E2E: Embracing User Heterogeneity to Improve Quality of Experience on the Web

Xu Zhang[1], Siddhartha Sen[2], Daniar Kurniawan[1], Haryadi Gunawi[1], Junchen Jiang[1]

[1]University of Chicago, [2]Microsoft Research

ABSTRACT

Conventional wisdom states that to improve quality of experience (QoE), web service providers should reduce the median or other percentiles of server-side delays. This work shows that doing so can be inefficient due to *user heterogeneity* in how the delays impact QoE. From the perspective of QoE, the sensitivity of a request to delays can vary greatly even among identical requests arriving at the service, because they differ in the wide-area network latency experienced *prior to* arriving at the service. In other words, saving 50ms of server-side delay affects different users differently.

This paper presents E2E, the first resource allocation system that embraces user heterogeneity to allocate server-side resources in a QoE-aware manner. Exploiting this heterogeneity faces a unique challenge: unlike other application-level properties of a web request (*e.g.*, a user's subscription type), the QoE sensitivity of a request to server-side delays cannot be pre-determined, as it depends on the delays themselves, which are determined by the resource allocation decisions and the incoming requests. This circular dependence makes the problem computationally difficult.

We make three contributions: (1) a case for exploiting user heterogeneity to improve QoE, based on end-to-end traces from Microsoft's cloud-scale production web framework, as well as a user study on Amazon MTurk; (2) a novel resource allocation policy that addresses the circular dependence mentioned above; and (3) an efficient system implementation with almost negligible overhead. We applied E2E to two open-source systems: replica selection in Cassandra and message scheduling in RabbitMQ. Using traces and our testbed deployments, we show that E2E can increase QoE (*e.g.*, duration of user engagement) by 28%, or serve 40% more concurrent requests without any drop in QoE.

CCS CONCEPTS

• **Information systems** → *Web services*; • **Human-centered computing**;

KEYWORDS:

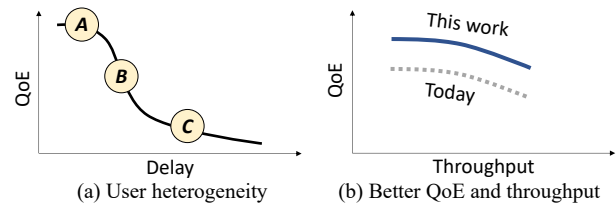Web Services, Quality of Experience, Resource Allocation

Figure 1: *(a) An example of three requests with different QoE sensitivities to server-side delays, and (b) the potential QoE/throughput improvement if we leverage user heterogeneity. These figures are illustrative; actual figures from our evaluation and trace analysis appear later (e.g., Figures 3, 6).*

1 INTRODUCTION

Improving end-to-end performance is critical for web service providers such as Microsoft, Amazon, and Facebook, whose revenues depend crucially on high quality of experience (QoE). More than ten years have passed since Amazon famously reported every 100ms of latency cost them 1% in sales, and Google found 0.5s of additional load time for search results led to a 20% drop in traffic [5]. Today, latency remains critical but the consequences have gotten steeper: an Akamai study in 2017 showed every 100ms of delay in website load time hurt conversion rates by 7% [6], and Google reported higher mobile webpage load times more than double the probability of a bounce [7]. Naturally, web service providers strive to cut server-side delays—the only delays they can control—to improve the end-to-end performance of each web request. Following this conventional wisdom, a rich literature has developed around reducing web service delays (*e.g.*, [21, 26, 28, 32, 45, 47, 49]).

Our work is driven by a simple observation: although reducing server-side delay generally improves QoE, the amount of QoE improvement varies greatly depending on the *external delay* of each web request, *i.e.*, the total delay experienced prior to arriving at the web service due to ISP routing, last-mile connectivity, and so forth. In other words, if we define *QoE sensitivity* as the amount QoE would improve if the server-side delay were reduced to zero, there is substantial heterogeneity in QoE sensitivity across users. This heterogeneity results from two empirical findings. First, as illustrated in Figure 1(a), QoE typically decreases along a sigmoid-like curve as delay increases. When the external delay is very short or very long (*e.g.*, A or C on the curve), QoE tends to be less sensitive to the server-side delay than when the external delay is in the middle (*e.g.*, B on the curve). We verified this trend using traces from Microsoft's cloud-scale production web framework, as well as a user study we ran on Amazon MTurk to derive QoE curves for several popular websites (§2.2).

Second, external delays are inherently diverse across user requests to the same web service, due to factors that are beyond the

control of the web service provider: *e.g.,* ISP routing, last-mile connectivity, DNS lookups, and client-side (browser) rendering and processing. Our analysis of our traces reveals substantial variability in external delays even among requests received by the same frontend web server, for the same web content (§2.2).

The heterogeneity in QoE sensitivity implies that following the conventional wisdom of minimizing server-side delays uniformly across all requests can be inefficient, because resources may be used to optimize requests that are not sensitive to this delay. Instead, we should reallocate these resources to requests whose QoE *is* sensitive to server-side delay.

At a high level, user heterogeneity is inherent to the Internet's loosely federated architecture, where different systems are connected together functionally (client devices, ISPs, cloud providers, etc.), but delay optimization is handled separately by each system. Our work does not advocate against this federated architecture; rather, we argue that web service providers should *embrace the heterogeneity of QoE sensitivity across users* to better allocate server-side resources to optimize QoE. Using our traces, we show that if we could reshuffle server-side delays among concurrent requests so that requests with more sensitive QoE get lower server-side delays, we could increase the average duration of user engagement (a measure of QoE) by 28% (§2.3).

To explore the opportunities of leveraging user heterogeneity, we present *E2E*, a resource allocation system for web services that optimizes QoE by allocating resources based on each user's sensitivity to server-side delay.[1] E2E can be used by any shared-resource service; for example it can be used for replica selection in a distributed database to route sensitive requests to lighter-loaded replicas.

The key conceptual challenge behind E2E is that, unlike static properties of a request (*e.g.,* basic vs. premium subscription, wireless vs. wired connectivity), one cannot determine the QoE sensitivity of an arriving request based solely on its external delay. Instead, QoE sensitivity depends on the server-side delay as well. As we show in §3.2, if the server-side delay is large enough, it could cause a seemingly less sensitive request (*A*) to suffer more QoE degradation than a seemingly more sensitive request (*B*). Thus, one cannot prioritize the allocation of resources without taking into account both the external delay distribution and the server-side delay distribution. The latter distribution, in turn, is affected by the resource allocation itself, which makes the problem circular and computationally expensive to solve at the timescale of a web serving system.

E2E addresses this challenge from both the algorithmic perspective and the systems perspective. From the algorithmic perspective, E2E decouples the resource allocation problem into two subproblems, each of which can be solved efficiently: (1) a workload allocation process, which determines the server-side delay distribution without considering QoE sensitivity; and (2) a delay assignment process, which uses graph matching to "assign" the server-side delays to individual requests in proportion to their QoE sensitivity. E2E solves the two subproblems iteratively until it finds the best workload allocation and delay assignment (§4).

From the systems perspective, E2E further reduces the cost of processing each request by coarsening the timescale and the granularity of resource allocation decisions. Observing that the optimal

	Page Type 1	Page Type 2	Page Type 3
Page loads (K)	682.6	314.1	600.2
Web sessions (K)	564.8	265.7	512.2
Unique URLs (K)	3.8	1.5	3.2
Unique users (K)	521.5	264.2	481.8

Table 1: *Dataset summary (date: 02/20/2018)*

allocation is insensitive to small perturbations in the external delay and server-side delay distributions, we allow the system to cache allocation decisions in a lookup table and only update them when a significant change is detected in either distribution (§5).

We demonstrate the practicality of E2E by integrating it into two open-source systems to make them QoE-aware: replica selection in a distributed database (Cassandra) and message scheduling in a message broker (RabbitMQ) (§6). We use a trace-driven evaluation and our testbed deployments to show that (1) E2E can improve QoE (*e.g.,* duration of user engagement) by 28%, or serve 40% more concurrent requests without any drop in QoE; and (2) E2E incurs negligible (4.2%) system overhead and less than $100\mu s$ delay (§7).

This paper focuses on applying E2E to an individual service, or to multiple services that serve unrelated requests. In a production web framework, it is often the case that multiple backend services work together to complete the same (high-level) web request. Focusing on individual backend services allows us to develop our key idea of prioritizing requests based on how sensitive their QoE is to server-side delays, without the added complexity introduced by dependencies across services. We discuss these issues in §9.

2 MOTIVATION

We first use our traces to show the prevalence of heterogeneity in how server-side delays impact the QoE of different users (§2.2). Then, we analyze the potential QoE improvement that could be attained by exploiting this heterogeneity for server-side resource allocation (§2.3).

2.1 Dataset

Our dataset consists of the traces of all web requests served by a production web framework cluster during one day in February 2018. The cluster is one of several located in an Eastern US datacenter serving the major websites and online storefront properties of Microsoft.[2] Importantly, the traces include both client-side (browser) and server-side event logs: the client-side logs record all page rendering events and issued requests, while the server-side logs record all backend processing operations required to fulfill each request. Overall, the dataset spans 1.17M unique users and 1.6M page load events, as summarized in Table 1.

For each web request, we define three delay metrics, shown visually in Figure 2:

- The *total delay* (also known as page load time) is the duration between when a user clicks a link that issues the request and when the last object associated with the request is rendered.

- The *server-side delay* is the time to process all server-side operations on the backend, which may involve multiple steps, such as fetching product IDs from a database and then querying a product catalog for HTML description snippets, before aggregating the results and sending them to the user.

---

[1]E2E takes an "end-to-end" view of web request delays.

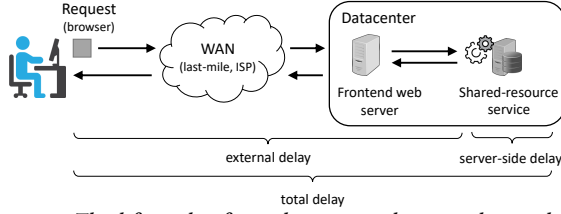[2]Examples include: microsoft, xbox.com, msn.com, etc..

Figure 2: *The life cycle of a web request, showing the total delay, server-side delay, and external delay.*

- The *external delay* includes all delays beyond the purview of server-side operations, *e.g.,* transferring data over the wide-area network, routing the request to the service, decoding and rendering the response in the client-side browser, etc..

