SkyStore: Cost-Optimized Object Storage Across Regions and Clouds

Shu Liu[1], Xiangxi Mo[1*], Moshik Hershcovitch[2*], Henric Zhang[1], Audrey Cheng[1]
Guy Girmonsky[2], Gil Vernik[2], Michael Factor[2], Tiemo Bang[1], Soujanya Ponnapalli[1]
Natacha Crooks[1], Joseph E. Gonzalez[1], Danny Harnik[2], Ion Stoica[1]
[1]UC Berkeley [2]IBM Research

ABSTRACT

Modern applications span multiple clouds to reduce costs, avoid vendor lock-in, and leverage low-availability resources in another cloud. However, standard object stores operate within a single cloud, forcing users to manually manage data placement across clouds, i.e., navigate their diverse APIs and handle heterogeneous costs for network and storage. This is often a complex choice: users must either pay to store objects in a remote cloud, or pay to transfer them over the network based on application access patterns and cloud provider cost offerings. To address this, we present SkyStore, a unified object store that addresses cost-optimal data management across regions and clouds. SkyStore introduces a virtual object and bucket API to hide the complexity of interacting with multiple clouds. At its core, SkyStore has a novel TTL-based data placement policy that dynamically replicates and evicts objects according to application access patterns while optimizing for lower cost. Our evaluation shows that across various workloads, SkyStore reduces the overall cost by up to 6× over academic baselines and commercial alternatives like AWS multi-region buckets. SkyStore also has comparable latency, and its availability and fault tolerance are on par with standard cloud offerings.

PVLDB Reference Format:
Shu Liu[1], Xiangxi Mo[1*], Moshik Hershcovitch[2*], Henric Zhang[1], Audrey Cheng[1], Guy Girmonsky[2], Gil Vernik[2], Michael Factor[2], Tiemo Bang[1], Soujanya Ponnapalli[1], and Natacha Crooks[1], Joseph E. Gonzalez[1], Danny Harnik[2], Ion Stoica[1] . SkyStore: Cost-Optimized Object Storage Across Regions and Clouds. PVLDB, 18(7): 2084 - 2096, 2025.
doi:10.14778/3734839.3734846

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/lynnliu030/vldb25.

1 INTRODUCTION

In the rapidly evolving landscape of cloud computing, applications increasingly span multiple regions and clouds. Organizations adopt multi-cloud software to reduce costs, avoid vendor lock-in, improve fault tolerance, increase the availability of specific capabilities beyond a single region or cloud, or support geo-distributed services [34, 51, 54]. For instance, deploying a model serving service on multiple clouds reduces monetary costs by up to 50% on low-availability resources (e.g., GPUs) compared to a single cloud [53]. Today, these applications rely on object storage services (e.g., Amazon S3, Google Cloud Storage, Azure Blob Storage, and IBM Cloud Object Store [11–14]) to manage vast amounts of data.

Unfortunately, existing object stores operate within their respective clouds and typically limit their operations to specific regions; commercial systems like AWS and GCP only support multi-region but not multi-cloud replication [17]. As a result, users manually handle data placement across clouds or regions, and their solutions cluster around two extremes: store locally or replicate everywhere. While storing all data in a single region simplifies data management and reduces storage costs, it increases egress expenses when data is accessed from another cloud region [2, 3, 8, 51]. On average, data transfers across clouds cost 23× more compared to transfers within the same cloud. On the other hand, replicating data to multiple regions and clouds [19, 32, 37, 48] may reduce network access fees, but can significantly increase storage expenses. For instance, storing the training data for a Llama3 model with 15 trillion training tokens (60 TB in size) [36] in AWS, GCP, and Azure standard storage buckets across different regions costs up to $300K per month [2, 3, 8].

A plethora of academic solutions have been proposed to address data storage in multi-region and multi-cloud settings. However, these solutions are optimized to reduce latency [15, 49, 50] and often ignore data transfer costs, which become prohibitive across multiple clouds. The most relevant work in this area is SPANStore [52], a multi-cloud storage system considering cost and latency tradeoffs. However, SPANStore does not account for replication costs and assumes that data access patterns do not change over time, significantly limiting its practical applicability.

Consequently, there is a need for a multi-region, multi-cloud data placement solution that minimizes the total monetary cost for various cloud applications. The key challenge in developing such a system is that cloud applications are highly diverse, and their data access patterns vary across several dimensions: object size, location distribution, and the recency and frequency of data accesses. For instance, for applications that perform repeated reads, like model training, it is cheaper to replicate data to accessed regions and avoid additional network costs for subsequent reads. In contrast, for applications that read infrequently, like satellite imagery, it is more cost-effective to pay for network transfers occasionally.

In this paper, we present SkyStore, a cost-optimized multi-cloud object store that adapts to the diverse and dynamic access patterns of applications. SkyStore provides a single uniform API that emulates a local object store and transparently manages data across clouds and regions while minimizing cost. In a nutshell, SkyStore provides an *overlay* cloud service on top of existing object stores

Proceedings of the VLDB Endowment, Vol. 18, No. 7 ISSN 2150-8097.
doi:10.14778/3734839.3734846

that operate in specific regions and clouds. It decides where an object should be stored and the locations it should be replicated to, if at all, and consistently manages these data copies.

At a high level, SkyStore solves a caching problem: it decides to "cache" (i.e., replicate objects to their accessed cloud region) and to "evict" (i.e., remove object copies from cloud regions if they are unlikely to be re-accessed) based on application access patterns. Unlike traditional caches that optimize for performance and have a finite capacity, SkyStore minimizes monetary costs as capacity is virtually unlimited; it accounts for the non-trivial network and storage costs incurred to cache objects.

Accordingly, SkyStore needs to make two decisions: (i) when to cache (i.e., replicate) an object and (ii) when to evict an object. First, SkyStore adopts a store-local and copy-on-read replication policy. When a user writes an object, SkyStore stores it in the local region to minimize cost and latency. If a user reads this object from another cloud region, SkyStore replicates it from an available region with the lowest transfer fees and ensures that network costs are only incurred for objects that are accessed.

Second, SkyStore leverages a novel adaptive Time-To-Live (TTL)-based eviction policy that balances the cost of storing and transferring an object if the object is accessed again. Our policy reduces costs by adapting the TTL assigned to objects based on the current workload and attempts to learn the optimal TTL for each replica. SkyStore captures past workload access patterns to estimate TTL costs and periodically updates these values while remaining robust to workload variance compared to traditional TTL-based methods (Section 6). We also show in Section 3.3.2 how latency considerations can be incorporated into our cost-centric framework, generalizing it to hybrid clouds [4, 37] and clouds without explicit cost models [6].

We implement SkyStore to seamlessly integrate with existing clouds (e.g., Azure Blob, AWS S3, and GCS). SkyStore offers a *virtual bucket* and *virtual object* abstraction via a standard S3-compatible API, allowing users to manage data as if all their data were in a single region. SkyStore provides the same consistency guarantees [21, 40, 46] as its underlying object stores and similar fault tolerance guarantees as existing services. We evaluate SkyStore on various workloads retrieved from IBM object store traces in the Storage Networking Industry Association (SNIA). Our prototype has comparable latency relative to the state-of-the-art systems, and our simulations show that SkyStore achieves up to 6× cost savings. In summary, our contributions are as follows:

(1) We design a novel cost-optimized data replication policy that can adapt to diverse workload patterns in the multi-region, multi-cloud setting.
(2) We implement SkyStore, a cost-efficient multi-cloud storage system that provides virtual object and bucket abstractions, seamlessly integrating with S3, GCS, and Azure Blob Storage as storage backends.
(3) We evaluate against state-of-the-art replication policies, showing that SkyStore's policy can substantially reduce cost by up to 6× over SNIA object store traces [30] compared to TTL-CC[24], SPANStore[52], and commercial systems like AWS Multi-Region Replication[17] and JuiceFS[35].

2 SKYSTORE PLACEMENT POLICY

We first provide an overview of SkyStore. SkyStore seeks to minimize dollar cost given a particular *cloud pricing model* (Section 2.1) and *modes of operations* (Section 2.2). To do so, it adopts an *on-demand* approach to object placement and leverages a simple *write-local* policy for data storage, which it combines with a *read-driven* policy for data replication (Section 2.3)

2.1 Cloud Pricing Models

Cloud pricing consists of data storage, network, and operational charges [2]. Most cloud vendors charge storage per GB per month based on the geographic region, provider, and storage class. For example, standard storage in *gcp:southamerica-east1* costs 1.75× more than S3 standard storage in *aws:us-east-1*. The cloud provider also charges network (egress) costs based on the volume of data moved out of a particular cloud region [3]. This can differ by up to 15× within the same cloud and 19× between different clouds. Operations made to the cloud storage service are also charged: this cost is usually much cheaper than storage and network charges, with an average of 0.04 cents per thousand requests. Thus, we will mainly consider storage and network pricing in our discussion.

2.2 Modes of Operations

We explore two modes of object replication and eviction. In the ***Fixed Base (FB)*** mode, each object has a designated primary region where its replica is never evicted. For example, data initially stored in AWS remains there permanently, while additional replicas are added or removed in other cloud regions based on demand. Alternatively, the ***Free Placement (FP)*** mode allows replicas to be placed in any region, with the only requirement being that at least k copies are always maintained (e.g., we explore $k = 1$).

2.3 SkyStore Overview

SkyStore, as a multi-cloud storage system, must fundamentally answer these questions: where to *write* objects, where to *read* objects from, and how to *replicate*. We briefly describe them in turn.

Write Policy. SkyStore adopts a *write-local* strategy. For data storage, SkyStore stores data in the region where the write request originates. This minimizes write latency and reduces egress costs for write operations, ensuring data is immediately available in the local region. In the fixed base mode, we set the base region of the object to the initial local write location. Consecutive write to the objects creates a new object with an updated version in its write location, where versioning is managed by SkyStore control plane (Section 4.2).

