables reading temperature, humidity, and pressure data from the BME280 sensor via I2C, while *adafruit_tinyloara* handles LoRaWAN communication through the TinyLoRa interface. These imports are essential for both TTN and ChirpStack because they enable the basic hardware interfacing and network communication required to send sensor data over LoRaWAN.

2.4.2. Sensor Protocols Initializations

```
# --- Initialize I2C for BME280 ---
i2c = busio.I2C(board.SCL, board.SDA)
bme280 = Adafruit_BME280_I2C(i2c, address=0x76) # Use address from i2cdetect

# --- Optional: Set sea level pressure for accurate altitude (adjust to your location) ---
bme280.sea_level_pressure = 1013.25

# --- Initialize SPI for RFM95 (LoRa) ---
spi = busio.SPI(board.SCK, MOSI=board.MOSI, MISO=board.MISO)
```

Figure 4: Sensor-specific protocols initialization section from the Python script

This section sets up communication between the Raspberry Pi and the BME280 sensor using the I2C protocol. The `busio.I2C` class initializes the I2C bus on the default SDA and SCL pins of the Raspberry Pi. The BME280 sensor is then initialized at address 0x76, which is its typical I2C address. SPI communication is established with the RFM95W LoRa module, which is responsible for transmitting data to the LoRaWAN network and the sea level pressure is set to ensure accurate pressure and altitude measurements. This section is common to both TTN and ChirpStack implementations because both require accurate environmental data to be read and transmitted over the network.

2.4.3. LoRa Module Initialization

```
# --- LoRa Pins Setup ---
cs = digitalio.DigitalInOut(board.D5) # Chip Select (NSS)
irq = digitalio.DigitalInOut(board.D6) # IRQ (DIO0)
rst = digitalio.DigitalInOut(board.D4) # Reset (RST)
```

Figure 5: LoRa module-specific initialization section from the Python script

In this section, the SPI bus is initialized using Raspberry Pi's default SPI pins, and specific GPIO pins are assigned for chip select (CS), interrupt request (IRQ), and reset (RST) functionalities. This setup is necessary for both TTN and ChirpStack since it enables the Raspberry Pi to control the LoRa module and manage data transmission through the correct hardware interfaces.

2.4.4. LoRaWAN ABP Keys and Network Setup

```
# --- ABP Keys and DevAddr
devaddr = bytearray([0x26, 0x0B, 0x70, 0xBC]) # DevAddr
nwkey = bytearray([
    0xB1, 0xF1, 0x71, 0x47, 0x4C, 0x02, 0x2A, 0x0D,
    0x31, 0x4B, 0x6A, 0xFA, 0x07, 0x81, 0xF4, 0x46]) # NwkSKey
app = bytearray([
    0xDB, 0x4A, 0x30, 0x05, 0x6F, 0xF9, 0x6B, 0x49,
    0x99, 0xA0, 0x5B, 0x4B, 0xF4, 0x52, 0x79, 0xD0]) # AppSKey

# --- Initialize LoRa and TTN session ---
config = TTN(devaddr, nwkey, app, country='EU') # Change country if needed
lora = TinyLoRa(spi, cs, irq, rst, config)
```

Figure 6: LoRa ABP Keys for the network setup section from the Python script

This section is crucial for connecting to either TTN or ChirpStack, as it establishes the LoRaWAN session using Activation by Personalization (ABP). Here, the device address (*DevAddr*), network session key (*NwkSKey*), and application session key (*AppSKey*) are defined, enabling secure and authenticated communication over the LoRaWAN network. In the TTN script, these credentials are specified under *ttn_config*, while in the ChirpStack script, they are specified under *chirpstack_config*. This difference reflects the need to configure each script for its specific network, even though the underlying method of defining keys and initializing the LoRa session is the same.

2.4.5. Data Encoding

```
# --- Data packet buffer (6 bytes for 3 values) ---
data = bytearray(6)

# --- Function to encode float into two-byte integer (scaled by 10) ---
def encode_value(value):
    value = int(value * 10) # Scale float to int (2 decimal places)
    return (value >> 8) & 0xFF, value & 0xFF # High byte, Low byte
```

Figure 7: Data encoding section from the Python script

The data encoding function converts floating-point sensor readings into two-byte integers, scaled by a factor of ten to preserve one decimal place of precision. This encoding is necessary because LoRaWAN has strict limitations on the size of data payloads that can be transmitted. By scaling and compressing the sensor data into a bytearray, the script ensures that all three measurements (temperature, humidity, pressure) can be sent efficiently. This function is identical in both TTN and ChirpStack scripts because both networks require compact data formats for transmission.

2.4.6. Reading Sensor Data

```
# --- Main loop ---
interval = 30 # Interval in seconds

print("Starting BME280 LoRaWAN Transmission... Press CTRL+C to stop.")

try:
    while True:
        # --- Read data from BME280 ---
        temperature = bme280.temperature # °C
        humidity = bme280.humidity # %
        pressure = bme280.pressure # hPa

        # --- Debug: Print readable sensor output ---
        print(f"\nCurrent Time: {time.strftime('%Y-%m-%d %H:%M:%S', time.localtime())}")
        print(f"\nTemperature: {temperature:.2f} °C")
        print(f"\nHumidity: {humidity:.2f} %")
        print(f"\nPressure: {pressure:.2f} hPa")

        # --- Encode sensor data into bytes ---
        data[0], data[1] = encode_value(temperature)
        data[2], data[3] = encode_value(humidity)
        data[4], data[5] = encode_value(pressure)

        # --- Debug: Show encoded bytes to send ---
        print(f"Encoded Data: {[hex(b) for b in data]}")

        # --- Send the data over LoRa ---
        print("Sending packet to TTN...")
        lora.send_data(data, len(data), lora.frame_counter)
        print("Packet sent successfully!")

        # --- Increment frame counter to avoid duplicates ---
        lora.frame_counter += 1

        # --- Wait before next reading ---
        print(f"Waiting {interval} seconds before next transmission...")
        time.sleep(interval)

except KeyboardInterrupt:
    print("\nTransmission stopped by user. Exiting...")
```

Figure 8: Main loop section from the Python script

The main loop is the core of the script, responsible for the continuous operation of the system. It reads the current values from the BME280 sensor, encodes these values into a *bytearray*, and transmits the data

over the LoRaWAN network using the `send_data` function. After each transmission, the LoRaWAN frame counter is incremented to ensure packet uniqueness and prevent duplication. The loop then pauses for a set interval (*30 seconds*) before repeating. This structure is essential for both TTN and ChirpStack implementations, as it ensures regular data collection and reporting to the network.

Throughout the code, print statements are used to display key information such as sensor readings, encoded data, and confirmation of packet transmission. These outputs are vital for debugging, as they allow the user to monitor what data is being sent and ensure that the system is operating correctly.

2.5. Setting up and Connecting to TTN

2.5.1. Creating Application and End Device

This section explains how to connect your LoRa-enabled device (BME280 + RFM95W on Raspberry Pi) to The Things Network (TTN) using Activation By Personalization (ABP). ABP allows your device to join the network using pre-generated keys. Go to The Things Network Console. If you don't already have an account, click Sign Up to create one. Once logged in, you will be taken to the TTN Console Dashboard.

