# 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.

(2 Jun 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 the lakehouse tier in Fluss).

What will we do:

This time round, we will again source from the same Confluent Kafka topics, create Apache Flink tables, but this time round we will be using the Pyflink framework to do some processing, progressing from as simply as we can, progressively getting more complicated.

* Lab 1 we will first publish the flattened IoT document back to our Confluent environment into3 topics.
* Lab 2 we will push the flattened IoT document to a Fluss table.
* Lab 3 we will push the flattened IoT document, processed using a UDF where we will simply uppercase the *unit* column into a Fluss table.
* Lab 4 we will push the flattened IoT document, processed using a UDF where we’ll be concatenating *siteId* & *deviceId* columns (after being converted to strings) into a new column (*complex*) into our Fluss table. The idea here is to show how multiple values can be pushed to a UDF to be analyzed.
* Lab 5 we will run a 1-minute tumbling window, which will be stored in our Fluss defined table.
* Lab 6, well this time we well extend Lab 5, we will use a UDF to analyze our Lab 5data and calculate a stability factor for the sensor based on the min, avg and max values per tumbling window. And yes, we went a bit further than originally planned.

Our Stack:

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 table stored on HDFS layer) for these labs as we’ve shown that working in the previous blog.

Our data is again generated 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.

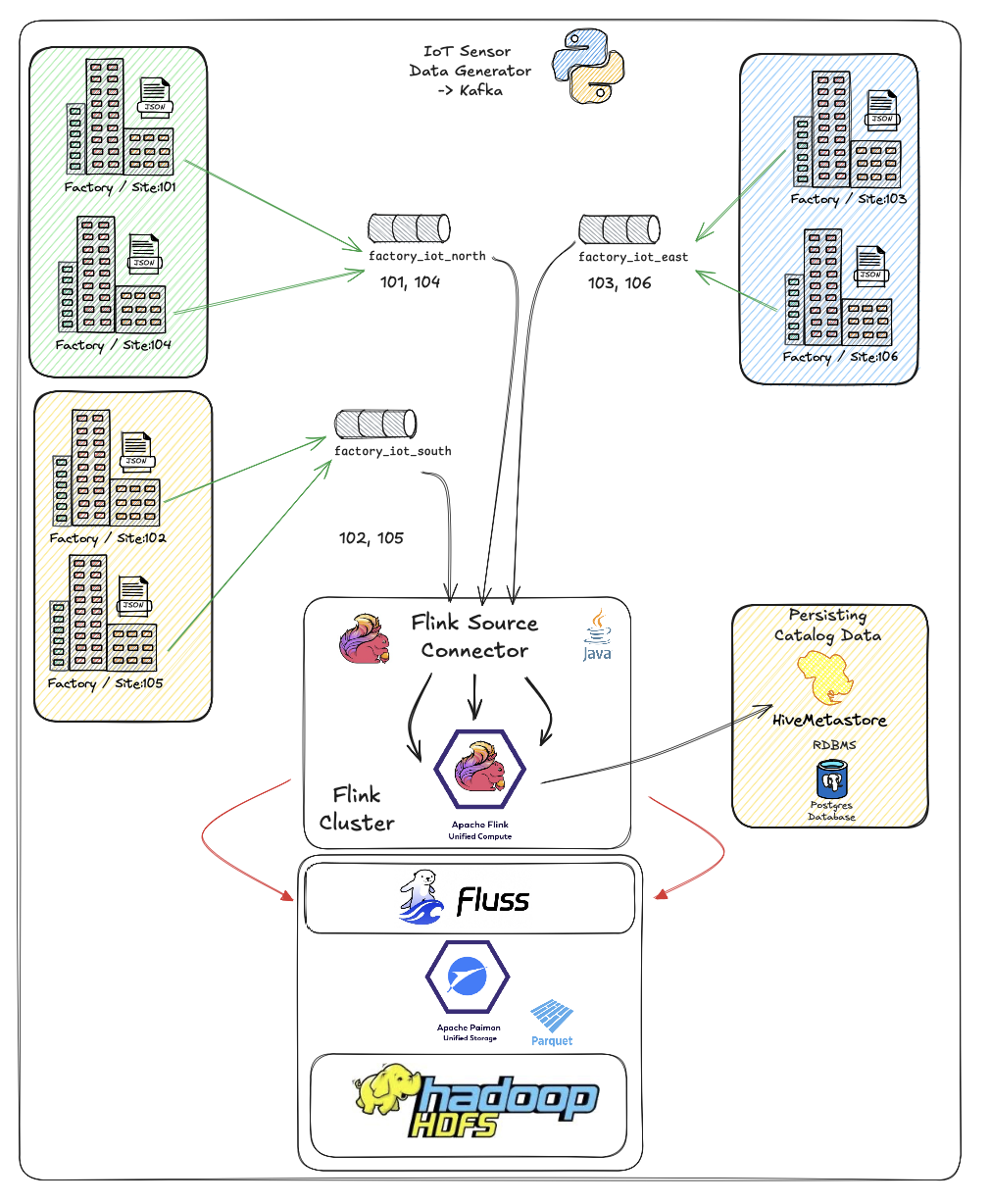
We will use our Python application to create a IoT data stream as if we’re reading sensors in factories fitted to machinery. These “[JSON](https://www.json.org/)” based IoT documents will be pushed onto our Kafka Topics called:

* *factory\_iot\_north*
* *factory\_iot\_south*
* *factory\_iot\_east*

The message “*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.

