

# Cost Effective Kafka

Strategies for Cost Reduction and Increased Efficiency

David Kjerrumgaard



# *Cost-effective Kafka*

BALANCING PERFORMANCE  
AND COST DEPLOYMENTS

DAVID KJERRUMGAARD



MANNING  
SHELTER ISLAND

For online information and ordering of this and other Manning books, please visit [www.manning.com](http://www.manning.com). The publisher offers discounts on this book when ordered in quantity.

For more information, please contact

Special Sales Department  
Manning Publications Co.  
20 Baldwin Road  
PO Box 761  
Shelter Island, NY 11964  
Email: [orders@manning.com](mailto:orders@manning.com)


© 2024 Manning Publications Co. All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by means electronic, mechanical, photocopying, or otherwise, without prior written permission of the publisher.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in the book, and Manning Publications was aware of a trademark claim, the designations have been printed in initial caps or all caps.

⊗ Recognizing the importance of preserving what has been written, it is Manning's policy to have the books we publish printed on acid-free paper, and we exert our best efforts to that end. Recognizing also our responsibility to conserve the resources of our planet, Manning books are printed on paper that is at least 15 percent recycled and processed without the use of elemental chlorine.

The author and publisher have made every effort to ensure that the information in this book was correct at press time. The author and publisher do not assume and hereby disclaim any liability to any party for any loss, damage, or disruption caused by errors or omissions, whether such errors or omissions result from negligence, accident, or any other cause, or from any usage of the information herein.

 Manning Publications Co.  
20 Baldwin Road  
PO Box 761  
Shelter Island, NY 11964

ISBN 9781633435469

Printed in the United States of America

# contents

---

Summary	1
Overview	3
Introduction	4
Cloud infrastructure costs of Kafka	4
Networking	6
Public networking	7
Private networking	8
Data replication	9
Storage	11
Local storage	12
Object storage	13
Computing costs	16
Cloud infrastructure costs observations	17
Reducing cloud infrastructure costs	18
Minimizing compute cost	19
Minimizing networking costs	21
<i>Reducing data transmission costs with private networking</i>	21
<i>Reducing replication costs with object storage</i>	22
Minimizing storage costs	22
<i>Separate I/O read and write paths</i>	23
<i>Reduce storage costs with tiered storage</i>	24
What is the ideal Kafka deployment?	24
Optimizing data storage tiers based on quality of service	24
Conclusion	26



## **Summary**

- Kafka has evolved from an Apache project to an open source protocol on which products are standardized. This coalescing around a single messaging protocol will continue to foster innovation while providing developers with a stable API to build their applications.
- The adoption of Kafka has been hindered by the significant operational costs of running it in the cloud, with infrastructure costs accounting for over 90% of the total expense. As a result, Kafka products designed to minimize these costs have emerged.
- An emerging architectural style within the Kafka ecosystem is so-called serverless implementations, which minimize storage and network costs by utilizing object storage instead of local disk. The drawback to these implementations is they can incur large latency penalties.
- With the right architecture and configuration, Kafka costs can be reduced without sacrificing performance for latency-sensitive use cases.



## **Overview**

Over the past few years, Kafka has transitioned from a streaming technology implementation to a messaging protocol that products standardize on. This transformation is evidenced in the proliferation of new Kafka-based platforms on the market today, such as RedPanda and WarpStream, each with its own value proposition. This paradigm shift from product to protocol has significantly increased competition within the streaming platform market and created a lot of confusion regarding the proper evaluation of these product offerings regarding architectural tradeoffs between costs and latency. Several products that focus on reducing the total costs associated with running Kafka do so at the expense of performance and vice versa.

This report tells you what you need to know to evaluate and manage the infrastructure costs associated with hosting a traditional Kafka deployment, in which the primary storage mechanism is local disk. It helps you better understand the proportionality of the costs across the three major cost areas for cloud computing: compute, storage, and network. Once you have a solid understanding of the economics of hosting Kafka in the cloud, we will discuss some common techniques used to minimize these costs, highlighting the architectural tradeoffs in terms of latency and availability.



## Introduction

Apache Kafka is a widely utilized technology with applications encompassing real-time data processing, stream processing, event sourcing, and messaging across various technical domains, including finance, healthcare, social media, and e-commerce. Donated to the Apache Software Foundation in 2011, Kafka quickly gained popularity due to its robust performance and scalability. It now enjoys an impressive adoption rate of over 30% by Fortune 500 companies (<https://mng.bz/lrPz>) and is the de facto industry standard for real-time data streaming and event-driven architectures.

This popularity has created a highly competitive market for data streaming platforms that adhere to the Kafka protocol, with several vendors seeking to differentiate themselves in the growing market. The most obvious example of this phenomenon is Confluent, which pioneered the Kafka market in 2014. Initially, its product offerings were tightly coupled with the Apache implementation. Over the past decade, Confluent Cloud's feature set has diverged significantly from the open source project, and the company now markets itself as 10× better than Apache Kafka (<https://mng.bz/Bgyq>).

More recently, there has been a shift toward serverless implementations of Kafka that rely solely on object storage for persisting messages. In addition to the serverless versions from the major players in the market, including Confluent, StreamNative, and Redpanda, several smaller companies, such as WarpStream and Bufstream, have serverless offerings.

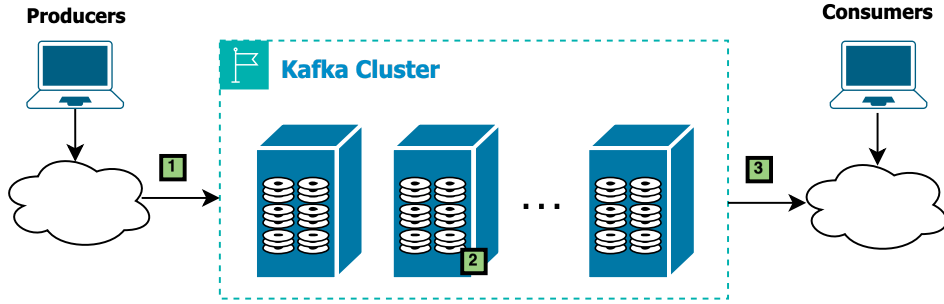
While all this innovation and competition is great for streaming platform users, it also presents challenges in evaluating streaming platforms holistically, including assessing the total cost of ownership and performance. To that end, StreamNative developed a methodology for evaluating streaming platforms that categorizes them based on capabilities. This methodology, known as the new CAP theorem for streaming (<https://mng.bz/dZ9z>), clarifies the architectural tradeoffs between cost, availability, and performance across the Kafka vendor offerings.

## Cloud infrastructure costs of Kafka

Let's examine the costs associated with hosting Kafka in a cloud environment, as they are the most significant cost factor in the total cost of ownership. In this section, I will highlight various factors that affect the infrastructure costs of running a streaming platform in the cloud regardless of the underlying deployment model.

Kafka is built to ingest and store substantial amounts of data from multiple sources sent over the network. The network bandwidth required to transfer this data is one of the most significant expenses associated with running Kafka in a cloud environment.

Once data is ingested, Kafka stores it on the brokers' local disks, as shown in figure 1. Kafka's architecture replicates the data across multiple brokers to ensure data durability in the event of a broker failure. While this replication is essential for maintaining the data's reliability and availability, it does increase the storage requirements threefold or more. This redundancy, combined with Kafka's ability to retain data for extended periods of time, makes storage a significant expense factor when running Kafka in a cloud environment.



**Figure 1.** Kafka ingests messages from producers and stores them on local disk before sending them to consumers. Each step of the three steps has an associated cost.

One of Kafka’s strengths is the ability to stream data to many consumers, possibly across different regions or services. After all, the entire point of publishing data to Kafka is to enable its dissemination to multiple consumers. Consequently, it is common for a single message to be transmitted to multiple consumers, thereby magnifying the outbound network bandwidth requirements by several orders of magnitude over the inbound network bandwidth (e.g., if a message is published once and consumed by 10 different consumers). Making matters worse is that unlike the inbound network traffic used to publish the data to Kafka, which is relatively inexpensive or even free, this network traffic is very expensive.

