# A Factory IoT instrumentation data stream, from source manipulated using Pyflink outputting into Kafka or Fluss.

IoT JSON documents stream from source via Kafka topics, consumed by Apache Flink, using PyFlink, flattened or aggregated/windowed and published back to Kakfa or sinked into Fluss.

(28 May 2025)

**Overview**

In a previous [blog](https://github.com/georgelza/DataPipeline-Kafka_Fluss_Paimon-Basic.git) our datastream came through [Apache Flink](https://alibaba.github.io/fluss-docs/) environment into Fluss as our near real time streamhouse datastore (referred to as lakehouse tier in Fluss).

This time round, we will again source from the same Confluent Kafka topics, create Apache Flink tables, but now defined in Apache Flink using Pyflink framework. In Apache Pyflink we will #1 flatten the IoT JSON document and #2 run a 1-minute windowing tumbled window where we count the measurements, determine a minimum, average and maximum value per sensor:

* For #1 we will first publish the flattened IoT document back to our Confluent environment into3 topics.
* For #1 we will push the flattened IoT document to our Fluss environment into a single table.
* For #2 we will run a 1-minute tumbling window, which will be stored in our Fluss defined table.

Our Fluss environment is as per the previous [blog](https://github.com/georgelza/DataPipeline-Kafka_Fluss_Paimon-Basic.git), I simply did not include/configure the lakehouse storage (Apache Paimon on HDFS) for these labs as we’ve shown that working in the previous blog.

Our Data is generated again by

* *<root>/app\_io1/site1.sh*
* *<root>/app\_io2/site2.sh*
* *<root>/app\_io3/site3.sh*
* The first *app\_iot1* creates our simplest IoT JSON payload.
  + This is accomplished by setting *TSHUMAN, STRUCMOD & DEVICETYPE* = 0
* In *app\_iot2* we extend the payload to include *TSHUMAN*=1 and *STRUCMOD*=1.
  + This adds a human readable date field and the location object to the metadata tag section.
* In *app\_iot3* we go one step further and add *DEVICETYPE=1* to the payload.
  + This adds a text string defining the device type.

For catalog services we will be using the [Apache Hive](https://hive.apache.org/)’s and their [Metastore](https://cwiki.apache.org/confluence/display/hive/design#Design-Metastore) functionality as created in a previous blogs (but with a little version update applied recently).

For those that have been following my previous blogs, you will notice I’ve upgraded my [Confluent](https://www.confluent.io/) Kafka Cluster (now 7.9.1) and the [Apache Flink](https://flink.apache.org/) environment (now 1.20.1). The [Apache Paimon](https://paimon.apache.org/) stack has also been upgraded to 0.9.0, Fluss 0.6.0

