CONFLUENT

Benchmark Your Dedicated Apache Kafka® Cluster on Confluent Cloud

Anna Povzner, Scott Hendricks

Table of Contents

Abstract

Confluent Cloud Dedicated cluster capacity is based on a pricing unit called a CKU (Confluent Unit for Apache Kafka®), where each unit is capable of sustaining 50 MB/s producer bandwidth and 150 MB/s consumer bandwidth. This white paper reports the results of the benchmarks we ran on a 2-CKU multi-zone Dedicated cluster, and shows the ability of a CKU to deliver the stated client bandwidth on AWS, GCP, and Azure clouds. The paper describes the setup up and execution of our benchmarks in sufficient detail to make them reproducible.

Introduction

Confluent Cloud Dedicated cluster capacity is based on a pricing unit called a CKU (Confluent Unit for Apache Kafka®). As of April 2020, one CKU is capable of sustaining 50 MB/s producer bandwidth and 150 MB/s consumer bandwidth, where a higher consumer bandwidth capacity is useful for a publish/subscribe model that lets multiple consumer groups retrieve their own copy of a topic. One CKU also comes with a set of upper limits on your workload behavior. Notable upper limits include the maximum number of simultaneously connected clients (1,000 per CKU) and the maximum number of topic partitions (3,000 per CKU). Increasing your allocated CKUs will linearly increase your upper limits. Independent of the number of CKUs, there is also an upper limit on per-partition producer bandwidth (5 MB/s) and per-partition consumer bandwidth (15 MB/s).

The specific performance of your clients with a CKU will depend on the behavior of your workload. In general, you will not be able to max out all the dimensions of your workload behavior and achieve the maximum CKU bandwidth at the same time. For example, if you reach the partition limit, you will not be able to reach the maximum CKU bandwidth. The benchmark described in the paper provides an example workload that is capable of reaching maximum per-CKU bandwidth. You'll note that our example workload uses 108 partitions, fewer than 20 connections, and specific client configuration. Your own workload is probably different and as a result, your application

bandwidth may be different.

Since the range of possible workload behaviors is large, you may need to benchmark your own Dedicated cluster with the configurations more relevant to your specific workload. This paper describes the benchmark that achieves the maximum CKU bandwidth for all clouds where Confluent Cloud is available: Amazon Web Services (AWS), Microsoft Azure, and Google Cloud Platform (GCP). For benchmarks, we used off-the-shelf Kafka producer and consumer performance tools, `kafka-producer-perf-test` and `kafka-consumer-perf-test`. You can use our benchmarks as a baseline to verify the capabilities of your cluster and also as a starting point for benchmarking a wider range of workload parameters, such as a different number of partitions and clients, different message sizes, and different client configurations.

To make our benchmarks fully reproducible, this paper first provides a step-by-step guide for setting up and running the benchmarks that test the capabilities of a 2-CKU Dedicated cluster, a minimum size for the multi-zone clusters. We present and explain the results of running these benchmarks on three multi-zone clusters, one per cloud (AWS, GCP, and Azure). Our results show that each setup was able to achieve the maximum 2-CKU bandwidth: 100 MB/s producer and 300 MB/s consumer bandwidth.

