Taiji: Managing Global User Traffic for Large-Scale Internet Services at the Edge

David Chou Tianyin Xu* Kaushik Veeraraghavan Andrew Newell Sonia Margulis Lin Xiao
Pol Mauri Ruiz Justin Meza Kiryong Ha Shruti Padmanabha Kevin Cole Dmitri Perelman
{davidchou,tianyin,kaushikv,newella,frumious,lxiao,pol,jjm,krha,shrupad,kbcole,dmitrip}@fb.com
Facebook Inc. *UIUC

Abstract

We present Taiji, a new system for managing user traffic for large-scale Internet services that accomplishes two goals: 1) balancing the utilization of data centers and 2) minimizing network latency of user requests.

Taiji models edge-to-datacenter traffic routing as an assignment problem—assigning traffic objects at the edge to the data centers to satisfy service-level objectives. Taiji uses a constraint optimization solver to generate an optimal routing table that specifies the fractions of traffic each edge node will distribute to different data centers. Taiji continuously adjusts the routing table to accommodate the dynamics of user traffic and failure events that reduce capacity.

Taiji leverages connections among users to selectively route traffic of highly-connected users to the same data centers based on fractions in the routing table. This routing strategy, which we term connection-aware routing, allows us to reduce query load on our backend storage by 17%.

Taiji has been used in production at Facebook for more than four years and routes global traffic in a user-aware manner for several large-scale product services across dozens of edge nodes and data centers.

1 Introduction

Modern Internet services operate on a bipartite architecture with dozens of data centers interconnecting with *edge nodes*, also known as point-of-presence [46, 60]. Data centers host a majority of the computing and storage capacity of most Internet services. Edge nodes are much smaller in size and are situated close to the end users for two major functions: (1) reverse proxies for terminating user connections close to their ISPs, and 2) caching and distribution of static content such as images and video.

SOSP '19, October 27–30, 2019, Huntsville, ON, Canada

ACM ISBN 978-1-4503-6873-5/19/10.
https://doi.org/10.1145/3341301.3359655

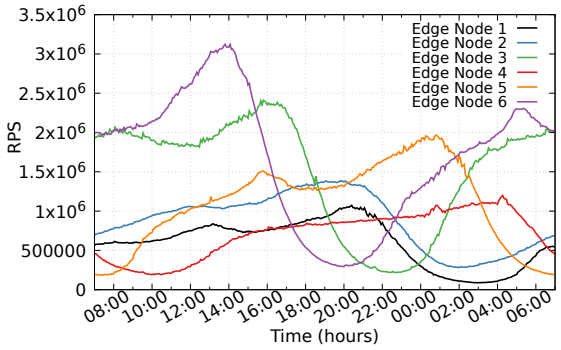


Figure 1. User requests per second (RPS) received at six edge nodes in different geo-locations over a 24-hour day. Observe that peak load differs substantially from trough at every edge node.

Static content is served predominantly from edge nodes. In case of a miss in the edge node, user requests for static content are then fetched from data centers. We leverage existing wide area network (WAN) traffic engineering solutions designed for content distribution networks to serve user requests for static content [8, 13, 14, 32, 33, 56, 61].

User requests for dynamic content, such as real-time messages and search queries, are the predominant source of network traffic from edge nodes to data centers. It is common practice for major Internet services to build private backbone networks or peering links that connect edge nodes to their data centers to avoid unpredictable performance and congestion on public WANs [19, 20, 22, 24, 29, 44]. Our focus in this paper is on how we can serve user requests for dynamic content while optimally utilizing our data center capacity and minimizing network latency.

For the first decade of Facebook's existence, we used a static mapping to route user requests from edge nodes to data centers. The static mapping became increasingly difficult to maintain as our services expanded globally. Figure 1 plots user traffic received at six edge nodes in different geographic regions over the course of a 24-hour day in March 2019. Observe that: 1) for each edge node, the traffic volume varies significantly during a day, and 2) the magnitude and peak time for edge nodes differ significantly. Taking Edge Node 1 as an example, daily peak traffic is 7× more than the trough. The dynamism and heterogeneity of global user traffic brings the following challenges:

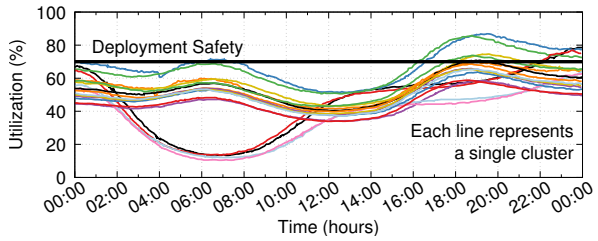


Figure 2. The normalized frontend utilization for Facebook's main web service on a day in April 2015 using static edge-to-datacenter traffic mapping. Each line denotes the utilization for one frontend cluster composed of several thousand web servers; each data center houses one or more frontend clusters. Observe that four clusters were nearly idle at 10% utilization, while seven were above 60% utilization. The black line denotes the "Deployment Safety" threshold above which we cannot safely rollout software updates that require restarting web servers.

- **Capacity crunch.** If a service's popularity and capacity needs outpace capacity acquisition plans, we might be unable to service all user requests. Concurrently, the sheer volume of user requests emanating from some locales experiencing explosive growth implies that a static edge-to-datacenter mapping would lead to capacity shortage in data centers serving some edges, and over-provisioned capacity in other data centers as depicted in Figure 2. This imbalance might result in load shedding or failures during peak load time.
- **Product heterogeneity.** As products evolved, the nature of user requests changed. We found that a larger fraction of our user traffic originated from products that provided an interactive experience that wished to maintain sticky routing between a user's device and our data centers. The stickiness makes it harder for a static mapping to manage user traffic, as static mapping cannot account for unmovable user sessions that are dynamically established.
- **Hardware heterogeneity.** The underlying infrastructure is constantly evolving as server generations are updated, capacity is added or removed and network infrastructure is improved, all of which affect how much traffic a data center can service. A traffic management system must be flexible and adapt routing strategies to infrastructure evolution.
- **Fault tolerance.** As the infrastructure footprint grows, it becomes increasingly likely that some fraction of edge and data center capacity becomes unavailable due to network failures, power loss, software misconfiguration, bad software releases etc, or other myriad causes [9, 16, 17, 28, 35, 36, 41, 58, 59]. A static edge to data center routing strategy is rigid and susceptible to failures when operational issues arise.

Our initial response was to develop operational tools for balancing the dynamics of traffic load, dealing with the daily chaos of peak demands, responding to failures, and maintaining good user experience. However, these manual operations were often inefficient and error prone.

We leveraged our operational experience to design and implement Taiji, a new system for managing global user traffic for our heterogeneous product services. Taiji was built with two goals in mind: 1) balancing the utilization of data centers and 2) minimizing network latency of user requests.

Taiji models edge-to-datacenter traffic routing as an assignment problem—assigning traffic objects at the edge to the data centers to satisfy the service-level objectives of balancing utilization and minimizing latency. Taiji uses a constraint optimization solver to generate a routing table that specifies the fractions of traffic each edge node will distribute to different data centers to achieve its configured goals. The *utilization* of a data center is aggregated over the frontend servers and represents a service's resource consumption characteristics. Taiji continuously adjusts the routing table to accommodate the dynamics of user traffic and the failure events that reduce capacity.

Taiji's routing table is a materialized representation of how user traffic at various edge nodes ought to be distributed over available data centers to balance data center utilization and minimize latency. The strawman approach is to leverage consistent hashing [31]. Instead, we propose that popular Internet services such as Facebook, Instagram, Twitter and YouTube leverage their shared communities of users. Our insight is that sub-groups of users that follow/friend/subscribe each other are likely interested in similar content and products, which can allow us to serve their requests while also improving infrastructure utilization.

We build on the above insight to propose *connection-aware routing*, a new strategy where Taiji can group traffic of highly-connected users and then route the traffic to the same data centers based on fractions specified in the routing table. By leveraging locality of user traffic (in terms of the content), we can improve cache hit rates and achieve other backend optimizations such as minimizing shard migration [1]. We find that connection-aware routing achieves a 17% query load reduction on our backend storage over an implementation based on social graph partitioning [26, 47].

This paper makes the following contributions:

- To the best of our knowledge, Taiji is the first system that manages user requests to dynamic content for Internet services in a user-aware manner.
- We describe how to model user traffic routing as an assignment problem which can be solved by constraint optimization solvers.
- We present connection-aware routing, which routes traffic from highly-connected users to the same data centers to achieve substantial improvements in caching effectiveness and backend utilization.

Taiji has been in production at Facebook for more than four years and routes global traffic in a user-aware manner for several large-scale product services across dozens of edge nodes and data centers.

2 Background

Facebook's user traffic serving infrastructure is similar to other Internet services [15, 32, 34, 38, 44, 53, 54].

Figure 3 depicts the serving path for a user request to www.facebook.com. The user's browser or mobile app sends the request to an ISP which maps the domain name to a Virtual IP (VIP) address using a DNS resolver. This VIP address points to one of the dozens of globally deployed edge nodes. An edge node consists of a small number of servers, typically co-located with a peering network [46, 60].

The user request will first hit an L4 load balancer which forwards the request to an L7 edge load balancer (Edge LB) where the user's SSL connection is terminated. Each Edge LB runs a reverse proxy that maintains persistent secure connections to all data centers. Edge LBs are responsible for routing user requests to frontend machines in a data center through our backbone network.

Within a data center, the user request goes through the same flow of an L4 load balancer and an L7 load balancer (named Frontend LB), as shown in Figure 3. The Frontend LB proxies the user request to a web server. This web server may communicate with tens or hundreds of micro services which, in turn, typically need to further communicate with other backend systems to gather the data needed to generate a response. The web server handling the user request is responsible for returning a response to the edge node which then forwards it to the user.

Backbone capacity. We would like to note that there is abundant backbone capacity between every edge node to data centers to allow for load spikes, link failures, maintenances, etc. Different from public WANs, backbone link capacity at Facebook is not a constraint (similar to other private backbone networks [18–20, 22, 24, 29, 44]), especially given that the traffic induced by dynamic content is only a small percentage of the overall bandwidth. Note that backbone capacity is constantly verified by regular load testing [53] and drain/DiRT-like testing [30, 54] that manipulate live traffic at edge nodes to simulate how data centers handle worst-case scenarios. Compared with the amount of traffic driven by these tests, the amount of dynamic user traffic Taiji manages does not stress our backbone links. Therefore, we currently do not consider the backbone link utilization in edge-to-datacenter user traffic management.