  We measure these delay metrics for each web request using the timestamps recorded in our traces. The total delay is measured by the difference between the first and the last timestamps associated with the request. The server-side delay is measured by the total delay of all backend operations (with overlapping delays excluded). As mentioned above, we assume there is a single backend service; we discuss complex dependencies between backend services in §9. Finally, the external delay of a web request is calculated by subtracting the server-side delay from the total delay; it includes both wide-area network and datacenter delays, as shown in Figure 2. Note that this estimate of external delay is conservative because the actual delay may be smaller if server-side processing overlaps with wide-area transfers or browser rendering—our results improve as server-side delay becomes larger relative to external delay.

2.2 QoE sensitivity and its heterogeneity

Our basic intuition is that the impact of the server-side delay of a request on its QoE, *i.e.,* its QoE sensitivity, varies greatly across users. This follows directly from two observations, which we empirically demonstrate here: the sigmoid-like relationship between QoE and total delay, and the variability in requests' external delays.

Sigmoid-like QoE-delay relationship: Figure 3 shows the QoE-delay relationship of requests to one particular page type. Like prior work, we estimate QoE by "time-on-site", measured as the difference between the start and end timestamps of their web session. A *web session* includes all of the user's engagement on the website, such as subsequent clicks and other interactions, with no period of inactivity greater than 30 minutes. Figure 3 groups the total delays into equal-sized buckets, each with at least 5,000 users, and plots the average QoE of users in each bucket. The key property of this graph is its *sigmoid-like* shape. Initially the total delay is small and the QoE is almost insensitive to any change in delay (the delay is too short for users to perceive); then the QoE starts to drop sharply with slope peaking at around 2,000 ms (this is the region where reducing total delay makes a difference); finally, when the total delay exceeds about 5,800 ms, the QoE becomes insensitive again (the delay is long enough that a little additional delay, while noticeable, does not substantially affect QoE). Accordingly, we can roughly categorize all user requests into three sensitivity classes:

- *Too-fast-to-matter* (left blue-shaded area): QoE is not sensitive to server-side delay if total delay is below 2000 ms.
- *Sensitive* (middle orange-shaded area): QoE is sensitive to server-side delay when total delay is between 2000 ms and 5,800 ms.
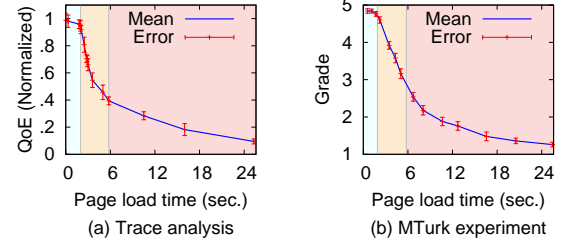

Figure 3: *We observe a non-linear relationship between QoE and total delay (a), so reducing delay by the same amount can have a dramatically different impact on QoE. We highlight different sensitivity regions with different colors. The same QoE-delay relationship is observed in our MTurk-based user study (b).*

- *Too-slow-to-matter* (right red-shaded area): QoE is not sensitive to server-side delay if total delay exceeds 5,800 ms.

  The sigmoid-like curve may look similar to deadline-driven utility curves commonly used in prior work (*e.g.,* [21, 41]), but there is a difference. Traditionally, a service deadline is set where the QoE starts to drop. But our analysis shows that when the total delay exceeds any threshold, the QoE does not drop to zero immediately, and instead decreases gradually as total delay increases. As we will see in §7.4, this difference can cause deadline-driven schemes to have suboptimal QoE.

  We acknowledge that time-on-site may not always reflect how satisfied users are with the web loading experience. Therefore, we complement the above analysis with an IRB approved user study[3] on Amazon MTurk [1]. We describe the detailed setup in Appendix B and only give a summary here. Following similar work in the crowdsourcing literature [48], we asked participants to watch a web page load with different total delays and then to rate their experience on a scale of 1-5. The total delays were randomly permuted per user to avoid any bias due to ordering. We ran this user study on the same web page as in Figure 3(a) and plot the resulting QoE curve in Figure 3(b). As the figure shows, the curve from the user study shares the same sigmoid-like shape as the curve from our trace analysis. We also repeated the user study on four other popular websites; all websites yielded similar sigmoid-like QoE curves, though the boundaries of the three sensitivity regions vary slightly across the sites.

  Although our observations about the QoE-delay relationship do not seem different from prior work (*e.g.,* [14, 22]), they have deeper implications when combined with the next empirical observation on the variability of external delays.

Variability in external delays: The sigmoid-like relationship between QoE and delay means that the sensitivity of QoE to server-side delay depends heavily on the external delay. Figure 4 shows the distribution of external delays among requests for the same web page received at the same frontend web cluster. We see a substantial fraction of requests in each of the three sensitivity classes (25% too-fast-to-matter, 50% sensitive, 25% too-slow-to-matter). The same kind of distribution holds across web pages and is stable over time in our traces.[4] Note that the variance in Figure 4 is unlikely due

---

[3]Our study was approved by U. Chicago, IRB18-1096. It does not raise ethical issues.
[4]The total delay distributions in our traces are consistent with those observed in prior work [16], though they may still vary with website type (*e.g.,* online shopping vs. search engine).
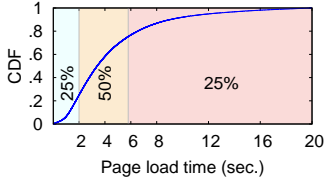
Figure 4: *External delays exhibit great variance even among requests received by the same web server cluster for the same page content.*

to datacenter-level geographical differences, since our traces use a global network of edge proxies to route users from the same region to the same datacenter cluster, although this does not exclude geographical differences users in the same region. It is also unlikely due to application-level differences, since the requests are all targeting the same web page. In practice, a web service provider may see even greater variability in external delays if its edge proxies are less widely distributed than our traces (causing each datacenter cluster to serve a larger geographic region), or if requests are processed by a more centralized architecture (*e.g.*, in many video streaming CDNs [51]).

Since external delays are beyond the control of the web service provider, they are an *inherent* property of the request from the perspective of the service provider. This is in contrast to server-side delays, which the service can influence.

2.3 Potential for improvement

We now use a trace-driven simulation to demonstrate the opportunity of leveraging the heterogeneity of QoE sensitivity to server-side delays. Suppose the dataset has n requests $R = \{r_1, \ldots, r_n\}$, and the server-side delay and external delay of request r_i are s_i and c_i, respectively. Let $Q(\cdot)$ be the QoE function that takes total delay as input and returns the expected QoE. The current QoE of r_i can thus be denoted by $V_i^{old} = Q(s_i + c_i)$. Table 2 summarizes our notation.

Reshuffling server-side delays: Now, let us consider a simple analysis to counterfactually estimate the benefit of allocating resources based on QoE sensitivity. We preserve both the external delay of each request and the collection of server-side delays, but we *re-assign* the server-side delays to requests as follows. We first rank all requests in order of their derivative on the QoE curve, $-\left.\frac{dQ}{dx}\right|_{x=c_i}$, representing the impact on QoE of a small change in server-side delay. Then, we assign the kth-largest server-side delay to the request with the kth-smallest derivative (*i.e.*, the kth-least sensitive request to server-side delay). Let π denote the resulting permutation of server-side delays, *i.e.*, request r_i now has server-side delay $s_{\pi(i)}$. So the new QoE of request r_i is $V_i^{new} = Q(s_{\pi(i)} + c_i)$.

Intuitively, the above re-assignment gives small server-side delays to requests that are sensitive to them, and larger delays to requests that are less sensitive. If the server-side delays s_i are sufficiently small, this assignment can be shown to be optimal, as follows. The average QoE can be written as $\frac{1}{n} \sum_{i=1}^{n} Q(s_{\pi(i)} + c_i) = \frac{1}{n} \sum_{i=1}^{n} s_{\pi(i)} Q'(c_i) + \frac{1}{n} \sum_{i=1}^{n} Q(c_i)$. Suppose the c_i are given and w.l.o.g. $c_1 \leq \cdots \leq c_n$, then this expression is maximized when $s_{\pi(1)} \leq \cdots \leq s_{\pi(n)}$.

Practicality of simulation: To avoid assigning improbable server-side delays to the requests, we first grouped the requests by page type within one-minute time windows, and only re-assigned server-side delays among requests in the same group and 10-second time
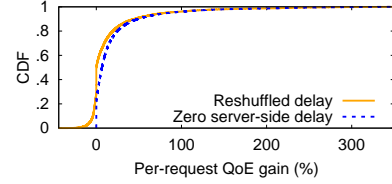

Figure 5: *Potential QoE gains through better allocation of server-side resources based on QoE sensitivity. By reshuffling server-side delays (solid yellow line), we achieve significant QoE gains that are close to the (unrealizable) ideal of zero server-side delays (dashed blue line).*

window. In other words, we do not assign the server-side delay of an off-peak-hour request to a peak-hour request, or the server-side delay of a simple static page request to a complex page request. We also verified that the server-side delay distributions exhibit only negligible changes within a time window. Nonetheless, there are two important caveats. First, our analysis assumes the server-side delays can be arbitrarily re-assigned among requests, which of course is impractical. Second, the analysis uses a very simple algorithm that assumes the set of server-side delays is fixed. In practice, server-side delays are difficult to predict and depend on how resources are allocated to requests. These issues make it challenging to achieve the QoE gains predicted by our simulation; later sections address the issues to extract as much gain as we can manage.

Potential gains in QoE and throughput: Figure 5 shows the distribution of QoE improvements over all requests, *i.e.*, $(Q_i^{new} - Q_i^{old})/Q_i^{old}$, as predicted by our simulation. We see that a small fraction of requests (less than 15.2%) suffer a marginally worse QoE under the new assignment, but a substantial fraction of requests (over 27.8%) see QoE improve by at least 20%. Overall, the new average QoE is 15.4% higher than the old QoE. These improvements are consistent across different page types in the traces. Note that although the new assignment may worsen tail QoE, requests at the tail have such small QoE derivatives that the additional degradation is marginal. We conclude that there is substantial room to improve QoE for a substantial fraction of users, without changing the distribution of server-side delays.

Similarly, we can also support more concurrent requests, *i.e.*, higher throughput, while maintaining a similar level of QoE. To estimate the gain in throughput, we apply our reshuffling of server-side delays to peak hours (higher throughput but worse QoE) and to off-peak hours (lower throughput but better QoE). Figure 6 shows the throughput and QoE during these two periods of time. We randomly select web requests from two peak hours (4pm and 9pm) and three off-peak hours (12am, 3am, 10pm), all in the Eastern Time Zone. For every 10 minutes, we pick the last 10-second window, reshuffle the server-side delays within the time window, and measure the new QoE as above. We can see that the new average QoE during peak hours is similar to (even higher than) the old QoE during off-peak hours. In other words, if we only apply our approach during peak hours, we could support 40% more users without any drop in average QoE.

