Synadia

TECHNICAL PAPER

Comparing NATS.io with Amazon Kinesis

More Functionality, Less Cost

By Jean-Noël Moyne, Field CTO, Synadia

Abstract

NATS.io, an edge-native, AI-ready platform for distributed systems and extending AI to the edge, delivers lower costs than Amazon Kinesis, while offering additional functionality that does not exist in Kinesis.

In a detailed comparison of the functionalities of NATS.io versus Kinesis and their associated costs, Jean-Noël Moyne, Field CTO at Synadia, argues how NATS.io, which includes all the streaming functionalities of Kinesis Client Library (KCL)/Kinesis Producer Library (KPL), enables you to do far more than streaming with less costs. Moyne reaches his cost conclusions by running a series of benchmarks of NATS performance and calculating what the costs associated with that performance would be when using Kinesis. When you scale your usage of streaming, Moyne found, Kinesis becomes more expensive than running a NATS cluster.

In terms of costs, Moyne ran benchmarks on

1. **Compute costs:** that is, the EC2 instance costs for NATS and the consumer costs in Kinesis.
2. **Storage costs (amount of data stored and length of retention):** that is, the EBS costs for NATS and the put costs in Kinesis.
3. **Network costs:** associated with replicating the data between three Availability Zones (AZs).

According to Moyne, Kinesis costs can be orders of magnitude larger than those incurred by NATS based on the difference in 'put' and consumer costs, thus resulting in NATS savings of between 59% and 87% of the equivalent Amazon Kinesis costs. See Figures 12 and 13.

NATS also offers additional functionality non-existent in Kinesis. The paper provides a detailed analysis of the functionalities of Kinesis and NATS, as well as an in-depth description of NATS and its fundamental differences when compared to Kinesis.

NATS is highly differentiated in terms of its handling of sharding, multiple groups of consuming application instances on a stream and data retention.

NATS offers added functionalities, for example: the ability to extend subject-based addressing to streams; the NATS JetStream persistence layer as a NoSQL data store (rather than just a Write Ahead Log); in-memory storage for streams, file-based persistence (with optional compression); and throughput and latency.

Architectural differences, including global deployment, durability and fault tolerance and security are also examined. NATS offers more options for flexible global deployment and replication between the NATS servers that is both automatic and capable of administrative influence. While offering similar security profiles, NATS also provides more cross-tenant data exchange flexibility.

Overall, NATS is feature-rich, comprehensive and flexible, enabling architects, developers and operators to extend connectivity for existing and AI-ready applications at less cost across clouds and geos, on premises, IoT and edge.

“So, you can do with NATS what you can do with Kinesis but also a lot more. Cost is by far the most important difference between the two; Kinesis becomes much more expensive than running a NATS cluster, especially as you scale your usage of streaming.”

Introduction

Amazon Kinesis is limited to streaming while open source NATS offers streaming plus multiple services for more functionality with less cost.

In the Amazon Web Services (AWS) ecosystem, Amazon Kinesis Data Stream (referred to as Kinesis in this document) is meant to do one thing and one thing only: **basic streaming** (that is, record messages into a stream where they can be replayed later).

Open Source NATS not only provides streaming but also many other services, such as request/reply microservices, key-value, object store, clustering and super clustering, mirroring sourcing, MQTT, WebSocket, constraints limits, compare and set, pub-sub and queuing. In contrast, with Kinesis, these same built-in NATS services must be implemented by a number of other AWS services, such as Amazon SNS, Amazon SQS, Amazon DynamoDB, AWS Lambdas, Amazon ElastiCache and Amazon S3.

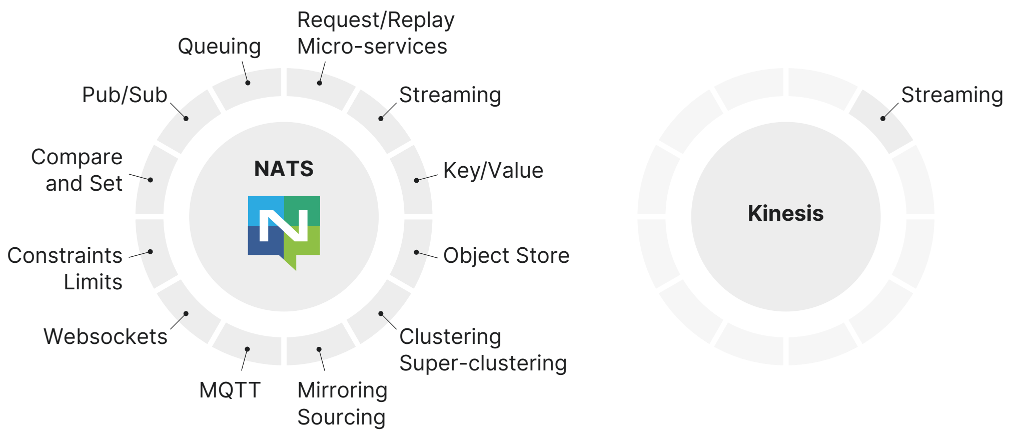


Figure 1: Functional Areas of NATS v. Amazon Kinesis

This document focuses solely on comparing Kinesis' functionality with the streaming functionalities of NATS. Please note that, in this context, Kinesis is not to be confused with Amazon Data Firehose (previously known as Amazon Kinesis Data Firehose), which is a higher-level stream-based (that is, built on top of Kinesis) integration platform with a collection of connectors and message transformation and therefore more comparable to something like Wombat.dev in the NATS world.

01

“Using EFO consumers in Kinesis for low-latency or multi-app scenarios drastically increases costs. NATS offers 59–87% savings in comparison.”

NATS: More functionality with less cost

This document provides a detailed comparison of the functionalities of NATS versus Kinesis.

The short of the story, however, is that NATS includes all the streaming functionalities of Kinesis KCL/KPL: so, you can do with NATS what you can do with Kinesis but also a lot more. Cost is by far the most important difference between the two; Kinesis becomes much more expensive than running a NATS cluster, especially as you scale your usage of streaming.

Your first choice when using Kinesis is whether to go with provisioned or on-demand service. They are billed differently: on-demand is priced purely on the amount of data that you publish and consume and automatically scales the number of shards depending on your traffic and the number of consuming applications, while provisioned mode means you control the number of shards you want to provision and that provisioning limits the maximum amount of data that you can store or consume from the stream per shard.

In both cases, the cost is composed of two parts:

1. **Compute costs:** that is, the EC2 instance costs for NATS and the consumer costs in Kinesis
2. **Storage costs (amount of data stored and length of retention):** that is, the Elastic Block Store (EBS) costs for NATS and the 'put' costs in Kinesis.
3. **Network costs** associated with replicating the data between three AZs.

To illustrate the cost differences, I created a small NATS cluster on AWS using EC2 instances and EBS for storage and I measured the kind of performance you can get out of that cluster (using the **`nats bench command`** from a client instance). All costs are in USD, as of January 2025.