Read and Replication Policy. SkyStore adopts a *replicate-on-read* strategy. Upon receiving a read request, SkyStore selects the cheapest region where the replica resides to retrieve the data and creates a local replica to optimize future reads. Replicate-on-read contrasts with proactive replicate-on-write methods used in AWS Multi-Region bucket and SPANStore [17, 52], which pushes all data to a predicted set of regions upon write operations. Unless the prediction is accurate, such a model can lead to high egress costs and storage charges (Section 3.2). SkyStore reactively replicates to reduce future egress costs and performs eviction described in Section 3 to keep storage costs in check.

3 SKYSTORE EVICTION POLICY

In this section, we discuss a cost-minimized cache eviction problem in a two-region base and cache setting (Section 3.1). We then introduce our cost-aware eviction policy (Section 3.2) and show how it extends to multiple regions and clouds (Section 3.3).

3.1 2-Region, Base and Cache Problem

Consider a two-region setup: a base region storing all the objects that never get evicted, and a cache region that reads from the base and replicates on read. We denote S as the storage cost (**$/GB*Month**) in the base region and N as the egress cost (**$/GB**) for moving an object over the network between the base and the cache region[1]. Aggressive replication in the cache region can lead to prohibitively high storage costs, especially as replicas accumulate over time. Thus, we now explore how to cost-effectively evict replicas in the cache region under this simple 2-region setup.

3.1.1 The Clairvoyant Greedy Policy (CGP). We measure ourselves against a cost-optimal policy that is given access to an *oracle* that knows exactly when an object will be read in the future (if at all) in the cache region. This is akin to the Belady cache eviction algorithm [23], but adapted to our problem setup. A key parameter in the clairvoyant strategy is the *break-even time*. This is the duration in which the cost of storing an object equals the cost of evicting it and fetching it again across the network (i.e., the storage cost equals the egress cost). We denote this as T_{even}, where:

$$T_{\text{even}} = N/S \tag{1}$$

For example, we have $\text{Cost}_{\text{storage}}$ = \$0.026 per GB per month for `aws:us-west1` and $\text{Cost}_{\text{egress}}$ = \$0.02 per GB between `aws:us-east1` and `aws:us-west1`. Thus, $T_{\text{even}} \approx 0.77$ months for the edge between these two regions.[2]

Since the eviction of one object is independent of others, it is clear that the best one can do is to cache an object as long as the cost of storage is lower than the cost of bringing it again over the network and vice versa. Every time an object is read, the clairvoyant policy accesses an oracle that returns the time duration until this object will next be read. We denote $T_{\text{next}}(o, i)$ as the time between the i^{th} and $i+1$ reads of object o. The strategy then compares T_{next} with the break-even time T_{even} and decides whether to evict the object. An object with no next GET is immediately evicted. In summary, the clairvoyant policy upon the i^{th} access to object o works as follows:

$$Clairvoyant(o, i) = \begin{cases} \text{evict}: & T_{\text{next}}(o, i) > T_{\text{even}} \\ \text{keep}: & T_{\text{next}}(o, i) \le T_{\text{even}} \end{cases}$$

3.1.2 The T_{even}-policy. A simple policy (T_{even}-policy) will be setting TTL to the break-even time $T_{\text{even}} = \frac{N}{S}$ and refresh upon each access. It has the following properties:

(1) The cost of the T_{even}-policy is at most twice the clairvoyant policy.
(2) $\forall$ eviction policy $\exists$ a workload for which the policy costs twice as much as the clairvoyant policy.

[1]For simplicity, we are ignoring the associated operation costs (e.g., cost for every PUT or GET) that are typically lower than the storage and egress costs.
[2]Prices taken in Sept. 2023.

Proof for (1). The cost per GB for a single object under the optimal clairvoyant policy includes paying network cost N for initial GET, storage cost $T_{\text{next}}(i) \cdot S$ for storing the object until the next access, and network cost N for re-fetching the object after eviction:

$$N + \sum_{i|T_{\text{next}}(i) \le T_{\text{even}}} T_{\text{next}}(i) \cdot S + \sum_{i|T_{\text{next}}(i) > T_{\text{even}}} N$$

The cost of the T_{even}-policy policy is:

$$N + \sum_{i|T_{\text{next}}(i) \le T_{\text{even}}} T_{\text{next}}(i) \cdot S + \sum_{i|T_{\text{next}}(i) > T_{\text{even}}} (\frac{N}{S} \cdot S + N) + \frac{N}{S} \cdot S$$

The first two parts are identical. However, for $T_{\text{next}}(i) > T_{\text{even}}$, T_{even}-policy needs to pay additional storage cost until the break-even point $\frac{N}{S} \cdot S$, evict it, and pay for network cost to re-fetch. The last $\frac{N}{S} \cdot S$ accounts for keeping this object around after its last GET. Thus, T_{even}-policy is bounded by 2× that of the optimal.

Proof for (2). We claim that for any eviction strategy, an adversarial workload exists that costs more than twice that of the optimal strategy. Consider a single object. After its first access, the eviction policy P must decide when to evict. If P decides to evict the object after more than T_{even} time, then the workload never asks for this object again. In such a case, the optimal cost ($\text{cost}_{\text{optimal}}$) is just the initial GET cost, N, while the cost for policy P (cost_{P}) is greater than $N + T_{\text{even}} \cdot S = 2N$, double the optimal.

Alternatively, if policy P evicts the object earlier (at $t < T_{\text{even}}$), the workload issues a new GET shortly after $t+\varepsilon$, where $t+\varepsilon < T_{\text{even}}$. The optimal cost $\text{cost}_{\text{optimal}} = N + (t + \varepsilon) \cdot S$ in this case , while the cost_{P} becomes $2N + t \cdot S$. Since the object is in the cache again, this process can repeat. After k iterations, we have $\text{cost}_{\text{optimal}} = N + (\sum(t_i + \varepsilon))S$ whereas $\text{cost}_{\text{P}} = (k+1)N + (\sum t)S$. Their difference grows as $\text{cost}_{\text{P}} - \text{cost}_{\text{optimal}} = kN + k\varepsilon$. For large enough k, since $\text{cost}_{\text{optimal}} < (k+1)N$, $k\varepsilon > N$ will cause $\text{cost}_{\text{P}} - \text{cost}_{\text{optimal}} > \text{cost}_{\text{optimal}}$ and give us a ratio ≥ 2.

These two properties demonstrate that if nothing is known apriori about the workload, then the T_{even}-policy is the safest strategy one could hope for. However, distributions are not chosen adversarially in reality. Quite a bit can be learned about the distributions of the workload at hand, which should be used to reduce costs.

3.2 SkyStore 2-Region Base & Cache Eviction

Now we discuss how SkyStore policy learns workload distributions over time and aims to set the cost-optimal TTL for objects in the 2-region setup.

3.2.1 TTL-based eviction. Inspired by T_{even}, SkyStore assigns each replica (i.e., a copy of an object) a TTL (Time To Live) value. For the free-placement (FP) mode, if the replica is not accessed within TTL time, it will be evicted as long as it is not the sole remaining replica. In fixed-base (FB) mode like the 2-region setup (Section 3.2), the replica can only be evicted if it is not in the base location.

There are two common TTL-based eviction methods. The first, used in CDNs [22], invalidates a cached object after its TTL expires, regardless of access frequency, to prevent stale data. SkyStore takes a different approach and resets the TTL on each access to reduce network costs and avoid evicting frequently accessed objects. SkyStore eviction policy periodically scans and evicts objects that have

Parameter	Description
$range(j)$	Time interval of the j^{th} cell
$t(j)$	Maximum time in $range(j)$
$\widehat{t}(j)$	Mean time in $range(j)$
$hist(j)$	Bytes re-read after time $t \in range(j)$
$last(j)$	Bytes not read in $t \in range(j)$

Table 1: Eviction parameters in SkyStore.

not been accessed within TTL time. The policy for evicting object o at region R is as follows:

$$Policy(o,R) = \begin{cases} \text{evict}: & \text{time since last access} > \text{TTL}(o,R) \\ & \text{\& not sole copy} \\ \text{keep}: & \text{otherwise} \end{cases}$$

3.2.2 Adaptive TTL. The crux of SkyStore's approach is setting the right TTLs for replicas. The main statistic we use to adapt TTL is the time between accesses of objects, represented as a distribution of T_{next} at the cache region. We build a weighted histogram where each cell corresponds to a time range, and the weight reflects the total size of GETs within T_{next} in that range. This histogram is collected per region per workload. We define relevant notations in Table 1. The value in each histogram cell is denoted as:

$$hist(j) = \sum_{o,i \mid T_{\text{next}}(o,i) \in range(j)} size(o)$$

This histogram accounts for all re-reads in the cache region for the workload. However, it does not account for what happens to objects after their last access. For this, we use an additional histogram called *last* to track the latest access time. Given these histograms and a TTL value, we compute the expected cost for the TTL as:

$$\begin{aligned} ExpectedCost(TTL) = & \sum_{\substack{o \in R \\ \text{Requested}}} Size(o) \cdot \mathbb{I}[\text{Fetched remotely}] \cdot N \\ & + \sum_{\substack{j \in hist \\ t(j) \le \text{TTL}}} hist(j) \cdot \widehat{t}(j) \cdot S \\ & + \sum_{\substack{j \in hist \\ t(j) > \text{TTL}}} hist(j) \cdot (N + \text{TTL} \cdot S) \\ & + \sum_{j \in last} last(j) \cdot \text{TTL} \cdot S \end{aligned}$$

The first term accounts for the initial read cost of all objects requested from the region R, using an identity function over remote reads. If it is a local read, the cost would be 0; if fetched remotely, the cost would be N. The second accounts for hits – objects that are re-read and exist in the region. The third accounts for misses – objects that are evicted and brought into the region with additional network cost, and the last term accounts for storage costs of objects that have not yet been re-read. We iterate over possible TTL values at the same granularity as the histogram and select the one with the lowest expected cost.

The best TTL chosen is influenced by the specific workload and the network and storage costs. Figure 1 shows an example of the expected cost as a function of TTL for an IBM trace with different pricing choices. A lower value of T_{even} indicates that storage costs

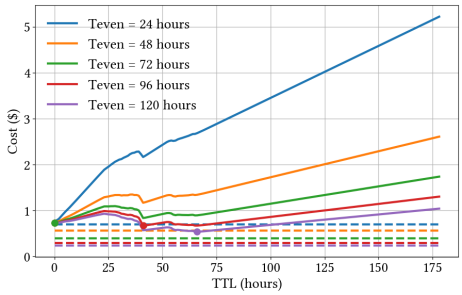

Figure 1: The expected cost as a function of TTL on a trace with an hourly histogram. The dashed line shows the optimal policy cost, and the dot marks the minimum cost point.

are higher (relative to the network costs) and means that shorter TTLs would fare better, as seen in the example.