Re our payloads, our first region is been naughty, they are populating the absolute minimum tags of our 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, the East Region 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 the 3 payloads and how to generate them at the source end.

Instead of creating a table inside Apache Flink and use Flink SQL, we will use PyFlink (which is the ability of running Python jobs directly on the Flink Cluster) to process our data stream.

We got 5 examples/labs as per above, each building on the previous, getting a bit more complex as we progress.

**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'

);

"""

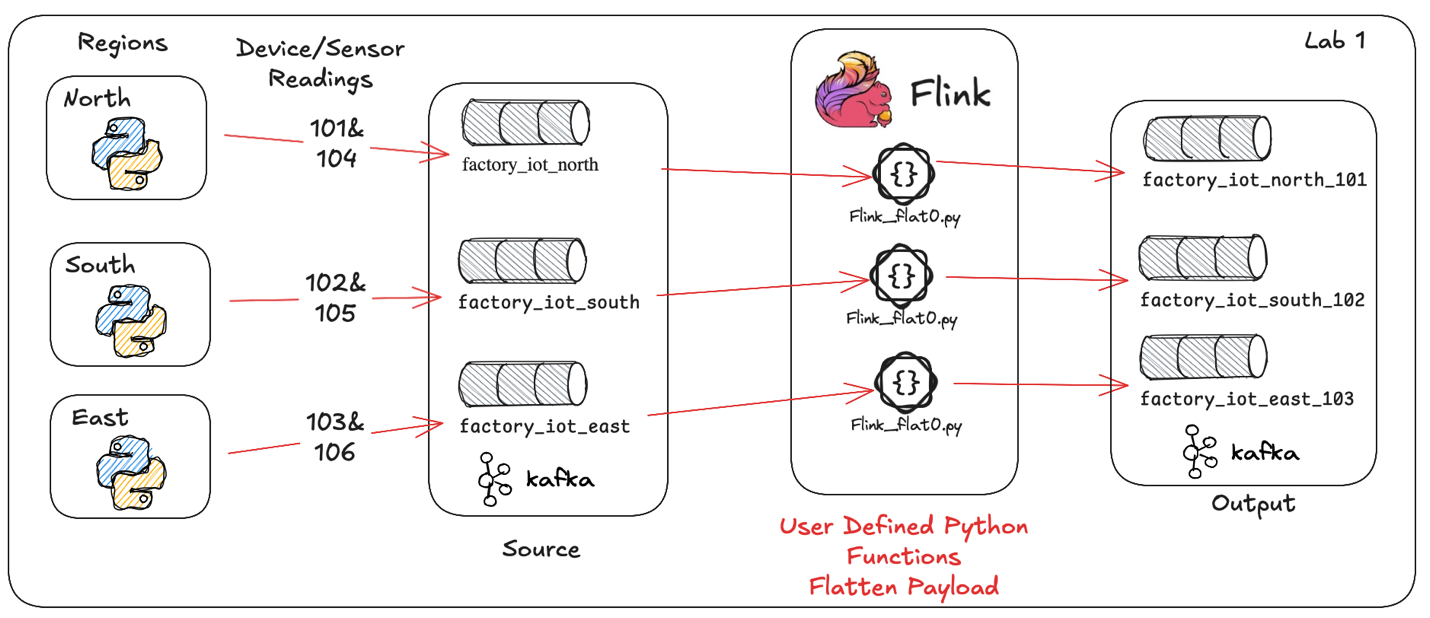
The above common input and output pattern is re-used in the all the labs.

**Lab 1**

In Lab 1 we will simply consume our messages, flatten the message and then publish each back onto a new topic.

Note: as part of the flatten process we also filter out based on a specific site. This is done for all the labs. You will notice below, the siteId to filter for is part of our command as an input argument.

