Benchmarking Apache Kafka for Cost-Performance on Amazon Web Services

White Paper

Index

[The Basics of Apache Kafka](#)

[Test Setup](#)

[Configuration](#)

[Producer Test Setup and Results](#)

[Consumer Test Setup and Results](#)

[Closing Remarks](#)

Gain 30% cost-performance savings with your Apache Kafka deployments on Arm Neoverse powered AWS Graviton2 processors.

When it comes to cloud computing, Amazon EC2 is an obvious choice for most developers and cloud users. However, the cost of deploying a given workload can vary widely depending on the instance type that you choose. Amazon EC2 provides a wide selection of instance types optimized to fit different use cases with varying combinations of CPU, memory, storage, and networking capacity and give you the flexibility to choose the appropriate mix of resources for your applications.

We've looked at how Amazon EC2 instances based on x86 processors compare to those based on the AWS Graviton2, the latest processor from Amazon Web Services (AWS) to use a 64-bit Arm Neoverse core. Our benchmarks results for workloads such as NGINX, Memcached, Elasticsearch, and many more have consistently shown that AWS Graviton2 instances can deliver significant advantages, in terms of efficiency and throughput, when compared to similarly equipped instances based on x86 processors.

In this paper, we explore the price/performance gains of using AWS Graviton2 to run memory-intensive workloads that process large data sets. We do this by comparing the results of running Apache Kafka, a popular event-streaming platform. Event-streaming workloads can process trillions of events in a day with real-time data analysis, so they need to run on memory-intensive instances that can quickly and efficiently process very large data sets.

For our benchmarks, we tested Apache Kafka on Arm-based Amazon EC2 R6g instances and x86-based R5 instances. We document each step in detail, so you can easily replicate our results in your AWS environment.

The key takeaway is that running Apache Kafka workloads on AWS Graviton2 based instances deliver throughput and latency values that are comparable to that of x86 instances, at a "30% cost-performance advantage. That's a significant cost savings, and a compelling reason to choose AWS Graviton2-based instances for the many use cases in industrial, financial, and consumer applications that make use of event streaming.

The Basics of Apache Kafka

Apache Kafka is an open-source platform for event streaming, designed to manage real-time data feeds. Kafka makes it possible to analyze data that pertains to a particular event – such as when a sensor transmits data, a company deposits money in a bank account, or a person books a hotel room – and then respond to that event in real time.

Kafka workloads can read (subscribe to) and write (publish) real-time events that occur within an application or service and then store them for immediate use or later retrieval. Kafka workloads can also manipulate, process, and react to event streams, either in real time or retrospectively, and can route streams to different destination technologies as needed.

Kafka workloads are used in a wide range of applications that deal with massive amounts of data and require real-time decision-making. In the financial sector, for example, Kafka workloads can be used to process payments, purchase stocks, and complete other transactions in real time, and in the supply chain they can be used for real-time fleet management, shipment tracking, and other logistical tasks. Industrial applications can use Kafka to capture and analyze sensor data from IoT devices and other equipment, and hospitals can use them to monitor patients and ensure timely treatment. Consumer interactions become more responsive with Kafka because hotel bookings, airline purchases, and other retail activities become both more individual and more automatic. Enterprises can use Kafka workloads, too, using them to connect, store, and make data more readily available throughout the organization, and as the foundation for data platforms, event-driven architectures, and microservices.

Test Setup

Our benchmark measures the throughput and latency of writing and reading events on a Kafka cluster. The throughput metric we use is Records Per Second (RPS), and the latency metric is the 99-percentile latency in milliseconds (ms).

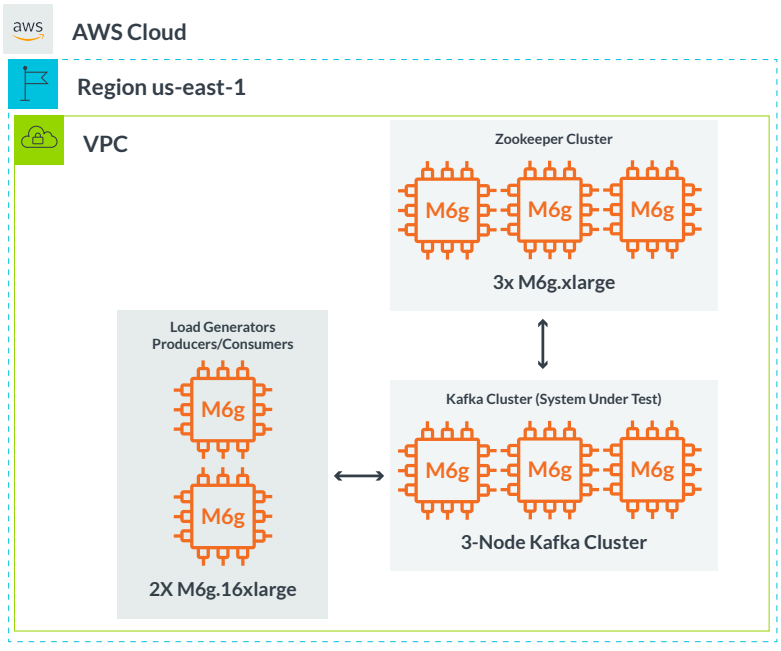
In Kafka, events are ca lled records. Records are written by producers and read by consumers. We wanted to get a sense for the factors that influence producer and consumer performance, so we designed our tests to stress the Kafka cluster as much as possible.

Keep in mind that we used a synthetic testing environment, so results on other use cases may vary from what is shown below.

2.1 Top-Level Test Setup

Below is the AWS setup for our tests.

AWS Test Setup for Kafka Benchmark



The setup includes three components: a Zookeeper cluster, a Kafka cluster, and load generators.

1: Zookeeper cluster

The three-node Zookeeper cluster is required for Kafka operation, and because it is not in the performance critical path, it remained unchanged throughout all tests. To show that Zookeeper works on Arm-based platforms without modification, we ran the cluster on general-purpose m6g.xlarge instances based on the AWS Graviton2 processor.

2: Kafka cluster

The three-node Kafka cluster is responsible for storing and serving records, as well as record replication (durability). This is the portion of the setup we stress tested. The diagram shows that the Kafka cluster is composed of m6g instances. However, as part of our testing we also tested three-node clusters with the instance types and sizes shown in Table 1. Note that the instance type with the lowest cost (USD/hr) is the AWS Graviton2 r6g.xlarge.