3.2.3 Granularity of Histogram. In prior discussions, we collected a histogram to study cache region access distribution. In object stores, bucket-level granularity often reflects the workload access patterns of all objects in the bucket of a particular region over time. Object-level statistics, however, can be misleading. For instance, in one IBM trace, there are bursts of 2-8 consecutive GETs to the same object within 10 minutes of each other, followed by no further access to that object. Methods that focus on learning each object's pattern separately [39] or assume Poisson-like distributions [24] fail to capture this bursty behavior. SkyStore generalize bucket-level patterns and assign a TTL that ensures replicas remain available during bursts but are evicted soon after. We show a variant of SkyStore that collects object-level statistics instead of bucket-level and show that it does not work very well in evaluated workloads (Section 6.3). Future work can study how to cluster objects with similar distribution (e.g., folders within a bucket) and collect histogram statistics separately for groups of objects.

The granularity of the histogram is also directly related to our possible choices for setting the TTL. A more granular histogram gives us additional information and allows us to choose TTLs more accurately, achieving better cost savings. On the other hand, a large histogram burdens the memory and computation requirements of the system. Recall that the histogram should potentially cover a time duration of many months, yet at times, the best eviction policy calls for evicting objects within minutes or even seconds. To balance this tradeoff, we support a variable range for histogram cells and attempt to have high granularity for small TTL values and low granularity for larger ones. For the first minute, we use a per-second granularity (taking up 60 cells). Beyond that, we employ a logarithmic base granularity with a low base of 1.02. This ensures that the ratio between two consecutive potential TTL values is no more than 2%. In turn, the difference in storage cost between two consecutive TTLs is also bounded at 2% as the cost is linear in the time the replicas are stored. Using 740 cells at this log granularity covers $(1.02)^{740}$ minutes, which amounts to almost 2 years. An additional 60 cells cover the first minute, and we thus manage to cover nearly 2 years with an 800-cell histogram.

To account for changing workload distributions and application behavior over time, we opt to periodically collect a new histogram

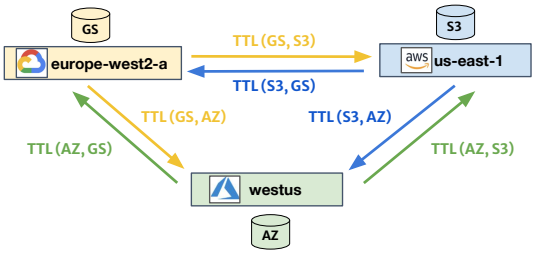

Figure 2: The multi-cloud setting as a directed graph with a TTL assigned per each directed edge.

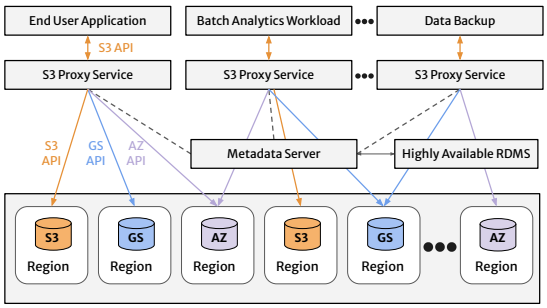

Figure 3: The system architecture of SkyStore.

(while still keeping the previous histogram). Once the new histogram has a sufficiently long history, the old histogram can be discarded. Our investigations indicate that the histogram should be longer than the T_{even} time to be effective.

3.3 SkyStore Multi-Region Eviction

3.3.1 Choosing adaptive TTLs. We tackle the multi-region setting by breaking the problem into a pairwise problem similar to the 2-region setup; then, we set the TTL for each pair of source and target regions. Namely, we view the multi-region setting as a fully directed graph where each node is a region, and for each directed edge, we compute a TTL corresponding to this edge (as shown in Figure 2).

The TTL assigned to an object at a specific region is then deduced from the TTLs of the edges directed at this region. We denote $\text{TTL}(R_i, R_j)$ as the chosen TTL value for the edge from region R_i to region R_j and $\text{TTL}(o, R_j)$ as the TTL assigned to an object o at region R_j. The eviction TTL of an object depends on the relevant regions that hold a replica of the object o. The TTL of the object at each region is then chosen to be the minimal TTL of edges from all such relevant regions. Namely:

$$\text{TTL}(o, R_j) = \min_{i | o \in R_i} \text{TTL}(R_i, R_j)$$

The TTL of an edge is assigned as a function of the incoming network cost, so the cheaper the cost, the lower the TTL. Since we use the cheapest available source in case of a cache miss, this corresponds to the minimal TTL. Our method for calculating an edge's TTL is detailed in Section 3.2: we take the storage cost at the target, the network cost from some source region to the target, and statistics histograms of the workload in the target region as input. This final component is what makes our choice of TTLs adaptive. As time goes by, we learn from the access patterns of the workloads and change the associated TTLs accordingly.

Our approach assigns a local TTL to each object, which is the minimum of all relevant edge TTLs where the source region has replicas. This assumes a remote replica will still exist after the local TTL expires, enabling cost-efficient retrieval. However, since TTLs are set independently, this assumption may not always hold. To ensure correctness, we filter out cases where the local TTL plus storage start time exceeds the remote replica's eviction time, calculated as the replica's start time plus its TTL. This prevents reliance on replicas that may already be evicted.

3.3.2 Latency Considerations. A cost-centric policy could implicitly model performance, as resources often have associated price tags. However, we observe that incorporating latency into a cost-driven framework is particularly challenging since it requires assigning a cost value to read performance, which is specific to users, applications, and objects.

We propose a potential solution to model the price of cache hits and ask how much a customer is willing to pay for cache hits. Namely, if all objects are equally important, how much cost would the user be willing to pay for additional low-latency local read? We denote this value as *user performance value* or $U_{\text{perf-val}}$, in dollar cost per byte. We incorporate this into our methodology of carefully choosing a TTL as follows: After finding the value of TTL that promises the lowest expected cost, we check if there is a higher TTL value for which the $U_{\text{perf-val}}$ bounds the average cost per additional cache hit. More formally, if TTL′ represents the eviction time that achieves the lowest expected cost, we choose the highest TTL value such that

$$\frac{\mathit{ExpectedCost}(\text{TTL}) - \mathit{ExpectedCost}(\text{TTL}')}{\text{object byte count between TTL and TTL}'} \leq U_{\text{perf-val}}$$

In this model, users pay for objects until they are evicted, and their TTLs are reset upon the next access. We plan to compare the effectiveness of this approach to SkyStore's cost-centric policy and estimate its cost and latency tradeoffs in the future.

4 SKYSTORE ARCHITECTURE

Building a cost-efficient multi-cloud object store requires addressing several key challenges. Such an object store must offer (1) a cohesive view of global objects stored across multiple regions and clouds, and (2) consistency across clouds and regions. Currently, consistency is typically guaranteed only within single-region object stores. (3) reliable data recovery in the event of failures, with guarantees comparable to single-region object stores.

SkyStore is designed as an overlay layer on top of existing cloud object storage systems, including AWS S3 [12], Google Cloud Storage [13], and Azure Blob Storage [11]. It consists of a client proxy service and a control plane, as shown in Figure 3. The client proxy fetches objects and supports the AWS S3 wire protocol [1], allowing users to seamlessly port applications using the S3 interface. The control plane, a stateless web server backed by a database, tracks object locations and redirects requests across cloud regions. We elaborate on the design of these components in Sections 4.1-4.3. We then summarize SkyStore's consistency guarantees (§4.4), and fault tolerance (§4.5).

4.1 API: Virtual Object & Bucket Abstraction

In object stores, objects are binary blobs identified by a key within a specific bucket. A bucket, serving as a namespace, is a collection of objects and the unit for placement and permission management. Traditionally, objects and buckets are confined to specific regions and clouds. As such, clients need to know the location and cloud of an object before accessing it. SkyStore abstracts this away with *virtual object* and *virtual bucket* that appear global to the user, with their physical locations managed transparently by SkyStore. This abstraction simplifies interaction with diverse cloud APIs by leveraging common concepts across providers. Users manage and access objects as if they were local, while SkyStore efficiently handles the routing and storage of these objects across regions and clouds.

4.2 Control Plane: SkyStore Metadata Server

The SkyStore metadata server acts as the central coordinator for routing requests across multiple regions and clouds. Importantly, the control plane does not handle actual object data, eschewing any potential bottlenecks. The metadata stored for each virtual object includes key information such as object size, last modified time, entity tag, and version ID. SkyStore also manages the mapping between virtual objects and their physical locations in each cloud region. A key component of this system is the `Policy` interface, which determines where to store objects on PUTs and where to fetch them on GETs. This interface supports various placement and eviction policies described in Section 6.2.2.

Eviction Process The metadata server collects T_{next} statistics into histograms to assist with SkyStore decision-making. A background process runs periodically (once per day) to scan for objects exceeding their TTLs and initiates `DELETE` requests in the respective cloud object stores. This process is computationally lightweight since it only involves handling metadata, with the actual deletion handled by the cloud providers, so no data transfer occurs. In practice, this method incurs minimal overhead, as shown in Section 6.6. Alternatively, configuring lifecycle policies [20] for objects in each bucket could remove the need for SkyStore to track TTLs, although these policies are typically limited to 1000 rules per bucket.

4.3 Data Plane: S3-Proxy

The data plane handles user requests by interfacing with physical object stores through a *S3-Proxy*. Requests are processed according to AWS S3 protocols [1], which we choose to implement due to its widespread popularity and market share [9][3]. The workflow of a single request proceeds as follows. First, a request is issued by client and received from the client proxy. Then, the proxy contacts the metadata server to determine the appropriate location for processing the request. Lastly, once the metadata server provides necessary information, the proxy interacts with the designated storage providers to issue the request and perform any required read or write operations. The stateless design of S3-Proxy ensures horizontal scalability.