If we set aside the costs of hosting the brokers for now and only focus on the cloud infrastructure costs, it is relatively easy to devise a naive formula to calculate the infrastructure costs for operating Kafka in the cloud:

$$\text{Cloud Costs} = \text{Ingress Network Cost} + \text{Storage Cost} + \text{Egress Network Cost}$$

To better understand the cloud cost, let’s use this formula to estimate the cost for a scenario where your Kafka cluster ingests data at a rate of 10 MB/s and stores it for 24 hours, and three different users consume it.

We will start by calculating the total data volume for the 24-hour period, which is  $10 \text{ MB} \times (24 \times 60 \times 60) = 864 \text{ GB}$ , and assume a replication factor of 3. Next, we multiply this value by the network and storage rates the cloud provider charges. For now, let’s use AWS-published rates of \$0.00/GB for network ingress, \$0.08/GB/month of storage, and \$0.09/GB for network egress:

$$\begin{aligned} \text{Cloud Costs} &= (\$0.00 \times 864) + (\$0.08 \times 864 \times 3) + (\$0.09 \times 864 \times 2) \\ &= \$0.00 + \$207.36 + \$155.52 \\ &= \$362.88 \end{aligned}$$

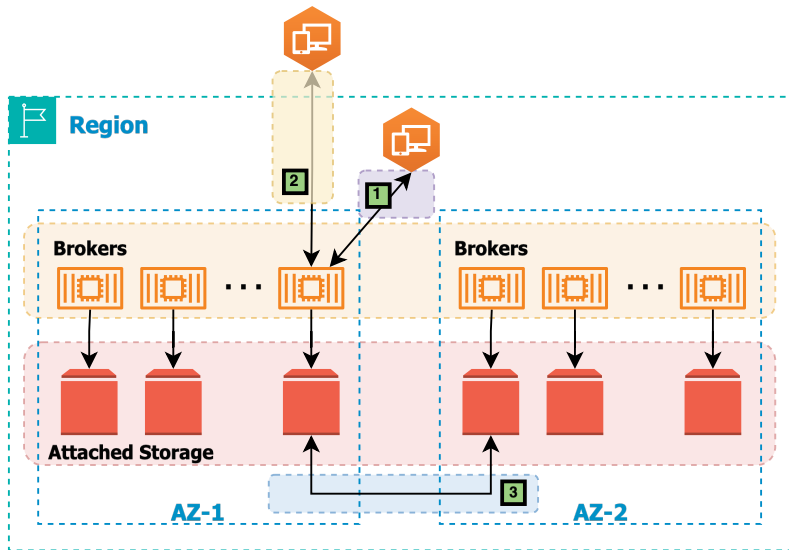
However, assessing the cost of operating a cloud-based streaming platform is a bit more complex than this example due to underlying variables affecting the price. Thus far, we have ignored the computing costs for hosting the Kafka brokers and the networking

associated with data replication. However, we will slowly factor in these costs as we build out the cost model.

## Networking

Because data streaming platforms' primary purpose is to move data from producers to consumers, networking emerges as a significant expense when hosting these platforms in the cloud. As you can see in figure 2, the networking costs associated with a cloud-hosted streaming platform fall into one of two categories:

- Data transmission costs associated with data ingress and egress due to messages published and consumed by client applications or services. Complicating matters is that data transfer can occur on different networking tiers, which have different costs associated with them.
- Costs associated with inter-availability zone (inter-AZ) replication network traffic due to Kafka's internal replication mechanism. This traffic travels over the inter-AZ networking tier.



**Figure 2.** The traffic between Kafka brokers (path 1) and clients (path 2) is referred to as data transmission, while the network traffic between Kafka brokers is referred to as replication traffic (path 3).

Considering these two categories, a more accurate formula for calculating networking costs begins to take shape—one that accounts for the cost differences between the networking tiers and includes the data replication costs:

$$\text{Network Costs} = \text{Data Transmission Costs} + \text{Replication Costs}$$

where

$$\text{Data Transmission Costs} = \text{Ingress Costs} + \text{Egress Costs}$$

The data transmission costs are those associated with the data traveling over paths 1 and 2, respectively, in figure 2. The only difference between them is the placement of the client applications relative to the Kafka brokers themselves. Both inter- and intra-regional network traffic are possible simultaneously.

This data transmission significantly affects infrastructure costs, as all major cloud service providers levy considerable fees for data egress and traffic between AZs and regions. Table 1 shows that data egress to the internet is the most expensive across all cloud providers.

**Table 1. Networking costs across the three major cloud vendors**

Traffic type	Cost/GB (In/Out)		
	AWS network costs <sup>a</sup>	Google network costs <sup>b</sup>	Azure network costs <sup>c</sup>
Internet	\$0.00/\$0.09	\$0.00/\$0.11	\$0.00/\$0.08
Inter-regional	\$0.01/\$0.01	\$0.02/\$0.02	\$0.02/\$0.02
Inter-AZ	\$0.01/\$0.01	\$0.01/\$0.01	\$0.00/\$0.00
Intra-AZ	\$0.00/\$0.00	\$0.00/\$0.00	\$0.00/\$0.00

<sup>a</sup><https://mng.bz/V2Qr>. <sup>b</sup><https://mng.bz/x6xq>. <sup>c</sup><https://mng.bz/AaXE>.

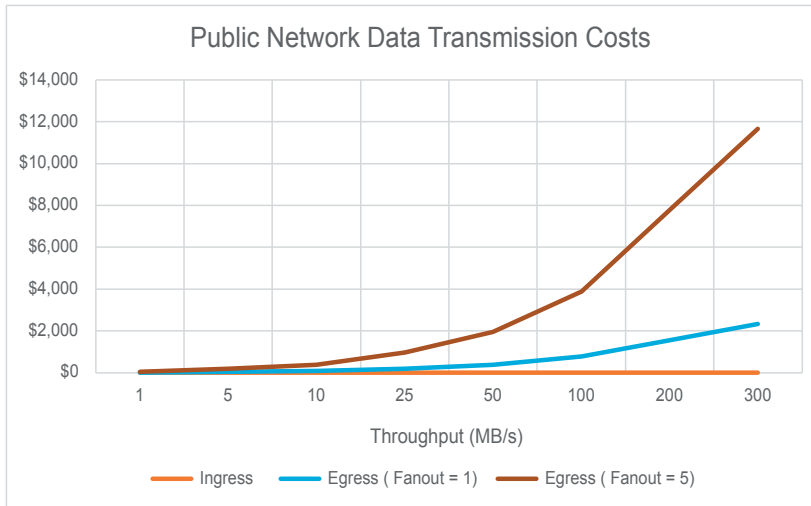
Given the differences in data transmission costs between networking tiers, examining these costs under two distinct scenarios is best. The first scenario is where all network traffic into and out of the Kafka cluster traverses the public internet in some way, including the use of VPNs. These scenarios are common when using a Kafka vendor's hosted solution that runs inside the vendor's cloud environment because all the data travels over the public internet.

## Public networking

Given that the internet's ingress cost is zero, the data transmission cost in these scenarios consists solely of the data egress costs, which are based on the number of times a message is read after it is published. This component is often referred to as the fanout ratio and varies from topic to topic. Thus, the formula for data transmission costs is

$$\text{Data Transmission Costs} = 0 + (\text{Fanout Ratio} \times \text{Workload} \times \text{Internet Egress Rate})$$

Figure 3 shows the data transmission costs associated with various throughput rates on AWS using internet-based ingress/egress. Again, since data ingress is free, the data transmission costs consist entirely of the egress cost. However, the egress costs grow linearly as a function of the fanout ratio, meaning the more times you read the data, the more it costs.



**Figure 3. Hosted data transmission costs for various workloads**

As you can see, even a modest fanout ratio significantly increases networking costs due to the cloud provider’s premium on outbound network traffic (e.g., \$0.09/GB). Thus, in use cases such as real-time analytics for IoT devices, in which different consumers might read the same IoT sensor data for anomaly detection, predictive maintenance, and real-time monitoring dashboards, these costs can add up quickly!

### Private networking

The second basic scenario for data transmission involves routing all network traffic to and from the Kafka cluster through a private network, such as VPC peering or private links. However, this option is feasible only if all client applications are also operating within the same private network, making it suitable for deployments where customers use their own cloud infrastructure to host the vendor’s Kafka services rather than depending solely on the infrastructure provided by the vendor, which is the case for hosted solutions such as Confluent Cloud, MSK, etc.

