

3 Libraries You Should Know to Master Apache Kafka in Python

A Handbook for Python Developers to use Kafka



Xiaoxu Gao

May 26 · 10 min read ★



Photo by [Jake Givens](#) on [Unsplash](#)

The world is empowered by data. We get tons of information every second, we clean it up, analyze it and create more valuable output, whether it is a log file, user activity, a chat message or something else. The faster we deliver, the more value

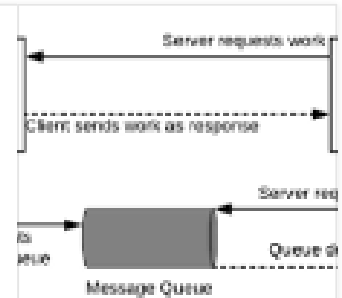
we will bring to our customers. We're in the era of a fast-paced and ever-changing environment.

Apache Kafka is a distributed streaming platform that can publish, subscribe, store and process messages in real-time. Its pull-based architecture reduces the pressure on the service with a heavy load and makes it easy to scale. It moves a huge amount of data from the source to the destination with low latency.

Thoughts on Push vs Pull Architectures

I've had a few discussions with people lately about the advantages and disadvantages of different service architectures...

medium.com



Kafka is a JVM based platform, so the mainstream programming language of the client is Java. But as the community is growing tremendously, high-quality open-sourced Python clients are also available and being used in production.

In this article, I will cover the most well-known Python Kafka clients: *kafka-python*, *pykafka* and *confluent-kafka* and compare them. In the end, I will give my opinion on the pros and cons of each library.

. . .

Why do we want Kafka?

First thing first. Why Kafka? Kafka is intended for boosting an event-driven architecture. It empowers the architecture by providing high throughput, low latency, high durability, and high availability solution. *(It doesn't mean you can have all of them at the same time, there is always a tradeoff. Read this whitepaper to understand more.)*

How to Deploy and Optimize Kafka for High Performance and Low Latency

Apache Kafka® is a power stream processing platformhis white paper discusses how to optimize Kafka deployments for...

www.confluent.io

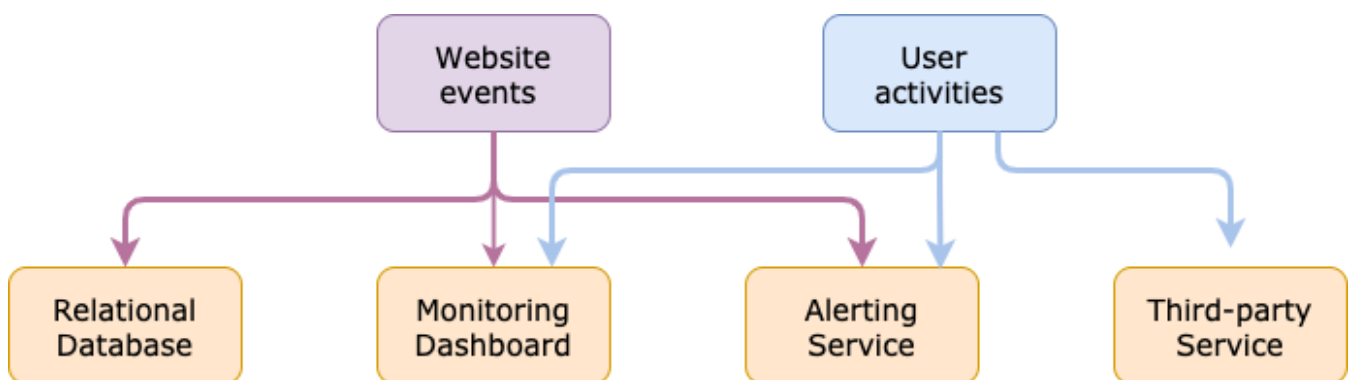


Besides its high performance, another attractive feature is the *publish/subscribe* model, where the sender doesn't send messages specifically to a receiver. Instead, the messages are delivered to a centralized place that receivers can subscribe to, depending on the *topic*.

