Creating
Dashboards Using Grafana

ABSTRACT:

- Generally in monitoring the data visualization plays a major role.
- The main goal of this document is to get the complete information regarding the exposure of the dashboards from the data that goona extracted.
- The problem in this is to find the errors which is occurred during the process and finding it is so difficult but that is for a minor reason due to the complexity of handling many servers at a time.
- So, this helps to solve those major errors which are for minor reasons.

AGENDA

INTRODUCTION

- **Grafana** is a visualization tool that is used to make some good visual representations through which the data is collected from different data sources.
- In the present documentation we used the **InfluxDB** data source which is a database that stores and shares the data in a time series.
- The data that is generated is industrial-based and is stored according to the protocols and the data is mostly generated and stored in the data logs.
- So, the data gonna extracted from the platform which is stored in a log is the **KEPserverEX** based upon the different protocols and it is a Connectivity platform that is just connects to the servers to extract the data.
- In this project, data follows the MQTT protocol and based upon that, data transfer takes place to the InfluxDB.
- To share the data InfluxDB has its in-built agent i.e., **Telegraf** which is a server between two clients.

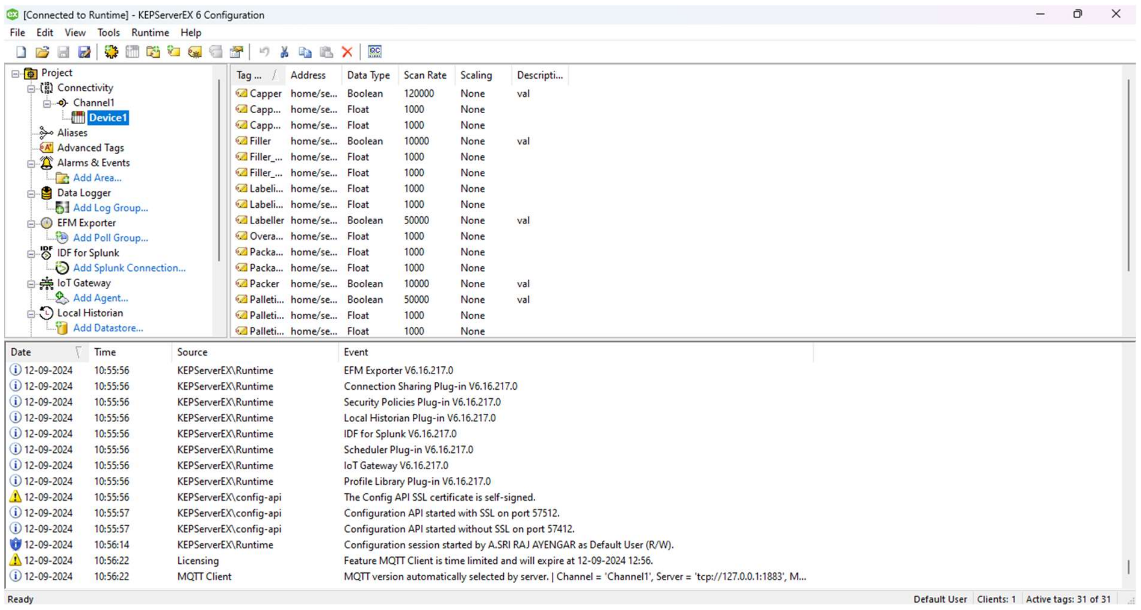
STEPS FROM EXTRACTION TO VISUALIZATION

There are various steps in the process of creating dashboards from the extraction of the data but the major things that are included in the process are:

- Data Extraction
- Data Configuration through Telegraf
- Setting up the data resources in Grafana
- Steps to create Visualization

EXTRACTION OF DATA

- The data stored in the KEPserverEX follow different protocols based upon which the project files or the data logs follow.
- Extraction of data conveys how the data gets extracted i.e., extraction of the protocol that the client of Data follows.
- To know on which protocol the data is stored, make sure to check the properties of the logs or devices which gives the type of protocol that it follows.
- In this project, an MQTT device and a Python JSON payload are used for the generation of tags and the data respectively, so that extraction of data can be done for further steps.
- For MQTT-type devices in KEPservereEX, it stores the data in the form of tags and the data that is stored is transmitted through the MQTT servers, and in this project, the data is transmitted from the Python code through the MQTT server.
- To serve the data to other platforms, it is necessary to get the addresses of the tags in the case of MQTT.
- In this project, 19 tags are created under the MQTT device under a single topic from the 19, 6 tags belong to the status, another 6 belong to the operating time, another 6 are failure count and the remaining 1 tag belongs to the scrap count of the 6 machines respectively.