Traffic types. User traffic can be stateless or sticky. Our web services are stateless—user requests can be routed to any available data center. Interactive services, such as instant messaging, pin a user's requests to the particular machines

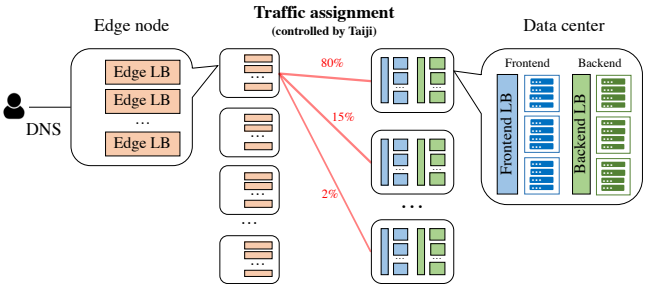


Figure 3. An overview of Facebook's infrastructure and the role Taiji plays. Taiji decides how dynamic user traffic at the edge nodes is routed to data centers by continuously adjusting the Edge LB's routing configurations.

that maintain the user sessions. For sticky traffic, a new request can be routed to any data center which will initialize a new session; once a session is established, the subsequent user requests are always routed to the same destination based on cookies in the HTTP header.

3 Taiji

Taiji serves as a user traffic load balancing system that routes user requests for dynamic content from edge nodes to available data centers. It determines the destination data center every user request is routed to. Figure 4 illustrates the architecture of Taiji which consists of two main components: Runtime and Traffic Pipeline.

Taiji's Runtime decides the fractions of traffic each edge node will send to available data centers in order to meet service level objectives specified in a *policy*. Taiji formulates traffic routing as an assignment problem that models the constraints and optimization goals set by a service. Runtime's output is a routing table that meets the policy. Taiji continuously adjusts the routing table to keep pace with the ebb and flow of diurnal traffic patterns as well as maintenances, failure events, and other operational issues.

Traffic Pipeline takes as input the routing table generated by Runtime and then leverages connection-aware routing to generate specific routing configuration for each Edge LB. The insight in connection-aware routing is that users in a shared community follow/friend/subscribe so they can engage with similar content. Connection-aware routing groups highly-connected users into "buckets" and selectively routes buckets to the same data centers, based on the traffic fractions specified in the routing table. Note that connection-aware routing is per-product but it is agnostic to service-level objectives—it faithfully follows the routing table generated by the Runtime.

The Edge LB parses each incoming user request, maps the user into an appropriate bucket, and then forwards the user's request to the data center containing that bucket in the routing configuration.

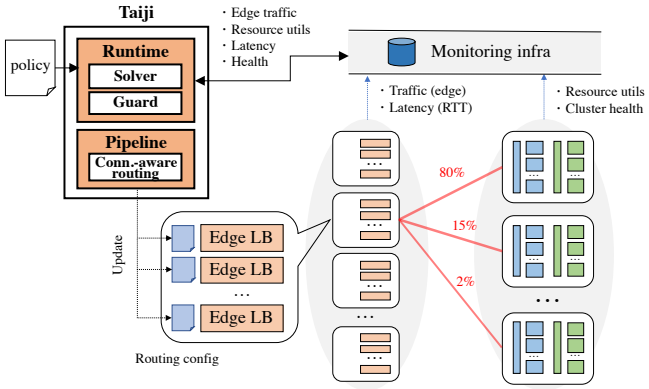


Figure 4. Taiji consists of two components: (1) a Runtime that generates a routing table based on various system inputs (traffic load, data center utilization, latency between each edge node and data center pair, etc.) and a service-level policy; (2) a Traffic Pipeline translates the routing table into fine-grained entries to instruct each Edge LB how to route traffic of a user to a particular data center.

3.1 Runtime

The Runtime component is responsible for generating a *routing table* which specifies the fraction of user traffic that each edge node ought to send to a data center. This routing table is a simple collection of tuples of the form:

```
{edge:{datacenter:fraction}}
```

We next describe the inputs provided to Runtime, how we model the constraint satisfaction problem, and the safety parameters considered when generating a routing table.

3.1.1 Inputs to Runtime

Taiji reads two types of dynamic data as inputs from the monitoring system [43]: (1) operational state of the infrastructure such as capacity, health and utilization of edge nodes and data centers, and (2) measurement data such as edge traffic volumes and edge-to-datacenter latency.

Taiji implements an abstraction called *reader* for data reading, normalization, and aggregation before feeding the data into modeling and solving. Readers decouple input reading and processing from the rest of Runtime. This allows us to create an hermetic test environment for Taiji by plugging in readers that can consume data from historical snapshots or synthetic data in test scenarios (§3.3.1).

3.1.2 Modeling and Constraint Solving

Taiji formulates edge-to-datacenter routing as an assignment problem that satisfies a service-specific policy. Our early design used a closed-form solver specific to a single product. As more products wished to use Taiji, we found that building and maintaining a closed-form solver for every product was time consuming and error prone. Further, as our infrastructure grew with new data centers and edge nodes, we had to continually re-build and re-deploy the closed-form solvers. We simplified the design and implementation of Taiji with constraint-based solving based on a generalization of all the closed-formed solvers we had implemented.

Taiji models traffic load as requests per second (RPS) for stateless traffic and as user sessions for sticky traffic. The model allows stateless traffic to be routed to any available data center while constraining sticky traffic to the same machine so as not to disrupt established sessions.

Another input to Taiji is the *utilization* of the data center—a measurement of how much traffic can be served per service. A good utilization metric should be easy to measure at the server level and aggregate on a data center scale, and be able to account for the heterogeneity in hardware generations. Services verify their utilization metric at the data center level by running regular load tests using live traffic [53].

Utilization metrics vary between services. Our main web service uses a normalized metric called *i-dyno score* based on the MIPS (Million Instructions Per Second) observed on web frontend servers, to account for heterogeneous processor architectures. Our mobile service, Facebook Lite, measures a server's utilization based on the number of active sessions. Taiji assumes that utilization for a service increases proportionally to the load being served in an epoch.

Taiji re-evaluates traffic allocation decisions in every epoch (set by a service) by reading current utilization directly from our monitoring systems.

A policy specifies *constraints* and *objectives*. Policies typically have the constraint of not exceeding predefined data center utilization thresholds to avoid overloading any data center. Our most commonly-used policy specifies the objectives of balancing the utilization of all available data centers, while optimizing network latency. An alternative policy has the objective of "closest data center first" which is modeled by optimizing edge-to-datacenter latency. We describe our formal model for the balancing policy in the appendix.

Taiji employs an assignment solver to solve the problem. Our solver employs a local search algorithm using the "best single move" strategy. It considers all single moves: swapping one unit of traffic between two data centers, identifying the best one to apply, and iterating until no better results can be achieved. Our solver takes advantage of symmetry to achieve minimal recalculation. We compare our solver with an optimal solver using mixed-integer programming. Our solver always generates the optimal results despite using local search because the safety guards (§3.1.3) limit the search space. The solver can generate a solution in 2.81 seconds on average. If we double our scale (2× data centers, 2× edge nodes, and 2× user traffic), we can solve the problem in 9.95 seconds. If we increase our scale by 4 times, we can solve the problem in 43.97 seconds.

3.1.3 Pacing and Dampening

Taiji employs several *safety guards* to limit the volume of traffic change permitted in each update of the routing table. First,

Taiji uses an onloading safety guard, a configurable constant (0.04 in production) to bound how much the utilization of a data center can increase during an epoch. The onloading constant is encoded as a constraint in the model (described in the appendix). The onloading safety guard breaks up large changes into multiple steps, each published in serialized routing table updates so all the downstream services in the data centers, such as our caching systems, have the opportunity to warm up over time. In addition, we use another safety guard to limit the max allowable fraction of traffic sent to a data center. This provides tighter bounds on the operating limits of Taiji to ensure safety.

Moreover, Taiji has two levels of protection to prevent unnecessary oscillations. First, we find that small traffic changes (e.g., shifting less than 1% of the traffic) provide little value but can cause oscillations. Taiji will skip the change if the shift does not reach *minimum shift limit*. Second, Taiji also allows specifying a *dampening factor*. Our linear model with load and utilization is not perfect; thus trying to make an exact traffic shift to the target can lead to overshooting. We use a dampening factor to aim for 80% of the target.

Sensitivity analysis. To determine the right limits for safety guards, we run a sensitivity analysis by shifting increasing amounts of load to a set of services and measuring the throughput, latency, cache efficiency and other characteristics of backend systems. We find that traffic shifts in stateful services are more costly than in stateless services, because the shifts disrupt sticky user sessions which need to be re-established. Traffic shifts for stateless services result in increased latency and throughput drops as the new requests will experience cache misses. We continually run this sensitivity analysis and tune the threshold presets.

3.2 Traffic Pipeline

Traffic Pipeline consumes the routing table output by Runtime and generates the specific routing entries in a configuration file using *connection-aware routing*. It then disseminates the routing configuration out to each Edge LB via a configuration management service [48, 50]. The routing entries specify the buckets based on which each edge node routes to data centers, in the form of

```
{edge:{datacenter:{bucket}}}
```

It takes about a minute to generate the routing entries and deploy them to the edge nodes. Currently, we use an epoch of 5 minutes as the duration between routing table updates.

3.2.1 Connection-Aware Routing

Connection-aware routing is built on the insight that user traffic requesting similar content has locality and can benefit the caching and other backend systems. Connection-aware routing brings locality in traffic routing by routing traffic from highly-connected users to the same data center.