By doing so, we can easily decouple applications and get rid of monolithic design. Let's look at an example to understand why decoupling is better.

You create a website that needs to send user activities somewhere, so you write a direct connection from your website to a real-time monitoring dashboard. It is a simple solution and works well. One day, you decide to store the user activities in a database for future analysis. So, you write another direct database connection to your website. Meanwhile, your website gets more and more traffic, and you want to empower it by adding an alerting service, real-time analysis service, etc.

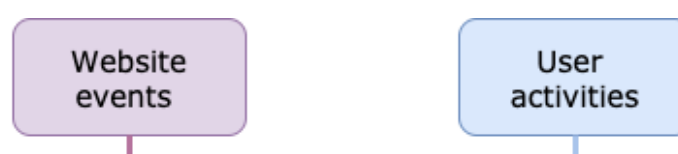
Your architecture will end up like this. Problems such as a massive code repo, security issues, scalability issues, and maintainability issues will hurt you.

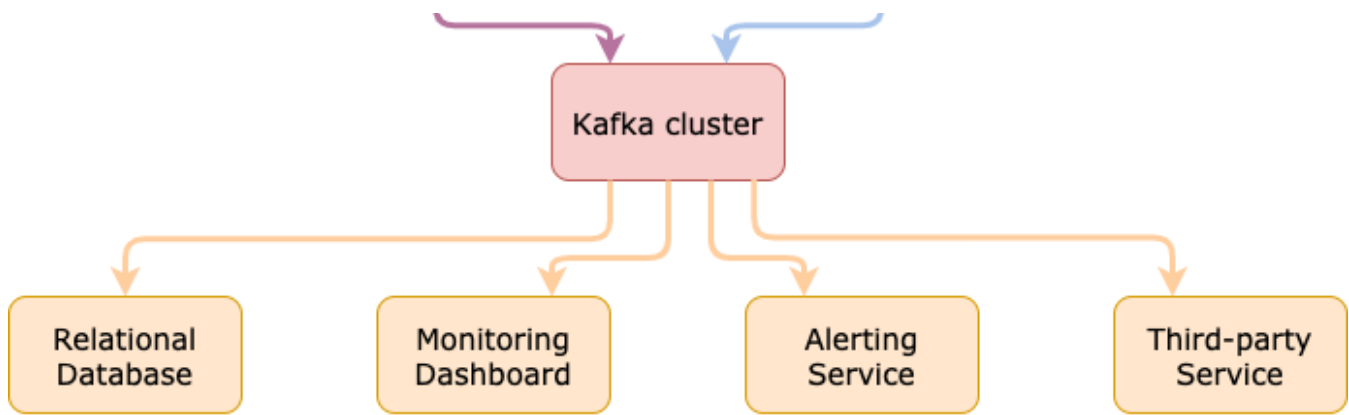


Architecture without decoupling (Created by Xiaoxu Gao)

You need a *hub* to separate applications with a different role. For applications that create events, we call them *producers*. They publish events to a centralized *hub*. Each event (i.e. message) belongs to a *topic*. On the other side of the *hub* sit *consumers*. They subscribe to the topics that they need from the *hub* without directly talking to the producers.

With this model in place, the architecture can be easily scaled and maintained. Engineers can focus more on the core business.