These bring-your-own-cloud (BYOC) managed service offerings can greatly reduce the networking costs associated with data egress by shifting the traffic to a lower-cost tier (e.g., inter-region or inter-AZ). This results in savings of 89%, 81%, and 75% per gigabyte of egress traffic on AWS, Google, and Azure, respectively. Unlike hosted solutions, however, costs are now associated with data ingress in these environments because inbound traffic usually travels on either inter-regional or inter-AZ networking. For the purpose of this analysis, let’s assume a worst-case scenario in which 100% of the ingress traffic comes from inside the private network. Updating the formula to reflect the costs associated with data transmission yields the following formula:

$$\text{Data Transmission Costs} = (W \times \text{Regional Ingress Cost}) + (FO \times W \times \text{Regional Egress Cost})$$

where

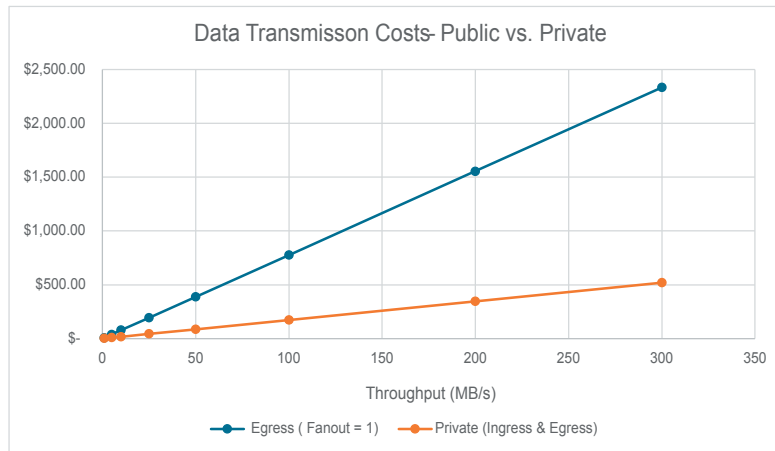
$$W = \text{Total Kafka Workload, and } FO = \text{Fanout Ratio}$$

Let's revisit the scenario in which your Kafka cluster ingests data at a rate of 10 MB/s and stores it for 24 hours, and three different users consume it. The data transmission costs on AWS would be

$$\text{Data Transmission Costs} = (\$0.00 \times 864) + (\$0.09 \times 864 \times 2) = \$155.50$$

While these BYOC offerings offer significant network savings, they are more complex in their network configuration, which requires some additional upfront effort. In addition, the licensing costs for these offerings are generally more expensive than hosted, which somewhat offsets the network savings.

Let's examine the potential cost savings associated with using private networking compared to public networking. Figure 4 compares the total data transmission costs (ingress and egress) across different workloads in both environments for a fanout ratio of one.



**Figure 4. Public vs. private data transmission costs**

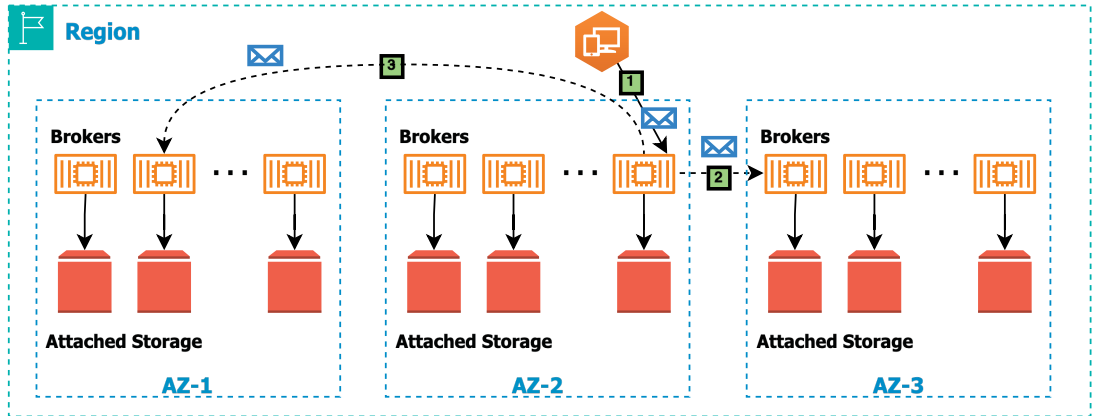
An increase in fanout ratios would compound these cost savings, making private networking an attractive option for analytical use cases where the same data is read multiple times. A good example would be real-time customer behavior analysis in which a continuously running Flink job reads the same topic data multiple times to perform sessionization, generate personalized recommendations, etc.

## Data replication

Now that we have effectively covered the data transmission costs, let's focus on the networking costs associated with Kafka's internal replication mechanism. Kafka's

leader–follower protocol can be configured to replicate data across multiple AZs to ensure high availability, fault tolerance, and data durability.

The volume of replication traffic is directly proportional to the number of AZs in which the cluster operates. This means that if you deploy your Kafka cluster across two AZs to survive a single zonal outage, at least one copy of the data must be replicated between the leader broker’s zone and a follower broker in the other zone. Similarly, if you have three AZs, the data must be transmitted to two different AZs, as shown in figure 5.



**Figure 5.** Messages published to AZ-2 (step 1) are automatically replicated to Kafka Brokers in AZ-1 and AZ-3 (steps 2 and 3).

In a multi-AZ deployment, Kafka’s replication mechanism will send a copy of every message to each AZ in the cluster, resulting in inter-AZ networking charges. Assuming that your replication factor is less than or equal to the number of AZs in which your cluster is deployed, the formula for data replication costs is

$$\text{Replication Cost} = \text{Workload} \times (\text{Replication Factor} - 1) \times \text{Inter-AZ Replication Cost}$$

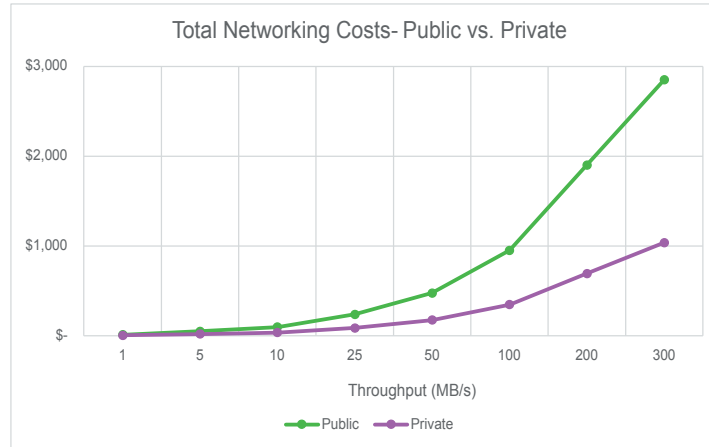
Since all this traffic is across AZs within the same region, these costs will be identical for public and private network deployments. Furthermore, this cross-AZ traffic is at the least expensive network tier, costing only \$0.01/GB on AWS or Google. Calculating the replication cost for our 10 MB/s ingestion scenario would yield the following result on AWS:

$$\text{Replication Cost} = 864 \text{ GB} \times (3 - 1) \times \$0.01/\text{GB} = \$17.28$$

As you can see, even though replication requires twice the network traffic, the cost is only a fraction of the data transmission charges for the same workload. This difference can be attributed to the significantly lower rates for inter-AZ traffic, as shown in table 1. The effect of data replication traffic is identical for both private and public networking options, adding \$0.02/GB to the networking costs in both environments because

a copy of the data must be sent to two different AZs, thus incurring the \$0.01/GB cost twice.

Accounting for both data transmission and data replication, figure 6 shows the networking costs incurred at different workloads for a Kafka cluster deployed across three AZs. As you can clearly see, the cost savings from using private networking to avoid the cloud providers' most expensive networking tier grows exponentially as the load increases.



**Figure 6. Public vs. private total network costs, including three AZ replications**

Thus, the use of private networking is a clear-cut winner in terms of lowering the total cost of operating a Kafka cluster in the cloud. However, this option is unavailable for any vendor-hosted options, as the Kafka cluster must run in the vendor's network rather than your own. This requirement makes internet-based access the only option, resulting in the higher costs depicted by the green line in figure 6.