4.4 Consistency

SkyStore employs a centralized metadata server to maintain global knowledge of data versions, enabling read-after-write and eventual consistency as supported by many existing cloud providers [21, 40, 46]. This approach simplifies consistency management while supporting the most generic consistency guarantees but incurs overheads that will be discussed in Section 6.7.

Read-After-Write Consistency ensures that after a write (PUT), the latest version of an object is immediately available for reads (GET). This model is critical for applications requiring fresh data, such as e-commerce systems [43]. As SkyStore tracks versions and locations of each logical object write with a centralized metadata server, read-after-write consistency can be trivially achieved by retrieving the last write location from the database to satisfy any read requests.

Eventual Consistency allows faster, local reads by serving possibly stale data. This approach minimizes network overheads and improves access speed when real-time consistency is not required, such as backup systems and non-critical analytics [45]. In SkyStore, eventual consistency is also achieved through the centralized server, which provides information about the closest region containing replicas of a logical object, even if those replicas are stale.

4.5 Fault Tolerance

SkyStore fault tolerance consists of two components: data fault tolerance and metadata fault tolerance.

Data Fault Tolerance For a single region, SkyStore ensures the same data fault tolerance guarantees as a typical cloud object store, assuming the bucket is fault-tolerant in this region. For region failures or provider outages, users can specify manual replication regions via SkyStore replication API for periodic backup.

Metadata Fault Tolerance SkyStore metadata server maintains object metadata and mappings of logical objects to their physical locations in the cloud object stores. To handle data plane failures (e.g., S3-proxy or network disruptions), SkyStore employs a two-phase commit protocol across the data plane and control plane to prevent metadata and object data corruption. The protocol logs intended actions and commits only after successful object writes, rolling back changes in case of errors.

For control plane (metadata) failures, SkyStore never loses the data. It maintains metadata fault tolerance (to discover the data) via a checkpoint and rollback mechanism. Metadata is periodically checkpointed to an object store bucket. The checkpoint location can be configured to a local region for server-level fault tolerance or a remote region for regional fault tolerance. If a metadata server fails, recovery involves re-provisioning the metadata server (which typically takes 5 minutes) and restoring metadata from the latest checkpoint. Any metadata lost between checkpoints is recovered by scanning objects in the bucket. Efficient recovery is achieved using prefix search on epoch-prefixed object names, which is faster than a full scan and is widely supported by storage providers. Alternatively, services like S3 Inventory [25] provide cost-effective daily metadata reports. We discuss the overheads of this approach in Section 6.7.

5 IMPLEMENTATION

We prototype SkyStore as described in Section 4 to compare the end-to-end latency of SkyStore against other policies. SkyStore's

[3]We support 14 common object store operations, including `create`, `delete`, `list` of buckets, and `head`, `get`, `put`, `delete(s)`, `list`, `copy`, and `multipart-upload` related operations. In our experience, this is sufficient to support almost all cloud workloads.

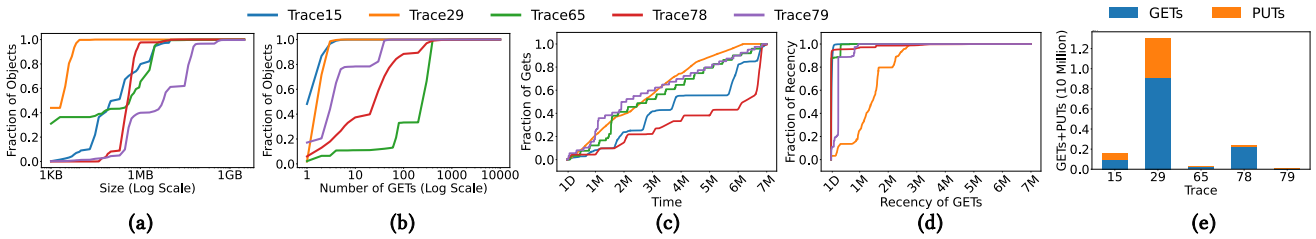

Figure 4: Trace Analysis: We showcase object sizes (a), access frequency (b), burstiness or the fraction of GETs over time (c), the recency of GETs (d), and the PUT to GET ratio (d), for five representative IBM traces. Days: (D) and months: (M).

metadata server is implemented in 3.5K lines of Python code to support various policies. It stores the metadata in a Postgres database by default [5] and can be configured to support an SQLite backend [47]. The S3-proxy is implemented in about 9k lines of Rust code, connecting to AWS S3 [12], Google Cloud Storage [32], and Azure Blob Storage [11]. In our experiments, we host the metadata server on `m5d.8xlarge` instance in `aws:us-east-1`. It contains 32 vCPUs, 128 GiB of memory, and 2 x 600 GB NVMe SSDs. We instantiate a S3-proxy on each client VM that uses `m5.8xlarge`, `Standard_D32ps_v5`, and `n2standard32` instance types on AWS, Azure, and GCP, respectively. These client VMs contain 32 vCPUs, 128 GiB of memory, and 64 GB of storage. We also implement all policies outlined in Section 6.2.2 in 1.9k lines of Python code to estimate the total cost across traces. Our simulations are run on a standard VM like `n4-standard-4` with 4 vCPUs, 16GB memory, and 32GB of storage.

6 EVALUATION

In our evaluation, we answer the following questions:

(1) What are the cost benefits of SkyStore's replication policy across two regions within a single cloud?
(2) Do the cost savings from SkyStore's policy scale to multiple regions across multiple clouds?
(3) What are the end-to-end latency and cost savings of SkyStore in a real multi-cloud deployment?

6.1 Workloads: Multi-Region and Multi-Cloud

We describe the object store traces we use (Section 6.1.1), outline their diverse characteristics (Section 6.1.2), and discuss our methodology to carefully generate multi-cloud workloads from these traces. This step is necessary as there are no publicly available multi-cloud traces to the best of our knowledge.

6.1.1 Workload generation from traces. Our workloads are drawn from the SNIA IBM Object Store traces [10]. These traces record a week of RESTful operations (e.g., GET, PUT, HEAD, DELETE) for a single region within the IBM cloud [31]. These traces effectively capture diversity across various dimensions: object sizes, recency, and frequency of accesses (as detailed in Section 6.1.2). However, object stores are typically designed for long-term data retention where objects are stored for several months to years [18]. Since these short, week-long traces inadequately capture the life of objects in the cloud, we expand a day in each trace to a month for single cloud experiments and to three months for multi-cloud settings without changing their inherent characteristics like read-to-write ratio or request distributions.

We pick five representative traces with salient characteristics in recency, frequency, size, burstiness, and PUT and GET distributions. We outline their characteristics and the key insights that inform the generation of multi-region and multi-cloud workloads. In the interest of space, we use multi-region, multi-cloud workloads generated from these representative traces for all of our experiments.

6.1.2 Trace characteristics. Cloud applications have unique access patterns across dimensions like object sizes, PUT to GET ratios, access frequency, recency, and burstiness, as shown in Figure 4. We summarize these characteristics in Table 2.

- *Object sizes*: We categorize objects in four size ranges: tiny (<1KB), small (1KB to 1MB), medium (1MB to 1GB), and large (>1GB). As seen in Figure-4a, most of the objects accessed are small or medium in size, some are tiny, and very few are large. Most traces have <0.5% of tiny objects except two traces (T29 and T65) with 30–45% of tiny objects. All traces have >35% of small objects and notably, three traces (T15, T29 and T78) have a majority (56–97%) of small objects. About 34% and >60% of objects in T65 and T79 are medium-sized, while less than 20% of the other traces have medium-sized objects. None of the traces have large objects except T65 and T79, which rarely have large objects (<0.4%).
- *Access frequency and one-hit wonders*: We categorize objects accessed in our traces as one-hits (1 GET), cold (1-10 GETs), warm (10-100 GETs), hot (100-1000 GETs), and super hot (>1K GETs). As seen in Figure 4b, our traces significantly differ in their distribution of repeated reads. Two traces (T15 and T29) are almost entirely composed of objects that are one-hits (98% and 2% respectively) or cold (52% and 98% respectively). In contrast, T78 has a majority (>51%) of warm objects and T65 has a majority (>67%) of hot objects. None except two traces (T65 and T78) have super hot objects in very small proportions (<0.1%).
- *Burstiness*: We define burstiness as the fraction of GETs over time. As seen in Figure 4c, our traces have distinct burst patterns over time with different spikes. While one trace (T15) has an even distribution of accesses and has no accesses in the last two months, another trace (T78) has a burst with 60% of GETs within the last two months. In the rest of the traces, about 50% of accesses arrive in the last two months. Three traces (T29, T65, and T79) nearly have an equal distribution of GETs, with a noticeable spike, where 30% of objects are accessed in short time intervals.
- *Recency of accesses*: Our traces also have varying recency, i.e., the time interval between consecutive GETs, as shown in Figure-4d. Two traces (T15 and T78) have inter-arrival times within a day. In contrast, T65 and T79 show about 10% of GET intervals

falling between one day and one month. In T29, >80% objects are read between one day and up to two months, and the remaining intervals even exceed two months.
- *Ratio of GET and PUT operations*: Our traces also capture read and write dominant workload patterns. Three traces (T65, T78, and T79) are read-heavy, as shown in the Figure 4e. The rest (T15 and T29) are write-heavy with 42% and 30% of PUTs. Note that T29 has >12M requests in total.

6.1.3 Multi-region, Multi-cloud workload generation. To address the lack of multi-cloud workloads, we use our five traces to synthetically generate such workloads in three steps.

Step 1: From one to two regions within a cloud. We first explore the single-cloud, two-region base and cache setup (as described in Section 3.1). This setup represents a popular approach [41] where data is already located in one region, but the computation is run elsewhere, for instance, due to low resource availability (e.g., geo-distributed model serving service on GPUs). Recall that our traces are from a single region within the IBM cloud. To support this setup, we generate a workload in which PUT operations are directed to the base region and GET operations to the cache region.