Specifically, I created a 3-node cluster using 3 EC2 c5n.xlarge instances (in US-East1) each one in a different Availability Zone (AZ), with 3 EBS volumes for storage (gp3) so each server writes to an EBS volume in a different AZ. Numbers would be higher (and no network costs would be incurred) if locating all the servers in the same AZ. Kinesis, however, claims to persist messages over multiple AZs (although AWS does not say if it's 2 or 3). I used c5n instance types as they have the best network performance. NATS' performance (like any other distributed broker-based messaging system) is very dependent on the network throughput, not just between the clients and the server but also between the servers themselves (for example, when using replication, the messages are sent over the network multiple times between the servers).

Note that all the benchmark results were basically the same when using large rather than xlarge instances sizes, but, in a production deployment, I would use at least xlarge instances in order to have some extra headroom.

Compute costs for NATS versus Kinesis

EC2 c5n.xlarge on-demand instances cost is 0.216 USD per Hour in US-East1 x 730 hours in a month = 157.68 USD per month per instance; so the compute cost is 473.04 USD per month for the cluster of 3 NATS servers.

When benchmarking a streaming system, you need to make several measurements to get a better understanding of the performance curve:

- The max performance numbers of how fast you can write to a stream when no one is listening and how fast you can read (replay) messages from the stream when no one is publishing new messages. Those numbers are interesting to record separately because the write performance is inevitably going to be 1) the limiting factor of how fast a single consuming application is going to be able to read new messages, and 2) much lower as it is more work for the servers to handle a write (replication and a RAFT vote between the three servers) than to service a write.
- The sustained performance numbers, where you publish messages while at the same time have client applications consuming those messages. Those performance numbers must be measured at different fan-out ratios as well.
- Two numbers that are measured for each type of performance measurement: the number of messages and the throughput, which both are dependent on the size of the messages being published. As a result, I made test measurements at 1KiB, 5KiB, and 10KiB.

All NATS benchmark numbers are for a stream replicated 3 times with file (EBS GP3) using 3 separate AZs for persistence. There also is an example of the **`nats bench command`** used to run a benchmark. Each benchmark run was for at least 1 GiB of payload data and at varying message sizes.

Please note that benchmarking is not an exact science. These numbers are the best recorded and repeatable results seen from a reasonable number of runs of each benchmark. There is always some variation from one run to the next. This variation is especially true for NATS when the number of producing or consuming clients connections is the same or less than the number of servers in the cluster (that is, it depends on how those few client connections finally are distributed over the servers in the cluster, as was the case in some of those measurements).

Another factor in the variation is the use of only one client EC2 instance and therefore located on the same AZ as only one of the servers in the cluster. AWS, however, has limits on the amount of throughput you get per TCP connection between two AZs. In real-life deployments, you would probably have more than one or 2 instances of client applications publishing messages. In addition, for each application consumer, you would have more than one instance of the client application getting messages from the stream at the same time and, therefore, would have less perceived variation in performance over time. Finally, when taking those numbers and comparing them to AWS costs, I always rounded the number of MiB/s down.

NATS max publication rate

Let's look at how fast you can publish with NATS when there are no consumers present at the same time. The numbers in Figure 2 give an indication of real-life 'burst' performance. Note that in streaming use cases, you would usually have applications actively pulling those messages at the same time. Using **`nats bench test --js --replicas 3 --pub 8 --size=1kb --msgs 1000000 --no-progress --purge`** for 1KiB messages such that 1 GiB of data is being published (adjusted for each message size), the results are:

1 KiB	69,592 msgs/sec ~ 67.96 MiB/sec
5 KiB	28,059 msgs/sec ~ 137.01 MiB/sec
10 KiB	17,531 msgs/sec ~ 171.20 MiB/sec
100 KiB	2,741 msgs/sec ~ 267.75 MiB/sec

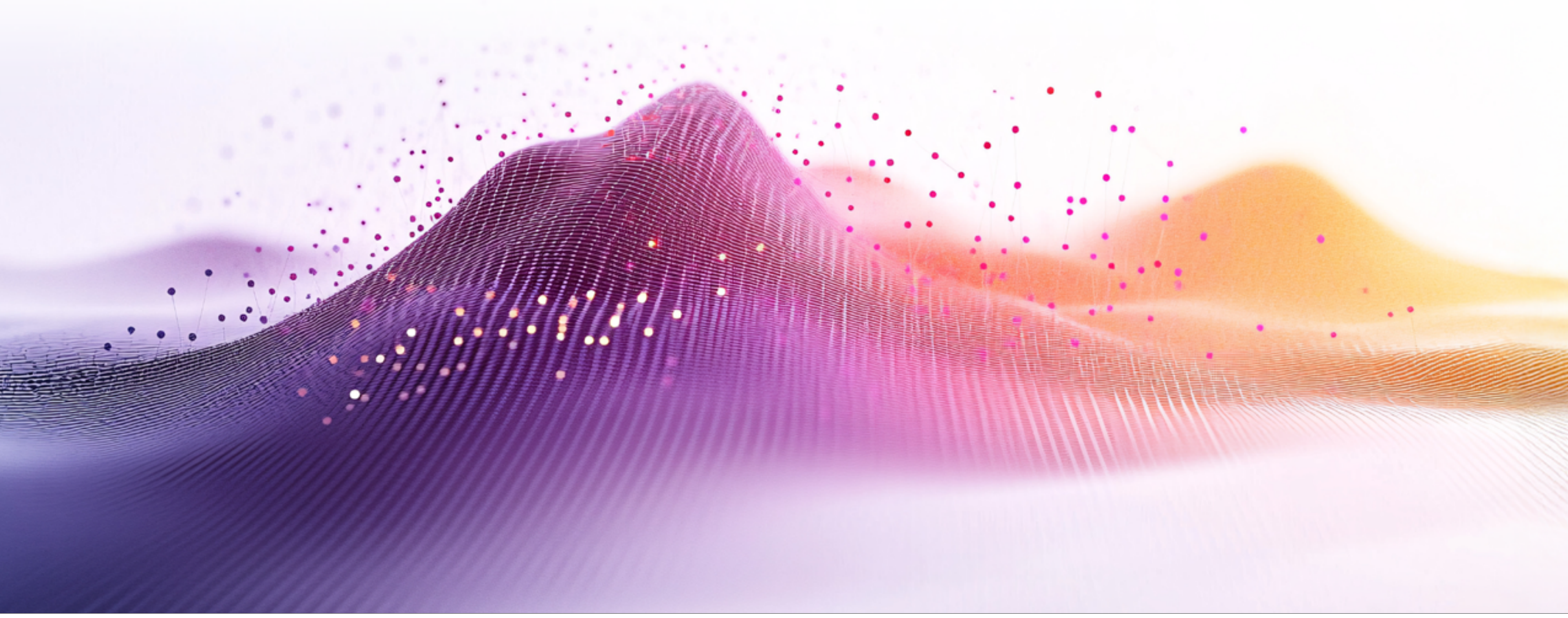
Figure 2: NATS max publication rates at 1, 5, 10 and 100 KiBs

NATS max consumption rates

Now let's consider max consumption rates with NATS. How fast can you replay messages from the stream using an ordered consumer (with no filter) while there are no new publications to the stream (the messages having already been published ahead), for a fan-out of 1 to 4 consuming applications (each consumer gets its copy of all the messages in the stream)? The numbers in Figure 3 give an indication of the 'burst' performance since, in streaming use cases, you would usually have new publications to the stream at the same time. An example command used is **`nats bench test --js --replicas 3 --sub 1 --size=1kb --msgs 1000000 --no-progress`** in this case for one group of consuming application instances with just one instance pulling messages and 1KiB messages. The numbers in Figure 3 are the sum of the throughputs for all of the consumers.