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



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 \

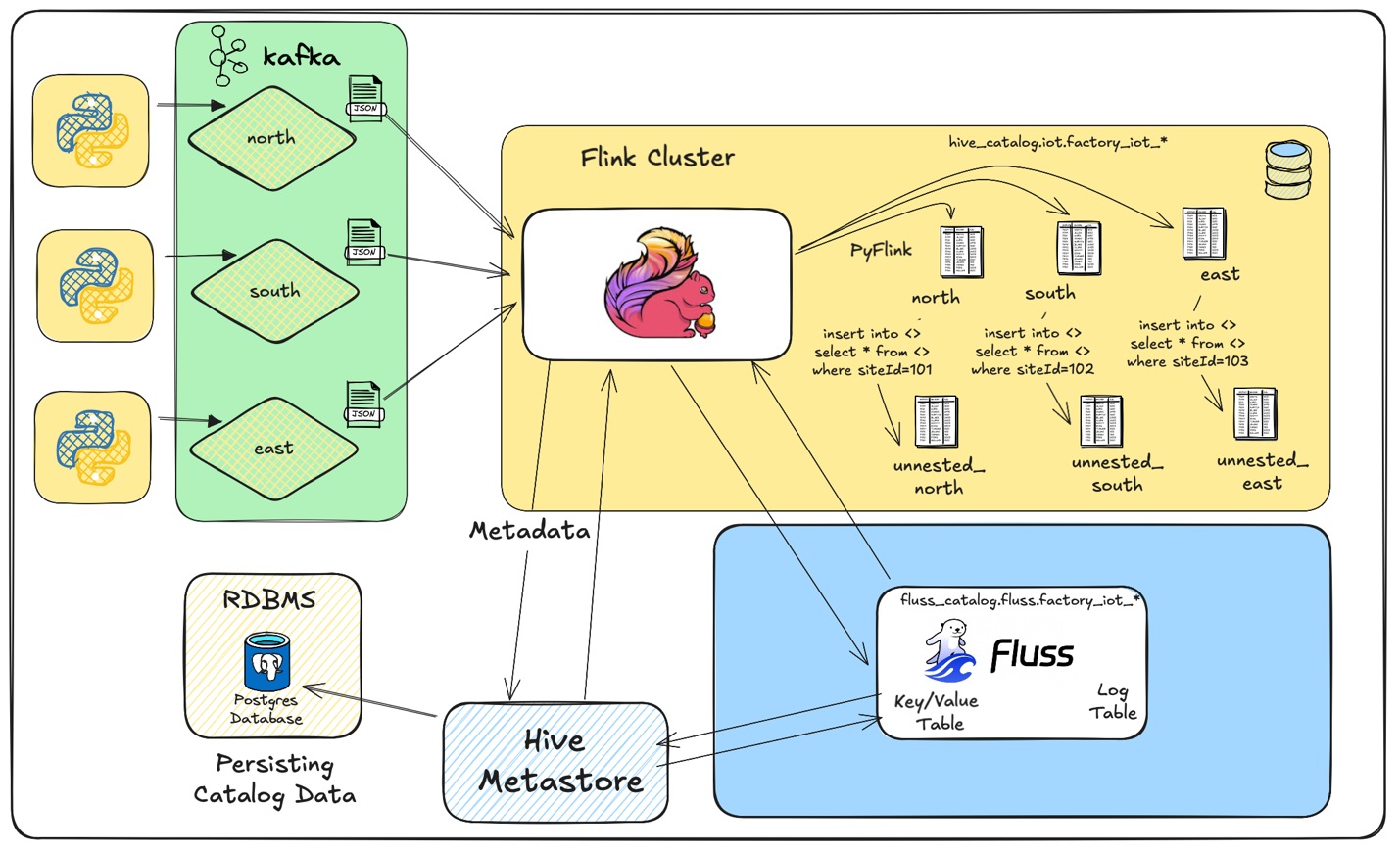
-py /pyapp/flink\_flat0.py \

--siteId 101 \

--source factory\_iot\_north

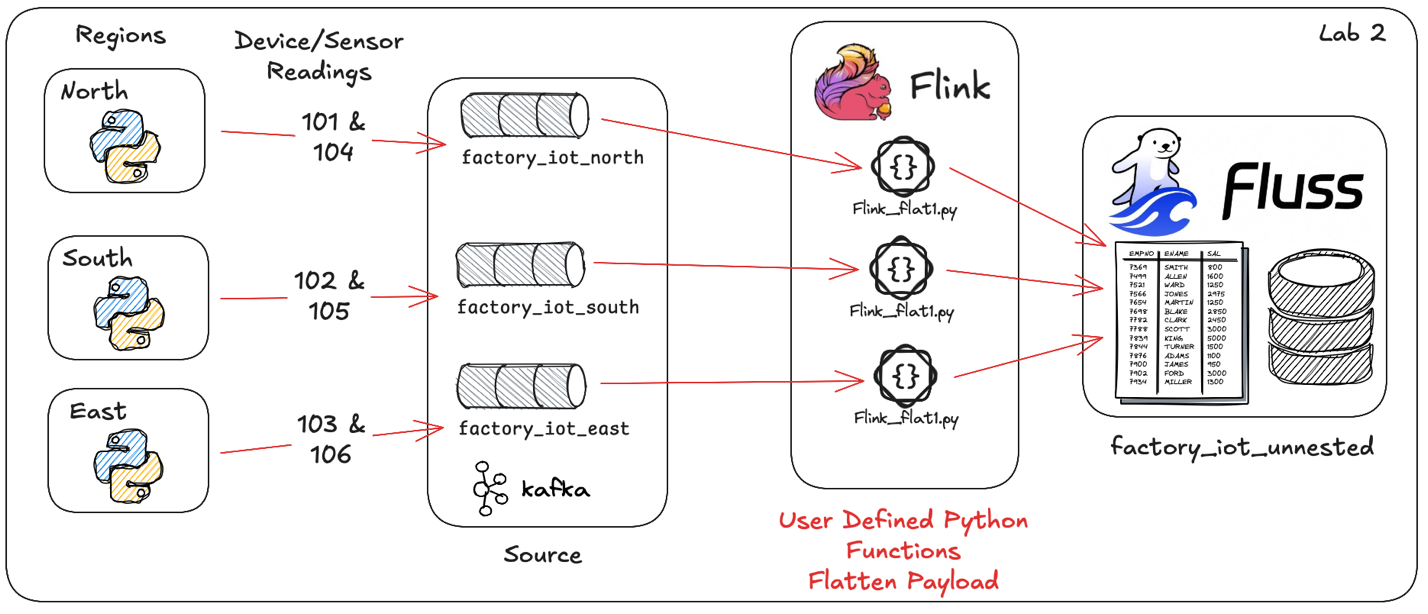
This examples output the flattened values to the above Kafka topics as per diagram which can be inspected via the Confluent Control Center / Topics / Messages tab.