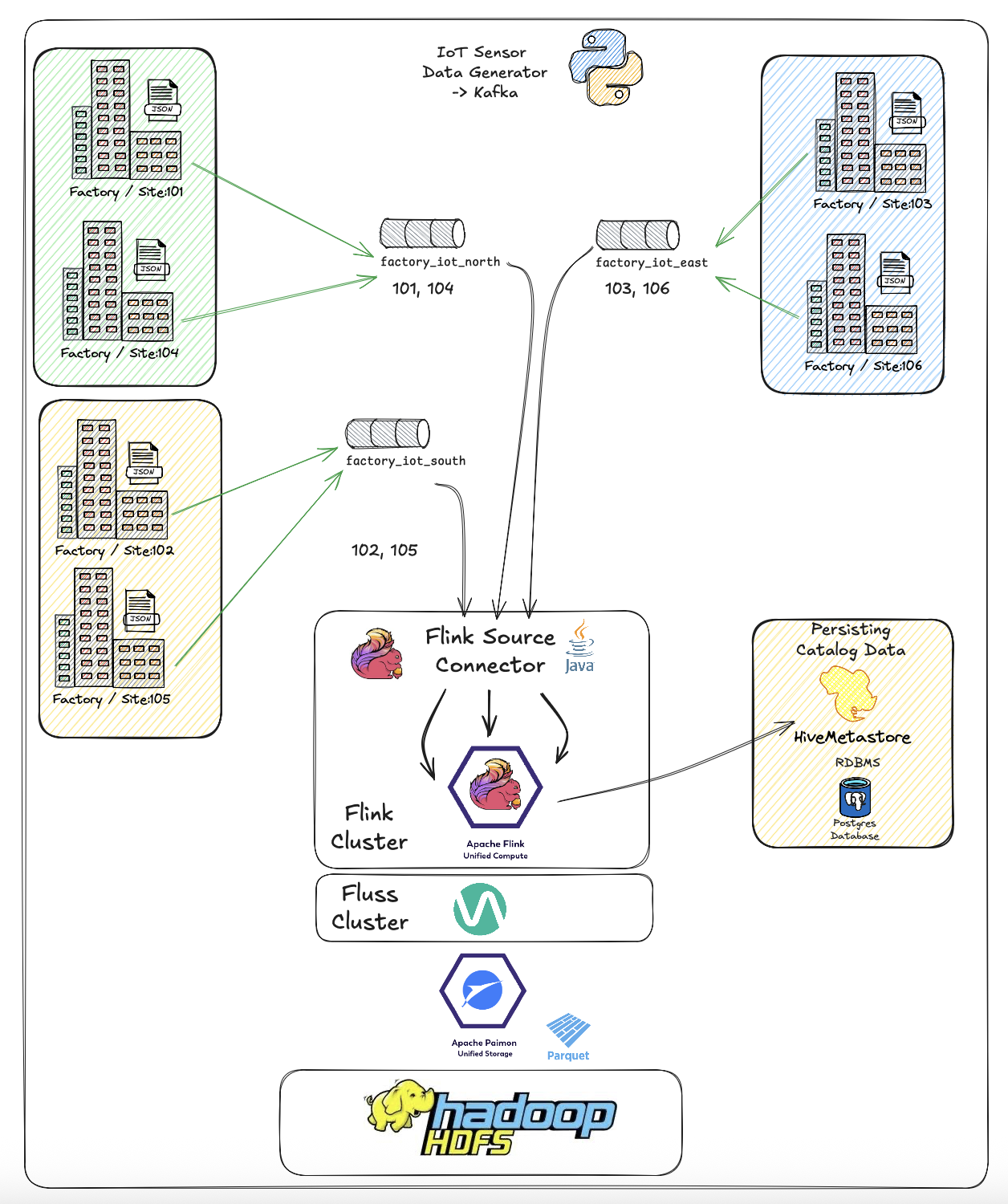
As always, all the code can be found in the [GIT repository](https://github.com/georgelza/DataPipeline-Kafka_Fluss_Paimon-PyFlink.git), and yes, we’re still using a substantial amount of *Makefiles*, *Docker-compose.yml* and *Dockerfile’s*.

Ok, back to why we’re here.

We will use our Python application to create a IoT data stream as if we’re reading sensors in factories fitted to machinery. These “[JSON](https://www.json.org/)” documents will be pushed onto our Kafka Topic called *factory\_iot*. The topic “*key”* is set to the *siteId* of the various factories.

We again have our three regions, each with 2 factories (sites), each factory has many machines (devices) and each machine is instrumented with multiple sensors. They are distributed as:

* North *<root>/app\_iot1*
  + siteId 101
  + siteId 104
* South <*root>/app\_iot2*
  + siteId 102
  + siteId 105
* East <*root>/app\_iot3*
  + siteId 103
  + siteId 106



To start navigate to the *<root>* folder and read the *README.md* file. This will give a similar overview as per above, that will then direct you to build the basic scaffolding using (<*root>/infrastructure/),* after which you will be directed to *<root>/devlab0/README.md* instructing you how to build and run the various examples.

Our first region has been naughty, and not all the required detail is populated into the IoT JSON document.

Executing */<root>/app\_iot1/site1.sh>* will thus produce the below document.

{

"ts" : 123421452622,

"metadata" : {

"siteId" : 1009,

"deviceId" : 1042,

"sensorId" : 10180,

"unit" : "Psi"

},

"measurement" : 1013.3997

}

Below we have the second region, they seem to have read the documentation/expectations a bit more and populated more of the fields of our IoT JSON document.

Region South data is generated using *<root>/app\_iot2/site2.sh* application. This as you can see below adds the *ts\_human* and *location* object to the payload.

