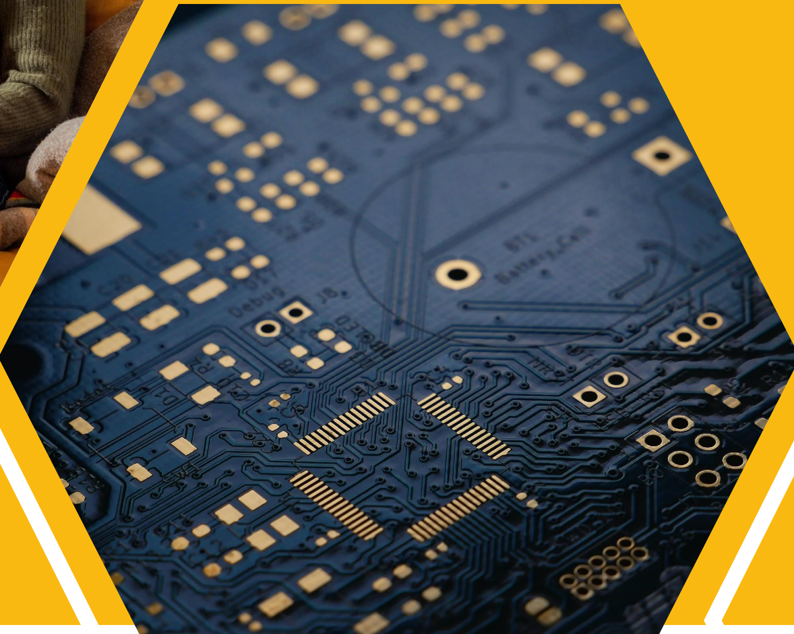
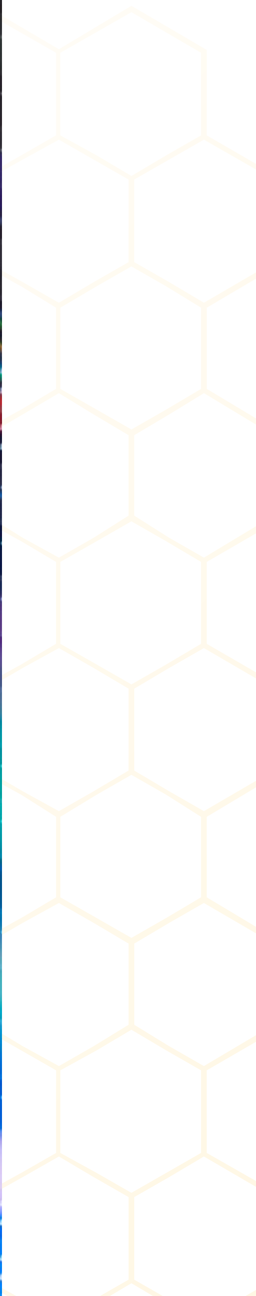


Real-Time Revolution:

Kickstarting Your Journey in Streaming Data

Real-time data is all around us

TikTok



Recommendations	Addictive TikTok
Ads	Annoying or not, at least less so with real-time relevance
Personalization	Mind reading in 2023, or at least it feels like it
Fraud Detection	Alerting us of our suspect train ticket purchases in foreign countries 😄

I'm Zander!

Working on Bytewax
→ github.com/bytewax/bytewax

bytewax

I’m Zander!

- Working on Bytewax
  → github.com/bytewax/bytewax
- Proud human and dog dad

I’m Zander!

- Working on Bytewax
  → github.com/bytewax/bytewax
- Proud human and dog dad
- This photo is me trying to look cool 😆

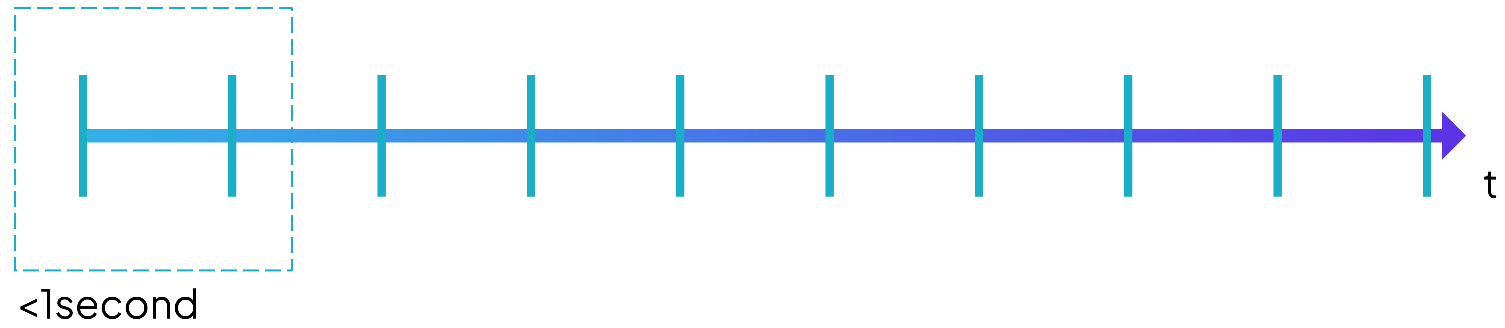
I'm Zander!

- Working on Bytewax
→ github.com/bytewax/bytewax
- Proud human and dog dad
- This photo is me trying to look cool
- You can find me in Bytewax slack, or in Santa Cruz → send me a LinkedIn and we can grab a coffee.

Today's Agenda

- Understanding real-time data
- Introduction to streaming data
- The hard parts of streaming
- How Bytewax makes the hard parts easier 🙂

What is Real-Time?