For the labs 2-5 we will output our data to the Fluss cluster.



**Lab 2**

For Lab 2 we will repeat the above but this time we’re going to output to Fluss based table.



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 \

--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 the SQL contained in <*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;

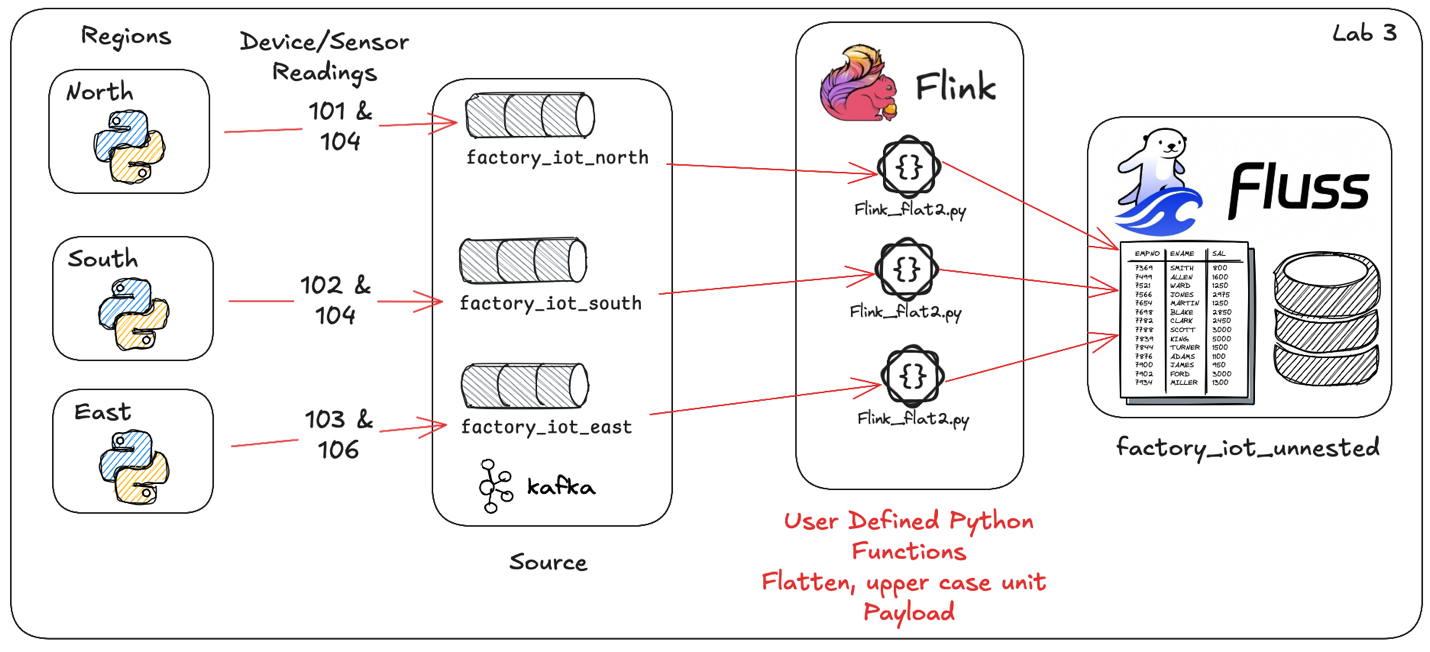
The above SQL can be executed by opening the *Flink SQL* client, this is accomplished by executing “*make fsql”* in the *<root>/devlab0/* directory.

**Lab 3**

In this version we’re going to implement a very simple inline User Defined Function that will uppercase our “unit” column value. The UDF is execute for each record processed.

The implementation is purely to demonstrate the scaffolding required. Once this is understood it can be seen how this can be repurposed to say call a AI/ML engine via a API call, or do a call to retrieve values from our in memory Fluss data store to compare against, in turn triggering a specific action.