MAX CONSUME - ONLY PERFORMANCE (AGGREGATED FOR ALL CONSUMERS)	1 CONSUMER	2 CONSUMERS	3 CONSUMERS	4 CONSUMERS
1 KiB	408,055 msgs/sec ~ 398.49 MiB/sec	723,248 msgs/sec ~ 706.30 MiB/sec	783,773 msgs/sec ~ 765.40 MiB/sec	699,929 msgs/sec ~ 683.53 MiB/sec
5 KiB	113,474 msgs/sec ~ 554.07 MiB/sec	114,158 msgs/sec ~ 557.42 MiB/sec	170,521 msgs/sec ~ 832.62 MiB/sec	151,914 msgs/sec ~ 741.77 MiB/sec
10 KiB	57,200 msgs/sec ~ 558.60 MiB/sec	57,293 msgs/sec ~ 559.51 MiB/sec	86,110 msgs/sec ~ 840.92 MiB/sec	76,864 msgs/sec ~ 750.63 MiB/sec
100 KiB	5,791 msgs/sec ~ 565.59 MiB/sec	11,008 msgs/sec ~ 1.05 GiB/sec	8,713 msgs/sec ~ 850.98 MiB/sec	11,577 msgs/sec ~ 1.10 GiB/sec

Figure 3: NATS max consume-only performance (aggregated for all consumers)



NATS sustained consumption rates when publishing and consuming at the same time

What throughput do you reach with NATS when publishing while, at the same time, having consumers consuming the messages for a fan out of 1 to 4 consuming applications? An example command used is **`nats bench test --js --replicas 3 --pub 8 --sub 1 --size=1kb --msgs 1000000 --no-progress --purge`** for, in this case, a fan-out of 1 application with just one instance pulling exactly 1 GiB of payload data (adjusted for message size).

MAX SUSTAINED CONSUME PERFORMANCE WHILE MESSAGES ARE BEING PUBLISHED (PER CONSUMER)	1 CONSUMER	2 CONSUMERS	3 CONSUMERS	4 CONSUMERS
1 KiB	51,408 msgs/sec ~ 50.20 MiB/sec	49,428 msgs/sec ~ 48.27 MiB/sec	50,096 msgs/sec ~ 48.92 MiB/sec	49,909 msgs/sec ~ 48.74 MiB/sec
5 KiB	24,739 msgs/sec ~ 120.80 MiB/sec	23,123 msgs/sec ~ 112.91 MiB/sec	21,899 msgs/sec ~ 106.93 MiB/sec	21,663 msgs/sec ~ 105.78 MiB/sec
10 KiB	14,000.5 msgs/sec ~ 136.72 MiB/sec	14,595 msgs/sec ~ 142.53 MiB/sec	13,333 msgs/sec ~ 130.21 MiB/sec	13,733 msgs/sec ~ 134.12 MiB/sec
100 KiB	1,746 msgs/sec ~ 170.55 MiB/sec	1,885 msgs/sec ~ 184.08 MiB/sec	1,790 msgs/sec ~ 174.84 MiB/sec	1,754 msgs/sec ~ 171.35 MiB/sec

Figure 4: NATS max sustained consume performance while messages are being published (per consumer)

Note that the numbers are 'per consumer': so for example, with 100 KiB messages and 4 consumers, the servers are transmitting 685.4 MiB/s of data in total. Those numbers show that, as expected, it's the replicating and writing of the messages to storage that limits the throughput of a stream; at those message sizes, the NATS servers, in this setup, can handle up to at least 4 consumers without any large decrease in throughput. At no time during those benchmarks did the CPU utilization on the NATS servers ever max out; the CPU utilization was typically between 50% and 75%. The numbers for 100 KiB are there to show that most of the throughput is already achieved with messages as small as 10 KiB, so I will round down those numbers and select the following message sizes and throughput values in the cost calculations.

MESSAGE PAYLOAD SIZE	THROUGHPUT VALUES USED FOR KINESIS COST CALCULATION WITH UP TO 4 CONSUMERS
1 KiB	45 MiB/s
5 KiB	100 MiB/s
10 KiB	125 MiB/s

Figure 5: NATS throughput values used for Kinesis cost calculation (up to 4 consumers)

NATS cost of storing the messages

Regardless of the size of the server instances, the storage costs with NATS remain the same when using EBS. NATS can also use local SSDs, or even memory storage as well. Using GP3 EBS file storage with default throughput (as was done for these benchmarks), it is easy to calculate an estimated cost by calculating the amount of storage required to store one day's worth of data at the specific throughput; for GP3, it's $0.08 per GiB per month.

MESSAGE PAYLOAD SIZE	SIZE PER REPLICA FOR 24H AT SUSTAINED RATE	NATS STORAGE COST PER MONTH FOR 3 REPLICAS
1 KiB	45 MiB/s*3600*24=3,888 GiB	$311.40*3=$934.20
5 KiB	100 MiB/s*3600*24=8,640 GiB	$691.20*3=$2,073.60
10 KiB	125 MiB/s*3600*24=10,800 GiB	$864.00*3=$2,592

Figure 6: NATS cost of writing and storing messages based on message size and 3 replicas

If you want to store for longer than 24 hours, with NATS, it's only a matter of more data in EBS at the exact same price of $0.08/GiB/month, while, with Kinesis, you're charged first extended data retention rates for the first seven days and then charged a lower GB-month rate for data stored beyond that. Also with NATS, you have the choice to limit the retention of data to a max size of the stream (for example, how many GB) or a max number of messages (and those limits can be combined), rather than just by time.

Network costs for replication with NATS

While NATS can provide high availability as soon as you use 3 nodes in a cluster, those nodes do not have to necessarily spread over 3 different AZs. Since Amazon Kinesis claims to have 99.9% availability, therefore the assumption is that the data is replicated over multiple AZs. Accordingly, the NATS cluster was deployed over 3 AZs. Although there are no network costs associated with sending data between EC2 instances in the same availability zone (and higher per TCP connection through-put limits as between AZs, the limit is 5 Gb/s), AWS does charge for:

- Transferring data over the network between AZs
- Sending on one side and receiving on the other side, for example, sending 1 GB of data from an EC2 instance in one AZ to another instance on another AZ will cost $0.02/GB.

As a message is stored into a stream, one of the nodes in the cluster will send that data again to two other nodes in the other AZs; therefore, to those EBS costs, one must add $0.04/GB of cross-AZ networking costs per GB of payload stored in the stream. Actual network costs could be higher depending on how you let the client applications connect to the cluster as a client may be sending or receiving data from a server located in another AZ; that scenario would add to the cross-AZ charges, but that is difficult to estimate.