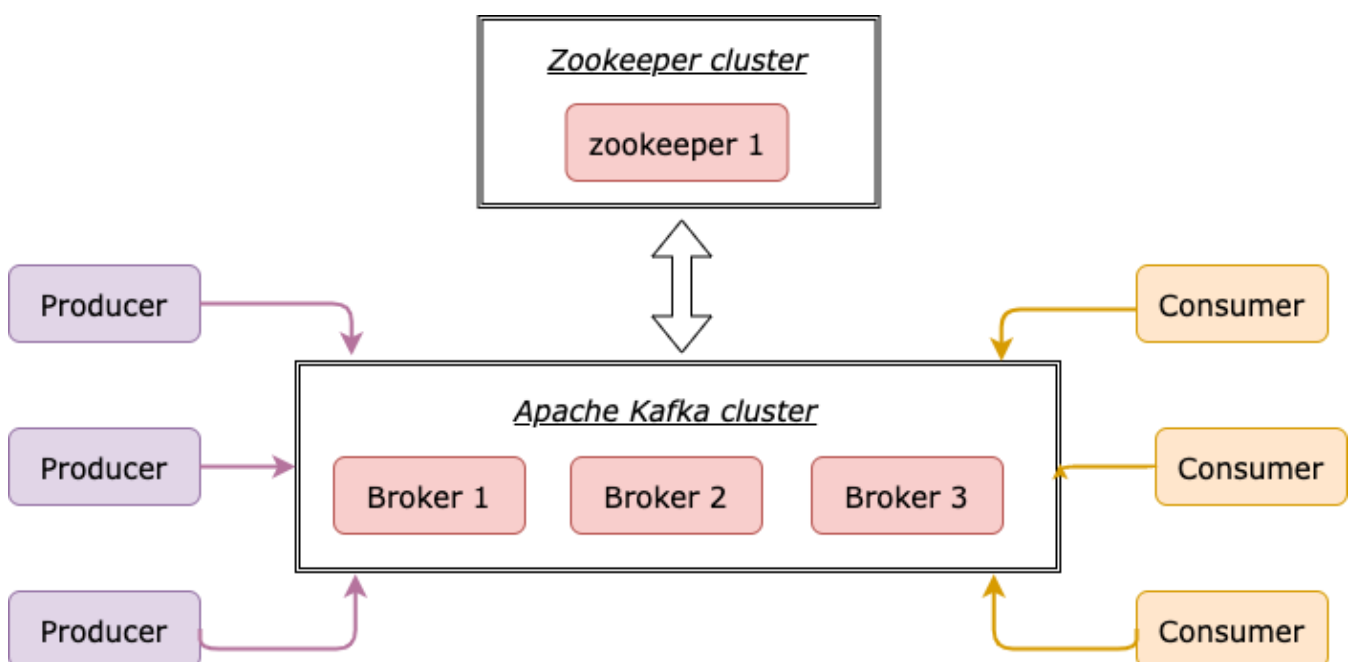
Architecture with decoupling (Created by Xiaoxu Gao)

Kafka setup in a nutshell

You can download Apache Kafka from [the official website](#). The [Quickstart](#) helps you to start up the server in 10 seconds.

You can also download Apache Kafka from [Confluent platform](#). It is by far the biggest Kafka-oriented streaming data platform. It provides a collection of infrastructure services surrounding Kafka to individuals and enterprises for making the data available as realtime streams. [The founders were part of the team originally created Apache Kafka.](#)

Each Kafka server is called a **broker**, you can run it in a standalone mode or form a cluster. Besides Kafka, we need [Zookeeper](#) to store metadata about Kafka. Zookeeper acts like an orchestrator that manages the status of each broker within the distributed system.



Kafka setup (Created by Xiaoxu Gao)

Let's say we've set up the infrastructure with 1 Zookeeper and 1 Kafka broker. It's time to connect to it! The original Java client provides **5 APIs**:

- Producer API: Publish messages to the topics in the Kafka cluster.
- Consumer API: Consume messages from the topics in the Kafka cluster.
- Streams API: Consume messages from the topics and transform them into other topics in the Kafka cluster. The operations can be filtering, joining, mapping, grouping, etc.
- Connect API: Directly connect the Kafka cluster to a source system or a sink system without coding. The system can be a file, a relational database, Elasticsearch, etc.
- Admin API: Manage and inspect topics and brokers in the Kafka cluster.