{

"ts" : 123421452622,

"metadata" : {

"siteId" : 1009,

"deviceId" : 1042,

"sensorId" : 10180,

"unit" : "Psi",

"ts\_human" : "2024-10-02T00:00:00.869Z",

"location": {

"latitude": -26.195246,

"longitude": 28.034088

}

},

"measurement" : 1013.3997

}

And lastly as we’re feeling very lucky, Region East paid attention and they are providing us a complete IoT Document. To generate their payload, execute *<root>/app\_iot3/site3.sh* application which will add the *deviceType* field to the metadata tag.

{

"timestamp" : "2024-10-02T00:00:00.869Z",

"metadata" : {

"siteId" : 1009,

"deviceId" : 1042,

"sensorId" : 10180,

"unit" : "Psi",

"ts\_human" : "2024-10-02T00:00:00.869Z",

"location": {

"latitude": -26.195246,

"longitude": 28.034088

},

"deviceType" : "Oil Pump",

},

"measurement" : 1013.3997

}

So we now have an overview of 3 payloads and how to generate them at the source end how we will get the data into our Confluent Kafka environment.

Instead of creating a table inside Apache Flink using FlinkSQL, we will use PyFlink (which is the ability of running Python jobs directly on the Flink cluster).

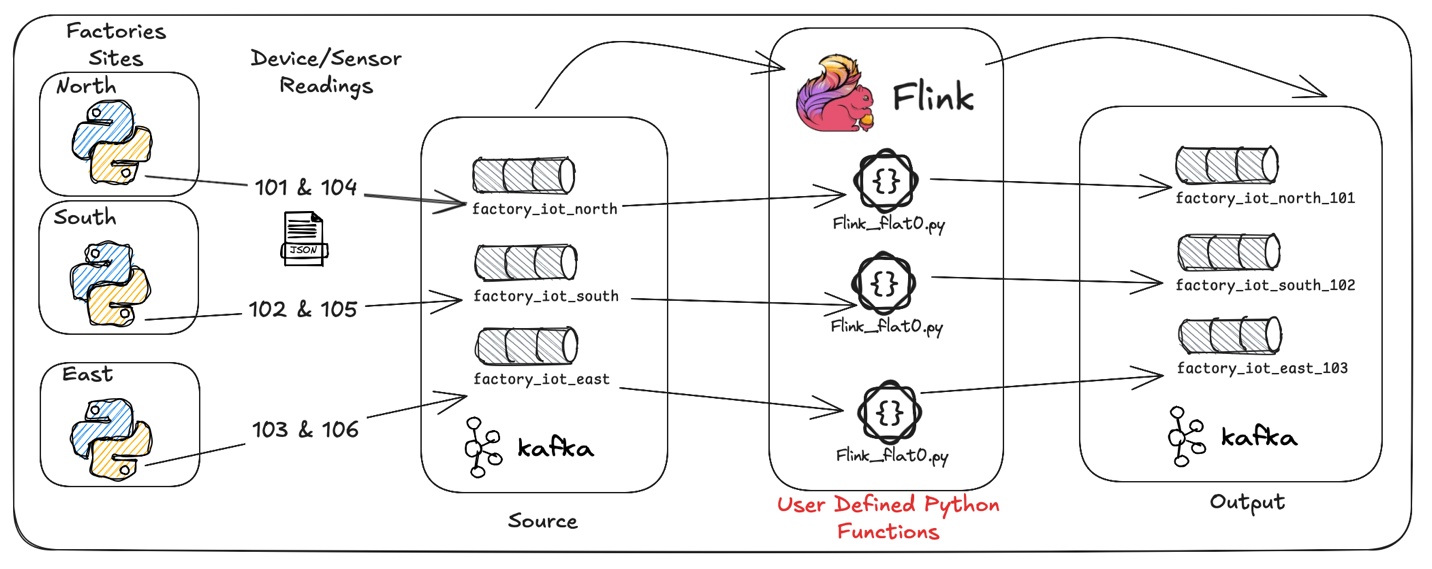
We got 3 examples/labs as per above, each building on the previous, a bit more complex as we go.

**Lab 1**

First we will use <*root>/devlab0/pyFlink/flink\_flat0.py* which will source from Kafka, flatten the payload and then post it back to 3 Kafka topics.

* *factory\_iot\_north\_101*
* *factory\_iot\_south\_102*
* *factory\_iot\_east\_103*

NOTE: As each PyFlink job is running in its own space and we’re creating a temporary source and target tables in PyFlink, each which are configured using a *connector=”kafka” setting* we can re-use the same source table name as *kafka\_source* and output as *kafka\_target* in the below examples.