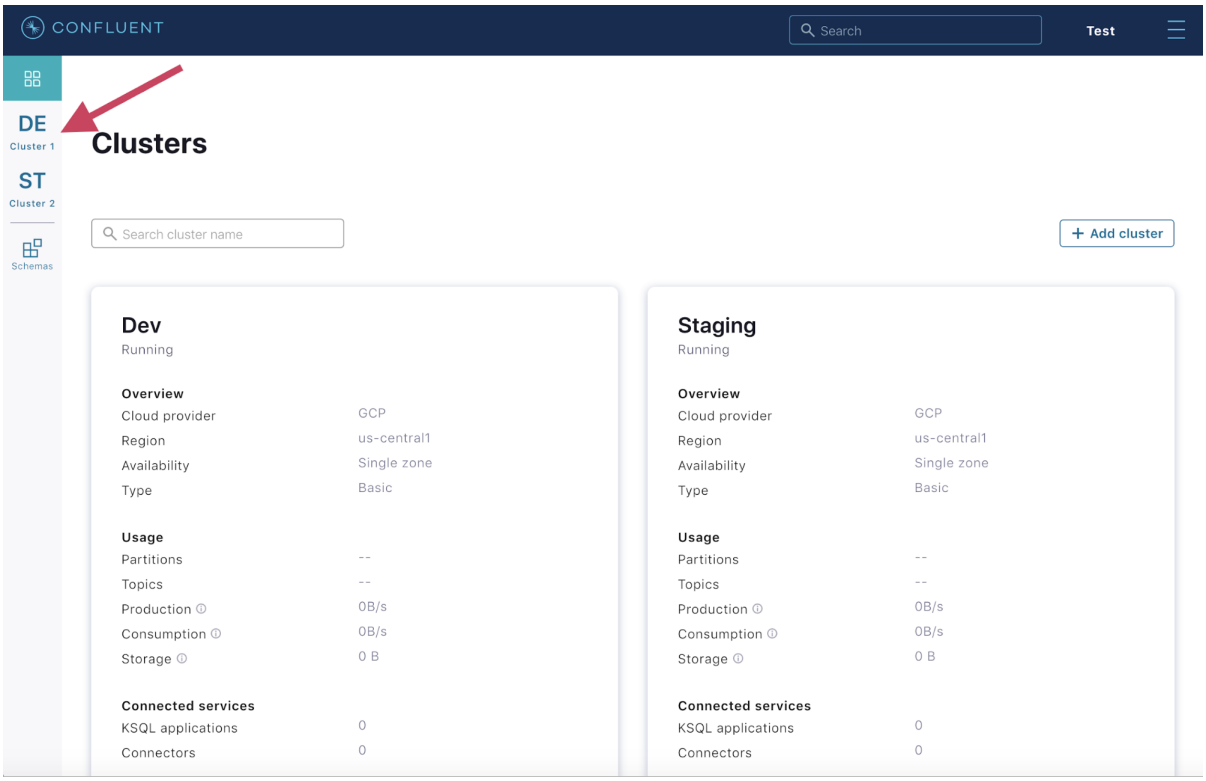
Setting Up Benchmark Clients

This section describes the steps for setting up VMs/containers that would run Kafka performance tools, `kafka-producer-perf-test` and `kafka-consumer-perf-test`, configuring performance tools to be able to access your Dedicated cluster, and the exact commands for running the performance tools. All the steps are relevant for testing any CKU Dedicated cluster, while our specific examples assume testing a 2-CKU cluster.

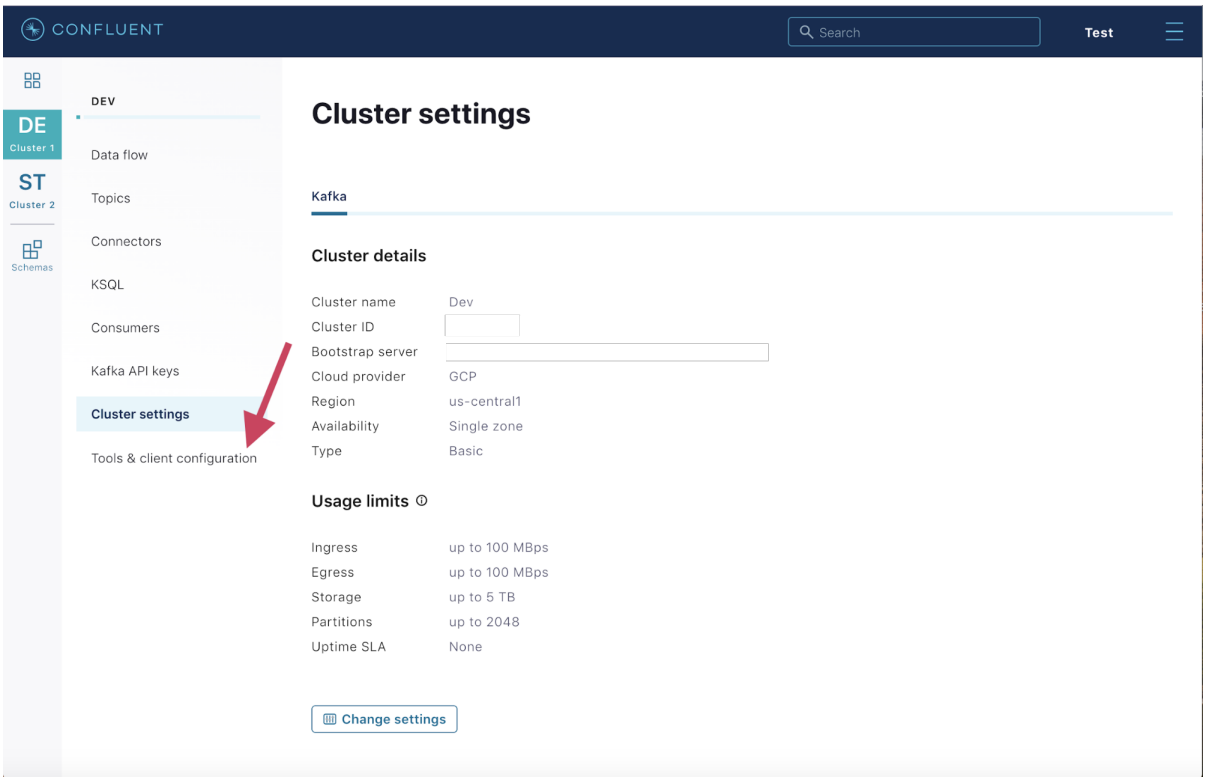
Download Your Java Config for the Cluster from the Confluent Cloud UI

You need to configure performance benchmarks to access your Dedicated cluster. Since `kafka-producer-perf-test` and `kafka-consumer-perf-test` are Java clients, download your Java config for the cluster from the Confluent Cloud UI as follows.