Lab 3 is implemented via <*root>/devlab0/pyFlink/flink\_flat2.py* building on what we’ve learned in <*root>/devlab0/pyFlink/flink\_flat1.py.*



For this we will be using <*root>/devlab0/pyFlink/flink\_flat2.py,* we will again output our records to the *fluss\_catalog.fluss.factory\_iot\_unnested* table.

You can refer to <*root>/devlab0/pyFlink/flink\_flat2.cmd* for commands to run the above 3 PyFlink streams for lab 3.

You will see we added the “-*pyfs /pyapp/upper\_udf.py”* tag specifying “*upper\_udf.py”* containing our upper function.

First option would have been to use the *“-j”* tag, as per below. This however did not resolve the problem, error as per below. What did remove the error stack was adding the libraries to the python file as per below. This can generally be found at lines 88-90 of the \*.py files.

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

*-j /opt/flink/opt/flink-python-1.20.1.jar \*

kafka\_connector\_jar = "file:///opt/flink/lib/flink/flink-sql-connector-kafka-3.3.0-1.20.jar" # Adjust version and path

flink\_python\_jar = "file:///opt/flink/opt/flink-python-1.20.1.jar" # <--- ENSURE THIS PATH AND FILENAME ARE EXACT

t\_env.get\_config().set("pipeline.jars", f"{kafka\_connector\_jar};{flink\_python\_jar}")

Without the above changes you will get the following error as per below. This error does not present itself at the terminal. The terminal command will succeed, you will however end with a running/restarting job in the Flink Console, if you click on the job and navigate to the “exceptions” tab you would find the below error stack. – Thanks to <https://gemini.google.com> for helping resolve the error.

025-05-29 09:55:30

org.apache.flink.streaming.runtime.tasks.StreamTaskException: Cannot load user class: org.apache.flink.table.runtime.operators.python.scalar.PythonScalarFunctionOperator

ClassLoader info: URL ClassLoader:

file: '/tmp/tm\_172.18.0.11:32801-5ee082/blobStorage/job\_17473d5ad47c2b77ee191e9f0e067394/blob\_p-9e7e2bb762e6cb489bcc76f2637e824fbb6f08c3-87a1dec287ef364fea85278f34b13f44' (valid JAR)

Class not resolvable through given classloader.

at org.apache.flink.streaming.api.graph.StreamConfig.getStreamOperatorFactory(StreamConfig.java:414)

at org.apache.flink.streaming.runtime.tasks.OperatorChain.createOperator(OperatorChain.java:869)

at org.apache.flink.streaming.runtime.tasks.OperatorChain.createOperatorChain(OperatorChain.java:836)

at org.apache.flink.streaming.runtime.tasks.OperatorChain.createOutputCollector(OperatorChain.java:732)

at org.apache.flink.streaming.runtime.tasks.OperatorChain.createOperatorChain(OperatorChain.java:825)

at org.apache.flink.streaming.runtime.tasks.OperatorChain.createOutputCollector(OperatorChain.java:732)

at org.apache.flink.streaming.runtime.tasks.OperatorChain.<init>(OperatorChain.java:202)

at org.apache.flink.streaming.runtime.tasks.RegularOperatorChain.<init>(RegularOperatorChain.java:60)

at org.apache.flink.streaming.runtime.tasks.StreamTask.restoreInternal(StreamTask.java:789)

at org.apache.flink.streaming.runtime.tasks.StreamTask.restore(StreamTask.java:771)

at org.apache.flink.runtime.taskmanager.Task.runWithSystemExitMonitoring(Task.java:970)

at org.apache.flink.runtime.taskmanager.Task.restoreAndInvoke(Task.java:939)

at org.apache.flink.runtime.taskmanager.Task.doRun(Task.java:763)

at org.apache.flink.runtime.taskmanager.Task.run(Task.java:575)

at java.base/java.lang.Thread.run(Thread.java:840)

Caused by: java.lang.ClassNotFoundException: org.apache.flink.table.runtime.operators.python.scalar.PythonScalarFunctionOperator

at java.base/java.net.URLClassLoader.findClass(URLClassLoader.java:445)

at java.base/java.lang.ClassLoader.loadClass(ClassLoader.java:592)

at org.apache.flink.util.FlinkUserCodeClassLoader.loadClassWithoutExceptionHandling(FlinkUserCodeClassLoader.java:67)

at org.apache.flink.util.ChildFirstClassLoader.loadClassWithoutExceptionHandling(ChildFirstClassLoader.java:65)

at org.apache.flink.util.FlinkUserCodeClassLoader.loadClass(FlinkUserCodeClassLoader.java:51)

at java.base/java.lang.ClassLoader.loadClass(ClassLoader.java:525)

at org.apache.flink.util.FlinkUserCodeClassLoaders$SafetyNetWrapperClassLoader.loadClass(FlinkUserCodeClassLoaders.java:197)

at java.base/java.lang.Class.forName0(Native Method)

at java.base/java.lang.Class.forName(Class.java:467)

at org.apache.flink.util.InstantiationUtil$ClassLoaderObjectInputStream.resolveClass(InstantiationUtil.java:76)

at java.base/java.io.ObjectInputStream.readNonProxyDesc(ObjectInputStream.java:2034)