MESSAGE SIZE	SIZE PER REPLICA FOR 24H AT SUSTAINED RATE	NATS STORAGE PLUS NETWORKING MONTHLY COST
1 KiB	45 MiB/s*3600*24=3,888 GiB	934.20+3888*.04=$1,089.72
5 KiB	100 MiB/s*3600*24=8,640 GiB	2073.60+8640*.04=$2,419.2
10 KiB	125 MiB/s*3600*24=10,800 GiB	2592+10800*.04=$3,024

Figure 7: NATS network costs for NATS storage and networking monthly cost

Kinesis cost for storing the messages

AWS charges you for writing messages to the stream according to the number of 'Put Payload Units per month cost' you need depending on your message size and rate of publication. Here is the table with Kinesis for the throughput values we selected at each message size as calculated by the AWS pricing calculator.

MESSAGE SIZE @ RATE PER SECOND	AMAZON KINESIS PUT PAYLOAD UNITS PER MONTH
1 KiB @ 45000 (45 MiB/s)	$1,655.64 ($36.79 per 1 MiB/s)
5 KiB @ 20000 (100 MiB/s)	$735.84 ($7.36 per 1 MiB/s)
10 KiB @ 12500 (125 MiB/s)	$459.90 ($3.67 per 1 MiB/s)

Figure 8: Amazon Kinesis monthly storage costs

Note that 'put' costs depend on the overall throughput and your message size, with the cheapest price being for a message size of exactly 25 KiB (or a multiple of 25KiB). Thus, in our scenarios, there is this significant decline in 'put' costs as the message size increases towards 25 KiB. To demonstrate the variation in Kinesis 'put' costs depending on message sizes rather than throughput, consider the wide variation of costs in Figure 9:

MESSAGE SIZE @ RATE PER SECOND	AMAZON KINESIS PUT PAYLOAD UNITS PER MONTH
500,000 messages/s at 1 KiB payload	$18,396.00
50,000 messages/s at 10 KiB payload	$1,839.60
20,000 messages/s at 25 KiB payload	$735.84
20,000 messages/s at 26 KiB payload	$1,471.68
10,000 messages/s at 50 KiB payload	$735.84

Figure 9: Amazon Kinesis' monthly 'put' cost for similar throughput according to message size

Note that these numbers are just the 'put' cost and not inclusive of the cost associated with the number of shards. This number of shards is dictated by the publication rate or the throughput (the max throughput per shard is 1 MiB/s or 1000 messages/s for writes, whichever limit you hit first). The number of shards you need to sustain your ingress is a minimum as you may need even more shards depending on your egress rate (that is, your fan-out).

Kinesis cost of delivering the messages

Pricing for getting the messages from Kinesis depends on three factors:
1. The rate at which you want to be able to consume the messages
2. The size of the messages (which dictates the number of shards you will need to buy for that throughput)
3. The number of groups of consuming application client instances (that is, the number of 'consumer applications' in the AWS pricing calculator).

The Kinesis egress costs start with a baseline and peak costs: in this case, I entered the same value for the number record/s for both base and peak (with 10% 'buffer for growth and to absorb un-expected peaks').

Kinesis non-enhanced versus enhanced fan-out consumers

The cost of delivering messages in Kinesis will depend greatly on the number of consumers you will have for the stream and whether you need Enhanced Fan-Out (EFO) consumers or not.

In practice with Kinesis, if you have more than one consuming application on a stream in real time, you need to use an EFO for each application. For example, as the instances of the KCL client application share the same lease table, only one application (group) can process each shard's data unless you use EFOs. Even if you don't use KCL, AWS asks you to consider using Enhanced Fan-Out (EFO) consumers if you need 70ms latency and have more than two consumers.

Regular non-enhanced consumers all share the same total read throughput of 2 MiB/s per shard. So, in our example of 45 MiB/s throughput with 50 shards, you can only have up to 2 regular consumers on the stream. Adding more regular consumers without also increasing the number of shards would not meet the performance requirement of the traffic flow (this is reflected by "N/A" in the tables below).

On the other hand, EFO consumers scale as consumers register to use enhanced fan-out. Each consumer registered to use enhanced fan-out receives its own read throughput per shard, up to 2 MiB/sec, independently of other consumers. They also provide lower latency of an average of 70 ms whether you have one consumer or five consumers. Regular consumers have an average of around 200 ms if you have one consumer reading from the stream; this average goes up to around 1000 ms if you have five consumers. EFO consumers costs are composed of an 'Enhanced Fan-Out consumer-shard hours cost' and an 'Enhanced Fan-Out data retrieval cost.'

In contrast with NATS, there are no shards and no throughput limits per consumer [the performance depends mostly on the network (and disk) throughput available to the server processes]. Typical latencies even with multiple consumers on the stream are in the very low number of milliseconds or less. So, I would contend that the consumers that you get out of the box from NATS are the equivalent of EFO consumers in Kinesis. See **Appendix A** for the tables showing the calculations for the retrieval costs for each message size with fan-outs from 1 to 4 consuming applications.

Comparing NATS with provisioned Kinesis

NATS' total costs consist of the compute, storage and network costs

MESSAGE SIZE @ RATE PER SECOND	NATS' TOTAL COSTS FOR UP TO 4 CONSUMERS
1 KiB @ 45000 (45 MiB/s)	$1,562.76
5 KiB @ 20000 (100 MiB/s)	$3,192.24
10 KiB @ 12500 (125 MiB/s)	$3,797.04

Figure 10: NATS total storage, compute and network monthly costs for up to 4 consumers

Kinesis total costs consist of the 'put' charges above plus the retrieval costs according to the number of consumers and number of EFOs (see Appendix A). Provisioned Kinesis comes with up to 2 non-EFO customers for free; you can technically have up to 2 regular (but high latency) consumers in addition to the EFO consumers. See the column with 0 EFO in Figure 11 but, in practice, you probably will need one EFO consumer per application when going beyond one application.

	0 EFO CONSUMER	1 EFO CONSUMER	2 EFO CONSUMERS	3 EFO CONSUMERS	4 EFO CONSUMERS
1 KIB @ 45000 (45 MIB/S)	$2,203.14	$4,216.8	$6,230.45	$8,244.11	$10,257.76
5 KIB @ 20000 (100 MIB/S)	$1,918.44	$6,359.17	$10,799.90	$15,240.63	$19,681.36
10 KIB @ 12500 (125 MIB/S)	$1,938.15	$7,489.07	$13,037.99	$18,590.90	$24,141.82

Figure 11: Kinesis total storage costs based on put charges plus retrieval costs according to the number of consumers and number of EFOs

Price Difference of Provisioned Kinesis versus NATS (% change from Kinesis to NATS)

As you can see in Figure 12, in the special case of 0 EFO consumers, Kinesis is cheaper than NATS for the larger message sizes. The Kinesis costs in that 0 EFO column are just the cost of the shards plus the number of put payload units (which, as shown in Figure 9, becomes more expensive the further away from a multiple of 25 KiB your message size is). In this case, you also will still need to increase the number of shards as you add non-EFO consumers.