1. Creating an Application.

Create application

Within applications, you can register and manage end devices and their network data. After setting up your device fleet, use one of our many integration options to pass relevant data to your external services. Learn more in our guide on [Adding Applications](#).

Application ID*
my-bme280

Application name
BME280

Description
Application to receive and transmit temperature, pressure and humidity sensor data

Optional application description: can also be used to save notes about the application

Create application

- ☒ Click “Applications” on the left sidebar.
- ☒ Click “+ Add application” or “Create an application”.
- ☒ Fill in the form:
 - Application ID:** A unique identifier (e.g., my-bme280).
 - Application Name:** A friendly name (e.g., "BME280").
 - Description:** Optional, short description of application features.
- ☒ Leave other defaults unchanged and click Create application.

2. Adding an End Device

Register end device

Does your end device have a LoRaWAN® Device Identification QR Code? Scan it to speed up onboarding.

Scan end device QR code [Device registration help](#)

End device type

Input method ⓘ

☐ Select the end device in the LoRaWAN Device Repository

☒ Enter end device specifics manually

Frequency plan ⓘ *

Europe 863-870 MHz (SF9 for RX2 - recommended) ▼

LoRaWAN version ⓘ *

LoRaWAN Specification 1.0.2 ▼

Regional Parameters version ⓘ *

RP001 Regional Parameters 1.0.2 ▼

Show advanced activation, LoRaWAN class and cluster settings ▼

- ☒ Within your newly created application, go to the “End Devices” tab
- ☒ Click + Add end device.
- ☒ Select "Manually" (instead of using a pre-provisioned template).
- ☒ Fill in the required fields:
 - Frequency plan:** Select the appropriate one for your region.
 - LoRaWAN Version:** Select LoRaWAN Specification 1.0.3 (or the version supported by TinyLoRa).
 - Regional Parameters Version:** Select one that fits the version selected

Step 3: Click on the Advanced Activation drop-down menu

Show advanced activation, LoRaWAN class and cluster settings ^

Activation mode

- ☐ Over the air activation (OTAA)
- ☒ Activation by personalization (ABP)
- ☐ Define multicast group (ABP & Multicast)

Additional LoRaWAN class capabilities

None (class A only) v

Network defaults

- ☒ Use network's default MAC settings

Cluster settings

- ☐ Skip registration on Join Server

- ☒ Select "ABP" (under activation mode).
- ☒ Leave other defaults unchanged.

Step 4: Generate Provisioning Information

Provisioning information

DevEUI

70 B3 D5 7E D0 06 F0 A7 Generate 2/50 used

Device address *

26 0B 56 C1 Generate

AppSKey *

83 C1 2D 02 4A 71 34 66 BA 19 0E 1C A0 58 F2 37 Generate

NwkSKey *

7A BD 0B 32 A0 FE 23 9B 3C E7 16 06 34 7E A9 75 Generate

End device ID *

my-raspberrypi

After registration

- ☒ View registered end device
- ☐ Register another end device of this type

Register end device

- ☒ Scroll down to “Provisioning information”.
- ☒ Generate keys automatically or insert your own (recommended to let TTN generate them for security).
- ☒ Fill “End device ID” with a unique device name
- ☒ Click “Register end device”

After creating the device, go to the Device Overview page where the essential keys are shown. Copy these keys carefully to insert them into your Python script as shown above.

2.5.2. Payload Formatter

In the Payload Formatters tab, you can add “Custom Javascript Formatter”. This is a decoder function to interpret the raw data bytes that were encoded in the previous Python script received from your device. The script is written under the “Uplink” tab because the data is received from the gateway, not the other way around (downlink).

```
1 function Decoder(bytes, port) {
2   var decoded = {};
3
4   // Decode the temperature (first 2 bytes)
5   decoded.temperature = (bytes[0] << 8) | bytes[1];
6
7   // Decode the humidity (next 2 bytes)
8   decoded.humidity = (bytes[2] << 8) | bytes[3];
9
10  // Decode the pressure (last 2 bytes)
11  decoded.pressure = (bytes[4] << 8) | bytes[5];
12
13  // Convert to proper values (considering you scaled by 100 earlier)
14  decoded.temperature /= 10;
15  decoded.humidity /= 10;
16  decoded.pressure /= 10;
17
18  return decoded;
19 }
```

Figure 9: Custom javascript formatter decoder function

2.5.3. Running the Python Script

Using a Python virtual environment (*venv*) is a best practice to manage project-specific dependencies and avoid version conflicts. Here's a step-by-step guide on how to do this for the Python script created earlier.

Install `python3-venv` if not already installed

☒ `sudo apt-get install python3-venv`

Create a virtual environment named 'env'

☒ `python3 -m venv env`

Activate `venv`

☒ `source env/bin/activate`

In a different GitBash console, add your Python file to the virtual environment:



The screenshot shows the Advanced IP Scanner interface. At the top, there are four status bars: a Raspberry Pi icon, the IP address '192.168.0.114', the IP address '192.168.0.114', and the text 'Raspberry Pi Foundation'. On the right, there is a MAC address 'B8:27:EB:38:08:B2'.

Figure 10: Screenshot from Advanced IP Scanner displaying the MCU's IP address

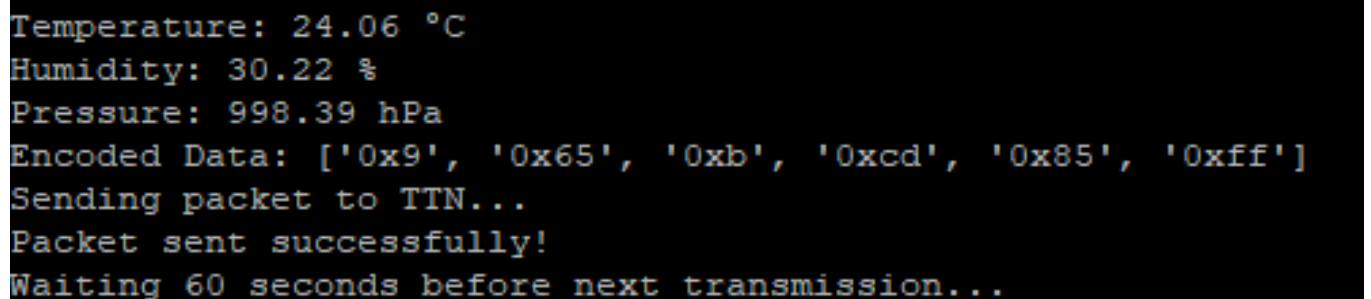
☒ `/c/<path_to_python_file> admin@<raspberrypi_ip_address>:/home/admin/`

Run the Python script

☒ `python3 <python_script>.py`

Deactivate when done

☒ `deactivate`



The screenshot shows the output of a Python script running on a Raspberry Pi. The text is displayed in a monospaced font on a black background. The output includes sensor readings for temperature, humidity, and pressure, followed by encoded data, a packet being sent to TTN, and a successful transmission confirmation, and finally a 60-second wait for the next transmission.