at java.base/java.io.ObjectInputStream.readClassDesc(ObjectInputStream.java:1898)

at java.base/java.io.ObjectInputStream.readOrdinaryObject(ObjectInputStream.java:2224)

at java.base/java.io.ObjectInputStream.readObject0(ObjectInputStream.java:1733)

at java.base/java.io.ObjectInputStream$FieldValues.<init>(ObjectInputStream.java:2606)

at java.base/java.io.ObjectInputStream.readSerialData(ObjectInputStream.java:2457)

at java.base/java.io.ObjectInputStream.readOrdinaryObject(ObjectInputStream.java:2257)

at java.base/java.io.ObjectInputStream.readObject0(ObjectInputStream.java:1733)

at java.base/java.io.ObjectInputStream.readObject(ObjectInputStream.java:509)

at java.base/java.io.ObjectInputStream.readObject(ObjectInputStream.java:467)

at org.apache.flink.util.InstantiationUtil.deserializeObject(InstantiationUtil.java:488)

at org.apache.flink.util.InstantiationUtil.deserializeObject(InstantiationUtil.java:472)

at org.apache.flink.util.InstantiationUtil.deserializeObject(InstantiationUtil.java:467)

at org.apache.flink.util.InstantiationUtil.readObjectFromConfig(InstantiationUtil.java:422)

at org.apache.flink.streaming.api.graph.StreamConfig.getStreamOperatorFactory(StreamConfig.java:400)

... 14 more

Now that we made the above changes, we can run the lab at the command prompt in the *jobmanager*, this can be accomplished by executing “*make jm”* in the *<root>/devlab0/* directory.

/opt/flink/bin/flink run \

-m jobmanager:8081 \

-py /pyapp/flink\_flat2.py \

-pyfs /pyapp/upper\_udf.py \

--siteId 101 \

--source factory\_iot\_north

For this example, we will reuse our previous defined *fluss\_catalog.fluss.factory\_iot\_unnested* table*.*

If you want to inspect the data in the Fluss target table you can execute <*root>/devlab0/pyFlink/flink\_flat2.sql* in a Flink SQL window. All we did here is add the unit column to the query when compared to lab 3.

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

SET 'parallelism.default' = '2';

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

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

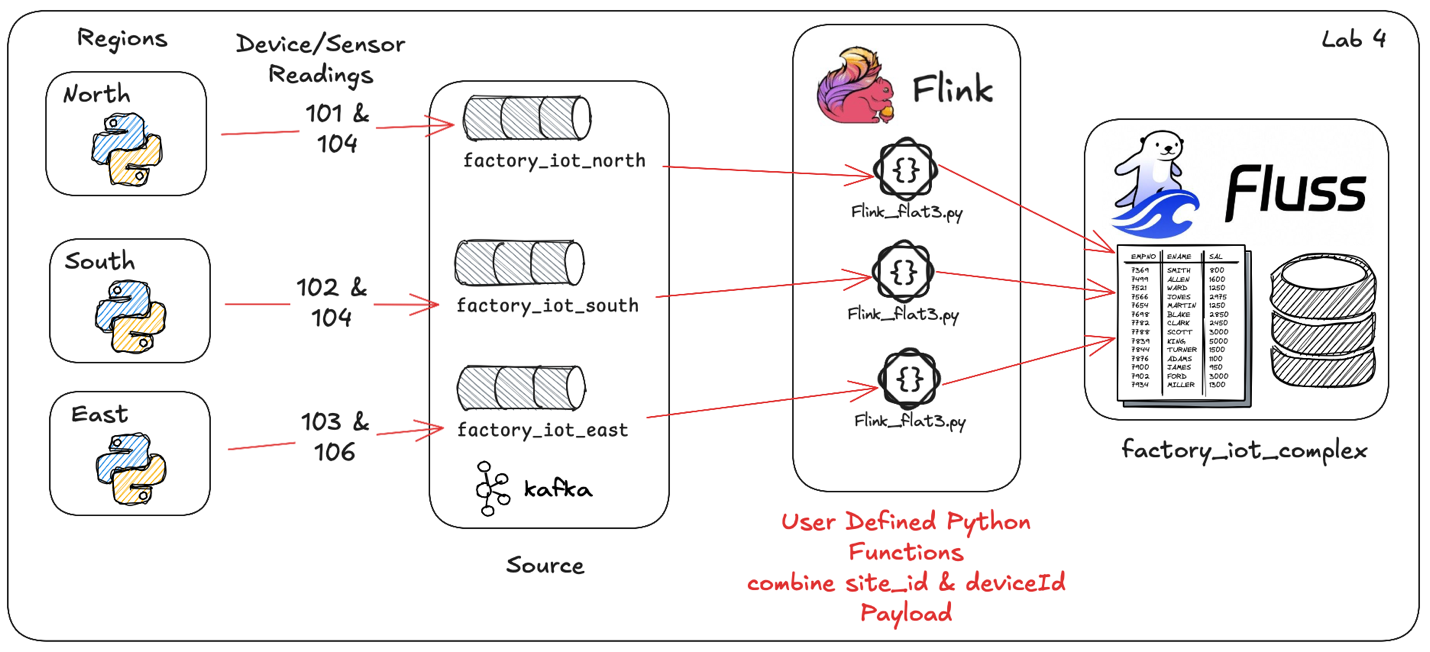
select ts, siteId, deviceId, sensorId, unit, measurement from fluss\_catalog.fluss.factory\_iot\_unnested;

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

**Lab 4**

We build even further on what we’ve done previous labs. We will as previously redo the flattening step, then as part of the insert statement we will call our UDF. This time we will combine the *siteId* and *deviceId* columns into a new column *complex* as a string.