Python libraries for Kafka

In the Python world, 3 out of 5 APIs have been implemented which are Producer API, Consumer API, and Admin API. There is no such Kafka Stream API yet in Python, but a good alternative would be **Faust**.

The testing in this section is executed based on 1 Zookeeper and 1 Kafka broker installed locally. This is not about performance tuning, so I'm mostly using the default configurations provided by the library.

Kafka-Python

kafka-python is designed to function much like the official java client, with a sprinkling of pythonic interfaces. It's best used with Kafka version 0.9+. The first release was in March 2014. It's being actively maintained.

Installation

```
pip install kafka-python
```

Producer

Each message is sent via `send()` **asynchronously**. When called it adds the record to a buffer and returns immediately. This allows the producer to send records to Kafka

brokers in batch mode for efficiency. Asynchronization can improve speed tremendously, but we should also understand a few things:

1. In the asynchronous mode, the ordering is not guaranteed. You can't control when each message is acknowledged (ack) by Kafka brokers.
2. It's a good practice to have a success callback and a failure callback for the producer. For example, you can write an Info log message in the success callback and an Exception log message in the failure callback.
3. Extra messages might be sent before you receive an exception in the callback due to the fact that ordering can not be guaranteed.

If you want to avoid these problems, you can choose to send messages synchronously. The return of `send()` is a `FutureRecordMetadata`. By doing `future.get(timeout=60)`, the producer will be blocked for at most 60 seconds until the message has been successfully acknowledged by the brokers. The drawback is the speed, it is relatively slow compared to asynchronous mode.

```
1  import time
2  from kafka import KafkaProducer
3
4  msg = ('kafkakafkakafka' * 20).encode()[:100]
5  size = 1000000
6  producer = KafkaProducer(bootstrap_servers='localhost:9092')
7
8  def kafka_python_producer_sync(producer, size):
9      for _ in range(size):
10         future = producer.send('topic', msg)
11         result = future.get(timeout=60)
12         producer.flush()
13
14  def success(metadata):
15      print(metadata.topic)
16
17  def error(exception):
18      print(exception)
19
20  def kafka_python_producer_async(producer, size):
21      for _ in range(size):
22         producer.send('topic', msg).add_callback(success).add_errback(error)
23         producer.flush()
```

Consumer

The consumer instance is a Python iterator. The core of the consumer class is `poll()` method. It allows the consumer to keep pulling messages from the topic. One of its input parameters `timeout_ms` is default to 0, which means the method returns immediately with any records that are pulled and available in the buffer. You can increase `timeout_ms` to return a larger batch.

By default, each consumer is an infinite listener, so it won't stop until the program breaks. But on the other side, you are allowed to stop the consumer based on the message you received. For example, you can exit the loop and close the consumer if it reaches to a certain offset.

The consumer can also be assigned to a partition or multiple partitions from multiple topics.

```
1  from kafka import KafkaConsumer, TopicPartition
2
3  size = 1000000
4
5  consumer1 = KafkaConsumer(bootstrap_servers='localhost:9092')
6  def kafka_python_consumer1():
7      consumer1.subscribe(['topic1'])
8      for msg in consumer1:
9          print(msg)
10
11 consumer2 = KafkaConsumer(bootstrap_servers='localhost:9092')
12 def kafka_python_consumer2():
13     consumer2.assign([TopicPartition('topic1', 1), TopicPartition('topic2', 1)])
14     for msg in consumer2:
15         print(msg)
16
17 consumer3 = KafkaConsumer(bootstrap_servers='localhost:9092')
18 def kafka_python_consumer3():
19     partition = TopicPartition('topic3', 0)
20     consumer3.assign([partition])
21     last_offset = consumer3.end_offsets([partition])[partition]
22     for msg in consumer3:
23         if msg.offset == last_offset - 1:
24             break
```

kafka-python-consumer.py hosted with ❤ by GitHub

[view raw](#)

[kafka-python-consumer.py](#)

This is the test result of *kafka-python* library. The size of each message is 100 bytes. The average throughput of the producer is 1.4MB/s. The average throughput of the consumer is 2.8MB/s.

