# CosTLO: Cost-Effective Redundancy for Lower Latency Variance on Cloud Storage Services

Zhe Wu*‡, Curtis Yu*, and Harsha V. Madhyastha*‡
*UC Riverside ‡University of Michigan

*Abstract*—**We present CosTLO, a system that reduces the high latency variance associated with cloud storage services by augmenting GET/PUT requests issued by end-hosts with redundant requests, so that the earliest response can be considered. To reduce the cost overhead imposed by redundancy, unlike prior efforts that have used this approach, CosTLO *combines* the use of multiple forms of redundancy. Since this results in a large number of configurations in which CosTLO can issue redundant requests, we conduct a comprehensive measurement study on S3 and Azure to identify the configurations that are viable in practice. Informed by this study, we design CosTLO to satisfy any application's goals for latency variance by 1) estimating the latency variance offered by any particular configuration, 2) efficiently searching through the configuration space to select a cost-effective configuration among the ones that can offer the desired latency variance, and 3) preserving data consistency despite CosTLO's use of redundant requests. We show that, for the median PlanetLab node, CosTLO can halve the latency variance associated with fetching content from Amazon S3, with only a 25% increase in cost.**

## 1 Introduction

Minimizing user-perceived latencies is critical for many applications as even hundreds of milliseconds of additional delay can significantly lower revenue [19, 10, 35]. Large-scale cloud services aid application providers in this regard by enabling them to serve every user from the closest among several geographically distributed data centers. For example, our measurements from over 120 PlanetLab nodes across the globe show that, when every node downloads 1 KB-sized objects from the closest Microsoft Azure data center, the median download latency is less than 100ms for over 90% of the nodes.

However, on today's cloud services, both fetching and storing content are associated with high latency variance. For example, for over 70% of the same 120 nodes considered above, the $99^{th}$ percentile and median download latencies from the closest Azure data center differ by 100ms or more. These high tail latencies are problematic both for popular applications where even 1% of traffic corresponds to a significant volume of requests [23], and for applications where a single request issued by an end-host requires the application to fetch several objects (e.g., web page loads) and user-perceived latency is constrained by the object fetched last. For example, our measurements show that latency variance in S3 more than doubles the *median* page load time for 50% of PlanetLab nodes when fetching a webpage containing 50 objects.

To enable application providers to avail of the cost benefits enabled by cloud services, without having latency variance degrade user experience, we develop CosTLO (Cost-effective Tail Latency Optimizer). Since we observe that the high latency variance is caused predominantly by isolated latency spikes, CosTLO uses the well-known approach [38, 22] for reducing variance by augmenting every GET/PUT request with a set of redundant requests, so that the earliest response can be considered. We tackle three key challenges in using this redundancy-based approach in CosTLO.

First, the end-to-end latency when any end-host uploads to or downloads from a cloud storage service has several components: latency over the Internet, latency over the cloud service's data center network, and latency within the storage service. To tackle the variance in all of these components, CosTLO exploits the fact that redundant requests to cloud storage services can be issued in a variety of ways, each of which impacts a different component of end-to-end latency. For example, while issuing redundant requests to *the same object* may elicit an earlier response due to differences in load across servers hosting replicas of the object, one can further reduce the impact of server load by issuing redundant requests to *a set of objects* which are all copies of the object being accessed. Alternatively, to reduce the impact of spikes in data center network latency, redundant requests can be issued to *different front-ends* of the storage service or relayed to the *same front-end via different virtual machines (VMs)*. Furthermore, when a client is accessing an object stored in a particular data center, redundant requests can be issued to *copies of the object in other data centers* in order to tackle the variance in Internet latencies.

However, not all forms of redundancy have utility in practice due to the complex architectures of cloud services. Therefore, second, we empirically evaluate the ways in which redundant requests should be issued for CosTLO's approach to be viable on Amazon S3 and Microsoft Azure, the two largest cloud storage services today. For example, when issuing concurrent requests to multiple data centers, we find that it is essential to leverage storage services offered by multiple cloud providers; utilizing a single cloud provider's data centers is insufficient to tame the variance in Internet latencies. Our study also shows that, due to load balancing within the data center networks of cloud services, concurrent requests to the same front-end of a storage service are sufficient to tackle spikes in data center network latencies, and more

complex approaches are unnecessary. To the best of our knowledge, this is the first work that identifies the key causes for latency variance in cloud storage services and studies the impact of different forms of redundancy.

Third, the number of configurations in which CosTLO can implement redundancy is unbounded—not only can CosTLO *combine* the use of various forms of redundancy, but it can also *vary* the number of redundant requests, the probability with which it issues redundant requests, etc.—and the impact on cost and latencies varies significantly across configurations. Therefore, for CosTLO to add redundancy in a manner that satisfies an application's goals for latency variance cost-effectively, it becomes essential that CosTLO be able to 1) *estimate*, rather than measure, the cost and latencies associated with any particular configuration, and 2) *search* for a cost-effective configuration, instead of enumerating through all possible configurations. To address these challenges, 1) we model the load balancing and replication within cloud storage services in order to accurately capture the dependencies between concurrent requests, and 2) we develop an efficient algorithm to identify a cost-effective CosTLO configuration that can keep latency variance below a target. Note that no prior work that uses redundant requests seeks to minimize cost.

We have implemented and deployed CosTLO across all data centers in S3 and Azure. To evaluate CosTLO, we use PlanetLab nodes at 120 sites as clients and replay a trace of Wikipedia's workload. Our results show that CosTLO can reduce the spread between $99^{th}$ percentile and median GET latencies by 50% for the median PlanetLab node, with only a 25% increase in cost.

## 2 Characterizing Latency Variance

We begin with a measurement study of Amazon S3 and Microsoft Azure. We 1) quantify the latency variance when using these services, 2) analyze the impact of latency variance on applications, and 3) identify the dominant causes of this variance.

**Overview of measurements.** To analyze client-perceived latencies when downloading from and uploading to cloud storage services, we gather two types of measurements for a week. First, we use 120 PlanetLab nodes across the world as representative end-hosts. Once every 3 seconds, every node uploaded a new object to and downloaded a previously stored object from the S3 and Azure data centers to which the node has the lowest median RTT. Second, from "small instance" VMs in every S3 and every Azure data center, we issued one GET and one PUT per second to the local storage service. In all cases, every GET from a data center was for a 1 KB object selected at random from 1M objects of that size stored at that data center, and every PUT was for a new 1 KB object. To minimize the impact of client-side overheads, we measure GET and PUT latencies on PlanetLab nodes as well as on VMs using timings from tcpdump.

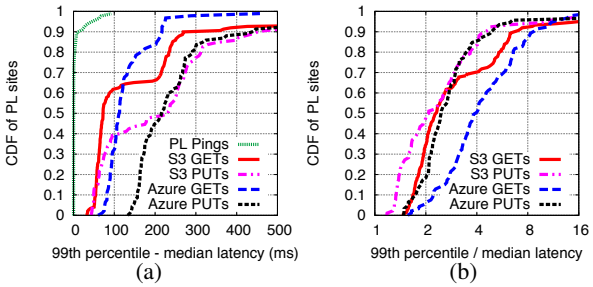
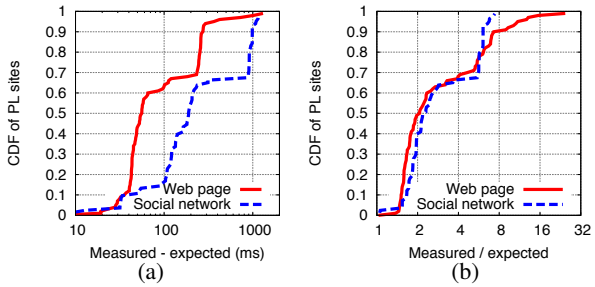



Figure 1: *(a) Absolute and (b) relative inflation in $99^{th}$ percentile latency with respect to median. Logscale x-axis in (b).*

In addition, we leverage logs exported by S3 [7] and Azure [9] to break down end-to-end latency minus DNS resolution time into its two components: 1) latency within the storage service (i.e., duration between when a request was received at one of the storage service's front-ends and when the response left the storage service), and 2) latency over the network (i.e., for the request to travel from the end-host/VM to a front-end of the storage service and for the response to travel back). We extract storage service latency directly from the storage service logs, and we can infer network latency by subtracting storage service latency from end-to-end request latency.