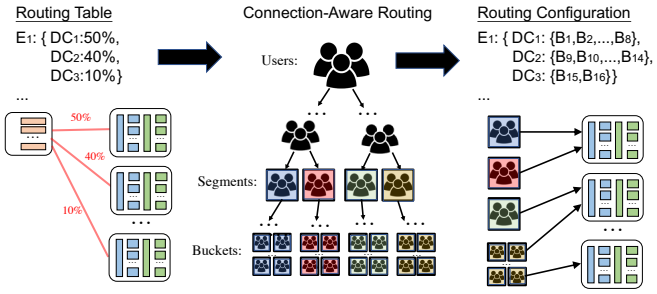


Figure 5. Connection-aware routing transforms a routing table into routing configuration entries that specify how to route user traffic based on user connections in a community graph. We do not depict the bucket weights (per-bucket traffic fraction) for clarity.

Connection-aware routing builds upon Social Hash [26, 47] that partitions the community graph into *user buckets*.[1] It uses classical balanced graph partitioning to ensure that each user bucket is roughly the same size and maximizes the connections within each bucket. Online routing mechanisms route traffic at the granularity of these buckets (which are computed offline). The online routing uses consistent hashing to ensure each data center gets an appropriate number of buckets (based on the fractions specified in the routing table) while also ensuring that routing is stable (i.e., users should be routed consistently to the same data center).

The number of users per bucket trades off routing accuracy with potential cache efficiency. To have fine-grained traffic shifting control, Taiji desires smaller bucket sizes of the order of 0.01% of global traffic. However, smaller bucket sizes leads to splintering of large community groups into separate buckets, each of which could be routed to different data centers, destroying their shared locality benefits.

Connection-aware routing overcomes this trade-off by coupling (1) offline user-to-bucket assignments with (2) online bucket-to-datacenter assignments. The user buckets are created in a hierarchical structure that allows the online component to route highly-connected buckets to the same data center. This enables Taiji to increase community connections routed to the same data center from 55% to 75%, resulting in substantial improvements of backend utilization.

Offline user-to-bucket assignment. The community hierarchy is created in the form of a complete binary tree, where leaf nodes are user buckets and inner nodes represent

[1] An alternative is to bucket users by their location information instead of social connections. We learned that location information is often unreliable—peering networks often route users from towns/cities in neighboring Asian countries to larger regional ISPs (e.g., Singapore) which forward the request to an edge node. Domain information such as country/state/city is unusable as users "from Singapore" might be different nationalities and speak different languages. Attributes such as user age, or other user-specific information is extremely difficult to use in a privacy-preserving manner. Therefore, we prefer leveraging the connection graph which provides sufficient information without leaking user information.

the total set of users in their subtree's buckets (Figure 5). This tree is constructed based on the Social Hash [26, 47] algorithm, except construction happens in successive bipartitions. The levels of the tree are represented as L, where $L = [0, H]$, and H represents the height of the tree. The level $L = 0$ is a single head node representing all the users. At level $L = H$ are 2^H leaf nodes, each representing a bucket of 0.01% users. A bucket is the finest routing granularity available to Taiji. The 2^{L+1} nodes for level $L + 1$ of the tree are generated by performing 2^L balanced bipartitions that minimize edge cut of the 2^L nodes in level L.

This partitioning is produced offline on a weekly basis to adapt to changes in the community graph. We cap user movement across buckets to 5% to limit the amount of rerouting that these weekly changes might induce. In practice, we find that less than 2% of the users need to be moved to maintain the same quality of partitioning over time, i.e., percentage of connections within buckets at varying levels of the community hierarchy.

When we started deploying Taiji, we found that there were two types of connections where tree-based partitioning does not work well: (1) *User to highly-connected entity* where one end of the connection is a user but the other end is a highly-connected entity, such as President Barack Obama or Justin Bieber. In this case, the connection is a subscription to news media rather than a social connection. Given the large subscription base, we cannot route all the subscribed users to the same data center; and (2) *One-time interactions* where users are connected temporarily (e.g., a user submits a payment to an acquaintance). We do not route users with one-off or minimal interactions to the same data center as such interactions are too weak to be qualified as "connections." Taiji excludes these two types in connection-aware routing.

Online bucket-to-datacenter assignment. Traffic Pipeline invokes the online component of connection-aware routing to transform the fractions in the routing table into bucket assignments that keep most large subtrees of buckets together while splitting a few to enable fine-grained traffic routing. This online component invokes the Stable Segment Assignment algorithm (Algorithm 1) based on per-bucket traffic and the capacity of each data center (normalized as bucket weights and data center weights in Algorithm 1).

Stable Segment Assignment strives to preserve bucket locality by assigning a whole level of buckets (called a *segment*) in the community hierarchy to the same data center; only a minimal number of segments need to be split. For stability, the same segments should be assigned to the same data centers as much as possible. A unique ordering of segments is generated for each data center using a random permutation (P) seeded by the data center name (Line 8). This ordering is used to represent a random preference of data centers to segments. Buckets are mapped to segments as per Line 10. Tuples (T) are created for every (data center, bucket) pair

Algorithm 1 Stable Segment Assignment

```
1:  B ← list of buckets
2:  D ← list of data center names
3:  W_B ← bucket weights
4:  W_D ← data center weights
5:  S ← number of segments
6:  T ← ∅
7:  for d ∈ D do
8:      P ← PermutationWithSeed(S, d)
9:      for b ∈ B do
10:         segment ← ⌊ b·|B| / S ⌋
11:         T ← T ∪ ⟨P[segment], d, b⟩
12:     end for
13: end for
14: SortLexicographically(T)
15: A ← ∅
16: for ⟨preference, d, b⟩ ∈ T do
17:     if b ∉ A ∧ W_D[d] > 0 then
18:         A[b] ← d
19:         W_D[d] ← W_D[d] − W_B[b] · (Σ W_D / Σ W_B)
20:     end if
21: end for
22: return A
```

which begins with the data center's preference for the segment that bucket belongs to (Line 11). All such tuples are sorted lexicographically, in the order of preference, then data center, and finally bucket (Line 14). Tuples are traversed in this order, and buckets are greedily assigned to data centers, until a data center has its desired weight (Lines 16-21).

Stable Segment Assignment achieves *stable* routing: the bucket-to-datacenter assignments need minimal changes to accommodate routing table changes in consecutive epochs. In other words, the set of data centers visited by each user is minimized to reduce storage cost and keep caches warm. The algorithm preserves *locality*, as buckets in the same segment are commonly routed to the same data center. We group buckets into $S = 2^L$ segments based on the level L in the community hierarchy. The choice of L is a trade-off between stability and locality. A smaller L leads to higher locality but lower stability (segments are more likely to be split into buckets). We empirically pick L to be 7 which will be discussed in depth in §4.3. Note that $L = H$ is equivalent to Social Hash [47].

3.2.2 Edge LB Forwarding

Connection-aware routing requires Edge LBs to support bucket forwarding. For a user request, an Edge LB routes the request to the data center according to the user's bucket specified in the routing configuration. Note that maintaining every user-to-bucket mapping at each Edge LB is inefficient—with billions of users, each Edge LB needs to maintain a gigabyte-sized mapping table in memory and synchronize

the table when the mapping changes. Thus, we maintain the fine-grained user-to-bucket mapping in data centers and let Edge LBs route traffic at the granularity of buckets.

The initial user request has no bucket information and therefore the Edge LB routes it to a nearby data center. When the request reaches the data center, the frontend writes the bucket ID in the HTTP cookie based on the user-to-bucket mapping. Then the subsequent requests will have bucket ID in the cookie and the Edge LB will route the request to the data center specified in the routing configuration.

Note that connection-aware routing (header parsing and bucket ID extraction) adds negligible computational overhead on the original request handling path. The mapping file is shipped to every edge node once every 5 minutes and is only 15KB in size, which is negligible data transfer overhead.

3.3 Correctness and Reliability

Correctness and reliability are important design principles of Taiji. A software bug or misconfiguration in Taiji that results in an incorrect routing table may trigger myriad failures ranging from dropping some fraction of user traffic, to a spike in traffic directed at a particular data center that overwhelms our backend systems leading to more cascading failures.

In this section, we provide some insights into our test strategy, our approach to input/output validation, how we deal with hardware failures, how we respond to larger infrastructure failures, and how we size our backend systems to function well both during steady state and failure scenarios.

3.3.1 Testing

The major risk to Taiji is not a fail-stop bug, which are easily detected via a litany of unit tests, integration tests, etc., but *semantic bugs* that generate incorrect routing configuration (e.g., due to numeric errors in the calculation). We find that semantic bugs are typically noticed by alerts firing after service-level metrics cross some preset thresholds, which implies that the service has already been affected.

We leverage the hermetic test environment of the Runtime combined with historical snapshots of data to construct *regression* tests to verify Taiji's correctness. The drawback of this approach is that we can only test for problematic scenarios that our deployment has experienced. That said, we have instituted a post-mortem practice where we snapshot any inputs/configuration/policies that cause Taiji to fail in production and add a test case to validate future builds.

When onboarding new services or changing policies, we perform online testing that covers weekly traffic patterns. One common practice is to set conservative safety guards in order to understand Taiji's behavior in a controlled manner before deploying to production.

Input and output validation. We place a number of validators for the input and output of each component in Taiji. The basic idea of validating inputs is to cross-check data from different sources, e.g., the current traffic allocation in production should be consistent with the latest routing configuration. The output validation is to check against predefined invariants, e.g., a data center cannot take more traffic than its remaining capacity can serve. The last-known-good configuration will be used upon validation failures. Last, the safety guards (§3.1.3) ensure that Taiji cannot change traffic allocation dramatically in an epoch.

3.3.2 Tolerating Failures

Taiji allows services to register metrics based on their own health monitoring. Taiji checks these metrics at each epoch. If the metrics indicate unhealthy service states, e.g., exceeding the safety threshold, Taiji alerts the service owners instead of adjusting traffic.

Dependencies. Taiji is built with minimal dependencies—it only depends on the monitoring infrastructure to read the inputs (§3.1.1) and the configuration management service to deploy routing configurations to the Edge LBs. Both configuration management and monitoring have similar or higher availability guarantees than Taiji [52].

It is possible, though extremely rare, that the monitoring system reports wrong data, which leads to the consequence of Taiji generating wrong routing configurations. Such cases are no different from Taiji's own bugs and we deal with them using input and output validation.