## Storage

Given that data streaming platforms' primary purpose is to retain data for analysis and future consumption, storage emerges as a significant expense when hosting these platforms in the cloud. Each cloud provider has various storage classes or tiers with differing costs based on performance, availability, and access frequency. These options range from locally attached NVMe and SSD drives that support thousands of IOPS (input/output operations per second) to object storage such as Amazon S3.

As one might expect, cloud providers charge a premium for disk performance. Consequently, local storage costs are considerably higher across all major cloud service providers than object storage. Table 2 shows the costs across the various storage tiers for three major cloud providers.



**Table 2. Storage costs across the three major cloud vendors**

Traffic type	AWS storage costs	Google storage costs	Azure storage costs
Block			
HDD	\$0.045	\$0.04	\$0.0265
SSD	\$0.08	\$0.17	\$0.196
Object store	\$0.023	\$0.026	\$0.0208

These storage options fit into one of two broad categories, each with its own cost structures and performance profiles. The most commonly used category is *local storage*, which refers to storage options directly attached to the compute instances. These options include ephemeral and elastic block-level storage devices, used when very low read and write latency is required.

The second category is *object storage*, which is optimized to store large amounts of unstructured data, provide scalability, and be a cost-effective means for storing data that does not require the low-latency, high-performance characteristics of block storage. Unlike local storage, which is directly associated with specific compute instances, object storage operates independently and can be accessed from any location via APIs.

### Local storage

Storage devices in this category provide the best performance in IOPS, with high throughput and low latency, and are necessary to support traditional real-time use cases with strict latency requirements, such as real-time fraud detection. Therefore, data associated with these use cases must be persisted in local storage.

Traditional Kafka implementations combine serving and storage functions within a single node, typically using general-purpose SSDs (e.g., gp3) attached volumes to support low latency writes. This configuration leads to the following calculation for Kafka's storage cost:

$$\text{Storage Costs} = (W \times RF) \times (DR \% \times \text{Local Tier Cost}) + (W \times RF) \times (DR \% \times \text{Object Tier Cost})$$

where

$$W = \text{Total Kafka Workload}, RF = \text{Replication Factor}, DR = \text{Data Retention Percentage by Tier}$$

As you can see from the formula, storage costs increase linearly as your data volume grows. Therefore, the easiest way to reduce these costs is to reduce your replication factor or data retention periods. However, doing so comes at the expense of lower data durability guarantees or smaller windows of data available for analysis. Calculating the storage cost for our 10 MB/s ingestion scenario using only attached SSD volumes yields the following result on AWS:

$$\text{Storage Costs} = (864 \text{ GB} \times 3) \times 100\% \times \$0.08 = \$207.36$$

In addition to its ability to provide consistent single-digit millisecond latency read and write performance for Kafka, local block storage also has the advantage of having no

additional costs associated with reading the data. Thus, it has a write-once/read-many-times cost structure, unlike other storage options.

## Object storage

Object storage is a highly scalable technology for storing large amounts of data; it is typically implemented as a stand-alone system, such as Amazon S3. These systems are designed to handle vast amounts of unstructured data across distributed environments. They offer seamless scalability, durability, and accessibility, making them ideal for read-heavy workloads and sequential access patterns, such as log ingestion and analysis, which involves processing large volumes of data in the order they were written.

Data stored in object storage can only be accessed via APIs (e.g., RESTful HTTP/HTTPS). This requirement introduces additional latency compared to directly attached storage. It also incurs an additional cost, as all major cloud providers charge for these API calls, which include PUT, GET, DELETE, and LIST operations. Factoring in the per-request costs associated with writing to object storage leads to the following formula:

$$\text{Cloud Storage Costs} = \text{Data Storage Costs} + \text{Data Write Costs}$$

where

$$\begin{aligned} \text{Data Storage Costs} &= (\text{Workload} \times \text{Data Retention \%} \times \text{Object Storage Cost}), \text{ and} \\ \text{Data Write Costs} &= \text{PUT Request} \times \text{Request Cost} \end{aligned}$$

Here, the storage costs are divided into two distinct categories. The first category is the traditional capacity-based cost calculation derived from the throughput, data retention percentage, and object storage costs. The second cost category is related to object storage requests.

The number of requests is at the crux of the data access cost calculation. The number of requests required to store the data is based on what is referred to as the *request size*, which represents the predetermined size of each payload that Kafka will buffer before publishing to object storage (e.g., 512 KB). Once a request size is chosen, the formula for calculating the number of PUT requests is straightforward:

$$\text{PUT Request} = \text{Workload} / \text{Request Size}$$

Let's calculate some real numbers using the 10 MB/s use case to better understand the costs associated with using object storage. We will use a request size of 1 MB, leading to the following calculations on Amazon S3, which charges \$0.005 per 1,000 PUT requests:

$$\begin{aligned} \text{Data Storage Costs} &= (864 \text{ GB} \times 100 \% \times \$0.023) = \$19.87 \\ \text{PUT Requests} &= 864 \text{ GB} / 1 \text{ MB} = 864,000 \\ \text{Data Write Costs} &= 864,000 \times \$0.000005 = \$4.32 \\ \text{Cloud Storage Costs} &= \$19.87 + \$4.32 = \$24.19 \end{aligned}$$

Clearly, the savings are significant compared to the costs of storing the data entirely on attached block storage, which is \$155.50. The key factors driving these steep savings are twofold: first and foremost, the per-GB storage cost for object storage is 75% less than

block storage (\$0.023 vs. \$0.08). Second, Amazon S3 automatically replicates your data across multiple AZs, eliminating the need to store multiple replicas of the data. Thus, your storage requirements and the corresponding cost are cut by 66%.

However, it is important to remember that, unlike block storage, reading the data from Kafka in the form of metered API GET calls has a cost. Therefore, we must account for this expense in our formula. Intuitively, we know that the number of GET API calls required to read an entire dataset is equal to the number of PUT calls it took to write; we can express the total number of GET requests in terms of the fanout ratio:

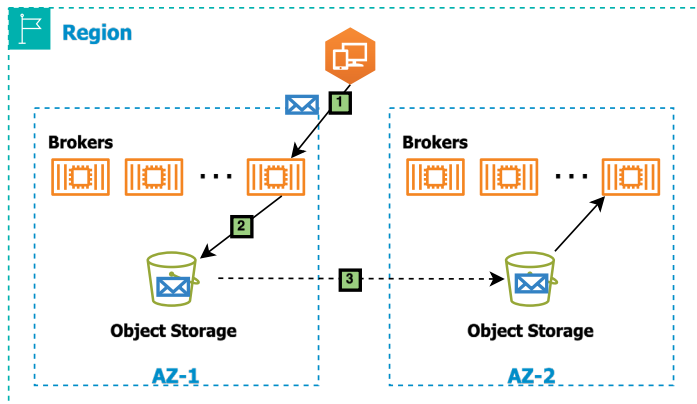
$$\begin{aligned} \text{Data Read Costs} &= \text{GET Request} \times \text{Request Cost} \\ \text{GET Request} &= \text{PUT Request} \times \text{Fanout Ratio} \end{aligned}$$

Thus, the higher the fanout ratio, the higher the read costs. Let's calculate the cost of reading our 10 MB/s dataset using a few different fanout ratios to illustrate that point better. Starting with the cost for fanout ratios of 1, 100, and 1,000, using the rate of \$0.0004 per 1,000 GET requests charged by AWS:

$$\begin{aligned} \text{Data Read Costs Fanout of 1} &= \$0.34 \\ \text{Data Read Costs Fanout of 100} &= \$34.00 \\ \text{Data Read Costs Fanout of 1,000} &= \$340.00 \end{aligned}$$

While these fanout numbers might seem high, they are not uncommon for analytical use cases in which the data inside Kafka is queried and analyzed by dozens of different Flink applications, dashboards, and ad hoc queries. Furthermore, they illustrate that the costs of reading the data can quickly erode the cost savings you expect to achieve from using object storage. This important cost factor is often overlooked when evaluating the total costs of using object storage.

Some Kafka systems use object stores as the only storage layer, accepting the higher latency of object storage in exchange for lower storage costs. These Kafka-compatible are often referred to as “serverless” due to the complete elimination of local disk storage. Figure 7 depicts an architecture that relies solely on object storage and its internal replication mechanism.