Now, there are two contributing factors that suggest why these potential gains can be realized over existing systems.

1. *Existing systems are agnostic to user heterogeneity.* Figure 7 shows the distribution of server-side delays in a 10-second window for requests whose external delays fall into different ranges. We see
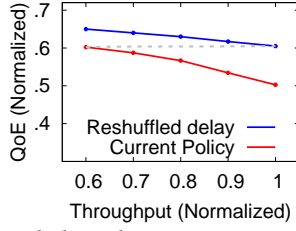
Figure 6: *Potential throughput improvement with similar QoE, achieved by reshuffling server-side delays during peak hours and off-peak hours.*
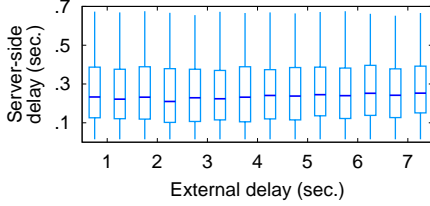

Figure 7: *Current server-side delays are uncorrelated with external delays, showing that the existing resource allocation policy is agnostic to QoE sensitivity. (Candlesticks show $\{5, 25, 50, 75, 95\}$ percentiles.)*
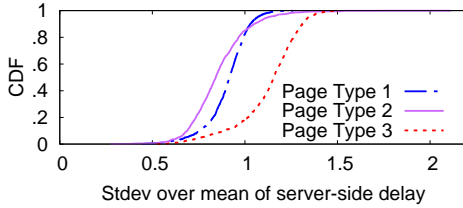

Figure 8: *Server-side delays are highly variable, and not just at the tail. This holds for different page types.*

that there is little correlation between the external delay and the corresponding server-side delay, which suggests that current resource allocation and processing of these requests is agnostic to QoE sensitivity. Our discussions with the Microsoft product teams represented in our traces corroborate this finding.

2. *Server-side delays are highly variable.* Figure 8 shows that there is a substantial variability in server-side delays even among requests for the same page type. Part of this variance is due to tail performance (as observed in prior work), but the lower percentiles also show substantial variance. This variance in server-side delays creates the "wiggle room" that makes the improvements in Figure 5 possible.

2.4 Summary of key observations

The findings in this section can be summarized as follows:

- The variability of external delays across users and the sigmoid-like relationship between QoE and page load time give rise to heterogeneity in the QoE sensitivity of users to server-side delays.
- Our trace-driven simulation shows that by allocating server-side delays based on the QoE sensitivity of each request, one could potentially improve QoE by 20% with the same throughput, or improve throughput by 40% with the same QoE.
- Existing server-side resource allocation is largely agnostic to external delays, while server-side delays exhibit high variance, which together create the opportunity to significantly improve QoE over current schemes.
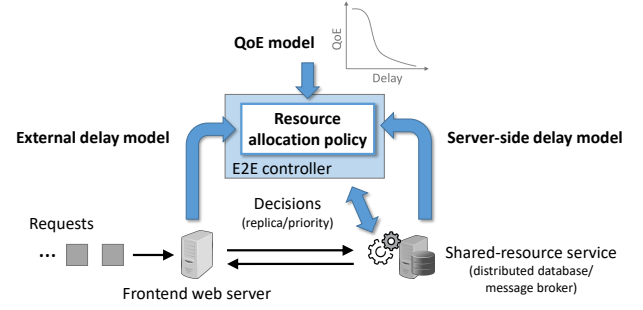

Figure 9: *Overview of E2E.*

3 E2E: OVERVIEW

The next few sections describe E2E, a general resource allocation system for web services that realizes the potential QoE and throughput gains of leveraging user heterogeneity.

3.1 Architecture

Figure 9 illustrates the main components of E2E and how it interacts with a web service system. Typically, a web request is first received by a frontend web server (Figure 9 depicts only one web server, but there may be multiple), which then forwards the request to a backend infrastructure service (*e.g.,* a distributed database or a message broker) whose compute/network resources are shared across requests. E2E provides a resource allocation policy for the shared service that makes a *decision* for each request, *e.g.,* telling it which replica to route the request to in a distributed database, or what priority to assign the request in a message broker. Figure 9 depicts only one shared-resource service, but in general E2E can serve multiple services (or multiple applications within a service) simultaneously, *provided these services do not interact on the same request.* We discuss interrelated services, such as those used to aggregate results for a high-level web request, in §9.

E2E takes as input three variables: an offline-profiled *QoE model* (such as the ones in Figure 3), an *external delay model* from the frontend web servers, and a *server-side delay model* from the shared-resource service. The external delay model provides the distribution of external delays across requests and an estimate of the current request's external delay. This external delay is then tagged as an additional field on the request and on any associated sub-requests (similar to [21]). The server-side delay model provides an estimate of the server-side delay of a request based on the decision and the current workload. Based on these inputs, E2E returns a decision per request for how to allocate resources to it. We discuss how server-side delays and external delays are estimated in §6.

Figure 10 gives two illustrative examples of how E2E might affect resource allocation policies, for a distributed database and a message broker. In particular, E2E can improve the requests' QoE in two ways. First, E2E can assign more QoE-sensitive requests to decisions that have lower server-side delays, *e.g.,* a less loaded replica in a distributed database. Second, E2E can allocate resources to affect the server-side delays, in order to reduce the delays for QoE-sensitive requests. Even if E2E cannot predict server-side delays exactly, it can still create a discrepancy between the delays experienced by requests of different QoE sensitivities. For instance, as illustrated in Figure 10(a), E2E can assign uneven loads across the replicas of a distributed database, so that less loaded replicas are available to process QoE-sensitive requests with faster response times.
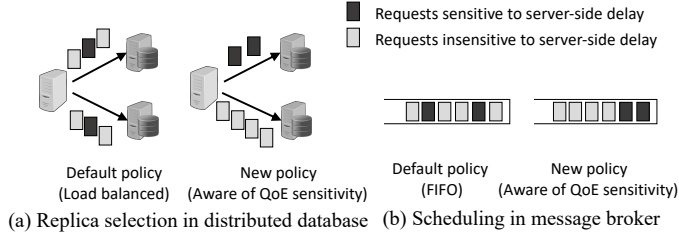
(a) Replica selection in distributed database   (b) Scheduling in message broker

Figure 10: *Examples of how E2E may allocate resources differently in (a) a distributed database and (b) a message broker.*

(a) Scenario 1: $QoE(s_1 \Rightarrow A) + QoE(s_2 \Rightarrow B) > QoE(s_2 \Rightarrow A) + QoE(s_1 \Rightarrow B)$

(b) Scenario 2: $QoE(s_1' \Rightarrow A) + QoE(s_2' \Rightarrow B) < QoE(s_2' \Rightarrow A) + QoE(s_1' \Rightarrow B)$
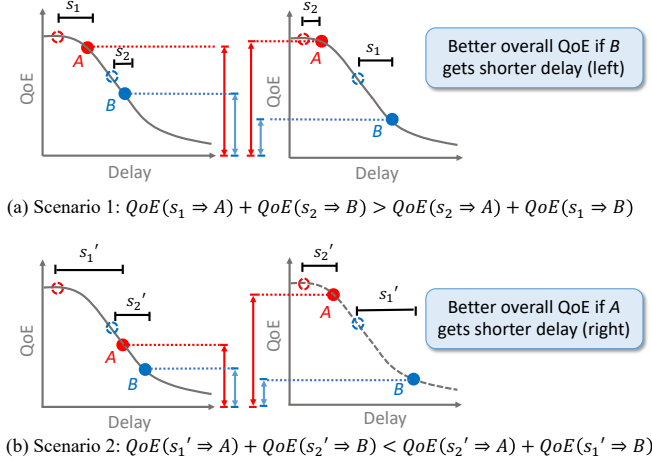
Figure 11: *Illustration of how allocating resources based solely on requests' external delays can lead to suboptimal QoE. Scenarios 1 and 2 have the same pair of requests but different server-side delays. We use the assignment of server-side delays to represent resource allocation. In scenario 1, assigning the shorter server-side delay (s_2) to B and the longer one (s_1) to A leads to better overall QoE. But in scenario 2, giving the shorter delay (s_2') to A leads to worse overall QoE.*

The next two sections present E2E's resource allocation policy and control interface, using the distributed database and message broker as two concrete examples of a shared-resource service. In general, E2E makes very few assumptions about how a shared service processes requests or physically shares its resources; it only requires the service to expose an API for controlling decisions (*e.g.,* the replica to select, the priority of a request, etc.). Also, our work places less emphasis on the prediction of external/server-side delays, or the implementation of a control plane on which E2E's resource allocation policy may run. Existing work already addresses and provides general solutions for these aspects (*e.g.,* [20, 21, 41]).

3.2 Key challenge

The key challenge behind E2E is that the optimal decision for a request *cannot* be determined from the request alone. Instead, the decision depends on the external delay distribution of other requests as well as the server-side delay distribution, which itself is a function of these decisions. Figure 11 illustrates a simple example where prioritizing requests purely based on external delay can lead to a bad decision, and shows how to improve it by taking the server-side delays and other requests' external delays into account. The key observation is that the non-convexity of the QoE-delay curve may cause the sensitivity of a request's QoE to flip depending on the external delay *and* the magnitude of the server-side delay.

Term	Brief description
$r_i; R$	request; vector of requests
$c_i; C$	external delay of r_i; vector of external delays
$s_i; S$	server-side delay of r_i; vector of server-side delays
$Q(\cdot)$	QoE model; $Q(d)$ returns the QoE of total delay d
$z_i; Z$	allocation decision of r_i; vector of decisions
$G(\cdot)$	server-side delay model; $G(Z)$ returns the server-side delay vector of decision vector Z

Table 2: *Summary of terminology*

This property makes it challenging to design a scalable decision-making policy. In particular, the circular dependence between server-side delays and resource allocation decisions makes the problem algorithmically expensive; and the need to account for other request's external delays adds processing overheads.

The above makes E2E conceptually different from many other request scheduling problems where each request has an innate property that indicates its urgency, such as subscription type (*e.g.,* premium vs regular users) or the application's delay sensitivity (*e.g.,* video streaming vs. web pages). Notably, Timecard [41] and DQBarge [21], two closely related systems to ours, use the external delay to directly determine the processing deadline of each request in isolation, without considering other requests or the global impact on available resources (see §8).