Step 2: To multiple regions and clouds. Next, we generate multi-region and multi-cloud workloads. Our synthesis of multi-cloud workloads is informed by our conversations with industry experts and their observations, which highlight the following patterns:

(1) *Uniform Workloads (Type A)*: Applications like networks of IoT sensors [42] and e-commerce websites [33] have uniformly random access patterns. For this workload, we distribute PUTs and GETs randomly across regions and clouds.
(2) *Region-Aware Workloads (Type B)*: Applications like satellite image analysis [7], disaster recovery [29], and cloudburst [27], ingest data in one but consume the data from another region. For this workload, we assign unique PUT and GET regions for each object and distribute requests accordingly.
(3) *Aggregation Workloads (Type C)*: Applications that collect data (like regional sales information, logs, etc.) at different regions [28] but access or analyze this data from a central region. For this workload, we distribute PUTs across regions, allowing data ingestion across regions, and dedicate GETs to a single region.
(4) *Replication Workloads (Type D)*: Applications like CDN [38], container registry [26], and geo-distributed model serving [16] typically write to a single region and read from multiple other regions. For this workload, we assign a dedicated PUT region for each object and distribute GETs across other regions.

Step 3: Multi-cloud workloads. We combine our multi-cloud workloads into a single workload (Type E) for a single trace (T65) for our end-to-end experiments with real cloud deployments. This is necessary as each workload above stores and accesses 6.7 TB of data on average, and it would cost about 0.2M dollars to evaluate SkyStore against all workloads and setups in the cloud.

6.2 Deployment Settings and Baselines

We evaluate SkyStore's policy in two deployment settings (Section 6.2.1) and compare against several baselines (Section 6.2.2).

6.2.1 Deployment settings and metrics. Policies can operate in one of two modes: fixed base (FB) and free placement (FP). SkyStore assumes FB mode by default, where each object has a fixed, non-evictable base region. In FP mode, any replicas can be evicted, but at least one always remains. Since our closest related work, SPANStore operates only in FP mode, SkyStore supports both modes and compares against SPANStore in FP mode. We assume read-after-write consistency with version enabled, where each read accesses the latest data version.

Multi-cloud deployment settings. Our multi-cloud deployments span across AWS S3, Azure Blob Storage, and GCS clouds. We run 3-region[4], 6-region[5], and 9-region[6] experiments where we select 1, 2, and 3 regions from each cloud provider, respectively.

Metrics. We compare SkyStore against other baselines on cost and latency metrics. We measure the total monetary cost of running a workload based on the standard storage offerings and bi-directional network costs between cloud regions. We also measure the average, p90, and p99 latency for GET and PUT requests.

6.2.2 Baselines. We compare against the following baselines:

- **Always Store / Always Evict** policy always replicates objects to regions where GET is initiated and never evicts, or stores each object in a single storage location and never replicates.
- **TTL-based Eviction** policies include (a) TTL = T_{even} (Section 1), (b) TTL-CC [24], a dynamic policy that stochastically sets TTL based on the cached object's behavior. (c) TTL-CC-Obj, a variant of TTL-CC to update fine-grained TTL based on per-object hits.
- **Clairvoyant Greedy Policy (CGP)** (Section 3.1.1) is an oracle that decides to store or evict given future access times of each object. CGP is cost-optimal in the two-region setup.
- **EWMA** uses an Exponentially Weighted Moving Average [39] to predict the next access time per object and chooses whether to evict it accordingly. We set the decay factor α to be 0.5.
- **SPANStore** is a multi-cloud replication policy [52] that replicates objects each hour to minimize access costs. SPANStore does not fix a storage location and hence, we evaluate it only in the free placement (FP) mode.
- **Industrial Baselines** include AWS Multi-Region Buckets [17] (and similarly, GCP Multi-Region Bucket [32]) and JuiceFS [35]. Upon PUT, an object is asynchronously replicated to the pre-configured secondary region(s). We evaluate AWS in a two-region setup and JuiceFS in a multi-cloud setup, assuming the object is replicated to all other regions.
- **SS-Obj** includes a simple SkyStore policy variant that updates TTL using per-object histogram statistics..

6.3 Single-Cloud Two Regions: Base and Cache

We now evaluate SkyStore in a two-region base and cache setup and showcase its merit as a standalone caching policy. SkyStore consistently maintains low costs across traces, while its alternatives have low or comparable costs in specific cases and become prohibitively more expensive in others. On average, SkyStore has **1.4–20× lower costs** compared to six baselines (Section-6.2.2) across five traces, as shown in Figure 5.

[4] `aws:us-east-1`, `azure:eastus`, `gcp:us-east1-b`
[5] `aws:us-east-1`, `aws:us-west-2`, `azure:eastus`, `azure:westus`, `gcp:us-east1-b`, `gcp:us-west1-a`
[6] `aws:us-east-1`, `aws:us-west-2`, `aws:eu-west-1`, `azure:eastus`, `azure:westus`, `azure:wasteurope`, `gcp:us-east1-b`, `gcp:us-west1-a`, `gcp:europe-west1-b`

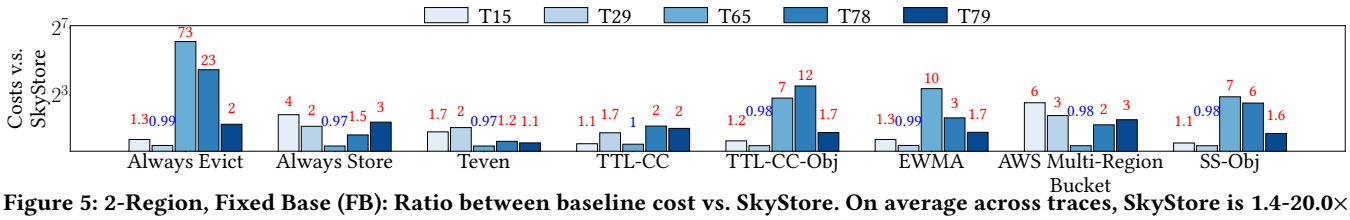

Figure 5: 2-Region, Fixed Base (FB): Ratio between baseline cost vs. SkyStore. On average across traces, SkyStore is 1.4-20.0× cheaper than other baselines.

IBM Trace Number	Size (%)					Read Frequency (%)						Request Arrival			Recency	Number of Requests		
	# Tiny (<1kB)	# Small (1KB-1MB)	# Medium (1MB-1GB)	# Large (>1GB)	Avg. (KB)	One-Hit Wonders	Cold (1–10)	Warm (10–100)	Hot (100-1K)	Super Hot (>1K)	Avg. #GETs	%in first 3 months	%in last 4 months	Avg. GET Tail months	Avg. days	GET (%)	PUT (%)	Total (M)
T15	0	80	20	0	628	**48**	52	0	0	0	3	42	58	2.3	0.6	57	**43**	1.6
T29	**44**	56	0	0	3	2	**98**	0	0	0	3	57	43	3.5	**41.6**	70	30	**13**
T65	31	34	34	0.03	1,536	2	9	22	**67**	**0.1**	**93**	52	48	3	1.3	**99**	1	0.3
T78	0	**98**	2	0	578	6	31	**51**	11	**0.1**	26	22	**78**	0.8	2.6	95	5	2.4
T79	0	40	**60**	**0.35**	**48,386**	17	61	22	0	0	9	**60**	40	**4.1**	8.3	89	11	0.1

Table 2: IBM Trace Characteristics: each trace with characteristics highlighted with bold and underscore.

AlwaysEvict is effective on one-hit-dominant traces, as it avoids unnecessary storage costs for objects never accessed again. For instance, in T15 where 48% of objects are accessed only once (Table 2), AlwaysEvict incus only 30% higher costs than SkyStore. It results in slightly higher network costs for the remaining cold objects accessed more than once in this trace. In traces like T29, where there is longer average recency between GETs (beyond T_{even}), the cost of storing data outweighs the cost of fetching it again on the subsequent access. In such cases, AlwaysEvict can even outperform SkyStore by 1% , as SkyStore reactively caches objects and requires time to adjust to a lower TTL. On the other hand, on traces like T65 and T78 with warm and hot accesses and shorter access recency, AlwaysEvict incurs 23–73× higher costs due to repeated network transfers on reads. Surprisingly, in T79 where 89% of objects are one-hits or cold, AlwaysEvict still costs 2× more than SkyStore. This is primarily due to the large average object size (48MB), which amplifies the network cost penalties from cache misses. On average, SkyStore is 20× cheaper than AlwaysEvict.

Always Store replicates objects on GETs and exhibits behavior that almost contrasts with AlwaysEvict. On traces with lots of hot objects such as T65, AlwaysStore outperforms SkyStore by 3%, as SkyStore may evict a few hot objects and incur higher network costs during the initial histogram warmup phase. However, on traces with more infrequent or sporadic access patterns like T29 and T15, AlwaysStore incurs 2–4× higher costs than SkyStore. Interestingly, on traces with frequent repeated reads, such as T79 where 80% of the objects are accessed multiple times, Always Store remains 3× more expensive than SkyStore. This is because 60% of the objects in T79 have a GET tail longer than 4.1 months (as seen in Table 2), which causes AlwaysStore to retain objects long after their last access. SkyStore evicts unaccessed objects earlier and outperforms AlwaysStore by 1.5–3× on T78 and T79. On average, SkyStore is 2.2× cheaper than Always Store.

T_{even} is a static TTL-based policy (Section 3.1.2) that stores objects until re-fetching them becomes less expensive and balances storage with network costs. In our setup, the TTL for T_{even} policy is one month, calculated as the ratio between average network cost and standard S3 prices across 22 AWS regions. T_{even} performs well when all GETs occur within a month (as in T65), enabling timely evictions and slightly outperforming SkyStore in this case. However, for infrequent accesses like in T15 and T29, it stores objects for a full one-month TTL, leading to 1.7-2× higher costs compared to SkyStore. In traces with moderate access frequency and short recency (like T15 and T29), T_{even} strikes a reasonable balance. However, SkyStore still outperforms it by 1.4× as, unlike T_{even}, SkyStore is aware of object access patterns and can reduce network costs with its adaptive TTL. On average, SkyStore is 1.4× more cost-effective than T_{even}.