Figure 11: Raspberry Pi console output after the Python code runs successfully

2.6. Setting up and Connecting to ChirpStack

This section describes the complete procedure for connecting the BME280 sensor system to ChirpStack via a dedicated gateway. It includes configuring the gateway, creating an application and device in ChirpStack, obtaining the necessary ABP keys, and setting up the Python virtual environment to run the Python script.

2.6.1. Configuring Gateway for ChirpStack

For this test case, the gateway used is the WisGate RAK7289V2 Outdoor Gateway. If you're unsure about the gateway's IP address, use online tools like Advanced IP Scanner to detect all nearby IP addresses.

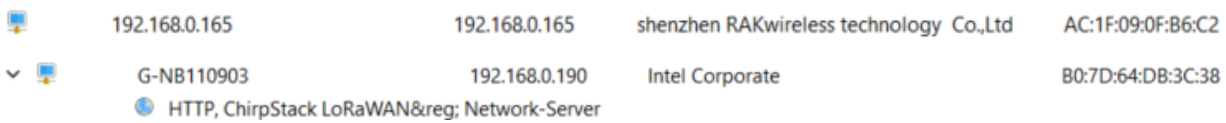
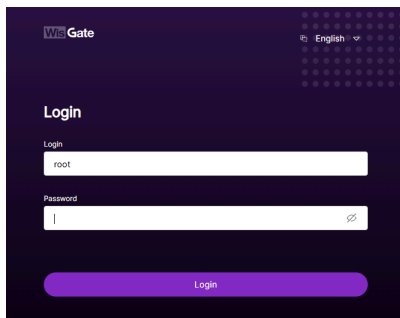


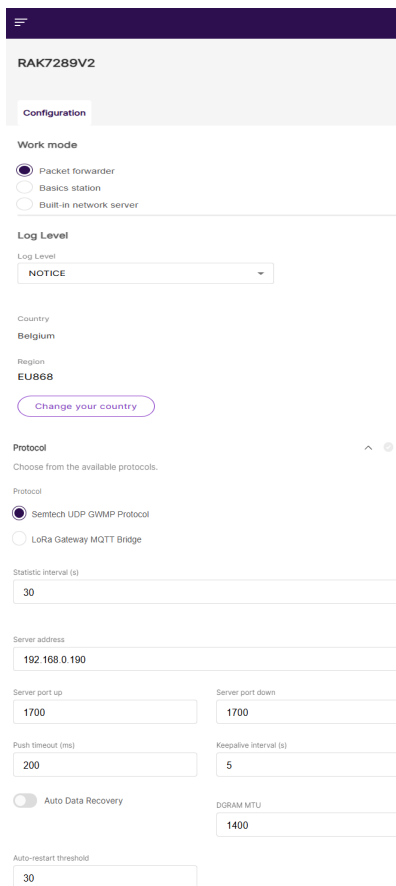
Figure 12: Screenshot from Advanced IP Scanner displaying the gateway and ChirpStack server IP addresses.

1. Logging in and Gateway Configuration



- ☒ Open a browser and navigate to the gateway's web interface
`http://<gateway_ip_address>`
- ☒ Login using credentials set by the owner of the gateway.
Username: root
Password: <password>
- ☒ Click Login

2. Configure Gateway for ChirpStack



Under the “Work mode” section:

- ☒ Select “Packet forwarder”
- ☒ Set the log level to “NOTICE”
- ☒ Change the country to the correct one from the list and choose the corresponding region.
- ☒ Leave the “Frequency plan” section as the default setting.
Verify that the frequency corresponding to your set region is added already, else add it yourself.

Under the “Protocol” section:

- ☒ Choose “Semtech UDP GWMP Protocol”
This is essential because ChirpStack expects LoRaWAN packet forwarders to use this standard UDP protocol.
- ☒ Set “Statistic Interval” to 30 seconds.
- ☒ Set “Server Address” as the IP address of the machine running ChirpStack Network Server on your local network.
- ☒ Ensure server port up and down is set to “1700”
- ☒ Leave other defaults unchanged.
- ☒ Click “Save changes” and exist.

The gateway will update its packet forwarder configuration and begin forwarding LoRa packets to ChirpStack at the defined IP and port.

If the gateway connection is not established yet i.e. in the dashboard there are no active devices: reboot the gateway via its web interface, this ensures all settings are applied cleanly and helps initiate a clean connection with ChirpStack.

2.6.2. Open ChirpStack Console

To run ChirpStack locally (or on a cloud machine), the easiest way is to use ChirpStack Docker. Below are the steps to clone the ChirpStack Docker repository, run it, and access the ChirpStack Network Server for registering devices and gateways.

Go to the home directory or the desired working directory

☒ `cd ~`

Clone ChirpStack Docker repository

☒ `git clone https://github.com/chirpstack/chirpstack-docker.git`

Change into the cloned directory

☒ `cd chirpstack-docker`

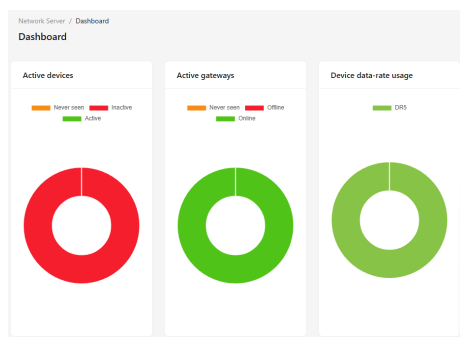
Run all services in the background (detached mode)

☒ `docker-compose up -d`

This will start with the ChirpStack Gateway Bridge, Network Server, Application Server, and PostgreSQL database. Redis and Mosquitto (MQTT broker).

2.6.2. Creating Application and End Device

1. Access the ChirpStack Application Server dashboard



☒ In your web browser type (from *Figure 11*):

`http://<YOUR_SERVER_IP>:8080`

☒ For local setups (on Raspberry Pi or the same machine), use:

`http://localhost:8080`

☒ Default credentials (first login):

Username: admin

Password: admin

2. Create Application

- ☒ On the left sidebar, click "Applications".
- ☒ Click "Add application" (top right corner).
- ☒ Fill out the application form:

Name: A name that reflects the purpose.

Description: (Optional) — Example: "Collecting temperature, humidity, and pressure data using Raspberry Pi and BME280".

- ☒ Click "Create application" to save.

3. Create/Add End Device

- ☒ In your new application, navigate to the "Devices" tab.
- ☒ Click "Add device".
- ☒ Fill in the device form:

Name: Unique name for the sensor node.

Device EUI: Click "Generate" or use a predefined one.

Join EUI: Click "Generate" or use a predefined one.

Device profile: Create a profile that matches the LoRaWAN version (e.g., *MAC V1.0.3* and *Class A*).

- ☒ Click "Submit" to create a new device.