Instance Type	Size	Virtual CPUs (vCPUs)	RAM (GiB)	Network (Gbps)	Cost (USD/hr)	Elastic Block Store (EBS) Volume (GiB)	Direct Attached (GB)
r6g	xlarge 2xlarge 12xlarge	4 8 48	32 64 384	10 10 20	0.2016 0.4032 2.4192	512 512 512	None None None
r5	xlarge 2xlarge 12xlarge	4 8 64	32 64 512	10 10 20	0.252 0.504 4.032	512 512 512	None None None
r5a	xlarge 2xlarge	4 8		20	2.4192	512 512	None None
r6gd	2xlarge	4	32	10	0.252	None	1 x 475
r6gd	2xlarge	8	64	10	0.576	None	1 x 300

Table 1. Instances Used for Kafka Cluster 3.

Load generators

We ran the producer and consumer tests on the two load generator instances in the diagram. When evaluating throughput, we used both load generators, but when evaluating latency, we used only one.

The table below lists key SW versions used for testing.

Table 2. Version Information

Ubuntu	20.04	
	Arm AMI	ami-008680ee60f23c94b
	X86-64 AMI	ami-0758470213bdd23b1
Kafka	Kafka 2.6.0 (built with Scala 2.13)	
Zookeeper	Packaged with Kafka 2.6.0	
JDK	openjdk-14-jdk (installed via apt)	

Configuration

3.1 Linux Configuration

Except for some network settings, we left the kernel-level configurations at their defaults. Below are the commands used to change the networking settings.

```
#########
# To view the default value before writing, remove the
assignment.
# For example, to view net.ipv4.ip, run “sysctl net.ipv4.ip_
local_port_range”
sysctl net.ipv4.ip_local_port_range=”1024 65535”
sysctl net.ipv4.tcp_max_syn_backlog=65535
sysctl net.core.rmem_max=8388607
sysctl net.core.wmem_max=8388607
sysctl net.ipv4.tcp_rmem=”4096 8388607 8388607”
sysctl net.ipv4.tcp_wmem=”4096 8388607 8388607”
sysctl net.core.somaxconn=65535
sysctl net.ipv4.tcp_autocorking=0
#########
```

3.2 Kafka Node Storage

For instances without direct-attached storage (r6g, r5, r5a), we used Elastic Block Storage (EBS) volumes of size 512GiB. These volumes were dedicated to Kafka to ensure high performance. For instances with direct-attached storage (r6gd, r5d), we used the direct-attached storage provided by the instance. This storage was also dedicated to Kafka to ensure high performance.

Note: EBS volumes larger than 170GiB are granted more bandwidth by AWS. This is explained in the AWS user guide for EBS volume types.

3.3 Zookeeper Cluster Configuration

Below is the Zookeeper config file used on the three Zookeeper nodes.

```
#############
tickTime=2000
dataDir=/tmp/zookeeper
clientPort=2181
maxClientCnxns=0
initLimit=10
syncLimit=5
server.1=zk_1_ip:2888:3888
server.2=zk_2_ip:2888:3888
server.3=zk_3_ip:2888:3888
#############
```

3.4 Kafka Cluster Configuration

Below is the Kafka config file used on the three Kafka nodes.

```
#############

# Config parameters are documented at https://kafka.apache.org/
documentation/#configuration

############################## Base Settings
##############################

broker.id=unique_id

zookeeper.connect=zk_1_ip:2181,zk_2_ip:2181,zk_3_ip:2181

zookeeper.connection.timeout.ms=60000

############################## Log Basics
##############################

log.dirs=/data/kafka-logs

############################## Socket Settings
##############################

listeners=PLAINTEXT://:9092

num.network.threads=24

num.io.threads=32

socket.send.buffer.bytes=-1

socket.receive.buffer.bytes=-1

############################## Group Coordinator Settings
##############################


group.initial.rebalance.delay.ms=0

#############
```

3.5 Topic Configuration

Kafka requires records to be stored in topics. We created separate topics for the producer and consumer tests. We also deleted the topic after each test run.

Below is the creation command for the producer test topic.

```
#######

./kafka_2.13-2.6.0/bin/kafka-topics.sh --create --topic
producer-load-test --bootstrap-server node_1_ip:9092,node_2_
ip:9092,node_3_ip:9092 --replication-factor 3 --partitions 64

#######

And here is the creation command for the consumer test topic.

########

./kafka_2.13-2.6.0/bin/kafka-topics.sh --create --topic
consumer-load-test --bootstrap-server node_1_ip:9092,node_2_
ip:9092,node_3_ip:9092 --replication-factor 3 --partitions 64

########
```

A few notes about the command switches used in the test topic commands:

--topic

This is the name of the topic we are creating.

--replication-factor

Set to three since we have three nodes in the cluster. Each node will contain a copy of each record committed into the cluster. Selecting a replication factor of three increases the stress on the cluster because the follower nodes are required to act as internal consumers to replicate records.

--partitions

Set to 64 because we found that this yields the highest throughput.

Producer Test Setup and Results

4.1 Producer Test Command and Configuration

We used the performance tests that come packaged with Kafka releases. The producer performance test is in the Kafka bin directory, under the name kafka-producer-perf-test.sh. Below is a sample command line for the producer test.

```
#########
./kafka_2.13-2.6.0/bin/kafka-producer-perf-test.sh --print-
metrics --topic producer-load-test --num-records num_records
--throughput -1 --record-size 100 --producer-props bootstrap.
servers=node_1_ip:9092,node_2_ip:9092,node_3_ip:9092 acks=1
buffer.memory=67108864 batch.size=65536 linger.ms=3
#########
```

Four instances of the above command were run simultaneously. Since we had two load generators, we ran two instances of this test on each load generator. We found this maximized load on the Kafka cluster. To determine throughput (RPS), we aggregated the reported throughput of each of the producer instances.

To determine latency, it does not make sense to aggregate latency percentiles, so we ran a single producer with a single instance of the above command. As a result, our latency test results should be taken as a best case.

A few comments about the switches in the sample command:

--topic

Selects the topic we write our records into.

--num-records

The number of records to write is shown as a variable because we tested with varying record counts.

--throughput

Set to -1, which means the test executed with no limiting on the request rate. In our testing, we found that not using rate-limiting resulted in the highest bandwidth (i.e., highest RPS).

--record-size

We used 100-byte records for all tests. We selected a small record size because it stresses things like vCPU, memory, interconnect, etc. more than larger records, which tend to stress the network interface.

--producer-props

These are the IP addresses of the three Kafka nodes.

--acks

Used to select how many Kafka nodes need to commit or replicate records before the leader node sends an acknowledgement to the producer. This setting governs the durability of records. A value of 1 is a balanced approach on durability versus performance and is a common setting.

--batch.size and --linger.ms

These are batch size and linger time. Through experimentation, we found that we gained throughput when we increased the batch size and added some linger time. This allowed the producer to group multiple records together and send them in one transaction, saving on transaction overhead.