	0 EFO + 1 NON-EFO CONSUMER	1 EFO CONSUMER	2 EFO CONSUMERS	3 EFO CONSUMERS	4 EFO CONSUMERS
1 KIB @ 45000 (45 MIB/S)	$640.38 (-29%)	$2,809.56 (-66%)	$4,823.21 (-77%)	$6,836.87 (-83%)	$8,850.52 (-86%)
5 KIB @ 20000 (100 MIB/S)	-$1,273.8 (+66%)	$3,812.53 (-60%)	$4,823.21 (-76%)	$12,693.99 (-83%)	$17,134.72 (-87%)
10 KIB @ 12500 (125 MIB/S)	-$1,858.89 (+95%)	$4,424.03 (-59%)	$8,253.26 (-76%)	$15,525.86 (-83%)	$21,076.78 (-87%)

Figure 12: Price difference of a provisioned Kinesis compared to NATS, using % change from Kinesis to NATS

But as soon as you have more than one KCL consuming application on the stream or need somewhat reasonable latency, you must use EFO consumers instead. There the Kinesis costs are orders of magnitude bigger than those incurred by NATS drawing the difference in ‘put’ costs and **resulting in savings of between 59 and 87% of the equivalent AWS costs.**

Compared to On-Demand Kinesis

One of the first choices you must make with Kinesis is whether to go with Provisioned or On-demand service (that is, assuming you can use KCL/KPL. If you want to re-implement your version of them, your only option is static provisioning).

One advantage of On-Demand kinesis is that you do not have to worry about the number of shards. But you will experience a not-so-speedy reaction time when scaling up, which is 'within minutes.' At the same time, you have to pay for this convenience as the costs can be much higher. If you were going to have sustained traffic, then you could calculate the number of shards you need and you would not use the Provisioned service

So while not perfectly comparable to the cost of an on-demand service, the provisioned service is still an interesting data point that brings some perspective to those cost numbers.

One difference with provisioned is that you pay for each of the first 2 (non EFO) consumers (see Appendix B for detailed breakdown). So, in Figure 13 with One Demand Kinesis, "0 EFO consumer" means there was just one regular consumer, and the rest of the columns are with only EFO consumers.

	0 EFO CONSUMER	1 EFO CONSUMER	2 EFO CONSUMERS	3 EFO CONSUMERS	4 EFO CONSUMERS
1 KIB @ 45000 (45 MIB/S)	$12,155.74 (-88%)	$13,283.56 (-89%)	$18,922.63 (-93%)	$24,561.71 (-94%)	$30,200.78 (-95%)
5 KIB @ 20000 (100 MIB/S)	$27,557.63 (-91%)	$30,063.89 (-92%)	$42,595.17 (-94%)	$55,126.45 (-95%)	$67,657.73 (-96%)
10 KIB @ 12500 (125 MIB/S)	$22,026.72 (-86%)	$37,690.82 (-92%)	$53,354.92 (-94%)	$53,354.92 (-94%)	$84,683.12 (-96%)

Figure 13: Price difference of On-Demand Kinesis versus NATS (% change from Kinesis to NATS)

Finally, the other difference with a provisioned NATS cluster is that you are not being charged by the number of streams you are using (or the number of subjects stored in a stream), just like you are not being charged by the number of consumers on a stream. For example in this setup, you could easily have a few more streams (although not necessarily with the same amount of traffic) for the same price (compared to Kinesis where you have to pay again for every stream you create).

Additional DynamoDB costs

KCL also makes use of DynamoDB under the covers, which will incur some additional costs on top of your Kinesis costs.

02

“Essentially, NATS covers all functionalities offered by KCL and KPL (except batching, which you have to do yourself): you can create streams, record published messages into those streams, have the message in those streams delivered or replayed to client applications, and a lot more.”

Functionalities

Kinesis APIs and Libraries

The very first thing to realize about Kinesis APIs is that they exist at 3 different levels:
1. At the lowest level, the core base API is HTTP
2. At the next level, the AWS SDK is a thin layer exposing the HTTP API in various languages
3. Finally, there is the Kinesis Producer Library (KPL) and the Kinesis Client Library (KCL).

Kinesis HTTP API

The Kinesis HTTP API is the lowest level of functionality, but it is available on any language and platform that can make HTTP requests. Its functionalities can be broadly categorized as basic low-level streaming. The Kinesis HTTP API allows you to:
- Create, delete, list and tag streams, adjust the retention period
- Get and set resources policies, start and stop stream encryption (see security section)
- Add, remove, split and merge shards for a stream
- Put records
- Retrieve a shard iterator (pointer) to use to read records from a specific shard
- Register/de-register stream consumers that can then subscribe to shards.

And that’s it! Using the HTTP API, the functionality of Kinesis is very limited with many things that you will need to implement yourself in the client application code.

AWS SDK

The AWS SDK exposes those HTTP API calls in a library, allowing you to access the functionality exposed by the Kinesis HTTP API, directly in the language of your choice without having to worry about the details of making HTTP API requests (for example, all the headers including the AWS authorization signature). In general, the AWS SDK is recommended for most use cases due to its simplicity and abstraction.

The AWS SDK is only a thin layer on top of the HTTP API. There are still functionalities that you will need to handle in your code:
- Automatic shard assignment to consumer instances: inter-worker coordination logic to share shard processing. monitoring for changes in shard structure, shards reassigning
- Failover and load balancing of the consumers for the shards
- The number of shards changing with no need for manual intervention
- Checkpointing: store the offsets yourself somewhere and handle state recovery
- Doing retries for transient errors (for example, handling ‘limit exceeded’ errors each time you hit the limit of the number of operations per second).
- Implement logic to ensure ordered processing yourself.

Kinesis Client Library and Kinesis Producer Library

Because of the limited functional scope of the HTTP API and the AWS SDK, and to move to a level of functionality closer to what you get from Apache Kafka for example, there are two higher level Kinesis client libraries that implement the listed functionalities that are missing from the AWS SDK: the **Kinesis Client Library (KCL)** that simplifies distributed consumption from a stream, and the **Kinesis Producer Library (KPL)** that simplifies data production to a stream by handling batching, aggregation, retries and error handling.

While the AWS SDK is available for many languages (Java, Python, JavaScript, C#, Go, Ruby, PHP, C++ and .NET plus a couple of others that are community supported), KCL/KPL are written in Java. Using KCL/KPL from one of the other languages they support (Python, Node.js, C# and Ruby) means running a KCL/KPL process on a JVM that the client-native libraries in other languages communicate with via a Multi-Language Interface (JSON over standard input/output).

In comparison, NATS via Synadia maintainers has officially supported native (no need to talk to a background JVM over stdio) client libraries for all those languages plus Rust, Swift, Elixir and Java Vertx (PHP is community contributed).

The set of functionalities implemented by KCL arguably brings Kinesis from the very basics of the HTTP API to a much more usable level comparable to what is offered by Apache Kafka. So, how do KCL streaming functionalities and NATS's streaming functionalities compare?

NATS Functionality

Common NATS functionalities

Essentially, NATS covers all functionalities offered by KCL and KPL (except batching, which you have to do yourself): you can create streams, record published messages into those streams, have the message in those streams delivered or replayed to client applications, and a lot more. The messages in a single stream can be distributed in some way between multiple instances of the client application consuming those messages, making it very easy to scale those consuming applications.

In terms of message consumption both Kinesis and NATS support:

- Replay of the messages from: the start of the stream, a specific point in time, or just the new messages.
- Keeping the messages in the stream available for replay for a certain period of time.
- Stream group of consumers-type functionality: the automatic distribution of the stream's messages between client application instances, with the ability to add or remove client application instances at any time from the stream and the ability to stop and restart applications without having to worry about checkpointing message sequence numbers.