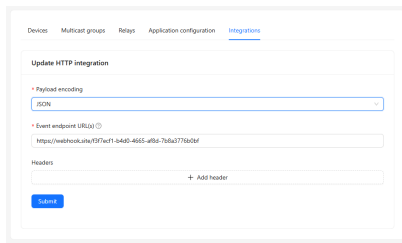
4. Generating ABP Activation Keys

- ☒ Navigate to the "Activation" within your Device.
- ☒ Generate keys automatically or insert your own (recommended to let ChirpStack generate them for security).
- ☒ Click "(Re)activate device"

Copy these keys carefully to insert them into your Python script as done earlier.

5. Integrations (Optional)

If you want to send data via POST requests to a specified URL whenever an event occurs (e.g., receiving a message from a device, device activation, or error events), add the HTTP Integration with the URL(s) of your desired webpage.



For example, when your BME280 device sends data, ChirpStack can forward that data as an HTTP POST request to an external web server or API for storage, visualization, or real-time processing.

2.6.3. Starting the Python Script

Using a Python virtual environment (*venv*) is a best practice to manage project-specific dependencies and avoid version conflicts. Here's a step-by-step guide on how to do this for the Python script created earlier.

Install `python3-venv` if not already installed

☒ `sudo apt-get install python3-venv`

Create a virtual environment named 'env'

☒ `python3 -m venv env`

Activate `venv`

☒ `source env/bin/activate`

In a different GitBash console, add your Python file to the virtual environment:

☒ `/c/<path_to_python_file> admin@<raspberrypi_ip_address>:/home/admin/`

Run the Python script

☒ `python3 <python_script>.py`

Deactivate when done

☒ `deactivate`

```
Temperature: 23.77 °C
Humidity: 30.58 %
Pressure: 997.94 hPa
Encoded Data: ['0x0', '0xed', '0x1', '0x31', '0x26', '0xfb']
Sending packet to ChirpStack...
Packet sent successfully!
Waiting 30 seconds before next transmission...
```

Figure 13: Raspberry Pi console output after the Python code runs successfully

2.7. Results

This section summarizes the entire process of integrating a BME280 environmental sensor with a LoRaWAN network using both The Things Network (TTN) and ChirpStack, as well as the key outcomes of this setup. The goal was to demonstrate how to transmit multiple sensor data (temperature, humidity, and pressure) over LoRaWAN and compare how data flows through both network infrastructures.

2.7.1. TTN

The images below illustrate the successful transmission and decoding of sensor data from the Raspberry Pi to The Things Network. On the left side, the Raspberry Pi console output displays the raw environmental sensor readings obtained from the BME280 sensor — including temperature, humidity, and pressure — along with their corresponding encoded hexadecimal values formatted for LoRaWAN transmission. On the right side, the TTN Live Data tab showcases the uplink frames received from the Raspberry Pi node. Each frame includes both the raw hexadecimal payload and the decoded JSON object containing the original sensor measurements. Notably, the hexadecimal data shown in the TTN console precisely matches the encoded data from the Raspberry Pi output, confirming the accuracy of the encoding and transmission process. This result validates the correct operation of the end-to-end system, from sensor reading and data encoding on the Raspberry Pi to successful delivery, decoding, and visualization within TTN.

Temperature: 23.30 °C
Humidity: 28.37 %
Pressure: 998.37 hPa
Encoded Data: ['0x9', '0x1a', '0xb', '0x14', '0x85', '0xfc']
Sending packet to TTN...
Packet sent successfully!
Waiting 60 seconds before next transmission...

Temperature: 23.30 °C
Humidity: 28.77 %
Pressure: 998.45 hPa
Encoded Data: ['0x9', '0x1a', '0xb', '0x3d', '0x86', '0x4']
Sending packet to TTN...
Packet sent successfully!
Waiting 60 seconds before next transmission...

Temperature: 23.29 °C
Humidity: 28.35 %
Pressure: 998.45 hPa
Encoded Data: ['0x9', '0x18', '0xb', '0x13', '0x86', '0x3']
Sending packet to TTN...
Packet sent successfully!
Waiting 60 seconds before next transmission...

Temperature: 23.32 °C
Humidity: 28.24 %
Pressure: 998.45 hPa
Encoded Data: ['0x9', '0xb', '0xb', '0x8', '0x86', '0x4']
Sending packet to TTN...
Packet sent successfully!
Waiting 60 seconds before next transmission...

Temperature: 23.32 °C
Humidity: 28.69 %
Pressure: 998.51 hPa
Encoded Data: ['0x9', '0x1c', '0xb', '0x3d', '0x86', '0x3']
Sending packet to TTN...
Packet sent successfully!
Waiting 60 seconds before next transmission...

Temperature: 23.32 °C
Humidity: 28.48 %
Pressure: 998.51 hPa
Encoded Data: ['0x9', '0xb', '0xb', '0x1f', '0x86', '0xa']
Sending packet to TTN...
Packet sent successfully!
Waiting 60 seconds before next transmission...

Temperature: 23.35 °C
Humidity: 28.42 %
Pressure: 998.58 hPa
Encoded Data: ['0x9', '0x1e', '0xb', '0x1a', '0x86', '0x9']
Sending packet to TTN...
Packet sent successfully!
Waiting 60 seconds before next transmission...