4.2 Producer Test Results

Below is a graph and table illustrating producer RPS across the various instances tested, with RPS on the Y-axis and total records written on the X-axis (all values in millions). Since we used 100-byte records, we can also calculate the total amount of data written per test. From left to right on the X-axis, the total amount of data written is 20GB, 40GB, 160GB, 320GB, and 480GB.

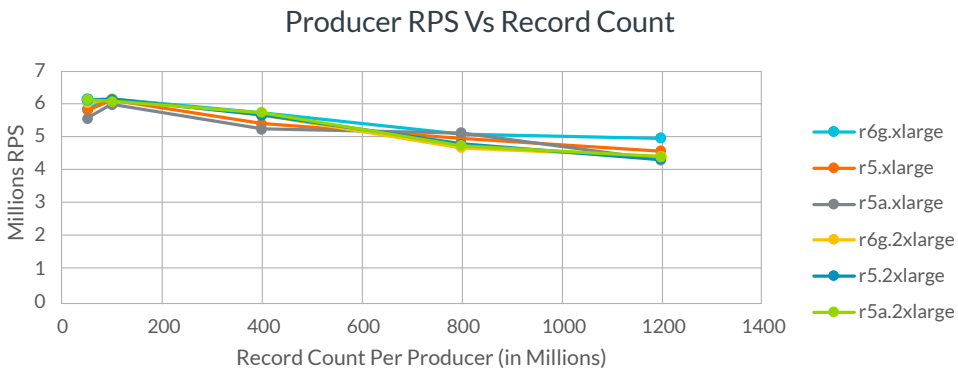


Table 3. Producer RPS Vs Record Count

Record Count	r6g.xlarge	r5.xlarge	r5a.xlarge	r6g.2xlarge	r5.2xlarge	r5a.2xlarge
50000000	5847850	5804823	5561190	6026173	6129683	6118690
100000000	6136060	6110560	5982410	6134607	6133523	6081900
400000000	5719797	5399067	5233680	5666380	5660707	5734730
800000000	5098367	4942530	5120800	4664100	4791267	4730640
1200000000	4955993	4571047	4291720	4391483	4315780	4391910

There are a few things to note from the data above. First, we see a slight upward slope between the first (50 million records) and second (100 million records) data point in the graph across most of the instances. This appears due to a warmup period on a fresh Kafka cluster. If we were to run the first point (50 million records) a second time, we would see the RPS increase to match the results of the second point (100 million records). Since we delete all topics between each test run, this warmup seems related to the Java Virtual Machine or other factors external to the topics and records we write/read during a test. Practically speaking, we should consider the RPS of the first point to be equal to the second point.

Next, notice how the lines are clustered together. The average RPS lines trend downward as we test with higher record counts (i.e., write more data per test). This is because storage performance becomes a bigger factor when we test with more records. This is how we expect Kafka to work. Kafka writes its records into files. When we write to a file, the record is usually not written to storage immediately, it is typically written to the OS page cache in memory first. The OS decides to flush the data from memory to storage when either a certain amount of time has passed or when a certain percentage of the page cache is full (note: there are OS settings that can adjust this behavior; we left the defaults). What this means for our test results is that when you write a small amount of data during the test (e.g., 50 million or 100 million records), we are mostly writing to memory. However, when we write a large amount of data during the test (e.g., 1.2 billion records), we are writing to both memory and storage. We know this because when we test with large record counts, we can see IOWait periodically spike to 95%+ during test execution. On the other hand, when we test with smaller record counts, we do not see any IOWait spikes during test execution. Since storage performance is lower than memory performance, we see the average RPS drop due to the flushes to storage. This tells us that vCPU performance is not as big a factor as we might have expected. As a result, the best instance to use here is the one with the lowest cost, r6g.xlarge.

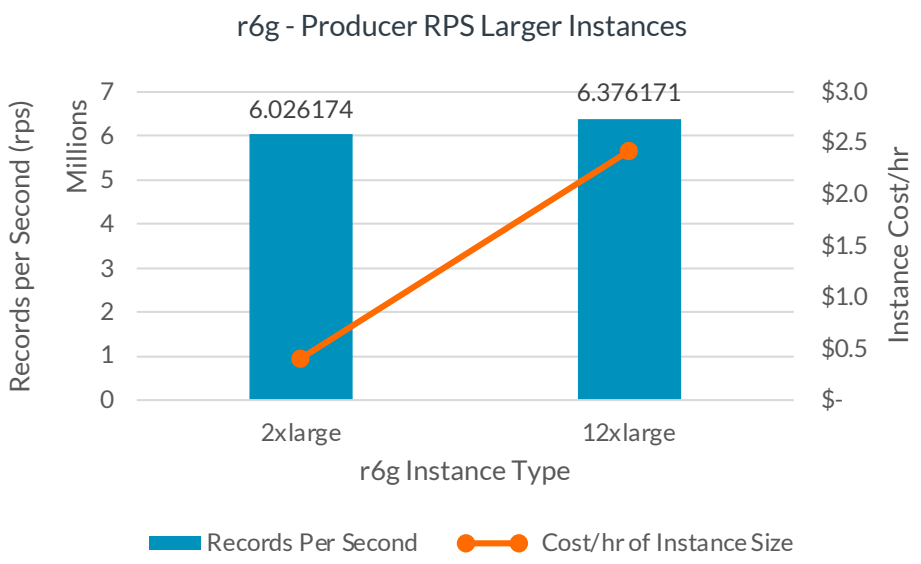
Below is a graph and tables showing the percent improvement between the r6g instances, and the r5 and r5a instances.

