Project 1:
Distributed computing applications.

Pol Jaimejuan Caubet
Cinta Miralles Jarque

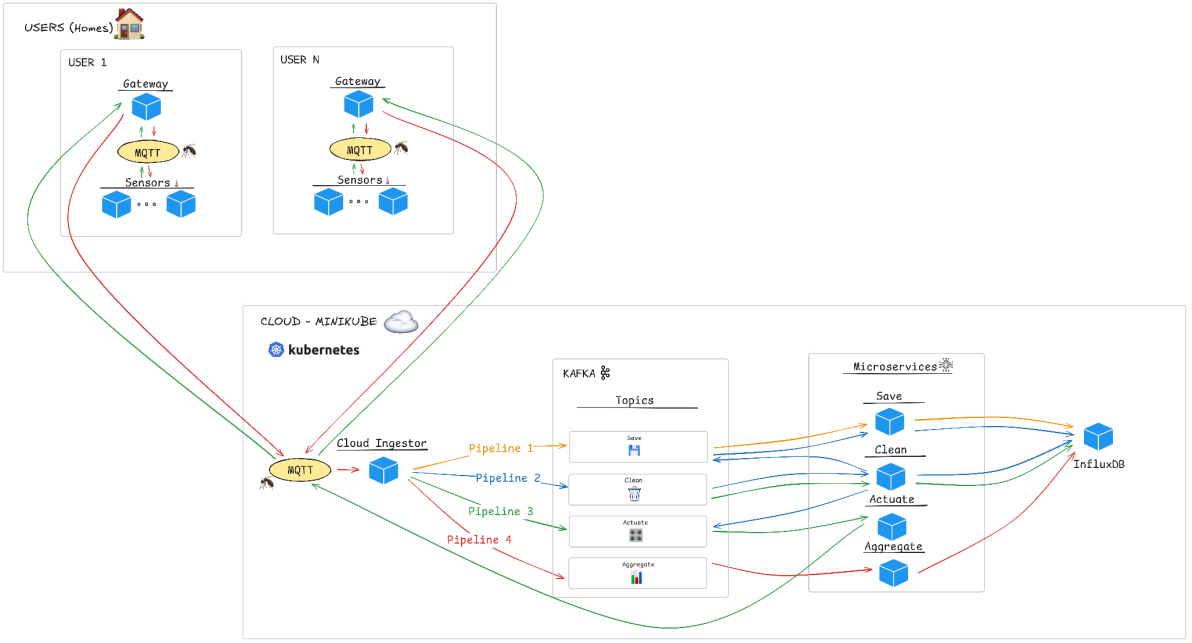
Computació distribuïda
2025/2026
UDL

repository: https://github.com/PolJaimejuanCaubet/minikube-data-stream

Index

1. Description of the components

Our *IoT* system's architectural design exactly mirrors the *framework* specified in the assignment's requirements. The visual representation of this structure is provided in the figure below.



[1]

The components are the following:

1. IoT Devices - sensors

The *entry point* of the system consists of simulated temperature sensors deployed in the *user's home* environment via *Docker Compose*.

- *Role*: These devices simulate environmental readings using a 1:60 time scale (1 second real-time = 1 minute simulation).
- *Implementation*: They are *pre-compiled Docker* containers that generate *JSON payloads* containing a *timestamp* and a *temperature value*.
- *Constraint*: Per project *requirements*, the *sensor source* code is immutable; we treat them as "*black box*" hardware devices.

[1] Figure 1: IoT application architecture. Source: original work.

2. Gateway - gateway.py

The *Gateway* is the critical edge component that bridges the isolated *home network* (*Docker internal_net*) with the external cloud infrastructure.

- *Script Logic*: The *gateway.py* script initializes two distinct *MQTT* clients:
  - *Local Client*: Subscribes to +/temperature to capture data from all room sensors.
  - *Cloud Client*: Connects to the *Mosquitto* broker exposed by *Minikube* (e.g., via NodePort 30001).
- *Bidirectional Traffic*:
  - *Uplink*: It forwards sensor telemetry to the cloud for processing.

3. MQTT Broker - Mosquitto

Deployed within the *Minikube* cluster using the configuration in *servicesk8s/mosquitto-cloud.yml*, this service acts as the public interface of the cloud platform.