TTL-CC policy[24] computes TTLs stochastically based on cache hits, assuming a Poisson distribution, and dynamically updates the TTL of all objects. This policy's cost is within 10% of the total cost of SkyStore for traces with hot- or one-hit-dominant objects (like T15 and T65). However, for mixed traces with warm and cold objects like T78 and T79, TTL-CC has 2× higher cost than SkyStore. TTL-CC also tends to store sporadically accessed objects for longer in T29 and incurs 1.7× higher cost. In summary, TTL-CC results emphasize that dynamic TTL-based policies are a better fit for cloud applications. However, access patterns in the cloud are more complex than Poisson distributions. Overall, SkyStore is more cost-efficient than TTL-CC by 1.6× on average.

TTL-CC-Obj policy is a variance of TTL-CC that computes TTLs per object based on individual object hits. While this approach can quickly adjust TTLs for individual objects that are unlikely to be accessed again, it lacks sufficient hit/miss data to accurately fit the access distribution and adjust TTLs effectively for other cases. It behaves like Always Evict in T15 and T29. In T65 and T78, it misjudges hit patterns, evicting objects too early and performing worse than TTL-CC (Always Store). Overall, TTL-CC-Obj average cost is 4.5× higher than SkyStore policy.

EWMA predicts object access times using exponentially weighted moving averages and stores objects with shorter access times. This policy can quickly evict one-hits and cold objects and reduces storage costs by 0.99–1.3× for traces T29 and T15, respectively. However, it carries this aggressive eviction strategy over to traces with hot and super-hot objects (T65, T78, and T79) and incurs 1.7–10× higher costs. Fine-tuning EWMA policy parameters, such as the

decay factor, could potentially reduce these overheads. On average, EWMA is 3.5× more expensive than SkyStore.

AWS Multi-Region Bucket [17] and similar commercially-available services behave like AlwaysStore but proactively replicate data on writes rather than reads. This leads to higher storage costs when GET appears later; in traces T15, T29, and T78, on average, objects are accessed 1–1.5 months after they are written, incurring 2–6× higher cost than SkyStore. For traces with more immediate reads (like T65 and T79), AWS multi-region buckets incur 0.98–3× higher costs than SkyStore. Overall, AWS Multi-Region Bucket is 3.1× more expensive than SkyStore on average.

SkyStore per-object histogram (SS-Obj). We also explore SkyStore with per-object statistics. This policy often results in frequent evictions, similar to EWMA, because a single object statistics might be mis-leading compared to bucket-level from each region. On average, SS-Obj is 3.3× more expensive than SkyStore.

Optimal CGP. We also compare SkyStore to CGP, an oracle with optimal cost policy (Table 3). On average, SkyStore operates within 15% of optimal, while others incur 1.6–22× higher costs. Its higher cost stems from bucket-granularity statistics, leading to 1.2–1.3× worse performance on traces (T78, T79) with mixed access patterns. However, T_{even} remains empirically within 2× of optimal cost, as proven in Section 3.1.2.

Policy	Cost vs. Optimal					
	T15	T29	T65	T78	T79	Avg
Always Evict	1.4	1.0	77.5	27.8	3.1	22.15
Always Store	3.9	2.3	1.0	1.9	3.4	2.49
Teven	1.9	2.2	1.0	1.4	1.5	1.59
TTL-CC	1.2	1.7	1.1	2.7	2.7	1.87
TTL-CC-obj	1.5	1.0	7.5	7.2	2.2	3.88
EWMA	1.4	1.0	11.0	3.8	2.3	3.90
AWS Multi-Region Bucket	6.3	3.5	1.0	2.8	3.8	3.49
SS-Obj	1.2	1.0	7.9	7.0	2.1	3.84
SkyStore	1.1	1.0	1.1	1.2	1.3	1.14

Table 3: Two-Region Base and Cache: Cost vs. Optimal across individual traces and their average.

6.4 Multi-Cloud: 3 Regions across 3 Clouds

We extend our evaluation to a multi-cloud setup with three regions across three clouds (Section 6.2.1). We use four workloads, i.e., uniform, region-aware, aggregation, and replication workloads (Section 6.1.3). Across these workloads, SkyStore consistently achieves lower costs compared to other baselines by **1.3–18.4**× on average.

Policy	Type A (Uniform)	Type B (Region)	Type C (Aggregation)	Type D (Replication)	Average
Always Evict	9.3	29.8	24.0	10.4	18.4×
Always Store	1.8	1.7	1.7	1.9	1.8×
T_{even}	1.3	1.3	1.3	1.3	1.3×
TTL-CC	1.7	1.2	1.3	1.8	1.5×
EWMA	2.9	4.9	4.4	3.0	3.8×
JuiceFS	4.8	1.9	1.9	4.8	5.7×

Table 4: 3-Region Fixed Base: baseline cost over SkyStore (×), averaged across traces and workload types. On average, SkyStore is 1.3 to 18.4× cheaper than six other baselines.

Table 4 summarizes the baseline's cost over SkyStore's, and averages it across traces and workload types. We compare SkyStore with JuiceFS instead of AWS Multi-Region Bucket as the latter does not operate across clouds. All policies in this experiment are run in the fixed base (FB) mode.

At a high level, 3-region multi-cloud results largely mirror the 2-region setup. Major trends across traces remain the same, but the absolute cost improvements differ from the two-region setup across each workload. This is primarily due to higher (1.8×) network fees in multi-cloud compared to a single-cloud setup. We highlight and explain outlier trends in costs for SkyStore and other baselines.

AlwaysEvict is 9.3–29.8× more expensive than SkyStore on average. AlwaysEvict has higher costs (29.8× and 24.0×) in region-aware and aggregation workloads; for read-heavy traces like T65, it can be 120× worse due to high cross-cloud network fees. AlwaysEvict performs slightly better (9.3× and 10.4×) than SkyStore on uniform and replication workloads. As GETs per object are distributed, policies have uniformly high network costs of cold misses in each region, narrowing their performance gap.

EWMA has similar costs to AlwaysEvict except that it retains objects for slightly longer in read-heavy traces, costing 2.9 – 4.9× more than SkyStore on average.

AlwaysStore and *JuiceFS* are 1.7–1.9× and 1.9–4.8× more expensive than SkyStore on average, respectively. Decreasing access frequency and increasing recency reduces caching benefits. Thus, AlwaysStore incurs low costs (1.7× of SkyStore) in region-aware and aggregation workloads and higher costs (1.8 – 1.9×) with uniform and replication workloads. Surprisingly, AlwaysStore beats SkyStore by 4-6% on read-heavy traces (T65) as SkyStore incurs higher network costs during initial metadata warmup periods. In contrast, JuiceFS has 2.7× higher costs than AlwaysStore on average because JuiceFS proactively replicates objects to all regions on PUTs and incurs high costs for infrequently read objects. However, if read locations are predictable, such as region-aware and aggregation workloads, JuiceFS is auto-configured to replicate to specific regions and incurs similar costs as AlwaysStore.

TTL-CC costs 1.2–1.8× more than SkyStore on average, adjusting TTLs based on cache hit rates. It has lower costs (1.2–1.3× vs. SkyStore) for region-aware and aggregation workloads, but higher costs (1.7–1.8×) for uniform or replication workloads. As an exception occurs in trace T79, where TTL-CC incurs 1.6× lower costs for uniform and replication workloads due to a high fraction (61%) of cold objects, leading to shorter TTLs. However, it struggles when reads are concentrated in specific locations, evicting warmer objects and incurs higher network costs.

The T_{even} policy incurs low costs consistently, i.e., 1.3× higher than SkyStore on average. It balances storage and network costs in multi-cloud setups and maintains low costs across different request distributions. T_{even} has up to 1.7× higher costs than SkyStore as it stores objects even if they are not read in the future.

6.5 Scalability: To 9 Regions across 3 Clouds

We now evaluate how SkyStore's cost savings scale in the multi-cloud setup with an increasing number of regions. Across three clouds, we compare cost savings of SkyStore in three, six, and nine regions relative to the other baselines. Note that we evaluate

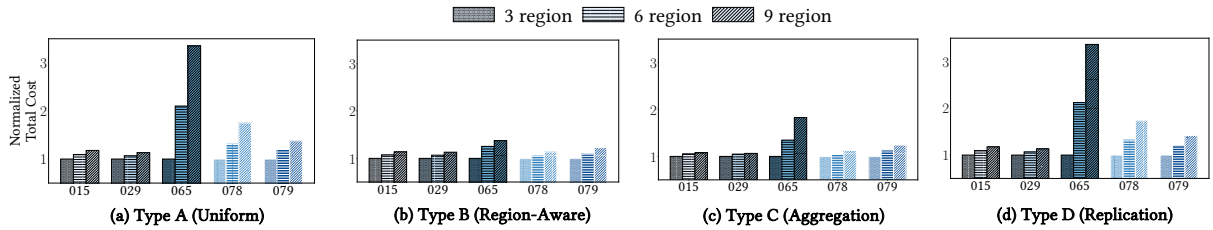

Figure 6: SkyStore Total Cost Normalized over 3-Region on 3, 6, and 9 Regions across Workloads A-D: SkyStore costs remain similar when scaling to more regions.

Policy	3-Region (FB)		6-Region (FB)		9-Region (FB)	
	Avg	Std Dev	Avg	Std Dev	Avg	Std Dev
Always Evict	18.4	31.6	15.0	27.0	12.9	23.9
Always Store	1.8	0.7	2.0	0.8	2.1	0.8
Teven	1.3	0.3	1.4	0.3	1.4	0.3
TTL-CC	1.5	1.0	1.3	0.5	1.4	0.7
EWMA	3.8	4.7	3.2	3.4	3.1	3.4
JuiceFS	3.4	2.9	7.3	7.2	8.6	10.9
Policy	**3-Region (FP)**		**6-Region (FP)**		**9-Region (FP)**	
	Avg	**Std Dev**	**Avg**	**Std Dev**	**Avg**	**Std Dev**
Always Evict	11.9	21.1	11.4	22.4	11.4	23.4
Always Store	1.3	0.2	1.4	0.2	1.5	0.2
JuiceFS	1.7	0.6	3.0	1.8	3.5	2.8
SPANStore	1.4	0.2	1.5	0.2	1.6	0.3

Table 5: 3, 6, 9-Region, 5 traces, Type A-D, Fixed Base (FB) and Free Placement (FP): Average and standard deviation of cost of baselines over SkyStore.