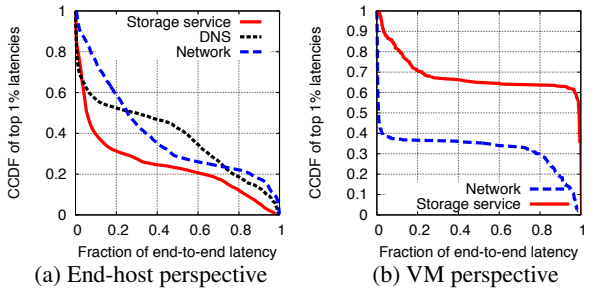
**Quantifying latency variance.** Figure 1 shows the distribution across nodes of the spread in latencies; for every node, we plot the absolute and relative difference between the $99^{th}$ percentile and median latencies. In both Azure and S3, the median PlanetLab node sees an absolute inflation greater than 200ms (70ms) in the $99^{th}$ percentile PUT (GET) latency as compared to the median latency; the median relative inflation is greater than 2x in both PUTs and GETs. To show that this high latency variance is not due to high load or slow access links of PlanetLab nodes, Figure 1 also plots for every node the difference between $99^{th}$ percentile and median latency to the node closest to it among all PlanetLab nodes.

**Impact on applications.** To show that high latency variance can significantly degrade application performance, we conduct measurement studies in two application scenarios. The first one is a webservice that serves static webpages containing 50 objects. The second one is a social network application, where an update from a user triggers a synchronization mechanism to make all of the user's followers fetch that update. In both applications, one user-level request requires the application to issue several requests to cloud storage, and user-perceived latency is constrained by the request that finishes last. We consider a setting in which (1) users only fetch objects from their closest data centers, (2) every user in the social network application has 200 followers [1], and (3) users and their followers have the same closest data centers.



(a) (b)

Figure 2: *(a) Absolute and (b) relative inflation in median user-level request latency with respect to ideal latency. Note logscale on x-axis in both graphs.*



(a) End-host perspective (b) VM perspective

Figure 3: *Breakdown of components of tail latencies.*

We setup clients on PlanetLab nodes and applications on S3, emulate interactions between users and applications using real world traces [6, 30], and measure the page load time/sync completion time.

Ideally, with no latency variance, in the webpage application, page load time should be the same as the latency of fetching a single object if clients fetch all objects on the page in parallel, and in the social network application, the sync completion time should be the same as the latency incurred by the farthest follower to fetch a single object. However, Figure 2 shows that, for over 80% of PlanetLab nodes, latency variance causes at least 50ms latency inflation in the *median* page load time and at least 100ms latency inflation in the *median* sync completion time. This corresponds to a 2x relative inflation for more than 50% of users.

**Causes for tail latencies.** We observe two characteristics that dictate which solutions can potentially reduce the tail of these latency distributions.

First, we find that neither are the top 1% of latency samples clustered together in time nor are they correlated with time of day. Thus, the tail of the latency distribution is dominated by isolated spikes, rather than sustained periods of high latencies. *Therefore, a solution that monitors load and reacts to latency spikes will be ineffective.*

Second, Figure 3(a) shows that all three components of end-to-end latency significantly influence tail latency values. DNS latency, network latency, and latency within the storage service account for over half the end-to-end latency on more than 40%, 25%, and 20% of tail latency samples. Since network latencies as measured from PlanetLab nodes conflate latencies over the Internet and within the cloud service's data center network, we also study the composition of tail latencies as seen in our measurements from VMs to the local storage service. In this case too, Figure 3(b) shows that both components of end-to-end latency—latency within the storage service, and latency over the data center network—contribute significantly to a large fraction of tail latency samples. *Thus, any solution that reduces latency variance will have to address all of these sources of latency spikes.*

## 3 Overview of CosTLO

**Goal.** We design CosTLO to meet any application's service-level objectives (SLOs) for the extent to which it seeks to reduce latency variance for its users. To ensure that CosTLO is broadly applicable across several classes of applications, we consider the most fundamental SLO that applications can build upon—SLOs that bound the variance of the latencies of individual PUT/GET operations; we discuss CosTLO's ability to handle more complex application-specific SLOs in Section 6.

Though there are several ways in which such SLOs can be specified, we do not consider SLOs that bound the absolute value of, say, $99^{th}$ percentile GET/PUT latency; due to the non-uniform geographic distribution of data centers, a single bound on tail latencies for all end-hosts will not help reduce latency variance for end-hosts with proximate data centers. Instead, we focus on SLOs that limit the tail latencies for any end-host *relative* to the latency distribution experienced by *that* end-host. Specifically, we consider SLOs which bound the difference, for any end-host, between $99^{th}$ percentile latency and its baseline median latency (i.e., the median latency that it experiences without CosTLO). Every application specifies such a bound separately for GETs and PUTs.

**Approach.** Since tail latency samples are dominated by isolated spikes, our high-level approach is to augment any GET/PUT request with a set of redundant requests, so that the first response can be considered. Though this is a well-known approach for reducing tail latencies [38, 22, 13], CosTLO is unique in exploiting several ways of issuing redundant requests in combination.

For example, consider downloads from the closest S3 data center at the PlanetLab node in University of Kansas. When this client fetches objects by issuing single GET requests, the difference between the $99^{th}$ percentile and median latencies is 214ms. The simplest way to reduce variance is to have the client issue two concurrent GET requests to download an object (Figure 4(a)). This decreases the gap between $99^{th}$ percentile and baseline median latency to 110ms, but doubles the cost for GET operations and network bandwidth. Alternatively, the client can issue a single GET request to a VM in the cloud, which can in turn issue two concurrent requests

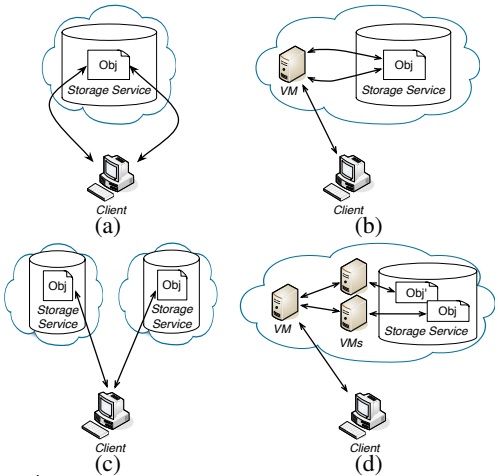



Figure 4: *Illustration of various ways in which CosTLO can concurrently issue requests: (a) to a single object in a storage service, (b) to a single object via a relay VM, (c) to storage services in multiple data centers, or (d) via multiple relay VMs.*

for the requested object to the local storage service (Figure 4(b)). While this adds VM costs and the $99^{th}$ percentile latency is now 135ms higher than the baseline median latency, relaying redundant requests via VMs reduces bandwidth costs (since a single copy of the object leaves the data center). A third option is to have the client concurrently fetch copies of the object from multiple data centers (Figure 4(c)), e.g., the two closest S3 data centers. This strategy—the best of the three options in terms of reducing variance (inflation in $99^{th}$ percentile compared to baseline median drops to 34ms)—eliminates the overhead of VM costs but increases storage costs.

**Challenges.** This example illustrates how various forms of redundancy differ in the tradeoff between reducing variance and increasing cost. Choosing from these various options, so as to satisfy an application's SLO cost-effectively, is challenging for several reasons.

- ***Large configuration space.*** There exist an unbounded number of configurations in which CosTLO can issue redundant requests. This is not only because the degree of parallelism is unbounded, but also because different types of redundancy can be combined with each other. For example, Figure 4(d) shows a configuration that both 1) uses multiple relay VMs to route around latency spikes in the data center network, and 2) issues requests to different objects that are copies of each other. This unbounded configuration space makes it impossible to simply *measure* the latency distribution offered by every candidate configuration of CosTLO.
- ***Complex service architectures.*** However, *predicting* the impact on latencies of any particular approach for issuing redundant requests is complicated by the fact that we have little visibility into the architecture of any cloud storage service. As we describe later, due to correlations between concurrent requests, we cannot estimate the latencies obtained with $k$ concurrent requests simply by considering the minimum of $k$ independent samples of a single request's latency distribution.
- ***Multi-dimensional pricing policies.*** Finally, minimizing CosTLO's cost overhead is made complex by the fact that cloud services charge customers based on a combination of storage, request, VM, and bandwidth costs. Each of the potential ways in which redundant requests can be issued impacts a subset of these pricing dimensions, and the extent to which it does so depends on the application's workload.

# 4 Characterizing Configuration Space

CosTLO's approach of issuing redundant requests to reduce tail latencies can broadly be applied in two ways. One way is to concurrently issue the same request multiple times in order to *implicitly* exploit load balancing in the Internet or inside cloud services. For example, issuing multiple GET requests concurrently to the same object may lower latencies either because different requests take different paths through the Internet to the same data center, or because different requests may be served by different storage servers that host replicas of the same object. An alternate way is to *explicitly* enforce diversity by concurrently issuing a set of requests that differ from each other, yet have the same effect, e.g., by storing multiple copies of an object and issuing concurrent requests to different copies, or by issuing concurrent requests to different front-ends of a storage service.

Here, we empirically evaluate on both S3 and Azure the efficacy of several approaches for reducing tail latencies in three components of end-to-end latency: Internet latency, data center network latency, and latency in the storage service. We ignore DNS latency since applications often do not control how clients perform DNS lookups and concurrently querying multiple nameservers to reduce DNS latencies has no impact on cost.