- *Function*: It serves as a high-throughput *Pub/Sub Hub*.
- *Connectivity*: It exposes a *port 30001* allowing external *Gateways* to connect securely.
- *Role*: It decouples the edge devices from the heavy internal processing logic, ensuring that temporary failures in the backend do not disconnect the sensors.

4. Ingestior - ingestor.py

This *microservice* acts as the protocol adapter between the synchronous *MQTT* world and the asynchronous *Kafka* streaming pipeline.

- Workflow:
  1. *Subscribes* to the *Cloud MQTT* broker to receive *raw* sensor readings.
  2. *Transforms* the *MQTT* payload into a structured event *object*.
  3. *Produces* this event to the *Kafka* topic named *raw_sensor_data*
- *Architecture Benefit*: By immediately offloading data to *Kafka*, the ingestor ensures fast acknowledgement and prevents data loss if downstream pipelines are slow.

5. Kafka Cluster

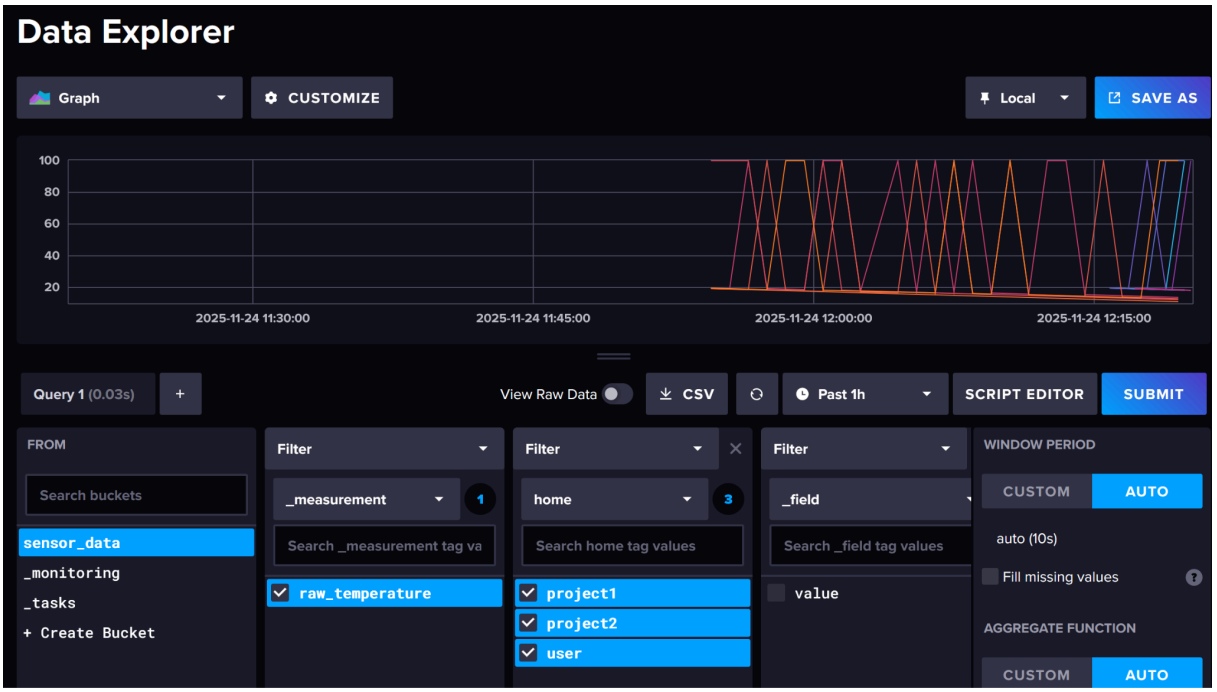
The core of the distributed system is *Wurstmeister Kafka*, deployed via *servicesk8s/kafka.yml* (along with *Zookeeper*). *Kafka* organizes the data flow into distinct stages using topics:

- *raw_sensor_data*: Stores unmodified incoming sensor messages.
- *clean-data*: Stores validated and filtered temperature readings.
- *Consumer Groups*: *Kafka* allows multiple microservices (P1, P2) to read the same data stream in parallel “without conflict” (😅a priori), enabling horizontal scalability.

6. Pipeline P1 - Save Raw Data - save_raw.py

This *service* implements the data auditing requirement.