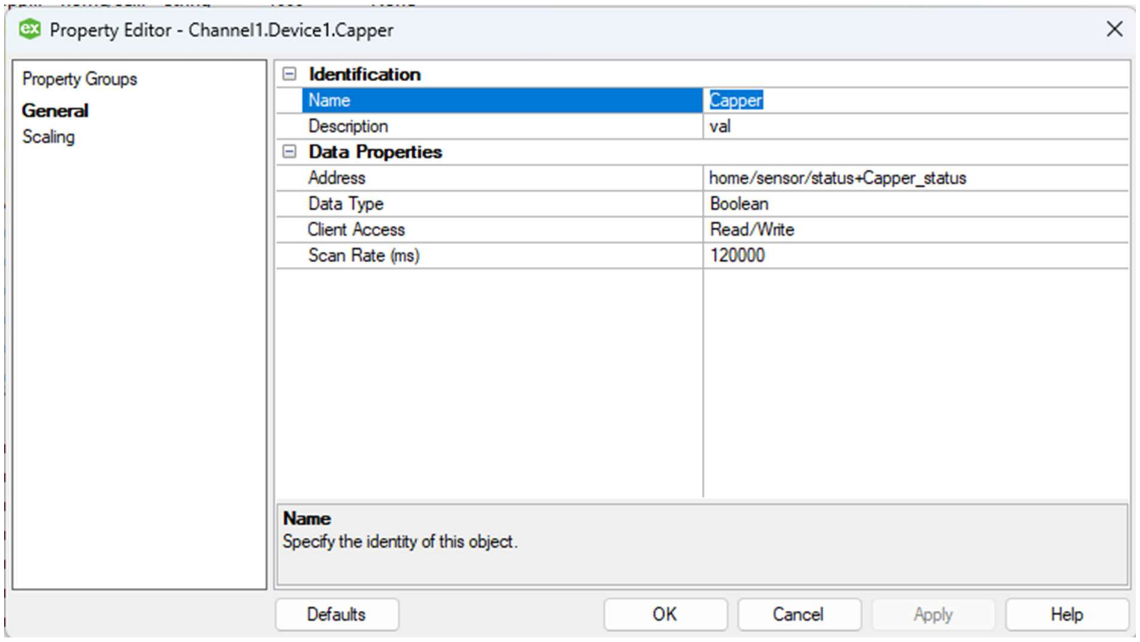


Tag ...	Address	Data Type	Scan Rate	Scaling	Descripti...
Capper	home/se...	Boolean	120000	None	val
Capp...	home/se...	Float	1000	None	
Capp...	home/se...	Float	1000	None	
Filler	home/se...	Boolean	10000	None	val
Filler_...	home/se...	Float	1000	None	
Filler_...	home/se...	Float	1000	None	
Labeli...	home/se...	Float	1000	None	
Labeli...	home/se...	Float	1000	None	
Labeller	home/se...	Boolean	50000	None	val
Overa...	home/se...	Float	1000	None	
Packa...	home/se...	Float	1000	None	
Packa...	home/se...	Float	1000	None	
Packer	home/se...	Boolean	10000	None	val
Palleti...	home/se...	Boolean	50000	None	val
Palleti...	home/se...	Float	1000	None	
Palleti...	home/se...	Float	1000	None	

```
Date        Time      Source                    Event
12-09-2024  10:55:56  KEPServerEX\Runtime       EFM Exporter V6.16.217.0
12-09-2024  10:55:56  KEPServerEX\Runtime       Connection Sharing Plug-in V6.16.217.0
12-09-2024  10:55:56  KEPServerEX\Runtime       Security Policies Plug-in V6.16.217.0
12-09-2024  10:55:56  KEPServerEX\Runtime       Local Historian Plug-in V6.16.217.0
12-09-2024  10:55:56  KEPServerEX\Runtime       IDF for Splunk V6.16.217.0
12-09-2024  10:55:56  KEPServerEX\Runtime       Scheduler Plug-in V6.16.217.0
12-09-2024  10:55:56  KEPServerEX\Runtime       IoT Gateway V6.16.217.0
12-09-2024  10:55:56  KEPServerEX\Runtime       Profile Library Plug-in V6.16.217.0
12-09-2024  10:55:56  KEPServerEX\config-api    The Config API SSL certificate is self-signed.
12-09-2024  10:55:57  KEPServerEX\config-api    Configuration API started with SSL on port 57512.
12-09-2024  10:55:57  KEPServerEX\config-api    Configuration API started without SSL on port 57412.
12-09-2024  10:56:14  KEPServerEX\Runtime       Configuration session started by A.SRI RAJ AYENGAR as Default User (R/W).
12-09-2024  10:56:22  Licensing                 Feature MQTT Client is time limited and will expire at 12-09-2024 12:56.
12-09-2024  10:56:22  MQTT Client               MQTT version automatically selected by server. | Channel = 'Channel1', Server = 'tcp://127.0.0.1:1883', M...
```

Note: For the unlisenced KEPserverEX the lisence of it for MQTT is 2 hours. So, for getting 2 hours of license again and again just do modify or reinstall the KEPserverEX

- Each tag will store a single information which changes dynamically according to time.
- The topic is generally found in the address of the tag in the format of "topic+tag_name" and here "tag_name" gives which information that the tag generates for the MQTT protocol.