## 4.1 Internet latencies

To examine the utility of different approaches on reducing Internet tail latencies, we issue pairs of concurrent GET requests from each PlanetLab node in three different ways and then compare the measured tail latencies with those seen with single requests. We use the notation "$n$x $C[m]$" to denote a setting in which every PlanetLab node issues $n$ concurrent requests to its $m^{th}$ closest data center in cloud $C$, where $C$ is either S3, Azure, or the union of data centers in the two ("S3/Azure").

**Multiple requests to same data center.** To account for spikes in Internet latency, we first consider every end-host concurrently issuing multiple requests to the storage service in the data center closest to it. Load balancing in the Internet [14] may result in concurrent requests tak-

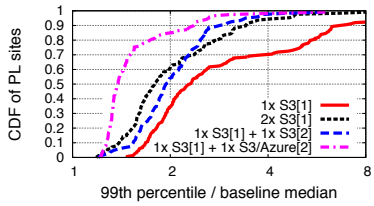



Figure 5: *Impact on Internet tail latencies of different ways to send two concurrent requests. Logscale on x-axis.*

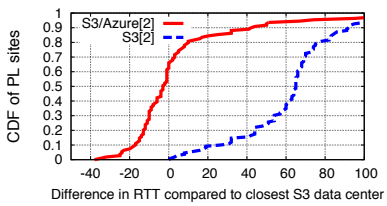



Figure 6: *Comparison of second closest data center within a cloud service and across cloud services.*

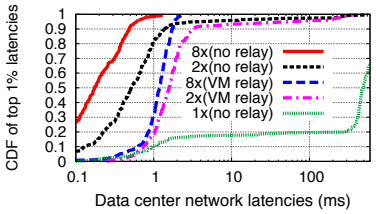



Figure 7: *Different ways of exploiting path diversity in the data center network. Note logscale on x-axis.*

ing different paths to the same data center.[1] However, as shown by the *"2x S3[1]"* line in Figure 5, though issuing two concurrent requests to the same data center does reduce the inflation in tail latencies, relative inflation seen at the median PlanetLab node remains close to 2x; the *"1x S3[1]"* line represents the baseline where end-hosts issue single GET requests to their closest data center.

**Requests to multiple data centers.** Since path diversity to the same data center is insufficient to tame Internet latency spikes, we next consider issuing concurrent requests to multiple data centers; in addition to a GET request to its closest S3 data center, we have every node issue a GET request in parallel to its second closest S3 data center. The *"1x S3[1] + 1x S3[2]"* line in Figure 5 shows that this strategy offers little benefit in reducing latency variance. This is because, for most PlanetLab nodes, the second closest data center is too far to help tame latency spikes to the node's closest data center.

The root cause for this is that any particular cloud service provider provisions its data centers in a manner that maximizes geographical coverage. Hence, any pair of data centers in the same cloud service are distant from each other. For example, the *"S3[2]"* line in Figure 6 shows that RTT to the second closest data center in S3 is 40ms greater than the RTT to the closest S3 data center for over 80% of PlanetLab nodes.

**Leveraging multiple cloud providers.** Though a single cloud provider's data centers are distant from each other, we observe that different cloud providers often have nearby data centers. For example, Figure 6 shows that, for over 80% of PlanetLab nodes, RTT to the second closest data center across S3 and Azure is within 25ms of the RTT to the closest S3 data center.

Therefore, leveraging the fact that storage services offered by all cloud providers largely offer the same PUT/GET interface, every client of an application can download copies of an object in parallel from 1) the closest data center among the ones on which the application is deployed, and 2) the second closest data center across all storage services that offer a PUT/GET interface. Figure 5 shows that doing so reduces the inflation in $99^{th}$ percentile GET latency to be less than 1.5x the baseline median at 70% of PlanetLab nodes. Note that the application itself can be deployed across a single cloud provider's data centers. As we describe later (Section 5.1), CosTLO can maintain copies of objects without the application's knowledge.

[1]Multiple requests may also help in surviving packet losses. However, loss rates in our measurements are below 0.1%, thus making them an insignificant factor in causing latency spikes.

### 4.2 Data center network latencies

Next, we consider strategies for tackling latency spikes within a cloud service's data center network.

In this case, we first attempt to implicitly exploit path diversity by issuing the same PUT/GET request multiple times in parallel from a VM to the local storage service. Load balancing within the data center network [24] may cause concurrent requests to take different routes to the same front-end of the storage service, thus enabling us to avoid latency spikes that occur on any one path.

Alternatively, we can explicitly exploit path diversity in two ways. When a VM issues a GET/PUT to the local storage service, we can either relay each request through a different VM (Figure 4(d)), or issue each request to a different front-end of the storage service. While the latter approach is applicable in S3, all requests issued by the same tenant are submitted to the same front-end [20] in Azure. Therefore, we only consider here the former way of explicitly exploiting path diversity.

In one of Azure's data centers, Figure 7 compares the distribution of tail latencies over the network in three scenarios for how a VM downloads objects from the local storage service: 1) a single request is issued, 2) concurrent requests are issued directly to the same front-end, and 3) concurrent requests are relayed via different VMs. In the latter two cases, we experiment with different levels of parallelism. We see that both implicit and explicit exploitation of path diversity significantly reduce tail latencies, with higher levels of parallelism offering greater reduction. However, using VMs as relays adds some overhead, likely due to requests traversing longer routes.

### 4.3 Storage service latencies

Finally, we evaluate two approaches for reducing latency spikes within the storage service, i.e., latency between when a request is received at a front-end and when it sends back the response. When issuing $n$ concurrent requests to a storage service, we either issue all $n$ requests for the same object or to $n$ different objects. The former attempts to implicitly leverage the replication of

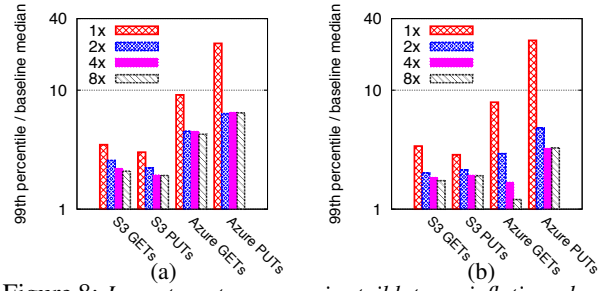



Figure 8: *Impact on storage service tail latency inflation when issuing concurrent requests to (a) the same object, and (b) to different objects. Note logscale on y-axis.*

objects within the storage service, whereas the latter explicitly creates and utilizes copies of objects. In either case, if concurrent requests are served by different storage servers, latency spikes at any one server can be overridden by other servers that are lightly loaded.

At one data center each in Azure and S3, Figure 8 shows that both approaches for issuing concurrent requests significantly reduce tail GET and PUT latencies. However, the takeaways differ between Azure and S3. On S3, irrespective of whether we issue multiple requests to the same object or to different objects, the reduction in $99^{th}$ percentile latency tails off with increasing parallelism. As seen later in Section 5, this is because, in S3, concurrent requests from a VM incur the same latency over the network, which becomes the bottleneck in the tail. In contrast, on Azure, $99^{th}$ percentile GET latencies do not reduce further when more than 2 concurrent requests are issued to the same object, but tail GET latencies continue to drop significantly with increasing parallelism when concurrent requests are issued to different objects. In the case of PUTs, the benefits of redundancy tail off at parallelism levels greater than 2 due to Azure's serialization of PUTs issued by the same tenant [20].

### 4.4 Takeaways

In summary, our measurement study highlights the following viable options for CosTLO to reduce latency variance via redundancy. First, CosTLO can tackle spikes in Internet latencies by issuing multiple requests to a client's closest data center. If greater reduction in Internet tail latencies is desired, CosTLO must concurrently issue requests to the two closest data centers to the client from the union of data centers in multiple cloud services. Second, for latency spikes in a data center's network, it suffices to issue multiple requests to the storage service in that data center. While explicitly relaying requests via VMs may help reduce bandwidth costs (as seen in our example earlier in Section 3), they do not offer additional benefits in reducing latencies. Finally, for latency spikes within the storage service, CosTLO can issue multiple requests either to the same object or to different objects that are all copies of the object being accessed.

## 5 Cost-effective Support for SLOs

Next, we describe how CosTLO combines the use of the above-mentioned viable redundancy options in order to satisfy an application's SLO cost-effectively.