**Figure 7.** Messages published to a Kafka broker in AZ-1 (step 1) are stored in object storage (step 2) and automatically replicated to different AZs (step 3).

These systems' exclusive use of object storage minimizes the infrastructure footprint. It eliminates inter-zone network traffic by relying upon the object storage's underlying replication mechanism to make the data available in other AZs.

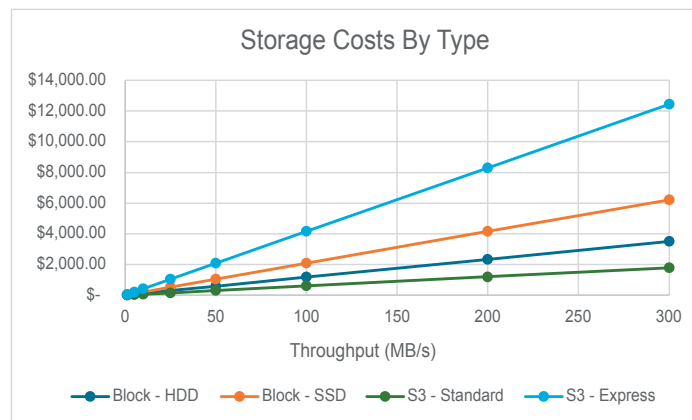
While these systems are fully optimized for cost efficiency in terms of using the lowest cost storage option and eliminating network traffic associated with data replication, the tradeoff in performance is significant due to the reliance upon cloud object storage.

Data publication and retrieval p99 latencies from object storage are typically measured in seconds instead of milliseconds, representing a 10× performance degradation compared to performance-optimized categories. For those not willing to sacrifice latency but still want to use a pure object storage approach, the best option is to use AWS's new S3-Express storage class, which provides low latency performance on par with local SSDs.

Unfortunately, this storage option costs twice that of traditional SSDs, which essentially defeats the purpose of using object storage in the first place. From a cost perspective, the increased storage cost outweighs the savings from eliminating inter-AZ replication. Combining the local storage costs and networking costs associated with Kafka replication yields a total cost of ( $\$0.08/\text{GB} + \$0.02/\text{GB}$ ), or  $\$0.10/\text{GB}$ . This is much cheaper than the  $\$0.16/\text{GB}$  storage rate associated with AWS' S3 Express storage.

So, unless the cost for this highly performant object storage class comes down considerably, such a configuration is not cost-effective. Making matters worse, Azure recently eliminated all network charges associated with inter-AZ traffic, which further erodes the cost savings of these “serverless” offerings as other cloud vendors may follow suit to stay competitive.

Figure 8 shows the storage costs associated with various data volumes across the storage options we have discussed. As expected, the storage costs increase linearly with respect to data volumes, as does the corresponding cost savings from using lower-cost storage classes.



**Figure 8.** The relative cost of storage across the different AWS storage classes

## Computing costs

Computing costs are the expenses associated with running virtual servers in the cloud. They include the hourly or per-minute charges for the compute instances, which can vary based on the instance type, size, and region. These costs can also differ significantly based on the cloud provider (AWS, Azure, or Google Cloud), the specific services used, the pricing models (e.g., on-demand, reserved instances, spot instances), and the region where the resources are deployed.

Thus far, we have been building out a cloud infrastructure cost model that is agnostic to the underlying Kafka implementation primarily because the implementation does not affect the amount of data that needs to be transmitted over the network and stored on attached disks. However, at this point in our total cost of ownership (TCO) model, we must acknowledge that different Kafka products will have different throughput performance characteristics. In fact, some Kafka products differentiate from others based primarily on their efficient use of computing resources.

To standardize our compute cost evaluation, let's consider the concept of a compute unit (CU), defined by the capacity of one CPU and 8 GB of memory, as a baseline for comparison. This standardization allows us to evaluate (i) the cost per compute unit and (ii) the throughput each technology can achieve per compute unit (aka, the throughput efficiency metric (TPE)). In simple terms, the TPE defines the number of megabytes per second of throughput that can be processed by a single computing unit; the higher this metric, the more efficient the platform is in throughput.

For Apache Kafka, we rely on the Open Messaging Benchmark framework (<https://openmessaging.cloud/docs/benchmarks/>) to calculate the TPE metrics using a hardware setup consisting of three i3en.6xlarge nodes, totaling 72 CUs. The benchmark report revealed that Kafka could support 280 MB/s for both ingress and egress traffic, equating to 3.89 MB/s TPE for both ingress and egress per CU.

The TPE metric allows us to develop the following formula to estimate the hourly cloud computing costs based on the workload size that accounts for differences in efficiency between Kafka systems:

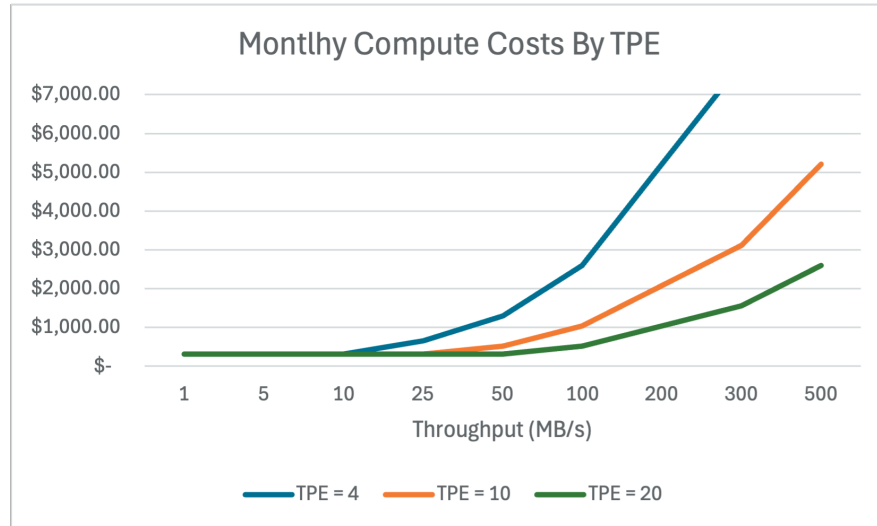
$$\text{Hourly Compute Cost} = (\text{Workload}/\text{TPE}) \times (\text{Vendor VM Cost}/\text{CU} - \text{Discounts})$$

We can take the total workload in terms of throughput and divide it by the vendor's specific efficiency metric (TPE) to arrive at the total number of VMs required to accommodate the workload. Then, we multiply that number by the vendor cost per VM, factoring in any cloud vendor discounts, to arrive at the hourly computing costs. Consider the following example to illustrate this point better for our hypothetical 10 MB/s workload running on a Kafka platform with a TPE of 4 on AWS:

$$\text{Hourly Compute Cost} = x \ \$0.1425 = \$0.35$$

Given a TPE of 4 and a workload of 10 MB/s, the Kafka brokers would require 2.5 CUs' worth of computing resources to handle this load—in practical terms, 2.5 vCPUs and 20 GB of RAM. Assuming a per-hour cost of \$0.1425 per CU, the computing resources for the Kafka brokers would cost \$0.35/h.

Figure 9 shows the effect of efficient Kafka implementations on cloud computing costs across various workloads. As expected, platforms capable of processing more throughput per computing unit require fewer computing resources overall.