Hardware. Taiji runs on commodity hardware. To tolerate machine failures, we deploy multiple Taiji instances in data centers in different geographic regions. These instances form a quorum using ZooKeeper [21] and maintain one unique leader at a time. Only the leader Taiji instance computes the routing configurations and deploys them to the Edge LBs. When the leader fails, a new leader will be elected.

3.3.3 Embracing Site Events

Taiji is not the only system that controls edge-to-datacenter traffic. At Facebook, traffic control is also used as the mechanism for two types of site events:

- *Reliability tests.* Facebook runs regular load tests [53] as well as drain and storm tests [54] which shift traffic into and out of a data center, respectively.
- *Failure mitigation.* We occasionally drain traffic out of a data center or a set of data centers to mitigate unexpected widespread failures.

When traffic is drained out of a failing data center, Taiji should recognize the resulting low utilization as intentional and not shift traffic back to the data center. We accomplish this by marking the status of any data center under maintenance or failure as `ABNORMAL`. Taiji excludes `ABNORMAL` data centers in its traffic balancing calculations.

Note that there could be a race condition in which Taiji and other tools submit conflicting traffic changes at the same time. We prevent race conditions from emerging by having all tools submit traffic shifts to the Traffic Pipeline, which is then responsible for detecting and resolving conflicts. Failure mitigation has the highest priority; if a traffic shift proposed by Taiji interferes with failure mitigation, the shift will be executed in the next eligible epoch.

When a data center suddenly goes offline, the Edge LBs have a fallback rule to route traffic globally to all data centers proportional to their capacity. This provides an immediate mitigation for datacenter-level disasters.

3.3.4 Backend Safety as a First Class Principle

A notable role of Taiji is to protect our backend systems from being overloaded by diurnal traffic patterns or peak load. As Taiji changes the traffic allocation across data centers, the adjustment itself should not incur excessive overhead to the backends (e.g., due to cold cache effects). As stated in §3.1.3, Taiji leverages safety guards to ensure that the frequency and magnitude of changes are smooth without load spikes. Taiji also accounts for minimum and maximum levels for operation, deciding when to bail out.

Unlike the backbone network, where traffic introduced by Taiji is insignificant, the capacity of backend systems is a bounding factor for the upper limit of traffic. Particularly, Facebook's backend systems are composed of many micro services with complex dependencies—lack of capacity at any of these services may cause cascading failures.

The disaster-readiness strategy at Facebook determines the upper bound of traffic at a data center. When a data center is down because of widespread failures (e.g. natural disasters), we drain traffic out of the data center and shift it to the remaining data centers [54]. Each of them will take an extra amount of traffic determined by Taiji's policy, and we prepare downstream services' capacity to serve such traffic.

We size the backend systems according to the utilization of the frontends. We plan the capacity of backend systems considering the extra traffic each data center can handle which is verified via regular drain tests and load tests. These tests are high fidelity exercises that provide us with confidence on the operational bounds within which Taiji can safely act.

We find that flash crowds that affect particular communities of people, or even geographies, are not significant enough to impact overall site load. That said, we do find that events like New Year's Eve and the World Cup are global phenomena that require provisioning capacity, customized traffic management, and other load management strategies.

4 Evaluation

Taiji has been in use at Facebook for more than four years, where it manages user traffic of multiple Internet services with varying traffic types and latency requirements. We evaluate Taiji using data collected from two of Facebook's main product services, a web service with stateless traffic and a mobile service with sticky traffic, with billions of daily active users. Our evaluation answers the following questions:

- How does Taiji balance large-scale Internet services with dynamic user traffic patterns and unexpected failure events?
- Can connection-aware routing improve cache efficiency and backend utilization?
- What is the role of pacing and safety guards?
- How stable are Taiji's routing strategies?
- Does Taiji support product services with different traffic types and varying routing policies?

4.1 Balancing Large-Scale Web Traffic

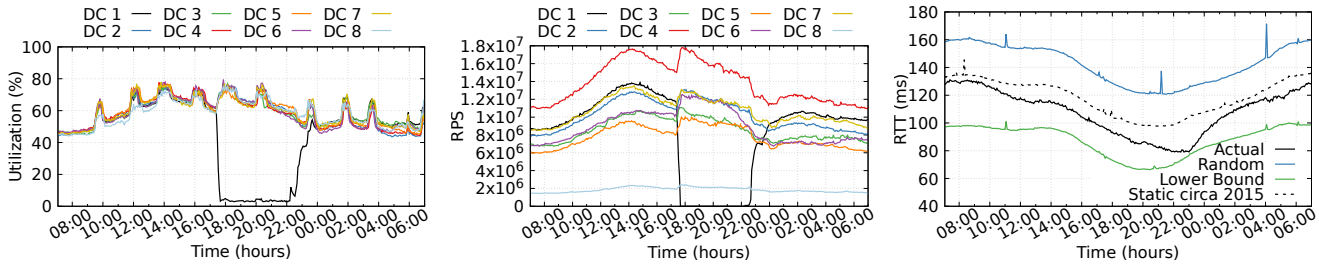
We show the effectiveness of Taiji using Facebook's main web service which is composed of stateless HTTPS requests. The web service uses a *latency-aware balancing* policy with the objectives of (1) balancing utilization of frontend web servers across data centers, while (2) optimizing RTT (Round Trip Time) of user requests[2] and (3) handling infrastructure failures seamlessly. Figure 6 depicts measurement data collected over a 24-hour period to cover the diurnal user traffic patterns shown in Figure 1. Note that we intentionally select a day with a major infrastructure failure that required us to drain traffic out of a data center; this event allows us to assess the effectiveness of Taiji's responsiveness to failures. We next delve into each of the subfigures of Figure 6 in turn.

4.1.1 Balancing Data Center Utilization

Figure 6a shows the frontend web server utilization of different data centers, henceforth referred to as "data center utilization". In contrast with the static edge-to-datacenter mapping used in 2015 (shown in Figure 2), the frontend utilization in 2019 across our data centers is always balanced. This balanced utilization enables us to support continuous deployment for software updates [45, 51]. In Figure 6a, each software update can be identified by a utilization spike as an update needs to restart the running software instance in a staged manner. Restarts reduce the available capacity for a short period which results in an increase in the utilization on the remaining servers. As shown in Figure 6a, Taiji balances data center utilization even with the spikes caused by temporal capability reduction. In contrast, we can see in Figure 2 that with static mapping, the utilization constantly exceeded the deployment-safety threshold and thus blocked software updates during peak time.

Figure 6b demonstrates that the volume of traffic (in Requests Per Second) routed to different data centers can be substantially different—the largest data center (DC-4) serves

[2]We do not use end-to-end latency as an input because it is not sufficiently stable and is affected by factors external to Taiji.



(a) Data center utilization **(b)** Requests Per Second **(c)** Round Trip Time

Figure 6. Taiji manages edge-to-datacenter traffic for Facebook's largest web service. This figure (from a day in 2019) depicts: (a) data center utilization, (b) traffic load (Requests Per Second) allocated to each data center, and (c) average network latency (Round Trip Time).

5× more than the smallest one (DC-8). All data centers are not created equal, in terms of size, server hardware, availability, etc. Taiji abstracts away the heterogeneity of data centers as well as edge nodes, and ensures that user traffic can always be served with optimized latency, as long as the overall capacity is sufficient for the global user demand.

4.1.2 Reacting to Unexpected Failure Events

Figures 6a and 6b depict the sudden traffic drop for DC-1 due to a data center drain event at 16:50 due to an unexpected failure. We observe that Taiji did not interfere with the higher priority drain event (§3.3.3) and no traffic is sent to DC-1 until the failure is resolved. Taiji seamlessly distributes the traffic previously served by DC-1 to the other data centers, as seen in the load spikes in Figure 6b. Figure 6a shows the additional traffic at 16:50 increases utilization of the other data centers proportionately, thus maintaining the service-level policy of balancing data center utilization.

4.1.3 Minimizing Latency

While meeting the constraint of balancing utilization across our data centers, Taiji also optimizes for minimizing latency by maximizing the probability of sending traffic to the nearest, under-utilized data center. Figure 6c illustrates this behavior by comparing the average RTT achieved by Taiji's traffic balancing strategy to random assignment and the theoretical lower bound of any traffic routing strategies. Both of them assume infinite data center capacity. The random assignment routes every user request to a random data center, while the theoretical lower bound is calculated by routing every request to the closest (in terms of RTT) data center. We also calculate the average RTT with a static edge-to-datacenter mapping (used in 2015). As shown in Figure 6c, with latency-aware balancing, Taiji achieves significant latency improvement over random assignments and is close to the lower bound (25 milliseconds gap in the worst case), and outperforms a manually crafted static mapping. Figure 7 further shows the latency distribution at the peak loads of different geographical locations. Note that the peak loads are the worst-case scenarios for latency because Taiji trades latency

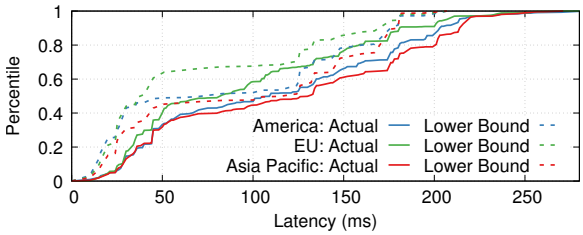


Figure 7. The CDF of the edge-to-datacenter latency (in terms of RTT) at the peak load of three different geographical locations.

for balancing at the peak. Still, the latency of Taiji's traffic allocations remains close to the theoretical lower bound. In Asia-Pacific, for example, the 50th, 75th, 90th, and 95th percentiles of latency only exceed the lower bound by 14.39, 19.56, 33.95, and 54.37 milliseconds, respectively.

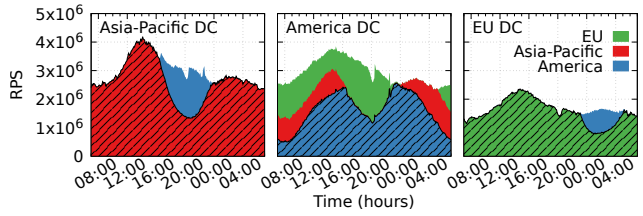
4.2 Efficient Capacity Utilization

Figure 8 illustrates how Taiji distributes user traffic from edge nodes to different data centers for Facebook's main web service. Our evaluation focuses on three geographical regions: EU, Asia-Pacific and America.