Applications > BME280 > End devices > STM32-RFM95 > Live data

STM32-RFM95
ID: ucbe-end1

Device overview

Live data

Messaging

Location

TIME	TYPE	DATA PREVIEW
↓ 12:25:40	Schedule data downlink for transmissi...	DevAddr: 26 00 70 BC MAC payload: 92 28 F9 41 E6 EA B7 BC 05 56 04 30 81 18 91 4E 52 60 01 EF 44 C8 0C 65 08 05 96 C3 2A 54 03... Rx1 Delay: 5
↑ 12:25:40	Forward uplink data message	DevAddr: 26 00 70 BC Payload: { humidity: 38.6, pressure: 997.9, temperature: 23.1 } 00 E7 01 32 26 FB FPort: 1 Data rate: SF7BM125 SNR: 7.75 RSSI: -39
↑ 12:25:40	Successfully processed data message	DevAddr: 26 00 70 BC
↓ 12:25:10	Schedule data downlink for transmissi...	DevAddr: 26 00 70 BC MAC payload: AB 24 3C 1A 18 62 54 64 4B 3D A6 ED F8 F2 B3 23 1F 6F 35 4E 2A 6B 39 AA 1C 94 52 9F 32 F5 E7... Rx1 Delay: 5
↑ 12:25:10	Forward uplink data message	DevAddr: 26 00 70 BC Payload: { humidity: 38.7, pressure: 997.9, temperature: 23.1 } 00 E7 01 33 26 FB FPort: 1 Data rate: SF7BM125 SNR: 9 RSSI: -43
↑ 12:25:10	Successfully processed data message	DevAddr: 26 00 70 BC
↓ 12:24:09	Schedule data downlink for transmissi...	DevAddr: 26 00 70 BC MAC payload: 97 6C 66 82 FD A4 E0 C3 05 91 04 AA 25 03 C4 BF A7 C6 00 ED C8 49 C2 5C 56 CF 0B 0B 0F E9 E8... Rx1 Delay: 5
↑ 12:24:09	Forward uplink data message	DevAddr: 26 00 70 BC Payload: { humidity: 31.2, pressure: 997.9, temperature: 23 } 00 E6 01 30 26 FB FPort: 1 Data rate: SF7BM125 SNR: 7.5 RSSI: -43
↑ 12:24:09	Successfully processed data message	DevAddr: 26 00 70 BC
↓ 12:23:39	Schedule data downlink for transmissi...	DevAddr: 26 00 70 BC MAC payload: 1E 9A 78 31 8A 75 F5 39 01 D3 FE 52 82 AB 79 32 A7 14 54 8D 47 5B DA 83 F6 F7 E9 97 05 66 58... Rx1 Delay: 5
↑ 12:23:39	Forward uplink data message	DevAddr: 26 00 70 BC Payload: { humidity: 38.9, pressure: 997.9, temperature: 23 } 00 E6 01 35 26 FB FPort: 1 Data rate: SF7BM125 SNR: 6.25 RSSI: -36
↑ 12:23:39	Successfully processed data message	DevAddr: 26 00 70 BC
↓ 12:22:09	Schedule data downlink for transmissi...	DevAddr: 26 00 70 BC MAC payload: E6 F2 22 92 70 65 7B 62 0B F8 05 8D BA 6B FC 05 5C 68 76 9A 72 03 E5 1C 6E D9 7F 35 2E 58 53... Rx1 Delay: 5
↑ 12:22:08	Forward uplink data message	DevAddr: 26 00 70 BC Payload: { humidity: 38.8, pressure: 997.9, temperature: 22.9 } 00 E5 01 34 26 FB FPort: 1 Data rate: SF7BM125 SNR: 9.25 RSSI: -43
↑ 12:22:08	Successfully processed data message	DevAddr: 26 00 70 BC
↓ 12:20:00	Schedule data downlink for transmissi...	DevAddr: 26 00 70 BC MAC payload: 2C 74 CC 4C E9 08 03 0D A1 A8 42 1E 05 E7 68 8D 46 F2 CF F7 C3 9B 3A 65 E9 0D 66 48 94 91 2A... Rx1 Delay: 5
↑ 12:20:00	Forward uplink data message	DevAddr: 26 00 70 BC Payload: { humidity: 38.7, pressure: 997.9, temperature: 22.9 } 00 E5 01 33 26 FB FPort: 1 Data rate: SF7BM125 SNR: 9.75 RSSI: -45

Figure 14: L-R, Raspberry Pi console after TTN script is run and the live data screen

2.7.2. ChirpStack

The images below demonstrate the full data transmission flow using ChirpStack, from the Raspberry Pi running the Python script to the decoded data visualization in the ChirpStack console. On the left side, the Raspberry Pi console displays live sensor readings (temperature, humidity, and pressure), along with the encoded hexadecimal packet that is sent over LoRaWAN. This confirms that the sensor data is being

correctly processed and prepared for transmission. In the middle section, the ChirpStack "Events" tab for the registered end device shows a series of successful uplink events, confirming that packets are consistently received and processed by the ChirpStack network server. On the right side, the detailed JSON payload view of an "UnconfirmedDataUp" event displays the original sensor data — accurately extracted from the received hexadecimal payload. The matching values between the Raspberry Pi console output and the decoded JSON data in ChirpStack verify that the encoding, transmission, and decoding pipeline works as intended, providing accurate end-to-end delivery of sensor data over the LoRaWAN network.

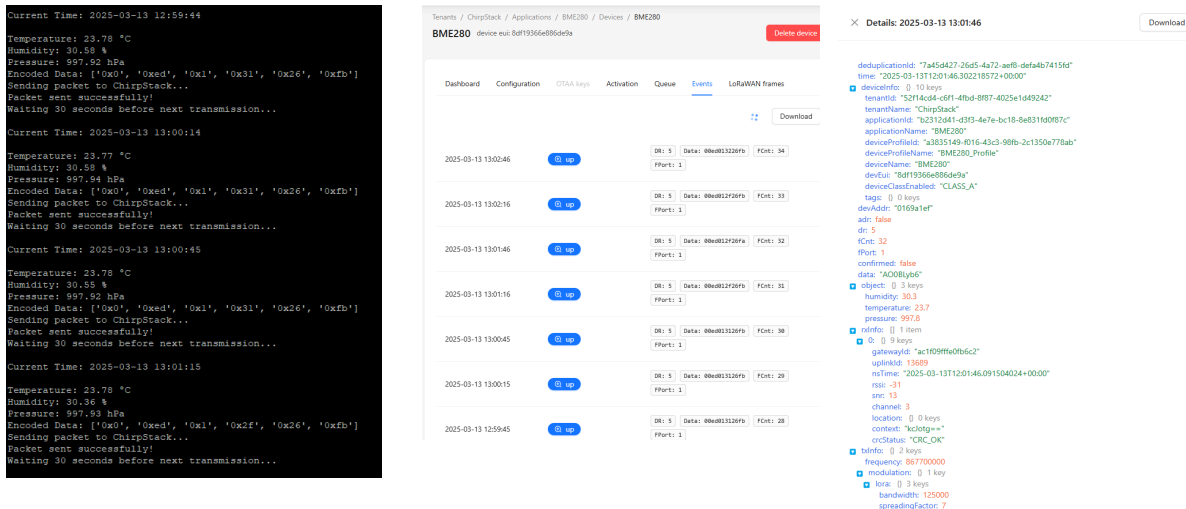


Figure 15: L-R, Raspberry Pi console after ChirpStack script is run, “Events” updated, and the “UnconfirmedDataUp”

2.8. Common Issues and Troubleshooting Tips

- Gateway Not Connecting to ChirpStack (No Packets Forwarded):** Wi-Fi connection errors prevented the gateway configuration from saving correctly. Rebooting the gateway after making changes resolved the issue.
- No Uplink Data Seen in ChirpStack or TTN Console:** A typographical error in the Device Address key within the Python script led to failed transmissions. Ensuring that ABP keys were accurately copied from the LoRaWAN console prevented further transmission issues.
- Data Payload Arrives but Not Decoded Properly:** The humidity value was not decoded correctly despite accurate encoded data. The issue was caused by an overflow in the stack due to incorrect data scaling. Modifying the encoding logic to scale by a factor of 10 instead of 100 resolved the decoding error.
- Frame-counter Errors or Duplicate Frame Issues:** The frame counter did not increment as expected, although data transmission remained functional. The issue was corrected by explicitly incrementing the counter in the Python script and disabling frame counter validation in ChirpStack during testing.
- Python Script Errors with LoRa or I2C Initialization:** The virtual environment on the Raspberry Pi failed to reflect updates to the Python script, causing outdated code to run. Manually copying the

updated script into the virtual environment after each change ensured the correct version was executed.