4 E2E: DECISION POLICY

This section describes E2E's decision-making policy for allocating resources to requests.

4.1 Problem formulation

We start by formulating the problem of E2E. Table 2 summarizes our terminology. We use r_i, c_i, s_i, z_i to denote the ith request, its external delay, server-side delay, and allocation decision, respectively. Given n concurrent requests $r_1, ..., r_n$ whose external delays $c_i ..., c_n$ are provided by the external delay model, E2E finds the decision vector $Z = (z_1, ..., z_n)$ that maximizes

$$\frac{1}{n} \sum_{i=1}^{n} Q(c_i + G(z_i, Z)),$$

where $Q(d)$ is the QoE of a request with total delay d, as estimated by the QoE model; and $G(z, Z)$ is the server-side delay of a request assigned to decision z given that the complete decision vector is Z, as estimated by the server-side delay model. We assume that the QoE, external delay, and server-side delay models are known and provided as input; we discuss their implementation in §6. For now we assume the server-side delay model $G(\cdot)$ returns precise (noise-free) estimates; we relax this assumption at the end of §4.3.

Unfortunately, solving this problem is computationally hard, because it has to take two dependencies into account:

1. The amount of resource allocated by z_i to a request i depends on how much impact the resource would have on the request's QoE. But this impact is not linear: as more resources are given to the request, the improvement to its QoE may increase or diminish (since Q is non-linear with respect to server-side delay $G(z_i)$).

2. The resource allocation among a set requests depends on the server-side delay distribution, which is itself a function of the resource allocation.
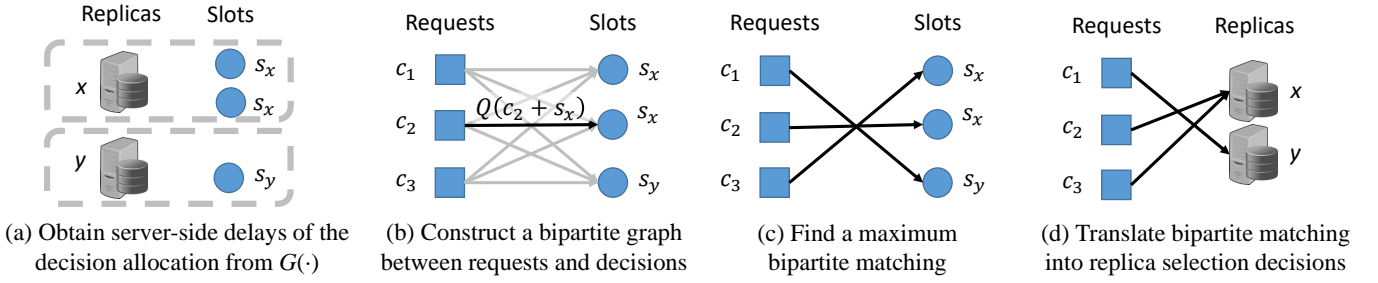
Figure 12: *Running our request-decision mapping algorithm on an example replica selection scenario with three requests (c_1, c_2, c_3) and two replicas (x, y). The given decision allocation is two requests for replica x and one request for the replica y. The final request-decision assignment is optimal for the decision allocation if and only if the corresponding bipartite matching is maximum.*

Mathematically, this problem is NP-hard; the proof is beyond the scope of this paper (readers can refer to [46]), but the basic hardness lies in the non-convexity of function Q.

4.2 Two-level decision-making policy

Our approach to addressing the above intractability is to *decouple* the problem into two levels, as shown in Algorithm 1. The bottom level finds the best *request-decision mapping* for a given decision allocation, where a *decision allocation* is the number of requests assigned with each possible decision (*e.g.,* in a distributed database the possible decisions are the different replicas). The top level uses a simple hill-climbing search to try different decision allocations, find the best request-decision mapping for each allocation (by invoking the bottom level), and repeating until a decision allocation with the best QoE is found. The rationale behind this search strategy is that requests are functionally identical, so the server-side delay model depends only on the decision allocation—*e.g.,* the number of requests assigned to each replica, not which specific requests are assigned—allowing us to drastically reduce the search space from all possible resource allocations to all possible decision allocations. Since the number of possible decisions is typically small (*e.g.,* the number of replicas or priority levels), this is a large savings.

On the other hand, finding the best request-decision mapping for a given decision allocation can be done optimally and efficiently, by viewing it as a graph matching problem. We present the details of this algorithm next.

4.3 Request-decision mapping algorithm

For a given decision allocation, we compute the optimal assignment of requests to decisions by following a four-step process, illustrated in Figure 12 through the example of a replica selection scenario:

1. **Figure 12(a)**: Create n "slots" corresponding to the decision allocation and obtain their server-side delays from $G(\cdot)$. In this case there are three slots, two for replica x and one for replica y, with server-side delays s_x, s_x, s_y.

2. **Figure 12(b)**: Construct an n-to-n bipartite graph where nodes on the left are requests and nodes on the right are slots, and the weight of the edge from request r_i to slot s is $Q(c_i + s)$, *i.e.,* the expected QoE of the request if assigned with this decision.

3. **Figure 12(c)**: Find a maximum bipartite matching, *i.e.,* a subgraph where each node has exactly one edge and the total weight is maximized.

4. **Figure 12(d)**: Translate the matching to a request-decision assignment: each request is assigned the decision corresponding

---

Algorithm 1: E2E's two-level decision-making policy.

Input: 1) A vector of n requests $(r_1, \ldots, r_n)$,
2) external delay of r_i is c_i,
3) Number of possible decisions k
Output: Decision vector $Z = (z_1, \ldots, z_n)$, z_i is decision of r_i

```
/* Initialize decision allocation                        */
```
1   $(n, 0, \ldots, 0) \rightarrow W$
```
/* Find the best decisions and QoE under W              */
```
2   RequestDecisionMappingAlgorithm(W) $\rightarrow Z$
3   $\sum_i Q(c_i + G(z_i, Z)) \rightarrow q$
4   **while** HillClimbing(W) $\rightarrow W' \neq \phi$ **do**
5   RequestDecisionMappingAlgorithm(W') $\rightarrow Z'$
6      $\sum_i Q(c_i + G(z_i', Z')) \rightarrow q'$
```
    /* Update Z if hillclimbing step improves QoE        */
```
7   **if** $Q' > Q$ **then**
8          $Z' \rightarrow Z, q' \rightarrow q$

---

to the slot it is linked to. In this example the final decisions are: $c_2 \rightarrow x, c_3 \rightarrow x, c_1 \rightarrow y$.

The key insight is to cast the problem of maximizing the QoE of a request-decision mapping to that of maximizing a matching in a bipartite graph, for which polynomial-time algorithms exist [24, 30]. The polynomial is cubic in the number of requests, so care must be taken to ensure an efficient implementation; this is addressed in §5.

In practice, the server-side delay model $G(\cdot)$ estimates a distribution of the server-side delay, not an exact value, so the request-decision mapping algorithm (Figure 12) needs to be modified as follows. Instead of labeling each slot with a fixed value in Figure 12(a) (*e.g.,* s_x), we label it with a probability distribution $f_x(s)$ (provided by $G(\cdot)$), and label the edge in Figure 12(b) between request r_i and the slot with the *expected* QoE over this distribution, *i.e.,* $\int_0^\infty Q(c_i + s)f_x(s)\mathrm{d}s$.

5 E2E: DECISION OVERHEAD

E2E's has to make a resource allocation decision for each request, and this decision might change if one or more of the input variables (QoE model, external delay model, server-side delay model) changes. This overhead can quickly become unscalable if left unchecked.

Our idea for reducing the decision-making overhead is to coarsen the granularity of decisions along two dimensions: (1) spatially grouping requests with similar characteristics, and (2) temporally caching decisions that are updated only when a significant change

occurs in the input variables. Although these are heuristics with no formal guarantees, we find that they work well in practice (§7).

Coarsening spatial granularity: We coarsen decisions spatially by grouping requests into a *constant* number of buckets based on their external delays. Specifically, we split the range of external delays into k intervals, and all requests whose external delays fall in the same interval are grouped in the same bucket. We then run E2E's decision-making policy over the buckets rather than individual requests, and assign the same final decision to all requests in a bucket. This coarsening ensures that the running time of the decision-making process is always constant, rather than growing with the cube of the number of requests (the fastest bipartite matching algorithm [24, 30]). To minimize the amount of QoE degradation caused by making decisions at the bucket level, the external delay intervals satisfy two criteria: (1) they evenly split the request population, and (2) the span of any interval does not exceed a predefined threshold δ. Our evaluation shows these criteria are effective.

Coarsening temporal granularity: We have empirically observed that the same decision assignment can yield close-to-optimal QoE even if some of the inputs to E2E's decision-making policy have changed slightly. Therefore, E2E caches its decision assignment in a *decision lookup table* that the shared-resource service can query for every new request. The keys in this table are the buckets of the external delays, and the corresponding value is the decision assigned to each bucket. The exact definition of decisions varies across use cases. For instance, in a distributed database, the decision of a specific external delay bucket is the probability of sending a request to each of the replicas, if the request's external delay falls in the bucket. The lookup table is only updated when one of the input variables has changed by a "significant amount". The policy for deciding this is orthogonal and not something we prescribe; *e.g.*, it could be if the J-S divergence [37] between the new and old distributions exceeds a certain threshold.

Fault tolerance of E2E controller: In E2E, a request needs to wait for its resource allocation decision from the E2E controller, which can therefore become a single point of failure for the whole system. This can be mitigated in three ways. First, if the E2E controller fails, the shared-resource service can still make QoE-aware decisions by looking up the request's external delay in the most recently cached decision lookup table (see above). Second, the E2E controller is replicated with the same input state (QoE model, external delay model, server-side delay model), so when the primary controller fails, a secondary controller can take over using standard leader election [15, 27]. Finally, in the case of total E2E failure, the shared-resource service can simply bypass E2E and use its default resource allocation policy.

6 USE CASES

We demonstrate E2E's practical usefulness by integrating it into two popular web infrastructure services, depicted in Figure 13: replica selection in a distributed database and message scheduling in a message broker. In both cases, E2E makes minimal changes to the shared-resource service and only relies on the control interface exposed by them. We evaluate E2E's overhead in §7.3.