Search this file...					
1	no. msg	1K	10K	100K	1M
2	kafka-python-producer-async	0.08s	0.67s	6.72s	68.26s
3	kafka-python-producer-sync	3.27s	30.81s	318.29	x
4	kafka-python-consumer	0.06s	0.78s	3.15s	33.94s

kafka-python-result.csv hosted with ❤ by GitHub [view raw](#)

[kafka-python-result.csv](#)

Confluent-kafka

Confluent-kafka is a high-performance Kafka client for Python which leverages the high-performance C client [librdkafka](#). Starting with version 1.0, these are distributed as self-contained binary wheels for OS X and Linux on PyPi. It supports Kafka version 0.8+. The first release was in May 2016. It's being actively maintained.

Installation

For OS X and Linux, librdkafka is included in the package, there is no need to install it separately.

```
pip install confluent-kafka
```

For windows user, by the time I wrote this article, *confluent-kafka* hasn't supported Python3.8 binary wheels on Windows yet. You will run into the issue of librdkafka. Please check their [release note](#), it's being actively developed. An alternative solution is to downgrade to Python3.7.

Producer

Confluent-kafka has incredible performance in terms of speed. The API design is somewhat similar to *kafka-python*. You can make it synchronous by putting `flush()` inside the loop.


```
1 from confluent_kafka import Producer
2 from python_kafka import Timer
3
4 producer = Producer({'bootstrap.servers': 'localhost:9092'})
5 msg = ('kafkatest' * 20).encode()[:100]
6 size = 1000000
7
8 def delivery_report(err, decoded_message, original_message):
9     if err is not None:
10         print(err)
11
12 def confluent_producer_async():
13     for _ in range(size):
14         producer.produce(
15             "topic1",
16             msg,
17             callback=lambda err, decoded_message, original_message=msg: delivery_report( # noc
18                 err, decoded_message, original_message
19             ),
20         )
21     producer.flush()
22
23 def confluent_producer_sync():
24     for _ in range(100000):
25         producer.produce(
26             "topic1",
27             msg,
28             callback=lambda err, decoded_message, original_message=msg: delivery_report( # noc
29                 err, decoded_message, original_message
30             ),
31         )
32     producer.flush()
```

confluent-kafka-producer.py hosted with ❤ by GitHub

[view raw](#)[confluent-kafka-producer.py](#)

Consumer

The Consumer API in *confluent-kafka* requires more code. Instead of calling a high-level method like `consume()`, you need to handle the while loop yourself. I would recommend you to create your own `consume()` which is essentially a Python generator. Whenever there is a message pulled and available in the buffer, it yields the message.

By doing so, the main function will be clean and you are free to control the behavior of your consumer. For example, you can define a “session window” in `consume()`. If no

messages are pulled within X seconds, then the consumer will stop. Or you can add a flag *infinite=True* as an input parameter to control if the consumer should be an infinite listener or not.

```
1  from confluent_kafka import Consumer, TopicPartition
2  size = 1000000
3
4  consumer = Consumer(
5      {
6          'bootstrap.servers': 'localhost:9092',
7          'group.id': 'mygroup',
8          'auto.offset.reset': 'earliest',
9      }
10 )
11
12 def consume_session_window(consumer, timeout=1, session_max=5):
13     session = 0
14     while True:
15         message = consumer.poll(timeout)
16         if message is None:
17             session += 1
18             if session > session_max:
19                 break
20             continue
21         if message.error():
22             print("Consumer error: {}".format(msg.error()))
23             continue
24         yield message
25     consumer.close()
26
27 def consume(consumer, timeout):
28     while True:
29         message = consumer.poll(timeout)
30         if message is None:
31             continue
32         if message.error():
33             print("Consumer error: {}".format(msg.error()))
34             continue
35         yield message
36     consumer.close()
37
38 def confluent_consumer():
39     consumer.subscribe(['topic1'])
40     for msg in consume(consumer, 1.0):
41         print(msg)
42
```

```

43 def confluent_consumer_partition():
44     consumer.assign([TopicPartition("topic1", 0)])
45     for msg in consume(consumer, 1.0):
46         print(msg)

