Synthetic IoT Telemetry Generator for Mining Site Monitoring

1. Overview

The synthetic_generator.py Python script generates synthetic telemetry data for a IoT-based monitoring system for a mining site, covering both underground and surface operations. This tool simulates realistic data streams to emulate a production environment where real sensor data is unavailable, enabling testing and development of IoT architecture. The generator supports a variety of asset types and operational modes, publishing data via MQTT to a local broker or AWS IoT Core.

1.1 Supported Asset Types

The generator simulates data for four categories of assets, each with five critical metrics tailored to mining operations:

- **Vehicles** (haul trucks, loaders, excavators):
  - GPS location (latitude, longitude)
  - Speed (km/h)
  - Engine temperature (°C)
  - Vibration/accelerometer (g)
  - Fuel/battery level (%)
- **Fixed Assets** (fans, ventilation systems, conveyor belts):
  - Vibration (mm/s)
  - Motor temperature (°C)
  - Operational status (running, idle, stopped)
  - Airflow rate (m^3/s)
  - Power consumption (W)
- **Environmental Sensors** (CO/CO_2, methane, dust, temperature/humidity, ground stability):
  - CO/CO_2 levels (ppm)
  - Methane levels (ppm)
  - Dust/particulate concentration ($\mu g/m^3$)
  - Temperature (°C) and humidity (%)
  - Ground vibration (mm/s)
- **Personnel Safety Devices** (wearable devices with BLE):
  - CO level exposure (ppm)
  - Proximity alert (boolean)
  - Heart rate (bpm)
  - Location (x, y coordinates in meters)
  - Fall detection (boolean)

1.2 Features

- **Operational Modes**:

  - normal: Generates realistic data within defined ranges with configurable variance for natural fluctuations.
  - fault: Injects random faults (e.g. gas spikes, overheating, conveyor jams) with a 20% probability per metric per publishing cycle, simulating critical events.
  - intermittent: Mimics connectivity issues by buffering data during random outages (10–30 seconds) and sending in batches upon reconnection.

- **Configurability**: Controlled via a config.json file for number of devices, publishing frequency, variance, seed, and broker settings.
- **Data Format**: Outputs JSON payloads compatible with IoT platforms.
- **Publishing**: Supports MQTT with QoS 1 for reliable delivery to local brokers (e.g. Mosquitto) or AWS IoT Core.
- **Reproducibility**: Uses a seed value to ensure consistent data generation for testing.
- **Extensibility**: Modular design allows adding new metrics or device types easily.

2. Requirements

- **Python**: 3.8 or higher.
- **Libraries**: Install via pip install paho-mqtt. No additional dependencies required for basic operation.
- **Local MQTT Broker (optional):** Eclipse Mosquitto for local testing, run via Docker on Windows (see Installation). Requires Docker Desktop installed.
- **AWS IoT Core** (optional): For cloud integration, requires an AWS account with:
  - IoT Core thing created.
  - X.509 certificates (client certificate, private key, CA certificate) downloaded.
  - AWS IoT endpoint configured (e.g. your-iot-endpoint.amazonaws.com).
- **Hardware and OS**: for local testing run on standard laptops with Windows OS; for AWS, deploy on edge devices like Raspberry Pi with AWS IoT Greengrass for BLE gateway integration.
- **MQTT Client**: MQTTX for viewing published messages (download from https://mqttx.app/).

3. Installation

A. **Clone or Download**: Save the script as synthetic_generator.py.

B. **Install Dependencies**:

```
pip install paho-mqtt>=2.0.0
```

C. **Set Up MQTT Broker**:

- **Local**: Run an Eclipse Mosquitto broker in Docker on Windows with no authentication:

```
docker run -d --name mosquitto-local -p 1883:1883 -p 9001:9001 eclipse-mosquitto:latest
```

This starts Mosquitto with anonymous access on port 1883 for MQTT and 9001 for WebSocket (optional for MQTTX).

- **AWS IoT Core**: Create a thing in AWS IoT Console, download certificates, and note the endpoint URL.