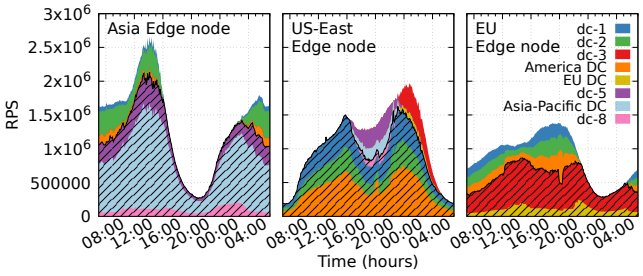
Figure 8a depicts the magnitude of requests served by each data center. Observe that Asia-Pacific DC primarily serves user traffic from Asia with a minor amount of American users. Similarly, EU DC primarily serves European users and a minor amount of American users. In contrast, America DC serves all three regions.

Figure 8b shows the destination data centers each edge node sends traffic to. If we examine the "Asia Edge node" traffic, we see that the vast majority of traffic from a node is sent to Asia-Pacific DC (light blue) but a significant fraction is also sent to non Asia-Pacific data centers. Similarly, the "EU Edge node" pane shows that we send some traffic to the EU DC but send a majority of the traffic to other data centers.

These two figures show that each edge node typically sends user traffic to the data center in its nearby geographical region which minimizes network latency. However, since this service is optimizing for utilization balance, some edge



(a) Distribution of traffic originated from different edge nodes received at three data centers (Asia-Pacific, America, and EU DCs).



(b) Distribution of traffic destined for different data centers at three edge nodes (Asia, US-East, and EU edge nodes).

Figure 8. Daily edge-to-datacenter traffic distribution. The shaded area denotes the traffic sent from/to the same geographical regions.

nodes will have to send traffic to remote data centers when the closest ones reach high utilization. This is most obvious in "US-East Edge node" in Figure 8b where we see that between 13:00-21:00, some traffic spills over to `Asia-Pacific DC`—this is because Asian users are likely asleep, causing `Asia-Pacific DC` to be underutilized unless users from other geographies are redirected there. Thus, we see that Taiji enables efficient utilization of data center capacity.

4.3 Effectiveness of Connection-Aware Routing

The principle of connection-aware routing is to route traffic of users from a segment to the same data centers to benefit from cached data and other backend optimizations. We optimize for a metric coined *connection locality* defined as the percentage of connections being routed together to the same data center. Our baseline implementation is Social Hash [47] which achieves 55% connection locality. Connection-aware routing improves this to 75%.

Figure 9 shows the reduction in query load on backend databases after we rolled out connection-aware routing compared to the baseline that uses Social Hash. We observe a 17% reduction in query load on the backends—the increased connection locality improves the efficiency of the caching system and other backend optimizations. This outcome at our deployment scale means a reduction of our infrastructure footprint by more than one data center.

We chose the bucket and segment sizes by analyzing the levels in the tree-based hierarchy and considering tradeoffs between locality, routing accuracy, and stability (§3.2.1).

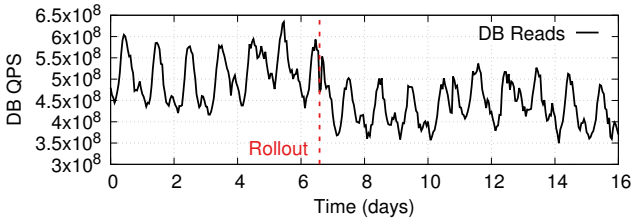


Figure 9. Reduction of database queries per second after the deployment of connection-aware routing which improves caching efficiency (the reduced queries are served at the cache layer).

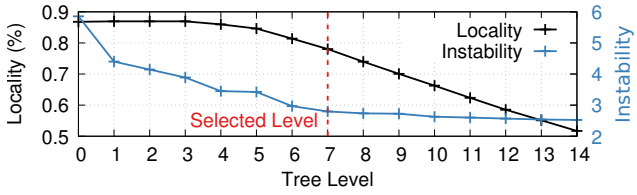


Figure 10. The tradeoff between locality and instability in selecting the tree level used by connection-aware routing. Locality is measured by the percentage of connections routed together, while the instability is measured by the average number of data centers a user bucket may be routed in a week.

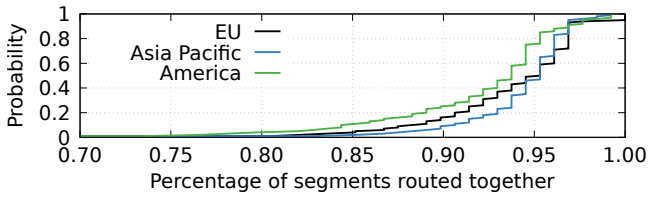


Figure 11. The probability of the percentage of segments routed together (to the same data centers) at three different edge nodes over the course of a day. Taiji routes 95+% of the segments together for 80% of the time (the other segments are further split up and routed at the bucket granularity).

First, we select bucket size at Level 14 of the tree (2^{14} = 16384 buckets) to ensure a 0.01% granularity in traffic allocation.

We select segment size as a tradeoff between locality and stability, discussed in §3.2.1. The principle is to select a segment size that is most likely to be routed together—too large a segment is more likely to be split into buckets due to the changes of routing table and results in instability. Figure 10 shows our sensitivity analysis for deciding segment size based on trace-based simulation. We select a segment at Level 7 (2^7 = 128 segments) which keeps the same level of stability as compared to Social Hash while improving the locality from 55% to 75%. Figure 11 shows the probability of the percentage of segments routed together at three different edge nodes. Taiji is able to route 95% of the 128 segments together in around 80% of the time.

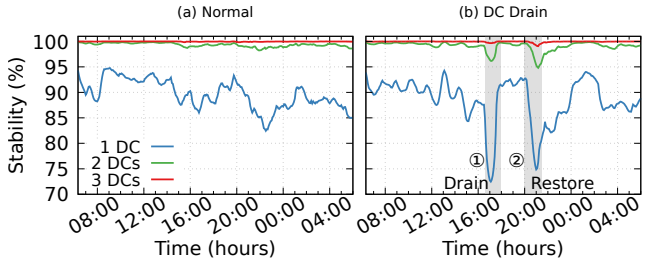


Figure 12. Stability of Taiji's routing (a) in a day with the normal state where 98+% users visit no more than 2 data centers, and (b) in a day with a datacenter-level drain events which increase oscillations during the traffic drain and restore periods.

4.4 Stability

Stability in routing means that traffic from the same user is *stably* routed to a small set of data centers instead of being bounced among a large number of data centers. Stable routing reduces the storage cost for keeping user data in cache and backend storage. In Taiji, a user can be routed to multiple data centers when Taiji changes the bucket-to-datacenter assignment (discussed in §3.2.1). The Stable Segment Assignment algorithm (Algorithm 1) minimizes the assignment changes to accommodate the proposals of traffic shifts.

Figure 12 illustrates the stability as the percentage of users that visit no more than N data centers in an hour. We show Taiji's stability both in the normal state (no major site events) and with the occurrence of ① a datacenter-level drain event that evicted all user traffic from a data center followed by ② a restore event that shifts the evicted traffic back. We can see that in the normal state, 98+% of our users visit less than two data centers, while more than 85% of our users visit only one data center most of the time. A user visits more than one data center because the edge nodes have different diurnal patterns–an increase in demand in a geographical region can push out users from the closest data center to the others. With Stable Segment Assignment, Taiji almost never routes users to more than three data centers. The region-level drain/restore events force user traffic to oscillate in a short period (the gray area) as shown in Figure 12. Such oscillations are expected and supported by our locality-based storage management services [1].

4.5 Pacing and Sensitivity Analysis

We demonstrate the effectiveness of pacing and safety guards discussed in §3.1.3 using a drain event. During this event, the utilization of a data center reaches its lower bound because it does not take any user traffic. Without pacing, when the data center is restored, traffic will flood into the data center and overload the backend services due to the cold cache.

Figure 13 shows how Taiji paces traffic shifts after a drain event which took place at minute 45–81 ②. Taiji stops routing user traffic to the data center during the drain event to

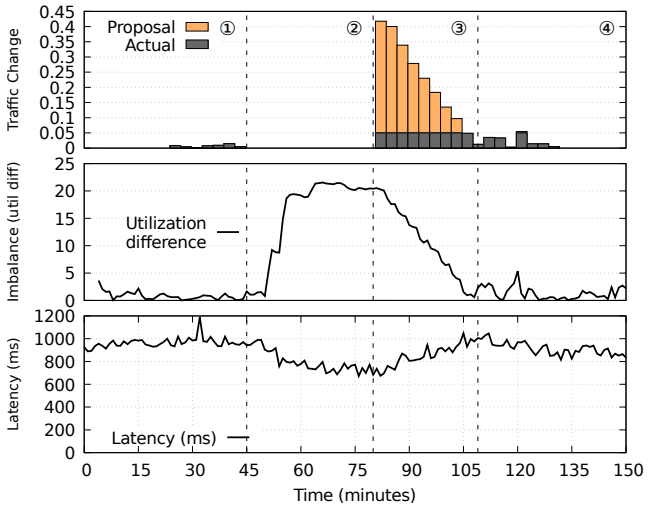


Figure 13. Taiji's pacing during datacenter-level drain and restore events. The uppermost subfigure shows the proposed traffic changes versus the actual changes after pacing. The middle subfigure shows the utilization divergence between the target data center and other data centers—the drain leads to significant divergence while Taiji gradually removes the divergence after the restoration. The bottom subfigure shows the 95th percentile of backend processing time—with Taiji's balancing strategy, the drain/restore events do not affect the backend processing latency.

avoid interference with our failure mitigation tool (§3.3.3)—it does not submit any change requests for the drained data center, as shown in Figure 13(a).

After the drain event, the data center resumes serving user traffic again ③. At the moment of restoration, since the utilization of the data center is dramatically different from the others, the assignment solver without safety guards would propose to shift back a large volume of user traffic as shown in Figure 13(a). However, the safety guards pace the proposed changes into a few small steps. These changes are gradually conducted in ③ and lead to a balanced steady state ④ in 30 minutes. At the steady states ① and ④, the changes are minimal to deal with organic traffic dynamics.