Identification	
Name	Capper
Description	val
Data Properties	
Address	home/sensor/status+Capper_status
Data Type	Boolean
Client Access	Read/Write
Scan Rate (ms)	120000

Name
Specify the identity of this object.

Note: What-ever the data that reads or writes according to the scan rate of the ordinal generated data takes place, in this project that is python code and that scan rate is only given in the properties of the tag of the KEPserverEX.

SET UP CONFIGURATION

- To set up configuration with the InfluxDB Database, there is an inbuilt agent named Telegraf.
- To do a general configuration with InfluxDB basic prerequisites of input and output are needed to be given in the Telegraf configuration file.
- In this project the input is MQTT and the output is InfluxDB based upon the input the URL of that particular input is given to connect that official server.
- In the case of configuring the data of the MQTT server, an input topic is given which is present in the address of the tag of KEPserverEX.
- Coming to the output it is necessary to fill in the details of the influx according to the version of the influx, as in this project influx version 2. x is installed it is necessary to give the name of the bucket which is already created and the organization name along with the token.

```
•  # # Configuration for sending metrics to InfluxDB 2.0
•  [[outputs.influxdb_v2]]
•  #   ## The URLs of the InfluxDB cluster nodes.
•  #   ##
•  #   ## Multiple URLs can be specified for a single cluster, only ONE of the
•  #   ## urls will be written to each interval.
•  #   ##   ex: urls = ["https://us-west-2-1.aws.cloud2.influxdata.com"]
•     urls = ["http:localhost:8086"]
•  #
•  #   ## Local address to bind when connecting to the server
•  #   ## If empty or not set, the local address is automatically chosen.
•  #   # local_address = ""
•  #
•  #   ## Token for authentication.
•     token = "*"
•  #
•  #   ## Organization is the name of the organization you wish to write to.
•     organization = "PSPTechhub"
•  #
•  #   ## Destination bucket to write into.
•     bucket = "hst1"
•  #
•  #   ## The value of this tag will be used to determine the bucket.  If this
•  #   ## tag is not set the 'bucket' option is used as the default.
•  #   # bucket_tag = ""
•  #
•  #   ## If true, the bucket tag will not be added to the metric.
•  #   # exclude_bucket_tag = false
•  #
```

```
#   ## Timeout for HTTP messages.
#   # timeout = "5s"
#
#   ## Additional HTTP headers
#   # http_headers = {"X-Special-Header" = "Special-Value"}
#
#   ## HTTP Proxy override, if unset values the standard proxy environment
#   ## variables are consulted to determine which proxy, if any, should be used.
#   # http_proxy = "http://corporate.proxy:3128"
#
#   ## HTTP User-Agent
#   # user_agent = "telegraf"
#
#   ## Content-Encoding for write request body, can be set to "gzip" to
#   ## compress body or "identity" to apply no encoding.
#   # content_encoding = "gzip"
#
#   ## Enable or disable uint support for writing uints influxdb 2.0.
#   # influx_uint_support = false
#
#   ## When true, Telegraf will omit the timestamp on data to allow InfluxDB
#   ## to set the timestamp of the data during ingestion. This is generally NOT
#   ## what you want as it can lead to data points captured at different times
#   ## getting omitted due to similar data.
#   # influx_omit_timestamp = false
#
#   ## HTTP/2 Timeouts
#   ## The following values control the HTTP/2 client's timeouts. These settings
#   ## are generally not required unless a user is seeing issues with client
#   ## disconnects. If a user does see issues, then it is suggested to set these
#   ## values to "15s" for ping timeout and "30s" for read idle timeout and
#   ## retry.
#   ##
#   ## Note that the timer for read_idle_timeout begins at the end of the last
#   ## successful write and not at the beginning of the next write.
#   # ping_timeout = "0s"
#   # read_idle_timeout = "0s"
#
#   ## Optional TLS Config for use on HTTP connections.
#   # tls_ca = "/etc/telegraf/ca.pem"
```

```
•  #   # tls_cert = "/etc/telegraf/cert.pem"
•  #   # tls_key = "/etc/telegraf/key.pem"
•  #   ## Use TLS but skip chain & host verification
•  #   # insecure_skip_verify = false
```

Note: Before going to run the .exe file of telegraf just make sure that the InfluxDB server execution file is running in the background otherwise telegraf execution file throw backs the error. The command for executing influx.exe file after navigating to that path in Cmd of Windows is “influxd”.