Use case #1: Distributed database. We choose Cassandra [2] as the distributed database, and use E2E to select the replica for each request (this operation is common to other distributed databases, and
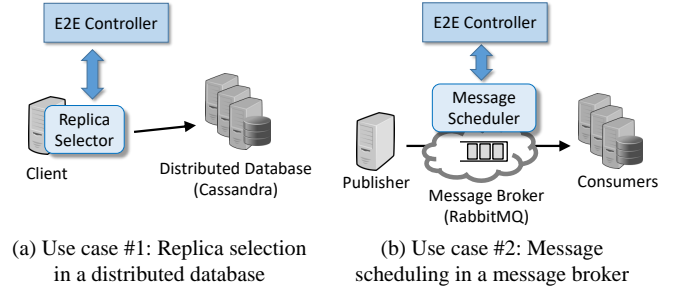

(a) Use case #1: Replica selection in a distributed database

(b) Use case #2: Message scheduling in a message broker

Figure 13: *Use cases of E2E*

not specific to Cassandra). In particular, we made two changes. First, we modified the existing replica selection logic (`getReadExecutor` of `ReadExecutor`) of the Cassandra client. Our new logic stores the decision lookup table (§5) received from the E2E controller in a local data structure. When a new request arrives, it looks up the request's external delay in the table to get the selected replica's IP. Second, we modified the client service callback function (in `RequestHandler`) to keep track of the load (number of concurrent requests) and the observed (server-side) delay of each replica. In practice, the replication level, *i.e.,* the number of replicas for each key, is usually much smaller than the total number of servers. A simple replication strategy, adopted by Cassandra and other databases like MongoDB [3], is to divide the servers into replica groups and store a copy of the entire database in each group. This replication strategy is a good fit for E2E, which now simply has to choose a replica group for each incoming request. It also allows E2E to affect server-side delays by ensuring that some replica groups are less loaded and used to process QoE-sensitive requests.

Use case #2: Message broker. We choose RabbitMQ [4] as the message broker (other message brokers can work with E2E in a similar way). RabbitMQ manages its resource by using priority queues and associating each request with a priority level. Requests with high priority are served before requests with low priority. Similar to the Cassandra implementation, we made two changes to integrate E2E. First, we wrote the E2E controller logic in a python script and pass it to RabbitMQ as the default scheduling policy (through `queue_bind`) when the RabbitMQ service is initialized. Second, we modified the per-request callback function (`confirm_delivery`) to track each request's progress and the queueing delay in the message broker.

Implementation details: E2E requires three models as input in order to run. We describe our realizations of these models below, though other approaches are certainly possible.

- *QoE model:* Our E2E prototype uses the QoE models derived from the Microsoft traces and our MTurk user study, shown in Figure 3 and detailed in Appendix B. The QoE model needs to be updated only when the web service changes its content substantially; we do not update it in our prototype.

- *External delay model:* Our E2E prototype builds the external delay distribution from per-request external delay measurements in recent history. The external delays are currently provided by our traces and are not calculated in real-time for each request, though the latter is necessary in a production deployment (see §9). We use batched updates to reduce the overhead of keeping the distribution up-to-date. Specifically, we found in our traces

that it is sufficient to update the external delay distribution every 10 seconds, because a 10-second time window usually provides enough requests to reliably estimate the distribution, and the distribution remains stable within this window.

- *Server-side delay model:* Our prototype builds the server-side delay model offline, by measuring the service delay distributions induced by different resource allocations. For instance, to build a server-side delay model for the distributed database, we measure the processing delays of one server under different input loads: {5%, 10%,..., 100%} of the maximum number of requests per second. For the message broker the profiling is slightly more complicated: we have to consider both the number of requests at each priority level and the total number of requests at higher priority levels. In practice we need not profile all possible allocations: it is sufficient to sample some of them and extrapolate the others. Also, the requests are homogeneous in both of our uses cases, as is typically the case in web services. For services that serve heterogeneous requests (*e.g.,* both CPU-intensive and memory-intensive jobs), or where the effects of different resource allocations do not easily extrapolate to each other, more advanced techniques may be required to ensure the profiling is efficient.

7 EVALUATION

We evaluate E2E using a combination of trace-driven simulations and real testbed experiments. Our key findings are:

- *E2E can substantially improve QoE:* Users spend 11.9% more web session time (more engagement) compared to the default resource allocation policy in our traces; this improvement accounts for 77% of the best-possible improvement if server-side delays were zero. (§7.2)
- *E2E has low system overhead:* E2E incurs only 0.15% additional server-side delay and requires 4.2% more compute resources per request. (§7.3)
- *E2E can tolerate moderate estimation errors* (up to 20%) on the external delays, while still retaining over 90% of the QoE improvement attainable if there are no errors. (§7.4)

7.1 Methodology

Both our trace-driven simulator and our testbeds use the external delay model derived from our traces (Table 1) and the QoE model from Figure 3. The simulator is described in more detail in §2.3.

Testbed setup: To complement our trace-driven simulations, which unrealistically assume the server-side delay distribution is fixed, we create two real testbeds on Emulab—one for Cassandra and one for RabbitMQ, as described in §6. We feed requests from our traces to each testbed in chronological order with their recorded external delays, and use the actual testbed processing time as the server-side delays. To show the impact of system load, we speed up the replay by reducing the interval between two consecutive requests by a *speedup ratio* (*e.g.,* a speed-up ratio of 2 means we halve the interval between every two consecutive requests). In the Cassandra (distributed database) testbed, each request is a range query for 100 rows in a table of 5 million keys, which are replicated to three replicas (three Emulab nodes), so each replica has a copy of each key. The key size is 70B and the value size is 1KB. In the RabbitMQ (messaging broker) testbed, each request is a 1KB message sent to RabbitMQ (one Emulab node), and a consumer pulls a message

from RabbitMQ every 5ms. Each Emulab node has one 3.0GHz Intel Xeon processor, 2GB RAM, and 2x146GB HDD storage, and are connected to each other by a 1Gbps Ethernet link.

We do not claim that this testbed is a faithful replication of the production system that generated our traces. Rather, we use the testbeds to allow resource allocation policies to affect the server-side delay distributions, as opposed to being constrained by the fixed server-side delays in our traces. We use the traces only to reflect the real external delays of users issuing requests to a service.

Baselines: We compare E2E against two baseline policies:

- *Default policy* (unaware of the heterogeneity of QoE sensitivity): In the simulator, it simply gives each request its recorded server-side delay. In RabbitMQ, it uses First-In-First-Out (FIFO) queueing. In Cassandra, it balances load perfectly across replicas.
- *Slope-based policy* (aware of the heterogeneity of QoE sensitivity but suffers from the problem described in §3.2): In the simulator, it gives the shortest server-side delay to the request whose external delay has the steepest slope in the QoE model, and so forth (see §2.3). In RabbitMQ, it gives the highest priority to the request whose external delay has the steepest slope in the QoE model, and so forth. In Cassandra, it is the same as E2E's policy, except it replaces the request-decision mapping algorithm with the slope-based algorithm above.

Metric of QoE gain: We measure the *QoE gain* of E2E (and its variants) by the relative improvement of their average QoE over that of the default policy, *i.e.,* $(Q_{E2E} - Q_{default})/(Q_{default})$.

7.2 End-to-end evaluation

Overall QoE gains: Figure 14 compares the QoE gains of E2E and the slope-based policy over the existing default policy, in our traces and our testbeds. For page types 1 and 2 we use time-on-site as the QoE metric (with Figure 3(a) as the QoE model), and for page type 3 we use user rating as the QoE metric (with Figure 3(b) as the QoE model). Using user rating vs. time-on-site has negligible impact on our conclusions, as they lead to very similar QoE models (Figure 3).

Figure 14(a) shows that in our traces, E2E achieves 12.6–15.4% better average QoE than the default policy, whereas the slope-based policy has only 4–8% improvement. This suggests that E2E addresses the limitation of the slope-based policy discussed in §3.2. To put these gains into perspective, we consider an idealized policy (labeled "idealized" in the figure) that cuts all server-side delays to zero (*i.e.,* the best a web service could possibly do by cutting server-side delays). We see that the QoE gain of E2E already accounts for 74.1–83.9% of the QoE gain of this idealized policy.

Figure 14(b) also compares the QoE of E2E and the baseline policies when feeding requests of page type 1 to the Cassandra and RabbitMQ testbeds. We used a 20× speedup ratio to sufficiently load the systems (we explore the tradeoff between system load and QoE gain below). The results show similar gains in QoE, with both systems achieving a large fraction of the best possible gains.

Better QoE-throughput tradeoffs: Figure 15 compares the QoE of E2E and the default policy under different loads, in our traces and our testbeds. E2E strikes a better QoE-throughput tradeoff than both the default policy and the slope-based policy.

Figure 15(a) shows the results for different hours of the day in our traces (12am, 4am, 3pm, 8pm, 10pm all in US Eastern Time), which
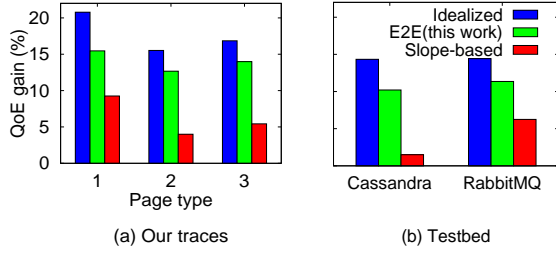
Figure 14: *Overall QoE improvement of E2E and the slope-based policy over the default policy.*
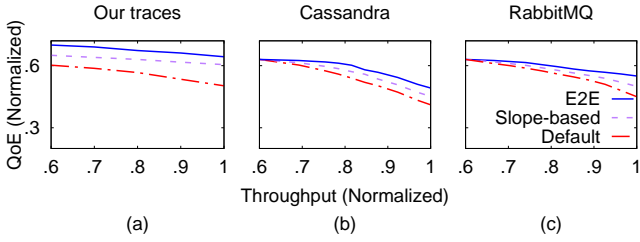

Figure 15: *QoE improvement of E2E under different levels of loads. Throughput is normalized against the highest per-hour throughput (a) and the total testbed capacity (b, c).*

exhibit a natural variation in load. Compared to the off-peak hour (leftmost, at 0.6), the peak hour (rightmost, at 1.0) sees 40% more traffic and, as a result, has 20.1% lower QoE. E2E achieves similar QoE during the peak hour as the default policy does during the off-peak hour. In other words, E2E achieves 40% higher throughput than the default policy without a drop in QoE.