### 5.1 System architecture

**Application interface.** As shown in Figure 9(a), application code on end-hosts links to CosTLO's client library and uses the GET operation[2] in this library to fetch data from cloud storage. The client library issues a set of GET requests to download an object and returns the object's data to the application as soon as any one GET completes. Unlike downloads, we let client-side application code upload data to its own VMs, because the application may need to update application-specific metadata before writing user-uploaded data to cloud storage. The application code in these VMs links to CosTLO's VM library and invokes the PUT operation in this library to write data to the local storage service. The VM library in turn issues a set of PUT requests to the local storage service, and informs the application that the PUT operation is complete once any one of the PUT requests finish. CosTLO offers the same consistency semantics as S3 [3]: read-after-write consistency for PUTs of new objects and eventual consistency for overwrite PUTs; we discuss how CosTLO can support strong consistency later in Section 7.

**Configuration selection.** CosTLO's central ConfSelector selects the configuration in which its client library and VM library should serve PUTs and GETs. ConfSelector divides time into epochs, and at the start of every epoch, it selects a new configuration separately for every IP prefix, since Internet latencies to any particular data center are similar from all end-hosts in a prefix [31]. To exploit weekly stability in workloads [12], we set epoch durations to one week; we do not consider exploiting diurnal workload patterns because we observe good cost-efficiency even when only leveraging weekly workloads stability. At the start of every epoch, the CosTLO library on every end-host and instances of CosTLO's VM library in every data center fetch the configurations that are relevant to them. Since all objects accessed by a client are replicated as per the configuration associated with the client's prefix, no per-object metadata is necessary. If a client loses its state, it simply re-fetches the configuration in the current epoch for its prefix from ConfSelector.

In the rest of this section, we address three questions: 1) how does ConfSelector identify a cost-effective configuration of CosTLO that can satisfy the application's SLO?, 2) while searching for this cost-effective configuration, how does ConfSelector *estimate* the tail latencies

[2]When ambiguous, we refer to applications invoking CosTLO's GET/PUT *operations*, and CosTLO issuing GET/PUT *requests* to storage services.

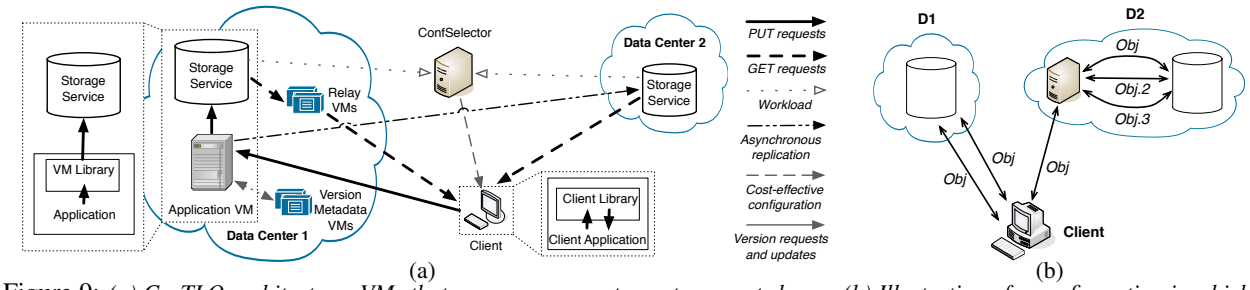



Figure 9: *(a) CosTLO architecture; VMs that run measurement agents are not shown. (b) Illustration of a configuration in which the tuples for data centers D1 and D2 are* (Copies=1, ReqPerCopy=2, VM=False) *and* (Copies=3, ReqPerCopy=1, VM=True)*. All edges are annotated with the name of the object for which GET requests are issued when the client requests object Obj.*

for any configuration, given that is impractical to *measure* the latencies offered by every configuration?, and 3) how does CosTLO preserve data consistency?

### 5.2 Selecting cost-effective configuration

**Characterization of workload and cloud services.** To estimate the cost overhead and latency variance associated with any CosTLO configuration, ConfSelector 1) takes as input the pricing policies at every data center, 2) uses logs exported by cloud providers to characterize the workload imposed by clients in every prefix, and 3) employs a measurement agent at every data center. Every agent gathers three types of measurements: 1) pings to a representative end-host in every prefix, 2) pairs of concurrent GETs and pairs of concurrent PUTs to the local storage service, and 3) the rates at which VMs can relay PUTs and GETs between end-hosts and the local storage service without any queueing. We ignore the impact of VM failures on tail latency since cloud providers guarantee over 99.95% of uptime for VMs [2, 8].

**Representation of configurations.** To search through the configuration space, ConfSelector represents every candidate configuration for a prefix as follows. First, a configuration's representation includes two three-tuples, which specify the manner in which end-hosts in the prefix should execute GETs. One three-tuple is for the data center from which the application serves the prefix and another for the data center closest to the prefix among all other data centers on which CosTLO is deployed. Either tuple specifies 1) number of copies of the object stored in that data center, 2) number of requests issued to each copy, and 3) whether all of these requests are relayed via a VM.[3] Figure 9(b) depicts an example.

Second, the configuration includes one two-tuple for the manner in which CosTLO's VM library should serve PUTs from the prefix. We use only one tuple in this case, since PUTs from an end-host are served solely at the data center closest to it, and we use a two-tuple, since the VM library does not relay PUTs through other VMs.

[3]If necessary, these three-tuples can be extended to include other dimensions, e.g., whether each request is issued to a different front-end. The dimensions we use here are based on the techniques that we found to be viable in reducing tail latencies on Azure and S3 (Section 4).

Third, to reduce the cost overhead associated with redundant requests, the client/VM library initially issues a single request when serving a GET/PUT. If no response is received for a period, the client/VM library times out and probabilistically issues redundant requests concurrently as specified by the tuples described above. The timeout period ensures that CosTLO's redundancy is focused on requests that incur a high latency, whereas probabilistically issuing redundant requests offers finer-grained control over latency variance. For both PUTs and GETs, the configuration representation specifies the values of the timeout period and probability parameters. Considering the same example from Figure 9(b) but with 70% probability and 50ms timeout period to issue redundant requests, the configuration would be [(1, 2, False), (3, 1, True), 50ms, 70%] (the PUT tuple is ignored here).

**Configuration search.** Given this representation of the configuration space, ConfSelector identifies a cost-effective configuration of CosTLO for any particular prefix as follows. It initializes the configuration for a prefix to reflect the manner in which an application serves its clients when not using CosTLO—by always issuing only a single request to the data center closest to a client. CosTLO imposes no cost overhead in this configuration.

Thereafter, our structured representation of the configuration space enables ConfSelector to step through configurations in the increasing order of cost. For this, ConfSelector maintains a pool of candidate configurations, from which it considers the minimum cost configuration in every step. ConfSelector computes the cost associated with a configuration as the sum of expected costs for storage, VMs, requests, and bandwidth based on the workload for the prefix and the manner in which the configuration mandates that GET/PUT operations be served. If the lowest cost configuration in the current pool does not satisfy the SLO, ConfSelector discards this configuration and inserts all neighbors of this configuration into the pool of candidates. Two configurations are neighbors if they differ in the value of exactly one parameter in the configuration representation. For example, configurations [(1, 2, False), 50ms, 70%] and [(2, 2, False), 50ms, 70%] are neighbors (we only show one GET tuple

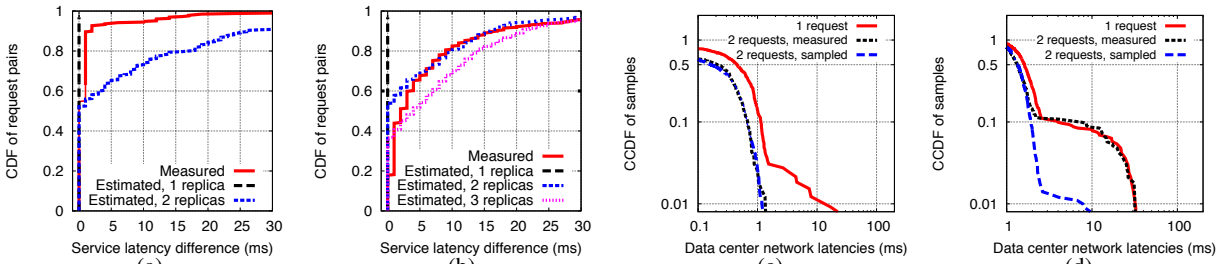



Figure 11: *Distribution of service latency difference between concurrent GET requests offers evidence for GETs to an object (a) being served by one replica in Azure, and (b) being spread across two replicas in S3. Data center network latencies for concurrent requests are (c) uncorrelated on Azure, and (d) correlated on S3. Note logscale on both axes of (c) and (d).*

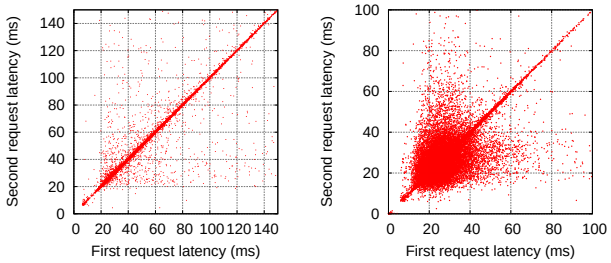



Figure 10: *Scatter plot of first vs. second request GET latency when issuing two concurrent requests to a storage service.*

here for simplicity). This process terminates once ConfSelector finds a configuration that satisfies the SLO.