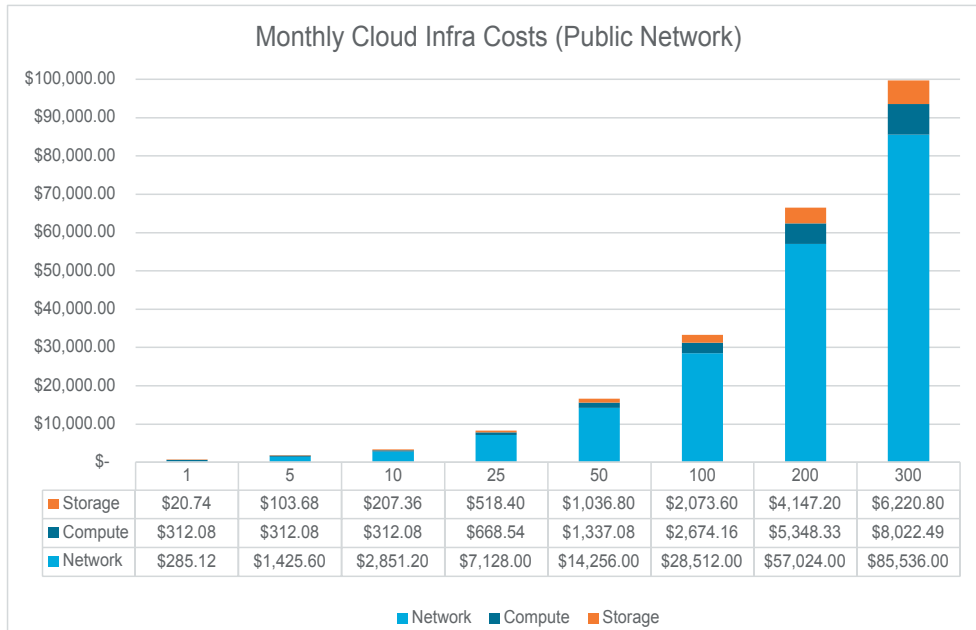
**Figure 9. Kafka's total cloud infrastructure costs for various workloads**

However, it is worth noting that the compute costs remain the same across the lower throughputs (i.e., 1–10 MB/s) due to Kafka's requirement of a minimum cluster size of three nodes.

### **Cloud infrastructure costs observations**

The numbers we have calculated thus far represent the infrastructure costs for a traditional Kafka deployment on AWS, deployed across three AZs, using local SSD storage for 100% of the data with a retention period of 24 hours and a fanout ratio of just 1. Figure 10 shows the breakdown of the cloud costs across all three areas—storage, networking, and compute—under these assumptions. Reviewing this data helps us understand the proportionality of the infrastructure costs across the three major cost areas for cloud computing.

As we can see, some interesting observations emerge from this data. First and foremost, networking costs quickly become the dominant factor, which can be attributed in part to the high cost associated with network egress relative to all other costs. Surprisingly, storage costs are not as significant a factor as one would expect, given the relatively high cost of GB storage. This result is because Kafka's data retention policies actively delete data, which, in turn, minimizes storage utilization.



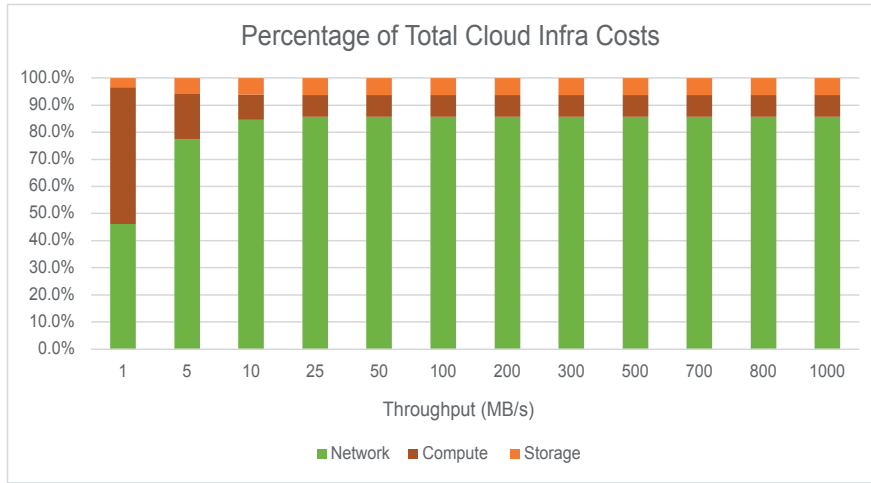
**Figure 10. Kafka's cloud infrastructure cost breakdown**

### Reducing cloud infrastructure costs

Today, there is a heightened focus on operational efficiency and cost reduction, which has slowed or, in some cases, even reversed the migration to cloud solutions. We navigate a complex landscape that straddles both on-premises and cloud environments, where cost and network efficiency are critical considerations. This scenario underscores the need to shift toward more cost-aware and sustainable architectural approaches in data streaming services.

The first step in this process was to conduct an in-depth analysis of the cloud costs associated with operating Kafka across various workloads. As illustrated in figure 11, the analysis revealed that cloud costs are distributed across three primary cost centers, with networking emerging as the largest cost factor by far. Armed with this insight, we strategically prioritized our efforts in the areas where we could maximize operational efficiency for Kafka users inside our new streaming platform called Ursa (<https://streamnative.io/ursa>).

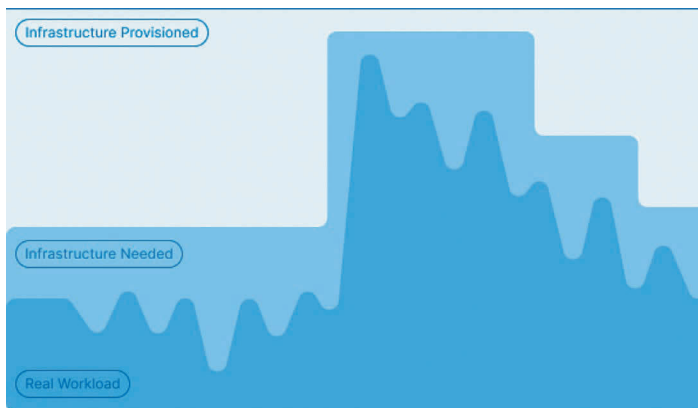
Ursa is an Apache Kafka API-compatible data streaming engine that runs directly on top of commodity object stores like AWS S3, GCP Google Cloud Storage (GCS), and Azure Blob Storage (ABS) and stores streams in lakehouse table formats (such as Hudi, Iceberg, and Delta Lake). Built on top of Apache Pulsar's two-tiered architecture, Ursa is designed to minimize cloud infrastructure costs across all three areas, as we will discuss.



**Figure 11.** The distribution of Kafka’s cloud infrastructure costs

### Minimizing compute cost

*Elasticity* refers to a system’s ability to automatically and dynamically adjust its resources in response to fluctuating workloads. With an elastic streaming platform, additional brokers can be spun up to handle the increased load during peak times. Conversely, during periods of low activity, unnecessary resources can be decommissioned, reducing costs associated with compute and storage. In short, elasticity ensures you’re only paying for the resources you need and avoiding the overprovisioning of resources, as shown in figure 12.



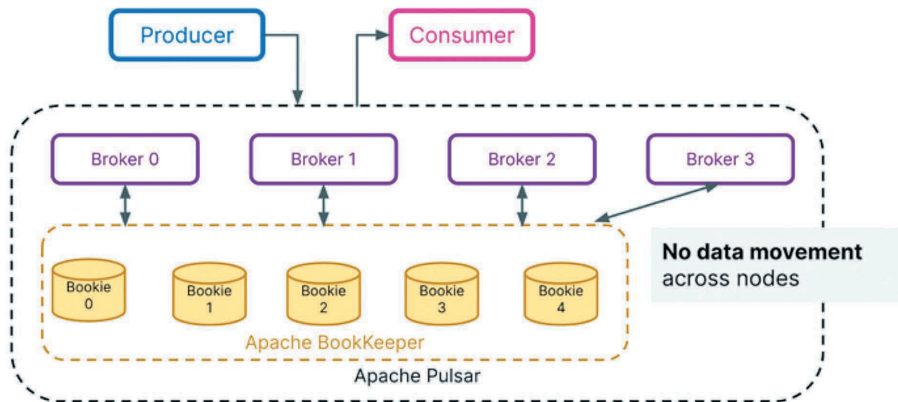
**Figure 12.** A platform’s elasticity allows it to adapt to workload fluctuations quickly, thereby reducing the cloud costs associated with overprovisioning.



In Apache Kafka, the data is stored directly on the broker nodes. In this single-tiered architecture, you cannot simply scale down your cluster by removing broker nodes, as it could lead to data loss. Consequently, this setup requires data rebalancing and movement whenever the cluster topology changes, such as when adding new nodes, removing old nodes, or dealing with failed nodes. The inelastic nature of Kafka’s architecture often results in overprovisioning the hardware for peak usage, which is very wasteful in terms of cloud infrastructure resource utilization.

We saw this inability to scale quickly as a major architectural flaw in Kafka, so while working at Yahoo in 2012, our founding team developed the first data streaming platform that separated the storage and compute layers, allowing them to scale independently. This revolutionary new approach resulted in a 100× improvement in elasticity compared to traditional streaming platforms, such as Apache Kafka, which stores the data on the brokers.