6. **Frame-counter Reset:** If the device abruptly goes offline and transmission is started again, a common error is “Frame-counter reset or rollover detected”. This occurs when the frame counters for both downlink and uplink transmissions have not been manually reset to 0 before restarting the transmission. To resolve this issue, navigate to the “Activation” tab of your device in ChirpStack and set both the “Uplink frame-counter” and “Downlink frame-counter” fields to 0, then click the “(Re)activate device” button.

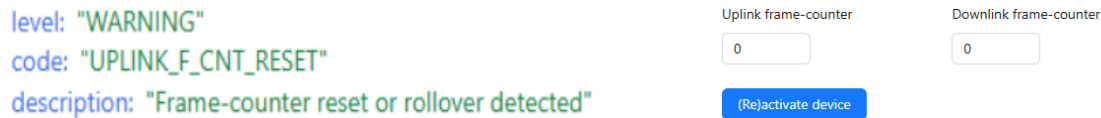


Figure 16: Example of frame-counter reset warning and activation settings in ChirpStack.

3. Feasibility Study and Solution Selection for the Gladiolen Project

3.1. Introduction to the Gladiolen Project

The Gladiolen Project is a smart waste management initiative aimed at improving the efficiency and sustainability of waste collection during large-scale events. With thousands of attendees spread across a vast festival venue, traditional waste collection methods often lead to overflowing bins, inefficient routing of collection staff, and increased operational costs. To address these challenges, the project implements a LoRaWAN-based monitoring system that enables real-time tracking of bin status and location. At the core of the system are ultrasonic distance sensors, which are used to detect the fill level of each waste bin by measuring the distance between the sensor and the waste surface. When the bin reaches a predetermined threshold, it is classified as “full.” In addition, each bin is equipped with a GPS module to provide location data, enabling event organizers to efficiently map and track waste collection points throughout the venue.

The collected data i.e. bin status and bin location are wirelessly transmitted using LoRaWAN technology, which is well-suited for low-power, long-range communication in outdoor environments. This allows for remote monitoring of all bins in the network without relying on power-hungry or short-range technologies like Wi-Fi or cellular. The focus of this section is on the wireless transmission of sensor data, where bin status and location are sent over a LoRa network (either ChirpStack or TTN) to an application server, where it is processed, stored, and visualized. This enables data-driven decision-making, such as dynamic route planning for collection teams and timely interventions when bins are near capacity, ultimately enhancing festival cleanliness, reducing manual checks, and optimizing resource allocation.

3.2. Defining Requirements for the Gladiolen Project

To ensure reliable transmission and network efficiency, the system is expected to meet certain requirements:

1. **Sensor Selection:** This was completed in the first phase of the project during the initial feasibility study, certain criteria were set to test the different sensor (ultrasonic distance sensors and Time-of-Flight sensors) options. The evaluation criteria included shock resistance, ease of maintenance, cost, and reliability. While ToF sensors offer higher accuracy, their higher cost and greater sensitivity to dirt and physical shock make them unsuitable for this industrial environment. Ultrasonic sensors were chosen as the optimal solution due to their robustness, lower cost, and reduced sensitivity to environmental factors.
2. **LoRaWAN Communication:** To achieve accurate and efficient data transmission, the Heltec Wireless Tracker was chosen as the LoRaWAN module. It includes an SX1276 LoRa transceiver which enables long-range communication over the established LoRaWAN network and also enables real-time tracking using the built-in GPS module. An onboard OLED alongside the ESP32 processor provides all the computing power needed for sensor integration data processing and visualization. This device ensures seamless integration of the sensor, GPS module, and LoRaWAN communication into a single, compact solution for transmitting bin status and location data.
3. **Reliable Gateway Connection:** For a project of this size, a reliable connection between the LoRaWAN gateway and the end devices (bin) is paramount for the real-time transmission of data. Depending on the type of network deployment used (private or public) a dedicated gateway might be required, with the gateway placed strategically to ensure coverage for all end devices (bins) within the application. Data transmission is encrypted to ensure security and prevent unauthorized access that might corrupt or disrupt the transmitted data.

3.3. Feasibility Analysis of TTN Approach

The integration of The Things Network (TTN) as a LoRaWAN provider for the Gladiolen Project presents both advantages and disadvantages in its implementation. This feasibility analysis is based on first-hand testing with the BME280 sensor, assessing network performance, reliability, and scalability, and extrapolating these insights to determine TTN's suitability for the Gladiolen Project's requirements. The evaluation focuses on critical factors for data transmission, including accuracy, transmission speed, ease of integration, and scalability.

3.3.1. Advantages

1. **Public Infrastructure:** The Things Network (TTN) provides a free and community-driven LoRaWAN network, eliminating the need for organizations to set up and maintain expensive private infrastructure. This significantly reduces deployment costs and complexity, as project stakeholders can leverage existing TTN gateways to transmit bin status and GPS data. In areas where TTN coverage is available, the system can function with only an indoor gateway, which is cost-effective

and easy to install. This makes TTN a viable solution for this smart waste management application where reducing initial infrastructure costs is a priority.

2. **Scalability:** TTN is designed to support small-to-large-scale deployments, making it an excellent choice for expanding the waste management system across this festival venue and other locations. The ability to register multiple devices and integrate them into a single network ensures that the system can be easily scaled as the number of waste bins increases. Additionally, because TTN is a global, community-powered network, it enables seamless integration with LoRaWAN infrastructure in other cities or regions, facilitating future expansion beyond the initial deployment site.
3. **Easy Integration:** TTN's web-based console provides an intuitive interface for device registration, application setup, and data visualization, making deployment straightforward. Furthermore, TTN offers several built-in integration options (like AWS and MQTT), which allow data to be seamlessly forwarded to external platforms for storage, visualization, and analysis. This flexibility enables real-time monitoring and automated responses, improving the efficiency of waste collection by allowing external applications to process sensor data and generate collection schedules dynamically.

3.3.2. Disadvantages

1. **Coverage Limitations:** One of the major drawbacks of TTN is that it relies on community-operated gateways, meaning coverage is not guaranteed in all areas. While urban environments or event locations with active TTN gateways may offer reliable service, some regions may suffer from poor or no connectivity. This forces projects like the Gladiolen waste management system to deploy private gateways in areas lacking coverage, increasing infrastructure costs and complexity. Additionally, even in areas with TTN coverage, network congestion can impact message delivery, leading to unreliable performance in high-traffic environments.
2. **Network Reliability:** Because TTN is a public network, project stakeholders do not have full control over network availability, performance, or maintenance. Gateway failures, maintenance downtimes, or network congestion caused by other devices using TTN in the same area can result in lost or delayed messages. If a gateway goes offline, all connected devices relying on that gateway may lose connectivity, interrupting real-time waste bin monitoring. This poses a significant risk for applications requiring consistent and predictable data transmission.
3. **Fair Use Policy:** TTN enforces a Fair Use Policy, which limits message frequency and payload size. This presents a serious challenge for applications requiring periodic data transmission, such as waste bin status updates. In testing, sending just 6 bytes of payload took up to 9 minutes, highlighting significant network latency. This issue likely arises due to network congestion, as multiple devices in the area compete for transmission slots. The delays and inconsistencies in data delivery make TTN unsuitable for time-sensitive applications where immediate updates are required. The system must be carefully optimized to only transmit data when necessary.