SPANStore only in FP mode as it does not support FB mode. Across 9 regions, SkyStore is **1.4–12.9**× and **1.5–11.4**× more cost-efficient than other baselines in FB and FP modes, respectively.

Table 5 summarizes how scaling affects baseline costs relative to SkyStore, so lower cost relative to SkyStore showcases better scalability. SkyStore remains consistent and incurs low costs when scaling regions. AlwaysEvict and EWMA (in FB mode) incur lower costs on increasing the number of regions from 3 to 9 (18.4 to 12.9×, 3.8 to 3.1×, respectively). On the other hand, AlwaysStore and JuiceFS incur higher costs (1.8 to 2.1×, 3.4 to 8.6×, respectively) compared to SkyStore as regions increase. This is primarily because the number of data replicas is proportional to the number of regions, and these policies incur high storage costs from extensive replication. Recall that JuiceFS proactively replicates data to all regions on PUT requests and pays for higher storage and network costs as regions scale. Both T_{even} and TTL-CC remain fairly consistent (1.3-1.4× and 1.3-1.5×), and show slight fluctuations in relative cost compared to SkyStore when scaling from 3 to 9 regions.

SkyStore and other policies have relatively lower costs in FP relative to FB mode as they incur no additional costs for the base region's storage. SPANStore has comparable costs to AlwaysStore as it does not effectively evict objects that remain unread for long time intervals. SPANStore incurs even higher costs for traces with a majority of one-hits and cold objects (like traces T29 and T79). In our evaluation, SPANStore's solver has access to an oracle with knowledge of workloads and showcases its costs in the best case. Across workloads and traces, SkyStore is 1.4–1.6× more cost efficient than SPANStore on average, with 9 regions across 3 clouds.

SkyStore's cost savings scale from 3 to 9 regions across workload types and traces in FB mode, as seen in Figure-6. On region-aware and aggregation-workloads (Figures 6b, 6c), SkyStore has minimal cost variations with more regions. In these workloads, GETs of objects are concentrated in a single region independent of the number of regions. As an exception, aggregation workloads for trace T65 experience higher costs on scaling to 6 and 9 regions due to a particular cloud region (`aws:us-east-1`), which has higher network ingress costs from all other regions. For uniform and replication workloads (Figure 6a, 6d), SkyStore's cost remains relatively stable as regions increase. This trend is evident in traces with cold objects (T15, T29, and T79), where scaling to 9 regions yields similar costs. However, for traces with warm and hot objects SkyStore's cost increases with the number of regions (like 1.5× and 1.2× for traces T65 and T78) as GETs are distributed across more regions which makes previously warm objects now colder, and increases network costs from evicting such objects.

6.6 Multi-Cloud: End-to-End Benchmark

Policy	GET Latency (ms)			PUT Latency (ms)			GET Lat. vs. AS	Cost ($) vs. AS
	Avg	P90	P99	Avg	P90	P99		
Always Store	172	235	340	840	562	784	1.00×	1.00×
Always Evict	278	440	762	800	507	715	1.61×	76.78×
SkyStore	184	230	408	822	520	782	1.06×	1.05×

Table 6: End-to-End System Evaluation on T65.

We now discuss the end-to-end cost and latency of SkyStore against AlwaysStore and AlwaysEvict baselines for a multi-cloud workload (Type E) on a single trace (T65) due to the prohibitively high cost (2M dollars) of evaluating all workloads and configurations (Section-6.1.3). We run SkyStore and baselines on 3 regions across 3 clouds. As seen in Table 6, SkyStore has comparable average and p99 latency as AlwaysStore, with 3% higher average GET latency due to its metadata overheads from maintaining per-bucket statistics as histograms and periodically updating them in the background. PUT latency remains similar across policies, as writes are local. AlwaysEvict, which avoids caching, incurs 1.6× higher GET latency. SkyStore and AlwaysStore maintain low costs, aligning with our cost simulations (Section 6.4), while AlwaysEvict sees up to **75× higher costs** and increased end-to-end latency.

6.7 Discussions: Overheads & Trade-offs

Operation	Object Size (Single Region)			Object Size (Remote Region)			Object Size (Far Remote Region)		
	128KB	43MB	1GB	128KB	43MB	1GB	128KB	43MB	1GB
PUT	20%	1.6%	0.07%	88%	11.2%	0.4%	2×	21.7%	1.0%
GET	17%	1.0%	0.03%	57%	6.3%	0.2%	1.6×	12.1%	0.5%

Table 7: Evaluation of SkyStore metadata latency overhead vs. AWS APIs, across client-server setups and object sizes.

6.7.1 Cost overheads. The cost of running SkyStore includes the S3-proxy, a client-side library with no extra cost, and the metadata server, hosted on an m5d.8xlarge VM at $1.81 per hour, comparable to typical cloud service operational costs. The cost of fault tolerance in SkyStore is modest. Metadata storage requires approximately 267 bytes per object and 92 bytes per bucket, enabling a single instance with two 600GB NVMe SSDs to store metadata for up to 4.4 billion objects across 10k buckets. Periodically checkpointing for 4.4B object metadata states costs around $27 per checkpoint in storage for a month. At the same time, recovery operations incur $0.01 per 1,000 LIST requests or $11 for listing all 4.4B objects using services like S3 Inventory [25] reports.

6.7.2 System Overheads. Every request in SkyStore to goes through a centralized metadata server. We measure this metadata overhead for PUT and GET operations on 500 objects (128KB, 43MB, to 1GB) across setups where client and metadata servers are in the same region (us-west-1), remote regions (us-west-1 and us-west-2), and far remote region (us-west-1 and us-east-1). We select 43MB as a representative medium size based on the average object size from 98 real-world IBM SNIA object store traces [14].

As shown in Table 7, metadata overhead is minimal for large objects and remains modest for medium ones. In a single-region setup, it accounts for 17-20% of access time for small objects, 1.0-1.6% for medium, and <0.1% for large. Overhead increases in multi-region setups, reaching up to 88% for small objects but staying below 0.4% for large ones. In far-remote region cases, overhead is very high for small objects, but remains under 21.7% and 1.0% for medium and large objects. We note that PUT incurs slightly more latency overhead due to its two-phase commit protocol to ensure fault tolerance (Section 4.5). To minimize latency, particularly for small objects, future work could explore metadata replication with a replicated state machine or local metadata caching for looser consistency, such as eventual consistency.

6.7.3 Overheads with scaling regions and buckets. SkyStore is designed to scale effectively with the increasing number of regions and buckets. Histograms are generated periodically (once or twice a day) for each bucket. The complexity of generating histograms is linear in size. For each bucket, the system calculates point-to-point access patterns. If there are ten regions, this results in $10^2 = 100$ edges per bucket. For 1000 buckets, this scales to 100,000 edges, which becomes manageable with daily or periodic updates.

6.7.4 How does SkyStore incorporate latency considerations? We illustrate latency considerations (Section 3.3.2) with an example. Storing an object costs $0.026/GB per month, with a $0.02/GB egress fee. A TTL of 0.77 months ($T_{\text{even}} = N/S$) caches objects accessed within that period. If a user values faster access at $0.005/GB, extending TTL to 1 month adds $0.006/GB in storage cost – exceeding the benefit, making it unjustified. Reducing TTL to 0.5 months saves more in storage than re-fetching, favoring shorter caching. Users with higher latency tolerance may lower $U_{\text{perf-val}}$, while time-sensitive applications may increase it to retain objects longer.

6.7.5 Real-World Use Cases. In collaboration with IBM, SkyStore has been prototyped to enable seamless multi-cloud storage with Kubernetes environments []. One example workload involves training a model on a Kubernetes cluster in Spain, which interacts with a nearby region object store for model snapshots and training data. Periodically, when the model accuracy is high enough, the model snapshot then needs to be accessed in another region in Venice. There, the model is read and cached automatically to serve clients from a local Kubernetes cluster, helping orchestrate massive evacuations during emergencies. This setup, classified as replication workloads mentioned in Section 6.1.3, highlights SkyStore's ability to transparently cache remote data locally and reduce egress and storage costs in a distributed cloud.

7 RELATED WORK

Geo-distributed Cloud Storage. Existing commercial offerings are mostly single-cloud and require manual placement. AWS [19] and GCP multi-region buckets[32] focus on disaster recovery rather than workload-aware replication. Cloudflare R2 [6] provides a global object store but also requires manual configuration. Volley and Nomad [15, 50] optimize data placement for geo-distributed applications but minimize access latency instead of monetary cost. SPANStore [52] does optimize multi-cloud object placement for cost, but performs proactive replication on writes and does not consider eviction and replication costs. It also requires apriori workload knowledge and cannot react to evolving workload patterns.

Traditional Caching Algorithms. A range of traditional cache eviction algorithms have been developed based on object statistics such as recency, frequency or size (e.g., LRU, LFU, GDSF, FIFO). However, these algorithms consider *cache space* as the primary driver for eviction. Object must thus be *ranked*; objects with the lowest ranking are evicted when the cache becomes full. In contrast, the cache in multi-cloud is not constrained by size but by cost. Each caching decision can thus be made independently for each object.

TTL-based Caching and Cloud Caching. TTL-based approaches have been used for cloud caching, setting a TTL per cache item according to object read frequency. Tokeep et al. [44] keeps an item for T_{even} time and evicts it if it has shown no hits. This is equivalent to the T_{even}-policy we evaluate. Carra et al. [24] offers an approach closer to our two-region approach of a single dynamic TTL for all items in a workload. They use a stochastic approach that modifies the TTL by tracking hits of each new item in the cache. Both prior works assume that object reads occur according to set distributions. In contrast, SkyStore adapts to changing workloads.

8 CONCLUSION

This paper explores the problem of designing a cost-optimized object store across regions and clouds. We propose a TTL-based cost-aware replication policy in the multi-region and multi-cloud setting and build a global object store as an overlay that sits on top of multiple existing cloud services. Our evaluation shows that SkyStore can achieve up to 6× cost savings over state-of-the-art baseline policies and systems in a real cloud setup.

REFERENCES

[1] Actions - Amazon Simple Storage Service — docs.aws.amazon.com. https://docs.aws.amazon.com/AmazonS3/latest/API/API_Operations.html. [Accessed 18-04-2024].