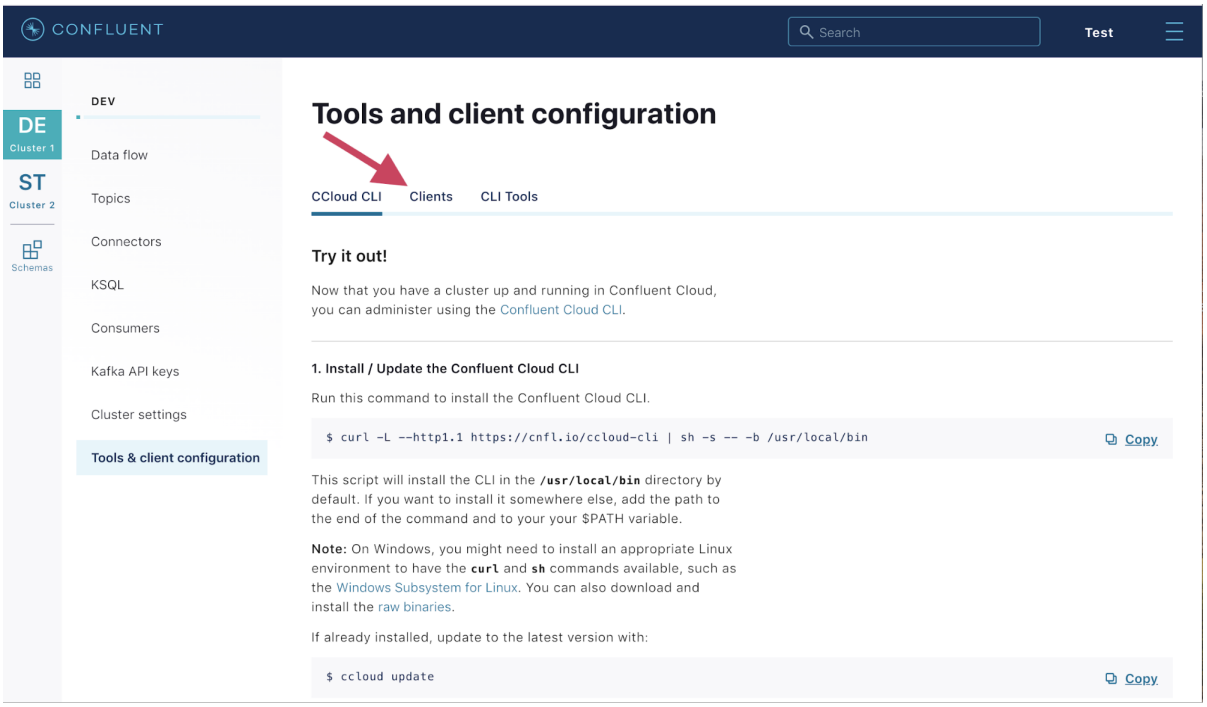
Select your cluster from the navigation bar:



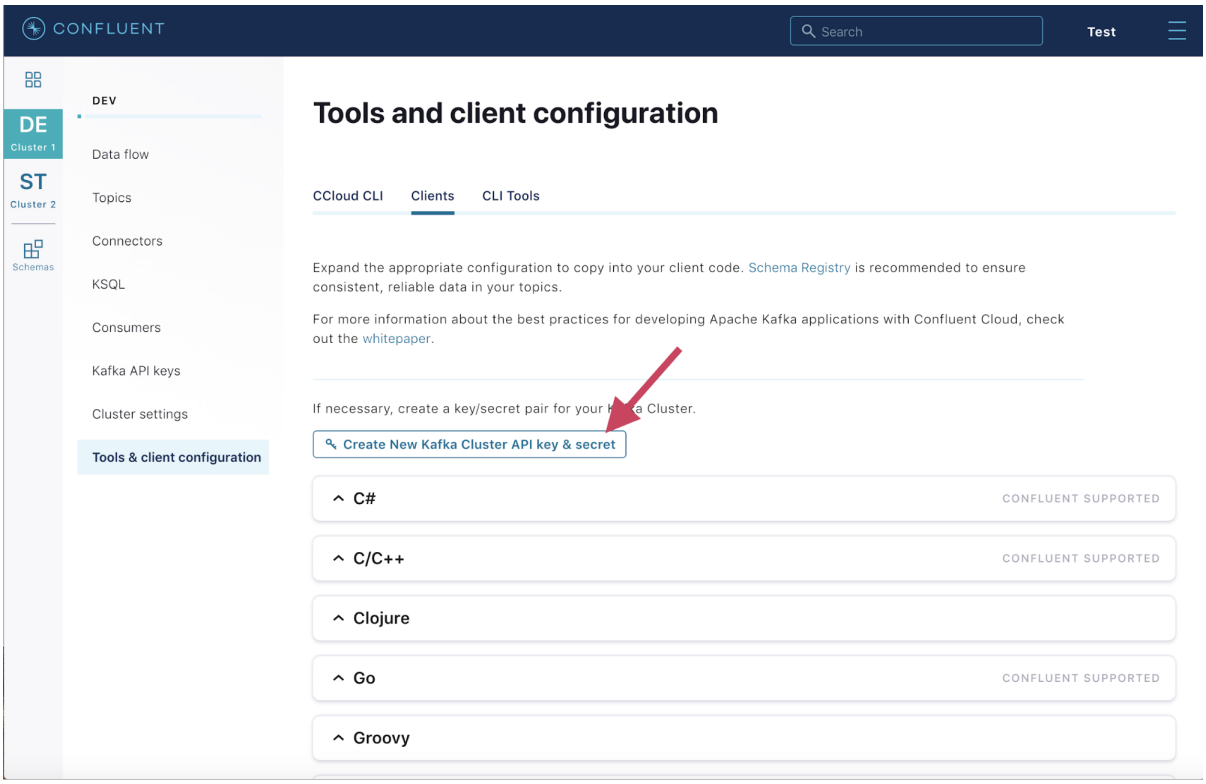
Click on `Tools & client configuration` menu:



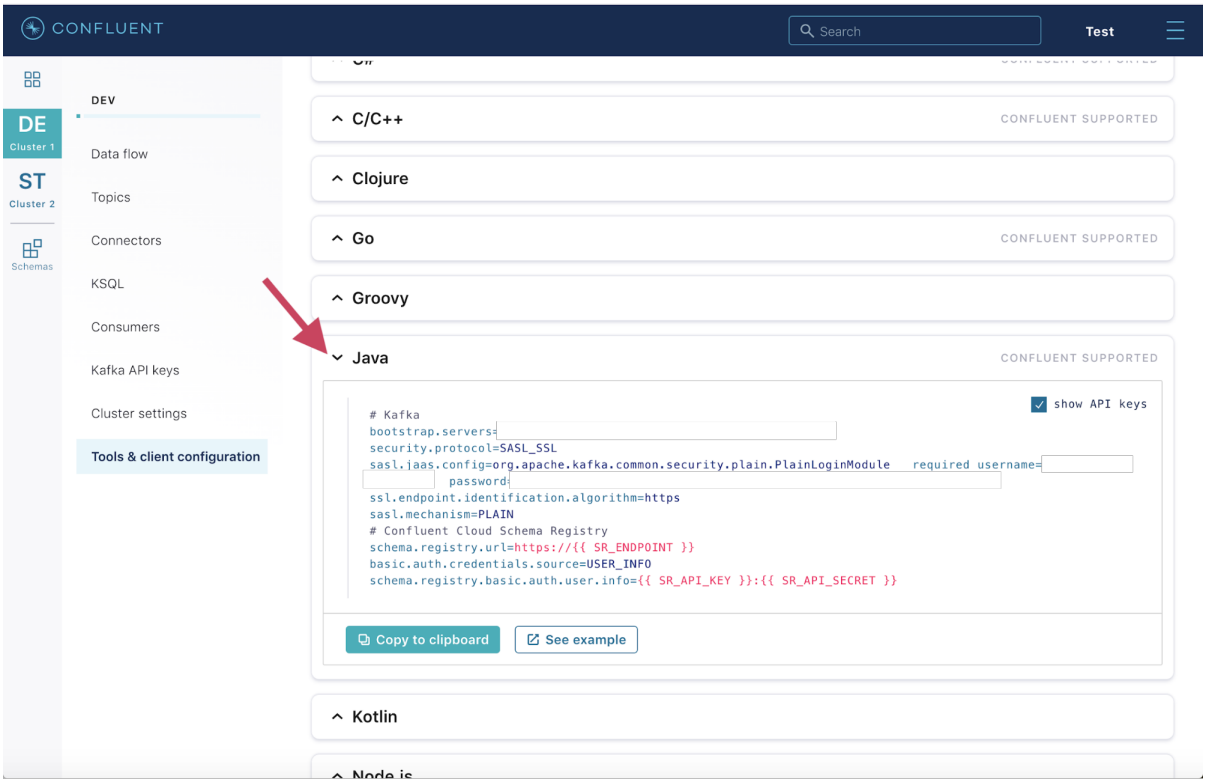
Select `Clients` at the top:



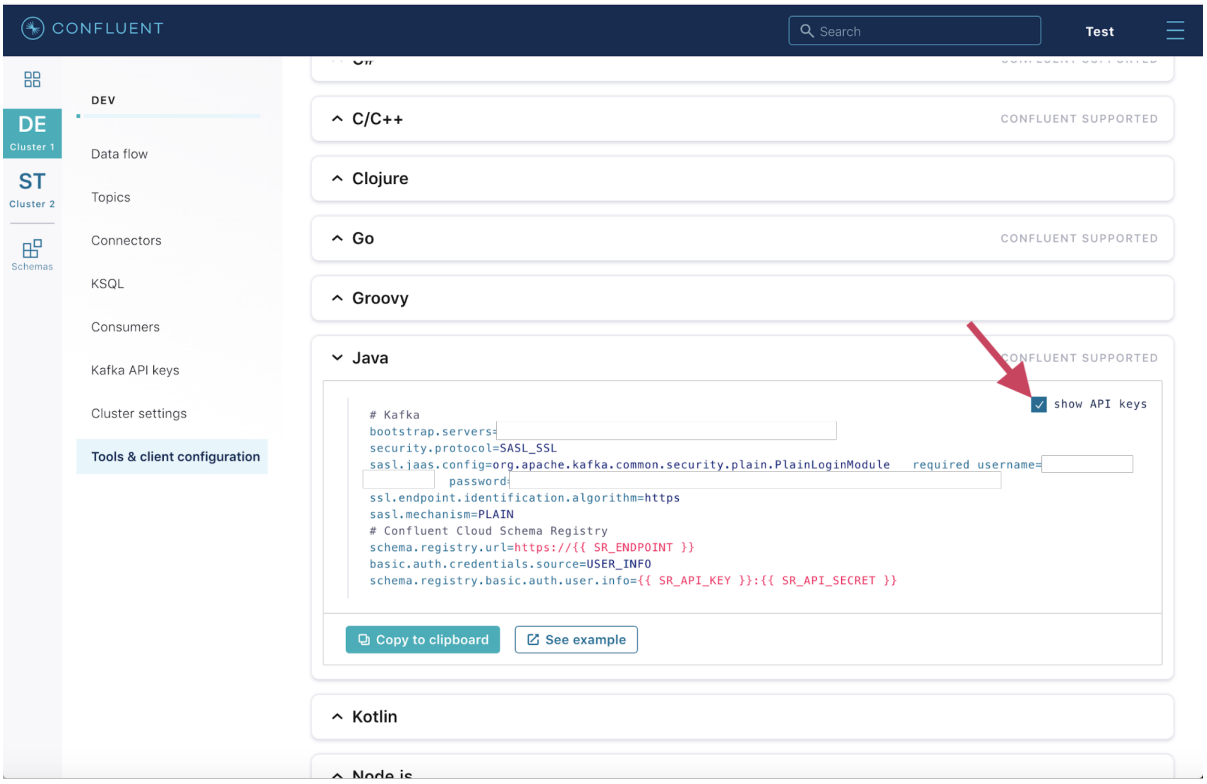
Generate a New Kafka Cluster `API Key & Secret`:



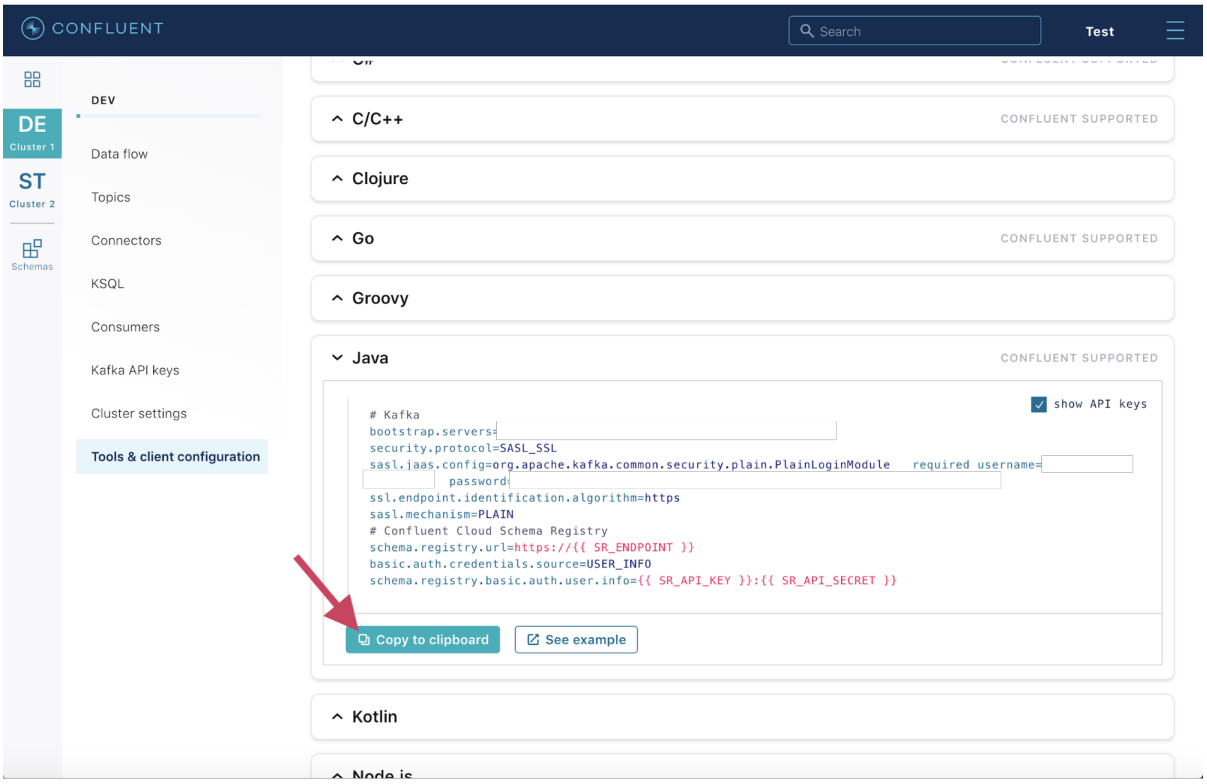
Expand `Java` in the list:



Check box to show API Keys to replace `{{ CLUSTER_API_KEY }}` and `{{ CLUSTER_API_SECRET }}` values with the real API Key and Secret you just created:



Click `Copy to clipboard`:



Save the contents of the **#Kafka** config section for later, ignoring every other section.

Container/VM Creation for Running Performance Benchmarks

The number of containers to run the performance benchmark was chosen as the minimum number of containers that ensures that clients are not a bottleneck, i.e., there is enough performance capacity for the benchmark producers and consumers to push enough bandwidth to the cluster.

Create two clean `confluentinc/cp-kafka` containers or VMs for each CKU in AWS or Azure, or four for each CKU in GCP. These should be running with a minimum of 4-cores and isolated from each other and any other heavy workloads. The test is very CPU intensive, so it is required to run on two separate systems for each CKU on AWS and Azure, and four separate systems for each CKU on GCP.

You should also aim to create these tests in the same cloud region as your Kafka cluster. This will ensure the numbers match our reported results.

Docker

On Docker, you can run the following command to create a new container running `confluentinc/cp-kafka`. You must note the Container IDs output from each command.

```
docker run -d confluentinc/cp-kafka tail -f /dev/null
```

To reproduce the results reported in this paper for a 2-CKU cluster, create four containers on AWS or Azure, and eight containers on GCP.

To get a prompt to run a performance benchmark command in each container, use the following command:

```
docker exec -it <CONTAINER 1 ID> bash
docker exec -it <CONTAINER 2 ID> bash
docker exec -it <CONTAINER 3 ID> bash
docker exec -it <CONTAINER 4 ID> bash
...
```

Kubernetes

On Kubernetes, you can create a pod by running the `kubectl run` command. Make sure you choose a different name for each pod you run.