Figure 13(c) shows the 95th percentile of backend processing time, a metric sensitive to both caching and downstream service behavior. We see that with Taiji's pacing, the traffic increase did not cause impact on the service or the backends.

Figure 14 shows the sensitivity analysis that determines the pacing (§3.1.3) in 2015 and 2019, respectively. We see that in both 2015 and 2019, when we increase traffic to the data center, the CPU utilization of the backend systems will increase substantially. In Figure 14(a), in the beginning of each step, we can see icicles—the introduction of cold traffic leads to cache misses and results in CPU spikes at the backend.

We see from Figure 14(b) that with the improvements of our backend services (e.g., caching and sharding), we are

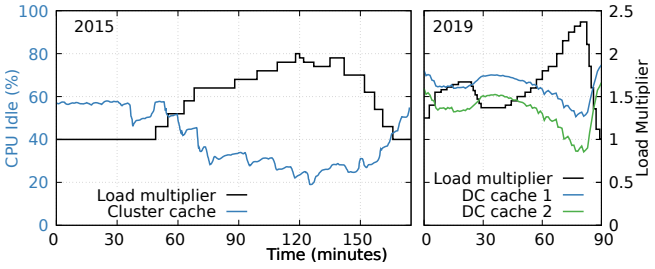


Figure 14. Sensitivity analysis for the same services in 2015 and 2019, respectively. The analysis in 2015 takes almost twice as long as the one in 2019. The impact on the backend systems is visible. Specifically, in 2015, there are icicles at the beginning of each step—the introduction of cold traffic leads to cache misses and results in CPU spikes at the backends.

able to shift traffic of the same service much faster and more smoothly in 2019 than in 2015.

4.6 Supporting Service Heterogeneity

As shown in the preceding sections, achieving balanced resource utilization has proven to be an efficient solution for traffic load balancing for global stateless services. However, Facebook also hosts services that serve traffic that could be stateful or require sticky connections. Other services could require custom traffic balancing to support products unique to a data center or other unique backend configurations. In other words, traffic management at Facebook cannot be formulated to have a one-size-fits-all solution. Taiji is built to be flexible and configurable to a service's need with intuitive constraints and robust testing infrastructure.

4.6.1 Diverse Traffic Characteristics

Besides the stateless traffic of the web service, Taiji is also used to manage *sticky* traffic for Facebook's mobile service, Facebook Lite, which also adopts the latency-aware balancing strategy described in §4.1. Compared with stateless traffic, sticky traffic requires modeling unmovable traffic objects—Taiji only manipulates new sessions without disrupting established sessions (§3.1.2). We quantify the effectiveness of Taiji in balancing sticky traffic based on the divergence of frontend utilization. At each point in time, the divergence d_i of the i-th data center is calculated as $d_i = \frac{|u^a - u_i|}{u^a}$, where u^a is the average utilization of all the data centers and u_i is the utilization of the i-th one. Figure 15 shows the Cumulative Distribution Function (CDF) of the divergence of all the data centers over a period of two weeks. We filter out cases where the data centers were drained or taken offline.

As shown in Figure 15, for 80% of the cases, the divergence is less than 3% for the stateless traffic and 4% for the sticky traffic. For the stateless web service, the divergence mainly comes from continuous deployment which restarts service instances and causes imbalance (§4.1).

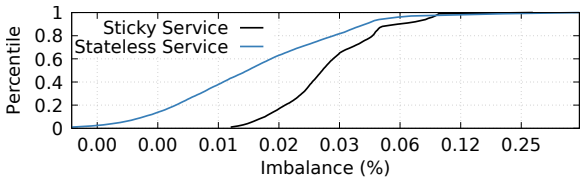
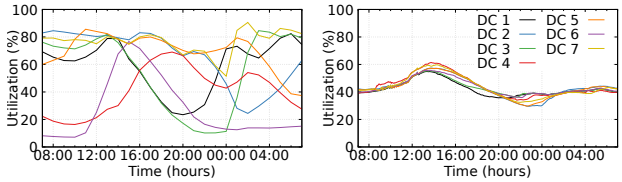


Figure 15. CDF of divergence from perfect utilization balancing for a stateless service and a sticky service over two weeks.



(a) Closest-datacenter-first **(b)** Latency-aware balancing

Figure 16. Frontend utilization of different data centers for Facebook's mobile service, Facebook Lite, with (a) closest-datacenter-first and (b) balancing with latency optimization.

Facebook Lite does not use continuous deployment but has slightly larger divergence. This is because Taiji does not disrupt unmovable traffic objects, but only controls new sessions. Therefore, the convergence is in general slower than for stateless traffic. On the other hand, by modeling session stickiness using unmovable objects, Taiji limits the divergence to within 4% for 80% of the cases.

4.6.2 Versatile Policy Choices

Taiji allows services to effectively experiment with different traffic management policies. Typically, the steps for a service to change its original policy include (1) setting the configuration (constraints, objectives and safety guards, §3.1.2), (2) running a load test and a drain test for validation, and (3) baking and monitoring in production.

Figure 16 shows the frontend utilization of Facebook Lite with two different policies. We started to use Taiji to manage its traffic from static edge-to-datacenter configuration (§1). Facebook Lite initially used a closest-datacenter-first policy for latency optimization. Figure 16a shows the daily pattern of frontend utilization in geo-located data centers—the utilization keeps increasing as the traffic from edge nodes in the same region climbs to the peak time, until hitting a predefined utilization threshold.

Facebook Lite later switched to the latency-aware balancing policy after evaluating the tradeoff between manageability and latency reduction. The utilization with a balancing policy is shown in Figure 16b. According to the service developers, the balancing policy makes it easier to understand the traffic behavior at the data center, with only slight latency increases (less than 20 milliseconds in most times of a day) as discussed in §4.1.3.

5 Experiences and Lessons Learned

Customizing load balancing strategy is key to managing infrastructure utilization. We find that different services, such as Facebook, Facebook Lite, and Messaging, optimize for different user experiences. Each service optimizes for some combination of infrastructure resources such as system CPU, memory, network bandwidth, network latency, etc. We recommend constructing a unified system that allows each service to provide a target load balancing function, rather than constructing multiple load balancing systems.

Build systems that keep pace with infrastructure evolution. We find that product evolution, data center footprint expansion, hardware generation changes, backbone network deployments and other factors trigger policy changes. For instance, we recently started deploying Skylake processors in our data centers, which resulted in our services needing to run effectively on three process generations. As we continually refresh hardware, the ratio of machines with different processors will keep changing. We have built tooling that leverages Taiji to run simulations and experiments to assess how changing the processor generation affects the utilization of backend systems, and all intervening dependencies to web servers, for different traffic mixes of Facebook, Facebook Lite, Messaging, etc. Thus, Tajii allows us to reason about how to distribute traffic when capacity, product and other changes occur in production.

Keep debuggability in mind. We often need to answer the question: "Why did Taiji choose to route traffic from this edge node to that data center"? Since Taiji continually updates the routing table, we focused on making every traffic shift understandable to human operators. In addition to displaying the "before" and "after" state of each traffic shift, we highlight which inputs changed, the magnitude of change, etc. This investment in transparency, beyond the norm of good log messages and alerts/monitoring, has allowed both service owners and site reliability engineers to build trust in Taiji.

Build tools to simplify operations. We have found it useful to build tools to solve specific problems, such as a command line tool that allows us to manually intervene and modify the routing table, which has proven useful in myriad situations ranging from complex network upgrades to failure mitigations. We find that it is preferable to build tools or custom configurations to solve problems than adding complexity to the traffic management system.

6 Limitation and Discussion

Taiji might increase the latency for some users during peak load. Balancing data center utilization sometimes requires sending users to farther data centers. For instance, during Europe's peak, Taiji might intentionally move some European traffic to our East Coast data centers to better balance capacity. Thus, while our overall infrastructure is better utilized, some users will experience additional latency.

Taiji only considers edge-to-datacenter latency. Service owners tend to care about the end-to-end latency. We have observed services where the backend processing latency dominates; for these services, routing traffic to the closest data center improves network latency, but can cause the backends to run at higher utilization and result in longer queuing and processing, offsetting the savings in network latency. In this case, a better policy is to balance data center utilization in a latency-aware manner. Taiji is not able to directly optimize for end-to-end latency, and instead relies on the service owner to configure Taiji based on their backends.

Taiji only controls the edge-to-datacenter routing. Taiji optimizes at the edge-to-datacenter request hop. However, there are separate systems for optimizing routing from a user's browser to the closest edge node, and for routing requests within a data center to specific backend machines. There may be untapped potential in Taiji to have end-to-end control of a user's request right from their browser to an edge node to a data center to backend machines.

7 Related Work

There are limited publications on managing user traffic at the edge for modern Internet services. User traffic routing from edge nodes to data centers is fundamentally different from content distribution over public WANs [14, 23, 32, 33, 39, 56, 57, 61]. For dynamic content, the constraints do not come from the capacity of network links, but from the capacity of data centers in terms of computation and response generation. The dedicated private backbone of modern Internet services [18–20, 22, 24, 29, 44] eliminate the bottlenecks of edge-to-datacenter network transmission. On the other hand, the subsystems deployed in the data centers are complex, dynamic, and interdependent. Taiji is designed for managing user traffic requesting dynamic content for modern Internet infrastructure instead of static content distribution.

Taiji is complementary to load balancing *within a data center*, including both L4 and L7 solutions [2, 6, 7, 10–12, 25, 27, 37, 40, 42, 49]. Taiji manages edge-to-datacenter traffic—once the traffic hits the frontends, it will be further distributed by load balancing inside the data center, as shown in Figure 3.

Taiji is also complementary to overload control including admission control and data quality tradeoffs [4, 5, 55, 62]. Taiji proactively avoids overloading backends with backend safety as a first class principle (§3.3.4).

A common strategy for managing edge-to-datacenter traffic is to route edge traffic to the nearest data centers with available capacity while autoscaling service capacity [3, 8, 31, 44]. We show that managing edge-to-datacenter traffic brings a number of benefits in terms of reliability and site utilization with good performance.