The crucial piece of the puzzle to making this a reality is Ursa’s segment-based storage architecture, which does not require data movement when scaling up storage. Brokers in Ursa divide the partitions into segments, which are evenly distributed across storage nodes. The location metadata of these segments is stored in a metadata storage for rapid access. Adding or removing a broker is straightforward and immediate, as no actual data is rebalanced or moved because the data resides in the storage layer, as shown in figure 13.



**Figure 13.** Ursa’s two-tiered architecture separates the storage of the message data from the brokers.

Ursa’s decoupled and rebalance-free architecture facilitates the independent scaling of cloud resources based on actual demand. Furthermore, these scaling events can be automated via horizontal pod autoscaling (HPA) or similar technology, resulting in significant cost savings.

## Minimizing networking costs

As you saw earlier in figure 11, the breakdown of infrastructure costs is not symmetrical across storage, networking, and compute. Networking costs are roughly 85% or more of the entire operational cost, depending on the fanout ratio. Furthermore, the networking costs do not decrease at scale. Therefore, the best way to minimize the overall operational cost of running Kafka in the cloud is to reduce these costs.

At StreamNative, we quickly realized that allowing our customers to run our Kafka service inside their own data center or cloud account is the best way to minimize the networking costs associated with data transmission. This is why we offer Ursa in several deployment models, including BYOC and on-prem.

### Reducing data transmission costs with private networking

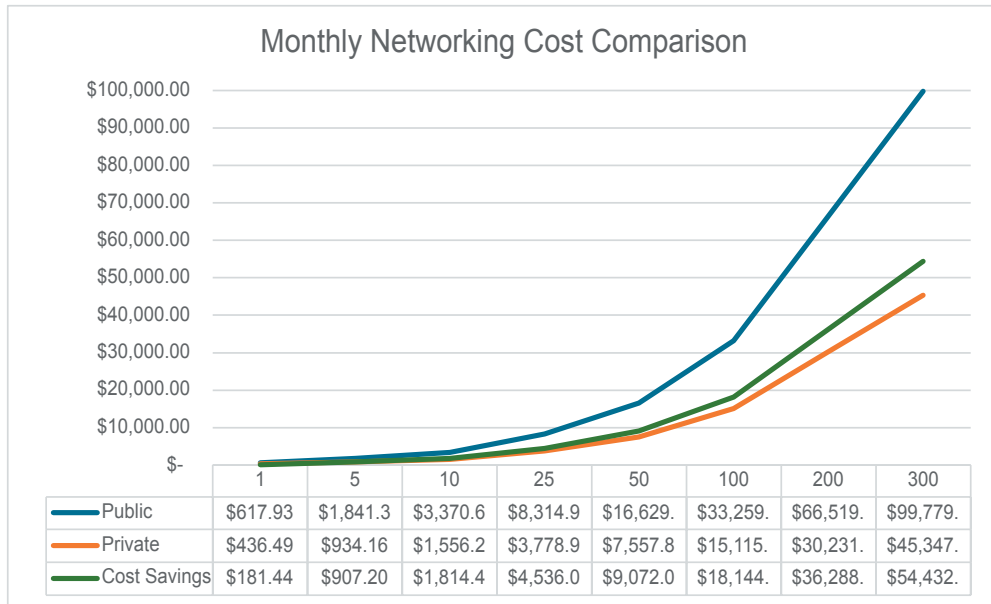
BYOC is a deployment model allowing organizations to run third-party software or services on their preferred cloud infrastructure within their accounts. This model offers businesses the flexibility to choose the cloud environment that best suits their operational needs, cost strategies, and compliance requirements. Furthermore, this option allows you to deploy your internal applications that need to interact with Kafka within the same cloud environment. Doing so improves latency performance and allows these applications to communicate with Kafka over lower-cost networking tiers, resulting in significant cost savings.

Both our BYOC and on-prem deployments support private networking options, such as VPC peering or private links. These allow companies to securely connect to services while reducing costs associated with public network traffic by interacting with Kafka over lower-cost networking tiers, including inter-AZ, which is free.

Figure 14 shows the total infrastructure costs across various workloads ranging from 1 MB/s up to 300 MB/s. The blue line represents the TCO when data transmission is done over the public internet, while the orange line shows the TCO when private networking is used instead. The cost savings, represented by the green line, quickly exceed the private networking costs.

As figure 14 shows, the ability to use less costly networking tiers results in significant savings over SaaS offerings that require public networking. For instance, at the 10 MB/s workload, the monthly TCO savings from transmitting data to Kafka clients over a private network is over \$1,814.

Not only does Ursa provide BYOC across all major cloud providers, including AWS, Google Cloud, and Azure, but more importantly, it doesn't sacrifice any functionality compared to our fully hosted SaaS offering. This is not the case for a lot of vendors (including Confluent Cloud; <https://www.confluent.io/learn/bring-your-own-cloud/>) that only offer certain services and features within their premium hosted tiers or don't offer BYOC at all.



**Figure 14.** The networking cost savings from running Kafka in a private networking environment grows exponentially as the workload increases.

### **Reducing replication costs with object storage**

There is an emerging trend of object-storage-based Kafka replacements, whose cost savings are derived from a combination of using lower-cost storage and eliminating networking costs associated with data replication. Instead of using Kafka's leader-follower protocol, these replacements rely upon the underlying object store to perform this replication.

Since this technique relies upon the cloud providers' object storage rather than some proprietary solution, it can be easily replicated by most Kafka products, including, but not limited to, Ursa. In fact, Ursa has the most mature tiered storage architecture in the Kafka market, with its first implementation becoming available in 2017. Since then, several improvements have been made that allow you to configure both the replica count and the frequency data, which is offloaded to object storage. Consequently, you can have a single replica on local storage before offloading the data object storage.

### **Minimizing storage costs**

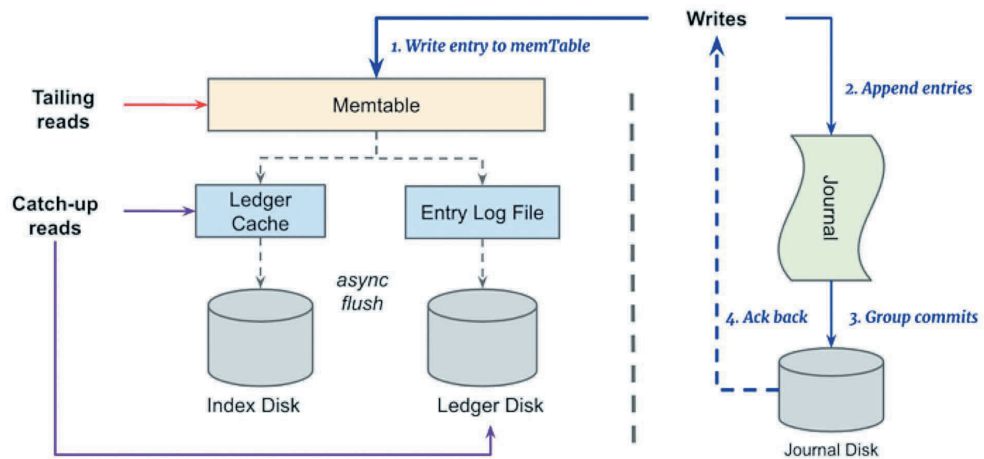
In traditional deployment configurations, Kafka combines serving and storage functions within a single node, typically using general-purpose SSDs (e.g., gp3) attached volumes to support low latency writes. However, as discussed in section 4, these costs are relatively high and grow linearly as your data volumes and data retention requirements increase.

Newer Kafka systems have incorporated lower-cost storage options into their offerings. These options provide cloud-based object storage for most platforms, which incur a significant latency penalty of 100 ms or more. However, StreamNative’s new Ursa™ platform does not provide this option; instead, it utilizes a dual-disk system on its storage nodes to handle write and read operations efficiently.

### Separate I/O read and write paths

Ursa™ uses separate storage devices to retain the message data. The journal disk acts as a buffer and ensures that all incoming messages are written to nonvolatile storage. Periodically, the incoming data is indexed and persisted to the ledger disk, which acts as long-term storage for the messages. Thus, the storage requirements for the journal disk are kept to a minimum, typically only enough to hold a few minutes’ worth of data.