You may need to tweak the commands below to get them to run on different nodes. For example, on AWS `r5.xlarge` (or any other 32GB node type) you can use the `--requests 'memory=20Gi'` flag to make sure they are on individual nodes. You can

also use the `--port/--hostport` flag combo to schedule these alone, as we have done below.

```
kubectl run kafka-benchmark1 \
    --image confluentinc/cp-kafka \
    --restart Never \
    --port 28130 \
    --hostport 28130 \
    [ --namespace optional-namespace ] \
    -- \
    tail -f /dev/null

kubectl run kafka-benchmark2 \
    --image confluentinc/cp-kafka \
    --restart Never \
    --port 28130 \
    --hostport 28130 \
    [ --namespace optional-namespace ] \
    -- \
    tail -f /dev/null
...
```

To reproduce the results reported in this paper for a 2-CKU cluster, run the above commands to create four pods in AWS/Azure or eight pods in GCP:

```
kafka-benchmark1, …, kafka-benchmark4 [, …, kafka-benchmark8]
```

To get a prompt to run a performance benchmark command in each container, use the following command:

```
kubectl exec -it [ -n optional-namespace ] kafka-benchmark1 -- bash
kubectl exec -it [ -n optional-namespace ] kafka-benchmark2 -- bash
kubectl exec -it [ -n optional-namespace ] kafka-benchmark3 -- bash
kubectl exec -it [ -n optional-namespace ] kafka-benchmark4 -- bash
...
```

Configuration

Important: Copy the configuration you saved from the Confluent Cloud UI to each system in a file called `/config`, making sure you have replaced the parameterized sections.

The file should look something like this, but with your actual values.

```
# Kafka
bootstrap.servers=pkc-12345.us-west-2.aws.devel.cpdev.cloud:9092
security.protocol=SASL_SSL
sasl.jaas.config=org.apache.kafka.common.security.plain.PlainLoginModule required username="REDACTED" password="redacted";
ssl.endpoint.identification.algorithm=https
sasl.mechanism=PLAIN
```

Benchmark Overview

As our benchmark, we used pre-packaged performance testing tools that ship with Kafka: `kafka-producer-perf-test` and `kafka-consumer-perf-test`. We used the following non-default client configurations and workload specifications:

Configuration / Workload Specification	Setting
Client version	Apache Kafka 2.4 / Confluent Platform 5.4
`acks`	`all`

Configuration / Workload Specification	Setting
`linger.ms`	10
`compression.type`	`none` (default)
Producer record size	value = 512 bytes
Number of topic partitions	108
Producer to consumer ratio	1:3

Our benchmarks do not use compression, which is the default client configuration. This shows the ability of a CKU to achieve the stated bandwidth over the wire, independent of compression and its resulting compression ratio. We recommend enabling lz4 or Snappy compression on the producer via `compression.type` producer configuration, which would both let you achieve higher client (pre-compression) bandwidth and save on ingress, egress, and storage costs. The trade-off of using lz4 or Snappy compression is a slightly higher CPU usage on both clients and brokers, while gzip could result in a more significant CPU overhead.

Our benchmark was designed to test a 2-CKU Dedicated cluster. The number of producers and consumers was the minimum number of clients that were able to push enough load given the CPU capabilities of the cloud provider's instance types running the benchmark clients. Our benchmark had four producers (eight on GCP) publishing messages to one topic with 108 partitions, and three consumer groups, each with four consumers (eight on GCP), consuming from the same topic. We chose a producer to consumer ratio of 1:3 in order to get the maximum CKU bandwidth (50:150 MB/s per CKU). You can use other ratios, but you may not get the maximum CKU bandwidth.

The next section will describe how to execute the benchmarks. Notice that consumers

start at the same time as producers, meaning that all consumers will be reading from the tail of the log. If your application has non-real-time consumers, which are reading further back from the log, you may not achieve the full CKU bandwidth because the workload will become more disk bound.

We selected 108 partitions as an example of not too small, not too large. You will need at least 10x the number of CKUs partitions to achieve the maximum CKU bandwidth, to account for per-partition bandwidth limits.

We used producer configuration `linger.ms=10` (default is 0) to ensure good batching of records, which optimizes throughput. We usually recommend `linger.ms` between five and 10 milliseconds if you care more about latencies, and up to 100 milliseconds if you care about throughput but have a lot of clients and partitions which may reduce the ability of each client to batch records together. The record size of 512 bytes was also chosen as large enough to optimize for throughput.

We chose to benchmark a highly fault-tolerant client configuration, because we expect it to be a more common scenario for production workloads: producer configuration `acks=all` (default is 1). Compared to the default, less fault-tolerant configuration, you will see higher producer latencies, because the producer has to wait for the message to be replicated to all the partition's replicas before receiving the acknowledgment.

We provide more suggestions on how to validate performance and optimize your workload at the end of this paper.

Running Benchmarks

This section describes the exact set of steps and commands that we used for benchmarking a 2-CKU cluster.

Create Topic

On only one of the Docker containers or Kubernetes Pods, run the following command

to create the topic. Make sure to replace **<BOOTSTRAP SERVER>:<PORT>** with the actual bootstrap server from your configs.

```
kafka-topics \
    --bootstrap-server <BOOTSTRAP SERVER>:<PORT> \
    --command-config /config \
    --create \
    --topic perf-test \
    --partitions 108 \
    --replication-factor 3
```

Execute Benchmark

To execute the tests, on each system add the following section to a script and run them all simultaneously, staggered by a couple seconds. Make sure to replace **<BOOTSTRAP SERVER>:<PORT>** with the actual bootstrap server from your configs.

Because the final output from the consumers is not a summary of the test, we must use `awk` to calculate the running average of the entire test. The `awk` command sums the incremental `fetch.time.ms` (column $8), and takes the final value for the running sum of `data.consumed.in.MB` (column $3). It then calculates MB/s based on those two values.

You must delete and re-create the topic for every rerun due to the way the consumers are configured here. Also, be sure the tests are not started at identical times on different servers to avoid consumer group race conditions.