- In this way the KEPserverEX connectivity platform get connects with the InfluxDB through the inbuilt Agent Telegraf by transmitting the data through the MQTT server.
- After completing the prerequisites and a basic execution of the telegraf.exe file, a basic configuration takes place which is used to check whether the MQTT server is connected to the database but no data transmission takes place.
- For doing data transmission along with basic configuration, it is necessary to include topic-parsing or field-pivoting codes for the MQTT protocol.
- Topic-parsing will give the value stored in a tag of a topic which is used to extract the data values of a single tag and a single tag will give a single information whereas Field-pivoting will give the values of the different tags of a sensor.
- In the Topic-parsing of Telegraf, pre-requisites that needed to be filled along with the configuration things in the input are to mention the path that navigates to the tag name, the field mentioned which is the name of the value of the information that it transmits and the sensor is given which is the topic again so that it will transmit the values of a single tag.
- In this project, “Field Pivoting” is used. For field pivoting, it is necessary to give different field names, tag names, and paths (there are 19 in this project) along with the configuration code so that it will transmit multiple tag values.

```
[[inputs.mqtt_consumer]]
  servers = ["tcp://127.0.0.1:1883"]
  topics = ["home/sensor/status"]
  data_format = "json"
  json_string_fields = ["Capper_status", "Filler_status",
"Labeling_status", "Packaging_status", "Palletizing_status",
"Wrapping_status"]

[[processors.rename]]
```

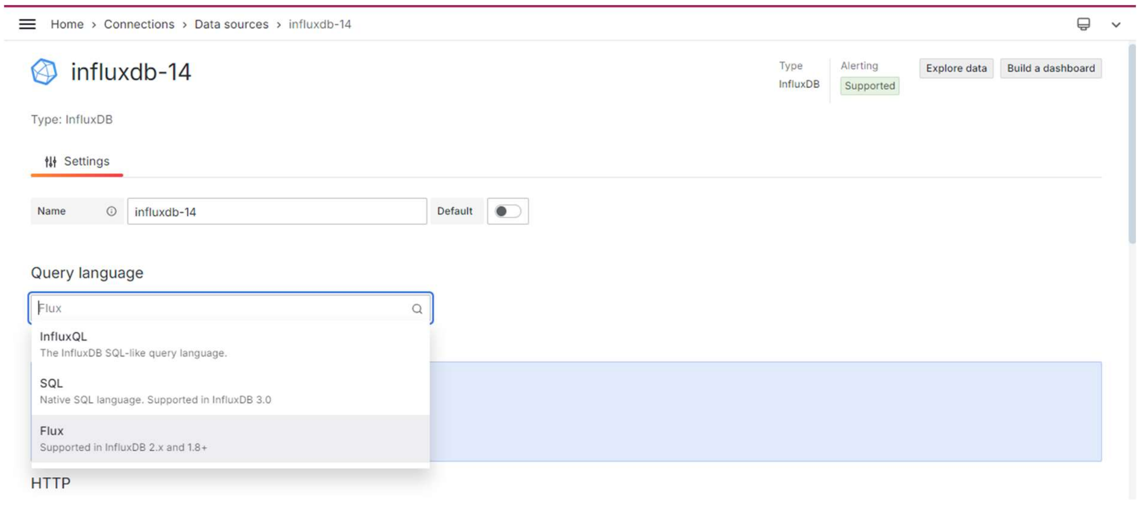
```
  [[processors.rename.replace]]
    measurement = "sensor_status"
    field = "Capper_status"
    dest = "Capper"
  [[processors.rename.replace]]
    measurement = "sensor_status"
    field = "Filler_status"
    dest = "Filler"
  [[processors.rename.replace]]
    measurement = "sensor_status"
    field = "Labeling_status"
    dest = "Labeller"
  [[processors.rename.replace]]
    measurement = "sensor_status"
    field = "Packaging_status"
    dest = "Packer"
  [[processors.rename.replace]]
    measurement = "sensor_status"
    field = "Palletizing_status"
    dest = "Palletizer"
  [[processors.rename.replace]]
    measurement = "sensor_status"
    field = "Wrapping_status"
    dest = "Wrapper"

[[processors.converter]]
  [processors.converter.fields]
    boolean = ["Capper", "Filler", "Labeller", "Packer", "Palletizer", "Wrapper"]
```

Note: The command to execute the telegraf configuration file in Windows Power Shell is ".\telegraf.exe --config telegraf.conf".

- In all the above three cases inputs are different and the output is the same.
- Now the data is stored in the InfluxDB database as a time-series.

SET UP WITH DATA SOURCES IN GRAFANA

- Generally data-sources are used to extract the data used after used to visualize it.
- There are different data sources provided in Grafana to collect the data.
- In this project, the InfluxDB data source is added, and to add that data source, it is necessary to add the InfluxDB data source based on the version installed.
- In this project, InfluxDB version 2. x is used, and the type of influx query builder is selected accordingly with the version of InfluxDB. Therefore, in this type, it is necessary to give prerequisites of bucket and organization names along with the token and URL of the InfluxDB that were previously used to store the data of KEPserverEX.