Lab 4 is implemented via <*root>/devlab0/pyFlink/flink\_flat3.py* building on what we’ve learned in <*root>/devlab0/pyFlink/flink\_flat2.py.*



/opt/flink/bin/flink run \

-m jobmanager:8081 \

-py /pyapp/flink\_flat3.py \

-pyfs /pyapp/upper\_udf.py \

--siteId 101 \

--source factory\_iot\_north

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

For this example, we will output our records to *fluss\_catalog.fluss.factory\_iot\_complex* table*.*

If you want to inspect the data in the Fluss target table you can execute <*root>/devlab0/pyFlink/flink\_flat3.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';

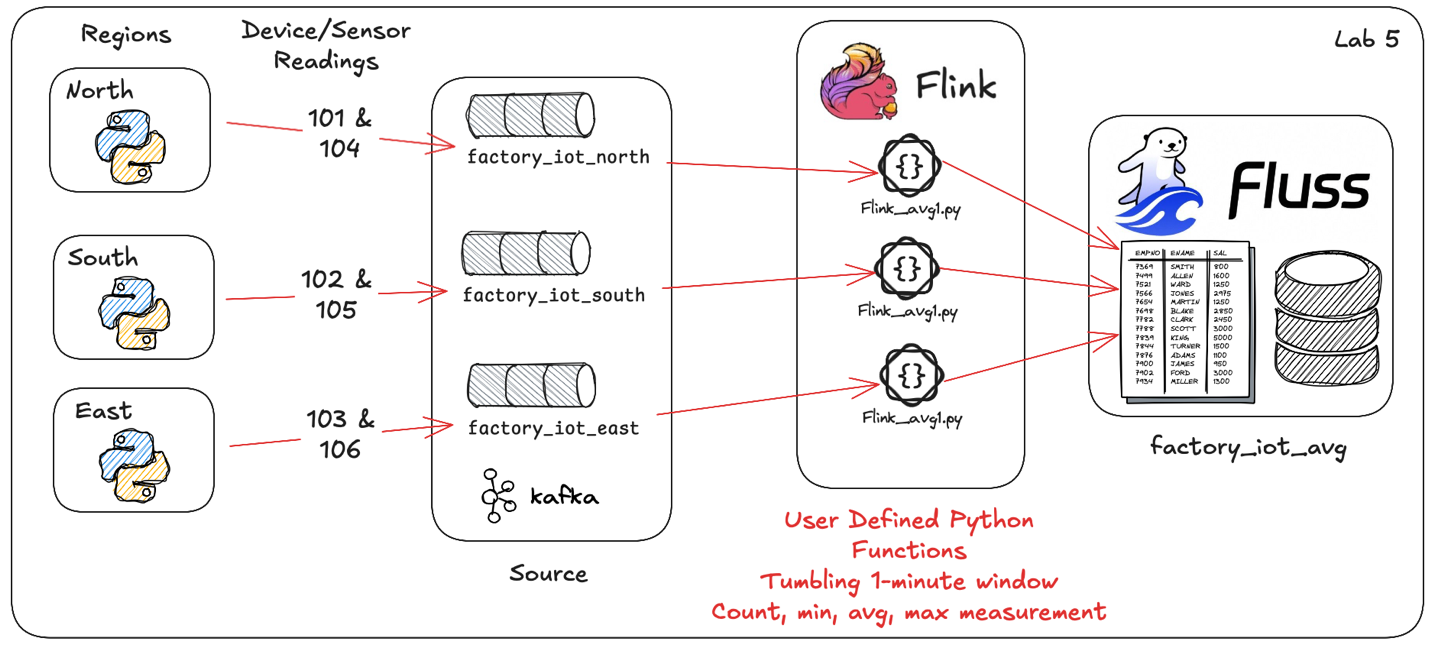
select ts, siteId, deviceId, sensorId, complex, unit, measurement from fluss\_catalog.fluss.factory\_iot\_complex;

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

**Lab 5**

And for the next lab using 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 *siteId, deviceId, sensorId*.

<*root>/devlab0/pyFlink/flink\_avg1.py*



You can see <*root>/devlab0/pyFlink/flink\_avg1.cmd* for commands to run the above 3 PyFlink streams for lab 5.