- *Input*: Consumes from the *raw-data-sensor Kafka* topic.
- *Logic*: It performs no transformations. It writes the exact payload received from the sensor directly into the *InfluxDB* raw bucket.
- *Purpose*: This creates an immutable "*Bronze Layer*" of data, allowing for replayability and debugging in case of errors in the cleaning logic.

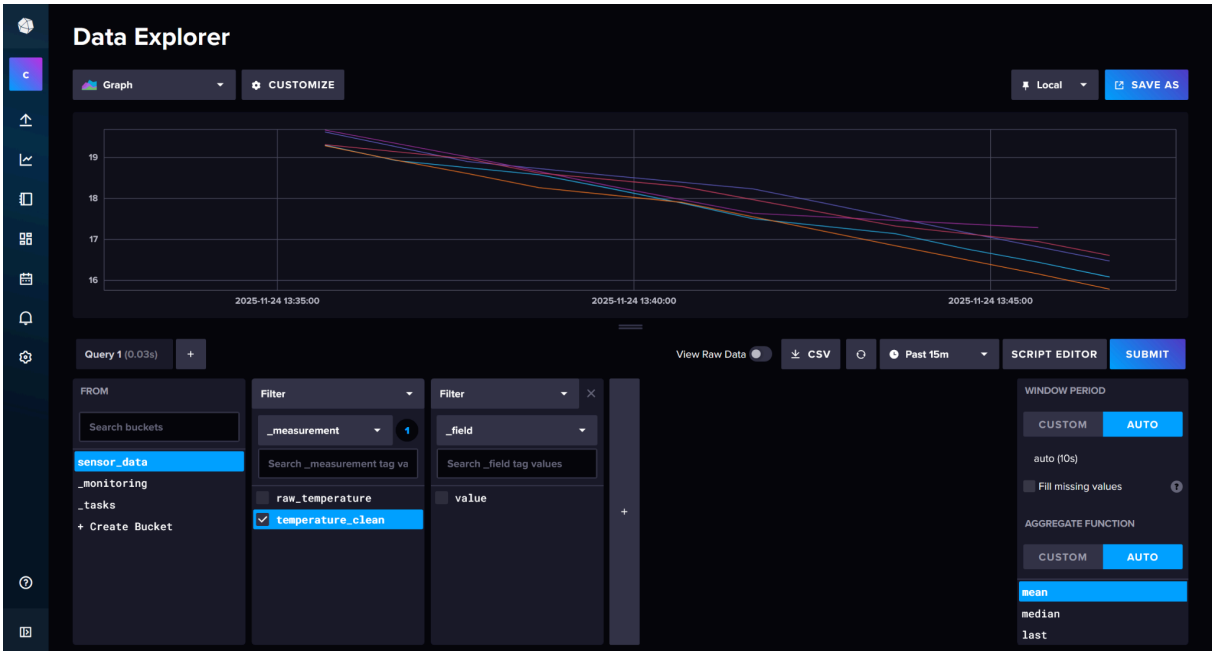


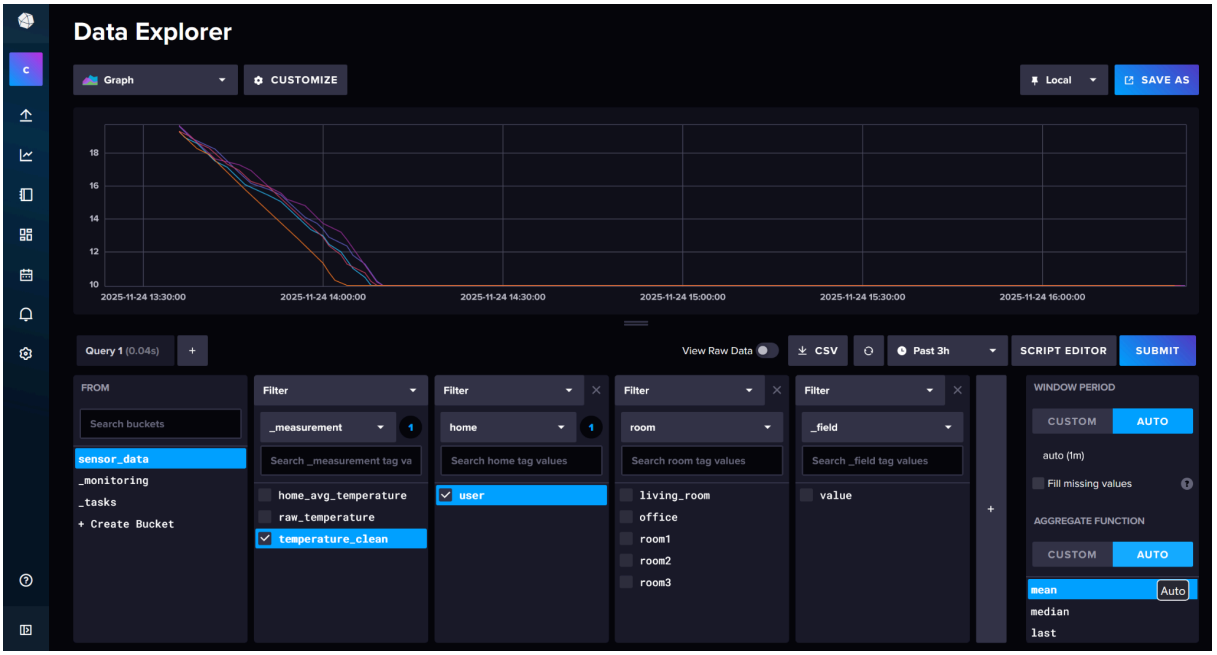
7. Pipeline P2 - Save Clean Data - clean.py