Name: influxdb-14

Query language: Flux

InfluxQL
The InfluxDB SQL-like query language.

SQL
Native SQL language. Supported in InfluxDB 3.0

Flux
Supported in InfluxDB 2.x and 1.8+

- In this way the InfluxDB data source is added to the Grafana.

STEPS TO CREATE A VISUALIZATION

- Visualization is a representation of the data in different forms.
- The set of visualizations as panels is known as the dashboard.
- In this project, 4 visualizations are used as a single dashboard.
- Coming to the extraction of required visualization firstly it is required to create a dashboard and in that create the panel and add up the data source.
- In this project, 4 panels are created and accordingly the four panels are from the same data source of InfluxDB.
- Coming to the extraction of the data from the source to the visual representation, it is necessary to give the prerequisites to the query section the query section has three types that are given before while adding the query type in adding the data source and the flux query is used in this project.
- In this project four types of flux codes are used:

a. To get the status of the machine:

Filler Machine Status code:

```
from(bucket: "hst1")
  |> range(start: v.timeRangeStart, stop: v.timeRangeStop)  // Adjust the time range as needed
  |> filter(fn: (r) => r._measurement == "mqtt_consumer")
  |> filter(fn: (r) => r._field == "Filler_status")
  |> yield(name: "filler_status")
```

Capper Machine Status code:

```
from(bucket: "hst1")
  |> range(start: v.timeRangeStart, stop: v.timeRangeStop)  // Adjust the time range as needed
  |> filter(fn: (r) => r._measurement == "mqtt_consumer")
  |> filter(fn: (r) => r._field == "Capper_status")
  |> yield(name: "capper_status")
```

Labeling Machine Status code:

```
from(bucket: "hst1")
  |> range(start: v.timeRangeStart, stop: v.timeRangeStop)  // Adjust the time range as needed
  |> filter(fn: (r) => r._measurement == "mqtt_consumer")
```

```
    |> filter(fn: (r) => r._field == "Labeling_status")
    |> yield(name: "labeling_status")

```

Palletizer Machine Status code:

```
from(bucket: "hst1")
    |> range(start: v.timeRangeStart, stop: v.timeRangeStop)  // Adjust the time range as needed
    |> filter(fn: (r) => r._measurement == "mqtt_consumer")
    |> filter(fn: (r) => r._field == "Palletizing_status")
    |> yield(name: "palletizing_status")

```

Packaging Machine Status code:

```
from(bucket: "hst1")
    |> range(start: v.timeRangeStart, stop: v.timeRangeStop)  // Adjust the time range as needed
    |> filter(fn: (r) => r._measurement == "mqtt_consumer")
    |> filter(fn: (r) => r._field == "Packaging_status")
    |> yield(name: "packaging_status")
```

Packaging Machine Status code:

```
from(bucket: "hst1")
    |> range(start: v.timeRangeStart, stop: v.timeRangeStop)  // Adjust the time range as needed
    |> filter(fn: (r) => r._measurement == "mqtt_consumer")
    |> filter(fn: (r) => r._field == "Wrapping_status")
    |> yield(name: "wrapping_status")
```

EXPLANATION:

- "from(bucket: "hst1")" specifies the data source (InfluxDB bucket) from which pulling the data. In this case, the bucket is called "hst1", which contains the MQTT data that has been sent through KepServerEX and Telegraf into InfluxDB.
- "|> range(start: v.timeRangeStart, stop: v.timeRangeStop)" defines the time range for the data query. In Grafana, v.timeRangeStart and v.timeRangeStop are dynamic variables that automatically adjust to the time range selected in the dashboard. For example, if you choose the last 24 hours in Grafana, this range will adjust to only fetch data within that time window.
- "|> filter(fn: (r) => r._measurement == "mqtt_consumer")" ensures that only pulling records from the measurement named "mqtt_consumer". Measurements in InfluxDB are like tables in traditional databases, so this narrows the data down to only the relevant MQTT data related to machine statuses.
- "|> filter(fn: (r) => r._field == "Filler_status") (or other machine-specific fields like Capper_status, Labeling_status, etc.)" specifies which machine's status want to extract. The _field corresponds to the status field (e.g., Filler_status, Capper_status, etc.) in your data. By using different fields, you can extract status information for specific machines.
- "|> map(fn: (r) => ({ r with _value: if r._value == true then "On" else "Off" }))" function allows to transform the data. In this case, it checks the value of the _value field, which is either true or false. This field represents the machine's status:
- If _value == true, the machine is "On".
- If _value == false, the machine is "Off".
- The map() function changes the boolean values (true or false) into human-readable strings ("On" or "Off") for easier interpretation in Grafana.
- "|> yield(name: "filler_status") (or other machine-specific yield names)" function outputs the result of the query. The optional name parameter allows to label of the output. For example, "filler_status" labels the query result as the status of the Filler machine.
- Each query will have a different yield name (capper_status, labeling_status, etc.), so it can easily identify which status belongs to which machine when setting up Grafana panels.

b. To get the OEE, MTBF, and scrap count:

OEE:

Simplified OEE calculation based on failures and total operating time.