In the world of data real-time means:

- The data is available for use as soon as it is generated
- The data is processed immediately
- The data is made available to the consumer post-processing

And this entire process happens in a **real-time low latency** manner

So then what is Streaming Data?

Batch Processing
Batch

Stream Processor
Stream

Streaming Data.ipynb

File Edit View Insert Runtime Tools Help All changes saved

+ Code + Text

Data Streams

Data streams can be reasoned with well as generators in Python. The generator is an iterable object that we can reason about with built-in methods.

We can define an infinite stream of randomly occurring integers.

```
[ ] import random

    def my_stream():
        while True:
            i = random.randint(0, 10)
            yield i
```

Now we have created a fake stream of data.

Streaming Data Consumers

Now to work with the data stream we need to create a consumer, this would be the generator that we assign to a variable. Then we can call next on the generator to receive the next object in the stream.

```
[ ] stream = my_stream()
```

```
[ ] next(stream)
```

```
2
```

```
[ ] next(stream)
```

```
5
```

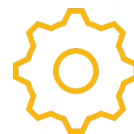
This seems easy, why do I need a stream processor?

- But what if our input had many different keys?
- Or if we needed a time based window?
- Or we needed to join multiple inputs together?
- Or if our machine died and we lost track of where we were in the stream or what our current average was for all of our keys?
- Or we needed to scale things up, and now we need to make sure the right data goes to the right process so our calculations are correct.

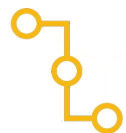
Bytewax

Open Source Python Stream Processing

Python native API



Performant Rust engine



Stateful stream processor: windowing, aggregations and connectors

DevOps light with deployment and observability (platform only)

Scalable and cloud-native

Production ready with disaster recovery included

bytewax

But what if our input had many different keys?

```python
import json

from bytewax import operators as op
from bytewax.connectors.kafka import KafkaSource
from bytewax.dataflow import Dataflow

# Define the dataflow object and kafka input.
flow = Dataflow("event time")
brokers = ["localhost:19092"]
topics = ["sensors"]
stream = op.input("inp", flow, KafkaSource(brokers, topics, tail=False))


# We expect a json string that represents a reading from a sensor.
def parse_value(key__data):
    _, data = key__data
    return json.loads(data)


parsed_stream = op.map("parse_value", stream, parse_value)


# Group the readings by sensor type, so that we only
# aggregate readings of the same type.
keyed_stream = op.key_on("extract_type", parsed_stream, lambda event: event["type"])
```

Or if we needed a time based window?

```python
# This is the accumulator function, and outputs a list of 2-tuples,
# containing the event's "value" and it's "time" (used later to print info)
def acc_values(acc, event):
    acc.append((event["value"], event["time"]))
    return acc


# This function instructs the event clock on how to retrieve the
# event's datetime from the input.
# Note that the datetime MUST be UTC. If the datetime is using a different
# representation, we would have to convert it here.
def get_event_time(event):
    return datetime.fromisoformat(event["time"])


# Configure the `fold_window` operator to use the event time.
cc = EventClockConfig(get_event_time, wait_for_system_duration=timedelta(seconds=10))

# And a 5 seconds tumbling window
align_to = datetime(2023, 1, 1, tzinfo=timezone.utc)
wc = TumblingWindow(align_to=align_to, length=timedelta(seconds=5))

running_avg_stream = window_op.fold_window(
    "running_average", keyed_stream, cc, wc, list, acc_values
)
```

Or we needed to join multiple inputs together?

```python
from bytewax import operators as op
from bytewax.connectors.stdio import StdOutSink
from bytewax.dataflow import Dataflow
from bytewax.testing import TestingSource

flow = Dataflow("join")


def key_getter(x):
    return str(x["user_id"])


inp1 = op.input("inp1", flow, TestingSource([{"user_id": 123, "name": "Bumble"}]))
(k_inp1,) = op.key_split("k1", inp1, key_getter, lambda x: x["name"])
inp2 = op.input(
    "inp2", flow, TestingSource([{"user_id": 123, "email": "bee@bytewax.com"}])
)
(k_inp2,) = op.key_split("k2", inp2, key_getter, lambda x: x["email"])
inp3 = op.input(
    "inp3", flow, TestingSource([{"user_id": 123, "color": "yellow", "sound": "buzz"}])
)
k_inp3, k_inp4 = op.key_split(
    "k3", inp3, key_getter, lambda x: x["color"], lambda x: x["sound"]
)

joined = op.join("j1", k_inp1, k_inp2, k_inp3, k_inp4)
op.output("out", joined, StdOutSink())
```

Or if our machine died and we lost track of where we were in the stream or what our current average was for all of our keys?

```
# Scaling up the workers
$ python -m bytewax.run dataflow:flow -w 3

# Scaling up the workers and processes


# Now, with recovery
# create our recovery partitions
$ python -m bytewax.recovery db_dir/ 4

# run our data flow using the recovery partitions
$ python -m bytewax.run dataflow:flow -r db_dir/
```

Or we needed to scale things up, and now we need to make sure the right data goes to the right process so our calculations are correct.

```
# Scaling up the workers
$ python -m bytewax.run dataflow:flow -w 3

# Scaling up the workers and processes
$ python -m bytewax.run dataflow:flow -w 3 -p 2

# Scaling up the workers and processes manually on different machines

# on machine one
$ python -m bytewax.run simple:flow -w 3 -i0 -a "cluster_one:2101;cluster_two:2101"

# and on machine two
$ python -m bytewax.run simple:flow -w 3 -i1 -a "cluster_one:2101;cluster_two:2101"

# Or run it cloud-natively on kubernetes
$ waxctl df deploy dataflow.py --name my-dataflow -w 3 -p 2
```

bytewax