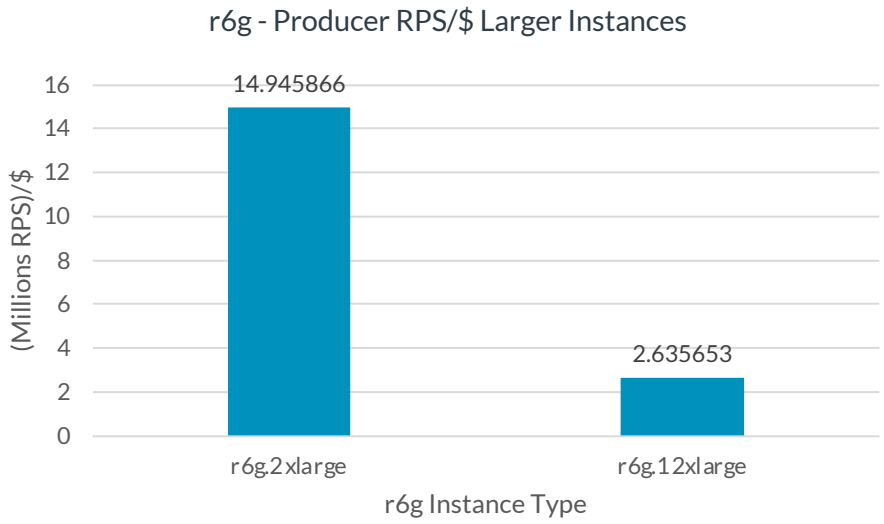
Instance Type and Size - Performance Percent Improvement				
Instance Type	**r6g.xlarge vs r5.xlarge**	**r6g.xlarge vs r5a.xlarge**	**r6g.2xlarge vs r5.2xlarge**	**r6g.2xlarge vs r5a.2xlarge**
50000000	**0.74**	**5.15**	**-1.69**	**-1.51**
100000000	**0.42**	**2.57**	**0.02**	**0.87**
400000000	**5.94**	**9.29**	**0.10**	**-1.19**
800000000	**3.15**	**-0.44**	**-2.65**	**-1.41**
1200000000	**8.42**	**15.48**	**1.75**	**-0.01**
AVG	**3.73**	**6.41**	**-0.49**	**-0.65**

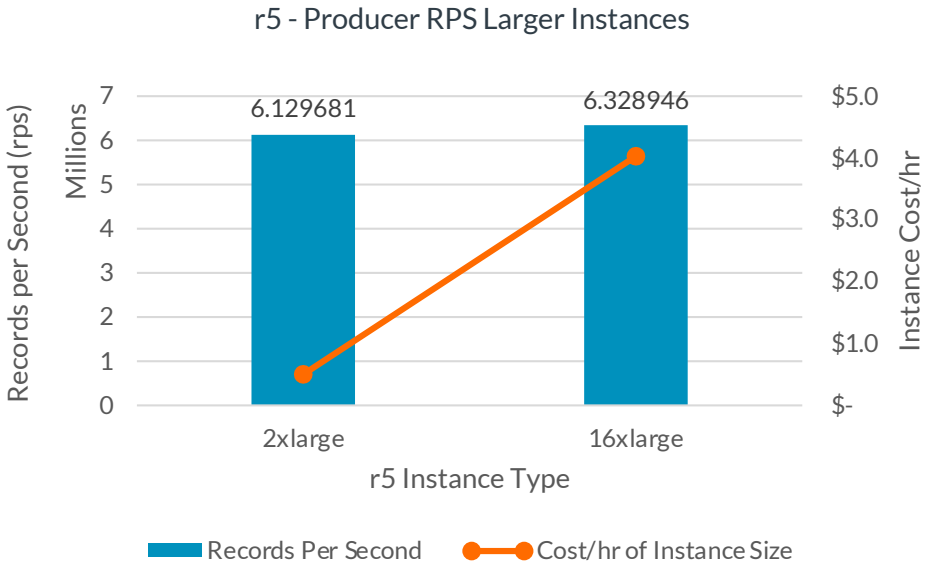
Table 4. Producer RPS Percent Improvement Vs Record Count

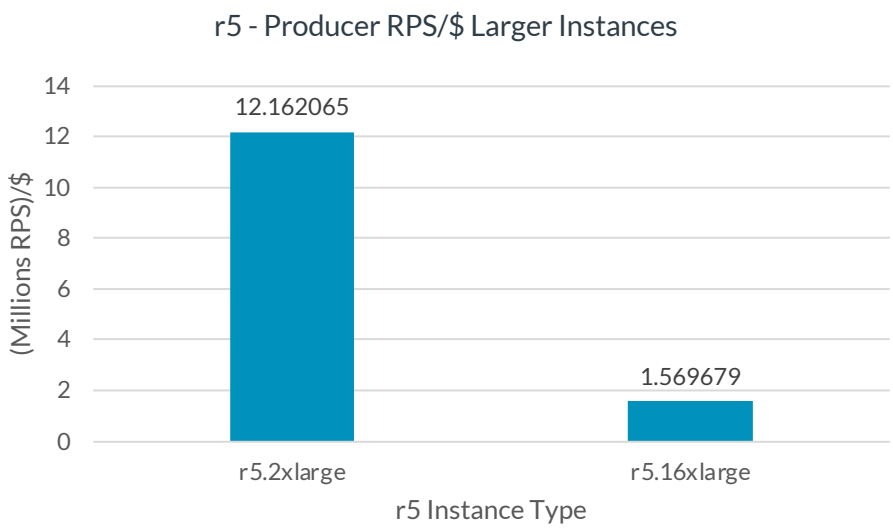
The performance percentage data shows that the r6g.xlarge instances outperform the r5.xlarge and r5a.xlarge instances. On average across all the record counts tested, the r6g.xlarge has about a 3.73% performance advantage over the r5.xlarge, and about a 6.41% performance advantage over the r5a.xlarge. When we look at the 2xlarge instances, on average we see the r6g slightly underperforms. However, this could also be run-to-run variation given that we do see the r6g.2xlarge outperform in some of the test runs.

Given these results, we concluded that it is not helpful to use larger instances. The below graphs illustrate this point. We tested 50 million records on a AWS Graviton2 r6g.12xlarge and an x86 r5.16xlarge. Both instances have a network connection of 20Gbps.


r6g - Producer RPS Larger Instances
Records per Second (rps)
Millions
7
6
5
4
3
2
1
0
6.026174
6.376171
$3.0
$2.5
$2.0
$1.5
$1.0
$0.5
$-
Instance Cost/hr
2xlarge
12xlarge
r6g Instance Type
Records Per Second
Cost/hr of Instance Size


r6g - Producer RPS/$ Larger Instances
(Millions RPS)/$
16
14
12
10
8
6
4
2
0
14.945866
2.635653
r6g.2xlarge
r6g.12xlarge
r6g Instance Type


r5 - Producer RPS Larger Instances
Records per Second (rps)
Millions
7
6
5
4
3
2
1
0
6.129681
6.328946
$5.0
$4.0
$3.0
$2.0
$1.0
$-
Instance Cost/hr
2xlarge
16xlarge
r5 Instance Type
Records Per Second
Cost/hr of Instance Size