FLUX CODE:

```
// Aggregate total operating time
total_operating_time = from(bucket: "hst1")
  |> range(start: v.timeRangeStart, stop: v.timeRangeStop)
  |> filter(fn: (r) => r["_measurement"] == "mqtt_consumer")
  |> filter(fn: (r) => r["_field"] =~ /(.+)_operating_time/)
  |> sum(column: "_value")
  |> group(columns: [])     // Aggregate across all machines
  |> rename(columns: {_value: "total_operating_time"})

// Aggregate failures
failures = from(bucket: "hst1")
  |> range(start: v.timeRangeStart, stop: v.timeRangeStop)
  |> filter(fn: (r) => r["_measurement"] == "mqtt_consumer")
  |> filter(fn: (r) => r["_field"] =~ /(.*)_failure_count/)
  |> sum(column: "_value")
  |> group(columns: [])     // Aggregate across all machines
  |> rename(columns: {_value: "failures"})

// Aggregate overall scrap count
scrap_count = from(bucket: "hst1")
  |> range(start: v.timeRangeStart, stop: v.timeRangeStop)
  |> filter(fn: (r) => r["_measurement"] == "mqtt_consumer")
  |> filter(fn: (r) => r["_field"] == "Overall_scrap_count")
  |> sum(column: "_value")
  |> group(columns: [])     // Aggregate across all machines
  |> rename(columns: {_value: "overall_scrap_count"})

// Combine aggregated results into a single table
combined = union(tables: [total_operating_time, failures, scrap_count])
  |> group(columns: [])
  |> reduce(
    fn: (r, accumulator) => ({
      total_operating_time: accumulator.total_operating_time + (if exists r.total_operating_time then float(v: r.total_operating_time) else 0.0),
      failures: accumulator.failures + (if exists r.failures then float(v: r.failures) else 0.0),
      overall_scrap_count: accumulator.overall_scrap_count + (if exists r.overall_scrap_count then float(v: r.overall_scrap_count) else 0.0)
    }),
    identity: {total_operating_time: 0.0, failures: 0.0, overall_scrap_count: 0.0}
  )

// Calculate OEE
oee = combined
  |> map(fn: (r) => ({
      _time: now(),
      _value: if r.total_operating_time == 0.0 then 0.0 else (1.0 - (r.failures / r.total_operating_time)) * 100.0  // Simplified OEE formula
  }))
  |> rename(columns: {_value: "OEE"})
  |> yield(name: "OEE")

```

EXPLANATION:

- "filter(fn: (r) => r["_field"] =~ /(.+)_operating_time/)" filters fields that end with _operating_time, which represent the operating times of different machines (e.g., Capper_operating_time, Filler_operating_time).
- "sum(column: "_value")" sums up all the operating times across the different machines.
- "group(columns: [])" removes any grouping, meaning the aggregation will occur across all machines.
- "rename(columns: {_value: "total_operating_time"})" renames the result to make it more readable as total_operating_time.

MTBF:

The mean time between failures calculation.

FLUX CODE:

```
// Aggregate total operating time
total_operating_time = from(bucket: "hst1")
  |> range(start: v.timeRangeStart, stop: v.timeRangeStop)
  |> filter(fn: (r) => r["_measurement"] == "mqtt_consumer")
  |> filter(fn: (r) => r["_field"] =~ /(.+)_operating_time/)
  |> sum(column: "_value")
  |> group(columns: [])     // Aggregate across all machines
  |> rename(columns: {_value: "total_operating_time"})

// Aggregate failures
failures = from(bucket: "hst1")
  |> range(start: v.timeRangeStart, stop: v.timeRangeStop)
  |> filter(fn: (r) => r["_measurement"] == "mqtt_consumer")
  |> filter(fn: (r) => r["_field"] =~ /(.*)_failure_count/)
  |> sum(column: "_value")
  |> group(columns: [])     // Aggregate across all machines
  |> rename(columns: {_value: "failures"})

// Combine aggregated results into a single table
combined = union(tables: [total_operating_time, failures])
  |> group(columns: [])
  |> reduce(
    fn: (r, accumulator) => ({  // Changed the order of arguments
      total_operating_time: accumulator.total_operating_time + (if exists r.total_operating_time then float(v: r.total_operating_time) else 0.0),
      failures: accumulator.failures + (if exists r.failures then float(v: r.failures) else 0.0)
    }),
    identity: {total_operating_time: 0.0, failures: 0.0}
  )

// Calculate MTBF from aggregated values
mtbf = combined
  |> map(fn: (r) => ({
      _time: now(),
      _value: if r.failures == 0.0 then 0.0 else r.total_operating_time / r.failures
  }))
  |> yield(name: "MTBF")
```

EXPLANATION:

- "filter(fn: (r) => r["_field"] =~ /(.*)_failure_count/)" filters for fields ending with _failure_count to gather the failure counts for each machine.
- "sum(column: "_value")" sums all failure counts across all machines.
- "group(columns: [])" aggregates the results across all machines.
- "rename(columns: {_value: "failures"})" renames the sum to failures.