### 5.3 Estimating latency distribution

To identify when it has found a configuration that will satisfy the application's SLO, for any particular configuration for a prefix, ConfSelector must be able to estimate the latency distribution that clients in that prefix will experience when served in that configuration. For brevity, we present here ConfSelector's estimation of latencies only for GETs, which it computes in four steps. First, for either data center used in the configuration, we estimate the latency distribution when a VM in that data center concurrently issues requests to the local storage service, where the number of requests is specified by the data center's tuple in the configuration representation. Second, we estimate the latency distribution for either data center's tuple by adding the distribution computed above with the latency distribution measured to the prefix from a VM in that data center. Simply adding these distributions works when objects are smaller than 1 KB, and in Section 7, we discuss how to extrapolate this distribution for larger objects. Third, we estimate the client-perceived latency distribution by independently sampling the latency distributions associated with either tuple in the configuration and considering the minimum. Finally, we adjust this distribution to account for the timeout and probability parameters.

The primary challenge here is the first step: estimating the latency distribution when a VM issues concurrent requests to the local storage service. This turns out to be hard due to the dependencies between concurrent requests. While Figure 10 shows the correlation in latencies between two concurrent GET requests to an object at one of Azure's and one of S3's data centers, we also see similar correlations for PUTs and even when the concurrent requests are for different objects. Attempting to model these correlations between concurrent requests by treating the cloud service as a black box did not work well. Therefore, we explicitly model the sources of correlations: concurrent requests may incur the same latency within the storage service if they are served by the same storage server, or incur the same data center network latency if they traverse the same network path.

**Modeling replication in storage service.** First, at every data center, we use CosTLO's measurements to infer the number of replicas across which the storage service spreads requests to an object. For every pair of concurrent requests issued during CosTLO's measurements, we compute the difference in service latency (i.e., latency within the storage service) between the two requests. We then consider the distribution of this difference across all pairs of concurrent requests to infer the number of replicas in use per object. For example, if the storage service load balances GET requests to an object across 2 replicas, there should be a 50% chance that two concurrent GETs fetch from the same replica, therefore the service latency difference is expected to be 0 half the time. We compare this measured distribution with the expected distribution when the storage service spreads requests across $n$ replicas, where we vary the value of $n$. We infer the number of replicas used by the service as the value of $n$ for which the estimated and measured distributions most closely match. For example, though both Azure [5] and S3 [4] are known to store 3 replicas of every object, Figures 11(a) and 11(b) show that the measured service latency difference distributions closely match GETs being served from 1 replica on Azure and from 2 replicas on S3.

On the other hand, for concurrent GETs or PUTs issued to different objects, on both Azure and S3, we see that the latency within the storage service is uncorrelated across requests. This is likely because cloud stor-

age services store every object on a randomly chosen server (e.g., by hashing the object's name for load balancing [23]), and hence, requests to different objects are likely to be served by different storage servers.

**Modeling load balancing in network.** Next, we identify whether concurrent requests issued to the storage service incur the same latency over the data center network, or are their network latencies independent of each other. At any data center, we compute the distribution obtained from the minimum of two independent samples of the measured data center network latency distribution for a single request. We then compare this distribution to the measured value of the minimum data center network latency seen across two concurrent requests.

Figure 11(c) shows that, on Azure, the distribution obtained by independent sampling closely matches the measured distribution, thus showing that network latencies for concurrent requests are uncorrelated. Whereas, on S3, Figure 11(d) shows that the measured distribution for the minimum across two requests is almost identical to the data center network latency component of any single request; this shows that concurrent requests on S3 incur the same network latency.

**Estimating VM-to-service latency.** Given these models for replication and load balancing, we estimate the end-to-end latency distribution as follows when a VM issues $k$ concurrent requests to the local storage service. If concurrent requests are known to have the same latency over the service's data center network, we sample the measured data center network latency distribution once and use this value for all requests; if not, we independently sample once for each request. If all $k$ requests are to the same object, then we randomly assign every request to one of the replicas of the object, where the number of replicas is identified as described above. If the $k$ requests are for $k$ different objects, then we assume that no two requests are served from the same storage server. In either case, for each storage server, we independently choose a sample from the service latency distribution for a single request and assign that to be the service latency for all requests assigned to that server. Finally, for each of the $k$ requests, we sum up their assigned data center network latency and service latency values, and estimate the end-to-end latency at the VM as the minimum of this sum across the $k$ requests.

Note that our latency estimation models may potentially break down at high storage service load. But, we have not seen any evidence of this so far, since we see the same latency distribution irrespective of whether we issue requests once every 3 seconds or once every 200ms.

### 5.4 Ensuring data consistency

CosTLO can afford to inform the application that a PUT operation is complete as soon as any of the PUT requests that it issues to serve the operation finish, because the underlying cloud services guarantee that the data written by a completed PUT request will be durable [5, 4]. However, this design decision makes it challenging for CosTLO to ensure that, eventually, all GETs for an object will consistently return the same data. First, if the application issues back-to-back or concurrent PUT operations on the same object, redundant PUT requests that are still pending from a completed PUT operation may potentially overwrite updates written by subsequent PUT operations. Second, if an application VM restarts after only a subset of the PUT requests issued to serve a PUT operation complete, the VM library will not realize if some of the remaining PUT requests fail, thus causing some of the copies of the object to potentially not reflect the latest update to the object.

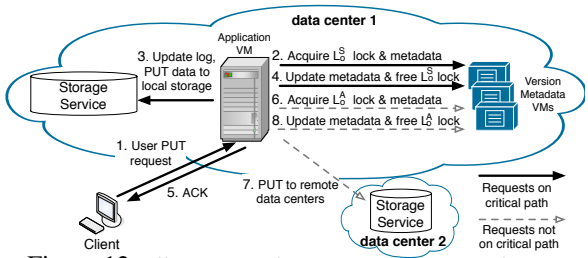


Figure 12: *Illustration of CosTLO's execution of PUTs.*

Figure 12 illustrates the execution of PUTs in CosTLO accounting for these concerns. In every data center, CosTLO maintains a set of VMs that store in memory (with a persistent backup) the latest version number and the status of two locks $L_o^S$ and $L_o^A$ for every object $o$ stored in that data center. We use $L_o^S$ for synchronous PUTs to local storage service and $L_o^A$ for asynchronous PUTs to remote storage services. When serving a PUT operation on object $o$, the VM library first queries the local cluster of CosTLO's VMs to obtain lock $L_o^S$ and learn $o$'s current version. Once it acquires the lock, the library appends to a persistent log (maintained locally on the VM) the update that needs to be written to $o$ and all the PUT requests that the library needs to issue as per the configuration for the client issuing this PUT operation. By appending the status of every response to the log, the library ensures that it knows which PUTs to re-issue, even across VM restarts. Once all PUT requests complete, the library releases lock $L_o^S$, updating $o$'s version in the process. At some point later, the library attempts to acquire lock $L_o^A$, and if $o$'s version has not changed by then, it updates the remaining copies of $o$ and subsequently releases the lock. If $o$'s version has changed, the library just needs to release the lock, since there exists a newer PUT operation on this key and that PUT's asynchronous propagation will suffice to update the remaining copies of $o$.

Note that, since the application is unaware of the replication of objects across data centers, all PUT operations on an object will be issued by the application's VMs in

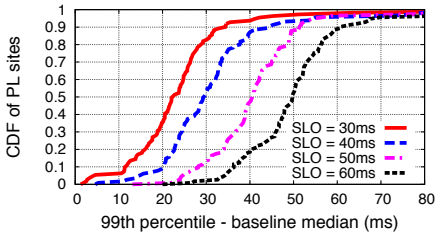



Figure 13: *Verification of CosTLO's ability to satisfy SLOs.*

the same data center. Hence, the VM library needs to acquire locks only from CosTLO's VMs within the local data center, thus ensuring that locking operations add negligible latency. Also note that, when an application issues back-to-back PUT operations, execution of the latter PUT has to wait for the lock $L_o^S$ (for the object $o$ being updated) to be released. This can potentially increase[4] tail latencies if multiple PUT requests need to complete before $L_o^S$ is released. Therefore, in the rare case when an application often issues back-to-back or concurrent PUTs for the same object, the application should choose an SLO that offers no improvement in PUT latency variance; this will ensure that CosTLO executes any PUT operation by issuing a single PUT request.

# 6 Evaluation

We evaluate CosTLO from three perspectives: 1) its ability to satisfy latency SLOs, 2) its cost-effectiveness in doing so, and 3) its efficiency in various respects. We perform our evaluation from the perspective of an application deployed across all of Amazon's data centers. We deploy CosTLO across Azure's and S3's data centers, and use PlanetLab nodes at 120 sites as clients.