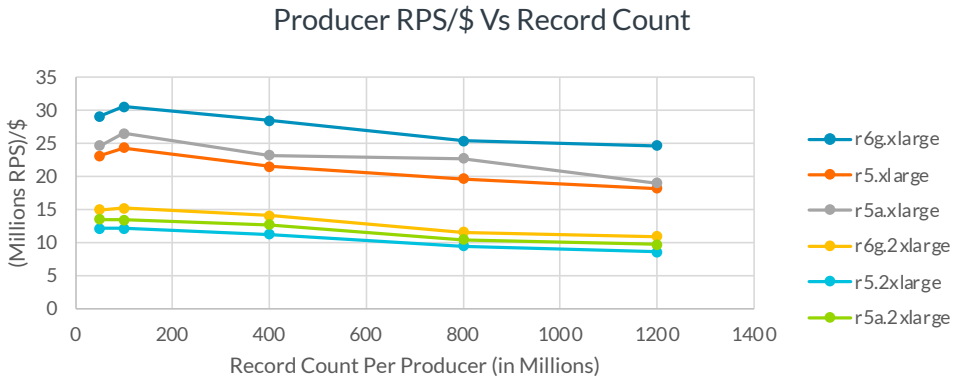


When we compare the r6g.2xlarge to the r6g.12xlarge, we see an increase in RPS of about 0.58%, but at a 6x higher cost. When we compare the r5.2xlarge to the r5.16xlarge, we see an RPS increase of about 3.2%, but at an 8x higher cost. The RPS/$ graphs further show the significant difference in value between the smaller and larger instances. This shows that using larger instances for a Kafka cluster is a poor value. We did not test the r5a.24xlarge (20Gpbs), but we should expect equivalent results.

To explore cost-performance, we divided the results shown in Table 3 by the instance cost listed in Table

1. The result is shown in the graph and Table.

Table 5. Producer RPS/$ Vs Record Count

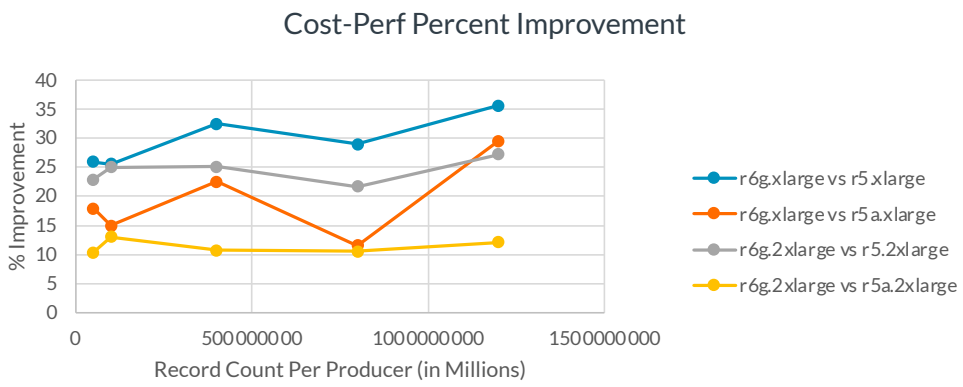


Instance Type and Size - (Request Per Second)/$ by Instance Type and Size						
Record Count	**r6g.xlarge**	**r5.xlarge**	**r5a.xlarge**	**r6g.2xlarge**	**r5.2xlarge**	**r5a.2xlarge**
50000000	29007192	23035012	24607035	14945866	12162069	13536925
100000000	30436806	24248254	26470841	15214799	12169688	13455531
400000000	28372009	21424869	23157876	14053522	11231562	12687456
800000000	25289519	19613214	22658407	11567708	9506482	10466018
1200000000	24583299	18139075	18989912	10891575	8563056	9716615

Table 5. Producer RPS/$ Vs Record Count

From the data in the Table 5, we created the cost-performance graph and table below.

Table 5. Produc From the data in the Table 5, we created the cost-performance graph and table below er RPS/$ Vs Record Count

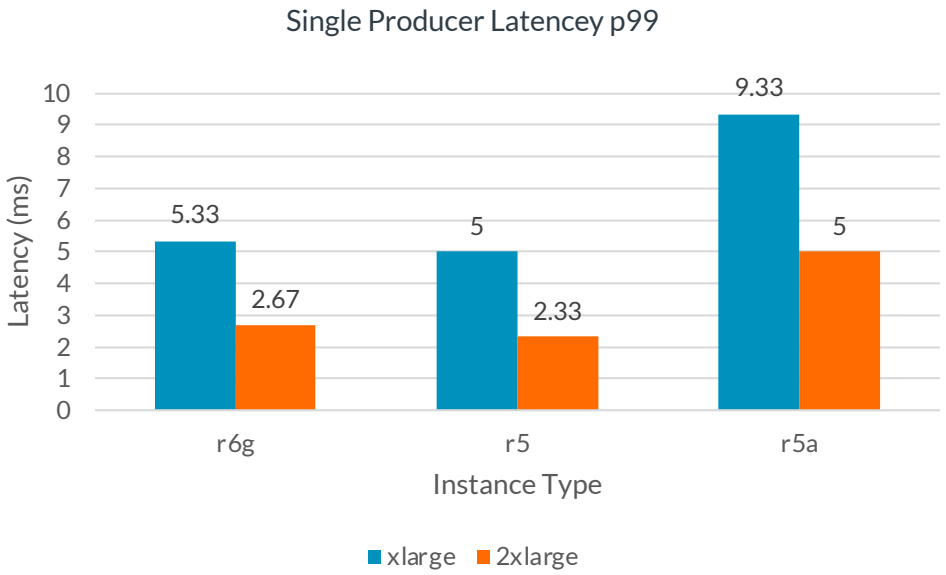


Instance Type and Size - (Request Per Second)/$ by Instance Type and Size				
Record Count	**r6g.xlarge**	**r5.xlarge**	**r5a.xlarge**	**r6g.2xlarge**
50000000	25.93	17.88	22.89	10.41
100000000	25.52	14.98	25.02	13.07
400000000	32.43	22.52	25.13	10.77
800000000	28.94	11.61	21.68	10.53
1200000000	35.53	29.45	27.19	12.09
AVG	29.67	19.29	24.38	11.37

Table 6. Producer Cost-Performance Percent Improvement

The cost-performance data shows that the xlarge instances are a better value for running Kafka. Of the xlarge instances, we see that the r6g is the best value. On average across all the record counts tested, the r6g.xlarge has about a 29.7% cost-performance advantage over the r5.xlarge, and about a 19.3% cost-performance advantage over the r5a.xlarge.

Below are the results for latency running a single producer with 50 million records.



As noted in the test setup section, these latency results should be taken as a best case. Overall, we see that the 2xlarge instances appear to have lower latency than the xlarge instances. This difference comes from the way we tested latency. We decided to run the test three times and then averaged the P99 result of each run. We did this because when we look at the individual run results (not shown), we see that for each instance, the first run is taken during the warm-up period (we saw this in the throughput results above). During the warmup period, latency is higher, and this pulls the P99 up. However, for the 2xlarge instances, the warmup period is shorter, which explains the lower latency for the 2xlarge instances. Given this observation, we decided to average the P99 because it allowed us to couple latency during warmup, latency after warmup, and the warmup period into a single value.