Scrap Count:

The sum of the overall scrap count.

FLUX CODE:

```
from(bucket: "hst1")
  |> range(start: v.timeRangeStart, stop: v.timeRangeStop)
  |> filter(fn: (r) =>
    r["_field"] == "Capper_failure_count" or
    r["_field"] == "Filler_failure_count" or
    r["_field"] == "Labeling_failure_count" or
    r["_field"] == "Packaging_failure_count" or
    r["_field"] == "Palletizing_failure_count" or
    r["_field"] == "Wrapping_failure_count")
  |> group()
  |> sum(column: "_value")
  |> rename(columns: {_value: "Total_scrap_count"})  // Renaming for better readability

```

EXPLANATION:

- "filter(fn: (r) => r["_field"] == "Overall_scrap_count")" filters for the field representing the overall scrap count.
- "sum(column: "_value")" sums up the total scrap count.
- "group(columns: [])" aggregates the scrap count across all machines.
- "rename(columns: {_value: "overall_scrap_count"})" renames the column to make it clear it represents the overall_scrap_count.

c. To get Machine Run time:

FLUX CODE:

```
from(bucket: "hst1")
  |> range(start: v.timeRangeStart, stop: v.timeRangeStop)  // Adjust the time range as needed
  |> filter(fn: (r) => r._measurement == "mqtt_consumer")
  |> filter(fn: (r) => r._field =~ /^(Capper_status|Filler_status|Labeling_status|Packaging_status|Palletizing_status|Wrapping_status)$/)
  |> filter(fn: (r) => r._value == true)  // Consider only "on" (true) values
  |> group(columns: ["_field"])  // Group by the status fields
  |> count(column: "_value")  // Count the number of true values
  |> map(fn: (r) => ({ r with _value: r._value * 30 }))  // Convert counts to runtime in minutes
  |> group()  // Optional: Ungroup to aggregate overall runtime if needed
  |> yield(name: "runtime_count")

```

EXPLANATION:

- "filter()" function is used to filter records based on the field names. In this case, it filters for failure counts of specific machines "Capper_failure_count", "Filler_failure_count", "Labeling_failure_count", "Packaging_failure_count", "Palletizing_failure_count", "Wrapping_failure_count".
- These fields correspond to the failure counts for each machine. The or operators ensure that only records where the field matches one of these specific machine failure counts are kept.
- "group()" groups the data by the default behavior (i.e., based on the measurement and the tag columns). In this context, it's grouping the results across all machines.
- "sum(column: "_value")" sums the values in the _value column, which represents the failure counts for each machine. The result will give the total number of failures across all machines during the specified time range.
- "rename(columns: {_value: "Total_scrap_count"})" renames the summed failure count result to Total_scrap_count to make it more readable in the output. Even though this is summing failure counts, it's treating them as a measure of scrap, likely implying that each failure corresponds to a scrap unit in the context of production.

d. To get “Status and Work order Quality”:

FLUX CODE:

```
from(bucket: "hst1")
  |> range(start: v.timeRangeStart, stop: v.timeRangeStop)
  |> filter(fn: (r) => r["_measurement"] == "mqtt_consumer")  // Adjust measurement name if needed
  |> limit(n: 10)
  |> yield(name: "sample_data")
```