Figures 15(b) and (c) compare the QoE of E2E with those of the baseline policies in our testbeds, while varying the load (speedup ratio 15× to 25×, normalized as 0.6 to 1 throughput). E2E always improves QoE, though to varying degrees. E2E's gain is marginal under low load, since all decisions have similar, good performance (*e.g.,* all replicas have low read latency when Cassandra is underloaded). As the load increases, however, E2E's gain grows rapidly: at system capacity, E2E achieves 25% QoE gain over the default policy. This can be explained as follows (using Cassandra as an example). The default policy (perfect load balancing) drives every replica to a moderately high load, so all requests are affected by bad tail latencies. In contrast, E2E allocates load unevenly so that at least one replica is fast enough to serve the QoE-sensitive requests.

7.3 Microbenchmarks

We examine the overheads incurred by E2E in computing cost, decision delay, and fault tolerance.

System overhead: We compare the total resource consumption of running each testbed with and without E2E. Figure 16 shows the additional overhead of E2E in CPU and RAM usage. We see that the overhead of E2E is several orders of magnitude lower than the total overhead of running the Cassandra or RabbitMQ testbeds themselves. Moreover, the CPU and RAM overheads grow more slowly than those of the testbed service as the load increases.

Decision delay: Figure 17 shows the effectiveness of our two decision delay-reduction optimizations (§5), using the Cassandra testbed (with speedup ratio 20x). We see that (1) spatial coarsening
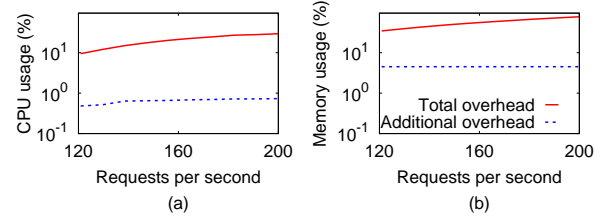

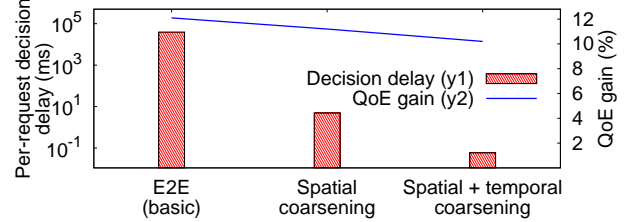**Figure 16:** *The additional overhead of E2E vs. the total overhead of running the testbeds.*

Figure 17: *Per-request delay reduction due to spatial and temporal coarsening (§5).*
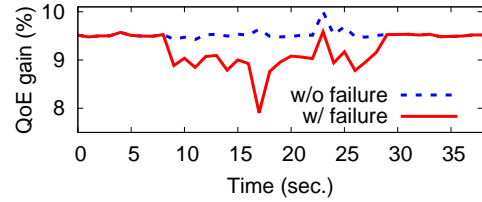

Figure 18: *E2E can tolerate loss of the controller.*

(bucketization of external delays) reduces the decision delay by four orders of magnitude, and (2) temporal coarsening (caching E2E decisions in a lookup table) reduces the decision delay by another two orders of magnitude. The resulting per-request response delay is well below 100μs, less than 0.15% of Cassandra's response delay. At the same time, we see that these reductions in decision-making delay only have a marginal impact on QoE. Note that E2E does not need to make a decision on the arrival of each request, due to these optimizations. Instead, decisions are made periodically and cached in the local memory of each Cassandra client; so when a request arrives, its decision can be read directly from the client's memory.

Fault tolerance: Finally, we stress test our prototype of E2E by disconnecting the E2E controller from the Cassandra testbed. Figure 18 shows a time-series of the QoE gain of requests. We disconnect the controller at the 25th second. First, we see that Cassandra's replica selection still uses the latest E2E's decisions cached in the lookup table, so although the QoE gain drops (as the lookup table becomes stale), it is still better than the default policy. At the 50th second, a backup controller is automatically elected, and by the 75th second, the new controller starts to make the same decisions as if the controller was never disconnected.

7.4 In-depth analysis

Operational regime: Figure 19 tests E2E's performance across a wide range of workloads, along three dimensions that influence E2E's performance. We synthetically generate requests by drawing external delays and server-side delays from two normal distributions, respectively, and test them on the trace-driven simulator
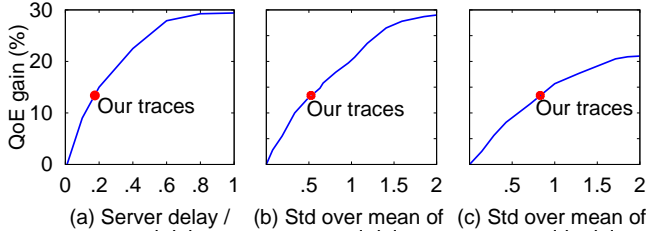
Figure 19: *The impact of three key workload dimensions on E2E's effectiveness. The red spot shows where the workload in our traces lies.*
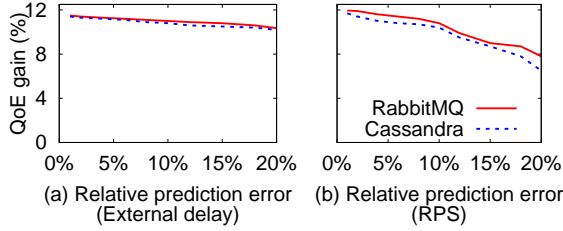

Figure 20: *Sensitivity of QoE improvement to prediction errors in external delay and requests per second.*

using the QoE model from Figure 3. Although the server-side and external delays in our traces do not exactly follow normal distributions, modeling them in this way allows us to test E2E's performance under different distribution conditions. For instance, we can test the impact of increasing the mean of server-side delay on E2E's performance while keeping the external delay distribution fixed.

We set the default mean and variance of each distribution to match those of the page type 1 requests in our traces, and vary one dimension at a time. We see that at the beginning, E2E does not yield any QoE gain, since there is no variability in the external and server-side delays for it to exploit. Then, the QoE gain of E2E starts to grow almost linearly with the server-side/external delay ratio, external delay variance, and server-side delay variance, which confirms that E2E is able to utilize the variance in external and server-side delays. To put this in the perspective of our traces, the workload in our traces is on the "fast-growing" part of all curves (red spots in Figure 19). This means we will see more QoE gain if the workload moves to the right in any of these dimensions.

Robustness to prediction errors: Figure 20 shows the impact that prediction errors, in the external delays and the number of requests per second (RPS), have on E2E's performance. We feed page type 1 requests to the Cassandra testbed (speedup ratio 20x), and inject a controllable error on the actual value to obtain the estimated value. Figure 20(a) shows that even if the external delay prediction is off by 20% on each request, E2E still retains over 90% of its QoE gain. Predicting the external delay with 20% (or 100-200ms) error seems reasonable for most users [41]. Figure 20(b) shows that E2E retains 91% of its QoE gain if the RPS is predicted with 10% error. Empirically, we find that 10% prediction error is possible when using the RPS history from the last 10 seconds (not shown).

QoE fairness: A natural concern is that E2E may create a less fair QoE distribution. As an example, we use the QoE distributions of E2E and the default policy from Figure 14(a) and page type 1. We calculate Jain's Fairness Index of the requests' QoE values, and
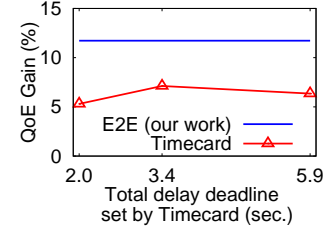

Figure 21: *E2E vs. Timecard (with different total delay deadlines).*

find that E2E's Jain index (0.68) is lower but still very close to that of the default policy (0.70). This is because E2E only deprioritizes requests that are insensitive to QoE; these requests experience only a marginal improvement in QoE when using the default policy.

E2E vs. deadline-driven scheduling: Unlike E2E, some prior work (*e.g.,* [21, 41]) models the impact of total delay on QoE as a hard deadline: QoE drops to zero immediately after the total delay exceeds the deadline. We use Timecard [41] as a canonical example of a deadline-driven scheduling policy, and compare E2E to it. Timecard sets a total delay deadline and, given the external delay of each request, tries to maximizes the number of requests served by the deadline. We compare E2E with Timecard under total delay deadlines of 2.0, 3.4, and 5.9 seconds, using RabbitMQ as the testbed. As Figure 21 shows, the QoE gain of E2E is consistently better than Timecard under different deadline settings. This is because the deadline-driven scheduler is agnostic to the different QoE sensitivities of requests that have already exceeded the deadline.

8 RELATED WORK
We briefly survey the most related work on web QoE, cloud resource allocation, and web performance measurements.

Web QoE modeling/optimization: QoE has been intensively studied in the context of web services (*e.g.,* [10, 22]), mobile apps (*e.g.,* [9]), video streaming (*e.g.,* [11, 23]), and measurement tools (*e.g.,* [48]). Prior work (*e.g.,* [14, 25, 39]) has observed a similar nonlinear relationship between page loading time and QoE. Although E2E uses a specific QoE model (based on our trace analysis), it can benefit from more precise models of how page loading time affects QoE. Unlike prior QoE optimization techniques that tune client-side knobs [16, 38] or provide server-side resources for individual sessions (*e.g.,* [42, 43]), E2E intelligently allocates server-side resources shared across a large number of heterogeneous users.

Web service resource allocation: There is a large literature on cutting the tail/median server-side delays through better web resource management, including distributed databases (*e.g.,* [45, 49]), partition-aggregation workloads (*e.g.,* [28, 32]), and caching (*e.g.,* [12, 13]). Cloud providers optimize WAN latency through better server selection (*e.g.,* [18, 35]) and WAN path selection [44, 50]. E2E is conceptually compatible with many existing resource sharing techniques (*e.g.,* replica selection and message scheduling). What distinguishes E2E is that it does not seek to minimize the median or tail performance; instead, it takes into account the QoE sensitivity of different users when allocating server-side resources.

End-to-end performance analysis: There have been attempts to measure the contribution of cloud, WAN, and client-side devices to end-to-end delays [19, 20, 40]. Our observations on heterogeneous QoE sensitivity corroborate some prior work (*e.g.,* [20]) that show

that cloud-side delays are not a constant fraction of end-to-end delays for all users. These studies offer useful insights for improving web service infrastructure [17, 31, 36] and building real-time resource management systems [8, 21, 41].

The works most closely related to E2E are Timecard [41] and DQBarge [21], which share with us the high-level idea of making server-side decisions based on the QoE of end users [29]. In particular, they estimate the "slack" time between receiving a request and its end-to-end delay deadline, and utilize this slack to maximize the quality of the response. Although they allocate different resources to different requests, they optimize individual requests in isolation, which can cause resource contention when the system is under stress or many requests have low slack time. In contrast, E2E optimizes QoE and resource allocation across requests, by harnessing their inherent heterogeneity. We also empirically show that when the QoE curve is like Figure 3, a deadline-based QoE model can be less effective than E2E (§7.4).