## 6.1 Ability to satisfy SLOs

**SLOs on individual operations.** To verify CosTLO's ability to satisfy latency SLOs, we mimic a deployment of Wikipedia using server-side logs of objects requested from the English version of Wikipedia [6]. We randomly select a 1% sample from the datasets for two consecutive weeks. We provide the workload from the first week to ConfSelector as input, and have it select cost-effective configurations for 120 PlanetLab nodes. We then run CosTLO with every node configured in the manner selected by ConfSelector. We replay the workload from the second week, with every GET request assigned to a random PlanetLab node. We repeat this experiment for four SLO values—30ms, 40ms, 50ms, and 60ms. In all cases, since we issue GETs/PUTs to S3 and Azure, our measurements are affected by Internet congestion and by contention with S3's and Azure's customers.

Figure 13 shows the distribution of the measured difference between the $99^{th}$ percentile and baseline median

[4]Note that we can reduce the extent of this increase in inflation by having CosTLO maintain a lock $L_{o,c}^S$ for every copy $c$ of object $o$, but we do not present such a design here to keep the discussion simple.

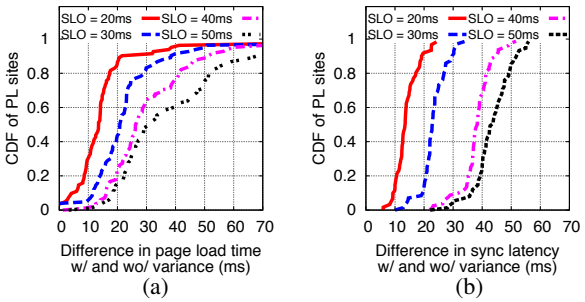



Figure 14: *CosTLO's ability to satisfy application-specific SLOs for (a) webpage and (b) social network applications.*

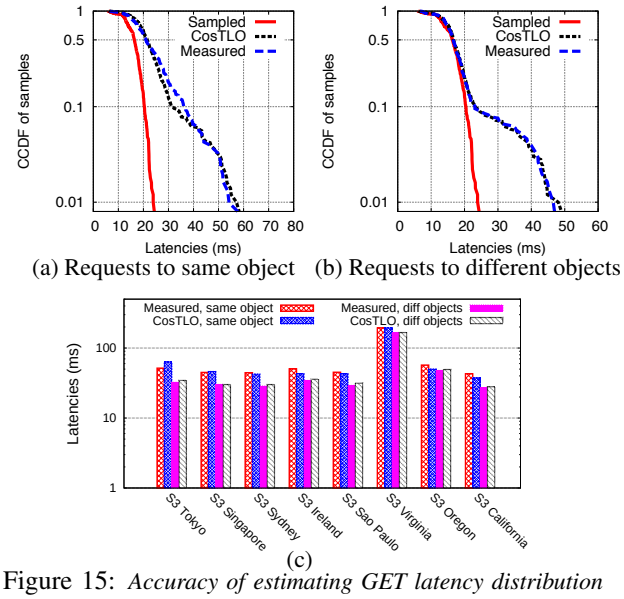
latencies at every PlanetLab node. For all SLOs, the latency variance delivered by CosTLO is within the input SLO on most nodes; without CosTLO, the difference between $99^{th}$ percentile and baseline median GET latencies is greater than 60ms for 75% of PlanetLab nodes (Figure 1(a)). Latency variance with CosTLO is, in fact, well below the SLO in many cases; due to discontinuous drops in the latency distribution across neighboring configurations, as ConfSelector steps through the configuration space, it often directly transitions from a configuration that violates the SLO to one that exceeds it.

Note that, though we only demonstrate CosTLO's ability to satisfy GET latency SLOs here (because the trace from Wikipedia only contains GETs), CosTLO can also reduce the latency variance for PUTs as described earlier. In contrast, in-memory caching of data can only reduce tail latencies for GETs, but not for PUTs.

**Application-specific SLOs.** CosTLO's design is easily extensible to handle application-specific SLOs, rather than the SLOs for the latencies of individual PUT/GET operations. Here, we show the results of using CosTLO to reduce user-perceived latencies in the two applications from Section 2. In the webpage application, we modify ConfSelector so that it uses the models in Section 5.3 to estimate the distribution for the latency incurred when the client library fetches 50 objects in parallel and waits for at least one GET to each of these objects to complete. In the social network application, since we need to estimate latencies from multiple users, we extend the configuration representation in ConfSelector such that it contains the configuration tuples of all of a user's followers. The sync completion time is determined when all followers have at least one GET completed. We use this modified version of ConfSelector to select configurations for all PlanetLab nodes and run CosTLO's client library on every node as per these configurations. Figure 14 shows that CosTLO is able to satisfy application-specific SLOs in both applications.

## 6.2 Accuracy of estimating latency distributions

CosTLO is able to meet latency SLOs due to its accurate estimation of the end-to-end latency distributions in any configuration. Our simple approaches of considering




(a) Requests to same object (b) Requests to different objects

(c)

Figure 15: *Accuracy of estimating GET latency distribution for 8 concurrent GET requests from VM to local storage service. (a and b) Comparison of latency distributions in one S3 region. (c) Comparison across all S3 regions of* $99^{th}$ *percentile latencies. Note logscale on y-axis of all three graphs.*

the minimum of the latency distributions across data centers and of adding VM-to-prefix and VM-to-service latency distributions work reasonably well; in either case, CosTLO's estimates show less than 15% error for 90% of PlanetLab nodes. Therefore, here we focus on demonstrating the accuracy of our estimation of the latency distribution when a VM concurrently issues a set of requests to the local storage service. Recall that CosTLO only gathers measurements when issuing pairs of concurrent requests. We evaluate its ability to estimate the latency distribution for higher levels of parallelism.

Figures 15(a) and 15(b) compare the measured and estimated latency distributions when issuing 8 concurrent GETs from a VM to the local storage service; all concurrent requests are for the same object in the former and to different objects in the latter. In both cases, our estimated latency distribution closely matches the measured distribution, even in the tail. In contrast, if we estimate the latency distribution for 8 concurrent GETs by independently sampling the latency distribution for a single request 8 times and considering the minimum, we significantly under-estimate the tail of the distribution. Additionally, Figure 15(c) shows that the relative error between the measured and estimated values of the $99^{th}$ percentile GET latency is less than 5% in the median S3 region; latencies are higher for S3's Virginia data center because it is the most widely used data center in S3.

### 6.3 Cost-effectiveness

An application that uses CosTLO incurs additional costs for storing copies of objects, for operations and bandwidth due to redundant requests, and for VMs used

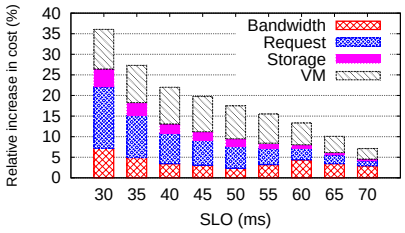
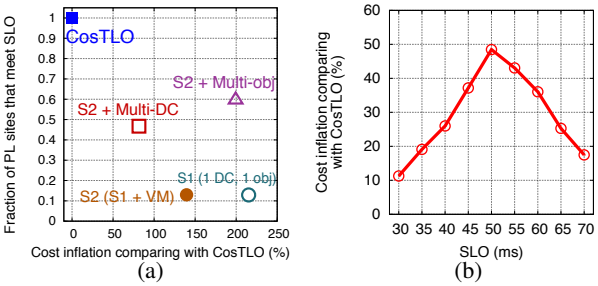



Figure 16: *CosTLO's cost-effectiveness in satisfying SLOs.*




(a) (b)

Figure 17: *(a) Utility of CosTLO's components in reducing cost and meeting SLO = 30ms. (b) Cost inflation when not using timeout and probability parameters.*

either as relays or to manage locks and version numbers. We again use Wikipedia's workload to quantify this overhead on an application provider's costs.

Figure 16 shows the relative cost overhead as a function of the latency SLO, with the cost split into its four components. At the higher end of the examined range of SLO values, CosTLO caps tail latency inflation at 70ms—which is less than the inflation observed at the median node when not using CosTLO—with less than 8% increase in cost. As the SLO decreases, i.e., as lower variance is desired, cost increases initially due to an increase in the number of redundant requests. Thereafter, as the SLO further decreases, CosTLO begins to use more relay VMs so that only one copy of any requested object leaves the data center, thus decreasing bandwidth costs at the expense of VM costs. As the SLO decreases further, CosTLO begins concurrently issuing requests to multiple data centers, thus again increasing bandwidth costs. Storage costs and cost for VMs that manage locks and version numbers remain low for all SLO values, because 1) on both Amazon's and Microsoft's cloud services, storage is significantly cheaper than GET/PUT requests, VMs, and network transfers, and 2) lock status and version numbers for all 70M objects in the English version of Wikipedia fit into the memory of a small instance VM, which costs less than $20 per month on EC2.