Kafka\_Source

source\_table\_creation\_sql = f"""

CREATE OR REPLACE TEMPORARY TABLE kafka\_source (

ts BIGINT

,metadata ROW<

siteId INTEGER

,deviceId INTEGER

,sensorId INTEGER

,unit STRING

,ts\_human STRING

,location ROW<

latitude DOUBLE

,longitude DOUBLE

>

,deviceType STRING

>

,measurement DOUBLE

,ts\_wm AS TO\_TIMESTAMP\_LTZ(ts, 3)

,WATERMARK FOR ts\_wm AS ts\_wm - INTERVAL '1' MINUTE

) WITH (

'connector' = 'kafka',

'topic' = '{input\_kafka\_topic}',

'properties.bootstrap.servers' = '{bootstrap\_servers}',

'properties.group.id' = 'flat0\_{site\_id\_filter}',

'scan.startup.mode' = 'earliest-offset',

'format' = 'json',

'json.fail-on-missing-field' = 'false',

'json.ignore-parse-errors' = 'true'

);

"""

Kafka\_Target

output\_table\_creation\_sql = f"""

CREATE OR REPLACE TEMPORARY TABLE kafka\_target (

ts BIGINT

,siteId INTEGER

,deviceId INTEGER

,sensorId INTEGER

,unit STRING

,ts\_human STRING

,latitude DOUBLE

,longitude DOUBLE

,deviceType STRING

,measurement DOUBLE

,ts\_wm TIMESTAMP\_LTZ(3)

) WITH (

'connector' = 'kafka',

'topic' = '{output\_kafka\_topic}',

'properties.bootstrap.servers' = '{bootstrap\_servers}',

'format' = 'json',

'json.fail-on-missing-field' = 'false',

'json.ignore-parse-errors' = 'true'

);

"""

The above common input and output pattern is re-used in the other labs below.

You can see <*root>/devlab0/pyFlink/flink\_flat0.cmd* for commands to run the above 3 PyFlink streams for lab 1.

This is to be executed at the command prompt in the *jobmanager*, this can be reached by executing “*make jm”* in the *<root>/devlab0/* directory.