It parses the JSON payload and applies a specific business rule: if the temperature value is exactly 100.0, the data point is discarded as an error.

Dual Output:

1. Valid data is written to the InfluxDB clean bucket for analytics.
2. Valid data is also republished to the clean-data Kafka topic, making it available for the Actuation (P3) and Aggregation (P4) pipelines.





8. Pipeline P3 - Actuate - actuate.py

Decision Logic: It consumes the *clean-data topic* and evaluates the *temperature*:

- < 20°C: Sends a "Start" command ({"status": “1”})
- > 25°C: Sends a "Stop" command ({"status": “0”})

Logs de p3-actuate en p3-actuate-544...

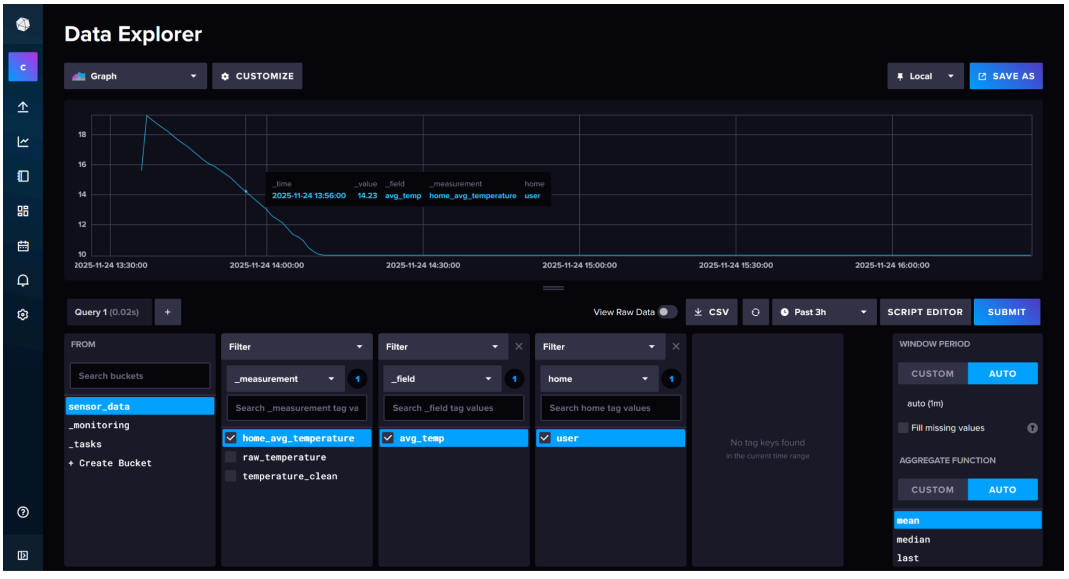
```
user/room3/heatpump/set = {"status": "1"} because temp is: 10.0º
user/living_room/heatpump/set = {"status": "1"} because temp is: 10.0º
user/room2/heatpump/set = {"status": "1"} because temp is: 10.778874751651670º
user/room1/heatpump/set = {"status": "1"} because temp is: 10.788950803205404º
user/office/heatpump/set = {"status": "1"} because temp is: 10.286732040074850º
user/room3/heatpump/set = {"status": "1"} because temp is: 10.249056781357234º
user/room1/heatpump/set = {"status": "1"} because temp is: 11.072992422647530º
user/office/heatpump/set = {"status": "1"} because temp is: 10.581661357693923º
user/living_room/heatpump/set = {"status": "1"} because temp is: 10.285756669006517º
user/room2/heatpump/set = {"status": "1"} because temp is: 11.024887200288056º
user/room1/heatpump/set = {"status": "1"} because temp is: 11.349167474872253º
user/room3/heatpump/set = {"status": "1"} because temp is: 10.487250346788140º
user/living_room/heatpump/set = {"status": "1"} because temp is: 10.536633325095110º
user/room1/heatpump/set = {"status": "1"} because temp is: 11.584329180593228º
user/office/heatpump/set = {"status": "1"} because temp is: 10.828925170127466º
user/room3/heatpump/set = {"status": "1"} because temp is: 10.754039614525660º
user/living_room/heatpump/set = {"status": "1"} because temp is: 10.792832781693730º
user/room1/heatpump/set = {"status": "1"} because temp is: 11.866402145814586º
user/office/heatpump/set = {"status": "1"} because temp is: 11.084822068224770º
user/room2/heatpump/set = {"status": "1"} because temp is: 11.314601435824889º
user/room1/heatpump/set = {"status": "1"} because temp is: 12.159455768206314º
user/office/heatpump/set = {"status": "1"} because temp is: 11.363815947971295º
```