### 6.4 Utility of CosTLO's components

CosTLO's ability to satisfy SLOs cost-effectively crucially depends on its *combined use* of various forms of issuing redundancy. We illustrate this in Figure 17(a) by comparing CosTLO with several strategies that each use a subset of the dimensions in CosTLO's configuration space. For each strategy, we compute the fraction of Pla-

netLab nodes for which it is able to satisfy an SLO of 30ms, and across the nodes on which the strategy does meet the SLO, we compare its cost with CosTLO's.

First, the simplest strategy $S1$, which only issues redundant requests to a single copy of any object in the data center closest to any client, can meet the SLO on only a little over 10% of nodes. Adding the use of relay VMs ($S2$) reduces cost inflation compared to CosTLO from over 200% to less than 150%, but the ability to meet the SLO remains unchanged. We can improve the ability to satisfy the SLO by adding the option of issuing redundant requests either to multiple copies of every object in the closest data center or to multiple data centers. However, the fraction of nodes on which the SLO can be met remains below 60% if we use one of these two options. Only by combining the use of relay VMs, multiple copies of objects, and multiple data centers is CosTLO able to meet the SLO at all nodes, at significantly lower cost.

In addition, we illustrate the utility of CosTLO waiting for a timeout period before issuing redundant requests and issuing redundant requests probabilistically. For every SLO in the range 30ms to 70ms, Figure 17(b) compares CosTLO's cost overhead when it uses the timeout and probability parameters versus when it does not. The cost overhead of not using the timeout and probability parameters is low when the SLO is extremely low or extremely high. In the former case, most PlanetLab nodes need to issue redundant requests at all times without any timeout in order to meet the SLO, whereas in the latter case, the SLO is satisfied for most PlanetLab nodes even without redundant requests. However, for many intermediate SLO values—that are neither too loose nor too stringent—not using the timeout and probability parameters increases cost significantly, by as much as 48%.

### 6.5 Efficiency

**Measurement cost.** The cost associated with CosTLO's measurements depends on the number of latency samples necessary to accurately sample latency distributions. To quantify the stationarity in latencies, we consider a dataset of 200K latency measurements gathered over a week from VMs in every S3 and Azure data center. We then consider subsets of these datasets, varying the number of samples considered. In all datasets, we find that 10K samples are sufficient to obtain a reasonably accurate value of the $99^{th}$ percentile latency. In the ping, GET, and PUT latency measurements, the $99^{th}$ percentile from a subset of 10K samples was off from the $99^{th}$ percentile in the entire dataset by only 2.9%, 3.8%, and 2.2% on average.

Thus, at every data center, CosTLO's weekly measurement costs include: 1) 20K GETs and PUTs (since CosTLO gathers data with pairs of concurrent requests), 2) 10K pings to every end-host prefix, and 3) one "small instance" VM (which is sufficient to support this scale

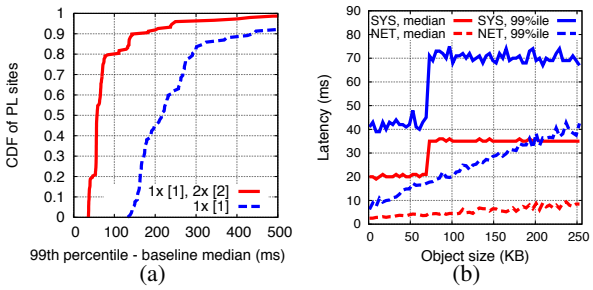



Figure 18: *(a) CosTLO's utility in reducing latency variance when offering strong consistency; 1 PUT request per copy. (b) latency breakdown for objects of different sizes.*

of measurements). Accounting for the roughly 120K IP prefixes at the Internet's edge [27], 8 S3 data centers, and 13 Azure data centers, these measurements translate to a total cost of $392 per week. These minimal measurement costs are shared across all applications that use CosTLO.

**Configuration selection runtime.** We run ConfSelector to select the configurations for 120 PlanetLab nodes, and we compute the average runtime per node. We repeat this for SLO values ranging from 20ms to 100ms. Extrapolating the average runtime per node, we estimate that, for all SLO values, ConfSelector needs less than a day to select the configuration for all 120K edge prefixes on a server with 16 cores. Hence, ConfSelector can identify the configurations for a particular week during the last day of the previous week. Moreover, since ConfSelector independently selects configurations for different prefixes, this runtime is easily reduced by parallelizing ConfSelector's execution across a cluster of servers.

## 7 Discussion

**Strong consistency.** Many applications (e.g., Google Docs) require their underlying storage to offer strong consistency. For such applications, CosTLO uses only strongly consistent storage services, e.g., it can use Azure but not S3. In addition, two modifications are necessary in the execution of a PUT operation on any object $o$. First, to ensure linearizability of PUTs, the VM library synchronously updates all copies of $o$ before releasing lock $L_o^S$. Second, instead of the library informing the application when any one PUT request completes, the application registers for two callbacks—1) *quorumPUTsDone*, for when at least one PUT request each completes on a quorum of $o$'s copies, and 2) *allPUTsDone*, when all PUTs finish. The *quorumPUTsDone* callback indicates to the application that subsequent GET operations on $o$ will fetch the latest version, if the client library waits for responses from a quorum of copies when serving GETs.

After these changes, Figure 18(a) shows the PUT latency variance offered by CosTLO when every object accessed by a PlanetLab node has one copy and two copies, respectively, in the closest Azure data center and the second closest data center across S3 and Azure; for this anal-

ysis, we ignore that S3 does not offer strong consistency. Despite having to wait for PUT requests on a quorum of copies to complete, and though a quorum of every object's copies are stored in a different data center than the application VMs that issue PUT operations on the object, CosTLO more than halves the PUT latency inflation for the median node. This again highlights the utility of redundant copies and requests, and of CosTLO's use of multiple cloud services.

**Latency estimation for larger objects.** One can potentially extend CosTLO's approach for estimating latency variance to larger objects as follows. We conduct measurements on objects from 1 KB to 256 KB when issuing one GET request at a time to each object from a local VM, and Figure 18(b) shows the results from one data center. We see that network latency is proportional to object size, and storage service latency is a step function of object size. Although different data centers may have different step functions (we observe that some data centers have the same storage service latency distribution for all sizes in the 256 KB range), the smallest range that has a fixed storage service latency distribution is until 64 KB, which is a typical block size in distributed storage systems [21]. Therefore, to estimate latencies for objects of different sizes, we can leverage the fact that objects with the same number of blocks have the same storage service latency distribution.

**Scale of adoption.** CosTLO's approach of issuing redundant requests makes it unviable if all applications adopt it. However, we believe that increasing adoption of CosTLO will emphasize the demand for latency SLOs and spur cloud providers to suitably modify their services. In the interim, CosTLO minimizes the cost overhead incurred by application providers who seek to improve predictability in user-perceived latencies without having to wait for any changes to cloud services. Moreover, cloud service providers have little control over reducing variance in the latency on the Internet path between end-hosts and their data centers.

# 8 Related Work

**Redesigning cloud services.** Several recent proposals redesign storage systems and data centers to improve tail latency performance [36], to offer bandwidth guarantees to tenants [15, 33, 16, 37], or to ensure predictable completion times for TCP flows [42, 26, 40]. However, all of these proposals require modifications to a cloud service's infrastructure. It is unclear when, and if, cloud services will revamp their infrastructure to these more complex architectures. CosTLO instead satisfies latency SLOs for applications deployed on the cloud without having to wait for any modifications to cloud services.

**Reducing tail latencies.** The approach of issuing redundant requests to reduce tail latencies has been considered previously [22, 38], but the focus has primarily been on understanding the implications of redundancy on system load. In contrast, our work demonstrates how the approach of using redundant requests should be applied in the context of cloud storage services, in order to meet latency SLOs while minimizing cost overhead.

Some application providers such as Facebook use in-memory caching of data to reduce tail latencies [32]. However, caching cannot reduce tail latencies associated with PUT requests. Moreover, caching at a single data center cannot tackle latency spikes on Internet paths, and not all application providers will be able to afford caches at multiple data centers that can accommodate enough data to reduce $99^{th}$ percentile GET latencies.

**Cloud measurement studies.** Prior studies have compared the performance offered by different cloud providers [29], reverse-engineered cloud service internals [34], and studied application deployments on the cloud [25]. Our measurement study of Azure and S3 is the first to quantify the latency variance on these storage services and to characterize the impact of different forms of redundancy. Moreover, unlike Bodik et al. [18], who focused on characterizing and modeling spikes in application workloads, our measurements show that an application using cloud storage can suffer latency spikes even when there is no spike in that application's workload.