E2E is similar to work (*e.g.,* [33, 34]) that considers requests with soft deadlines: *i.e.,* QoE decreases gradually to zero after the total delay exceeds a time threshold. These soft-deadline-driven schedulers set the same threshold for all requests and do not take the heterogeneity of web requests into account, whereas the resource allocation in E2E considers different QoE sensitivities.

9 DISCUSSION

Incentives of other service providers: One concern about using E2E is that another service provider (*e.g.,* an ISP) may try to manipulate the external delays of its users to get better service from E2E, by making them look more urgent. However, we prove in Appendix A that *it is impossible to improve a group of users' QoE without reducing at least some of their external delays.* In other words, E2E creates an incentive for other service providers to reduce their delays, rather than gaming E2E by deliberately adding delays.

Security threat: In theory, E2E may introduce a new attack, in which a large group of users hurt the QoE of other users by making themselves look more urgent, thus starving the other users of resources (similar to a Denial-of-Service attack). We can envision several detection/mitigation techniques for such an attack, such as detecting abnormal changes to the external delay distribution, or adding randomization to the actual server-side delays. We leave investigation of these security issues to future work.

Interaction with existing policies: A web service provider often integrate multiple resource allocation policies. Conceptually, E2E is compatible with other prioritization schemes; they can be included as input into E2E's decision-making policy (*e.g.,* by upweighting the $Q(\cdot)$ values of premium traffic), or E2E can be applied separately to each priority class (*e.g.,* premium users vs. basic users).

Complex request structures: In a real web framework like Microsoft's, a high-level web request usually results in calls to multiple backend services, and the request is not complete until it hears a response from all the backend services [12]. A straightforward way to handle this request structure is to apply E2E to each service in isolation. However, this approach is suboptimal, because it may cause a service to prioritize requests whose server-side delays are determined by other backend services. For example, in Figure 11(a), E2E prioritizes request B over A, since prioritizing A would cause B to suffer a significant QoE drop. But if B also depends on another,

much slower service, speeding up B will not have a direct impact on the user's QoE. In this case, it would have been better to prioritize A, whose QoE could actually have been improved. We can see that an optimal resource allocation scheme for requests with complex structure needs to take these backend service dependencies into account. We leave this problem to future work.

Deployment at scale: E2E must face the following issues when deployed in a large-scale production system.

- *Multiple agents:* For a web service to scale, it typically uses distributed agents (*e.g.,* Cassandra clients or RabbitMQ message brokers), each making resource-allocation decisions independently. In E2E, although each agent might see a different subset of web requests, its decisions are based on a global decision lookup table built upon the global external delay distribution. In the unlikely event that the requests are load balanced poorly across the agents, it is possible for the resulting decisions to be suboptimal: *e.g.,* in the case of RabbitMQ, if one message broker only sees insensitive requests, those requests will be at the head of its queue (there are no sensitive requests to place ahead of them). We have not investigated such scenarios in our current evaluation.

- *Real-time external delay estimation:* Our current prototype relies on the external delays provided by our traces, but a real deployment would need to compute the external delay in real-time for each request. E2E could accomplish this by borrowing ideas from Timecard [41] and Mystery Machine [20]. Like Timecard, the WAN-induced delay of a request could be derived from the round-trip time of the TCP handshake packets and the TCP sliding window size. To estimate the browser rendering time of a request, E2E could use a model trained on historical traces (Mystery Machine) or on traces and the system configuration (Timecard). Timecard provides more accurate estimates but requires user permission to access the system configuration. Mystery Machine does not need user cooperation but has lower accuracy, especially for first-time users. Since E2E is not very sensitive to the accuracy of the external delay estimates (Figure 20(a)), Mystery Machine's method could allow E2E to scale out and support more requests.

10 CONCLUSION

We have described E2E, a resource allocation system that optimizes QoE by exploiting user heterogeneity. E2E gives any shared-resource service an end-to-end view of request delays, allowing it to prioritize the handling of these requests based on how sensitive their QoE is to delay. E2E can be used by multiple services if the services do not interact on the same request. As we have discussed, many web frameworks coordinate multiple services to complete a single (high-level) request. Extending E2E to handle such complex request dependencies is our primary direction of future work.

REFERENCES

[1] Amazon Mechanical Turk. https://www.mturk.com/.
[2] Apache Cassandra. https://cassandra.apache.org.
[3] MongoDB. https://www.mongodb.com/.
[4] RabbitMQ. https://www.rabbitmq.com/.
[5] Marissa Mayer at Web 2.0. http://glinden.blogspot.com/2006/11/marissa-mayer-at-web-20.html, 2006.
[6] Akamai online retail performance report: Milliseconds are critical. https://www.akamai.com/uk/en/about/news/press/2017-press/akamai-releases-spring-2017-state-of-online-retail-performance-report.jsp, 2017.
[7] Find out how you stack up to new industry benchmarks for mobile page speed. https://www.thinkwithgoogle.com/marketing-resources/data-measurement/mobile-page-speed-new-industry-benchmarks/, 2018.
[8] Victor Agababov, Michael Buettner, Victor Chudnovsky, Mark Cogan, Ben Greenstein, Shane McDaniel, Michael Piatek, Colin Scott, Matt Welsh, and Bolian Yin. Flywheel: Google's data compression proxy for the mobile web. In *NSDI*, volume 15, pages 367–380, 2015.
[9] Vaneet Aggarwal, Emir Halepovic, Jeffrey Pang, Shobha Venkataraman, and He Yan. Prometheus: Toward quality-of-experience estimation for mobile apps from passive network measurements. In *Proceedings of the 15th Workshop on Mobile Computing Systems and Applications*, page 18, 2014.
[10] Athula Balachandran, Vaneet Aggarwal, Emir Halepovic, Jeffrey Pang, Srinivasan Seshan, Shobha Venkataraman, and He Yan. Modeling web quality-of-experience on cellular networks. In *Proceedings of the 20th annual international conference on Mobile computing and networking*, pages 213–224, 2014.
[11] Athula Balachandran, Vyas Sekar, Aditya Akella, Srinivasan Seshan, Ion Stoica, and Hui Zhang. Developing a predictive model of quality of experience for internet video. In *ACM SIGCOMM Computer Communication Review*, volume 43, pages 339–350. ACM, 2013.
[12] Daniel S Berger, Benjamin Berg, Timothy Zhu, Siddhartha Sen, and Mor Harchol-Balter. Robinhood: Tail latency aware caching–dynamic reallocation from cache-rich to cache-poor. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, pages 195–212, 2018.
[13] Daniel S Berger, Ramesh K Sitaraman, and Mor Harchol-Balter. Adaptsize: Orchestrating the hot object memory cache in a content delivery network. In *NSDI*, pages 483–498, 2017.
[14] Enrico Bocchi, Luca De Cicco, and Dario Rossi. Measuring the quality of experience of web users. *ACM SIGCOMM Computer Communication Review*, 46(4):8–13, 2016.
[15] Mike Burrows. The Chubby lock service for loosely-coupled distributed systems. In *OSDI*, 2006.
[16] Michael Butkiewicz, Daimeng Wang, Zhe Wu, Harsha V Madhyastha, and Vyas Sekar. Klotski: Reprioritizing web content to improve user experience on mobile devices. In *NSDI*, volume 1, pages 2–3, 2015.
[17] Matt Calder, Manuel Schröder, Ryan Stewart Ryan Gao, Jitendra Padhye, Ratul Mahajan, Ganesh Ananthanarayanan, and Ethan Katz-Bassett. Odin: Microsoft's scalable fault-tolerant CDN measurement system. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, 2018.
[18] Fangfei Chen, Ramesh K Sitaraman, and Marcelo Torres. End-user mapping: Next generation request routing for content delivery. In *ACM SIGCOMM Computer Communication Review*, volume 45, pages 167–181. ACM, 2015.
[19] Yingying Chen, Ratul Mahajan, Baskar Sridharan, and Zhi-Li Zhang. A provider-side view of web search response time. In *ACM SIGCOMM Computer Communication Review*, volume 43, pages 243–254, 2013.
[20] Michael Chow, David Meisner, Jason Flinn, Daniel Peek, and Thomas F Wenisch. The mystery machine: End-to-end performance analysis of large-scale internet services. In *OSDI*, pages 217–231, 2014.
[21] Michael Chow, Kaushik Veeraraghavan, Michael J Cafarella, and Jason Flinn. DQBarge: Improving data-quality tradeoffs in large-scale internet services. In *OSDI*, pages 771–786, 2016.
[22] Diego Neves da Hora, Alemnew Sheferaw Asrese, Vassilis Christophides, Renata Teixeira, and Dario Rossi. Narrowing the gap between QoS metrics and web QoE using above-the-fold metrics. In *International Conference on Passive and Active Network Measurement*, pages 31–43. Springer, 2018.
[23] Florin Dobrian, Vyas Sekar, Asad Awan, Ion Stoica, Dilip Joseph, Aditya Ganjam, Jibin Zhan, and Hui Zhang. Understanding the impact of video quality on user engagement. In *ACM SIGCOMM Computer Communication Review*, volume 41, pages 362–373, 2011.
[24] Michael L Fredman and Robert Endre Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM (JACM)*, 34(3):596–615, 1987.
[25] Qingzhu Gao, Prasenjit Dey, and Parvez Ahammad. Perceived performance of top retail webpages in the wild: Insights from large-scale crowdsourcing of above-the-fold QoE. In *Proceedings of the Workshop on QoE-based Analysis and Management of Data Communication Networks*, pages 13–18, 2017.