[2] All networking pricing. Virtual Private Cloud. Google Cloud — cloud.google.com. https://cloud.google.com/vpc/network-pricing#standard-pricing. [Accessed 14-04-2024].

[3] AWSâĂŹs Egregious Egress — blog.cloudflare.com. https://blog.cloudflare.com/aws-egregious-egress. [Accessed 14-04-2024].

[4] Ceph Object Gateway Ceph Documentation — docs.ceph.com. https://docs.ceph.com/en/quincy/radosgw/. [Accessed 20-04-2024].

[5] Chapter 27. High Availability, Load Balancing, and Replication — postgresql.org. https://www.postgresql.org/docs/current/high-availability.html. [Accessed 18-04-2024].

[6] Cloudflare R2 | Zero Egress Fee Distributed Object Storage | Cloudflare — cloudflare.com. https://www.cloudflare.com/developer-platform/r2/. [Accessed 14-04-2024].

[7] Landsat data Âă|Âă Cloud Storage Âă|Âă Google Cloud — cloud.google.com. https://cloud.google.com/storage/docs/public-datasets/landsat. [Accessed 14-04-2024].

[8] Pricing - Bandwidth | Microsoft Azure — azure.microsoft.com. https://azure.microsoft.com/en-us/pricing/details/bandwidth/. [Accessed 14-04-2024].

[9] Comparing AWS, Azure, GCP — digitalocean.com. https://www.digitalocean.com/resources/article/comparing-aws-azure-gcp, 2023. [Accessed 20-04-2024].

[10] SNIA: IOTTA repository. http://iotta.snia.org/traces/key-value/36305, 2023.

[11] Azure Blob Storage | Microsoft Azure — azure.microsoft.com. https://azure.microsoft.com/en-us/products/storage/blobs, 2024. [Accessed 20-04-2024].

[12] Cloud Object Storage - Amazon S3 - AWS — aws.amazon.com. https://aws.amazon.com/s3/, 2024. [Accessed 20-04-2024].

[13] Cloud Storage — cloud.google.com. https://cloud.google.com/storage?hl=en, 2024. [Accessed 20-04-2024].

[14] Cloud Storage Services | IBM — ibm.com. https://www.ibm.com/cloud/storage, 2024. [Accessed 20-04-2024].

[15] Sharad Agarwal, John Dunagan, Navendu Jain, Stefan Saroiu, Alec Wolman, and Harbinder Bhogan. Volley: Automated data placement for geo-distributed cloud services. In *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation*, NSDI'10, page 2, USA, 2010. USENIX Association.

[16] Nawras Alkassab, Chin-Tser Huang, and Tania Lorido Botran. Deepref: Deep reinforcement learning for video prefetching in content delivery networks, 2023.

[17] Amazon s3 multi-region access points. https://aws.amazon.com/s3/features/multi-region-access-points/. Accessed on 12/15/2022.

[18] Amazon s3 pricing. https://aws.amazon.com/s3/pricing/. Accessed on 09/29/2024.

[19] Aws cross-region replication. https://docs.aws.amazon.com/AmazonS3/latest/userguide/replication.html. Accessed on 12/15/2022.

[20] aws-lifecycle-policy. https://docs.aws.amazon.com/AmazonS3/latest/userguide/intro-lifecycle-rules.html.

[21] Microsoft Azure. Managing concurrency in Blob storage - Azure Storage — learn.microsoft.com. https://learn.microsoft.com/en-us/azure/storage/blobs/concurrency-manage. [Accessed 19-04-2024].

[22] Soumya Basu, Aditya Sundarrajan, Javad Ghaderi, Sanjay Shakkottai, and Ramesh Sitaraman. Adaptive ttl-based caching for content delivery. *IEEE/ACM transactions on networking*, 26(3):1063–1077, 2018.

[23] Laszlo A. Belady. A study of replacement algorithms for a virtual-storage computer. *IBM Systems journal*, 5(2):78–101, 1966.

[24] Damiano Carra, Giovanni Neglia, and Pietro Michiardi. Ttl-based cloud caches. *IEEE INFOCOM 2019 - IEEE Conference on Computer Communications*, pages 685–693, 2019.

[25] Cataloging and analyzing your data with s3 inventory. https://docs.aws.amazon.com/AmazonS3/latest/userguide/storage-inventory.html. Accessed on 12/15/2022.

[26] Jun Lin Chen, Daniyal Liaqat, Moshe Gabel, and Eyal de Lara. Starlight: Fast container provisioning on the edge and over the WAN. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 35–50, Renton, WA, April 2022. USENIX Association.

[27] Cloud bursting. https://aws.amazon.com/what-is/cloud-bursting/. Accessed on 09/29/2024.

[28] Databricks lakehouse use cases. https://www.databricks.com/blog/2020/01/30/what-is-a-data-lakehouse.html. Accessed on 09/29/2024.

[29] Disaster recovery workloads. https://docs.aws.amazon.com/whitepapers/latest/disaster-recovery-workloads-on-aws/disaster-recovery-options-in-the-cloud.html. Accessed on 09/29/2024.

[30] Ohad Eytan, Danny Harnik, Effi Ofer, Roy Friedman, and Ronen Kat. IBM object store traces (SNIA IOTTA trace set 36305). In Geoff Kuenning, editor, *SNIA IOTTA Trace Repository*. Storage Networking Industry Association, July 2019.

[31] Ohad Eytan, Danny Harnik, Effi Ofer, Roy Friedman, and Ronen I. Kat. It's Time to Revisit LRU vs. FIFO. In *HotStorage 2020*, 2020.

[32] Gcp multi-region bucket. https://cloud.google.com/storage/docs/locations#location-mr. Accessed on 12/15/2022.

[33] Ilija Hristoski and Pece Mitrevski. Evaluation of business-oriented performance metrics in ecommerce using web-based simulation. *Journal of Emerging research and solutions in ICT*, 1(1):1âĂŞ16, April 2016.

[34] Paras Jain, Sam Kumar, Sarah Wooders, Shishir G Patil, Joseph E Gonzalez, and Ion Stoica. Skyplane: Optimizing transfer cost and throughput using cloud-aware overlays. *arXiv preprint arXiv:2210.07259*, 2022.

[35] Juicefs data synchronization. https://juicefs.com/docs/community/guide/sync#distributed-sync. Accessed on 09/29/2024.

[36] llama3 details. https://ai.meta.com/blog/meta-llama-3/.

[37] Inc. MinIO. MinIO | S3 & Kubernetes Native Object Storage for AI — min.io. https://min.io/. [Accessed 14-04-2024].

[38] Leonardo Peroni and Sergey Gorinsky. An end-to-end pipeline perspective on video streaming in best-effort networks: A survey and tutorial, 2024.

[39] Marcus Perry. *The Exponentially Weighted Moving Average*. 06 2010.

[40] Google Cloud Platform. Consistency | Cloud Storage | Google Cloud — cloud.google.com. https://cloud.google.com/storage/docs/consistency. [Accessed 19-04-2024].

[41] Adam Prout. "learnings from snowflake and aurora: Separating storage and compute for transaction and analytics". https://www.singlestore.com/blog/separating-storage-and-compute-for-transaction-and-analytics/, 2021. [Accessed 26-08-2024].

[42] Asmad Bin Abdul Razzaque and Andrea Baiocchi. Analysis of status update in wireless networks with successive interference cancellation, 2024.

[43] s3-consistency-model. https://aws.amazon.com/s3/consistency/. Accessed on 10/01/2024.

[44] Nicolas Le Scouarnec, Christoph Neumann, and Gilles Straub. Cache policies for cloud-based systems: To keep or not to keep. In *2014 IEEE 7th International Conference on Cloud Computing*, pages 1–8, 2014.

[45] Scylladb eventual consistency. https://www.scylladb.com/glossary/eventual-consistency/. Accessed on 10/01/2024.

[46] Amazon Web Services. Amazon S3 | Strong Consistency | Amazon Web Services — aws.amazon.com. https://aws.amazon.com/s3/consistency/. [Accessed 19-04-2024].

[47] sqlite-usecases. https://www.sqlite.org/features.html. Accessed 10-01-2024.

[48] stevenmatthew. Data redundancy - Azure Storage — learn.microsoft.com. https://learn.microsoft.com/en-us/azure/storage/common/storage-redundancy. [Accessed 14-04-2024].

[49] Douglas B. Terry, Vijayan Prabhakaran, Ramakrishna Kotla, Mahesh Balakrishnan, Marcos K. Aguilera, and Hussam Abu-Libdeh. Consistency-based service level agreements for cloud storage. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, page 309âĂŞ324, New York, NY, USA, 2013. Association for Computing Machinery.

[50] Nguyen Tran, Marcos K. Aguilera, and Mahesh Balakrishnan. Online migration for geo-distributed storage systems. In *2011 USENIX Annual Technical Conference (USENIX ATC 11)*, Portland, OR, June 2011. USENIX Association.

[51] Sarah Wooders, Shu Liu, Paras Jain, Xiangxi Mo, Joseph E. Gonzalez, Vincent Liu, and Ion Stoica. Cloudcast: High-Throughput, Cost-Aware overlay multicast in the cloud. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, pages 281–296, Santa Clara, CA, April 2024. USENIX Association.

[52] Zhe Wu, Michael Butkiewicz, Dorian Perkins, Ethan Katz-Bassett, and Harsha V. Madhyastha. Spanstore: Cost-effective geo-replicated storage spanning multiple cloud services. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, page 292âĂŞ308, New York, NY, USA, 2013. Association for Computing Machinery.

[53] Tian Xia, Zhanghao Wu, Ziming Mao, and Zongheng Yang. Introducing SkyServe: 50 https://blog.skypilot.co/introducing-sky-serve/. [Accessed 14-04-2024].

[54] Zongheng Yang, Zhanghao Wu, Michael Luo, Wei-Lin Chiang, Romil Bhardwaj, Woosuk Kwon, Siyuan Zhuang, Frank Sifei Luan, Gautam Mittal, Scott Shenker, and Ion Stoica. SkyPilot: An intercloud broker for sky computing. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 437–455, Boston, MA, April 2023. USENIX Association.