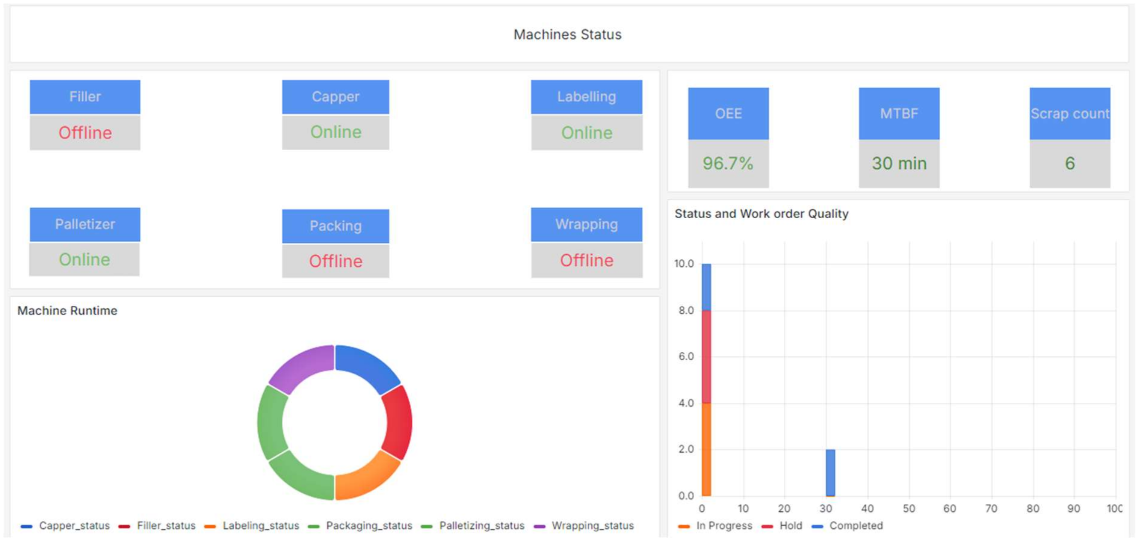
EXPLANATION:

- "limit(n: 10)" function limits the result set to 10 records. It is useful when only want to see a sample or a small subset of data.

Panel settings:

- To name the title for the panel go to panel settings which are present at the right end and create a panel title.
- In the case of giving status in the first panel (Status of the machine) just make search for the option mapping in the after scrolling the panel options at the right end and just map the values like in this case value “true” is “Online” and “false” is “Offline”.
- While going to edit the panel just select the visualization on which it is gonna prepared which is present on the right top.
- Now to change the colour of the value or to change the properties of their own wish just select the Overrides which are also present down to the panel settings and just select the query type of which property is to be changed.
- After all the things it is mandatory to save them as an overall dashboard.

After all the codes are executed the final dashboard will be:


Machines Status

Filler Offline

Capper Online

Labelling Online

Palletizer Online

Packing Offline

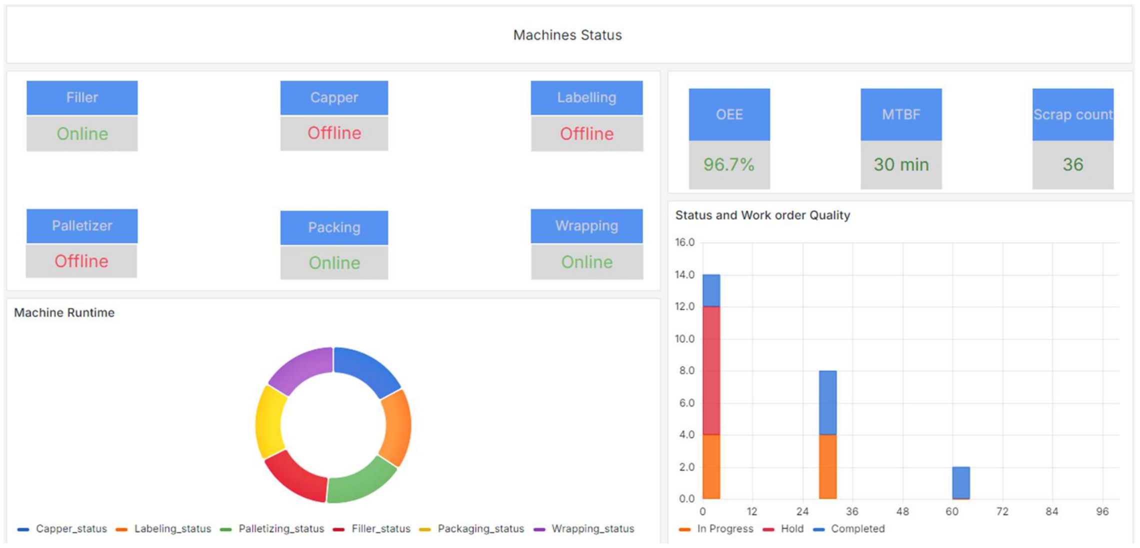
Wrapping Offline

OEE 96.7%

MTBF 30 min

Scrap count 6

The online status will update for every 30 minutes, so the dashboard will be:


Machines Status

Filler Online

Capper Offline

Labelling Offline

Palletizer Offline

Packing Online

Wrapping Online

OEE 96.7%

MTBF 30 min

Scrap count 36

Note: Make sure to check whether the telegraf sever is running or to execute the .exe file of telegraf if the values are to be updated in the Grafana at present.

CONCLUSION:

- To get the data from any platform it is mandatory to follow the prequel steps that guides how the connectivity platforms or servers work.
- So, making dashboard is not only the creating the dashboard but also to gather the data which are mandatory for the visualizations.
- Concluding that by following these steps can reach the final assumed dashboards.

RESULT:

Got assumed dashboard successfully after following all the previous notes from extraction of the data to the creating dashboard.