```

confluent-kafka-consumer.py hosted with ❤ by GitHub

[view raw](#)

[confluent-kafka-consumer.py](#)

This is the test result of *confluent-kafka* library. The size of each message is 100 bytes. The average throughput of the producer is 21.97MB/s. The average throughput of the consumer is 16.8~28.7MB/s.

Search this file...					
1	no. msg	1K	10K	100K	1M
2	confluent-producer-async	0.02s	0.06s	0.53s	4.34s
3	confluent-producer-sync	3.15s	31.12s	309.29s	x
4	confluent-consumer	3.07s	3.08s	3.32s	5.66s

confluent-kafka-result.csv hosted with ❤ by GitHub

[view raw](#)

[confluent-kafka-result.csv](#)

PyKafka

PyKafka is a programmer-friendly Kafka client for Python. It includes Python implementations of Kafka producers and consumers, which are optionally backed by a C extension built on librdkafka. It supports Kafka version 0.82+. The first release was in Aug 2012, but it hasn't been updated since Nov 2018.

Installation

```
pip install pykafka
```

librdkafka doesn't come with the package, you need to install it separately in all the operating systems.

Producer

pykafka has a *KafkaClient* interface that covers both *ProducerAPI* and *Consumer API*.

Messages can be sent in both async and sync mode. What I found out is that *pykafka* modifies the default value of some producer configurations such as *linger_ms* and *min_queued_messages*, which can have an impact on sending a small volume of data.


You can compare it with the default configuration on [Apache Kafka website](#).

If you want to get the callback of each message, make sure you change *min_queued_messages* to 1, otherwise you won't get any report if your data set is smaller than 70000.

```
class pykafka.producer.Producer(cluster, topic, partitioner=<function random_partitioner>,
compression=0, max_retries=3, retry_backoff_ms=100, required_acks=1, ack_timeout_ms=10000,
max_queued_messages=100000, min_queued_messages=70000, linger_ms=5000, block_on_queue_full=True,
sync=False)
```

pykafka-producer-config

```
1  from pykafka import KafkaClient
2  client = KafkaClient(hosts="localhost:9092")
3  topic = client.topics[b'topic1']
4  msg = ('kafkatest' * 20).encode()[:100]
5
6  def pykafka_producer_sync(size):
7      with topic.get_sync_producer() as producer:
8          for i in range(size):
9              producer.produce(msg)
10             producer.stop()
11
12  def pykafka_producer_async(size):
13      with topic.get_producer() as producer:
14          for i in range(size):
15              producer.produce(msg)
16             producer.stop()
17
18  def pykafka_producer_async_report(size):
19      with topic.get_producer(delivery_reports=True, min_queued_messages=1) as producer:
20          for i in range(size):
21              producer.produce(msg)
22              while True:
23                  try:
24                      report, exc = producer.get_delivery_report(block=False)
25                      print(report)
26                  except Exception:
27                      break
28             producer.stop()
```

pykafka-producer.py hosted with  by GitHub


view raw

[pykafka-producer.py](#)

Consumer

You can get a `SimpleConsumer` from the `KafkaClient` interface. This is similar to *kafka-python*, where the poll is wrapped in the `SimpleConsumer` class.


```
1 from pykafka import KafkaClient
2 from pykafka.simpleconsumer import OffsetType
3
4 client = KafkaClient(hosts="localhost:9092")
5 topic = client.topics[b'topic1']
6
7 def pykafka_consumer():
8     consumer = topic.get_simple_consumer(consumer_group="mygroup", auto_offset_reset=OffsetType
9     for message in consumer:
10         print(message)
```

pykafka-consumer.py hosted with  by GitHub


view raw

[pykafka-consumer.py](#)

This is the test result of *pykafka* library. The size of each message is 100 bytes. The average throughput of the producer is 2.1MB/s. The average throughput of the consumer is 1.57MB/s.