NATS fundamental differences

Sharding

Sharding is a major concern in Kinesis provisioned mode because of the direct cost implications.

While the KCL abstracts the current number of shards a stream has from your client application code, it is still something that, in provisioned Kinesis, you need to worry about and manage administratively for every single stream.

The two factors that affect the number of partitions you need to provision for a stream in Kinesis are:
1. The parallelization factor that your client applications will need to sustain the amount of traffic to consume (that is, the max number of instances you need to run at any time): you can only have one client application instance consuming from a stream shard at a time per application. If you need to scale up, you can't just increase the number of client applications, you also may have to increase the number of shards for that stream.
2. The ingress and egress data rates you will need: there are hard limits to both the amount of data and number of API requests that you can make to a stream shard, for example, 1MiB or 1000 records per second for writes, 2MiB per second for reads and 5 GetRecords API per second.

In contrast, sharding is simply not required in NATS to distribute a stream's messages between a group of client application instances; the client application instances can scale elastically from 0 to any number without having to worry about the number of partitions. Should you need consistent hashing-based partitioning of a stream, it is just an admin operation to automatically insert a partition number to the messages recorded in a stream using the subject mapping functionality.

More than one group of consuming client applications instances on a stream

With NATS, you can have as many groups of consuming client application instances ('consumers') as you want on a given stream. While with Kinesis, the first group is free, but if you care about latency or need more than one consumer you will need to buy EFO consumers, thus incurring much additional costs.

Data retention

By default, the retention period of Kinesis is 24h exactly (it cannot be less). You are then charged extended data retention rates for the first seven days and then charged a lower GB-month rate for data stored beyond seven days up to your specified retention period (with a maximum of 8760 hours (365 days)). There are no other options available besides time to control the amount of data being retained for replay.

NATS, on the other hand, also can limit the retention of messages in the stream by size (number of messages or data size of all the messages in the stream), and by number of messages per subject (for example, you can limit the retention to 1 message per subject, which allows you to use a stream as a 'last value cache'). And when using time as the limit, you can set the limit to be as small (down to 1 second) or as large (up to forever) as you want.

NATS additional functionalities

Besides the fact that it is partition-less, NATS has 2 major architectural differences with Kinesis (and other streaming systems, for that matter) from which a number of additional features derive.

NATS extends subject-based addressing to streams

NATS is unique in extending subject-based addressing to streams: the messages stored in its streams have a subject name associated with each message. Streams can store messages on any number of subjects and can directly capture messages over whole hierarchies of subject names. For example, you can create a stream that captures messages using the filter "orders.>" and then publish your customer order messages with subjects of the form "orders.<customer id>.<order id>." You do not need to know ahead of time the list of customer ids and order ids for each customer.

On the consumption side, NATS stream 'consumers' (that is, groups of consuming application instances) can have any number of subject filters, and the filtering of the messages according to those filters is performed by the NATS servers (only the matching messages are transmitted to the consuming applications). And you can also create consumers that only deliver the last messages for each of the subjects in the stream. For example, to get all the messages for all the orders from customer one, you would create a consumer with a subject filter of "orders.1.*"

In comparison, Kinesis has no concept of subject-based addressing: each stream has a single name (like a topic) and while the applications publishing messages to the stream can specify a key for each message, this key is only used to compute the shard the message gets assigned to. You must create streams before you can publish to them and the streams have to be created before you can subscribe to them. There is no concept of filtering the messages in a stream at consumption time (not even by the message's key).

To do in Kinesis the equivalent of what you can very easily do with NATS using subject-based addressing, you would have to implement in your application code the following functionalities (none of them being especially simple or easy to implement properly):

- Every time you publish a message, check if a stream already exists for that topic and create it if it doesn't (which has a cost impact since you are billed per stream and per shard on the stream if in provisioned mode)) before sending the message to Kinesis.
- Build the current list of existing streams and continuously monitor for new streams being created (or existing ones being deleted), to create groups of consumers on all the streams (topics) you may be interested in.
- Because you are charged per stream for each hour the stream is defined, you may also want to implement something that collects streams as garbage when they are un-used to avoid your monthly bill continuously growing over time.

The NATS JetStream persistence layer is a NoSQL data store rather than just a Write Ahead Log

Kinesis only offers the functionalities of a Write Ahead Log: you can add messages to the head of the stream (without constraints) and messages are dropped from the tail of the stream as time advances, and that's it. In contrast, NATS JetStream has the functionalities not just of a WAL but of a proper NoSQL data store. Specifically, with NATS you can:

- Delete any message in the stream (and not just trim the tail end of the stream with the advance of time) either explicitly by sequence number or by purging messages according to a subject filter.
- You can have stream limits defined besides the retention time (max age): number of messages, size, number of messages per subject (for example, you can configure a stream to keep only 1 message for each subject).
- You have the choice of how to react when a limit is breached by a new message being published: either deleting old messages to be able to accept new messages or refusing the new message (like a 'constraint' in a SQL database). Kinesis (or any other streaming system besides NATS for that matter) has no concept of limits or of being able to reject published messages.
- Set a specific TTL for each message that overrides the stream overall max age setting.

NATS additional streaming features

NATS supports in-memory storage for streams, as well as file-based persistence (with optional compression).

When consuming (that is, receiving and successfully processing) messages from a stream, they can be individually acknowledged by the client application, with automated re-delivery if not positively acknowledged within a period of time, plus the ability to 'terminate' or 'negatively acknowledge' (with the option to specify a back-off time for the re-delivery attempt) each message.

Streams have message de-duplication features, which you need to have along with individual message consumption acknowledgement to achieve ‘exactly-once’ rather than ‘at-least-once’ message delivery and consumption.

This, in turn, is part of the ability in NATS to use a stream as a distributed ‘work queue,’ where messages are distributed and then automatically deleted from the stream as they are successfully ‘consumed’ (individually acknowledged by the consuming application). Kinesis is strictly a streaming service and doesn’t do queueing, as that’s the job of the SQS service in the AWS eco-system.

You can also do ‘compare and set’ publications to streams (that is, optimistic concurrency access control of ‘writes’ to the stream) where the publication will succeed only if the stream’s current last sequence number matches the one specified in the publication, or only if the sequence number of the last message in the stream that matches a specified subject filter matches the one specified in the publication.

You can do ‘roll-up’ publications to a stream, allowing you to publish summary events or aggregate state messages that automatically (and atomically if also using compare and set at the same time) deletes all the prior messages (or all the prior messages with the same subject) in the stream.

You can create streams that ‘mirror’ another stream, or that ‘source’ other streams.

You can define subject transformation mappings using functions to slice, split, drop or insert subject tokens (for example, insert a partition number that is calculated using a consistent hash of one of more of the tokens in the subject) that are automatically applied to messages as they are ingested (and sourced or mirrored) into the stream.

Performance and latency

Throughput: Kinesis versus NATS

Theoretically, there is no upper limit to the number of messages per second that you can ‘put’ or get from Kinesis other than what you can afford to spend as it is a managed service.