Consumer Test Setup and Results

5.1 Consumer Test Command and Configuration

The consumer performance test is available in the Apache Kafka GitHub repository, in the release bin directory, under the name kafka-consumer-perf-test.sh. Below is a sample command line for the consumer test.

```
########

./kafka_2.13-2.6.0/bin/kafka-consumer-perf-test.sh --print-
metrics --topic consumer-load-test --bootstrap-server node_1_
ip:9092,node_2_ip:9092,node_3_ip:9092 --messages num_records
--threads 10 --num-fetch-threads 1 --timeout 6000000

#########
```

64 instances of the above command were executed simultaneously. Since we have two load generators, each load generator ran 32 instances of this command. To determine throughput, we aggregated the reported throughput from each of the 64 consumer instances. We selected 64 consumer instances because this matches the number of partitions in the topic. We experimented with higher and lower numbers of consumer instances and found that 64 gave us the highest throughput. Going higher than 64 had no effect on the throughput, even when we increased partitions to match.

Another point to note is that the command line has switches called --threads and --num-fetch-threads (set to their defaults above). We experimented with these and found that they had no effect. Therefore, we opted to run multiple instances of the test script rather than rely on this broken (or misunderstood) threading option.

A few comments about the switches in the sample command:

--messages

The number of records to be read. Before we started the test, we wrote this number of records into the topic. In our tests, the prewritten messages are 100 bytes in size. As with the producer test, we chose a small record size because it stresses things like vCPU, memory, interconnect, etc. more than larger records, which tend to stress the network interface.

--threads

The number of processing threads. We used the default setting because we found that changing this number did not change the results.

--num-fetch-threads

The number of fetch threads. We used the default setting of 1 because changing this number did not change the results.

5.2. Consumer Test Results

Below is a graph showing Consumer RPS across the various instances tested, with the RPS on the Y-axis and total records read on the X-axis. Since we used 100-byte records, we can also calculate the total amount of data read per test. From left to right on the X-axis, the total amount of data read is 32GB, 64GB, 128GB, and 256GB.

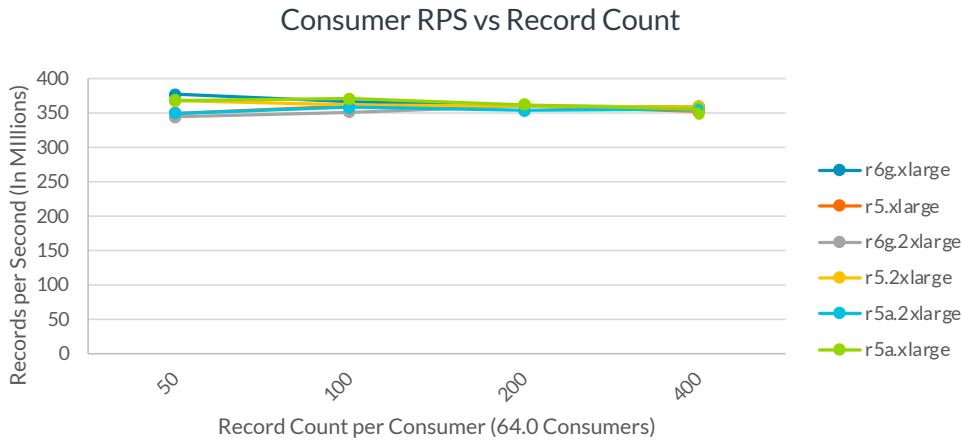
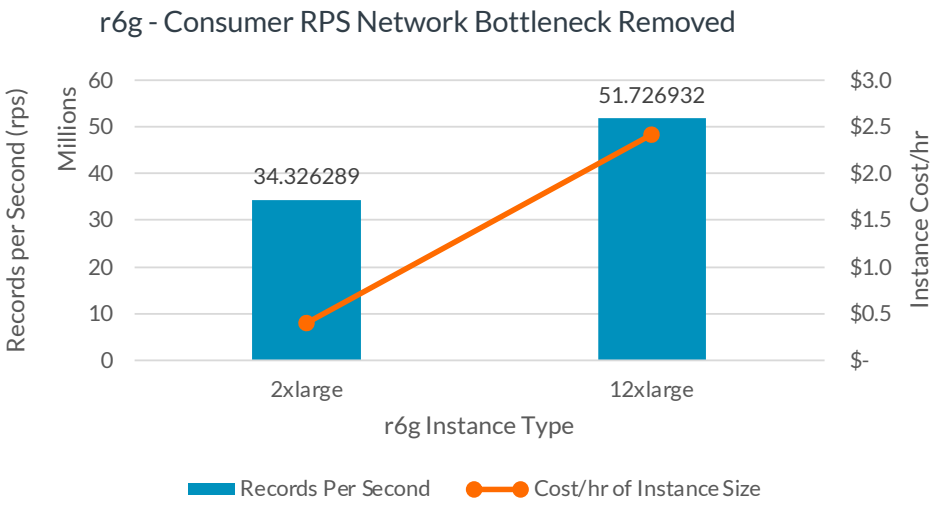
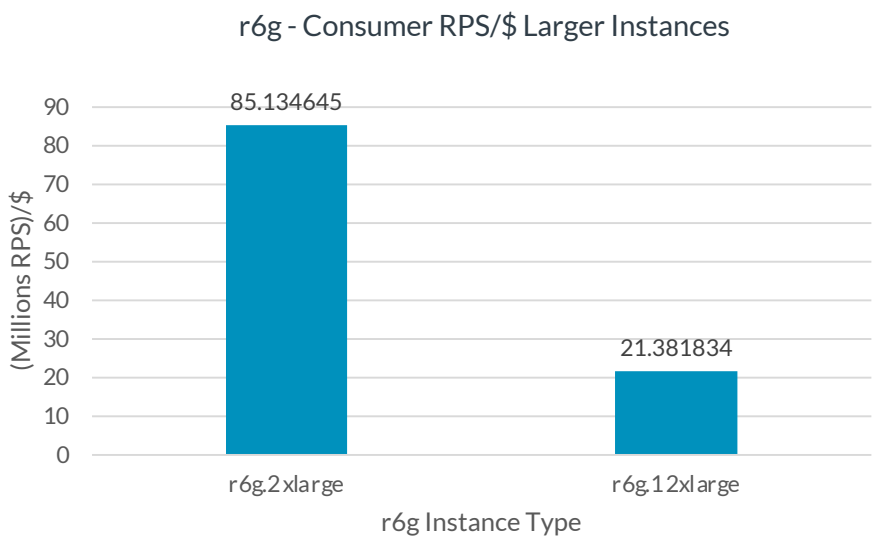