AWS and Azure

These tests are designed to run up to 30 MB/s ingress, 90 MB/s egress per test container/VM to the Kafka cluster. Please note, traffic will likely not be equal among all

containers/VMs. Some may run at a higher rate on one and a lower rate on another.

```
kafka-producer-perf-test \
    --topic perf-test \
    --record-size 512 \
    --producer.config config \
    --throughput 60000 \
    --num-records 54000000 \
    --producer-props acks=all linger.ms=10 \
    | tee /tmp/producer &

kafka-consumer-perf-test \
    --broker-list <BOOTSTRAP SERVER>:<PORT> \
    --consumer.config /config \
    --topic perf-test \
    --messages 53950000 \
    --group=perf-test-1 \
    --show-detailed-stats \
    --hide-header \
    --timeout 60000 \
    | tee /tmp/consumer1 &

kafka-consumer-perf-test \
    --broker-list <BOOTSTRAP SERVER>:<PORT> \
    --consumer.config /config \
    --topic perf-test \
    --messages 53950000 \
    --group=perf-test-2 \
    --show-detailed-stats \
    --hide-header \
    --timeout 60000 \
    | tee /tmp/consumer2 &

kafka-consumer-perf-test \
    --broker-list <BOOTSTRAP SERVER>:<PORT> \
```

```
    --consumer.config /config \
    --topic perf-test \
    --messages 53950000 \
    --group=perf-test-3 \
    --show-detailed-stats \
    --hide-header \
    --timeout 60000 \
    | tee /tmp/consumer3 &

wait

echo "Test Results:"
tail -n 1 /tmp/producer | sed 's|.(\(.\))|Producer: \1|g'
echo "Consumer 1:" \
    `cat /tmp/consumer1 \
     | awk -F"," '{if($8>0){msec+=$8};mb=$3}END{print mb*1000/msec}'`\
    "MB/sec"
echo "Consumer 2:" \
    `cat /tmp/consumer2 \
     | awk -F"," '{if($8>0){msec+=$8};mb=$3}END{print mb*1000/msec}'`\
    "MB/sec"
echo "Consumer 3:" \
    `cat /tmp/consumer3 \
     | awk -F"," '{if($8>0){msec+=$8};mb=$3}END{print mb*1000/msec}'`\
    "MB/sec"
```

GCP

Since we run GCP with more client nodes, we need to decrease the amount of traffic sent per node to keep the test at 15 minutes. These tests are designed to run up to 15 MB/s ingress, 45 MB/s egress per test container/VM to the Kafka cluster. Please note, some containers or VMs may run at a higher consumer rate on one and a lower consumer rate on another.

```
kafka-producer-perf-test \
    --topic perf-test \
    --record-size 512 \
    --producer.config config \
    --throughput 30000 \
    --num-records 27000000 \
    --producer-props acks=all linger.ms=10 \
    | tee /tmp/producer &

kafka-consumer-perf-test \
    --broker-list <BOOTSTRAP SERVER>:<PORT> \
    --consumer.config /config \
    --topic perf-test \
    --messages 26950000 \
    --group=perf-test-1 \
    --show-detailed-stats \
    --hide-header \
    --timeout 60000 \
    | tee /tmp/consumer1 &

kafka-consumer-perf-test \
    --broker-list <BOOTSTRAP SERVER>:<PORT> \
    --consumer.config /config \
    --topic perf-test \
    --messages 26950000 \
    --group=perf-test-2 \
    --show-detailed-stats \
    --hide-header \
    --timeout 60000 \
    | tee /tmp/consumer2 &

kafka-consumer-perf-test \
    --broker-list <BOOTSTRAP SERVER>:<PORT> \
    --consumer.config /config \
    --topic perf-test \
    --messages 26950000 \
```

```
    --group=perf-test-3 \
    --show-detailed-stats \
    --hide-header \
    --timeout 60000 \
    | tee /tmp/consumer3 &

wait

echo "Test Results:"
tail -n 1 /tmp/producer | sed 's|.(\(.\))|Producer: \1|g'
echo "Consumer 1:" \
    `cat /tmp/consumer1 \
     | awk -F"," '{if($8>0){msec+=$8};mb=$3}END{print mb*1000/msec}'`\
    "MB/sec"
echo "Consumer 2:" \
    `cat /tmp/consumer2 \
     | awk -F"," '{if($8>0){msec+=$8};mb=$3}END{print mb*1000/msec}'`\
    "MB/sec"
echo "Consumer 3:" \
    `cat /tmp/consumer3 \
     | awk -F"," '{if($8>0){msec+=$8};mb=$3}END{print mb*1000/msec}'`\
    "MB/sec"
```

Example Output

For each test, you should see streaming output while the test is running. This can be largely ignored, but designed to show progress is being made:

```
249231 records sent, 49746.7 records/sec (24.29 MB/sec), 14.1 ms avg latency, 212.0
ms max latency.
2020-03-26 17:47:20:530, 0, 493.1592, 24.1315, 1009990, 49421.2315, 0, 5002, 24.1315,
49421.2315
2020-03-26 17:47:20:630, 0, 496.1494, 24.3395, 1016114, 49847.2306, 0, 5001, 24.3395,
49847.2306
2020-03-26 17:47:25:157, 0, 605.2256, 24.0767, 1239502, 49309.0000, 0, 5000, 24.0767,
49309.0000
251874 records sent, 50364.7 records/sec (24.59 MB/sec), 18.3 ms avg latency, 149.0
ms max latency.
2020-03-26 17:47:25:550, 0, 616.1587, 24.5019, 1261893, 50179.8805, 0, 5020, 24.5019,
50179.8805
2020-03-26 17:47:25:630, 0, 619.0649, 24.5831, 1267845, 50346.2000, 0, 5000, 24.5831,
50346.2000
2020-03-26 17:47:30:157, 0, 729.5664, 24.8682, 1494152, 50930.0000, 0, 5000, 24.8682,
50930.0000
250346 records sent, 50059.2 records/sec (24.44 MB/sec), 11.6 ms avg latency, 130.0
ms max latency.
2020-03-26 17:47:30:550, 0, 739.1626, 24.6008, 1513805, 50382.4000, 0, 5000, 24.6008,
50382.4000
2020-03-26 17:47:30:630, 0, 739.8574, 24.1585, 1515228, 49476.6000, 0, 5000, 24.1585,
49476.6000
2020-03-26 17:47:35:160, 0, 851.5815, 24.3884, 1744039, 49947.4315, 0, 5003, 24.3884,
49947.4315
```