/opt/flink/bin/flink run \

-m jobmanager:8081 \

-py /pyapp/flink\_avg1.py \

--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';

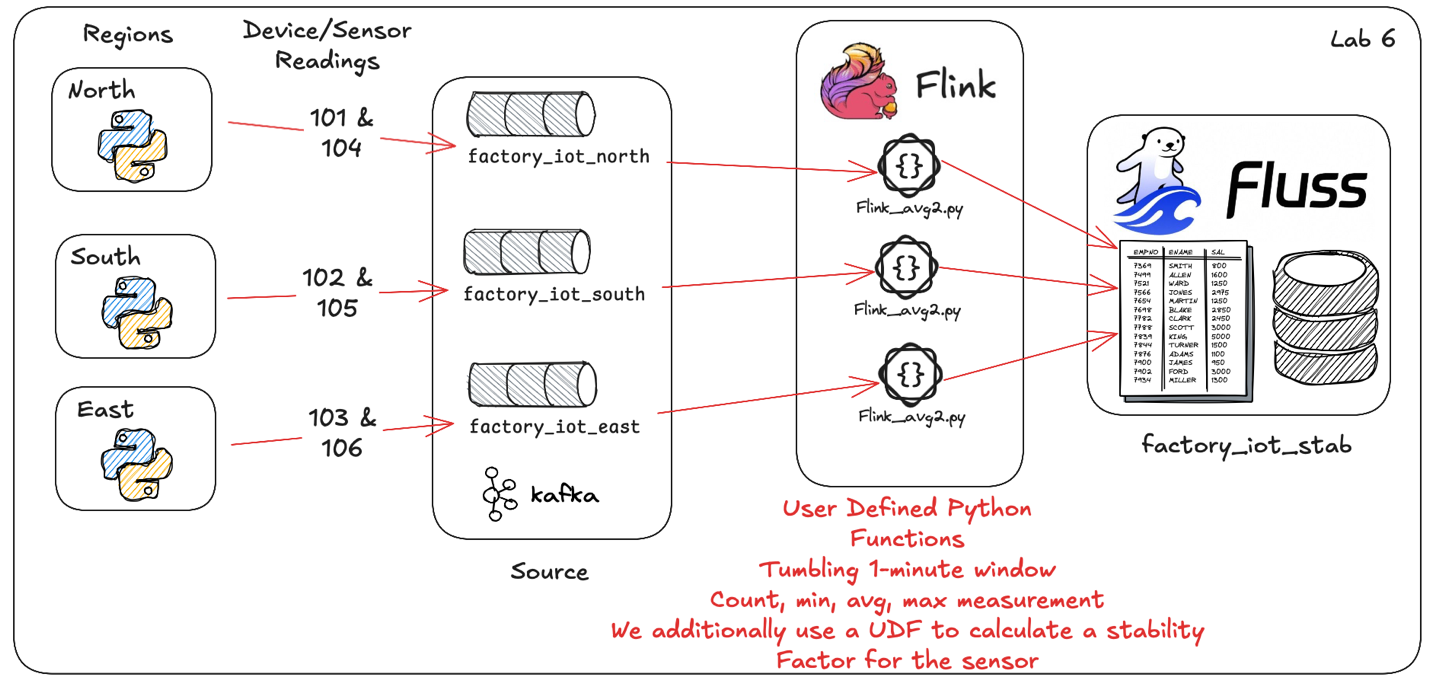
select siteId, deviceId, sensorId, measurement\_count, avg\_measurement, window\_start, window\_end from fluss\_catalog.fluss.factory\_iot\_avg;

The above SQL can be executed by opening the *Flink SQL* client, this is again, and this time lastly accomplished by executing “*make fsql”* in the *<root>/devlab0/* directory.

**Lab 6**

And Finally ☺ for the last lab using we will be combining the UDF capability and Apache 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 *siteId, deviceId, sensorId*. But then go further and push the min, avg and max measurements into a UDF to determine a stability factor of the sensors readings.

<*root>/devlab0/pyFlink/flink\_avg2.py*



You can see <*root>/devlab0/pyFlink/flink\_avg2.cmd* for commands to run the above 3 PyFlink streams for lab 6.

/opt/flink/bin/flink run \

-m jobmanager:8081 \

-py /pyapp/flink\_avg2.py \

-pyfs /pyapp/satability\_udf.py \

--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\_stab.*

If you want to see the data in the Fluss target you can execute <*root>/devlab0/pyFlink/flink\_avg2.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, stability\_factor, min\_measurement, avg\_measurement, max\_measurement, window\_start, window\_end from fluss\_catalog.fluss.factory\_iot\_avg;

The above SQL can be executed by opening the *Flink SQL* client, this is again, and this time lastly accomplished by executing “*make fsql”* in the *<root>/devlab0/* directory.



And 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 for lab 2 – 6. Every lab builds on what we accomplished in the previous lab.

Why all of this… Well in the world of industrial systems, we can use this PyFlink layer running on our Apache Flink cluster to compare real time metrics against known good values, swim lanes as may, analyse using ML to raise alarms before a breakdown 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 a transaction against known previous behaviour of the account holder, outputting alarms in near real time, all while outputting all our inbound and computed data to our Fluss platform for further real time 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 by reading <root>/README.md 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>