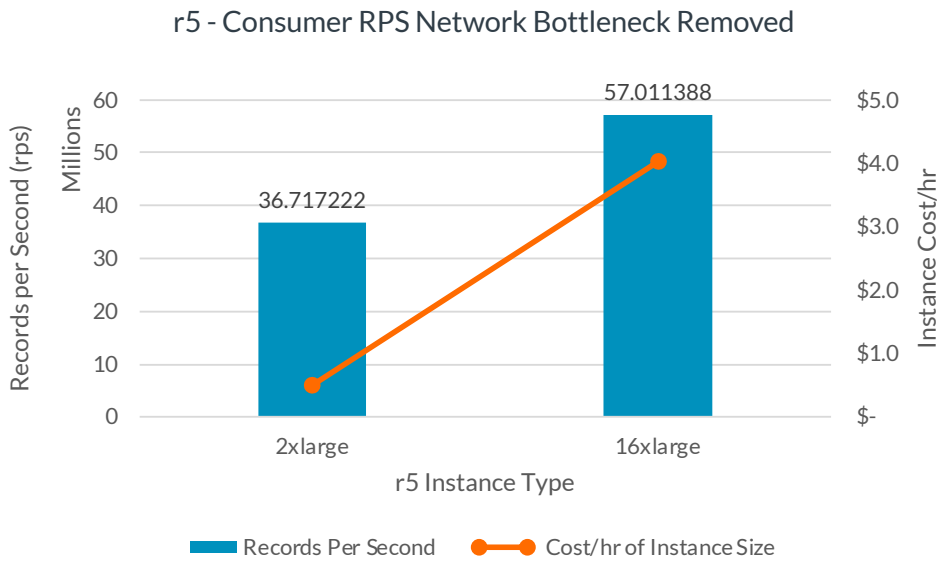
Table 7. Consumer RPS Vs Record Count

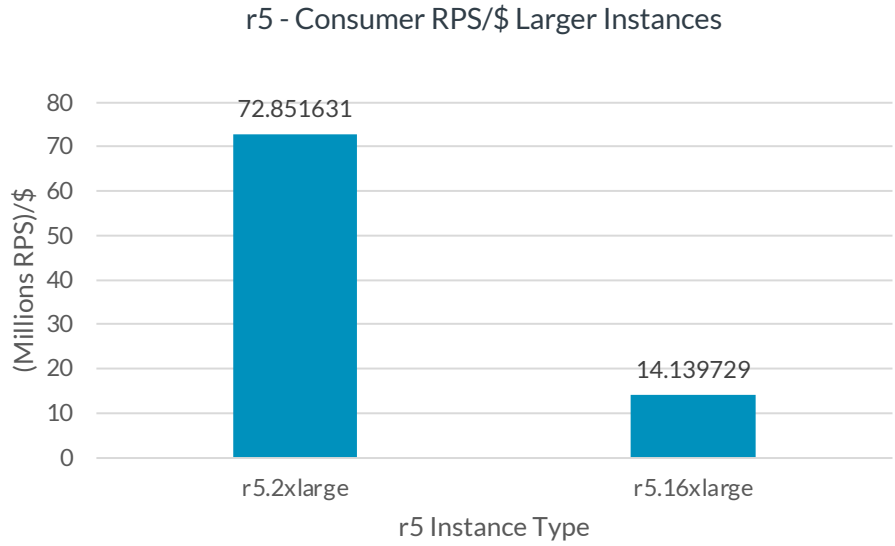
Instance Type and Size - Request Per Second by Instance Type and Size						
Record Count	**r6g.xlarge**	**r5.xlarge**	**r5a.xlarge**	**r6g.2xlarge**	**r5.2xlarge**	**r5a.2xlarge**
5000000	37599800	34930633	36768425	34326289	36717222	34892250
10000000	36637633	35930167	36977333	35034267	36116167	35726033
20000000	35969800	36027767	36115200	35953267	35963633	35297533
40000000	35828567	35154200	35520833	35399667	35849800	35524800

The above results show all the instances at roughly the same RPS. This is because all these instances are network-limited at 10Gbps. Below are the results when we use the bigger r6g.12xlarge (20Gbps) and r5.16xlarge (20Gbps) instances.




r6g - Consumer RPS/$ Larger Instances
(Millions RPS)/$
90
80
70
60
50
40
30
20
10
0
85.134645
21.381834
r6g.2xlarge
r6g.12xlarge
r6g Instance Type

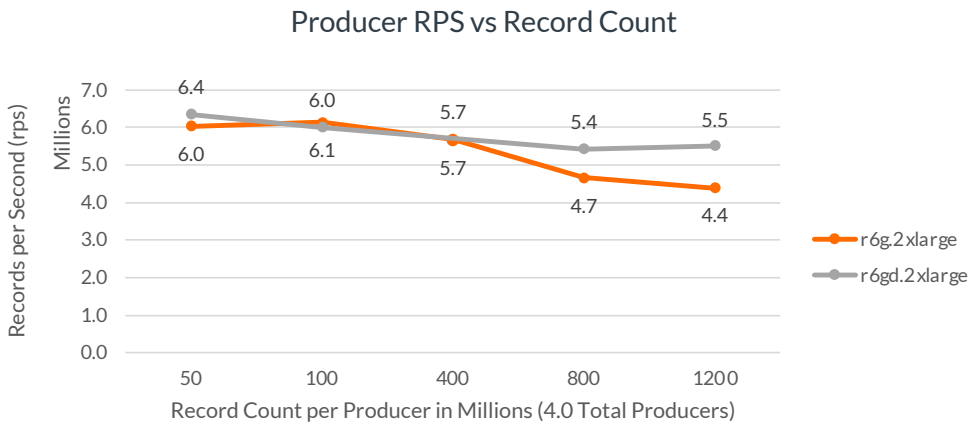

r5 - Consumer RPS Network Bottleneck Removed
Records per Second (rps)
Millions
60
50
40
30
20
10
0
36.717222
57.011388
$5.0
$4.0
$3.0
$2.0
$1.0
$-
Instance Cost/hr
2xlarge
16xlarge
r5 Instance Type
Records Per Second
Cost/hr of Instance Size