9. Pipeline P4 - Aggregation - aggregate.py

This service handles real-time analytics.

- Logic: It consumes from clean-data, buffers readings in memory, and calculates the mean temperature for a specific user/home.
- Persistence: The calculated averages are stored in the InfluxDB aggregate bucket.
- Use Case: This data is optimized for dashboard visualization, providing a high-level view of home comfort levels without querying millions of raw data points.

-



10. Storage Layer: InfluxDB

The entire persistence layer is built on *InfluxDB*, a specialized *Time-Series Database*, deployed via s*ervicesk8s/influxdb.yaml*.

- *Buckets*: Data is segregated into *raw*, *clean*, and *aggregate* buckets to manage data lifecycle and query efficiency.
- *Data Model*: It uses "*Tags*" (Home, Room) for efficient indexing and "*Fields*" (Temperature Value) for storage.

Database structure includes:

- Measurement:
  - `raw_temperature`
  - `temperature_clean`
  - `home_avg_temperature`

11. Infrastructure: Minikube & Kubernetes

The cloud portion of the project is orchestrating using Kubernetes on Minikube.

- Deployment Strategy: All components (Mosquitto, Kafka, InfluxDB, and Python pipelines) are defined as Kubernetes Deployments and Services in the servicesk8s folder.
- This ensures process isolation, automatic restarts upon failure, and easy scalability, mimicking a production-grade distributed system.
- We've helped with some AI Chatmodels to ensure the yml deployment of each service

2. Main Design Decisions

This chapter outlines the architectural principles, technology selection, and data modeling strategies adopted to build a scalable, IoT monitoring system.

2.1 Architecture Pattern

The system is designed using a microservices approach. Instead of having one single application doing all the work, the system is divided into several small services (Gateway, Cleaner, Actuator, Aggregator).
Each service has its own responsibility and they communicate with each other through messages.

This design gives the system two main benefits:

- Decoupling:
  The sensors only send their data and do not need to know what happens after. Each part of the system can be updated or scaled independently without affecting the others.

- Asynchronous Processing:
  The services do not block each other. Data can be received, cleaned, stored, and processed at the same time, even when the number of messages becomes very high.

2.2 Technology Stack Selection

The technologies used in the system were selected based on the specific needs of IoT data management: a high volume of messages per second, the importance of timestamps, and the requirement for continuous information processing. Each component employs a tool that fits its purpose:

- MQTT: lightweight protocol, ideal for small devices and sensors. It uses a publish/subscribe model, which makes sending data to multiple receivers simple and fast.
- Kafka: enables efficient data transfer between services. It is designed to handle a very high volume of messages. It also provides the ability to reprocess historical data when needed.
- InfluxDB: db optimized for time-based information. Supports fast inserts and efficient queries on metrics such as temperature over time.