3.4. Feasibility Analysis of ChirpStack Approach

ChirpStack offers a private, self-hosted LoRaWAN network solution, granting full control over data transmission and network management. This feasibility analysis evaluates ChirpStack's advantages and disadvantages, considering its suitability for the Gladiolen Project based on first-hand testing with the BME280 sensor. The assessment focuses on network reliability, scalability, integration flexibility, security, and real-time performance, which are critical for efficient waste management operations.

3.4.1. Advantages

1. **Full Network Control:** One of the biggest advantages of ChirpStack is the complete control it provides over network infrastructure, device management, and data flow. Unlike TTN, where devices rely on public gateways maintained by third parties, ChirpStack requires the deployment of private LoRaWAN gateways, ensuring that the network is always available and optimized for the project's specific needs. This eliminates issues related to network downtime and gateway availability that can occur when using a public LoRaWAN service.
2. **Improved Network Reliability:** Because ChirpStack operates on privately managed infrastructure, it removes dependencies on external gateways and public network constraints. In contrast to TTN, which can experience latency issues and transmission delays due to network congestion, ChirpStack offers dedicated bandwidth, ensuring faster and more consistent data transmission. This is especially critical for applications that require periodic or time-sensitive updates, such as monitoring bin status in real-time.
3. **Strong Data Security and Privacy:** With ChirpStack, all data transmission, processing, and storage remain within the organization's servers or a secured cloud environment, eliminating the risks associated with public network exposure. This ensures higher security by allowing full encryption, authentication, and access control policies tailored to project needs. For a smart waste management system, keeping data secure and private prevents unauthorized access or manipulation of bin status and location data.

3.4.2. Disadvantages

1. **Higher Initial Setup Complexity:** Unlike TTN, which provides a ready-to-use, community-managed infrastructure, ChirpStack requires manual setup and configuration of multiple components, including the LoRaWAN Network Server, Application Server, Gateway Bridge, and MQTT broker. This introduces significant complexity, requiring expertise in network configuration, Docker container management, and server administration. Setting up ChirpStack can be time-consuming, requiring ongoing maintenance to ensure server uptime, data integrity, and security.
2. **Hardware Costs and Gateway Deployment:** Since ChirpStack operates on private infrastructure, dedicated LoRaWAN gateways must be deployed in areas where waste bins are installed. This increases hardware costs, as multiple LoRaWAN gateways (such as WisGate RAK Wireless) may be required to ensure full coverage. Additionally, gateways used for ChirpStack require special configuration and cannot be shared between TTN and ChirpStack simultaneously. This means that if a

region already has TTN gateways, they cannot be reused for ChirpStack without dedicating separate hardware, further increasing deployment costs.

3. **User Interface Readability:** The user interface for viewing decoded payloads in ChirpStack is less user-friendly compared to TTN. In TTN, the decoded payload is displayed in plain text, making it easy to read sensor data directly within the TTN console. In contrast, ChirpStack presents decoded payloads in JSON format, which can be less intuitive for quick data interpretation. However, ChirpStack is designed for integration with external cloud platforms, meaning the decoded payload is often intended to be processed and displayed externally via custom dashboards, third-party cloud services, or applications such as AWS SNS, myDevices, HTTP, and InfluxDB. This adds an extra step in the implementation process, requiring the setup of an additional platform for real-time sensor data visualization.

3.5. Comparison Summary: TTN vs. ChirpStack

This table summarizes the key differences between TTN and ChirpStack, comparing factors such as network control, scalability, reliability, ease of integration, and cost to determine the best fit for the project's requirements.

Table 1: Comparison between features in both LoRaWAN network deployment types.

Feature	The Things Network (TTN)	ChirpStack
Network Type	Public, community-driven	Private, self-hosted
Infrastructure Control	No control over infrastructure (uses community gateways)	Full control over infrastructure, gateways, and servers
Coverage	Limited, depends on community gateways	Unlimited coverage depends on the number of private gateways
Data Security & Privacy	Data passes through public servers	Data remains within the private infrastructure
User Interface Readability	More user-friendly; payloads displayed as plain text	Less user-friendly; payloads displayed in JSON, requiring external visualization
Ease of Integration	Easy, built-in integrations (HTTP, MQTT, AWS, etc.)	Requires additional setup for integrations (MQTT, HTTP, cloud storage)
Latency & Transmission Speed	High latency (up to 9 minutes for 6-byte payloads) due to congestion	Lower latency, more reliable transmission (private gateway avoids congestion)
Scalability	Easier to scale, but depends on community gateways	Highly scalable, but requires more infrastructure investment
Fair Use Policy	Strict limitations on message frequency and payload size	No restrictions on message frequency or payload size
Message Reliability	Unreliable in congested networks; can have message delays	Reliable, since network congestion is controlled
Setup Complexity	Simple setup, web-based console for easy	Complex setup (requires configuring multiple

	device registration	services like LNS, MQTT, and DB for optimization)
Cost	Free to use, but requires public gateway access	Higher initial cost due to private gateways and server hosting

3.6. Recommendations for Implementing the Gladiolen Project

Based on hands-on testing using the BME280 sensor, and an evaluation of both TTN and ChirpStack for the data transmission requirements of the Gladiolen Project, the following recommendations are proposed to optimize performance, reduce network load, and improve system reliability. These suggestions aim to balance data accuracy, network efficiency, and implementation practicality for large-scale deployment.

1. **Reduce Data Size:** In the Gladiolen project where multiple end devices are transmitted from across a large venue, the size of each payload significantly affects transmission success and speed. The existing system already has something similar implemented that classifies bin status based on the distances recorded by the ultrasonic sensor and then transmits a string of the “full” or “not full” status. Replacing this with binary logic that represents the two statuses of the bin i.e. 1 for full and 0 for not full drastically reduces payload size, improves transmission speed, and eliminates the need to decode long strings or float values. This is particularly useful in public networks like TTN, where payload size and transmission frequency are restricted.
2. **Assigning Unique Bin IDs:** Following the style of LoRaWAN device addressing, each bin should be assigned a unique ID hardcoded into the firmware. Instead of sending GPS coordinates with every transmission (which increases payload size and power usage), the bin can simply send its unique ID along with the binary status (1 or 0). The location and metadata for each bin can be managed and visualized on the external dashboard, allowing the device itself to transmit the smallest possible payload while still providing useful information.
3. **Event-Based Transmission:** The BME280 test setup transmitted data every 30 seconds, which made sense for fast-changing environmental data. However, in the Gladiolen Project, bin status does not change that frequently, so such frequent transmissions are unnecessary and wasteful. Instead, the system should be programmed to transmit only when the bin status changes from 1 to 0 when the bin is emptied and 0 to 1 when it becomes full. To ensure connectivity and data validity, a periodic heartbeat transmission every 30 minutes can be included even if no state change occurs. This reduces unnecessary network traffic, improves battery life, and complies with fair use policies on public networks like TTN.
4. **Flag Transmission State:** In the BME280 test case, a generic printout ("Packet Sent Successfully") confirms transmission. For the Gladiolen Project, implement a transmission flag in software: once a bin sends a 1 (full) message, it should not retransmit the same status unless it changes back to 0 and becomes full again. This prevents duplicate uplinks and conserves airtime. Alternatively, a downlink acknowledgment from the application server can be used to confirm that the bin's full status has been received and only then should the device reset and prepare for future transmissions.
5. **Appropriate Dashboard or External Platform:** Given the multiple integration options supported by TTN and ChirpStack (e.g., HTTP, MQTT, AWS SNS, InfluxDB, myDevices, ThingsBoard), it is