 Search this file...

	no. msg	1K	10K	100K	1M
1	pykafka-producer-async	10.04s	15.38s	12.41s	45.69s
2	pykafka-producer-sync	6.93s	24.33s	188.89	x
3	pykafka-consumer	0.26s	0.84s	6.91s	60.69s

pykafka-result.csv hosted with  by GitHub

view raw

[pykafka-result.csv](#)

. . .

Conclusion

So far, I've explained the Producer API and Consumer API of each library. In terms of Admin API, *kafka-python* and *confluent-kafka* do provide explicit Admin API. You can use it in your unit testing where you want to create a topic and then delete it before executing the next test. Besides, if you'd like to build a Kafka monitoring dashboard in Python, Admin API can help you retrieve metadata of the cluster and topics.

Q Search this file...

1	no. msg	1K	10K	100K	1M
2	confluent-producer-async	0.02s	0.06s	0.53s	4.34s
3	confluent-producer-sync	3.15s	31.12s	309.29s	x
4	confluent-consumer	3.07s	3.08s	3.32s	5.66s
5	pykafka-producer-async	10.04s	15.38s	12.41s	45.69s
6	pykafka-producer-sync	6.93s	24.33s	188.89	x
7	pykafka-consumer	0.26s	0.84s	6.91s	60.69s
8	kafka-python-producer-async	0.08s	0.67s	6.72s	68.26s
9	kafka-python-producer-sync	3.27s	30.81s	318.29	x
10	kafka-python-consumer	0.06s	0.78s	3.15s	33.94s

kafka-python-result.csv hosted with ❤ by GitHub [view raw](#)

[kafka-python-result.csv](#)

Confluent-kafka:

Confluent-kafka has no doubt the best performance among the 3 libraries. The API is well designed and parameters keep the same name and same default as the original Apache Kafka. You can easily link it to the original parameter. Personally, I like the flexibility to customize consumer behavior. It is also being actively developed and supported by Confluent.

A disadvantage is the fact that Windows users might need to struggle a bit to make it work. And debug can be tricky because of its C extension.

kafka-python:

kafka-python is a pure Python library without a C extension. The API is well designed and straightforward to use for beginners. It's an actively developed project as well.

The disadvantage of *python-kafka* is its speed. If you do care about the performance, I would recommend you to switch to *confluent-kafka*.

pykafka:

Compared to *kafka-python* and *confluent-kafka*, the development of *pykafka* is less active. The [release history](#) shows that it hasn't been updated since Nov 2018. Besides, *pykafka* has different API designs and uses different default parameters which might be not straightforward for the first time.

I hope you enjoy this article! Leave your comments below if you have any thoughts.

Resources

kafka-python

Python client for the Apache Kafka distributed stream processing system. kafka-python is designed to function much like...

pypi.org



confluent-kafka

Confluent's Python client for Apache Kafka

pypi.org



pykafka

PyKafka is a programmer-friendly Kafka client for Python. It includes Python implementations of Kafka producers and...

pypi.org



Apache Kafka: The Definitive Guide | Confluent

What is Kafka, and how does it work? In this comprehensive e-book, you'll get full introduction to Apache Kafka[®], the...

www.confluent.io



Programming

Data Science

Python

Kafka

Towards Data Science

Medium

[About](#) [Help](#) [Legal](#)

Get the Medium app