2.3 Data Model Design

Data model utilizes the "Tag set" and "Field set" paradigm native to Time-Series Databases.

The schema is defined as follows:

- Measurement (_measurement): Acts as the high-level container similar to a SQL table

    - Examples: raw_temperature, clean_temperature, home_avg_temperature.
- Tags : Used for high-cardinality filtering. These are indexed to allow fast lookups.
    - home: Represents the tenant or user instance.
    - room: Identifies the specific sensor location (e.g., room3, office).
- Fields: The actual telemetry readings.
    - value: float representing the temperature.
- Timestamp: Represents the event generation time, not the ingestion time.

```
point = (
    Point("raw_temperature")
    .tag("home", event["home"])
    .tag("room", event["room"])
    .field("value", float(event["value"]))
    .time(datetime.fromisoformat(event["timestamp"]))
)
```

2.4 Pipeline Architecture

The data pipeline is designed to ensure data integrity and isolation between raw ingestion and business logic. The flow is segmented into Kafka Topics:

- Ingestion Layer: Raw data lands in the system via MQTT and is immediately buffered into Kafka.
- Processing Layer (Stream Processing):
    - Isolation: Separate topics (e.g., raw_sensor_data, clean-data) ensure that a failure in the aggregation service does not impact data cleaning or alerting.
    - Ordering: Utilizing Kafka partition keys (hashed by ‘home’) ensures that temperature readings for a specific home are processed in “supposed” strict chronological order.

3. How to run the project

To run the project, follow the following steps:

Create a minikube instance:

```
minikube start

minikube dashboard
```

Go to the link of the dashboard and check about deployed pods.

Once created the instance, build an ingestor image, load and deploy, kafka, mosquitto and zooekeeper to cluster:

```
docker build -t ingestor ./cloud/ingestor

minikube image load ingestor

kubectl apply -f cloud/servicesk8s/ingestor.yml -f cloud/servicesk8s/kafka.yml -f
cloud/servicesk8s/mosquitto-cloud.yml -f cloud/servicesk8s/zookeeper.yml
```

Wait for the deployment of this 4 pods, then:

```
kubectl port-forward svc/mosquitto-svc 1883:1883

This command is to access the Mosquitto broker inside Minikube from your localhost and
if you have installed MQTT EXPLORER then connect to 1883 port and localhost, there
you'll see the user's connections once they were composed. After next command.
```

DB deployment, wait until deployment finishes tu run the second command:

```
kubectl apply -f cloud/servicesk8s/influx-db.yml

kubectl port-forward svc/influxdb-svc 8086:8086

http://localhost:8086/

user: admin

password: cloudpass
```

Building, loading and deploying pipeline images. Wait until the pod's deployment finishes to build the next image:

```
docker build -t p1-save ./cloud/save_raw

minikube image load p1-save
```

```
kubectl apply -f cloud/servicesk8s/save-raw-data.yml


docker build -t p2-clean ./cloud/clean

minikube image load p2-clean

kubectl apply -f cloud/servicesk8s/clean.yml


docker build -t p3-actuate ./cloud/actuate

minikube image load p3-actuate

kubectl apply -f cloud/servicesk8s/actuate.yml


docker build -t p4-aggregate ./cloud/aggregate

minikube image load p4-aggregate

kubectl apply -f cloud/servicesk8s/aggregate.yml
```

Start all the services defined on the root .yml, second command for create multiple users:

```
docker compose up --build

docker-compose -p <several-projects> up
```

4. References

https://eclipse.dev/paho/files/paho.mqtt.python/html/client.html

https://kafka.apache.org/documentation/

https://minikube.sigs.k8s.io/docs/start/?arch=%2Flinux%2Fx86-64%2Fstable%2Fbinary+download

And some AI ChatModels, for the creation of the services deployments .yml such as kafka, mosquitto etc..

IF YOU WANT TO CHECK THE REPO, OR CLONE IT VISIT, HERE ARE BETTER INSTRUCTIONS ABOUT HOW TO RUN THE PROJECT BECAUSE IT'S ON MARKDOWN FORMATT, MORE READABLE.

https://github.com/PolJaimejuanCaubet/minikube-data-stream