Many cloud platforms have started to provide customers with global traffic routing configuration, including Azure Front Door [38] and AWS Global Accelerator [44]. These

services allow customers to configure how user traffic is routed to each service endpoint in data centers. We show that static configurations of traffic routing is insufficient in accommodating dynamic user traffic for Internet services at scale. Taiji shows that a dynamic system can be built based on existing configuration interfaces, which bring meaningful benefits to the site reliability, efficiency, and manageability.

The idea of clustering requests of connected users was first explored in Social Hash [26, 47]. However, we find that Social Hash alone significantly misses out on locality: 1) Social Hash treats each bucket independently and misses the connections between the buckets, and 2) Social Hash has no knowledge of the requirements of edge-to-datacenter traffic management—it is unclear how to assign buckets under dynamic traffic adjustment. Compared with Social Hash [47], connection-aware routing significantly increases the connection locality by 20% and reduces 17% query load on our backend storage.

8 Conclusion

This paper shows that managing edge-to-datacenter traffic has important implications on data center resource utilization and user experience in modern Internet services with a global user base. We present Taiji, a new system for managing global user traffic for large-scale Internet services at Facebook. Taiji has successfully achieved its two design objectives: 1) balancing the utilization of data centers and 2) minimizing network latency of user requests. Furthermore, Taiji optimizes backend utilization by improving locality in user traffic routing. Taiji has been used in production at Facebook for more than four years, and is an important infrastructure service that enables the global deployment of several large-scale user-facing product services.

Acknowledgment

We thank the reviewers and our shepherd, Sujata Banerjee, for comments that improved this paper. We thank Jason Flinn, Mayank Pundir, Alex Gyori, Chunqiang Tang, and Dan Meredith for the discussions and feedback on the paper. We thank Yun Jin, Ashish Shah, Aravind Narayanan, and Yee Jiun Song for managing the team. We also thank Yuliy Pisetsky, Anca Agape, Ayelet Regev Dabah, Sean Barker, Alison Huang, Dimitris Sarlis, and the numerous engineers at Facebook who have helped us understand various systems and offered suggestions for improving user traffic management.

Appendix

The appendix provides the formal model of the balancing policy used by Taiji, as described in §3.1.2.

Let D={d} be the set of data centers, E={e} be the set of edges, and $x'_{e,d}$ be the new *traffic weights* for proportion of traffic to send from e to d in the next epoch. The following constraints ensures $x'_{e,d}$ are non-negative:

$$x'_{e,d} \quad \geq \quad 0 \tag{1}$$

The following constraints ensures weights are valid proportions between 0 and 1 which is applied for all $e \in E$:

$$\sum_{d \in D} x'_{e,d} \quad = \quad 1 \tag{2}$$

Let u'_d be our estimate of utilization given $x'_{e,d}$, $u_d \in [0, 1]$ be the utilization measured at data center d, t_e be the current load measures from edge node e, and $x_{e,d}$ be the values chosen in the current epoch. The following expression captures the relationship between traffic shift choices and the estimated utilization for all $d \in D$:

$$u'_d = u_d \cdot \left(1 + \frac{\sum_{e \in E}(t_e \cdot x'_{e,d}) - \sum_{e \in E}(t_e \cdot x_{e,d})}{\sum_{e \in E}(t_e \cdot x_{e,d})}\right) \tag{3}$$

Primary objective: Balancing. Balancing can be expressed as minimizing the highest data center utilization:

$$\text{minimize} \qquad \max_{d \in D} u'_d \tag{4}$$

Secondary objective: Squared latency minimization. Let $l_{e,d}$ be the average latency from edge e to data center d. Minimizing squared latency avoids worst-case scenarios:

$$\text{minimize} \qquad \sum_{e \in E, d \in D} x'_{e,d} \cdot t_e \cdot l^2_{e,d} \tag{5}$$

Onloading constraints. Let M be an onloading constant bound (0.04 in production). The following bounds how much new utilization one data center can receive during an epoch:

$$u'_d - u_d \quad \leq \quad M \tag{6}$$

Capacity constraints. The following enforces basic capacity concerns to ensure no data center is over capacity:

$$u'_d \quad \leq \quad 1 \tag{7}$$

The above problem can be posed as a linear program where $x'_{e,d}$ and u'_d are variables while everything else is a constant read from the monitoring system as input to the problem. The problem is of the size $O(|E| \cdot |D|)$ in terms of constraints and variables. For the objectives, we place a large constant coefficient on the primary objective and then sum it with the secondary objective. The primary objective coefficient is set large enough such that balance is always achieved.

We transform our problem by breaking user traffic from each edge node into N discrete traffic objects $y_{e,d,i}$ (i = 1, ..., N), each as 0/1 binary variables. Any legal assignments have to satisfy the following for all $e \in E$ and i = 1, ..., N:

$$\sum_{d \in D} y_{e,d,i} \quad = \quad 1 \tag{8}$$

We then define the relationship between $x'_{e,d}$ and $y_{e,d,i}$:

$$x'_{e,d} \quad = \quad \frac{\sum_{i=1,\ldots,N} y_{e,d,i}}{N} \tag{9}$$

The above problem can be efficiently solved by the assignment solver described in §3.1.2.

References

[1] Muthukaruppan Annamalai, Kaushik Ravichandran, Harish Srinivas, Igor Zinkovsky, Luning Pan, Tony Savor, David Nagle, and Michael Stumm. 2018. Sharding the Shards: Managing Datastore Locality at Scale with Akkio. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI'18)*. Carlsbad, CA, USA.

[2] J. Taveira Araújo, Lorenzo Saino, Lennert Buytenhek, and Raul Landa. 2018. Balancing on the Edge: Transport Affinity without Network State. In *Proceedings of the 15th USENIX Symposium on Networked Systems Design and Implementation (NSDI'18)*. Renton, WA, USA.

[3] Cooper Bethea, Gráinne Sheerin, Jennifer Mace, Ruth King, Gary Luo, and Gary O'Connor. 2018. Managing Load. *The Site Reliability Workbook: Practical Ways to Implement SRE, Chapter 11, O'Reilly Media Inc.* (Aug. 2018), 224–243.

[4] Ludmila Cherkasova and Peter Phaal. 1998. *Session Based Admission Control: A Mechanism for Improving the Performance of an Overloaded Web Server*. Technical Report HPL-98-119. Hewlett-Packard Company.

[5] Alejandro Forero Cuervo. 2016. Handling Overload. *Site Reliability Engineering: How Google Runs Production Systems, Chapter 21, O'Reilly Media Inc.* (April 2016), 231–246. https://landing.google.com/sre/sre-book/chapters/handling-overload/.

[6] Alejandro Forero Cuervo. 2016. Load Balancing in the Datacenter. *Site Reliability Engineering: How Google Runs Production Systems, Chapter 20, O'Reilly Media Inc.* (April 2016), 231–246. https://landing.google.com/sre/sre-book/chapters/load-balancing-datacenter/.

[7] Daniel E. Eisenbud, Cheng Yi, Carlo Contavalli, Cody Smith, Roman Kononov, Eric Mann-Hielscher, Ardas Cilingiroglu, Bin Cheyney, Wentao Shang, and Jinnah Dylan Hosein. 2016. Maglev: A Fast and Reliable Software Network Load Balancer. In *Proceedings of the 13th USENIX Symposium on Networked Systems Design and Implementation (NSDI'16)*. Santa Clara, CA, USA.

[8] Ashley Flavel, Pradeepkumar Mani, David A. Maltz, Nick Holt, Jie Liu, Yingying Chen, and Oleg Surmachev. 2015. FastRoute: A Scalable Load-Aware Anycast Routing Architecture for Modern CDNs. In *Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI'15)*. Oakland, CA, USA.

[9] Daniel Ford, François Labelle, Florentina I. Popovici, Murray Stokely, Van-Anh Truong, Luiz Barroso, Carrie Grimes, and Sean Quinlan. 2010. Availability in Globally Distributed Storage Systems. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation (OSDI'10)*. Vancouver, BC, Canada.

[10] Rohan Gandhi, Y. Charlie Hu, Cheng kok Koh, Hongqiang Liu, and Ming Zhang. 2015. Rubik: Unlocking the Power of Locality and Endpoint Flexibility in Cloud Scale Load Balancing. In *Proceedings of the 2015 USENIX Annual Technical Conference (USENIX ATC'15)*. Santa Clara, CA.

[11] Rohan Gandhi, Y. Charlie Hu, and Ming Zhang. 2016. Yoda: A Highly Available Layer-7 Load Balancer. In *Proceedings of the 11th European Conference on Computer Systems (EuroSys'16)*. London, United Kingdom.

[12] Rohan Gandhi, Hongqiang Harry Liu, Y. Charlie Hu, Guohan Lu, Jitendra Padhye, Lihua Yuan, and Ming Zhang. 2014. Duet: Cloud Scale Load Balancing with Hardware and Software. In *Proceedings of the 2014 ACM SIGCOMM Conference (SIGCOMM'14)*. Chicago, Illinois, USA.

[13] Aditya Ganjam, Junchen Jiang, Xi Liu, Vyas Sekarx, Faisal Siddiqi, Ion Stoica, Jibin Zhan, and Hui Zhang. 2015. C3: Internet-Scale Control Plane for Video Quality Optimization. In *Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI'15)*. Oakland, CA, USA.

[14] David K. Goldenberg, Lili Qiu, Haiyong Xie, Yang Richard Yang, and Yin Zhang. 2004. Optimizing Cost and Performance for Multihoming. In *Proceedings of the 2004 ACM SIGCOMM Conference (SIGCOMM'04)*. Portland, Oregon, USA.

[15] Ramesh Govindan, Ina Minei, Mahesh Kallahalla, Bikash Koley, and Amin Vahdat. 2016. Evolve or Die: High-Availability Design Principles Drawn from Googles Network Infrastructure. In *Proceedings of the 2016 ACM SIGCOMM Conference (SIGCOMM'16)*. Florianópolis, Brazil.