/opt/flink/bin/flink run \

-m jobmanager:8081 \

-py /pyapp/flink\_flat0.py \

-j /opt/flink/lib/flink/flink-sql-connector-kafka-3.3.0-1.20.jar \

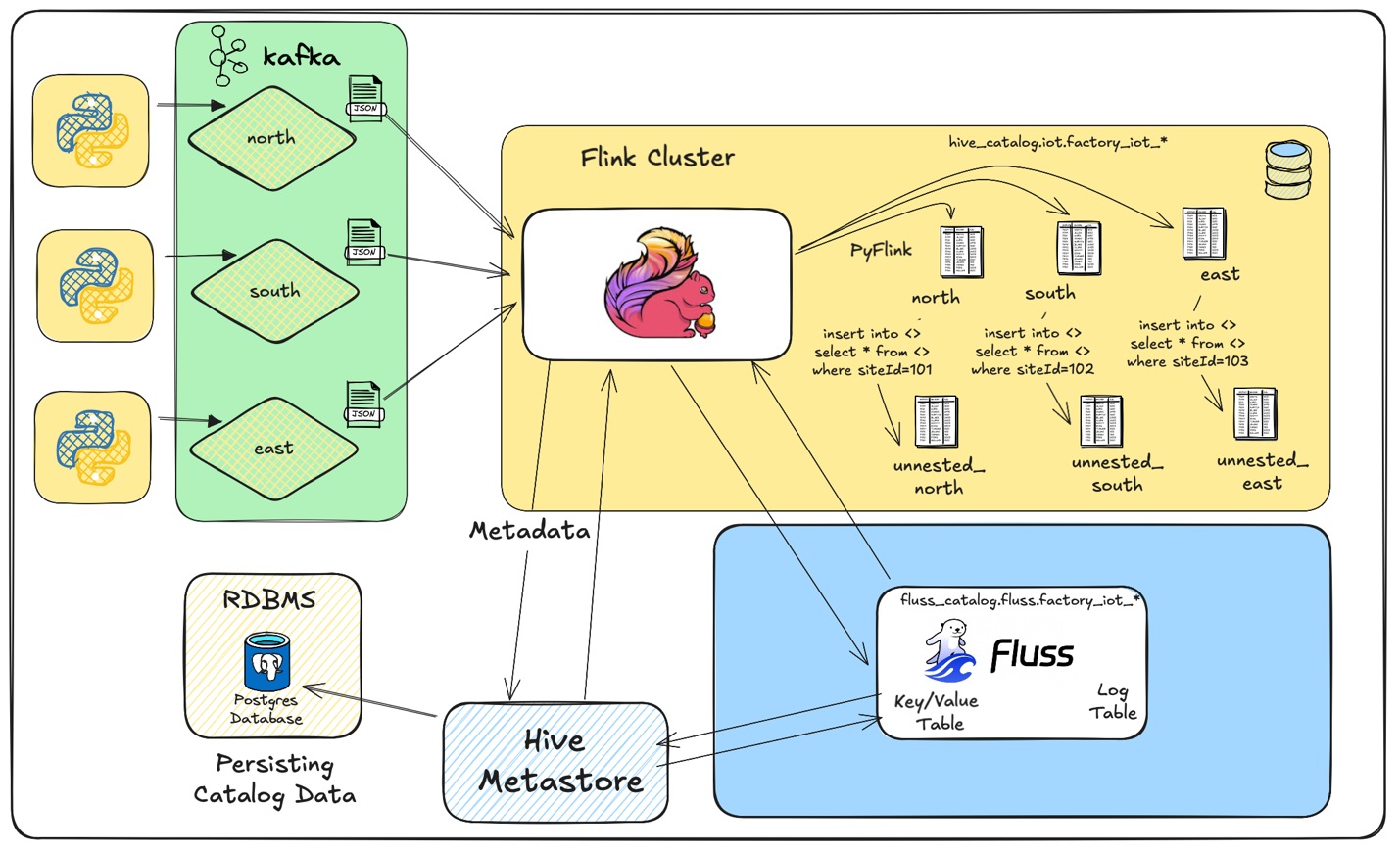
--siteId 101 \

--source factory\_iot\_north

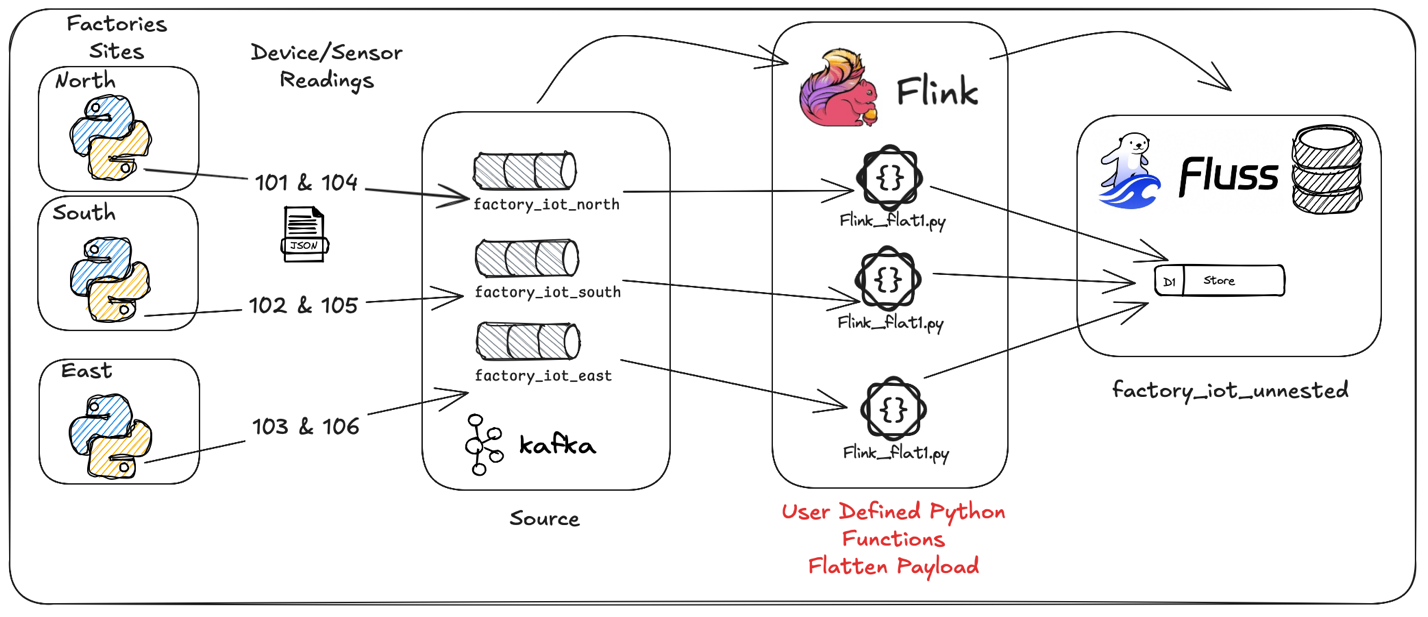
This examples output the flattened values to the below Kafka topics which can be inspected via the Confluent control Center.