In Kinesis provisioned mode, the maximum throughput is strictly derived from the number of shards on the stream: each shard is limited to 1MiB per second or 1,000 records per seconds write throughput and up to 2 MiB per second or 2,000 records per second read throughput. To get more throughput, you just buy more shards, up to the default AWS shard quota at 200, except in 3 regions (US East (N. Virginia), US West (Oregon), and Europe (Ireland) where it is 500. To go beyond those numbers, you need to request a quota increase for your AWS account.

In Kinesis on-demand mode, new streams are initially created with a capacity of 4 MiB per second of write and 8 MiB per second of read throughput and can automatically scale up to 200 MiB per second of write and 400 MiB per second of read throughput, except in the same 3 regions – (US East (N. Virginia), US West (Oregon), and Europe (Ireland – where it can scale up to 10 GB per second write and 20 GB per second read (if you submit a support ticket).

Regardless of provisioned or on-demand mode you can only create up to 20 registered 'consumers' (groups of consuming application instances) per stream (the enhanced fan-out limit).

There are also no hard-coded limits to the number of messages you can 'put' and get from NATS. Practically, there is a limit of how many messages per second you can store into a single stream (it can go into the hundreds of thousands per second), and another of how many messages per second you can get or consume from a single stream (it can up to the millions per second range), which depends on many factors. Some of those factors are configurable (for example, the number of replicas for the stream) but ultimately the performance depends on the infrastructure the NATS servers are deployed on (CPU and i/o speed) and the network bandwidth available (or the max allowed for a VM by the operator in clouds). You can also automatically spread your traffic over more than one stream, and those streams are distributed over the NATS servers in your cluster, which you can scale horizontally (so very similar to how you can scale Kinesis streams in provisioned mode by using more shards).

Latency: Kinesis versus NATS

Kinesis does not technically 'push' messages in real-time to client applications, only to AWS Lambdas. To get close, the clients must continuously pull for new messages and, while the KCL library (its JVM process) does this continuous polling for you, the Kinesis hard limit of a max of 5 API calls per second per shard still applies. Kinesis is designed for throughput rather than low latency, meaning that Kinesis message delivery latency is typically measured in seconds and can increase considerably in high traffic scenarios. If you want any kind of lower latencies than 'seconds,' you can get down to the 70ms to 500ms range if you buy EFO consumers, but that has a significant impact on cost.

In contrast, NATS latencies are a completely different order of magnitude as it is a fully-fledged subject-based-addressed, low latency messaging system at its core and is designed to deliver messages with latencies measured in micro-seconds with real-life latencies over LANs typically

03

“NATS servers can be deployed as ‘leaf nodes’ meaning that you can ‘extend’ the NATS streaming service all the way to the edge by deploying NATS leaf node servers that connect back to your cluster or super-cluster in the cloud or in your data centers.”

Architectural and deployment differences

Global deployment: Kinesis versus NATS

Kinesis is designed to operate within a single AWS region by default. To replicate data from a Kinesis instance in one region to another in another region additional mechanisms or services are required, since you need to write an application that consumes messages in one region and re-publishes them in other regions (for example, using AWS Lambdas). There is no way to 'extend' Kinesis outside of an AWS region; you cannot deploy Kinesis servers in another cloud provider, into your premises or to the edge.

In contrast, NATS has many features related to deploying the service globally: you can deploy multiple clusters (that is, one per region) and connect those clusters together into a 'super-cluster.' Client applications can transparently use any service or stream over a 'super-cluster' regardless of which cluster the client is connected to. Mirroring of streams from one cluster to another is built-in to the NATS servers, so there is no need to use (or write) anything.

Furthermore, NATS servers can be deployed as leaf nodes meaning that you can 'extend' the NATS streaming service all the way to the edge by deploying NATS leaf node servers that connect back to your cluster or super-cluster in the cloud or in your data centers. And because mirroring and sourcing happens in an asynchronous but guaranteed no-loss store and forward manner, those leaf nodes do not even need to always be able to connect to the hub cluster(s).

Durability and fault tolerance: Kinesis versus NATS

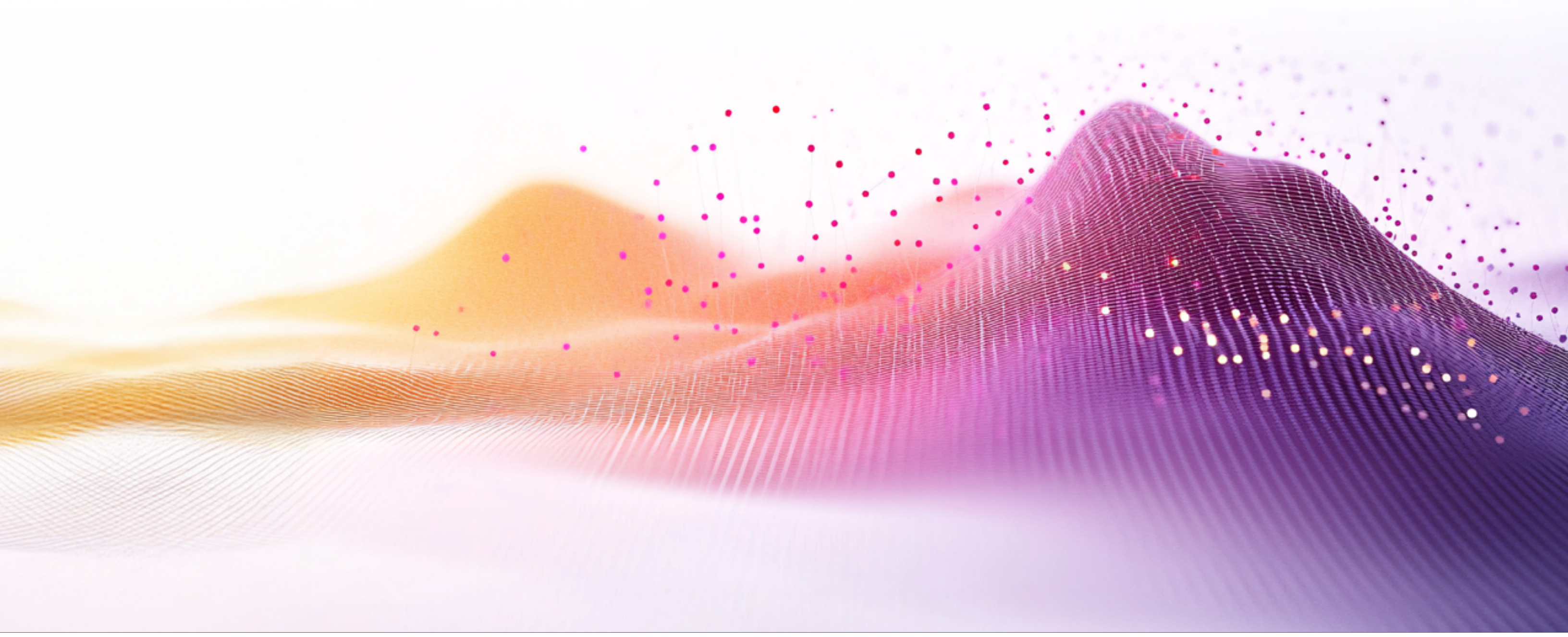
Both NATS and Kinesis provide fault-tolerant durability of the storage of messages through clustering and data replication. Kinesis's replication is entirely handled by AWS; it is a black box with no knobs that is built around replicating the data between the availability zones of a region.