D. **Create Configuration**: Save the following as config.json in the same directory as the script:

```
{
" num_vehicles ": 2 ,
" num_fixed_assets ": 3 ,
 " num_env_sensors ": 4 , " num_personnel ": 2 ,
" frequency ": 5 , " variance ": 0.1 ,
" seed ": 42 ,
" mode ": " normal",
" broker ": " localhost", " port ": 1883 ,
" aws_endpoint ": null , " cert ": null ,
" key ": null ,
" ca ":  null
}
```

4. Configuration Details

The config.json file controls the generator’s behavior. Key fields:

- *num_vehicles, num_fixed_assets, num_env_sensors, num_personnel: Integer counts for each asset type (e.g. 2 vehicles, 3 fixed assets).*
- *frequency: Seconds between data publications (e.g. 5 for every 5 seconds).*
- *variance: Float (0–1) for random noise in metric values (e.g. 0.1 for ś10% variation).*
- *seed: Integer for reproducible random data (e.g. 42).*
- *mode: String, one of "normal", "fault", or "intermittent".*
- *broker: String, "localhost" for local MQTT or "aws" for AWS IoT Core.*
- *port: Integer, typically 1883 for local, 8883 for AWS IoT Core.*
- *aws_endpoint: String, AWS IoT Core endpoint (e.g. your-iot-endpoint.amazonaws.com) or null for local.*
- *cert, key, ca: File paths to AWS IoT certificates (e.g. cert.pem, key.pem, ca.pem) or null for local.*

If config.json is missing, the script uses default values (see synthetic_generator.py).

5. Usage

A. **Prepare Configuration**:

- For local testing, use the default config.json above.
- For AWS IoT Core:
  - *Set "broker":"aws".*
  - *Set "aws_endpoint":"your-iot-endpoint.amazonaws.com".*
  - *Set "cert", "key", and "ca" to paths of your certificate files.*

    - *Set "port":8883.*

- For fault simulation: Set "mode": "fault". Faults (e.g. engine temp >100, methane >5ppm, conveyor jammed) are injected randomly with a 20% chance per metric per published cycle.
- For intermittent connectivity: Set "mode":"intermittent" to simulate random outages (10–30s) with buffered batch sends.
- For reproducibility: Use the same "seed" value (e.g. 42) across runs.

B. **Run the Script**:

```
python synthetic_generator . py -- config config . json
```

The script reads config.json and starts generating data for all specified devices, publishing to the configured MQTT broker. Press Ctrl+C to stop.

C. **Verify Output**:

- Use the MQTTX client to connect to the local MQTT broker and subscribe to the topic “mining/#” to view messages on the “mining/{device_id}” topic.
- For AWS, check IoT Cores MQTT test client in the AWS IoT Console.

6. Example JSON Payload

Each device publishes a JSON payload like:

```
{
" device_id ": " haul_truck_ 001 ", " type ": " vehicle ",
" timestamp ": "2025 -08 -26 T12 :00:00 Z",
" metrics ": {
" gps": {" lat": 37.7749 , " lon ": -122.4194} ,
" speed ": 25.5 ,
" engine_temp ": 85.2 , " vibration ": 1.2 ,
" fuel_level ": 75.0
},
" status ": " normal"
}
```

7. Fault Injection Details

In "fault" mode, each metric has a 20% chance per publishing cycle to trigger a fault, such as:

- **Vehicles**: Engine temperature >100 (fault_overheat).
- **Fixed Assets**: Operational status set to "jammed" (fault_jam).
- **Environmental Sensors**: Methane >5ppm or CO/CO >50ppm (fault_gas).
- **Personnel Devices**: Fall detection set to true (fault_fall) or proximity alert triggered.

8. Intermittent Mode

In "intermittent" mode, the script simulates connectivity loss:

- 30% chance per publish cycle to enter an outage (10–30s duration).

- Data is buffered locally during outages.
- Buffered data is sent as batch upon reconnection.

9. Testing

A. **Local Broker**:

- Start Mosquitto:

```
docker run -d --name mosquitto-local -p 1883:1883 -p 9001:9001 eclipse-mosquitto:latest
```

- Run: python synthetic_generator.py –config config.json.
- Use MQTTX to connect to localhost:1883 and subscribe to mining/#.

B. **AWS IoT Core**:

- Configure config.json with AWS endpoint and certificate paths.
- Run the script and use AWS IoT Consoles MQTT test client to verify messages on mining/* topics.

C. **Fault Testing**: Set "mode":"fault", monitor for status fields like fault_overheat.

D. **Intermittent Testing**: Set "mode":"intermittent", observe delayed batch sends in MQTT client.

E. **Reproducibility**: Run with same seed (e.g. 42) to verify identical data sequences.

10. Troubleshooting

- **No Data Published**: Check config.json for valid values; ensure frequency > 0.
- **Faults Not Triggering**: Increase publish cycles or verify "mode":"fault".