**For Lab 2 and Lab 3**

For the following 2 labs we will output our data to the Fluss cluster.



**Lab 2**



First, we will redo the flattening as per previous, but this time we’re going to output to Fluss. For this we will be using <*root>/devlab0/pyFlink//flink\_flat1.py,* we changed this just a tiny bit as we want to push our flattened payload into a single fluss table *fluss\_catalog.fluss.factory\_iot\_unnested.*

You can see <*root>/devlab0/pyFlink/flink\_flat1.cmd* for commands to run the above 3 PyFlink streams for lab 2.

/opt/flink/bin/flink run \

-m jobmanager:8081 \

-py /pyapp/flink\_flat1.py \

-j /opt/flink/lib/flink/flink-sql-connector-kafka-3.3.0-1.20.jar \

--siteId 101 \

--source factory\_iot\_north

The above can be executed at the command prompt in the *jobmanager*, this can be reached by executing “*make jm”* in the *<root>/devlab0/* directory.

For this example, we will output the values to a single Fluss based table *fluss\_catalog.fluss.factory\_iot\_unnested.*

If you want to see the data in the Fluss target table you can execute <*root>/devlab0/pyFlink/flink\_flat1.sql* in a Flink SQL window.

SET 'sql-client.execution.result-mode' = 'tableau';

SET 'parallelism.default' = '2';

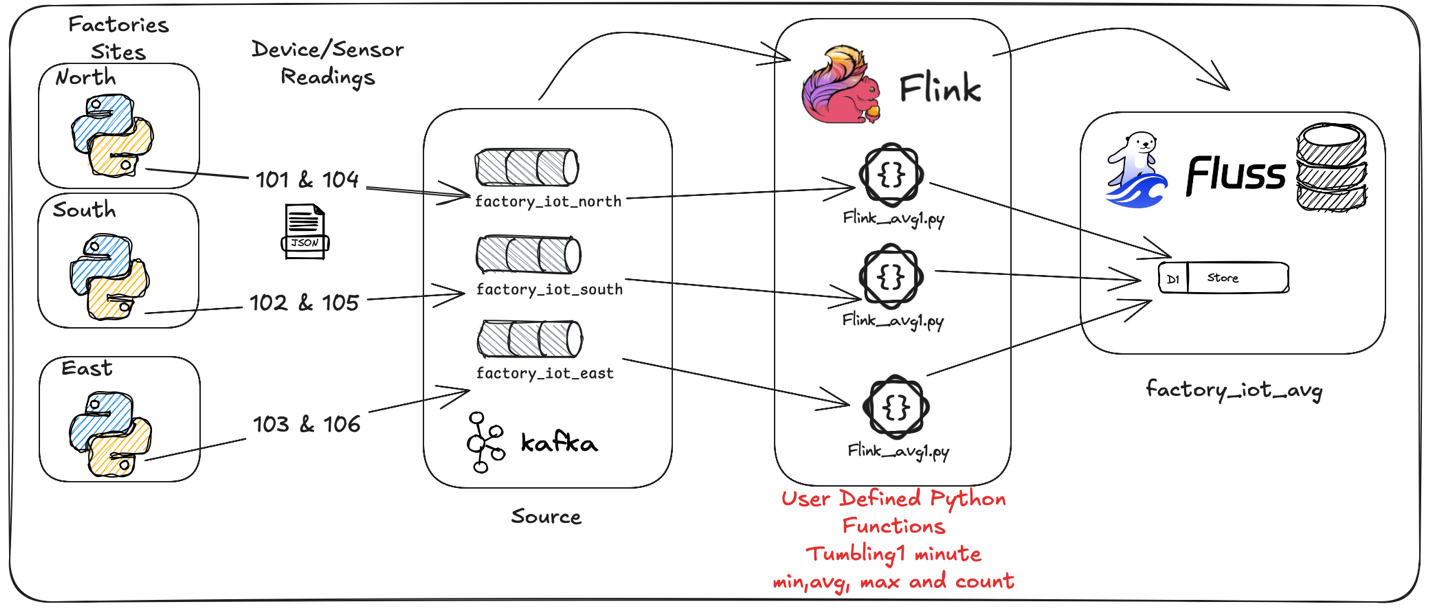
SET 'sql-client.verbose' = 'true';