For read operations, tailing reads are sourced directly from the in-memory cache, while catch-up reads come from the ledger disk and index disk, as shown in figure 15. This dual-disk strategy ensures that intense read activities do not affect the performance of incoming writes due to their isolation on different physical disks. Furthermore, it allows us to use different storage classes for each disk rather than a single disk for both reads and writes.



**Figure 15.** Ursa’s local storage architecture isolates the read and write paths, allowing you to use cost-effective storage for your data without sacrificing latency.

Consequently, we can employ general-purpose HDDs, which support high-throughput reads, for the ledger and index disks. Not only are the storage costs for these disks nearly 50% cheaper than SSDs (\$0.045 vs. \$0.08 per GB), but these disks represent nearly 95% of the storage capacity required. Only the remaining 5% of the capacity dedicated to the journal disk requires SSD drives.

Ursa's ability to utilize cheaper storage for storing messages results in significant savings over the local storage costs of traditional Kafka systems, which utilize a single storage type. This significant reduction in storage costs is consistent across a broad range of workload types and retention periods.

### **Reduce storage costs with tiered storage**

Kafka is known for its robust, scalable log storage capabilities. However, it was originally designed to use locally attached storage, such as high-performance SSDs, to provide low-latency streaming reads without compromising write latency.

As organizations seek to retain data for extended periods, this design can quickly result in higher storage costs due to the sheer volume of data. A common approach to alleviating storage costs is to move some of the data from local disks to lower-cost, scalable storage options such as cloud-based object storage (e.g., Amazon S3 or GCS). This process is commonly referred to as tiered storage offloading, and it leads to significant cost savings by optimizing the use of storage resources. It separates data into “hot” and “cold” tiers, with hot data being frequently accessed and stored on high-performance, more expensive local storage like SSDs.

This approach reduces the need for large, high-performance local disks, cutting down on storage costs. Additionally, object storage systems offer better scalability at a lower cost per unit of storage than local disks, allowing Kafka users to manage growing data volumes efficiently without significant upfront investment.

Apache Pulsar, upon which Ursa is built, has pioneered the concept of tiered storage. Using a segmented stream model, Pulsar's tiered storage was introduced with Pulsar 2.2.0. When a segment in Pulsar is sealed, it's offloaded to tiered storage based on configured policies. Data is stored in Pulsar's format with additional indices for efficient reading. Unlike some other tiered storage solutions, this approach allows brokers to read directly from tiered storage, saving memory, bandwidth, and cross-zone traffic.

### **What is the ideal Kafka deployment?**

One of the common patterns we have observed over the years of working with enterprise customers is that the latency requirements for different streaming datasets vary significantly depending on the use case.

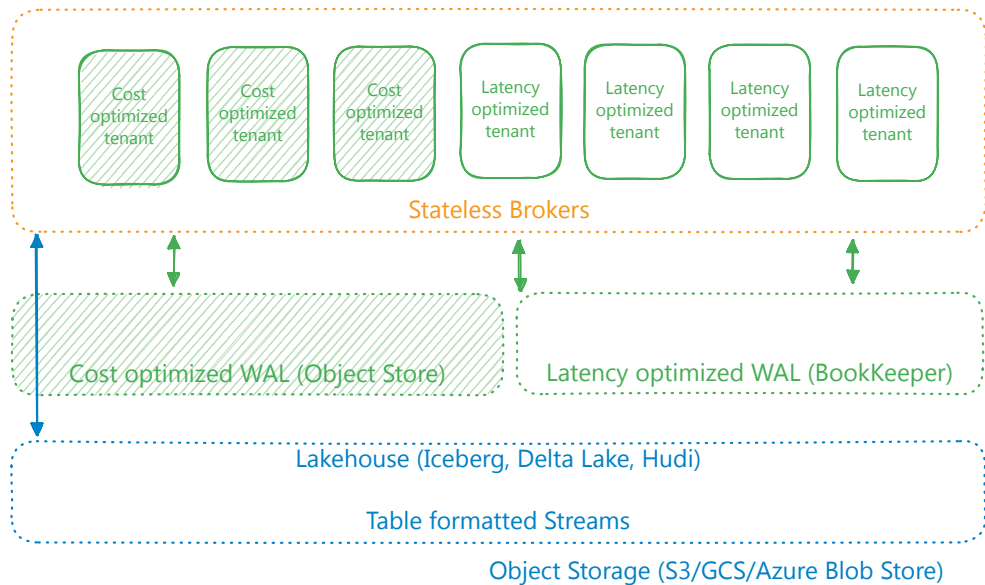
Take WeChat (<https://mng.bz/JNRv>), for example; only 1% of WeChat's use cases demand real-time data processing, characterized by message lifecycles of less than 10 minutes. In contrast, 9% of use cases necessitate catch-up reads and batch processing, relying on data freshness within a 2-hour window. The remaining 90% of use cases revolve around data replay and data backup, spanning data older than 2 hours.

If we were to keep all the data with varying lifecycle requirements in the local storage layer, it would pose a large cost challenge. Conversely, using an object storage-only Kafka engine such as Warp Stream would pose a challenge in meeting your latency service level agreements. Therefore, the optimal solution for this would be a Kafka platform that gives you the option to use the storage options that best suit your needs.

### Optimizing data storage tiers based on quality of service

While locally attached SSDs are required for latency-optimized workloads, this storage option depends on replicating data across multiple storage nodes in different availability zones as part of Kafka’s replication mechanism. This replication requires significant inter-AZ traffic for high-volume data streaming workloads, making operating in a multi-AZ deployment expensive. As we have pointed out earlier, the cost of inter-AZ data transfer can account for up to 90% of infrastructure costs when self-managing Apache Kafka.

Since Ursa already utilizes object storage, what if we eliminate the need for local disk as a WAL storage solution and instead directly leverage commodity object storage (S3, GCS, or ABS) as a write-ahead log for data storage and replication? This approach would eliminate the need for inter-AZ data replication and its associated costs. It is the essence of introducing a cost-optimized WAL based on object storage, which is at the heart of the Ursa engine, as illustrated in figure 16.



**Figure 16.** Ursa provides the option of latency-optimized or cost-optimized WAL on a per-tenant basis.

Ursa stores all message data in a giant, aggregated write-ahead log (WAL) for durability and supports two different implementations. The latency-optimized option stores the data on attached EBS volumes inside its storage layer (Apache BookKeeper), which ensures ultra-low read and write latency in the 10ms range. The cost-optimized option leverages commodity object storage (S3, GCS, or ABS) as a WAL for topic data.

What makes Ursa unique is its ability to support both storage options in a single platform, giving you the flexibility to choose the WAL implementation on a per-tenant basis. Workloads that must be optimized for latency can continue using a latency-optimized WAL backed by local disk, while latency-relaxed workloads can choose a cost-optimized WAL to avoid costly cross-AZ data transfers.

Ursa's flexible architecture gives you fine-grained control over your infrastructure spend, allowing you to make critical cost-benefit decisions for your streaming data. This architecture is unlike other solutions that utilize a single WAL implementation for the entire platform.

## **Conclusion**

In this whitepaper, we explored the costs of running Apache Kafka in the cloud, especially the hefty expenses tied to data ingestion, storage, and network usage. As we've seen, as data volumes grow, these costs become more significant, largely due to the need for fast local storage, extensive data replication, and high network usage. To address these challenges, we presented an alternative solution, Ursa, that allows users to choose storage options based on their specific latency requirements on a per-topic basis. This approach lets users balance cost and performance on a granular level, allowing you to save money on the bulk of your streaming storage costs without sacrificing performance for critical low latency use cases. In the end, Ursa helps maintain the perfect balance between performance and costs.

Ursa represents a significant advancement in data streaming. It simplifies the deployment and operation of data streaming platforms, accelerating data availability for your applications and lakehouses and reducing the costs of managing a modern data streaming stack.

While Ursa is still in its early stages, you can experience its capabilities. The Ursa engine, featuring Kafka API compatibility, lakehouse storage, and the No Keeper architecture, is available on StreamNative Cloud (<https://console.streamnative.cloud/>). If you want to learn more or try it out, sign up today (<https://console.streamnative.cloud/>) or talk to our data streaming experts (<https://streamnative.io/contact>).