[26] Mingzhe Hao, Huaicheng Li, Michael Hao Tong, Chrisma Pakha, Riza O Suminto, Cesar A Stuardo, Andrew A Chien, and Haryadi S Gunawi. MittOS: Supporting millisecond tail tolerance with fast rejecting SLO-aware OS interface. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 168–183. ACM, 2017.
[27] Patrick Hunt, Mahadev Konar, Flavio Paiva Junqueira, and Benjamin Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *USENIX ATC*, 2010.
[28] Virajith Jalaparti, Peter Bodik, Srikanth Kandula, Ishai Menache, Mikhail Rybalkin, and Chenyu Yan. Speeding up distributed request-response workflows. In *ACM SIGCOMM Computer Communication Review*, volume 43, pages 219–230, 2013.
[29] Junchen Jiang, Xi Liu, Vyas Sekar, Ion Stoica, and Hui Zhang. Eona: Experience-oriented network architecture. In *Proceedings of the 13th ACM Workshop on Hot Topics in Networks*, page 11. ACM, 2014.
[30] Roy Jonker and Anton Volgenant. A shortest augmenting path algorithm for dense and sparse linear assignment problems. *Computing*, 38(4):325–340, 1987.
[31] Rupa Krishnan, Harsha V Madhyastha, Sridhar Srinivasan, Sushant Jain, Arvind Krishnamurthy, Thomas Anderson, and Jie Gao. Moving beyond end-to-end path information to optimize CDN performance. In *Proceedings of the 9th ACM SIGCOMM conference on Internet measurement*, pages 190–201. ACM, 2009.
[32] Gautam Kumar, Ganesh Ananthanarayanan, Sylvia Ratnasamy, and Ion Stoica. Hold'em or fold'em?: Aggregation queries under performance variations. In *Proceedings of the Eleventh European Conference on Computer Systems*, page 7, 2016.
[33] Peng Li, Binoy Ravindran, and E Douglas Jensen. Utility accrual resource access protocols with assured timeliness behavior for real-time embedded systems. In *ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*, 2005.
[34] Peng Li, Haisang Wu, Binoy Ravindran, and E Douglas Jensen. A utility accrual scheduling algorithm for real-time activities with mutual exclusion resource constraints. *IEEE Transactions on Computers*, 55(4):454–469, 2006.
[35] Hongqiang Harry Liu, Raajay Viswanathan, Matt Calder, Aditya Akella, Ratul Mahajan, Jitendra Padhye, and Ming Zhang. Efficiently delivering online services over integrated infrastructure. In *NSDI*, volume 1, page 1, 2016.
[36] Xi Liu, Florin Dobrian, Henry Milner, Junchen Jiang, Vyas Sekar, Ion Stoica, and Hui Zhang. A case for a coordinated internet video control plane. *ACM SIGCOMM Computer Communication Review*, 42(4):359–370, 2012.
[37] Christopher D Manning, Christopher D Manning, and Hinrich Schütze. *Foundations of statistical natural language processing*. MIT Press, 1999.
[38] Ravi Netravali, Ameesh Goyal, James Mickens, and Hari Balakrishnan. Polaris: Faster page loads using fine-grained dependency tracking. In *NSDI*, pages 123–136, 2016.
[39] Ravi Netravali, Vikram Nathan, James Mickens, and Hari Balakrishnan. Vesper: Measuring time-to-interactivity for web pages. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2018.
[40] Lenin Ravindranath, Jitendra Padhye, Sharad Agarwal, Ratul Mahajan, Ian Obermiller, and Shahin Shayandeh. AppInsight: Mobile app performance monitoring in the wild. In *OSDI*, volume 12, pages 107–120, 2012.
[41] Lenin Ravindranath, Jitendra Padhye, Ratul Mahajan, and Hari Balakrishnan. Timecard: Controlling user-perceived delays in server-based mobile applications. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 85–100, 2013.
[42] Sanae Rosen, Bo Han, Shuai Hao, Z Morley Mao, and Feng Qian. Push or request: An investigation of HTTP/2 server push for improving mobile performance. In *Proceedings of the 26th International Conference on World Wide Web*, pages 459–468. International World Wide Web Conferences Steering Committee, 2017.
[43] Vaspol Ruamviboonsuk, Ravi Netravali, Muhammed Uluyol, and Harsha V Madhyastha. Vroom: Accelerating the mobile web with server-aided dependency resolution. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 390–403. ACM, 2017.
[44] Brandon Schlinker, Hyojeong Kim, Timothy Cui, Ethan Katz-Bassett, Harsha V Madhyastha, Italo Cunha, James Quinn, Saif Hasan, Petr Lapukhov, and Hongyi Zeng. Engineering egress with edge fabric: Steering oceans of content to the world. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 418–431. ACM, 2017.
[45] P Lalith Suresh, Marco Canini, Stefan Schmid, and Anja Feldmann. C3: Cutting tail latency in cloud data stores via adaptive replica selection. In *12th USENIX Symposium on Networked Systems Design and Implementation*, pages 513–527, 2015.
[46] Madeleine Udell and Stephen Boyd. Maximizing a sum of sigmoids. *Optimization and Engineering*, 2013.
[47] Balajee Vamanan, Jahangir Hasan, and TN Vijaykumar. Deadline-aware datacenter tcp (d2tcp). *ACM SIGCOMM Computer Communication Review*, 42(4):115–126, 2012.
[48] Matteo Varvello, Jeremy Blackburn, David Naylor, and Konstantina Papagiannaki. Eyeorg: A platform for crowdsourcing web quality of experience measurements. In *Proceedings of the 12th International on Conference on emerging Networking*

EXperiments and Technologies, pages 399–412, 2016.

[49] Zhe Wu, Curtis Yu, and Harsha V Madhyastha. Costlo: Cost-effective redundancy for lower latency variance on cloud storage services. In *NSDI*, pages 543–557, 2015.

[50] Kok-Kiong Yap, Murtaza Motiwala, Jeremy Rahe, Steve Padgett, Matthew Holliman, Gary Baldus, Marcus Hines, Taeeun Kim, Ashok Narayanan, Ankur Jain, Victor Lin, Colin Rice, Brian Rogan, Arjun Singh, Bert Tanaka, Manish Verma, Puneet Sood, Mukarram Tariq, Matt Tierney, Dzevad Trumic, Vytautas Valancius, Calvin Ying, Mahesh Kallahalla, Bikash Koley, and Amin Vahdat. Taking the edge off with Espresso: Scale, reliability and programmability for global internet peering. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 432–445. ACM, 2017.

[51] Minlan Yu, Wenjie Jiang, Haoyuan Li, and Ion Stoica. Tradeoffs in CDN designs for throughput oriented traffic. In *Proceedings of the 8th international conference on Emerging networking experiments and technologies*, pages 145–156. ACM, 2012.

Appendices are supporting material that has not been peer reviewed.

A INCENTIVE TO IMPROVE LATENCY

We show that it is impossible to improve a group of users' QoE without reducing at least some of their external delays. Formally, this can be expressed by the following theorem.

THEOREM 1. *For n requests $r_1, \ldots, r_n$, suppose S and S' are the server-side delay vector when the external delay vector is C and C', respectively. Then $\sum_i Q(c'_i + s'_i) > \sum_i Q(c'_i + s_i)$ only if there is an r_i such that $c'_i < c_i$.*

PROOF. Our proof is by contradiction. Suppose that $\sum_i Q(c'_i + s'_i) > \sum_i Q(c_i + s_i)$ and for all i, $c'_i \geq c_i$ (*i.e.,* no request has a better external delay). Then

$$\sum_i Q(c_i + s_i) \geq \sum_i Q(c_i + s'_i) \quad (S \text{ is better than } S \text{ given } C)$$

$$\geq \sum_i Q(c'_i + s'_i) \quad\quad (Q \text{ is monotonic})$$

which contradicts the assumption.   □

B USER STUDY ON WEB QUALITY OF EXPERIENCE

We provide more details about our user study, which measures the relationship between page load time (PLT) and quality of user experience. We conduct this study on participants hired through Amazon MTurk [1], a crowdsourcing marketplace.

Test procedure: Before entering the study, participants have to fill out a questionnaire about their basic information, such as age group, nationality, gender, time spent online per day, and primary use of internet. Each participant is asked to rate their experience of the same web page when it is loaded with different PLTs. Since the actual PLT of a page may be affected by many factors—*e.g.,* the participants' browsers, operating systems, and network conditions—we show each participant a video recording of the web page being loaded at a certain speed, rather than letting them load the web page. This ensures that all the participants experience the same PLTs. The videos of different PLTs are played in a random order. After watching a video, the participants rate the video with a score ranging from 1 to 5 (with 1 being the least satisfactory and 5 being the most satisfactory), and this score is regarded as the QoE for the PLT shown in the video.

Video recording: In our study, we need to show videos of certain PLTs. To avoid uncontrollable WAN and server-side delays, we first download all web page content on a local machine, and then load the pages on this machine. This reduces the factors affecting PLT to (1) the browser rendering time on the machine, which is a function of system configuration (*e.g.,* operating system, computer hardware, browser version, etc.) but remains fixed, and (2) the web data packet arrival rate. Since the data packets are loaded from the local machine itself, we can achieve the desired PLT by tuning the per-packet delay using a Chrome developer tool called NoThrottling. This allows us to load each web page at the desired PLT, and record a video of the loading process. These are the videos that are downloaded and shown to the participants during the study.

Results: We ran the user study on the three page types in our traces (Table 1), as well as four other web pages: a Google search results page and the homepages of Amazon, CNN News, and YouTube. For each page, we use 50 MTurk participants. Figure 22 shows the results for the four web pages. We can see that although the websites have different PLT sensitivities, a sigmoid-like relationship between QoE and PLT exists for all of them.
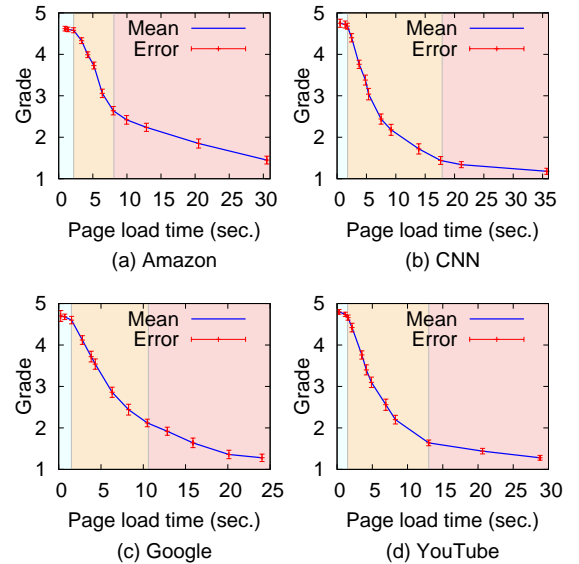

Figure 22: *The relationship between page load time and user rating in different websites.*

Response validation: A common problem in crowdsourcing is the validation of participants' responses. We filtered invalid responses in two ways:

- *Engagement:* Spending too long or too short on a video may indicate that the participant is distracted or unengaged. We set time thresholds for identifying such participants, and remove any response that takes more than 35 seconds or less than 2 seconds.
- *Outliers:* We view the average of all responses as the "ground truth". We drop responses from participants whose ratings consistently deviate from the ground truth by 3, across all videos.