SET 'execution.runtime-mode' = 'streaming';

select ts, siteId, deviceId, sensorId, measurement from fluss\_catalog.fluss.factory\_iot\_unnested;

The above sql can be executed by opening the *FlinkSql* client, this is accomplished by executing “*make fsql”* in the *<root>/devlab0/* directory.

**Lab 3**



Lastly using <*root>/devlab0/pyFlink/flink\_avg1.py,* we will now use Flink’s tumbling Window functionality to count the number of measurements, determine the min measurement, average measurement and the max measurement value per a 1-minute window for each *sensorId*.

You can see <*root>/devlab0/pyFlink/flink\_avg1.cmd* for commands to run the above 3 PyFlink streams for lab 3.

/opt/flink/bin/flink run \

-m jobmanager:8081 \

-py /pyapp/flink\_avg1.py \

-j /opt/flink/lib/flink/flink-sql-connector-kafka-3.3.0-1.20.jar \

--siteId 101 \

--source factory\_iot\_north

This is to be executed at the command prompt in the *jobmanager*, this can be reached by executing “*make jm”* in the *<root>/devlab0/* directory.

For this example, we will output the values to a single Fluss based table *fluss\_catalog.fluss.factory\_iot\_avg.*

If you want to see the data in the Fluss target you can execute <*root>/devlab0/pyFlink/flink\_avg1.sql* in a Flink SQL window.

SET 'sql-client.execution.result-mode' = 'tableau';

SET 'parallelism.default' = '2';

SET 'sql-client.verbose' = 'true';

SET 'execution.runtime-mode' = 'streaming';

select siteId, deviceId, sensorId, measurement\_count, avg\_measurement, window\_start, window\_end from fluss\_catalog.fluss.factory\_iot\_avg;

The above sql can be executed by opening the *FlinkSql* client, this is accomplished by executing “*make fsql”* in the *<root>/devlab0/* directory.

So, in summary, we build a data pipeline from our source factory, into Apache Kafka Topic, flattened by Apache Flink using Pyflink based function written using Python. We then send our data either back to Kafka (lab 1) into new topics or into Fluss tables (lab 2 and 3).

Why all of this… Well in the world of industrial systems, we can using this PyFlink layer running on our Apache Flink cluster compare real time metrics against known good values, swim lanes using ML to raise alarms before a problem is experienced.

In the world of finances, we can use the same pattern to profile an inbound stream of financial transactions and use a Fraud based AI engines to fraud score for a transaction against known previous behaviour of the account holder, outputting alarms in near real time, all while outputting all our data to our Fluss platform for further analytics.

I personally think this is pretty neat… Hope you enjoyed the exploration.

Good luck, this is all fraught with rabbit holes, as always, so many and you can disappear so easily… but then that’s ½ the fun.



*Note: to execute this blog start with README.md located in the root folder and work from there, it will tell you exactly what to execute in which order to download all the dependencies and build everything.*

**About Me**

I’m a techie, a technologist, always curious, love data, have for as long as I can remember always worked with data in one form or the other, Database admin, Database product lead, data platforms architect, infrastructure architect hosting databases, backing it up, optimizing performance, accessing it. Data data data… it makes the world go round.

In recent years, pivoted into a more generic Technology Architect role, capable of full stack architecture.

[George Leonard](https://www.linkedin.com/in/george-leonard-945b502/)

[georgelza@gmail.com](mailto:georgelza@gmail.com)

<https://www.linkedin.com/in/george-leonard-945b502/>

<https://medium.com/@georgelza>