**Combining cloud providers.** Others have combined the use of multiple cloud providers to improve availability [11, 28], to offer more secure storage [17], and to reduce cost [39, 41]. CosTLO uses cloud storage services offered by multiple providers because 1) the combination offers more data center pairs that are close to each other, and 2) latency spikes on the Internet paths to data centers in different cloud services are uncorrelated.

# 9 Conclusions

Our measurements of the Azure and S3 storage services highlight the high variance in latencies offered by these services. To enable applications to improve predictability, without having to wait for these services to modify their infrastructure, we have designed and implemented CosTLO, a framework that requires minimal changes to applications. Based on several insights about the causes for latency variance on cloud storage services that we glean from our measurements, our design of CosTLO judiciously combines several instantiations of the approach of issuing redundant requests. Our results show that, despite the unbounded configuration space and opaque cloud service architectures, CosTLO cost-effectively enables applications to meet latency SLOs.

# Acknowledgments

This work was supported in part by the National Science Foundation under grants 1463126 and 1150219.

## References


[1] 250+ amazing Twitter statistics. http://expandedramblings.com/index.php/march-2013-by-the-numbers-a-few-amazing-twitter-stats/.

[2] Amazon EC2 service level agreement. http://aws.amazon.com/ec2/sla/.

[3] Amazon S3 FAQs. http://aws.amazon.com/s3/faqs/.

[4] Announcing Amazon S3 reduced redundancy storage. http://aws.amazon.com/about-aws/whats-new/2010/05/19/announcing-amazon-s3-reduced-redundancy-storage/.

[5] Azure storage pricing. http://azure.microsoft.com/en-us/pricing/details/storage/.

[6] Page view statistics for Wikimedia projects. http://dumps.wikimedia.org/other/pagecounts-raw/.

[7] Server access log format - Amazon simple storage service. http://docs.aws.amazon.com/AmazonS3/latest/dev/LogFormat.html.

[8] Windows Azure service level agreements. http://azure.microsoft.com/en-us/support/legal/sla/.

[9] Windows Azure storage logging: Using logs to track storage requests. http://blogs.msdn.com/b/windowsazurestorage/archive/2011/08/03/windows-azure-storage-logging-using-logs-to-track-storage-requests.aspx.

[10] Amazon found every 100ms of latency cost them 1% in sales. http://blog.gigaspaces.com/amazon-found-every-100ms-of-latency-cost-them-1-in-sales/, 2008.

[11] H. Abu-Libdeh, L. Princehouse, and H. Weatherspoon. RACS: A case for cloud storage diversity. In *Proceedings of the 1st Annual Symposium on Cloud Computing*, 2010.

[12] S. Agarwal, J. Dunagan, N. Jain, S. Saroiu, and A. Wolman. Volley: Automated data placement for geo-distributed cloud services. In *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation*, 2010.

[13] D. G. Andersen, H. Balakrishnan, M. F. Kaashoek, and R. Rao. Improving web availability for clients with MONET. In *Proceedings of the 2nd USENIX Conference on Networked Systems Design and Implementation*, 2005.

[14] B. Augustin, X. Cuvellier, B. Orgogozo, F. Viger, T. Friedman, M. Latapy, C. Magnien, and R. Teixeira. Avoiding traceroute anomalies with Paris traceroute. In *Proceedings of the 6th ACM SIGCOMM Conference on Internet Measurement*, 2006.

[15] H. Ballani, P. Costa, T. Karagiannis, and A. Rowstron. Towards predictable datacenter networks. In *Proceedings of the ACM SIGCOMM Conference*, 2011.

[16] H. Ballani, K. Jang, T. Karagiannis, C. Kim, D. Gunawardena, and G. O'Shea. Chatty tenants and the cloud network sharing problem. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, 2013.

[17] A. Bessani, M. Correia, B. Quaresma, F. Andre, and P. Sousa. DEPSKY: Dependable and secure storage in a cloud-of-clouds. In *Proceedings of the 6th ACM European Conference on Computer Systems*, 2011.

[18] P. Bodik, A. Fox, M. J. Franklin, M. I. Jordan, and D. A. Patterson. Characterizing, modeling, and generating workload spikes for stateful services. In *Proceedings of the 1st Annual Symposium on Cloud Computing*, 2010.

[19] J. Brutlag. Speed matters for Google web search. http://services.google.com/fh/files/blogs/google_delayexp.pdf, 2009.

[20] B. Calder, J. Wang, A. Ogus, N. Nilakantan, A. Skjolsvold, S. McKelvie, Y. Xu, S. Srivastav, J. Wu, H. Simitci, J. Haridas, C. Uddaraju, H. Khatri, A. Edwards, V. Bedekar, S. Mainali, R. Abbasi, A. Agarwal, M. F. u. Haq, M. I. u. Haq, D. Bhardwaj, S. Dayanand, A. Adusumilli, M. McNett, S. Sankaran, K. Manivannan, and L. Rigas. Windows Azure storage: A highly available cloud storage service with strong consistency. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles*, 2011.

[21] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, 2006.

[22] J. Dean and L. A. Barroso. The tail at scale. *Communications of the ACM*, 2013.

[23] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles*, 2007.

[24] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta. VL2: A scalable and flexible data center network. In *Proceedings of the ACM SIGCOMM Conference*, 2009.

[25] K. He, L. Wang, A. Fisher, A. Gember, A. Akella, and T. Ristenpart. Next stop, the cloud: Understanding modern web service deployment in EC2 and Azure. In *Proceedings of the 13th ACM SIGCOMM Conference on Internet Measurement*, 2013.

[26] C.-Y. Hong, M. Caesar, and P. B. Godfrey. Finishing flows quickly with preemptive scheduling. In *Proceedings of the ACM SIGCOMM Conference*, 2012.

[27] E. Katz-Bassett, H. Madhyastha, J. John, A. Krishnamurthy, D. Wetherall, and T. Anderson. Studying black holes in the Internet with Hubble. In *Proceedings of the 5th USENIX Conference on Networked Systems Design and Implementation*, 2008.

[28] R. Kotla, L. Alvisi, and M. Dahlin. SafeStore: A durable and practical storage system. In *Proceedings of the USENIX Annual Technical Conference*, 2007.

[29] A. Li, X. Yang, S. Kandula, and M. Zhang. CloudCmp: Comparing public cloud providers. In *Proceedings of the 10th ACM SIGCOMM Conference on Internet Measurement*, 2010.

[30] R. Li, S. Wang, H. Deng, R. Wang, and K. C.-C. Chang. Towards social user profiling: Unified and discriminative influence model for inferring home locations. In *Proceedings of the 18th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2012.

[31] H. V. Madhyastha, T. Isdal, M. Piatek, C. Dixon, T. Anderson, A. Krishnamurthy, and A. Venkataramani. iPlane: An information plane for distributed services. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, 2006.

[32] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, D. Stafford, T. Tung, and V. Venkataramani. Scaling memcache at Facebook. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, 2013.

[33] L. Popa, G. Kumar, M. Chowdhury, A. Krishnamurthy, S. Ratnasamy, and I. Stoica. FairCloud: Sharing the network in cloud computing. In *Proceedings of the ACM SIGCOMM Conference*, 2012.

[34] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage. Hey, you, get off of my cloud: Exploring information leakage in third-party compute clouds. In *Proceedings of the 16th ACM Conference on Computer and Communications Security*, 2009.

[35] S. Souders. Velocity and the bottom line. http://radar.oreilly.com/2009/07/velocity-making-your-site-fast.html, 2009.

[36] L. Suresh, M. Canini, S. Schmid, and A. Feldmann. C3: Cutting tail latency in cloud data stores via adaptive replica selection. In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation*, 2015.

[37] E. Thereska, H. Ballani, G. O'Shea, T. Karagiannis, A. Rowstron, T. Talpey, R. Black, and T. Zhu. IOFlow: A software-defined storage architecture. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles*, 2013.

[38] A. Vulimiri, P. B. Godfrey, R. Mittal, J. Sherry, S. Ratnasamy, and S. Shenker. Low latency via redundancy. In *Proceedings of the 9th ACM Conference on Emerging Networking Experiments and Technologies*, 2013.

[39] A. Wieder, P. Bhatotia, A. Post, and R. Rodrigues. Orchestrating the deployment of computations in the cloud with Conductor. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, 2012.

[40] C. Wilson, H. Ballani, T. Karagiannis, and A. Rowstron. Better never than late: Meeting deadlines in datacenter networks. In *Proceedings of the ACM SIGCOMM Conference*, 2011.

[41] Z. Wu, M. Butkiewicz, D. Perkins, E. Katz-Bassett, and H. V. Madhyastha. SPANStore: Cost-effective geo-replicated storage spanning multiple cloud services. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles*, 2013.

[42] D. Zats, T. Das, P. Mohan, D. Borthakur, and R. Katz. DeTail: Reducing the flow completion time tail in datacenter networks. In *Proceedings of the ACM SIGCOMM Conference*, 2012.