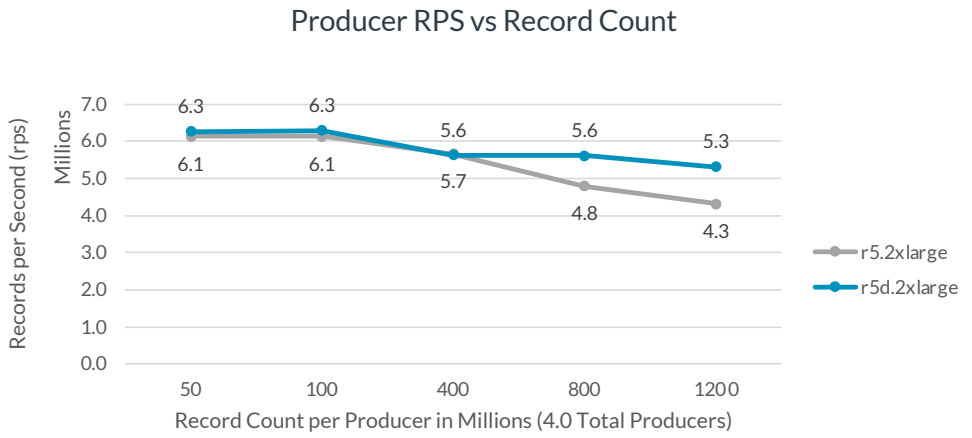

r5 - Consumer RPS/$ Larger Instances
(Millions RPS)/$
80
70
60
50
40
30
20
10
0
72.851631
14.139729
r5.2xlarge
r5.16xlarge
r5 Instance Type

Comparing the r6g.2xlarge to the r6g.12xlarge, there is an RPS increase of about 50% at a 6x higher cost. Comparing the r5.2xlarge to the r5.16xlarge, there is an RPS increase of about 50% at an 8x higher cost. The RPS/$ graphs further show the significant difference in value between the smaller and larger instances. This shows that using larger instances for a Kafka cluster is a poor value. We did not test the r5a.24xlarge (20Gpbs), but we should expect equivalent results.

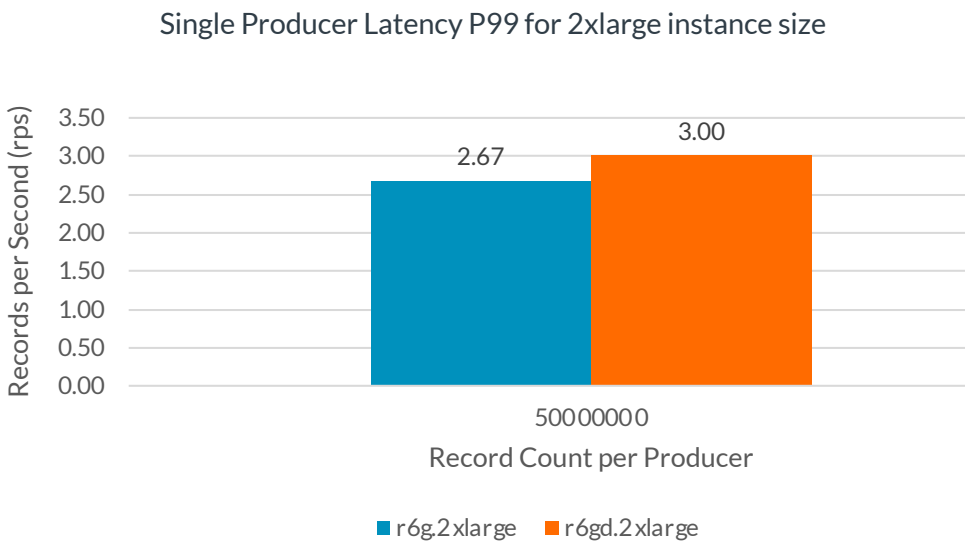
5.3 Direct-Attached Storage Test Results

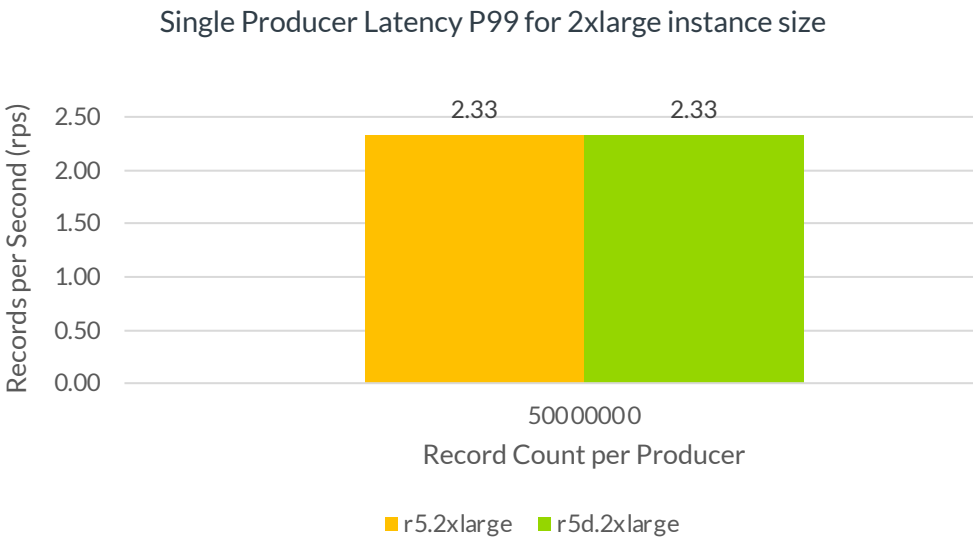
Finally, we looked at the difference between using EBS storage and direct-attached storage. Direct-attached storage instances are indicated by the 'd' post fixed to the instance type name. For example, an r6gd has direct-attached storage, while an r6g does not. We compared the r6g.2xlarge to the r6gd.2xlarge, and the r5.2xlarge to the r5d.2xlarge. Below are the results.





Since the instances with direct-attached storage have higher IO bandwidth than the EBS volumes, the downward trend is less than with the EBS volumes. Although we did not test the r5ad.2xlarge, we would expect to see the same behavior. The P99 latency tests for 50 million records are shown below.





Latency results are similar between instances with EBS volume and direct-attached storage.

Closing Remarks

According to our testing, it is best to use smaller instances like the xlarge. Of all the xlarge instances tested, the r6g had the highest performance and lowest cost when compared against the r5 and r5a. The r6g.xlarge had about a 3.7% throughput advantage over the r5.xlarge, and about a 6.41% throughput advantage over the r5a.xlarge. When we consider the lower cost of the r6g.xlarge, we see about a 30% cost-performance advantage over the r5.xlarge, and about a 19% cost-performance advantage over the r5a.xlarge. That said, this was a synthetic testing environment, so results may vary in other scenarios. For this reason, we want to encourage readers to experiment with running Kafka on AWS Graviton2-based instances using their particular use cases.