essential to choose a user-friendly, scalable dashboard to visualize bin status. Since ChirpStack does not include an easily readable payload viewer, using an external platform like ThingsBoard or Grafana (via InfluxDB) is ideal for managing, categorizing, and displaying live data. A practical and accessible solution is to use HTTP integration with Google Sheets via Google Apps Script. This method enables automated logging of each transmission into a shared, cloud-based spreadsheet, making it easy for team members to view, filter, and analyze bin data in real-time without the need for a complex backend. It also supports structured entries (e.g., timestamp, bin ID, status), offering both visibility and traceability in a familiar format. This setup is especially useful for project partners or festival staff who may not have technical expertise but still need access to the data.

6. **Optimize Sensor Read Interval:** Since bin fill levels change gradually, reading sensor data at short intervals (e.g., every 10 seconds) is unnecessary and leads to excessive power consumption. A more efficient approach is to configure the microcontroller to perform a lightweight distance check every 1 to 5 minutes, depending on the expected usage rate and bin fill behavior. To further conserve power and reduce processing load, the sensor should only proceed with full evaluation or data transmission if a threshold distance is consistently met (e.g., multiple consecutive readings indicate the bin is near full). This strategy reduces microcontroller activity, extends battery life, and ensures that only meaningful data is transmitted, maintaining reliable system performance without unnecessary resource usage.
7. **Plan Gateway Placement Strategically:** When using ChirpStack, gateways should be placed strategically to maximize coverage while minimizing hardware costs. For the Gladiolen Project, the current plan proposes about 50 waste bins (end devices) per gateway, which is feasible for LoRaWAN under moderate traffic conditions. Each gateway is equipped with a private Wi-Fi router connected via Ethernet, minimizing issues with Wi-Fi congestion or instability common at large festivals. This local networking setup ensures that data transmission remains stable and isolated from public network fluctuations.
8. **Create a Backup Plan:** To improve system reliability, it is advisable to implement a backup transmission pathway. If the primary system uses ChirpStack with a private gateway, an optional fallback could be to set up a secondary gateway configured for TTN, or vice versa. This ensures that if transmission errors, network outages, or gateway failures occur on the main network, critical sensor data can still be transmitted through the backup. This is especially useful in high-stakes or time-sensitive deployments, where uninterrupted monitoring is important. While not required for initial deployment, this recommendation provides resilience and flexibility and can be implemented as resources allow or if network instability is observed during testing or live use.

4. Conclusion

After testing with the BME280 sensor, evaluating both The Things Network (TTN) and ChirpStack, and considering the specific needs of the Gladiolen Project, it is clear that ChirpStack is the preferred solution for long-term deployment. Its advantages such as full network control, higher reliability, unrestricted payload size, improved latency, and better data security make it more suitable for a large-scale, event-based waste monitoring system.

While TTN does offer easier initial setup and free access to community infrastructure, its disadvantages such as network congestion, fair use limitations, and inconsistent transmission times pose challenges for real-time applications like bin status monitoring. However, several of these drawbacks can be partially mitigated through the recommendations outlined in this report, including reducing payload size, adopting event-based transmission, assigning unique bin IDs, and optimizing read intervals. Even with these improvements, TTN remains more suitable for small-scale, low-priority, or temporary setups where quick deployment is critical and infrastructure is limited. For the main Gladiolen deployment, especially in high-traffic or critical zones, ChirpStack is strongly recommended. With the proposed software and transmission optimizations, ChirpStack becomes not only more powerful but also more efficient and scalable, offering a robust foundation for a smart waste management system capable of handling future expansion.

Finally, while TTN may be used as a fallback or for initial testing, the Gladiolen Project should prioritize ChirpStack for its superior control, reliability, and compatibility with custom dashboards and real-time monitoring platforms. Implementing the recommended strategies will maximize ChirpStack's potential and ensure a resilient, low-maintenance, and scalable solution for smart bin monitoring.

5. References

- [1] The Things Network, "What is LoRaWAN?," *The Things Network Documentation*, [Online]. Available: <https://www.thethingsnetwork.org/docs/lorawan/what-is-lorawan/>. [Accessed: Mar. 17, 2025].
- [2] The Things Industries, "LoRaWAN basics," *The Things Industries Documentation*, [Online]. Available: <https://www.thethingsindustries.com/docs/getting-started/lorawan-basics/>. [Accessed: Mar. 17, 2025].
- [3] ChirpStack, "ChirpStack open-source LoRaWAN Network Server stack," *ChirpStack Documentation*, [Online]. Available: <https://www.chirpstack.io/>. [Accessed: Mar. 17, 2025].
- [4] The Things Industries, "What is a LoRaWAN Network Server?," *The Things Industries News*, Aug. 15, 2023. [Online]. Available: <https://www.thethingsindustries.com/news/what-lorawan-network-server/#:~:text=The%20LoRaWAN%20Network%20Server%20is,data%20routing%20throughout%20the%20network>. [Accessed: Mar. 17, 2025].
- [5] Heltec Automation, "Wireless Tracker," *Heltec.org*. [Online]. Available: <https://heltec.org/project/Wireless-Tracker/>. [Accessed: Mar. 20, 2025].
- [6] Adafruit, "Installing CircuitPython on Raspberry Pi," *Adafruit Learning System*. [Online]. Available: <https://learn.adafruit.com/circuitpython-on-raspberrypi-linux/installing-circuitpython-on-raspberry-pi>. [Accessed: Mar. 20, 2025].
- [7] ChirpStack, "ChirpStack open-source LoRaWAN Network Server," *ChirpStack Documentation*. [Online]. Available: <https://www.chirpstack.io/>. [Accessed: Mar. 20, 2025].
- [8] Google Developers, "Google Apps Script Web Apps," *Google Developers*. [Online]. Available: <https://developers.google.com/apps-script/guides/web>. [Accessed: Mar. 20, 2025].
- [9] ThingsBoard, "ThingsBoard Open-source IoT Platform," *ThingsBoard.io*. [Online]. Available: <https://thingsboard.io/>. [Accessed: Mar. 20, 2025].
- [10] InfluxData, "InfluxDB Overview," *InfluxData Documentation*. [Online]. Available: <https://docs.influxdata.com/>. [Accessed: Mar. 20, 2025].
- [11] IFTTT, "Webhooks," *IFTTT Platform*. [Online]. Available: https://ifttt.com/maker_webhooks. [Accessed: Mar. 20, 2025].