With NATS, the replication happens between the NATS servers assigned to service the stream and is also handled automatically, but you have some things that you can administratively influence.

You can choose file or in-memory persistence for the stream: when selecting the file, the data is persisted by each server assigned to service the stream wherever each server is configured to write its files to (which can be a local drive or an EBS volume). You can select the replication degree: 1 (no replication), 3 or 5 (and adjust it at any time later). You can also influence the selection of the nodes that will replicate a stream using 'server tags' to affect the placement of streams, for example, to ensure that each server replicating the stream is located on a different availability zone.

Security:
Kinesis versus NATS

Kinesis as an AWS service is inherently multi-tenant and so is NATS. They both support encryption in transit and at rest. Authentication and authorization are purely IAM based for Kinesis. They both have a granularity of access control down to a stream and all the messages in it and the kind of operations allowed. NATS supports multiple security models (centralized or delegated) and type of authentication (passwords, JWTs, Certificates) and can integrate with any IdP through the implementation of an authorization callout service. They differ a little bit when it comes to cross-tenant data exchange: both allow direct cross account access (read or write) of a stream from one account to the other, but NATS also gives you the ability to do subject-filtered cross account mirroring and sourcing instead.



Appendices

Appendix A

The monthly cost for provisioned Amazon Kinesis consisting of shards, plus enhanced fan-out and data retrieval costs (source https://calculator.aws/#/createCalculator/KinesisDataStreams).

	0 EFO CONSUMER	1 EFO CONSUMER	2 EFO CONSUMERS	3 EFO CONSUMERS	4 EFO CONSUMERS
1 CONSUMER	$640.38 (-29%)	547.50+ 547.50+ 1466.16= $2,561.16	N/A	N/A	N/A
2 CONSUMERS	-$1,273.8 (+66%)	547.50+ 547.50+ 1466.16= $2,561.16	547.50+ 1095+ 2932.31= $4,574.81	N/A	N/A
3 CONSUMERS	-$1,858.89 (+95%)	547.50+ 547.50+ 1466.16= $2,561.16	547.50+ 1095+ 2932.31= $4,574.81	547.50+ 1642.50+ 4398.47= $6,588.47	N/A
4 CONSUMERS	N/A	N/A	N/A	547.50+ 1642.50+ 4398.47= $6,588.47	547.50+ 2190+ 5864.62= $8,602.12

Figure A.1: KiB @ 45000 (45 MiB/s, 50 shards)

	0 EFO CONSUMER	1 EFO CONSUMER	2 EFO CONSUMERS	3 EFO CONSUMERS	4 EFO CONSUMERS
1 CONSUMER	$1182.60	1182.60+ 1182.6+ 3258.13= $5,623.33	N/A	N/A	N/A
2 CONSUMERS	$1182.60	1182.60+ 1182.6+ 3258.13= $5,623.33	1182.60+ 2365.20+ 6516.26= $10,064.06	N/A	N/A
3 CONSUMERS	N/A	1182.60+ 1182.6+ 3258.13= $5,623.33	1182.60+ 2365.20+ 6516.26= $10,064.06	1182.60+ 3547.80+ 9774.39= $14,504.79	N/A
4 CONSUMERS	N/A	N/A	N/A	1182.60+ 3547.80+ 9774.39= $14,504.79	1182.60+ 4730.40+ 13032.52= $18,945.52

Figure A.2: 5 KiB @ 20000 (100 MiB/s, 108 shards)

	0 EFO CONSUMER	1 EFO CONSUMER	2 EFO CONSUMERS	3 EFO CONSUMERS	4 EFO CONSUMERS
1 CONSUMER	$1478.25	1478.25+ 1478.25+ 4072.67= $7,029.17	N/A	N/A	N/A
2 CONSUMERS	$1478.25	1478.25+ 1478.25+ 4072.67= $7,029.17	1478.25+ 2956.50+ 8143.34= $12,578.09	N/A	N/A
3 CONSUMERS	N/A	1478.25+ 1478.25+ 4072.67= $7,029.17	1478.25+ 2956.50+ 8143.34= $12,578.09	1478.25+ 4434.75+ 12218= $18,131	N/A
4 CONSUMERS	N/A	N/A	N/A	1478.25+ 4434.75+ 12218= $18,131	1478.25+ 5913+ 16290.67= $23,681.92

Figure A.3: 10 KiB @ 12500 (125 MiB/s, 135 shards)

Appendix B

Amazon Kinesis on-demand monthly cost calculations tables (source: https://calculator.aws/#/createCalculator/KinesisDataStreams).

	0 EFO CONSUMER	1 EFO CONSUMER	2 EFO CONSUMERS	3 EFO CONSUMERS	4 EFO CONSUMERS
0 CONSUMER	$9,051.72 (i.e. put cost only)	$14,690.80	$20,329.87	$25,968.95	$31,608.02
1 CONSUMER	$13,562	$14,960.80	N/A	N/A	N/A
2 CONSUMERS	$18,074.24	$19,202.06	$20,329.87	N/A	N/A
3 CONSUMERS	N/A	$23,713.32	$24,841.13	$25,968.95	N/A
4 CONSUMERS	N/A	N/A	N/A	$30,480.21	$31,608.02

Figure B.1: Total cost for 1 KiB @ 45000 (45 MiB/s)

	0 EFO CONSUMER	1 EFO CONSUMER	2 EFO CONSUMERS	3 EFO CONSUMERS	4 EFO CONSUMERS
0 CONSUMER	$20,079.25	$32,610.53	$45,141.81	$57,673.09	$70,204.37
1 CONSUMER	$30,104.27	$32,610.53	N/A	N/A	N/A
2 CONSUMERS	$40,129.30	$42,635.55	$45,414.81	N/A	N/A
3 CONSUMERS	N/A	$52,660.58	$55,166.83	$57,673.09	N/A
4 CONSUMERS	N/A	N/A	N/A	$67,698.11	$70,204.37

Figure B.2: Total cost for 5 KiB @ 20000 (100 MiB/s)

	0 EFO CONSUMER	1 EFO CONSUMER	2 EFO CONSUMERS	3 EFO CONSUMERS	4 EFO CONSUMERS
0 CONSUMER	$25,091.76	$40,755.86	$56,419.96	$72,084.06	$87,748.16
1 CONSUMER	$37,623.04	$40,755.86	N/A	N/A	N/A
2 CONSUMERS	$50,154.32	$53,287.14	$56,419.96	N/A	N/A
3 CONSUMERS	N/A	$65,818.42	$68,951.24	$72,084.06	N/A
4 CONSUMERS	N/A	N/A	N/A	$84,615.34	$87,748.16

Figure B.3: Total cost for 10 KiB @ 12500 (125 MiB/s)

Author

Jean-Noël Moyne, Field CTO, Synadia

A full-stack CTO with extensive experience in distributed systems, messaging, networking and data stores and products spanning all of these attributes, including AI infrastructure and edge computing. Previously: Lava.ai CTO,TIBCO Software Fellow and LBL Staff Scientist

/in/jean-noel-moyne /JohnnyXmas

015-TP-1-25-03