Once the test finishes, the output should resemble this:

```
Producer: 24.79 MB/sec, 16.88 ms avg latency, 1851.00 ms max latency, 7 ms 50th, 57
ms 95th, 211 ms 99th, 566 ms 99.9th.
Consumer 1: 24.7757 MB/sec
Consumer 2: 24.7930 MB/sec
Consumer 3: 24.8054 MB/sec
```

This shows 25 MB/s ingress to the cluster and 75 MB/s egress for this script. Add up the summary output from all the scripts to get the final total.

Topic Cleanup

To clean up and restart, run this command to delete the topic and start over at the top of this section. Make sure to replace **<BOOTSTRAP SERVER>:<PORT>** with the actual bootstrap server from your configs.

```
kafka-topics \
    --bootstrap-server <BOOTSTRAP SERVER>:<PORT> \
    --command-config /config \
    --topic perf-test \
    --delete
```

Results

We ran the benchmarks described in the previous section on 2-CKU multi-zone clusters created in AWS, GCP, and Azure clouds. 2 CKUs are capable of sustaining 100 MB/s producer bandwidth and 300 MB/s consumer bandwidth, but may support larger bandwidth bursts. The bursts may vary over time, and while our results show slightly higher than sustained CKU bandwidth, such bursts are not guaranteed.

In order to demonstrate the maximum throughput we could achieve with our benchmark from a 2-CKU cluster, we configured producers in the benchmarks to produce with the rate slightly higher than sustained 2-CKU rate: 120 MB/s. Keep this in mind when looking at resulting producer latencies, because clients producing with the rate higher than the system's sustained rate causes longer delays on clients. So, if you care about short latencies, make sure that your clients do not try to send more bandwidth than your maximum sustained CKU bandwidth.

The table below reports total producer and consumer bandwidth achieved by running the benchmark on each of the cloud types: AWS, GCP, and Azure.

Cloud	Producer bandwidth (MB/s)	Consumer bandwidth (MB/s)
AWS	117	351
GCP	106	318
Azure	106	318

We were able to achieve higher bandwidth on AWS because periodic log flushes to disk were absorbed by EBS burst balance. On other clouds, bandwidth fluctuated between about 100 MB/s and 150 MB/s, where log flushes to disk caused dips in bandwidth followed by bandwidth spikes due to clients catching up to their throughput rate. In all environments, consumers were able to keep up with producers, resulting in 3x the producer bandwidth (3 consumer groups).

Performance Troubleshooting Tips

Benchmarking Your Dedicated Cluster

If you try our benchmark on your own 2-CKU cluster and get somewhat worse bandwidth than 100 MB/s producer and 300 MB/s consumer bandwidth, make sure that:

- The hypervisors, Docker hosts, or Kubernetes nodes are not overloaded. For example, multiples of your benchmark containers are not placed on the same node.
- Your client benchmarks are not set up in the remote region or not going through an overloaded network or proxy.

- On AWS, it is not caused by a warm-up time for Elastic Load Balancer (ELB), in which case, try re-running the benchmark.

If your cluster is a different size than 2-CKU, you should also be able to get a CKU maximum bandwidth by creating the appropriate number of benchmark containers/VMs as described in the "Setting Up Benchmark Clients" section: twice the number of CKUs on AWS and Azure, and four times the number of CKUs on GCP. On each container, execute performance testing tools exactly as described in this paper for a 2-CKU cluster.

If you checked the above and you are still not seeing 50 MB/s ingress and 150 MB/s egress per CKU, please contact Confluent Support for assistance.

What Performance Should I Expect from My Own Workload?

Your specific performance will depend on many factors including number of clients, partitions, producer partitioning strategy, rate of the connection attempts from your application layer, and so on. If you want to maximize the bandwidth you get from a CKU:

- Make sure that your clients are not a bottleneck, i.e., they are capable of pushing the desired load.
- Keep in mind there is a per-partition bandwidth limit. The limit is soft in a sense that it may be possible to get more than the limit, but achieving a CKU bandwidth may require a minimum number of partitions based on the CKU partition limits.
- You should be able to achieve the maximum CKU bandwidth if your workload is reasonably efficient in using bandwidth, without requiring too many requests or connection attempts. In other words, if your workload behavior starts approaching one or more upper limits defined by the number of your CKUs, your

client bandwidth may decrease as the workload starts saturating CKU processing capacity. In this scenario, you may either improve the behavior of your clients or purchase more CKUs. In general, two CKUs will deliver twice the bandwidth of one CKU.

We recommend the following reading material for further information on designing your client applications for better throughput and/or latency:

- 99th Percentile Latency at Scale with Apache Kafka: while this paper focuses on latency, architecting for low latency also makes your clients more efficient with using bandwidth, which is also discussed in the paper.
- Optimizing Your Apache Kafka Deployment provides a comprehensive list of guidelines for configuring your Kafka deployment to optimize for various goals: throughput, latency, durability, and availability.