[16] Haryadi S. Gunawi, Mingzhe Hao, Tanakorn Leesatapornwongsa, Tiratat Patana-anake, Thanh Do, Jeffry Adityatama, Kurnia J. Eliazar, Agung Laksono, Jeffrey F. Lukman, Vincentius Martin, and Anang D. Satria. 2014. What Bugs Live in the Cloud? A Study of 3000+ Issues in Cloud Systems. In *Proceedings of the 5th ACM Symposium on Cloud Computing (SoCC'14)*. Seattle, WA, USA.

[17] Haryadi S. Gunawi, Mingzhe Hao, Riza O. Suminto, Agung Laksono, Anang D. Satria, Jeffry Adityatama, and Kurnia J. Eliazar. 2016. Why Does the Cloud Stop Computing? Lessons from Hundreds of Service Outages. In *Proceedings of the 7th ACM Symposium on Cloud Computing (SoCC'16)*. Santa Clara, CA, USA.

[18] Nikola Gvozdiev, Stefano Vissicchio, Brad Karp, and Mark Handley. 2018. On low-latency-capable topologies, and their impact on the design of intra-domain routing. In *Proceedings of the 2018 ACM SIGCOMM Conference (SIGCOMM'18)*. Budapest, Hungary.

[19] Chi-Yao Hong, Srikanth Kandula, Ratul Mahajan, Ming Zhang, Vijay Gill, Mohan Nanduri, and Roger Wattenhofer. 2013. Achieving High Utilization with Software-driven WAN. In *Proceedings of the 2013 ACM SIGCOMM Conference (SIGCOMM'13)*. Hong Kong, China.

[20] Chi-Yao Hong, Subhasree Mandal, Mohammad Al-Fares, Min Zhu, Richard Alimi, Kondapa Naidu B., Chandan Bhagat, Sourabh Jain, Jay Kaimal, Shiyu Liang, Kirill Mendelev, Steve Padgett, Faro Rabe, Saikat Ray, Malveeka Tewari, Matt Tierney, Monika Zahn, Jonathan Zolla, Joon Ong, and Amin Vahdat. 2018. B4 and After: Managing Hierarchy, Partitioning, and Asymmetry for Availability and Scale in Google's Software-defined WAN. In *Proceedings of the 2018 ACM SIGCOMM Conference (SIGCOMM'18)*. Budapest, Hungary.

[21] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. 2010. ZooKeeper: Wait-free Coordination for Internet-scale Systems. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference (USENIX ATC'10)*. Boston, MA.

[22] Sushant Jain, Alok Kumar, Subhasree Mandal, Joon Ong, Leon Poutievski, Arjun Singh, Subbaiah Venkata, Jim Wanderer, Junlan Zhou, Min Zhu, Jon Zolla, Urs Hölzle, Stephen Stuart, and Amin Vahdat. 2013. B4: Experience with a Globally-deployed Software Defined WAN. In *Proceedings of the 2013 ACM SIGCOMM Conference (SIGCOMM'13)*. Hong Kong, China.

[23] Wenjie Jiang, Rui Zhang-Shen, Jennifer Rexford, and Mung Chiang. 2009. Cooperative Content Distribution and Traffic Engineering in an ISP Network. In *Proceedings of the 11th International Joint Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '09)*. Seattle, WA, USA.

[24] Mikel Jimenez and Henry Kwok. 2017. Building Express Backbone: Facebook's new long-haul networks. https://code.fb.com/data-center-engineering/building-express-backbone-facebook-s-new-long-haul-network/.

[25] Theo Julienne. 2018. GLB: GitHub's open source load balancer. https://githubengineering.com/glb-director-open-source-load-balancer/.

[26] Igor Kabiljo, Brian Karrer, Mayank Pundir, Sergey Pupyrev, and Alon Shalita. 2017. Social Hash Partitioner: A Scalable Distributed Hypergraph Partitioner. *Journal Proceedings of the VLDB Endowment* 10, 11 (Aug. 2017).

[27] Nanxi Kang, Monia Ghobadi, John Reumann, Alexander Shraer, and Jennifer Rexford. 2015. Efficient Traffic Splitting on Commodity Switches. In *Proceedings of the 11th ACM Conference on Emerging Networking Experiments and Technologies (CoNEXT'15)*. Heidelberg, Germany.

[28] Kimberly Keeton, Cipriano Santos, Dirk Beyer, Jeffrey Chase, and John Wilkes. 2002. Designing for Disasters. In *Proceedings of the 3rd USENIX Conference on File and Storage Technologies (FAST'04)*. San Francisco,

CA, USA.

[29] Yousef Khalidi. 2017. How Microsoft builds its fast and reliable global network. https://azure.microsoft.com/en-us/blog/how-microsoft-builds-its-fast-and-reliable-global-network/.

[30] Kripa Krishnan. 2012. Weathering the Unexpected. *Communications of the ACM (CACM)* 55, 11 (Nov. 2012), 48–52.

[31] Piotr Lewandowski. 2016. Load Balancing at the Frontend. *Site Reliability Engineering: How Google Runs Production Systems, Chapter 19, O'Reilly Media Inc.* (April 2016), 223–229. https://landing.google.com/sre/sre-book/chapters/load-balancing-frontend/.

[32] Hongqiang Harry Liu, Raajay Viswanathan, Matt Calder, Aditya Akella, Ratul Mahajan, Jitendra Padhye, and Ming Zhang. 2016. Efficiently Delivering Online Services over Integrated Infrastructure. In *Proceedings of the 13th USENIX Symposium on Networked Systems Design and Implementation (NSDI'16)*. Santa Clara, CA, USA.

[33] Hongqiang Harry Liu, Ye Wang, Yang Richard Yang, Hao Wang, and Chen Tian. 2012. Optimizing Cost and Performance for Content Multihoming. In *Proceedings of the 2012 ACM SIGCOMM Conference (SIGCOMM'12)*. Helsinki, Finland.

[34] Anil Mallapur and Michael Kehoe. 2017. TrafficShift: Load Testing at Scale. https://engineering.linkedin.com/blog/2017/05/trafficshift--load-testing-at-scale.

[35] Ben Maurer. 2015. Fail at Scale: Reliability in the Face of Rapid Change. *Communications of the ACM (CACM)* 58, 11 (Nov. 2015), 44–49.

[36] Justin Meza, Tianyin Xu, Kaushik Veeraraghavan, and Yee Jiun Song. 2018. A Large Scale Study of Data Center Network Reliability. In *Proceedings of the 2018 ACM Internet Measurement Conference (IMC'18)*. Boston, MA, USA.

[37] Rui Miao, Hongyi Zeng, Changhoon Kim, Jeongkeun Lee, and Minlan Yu. 2017. SilkRoad: Making Stateful Layer-4 Load Balancing Fast and Cheap Using Switching ASICs. In *Proceedings of the 2017 ACM SIGCOMM Conference (SIGCOMM'17)*. Los Angeles, CA, USA.

[38] Microsoft Docs. 2019. Azure Front Door Service Documentation. https://docs.microsoft.com/en-us/azure/frontdoor/.

[39] Srinivas Narayana, Joe Wenjie Jiang, Jennifer Rexford, and Mung Chiang. 2012. *To Coordinate Or Not To Coordinate? Wide-Area Traffic Management for Data Centers.* Technical Report TR-998-15. Department of Computer Science, Princeton University.

[40] Vladimir Olteanu, Alexandru Agache, Andrei Voinescu, and Costin Raiciu. 2018. Stateless Datacenter Load-balancing with Beamer. In *Proceedings of the 15th USENIX Symposium on Networked Systems Design and Implementation (NSDI'18)*. Renton, WA, USA.

[41] David Oppenheimer, Archana Ganapathi, and David A. Patterson. 2003. Why Do Internet Services Fail, and What Can Be Done About It?. In *Proceedings of the 4th Conference on USENIX Symposium on Internet Technologies and Systems (USITS'03)*. Seattle, WA, USA.

[42] Parveen Patel, Deepak Bansal, Lihua Yuan, Ashwin Murthy, Albert Greenberg, David A. Maltz, Randy Kern, Hemant Kumar, Marios Zikos, Hongyu Wu, Changhoon Kim, and Naveen Karri. 2013. Ananta: Cloud Scale Load Balancing. In *Proceedings of the ACM 2013 SIGCOMM Conference*. Hong Kong, China.

[43] Tuomas Pelkonen, Scott Franklin, Justin Teller, Paul Cavallaro, Qi Huang, Justin Meza, and Kaushik Veeraraghavan. 2015. Gorilla: A Fast, Scalable, In-Memory Time Series Database. In *Proceedings of the 41st International Conference on Very Large Data Bases (VLDB'15)*. Kohala Coast, HI, USA.

[44] Shaun Ray. 2018. AWS Global Accelerator for Availability and Performance. https://aws.amazon.com/global-accelerator/.

[45] Chuck Rossi, Elisa Shibley, Shi Su, Kent Beck, Tony Savor, and Michael Stumm. 2016. Continuous Deployment of Mobile Software at Facebook. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE'16)*. Seattle, WA, USA.

[46] Brandon Schlinker, Hyojeong Kim, Timothy Cui, Ethan Katz-Bassett, Harsha V. Madhyastha, Italo Cunha, James Quinn, Saif Hasan, Petr Lapukhov, and Hongyi Zeng. 2017. Engineering Egress with Edge Fabric: Steering Oceans of Content to the World. In *Proceedings of the 2017 ACM SIGCOMM Conference (SIGCOMM'17)*. Los Angeles, CA, USA.

[47] Alon Shalita, Brian Karrer, Igor Kabiljo, Arun Sharma, Alessandro Presta, Aaron Adcock, Herald Kllapi, and Michael Stumm. 2016. Social Hash: An Assignment Framework for Optimizing Distributed Systems Operations on Social Networks. In *Proceedings of the 13th USENIX Symposium on Networked Systems Design and Implementation (NSDI'16)*. Santa Clara, CA, USA.

[48] Alex Sherman, Philip A. Lisiecki, Andy Berkheimer, and Joel Wein. 2005. ACMS: The Akamai Configuration Management System. In *Proceedings of the 2nd Conference on Symposium on Networked Systems Design and Implementation (NSDI'05)*. Boston, MA, USA.

[49] Daniel Sommermann and Alan Frindell. 2014. Introducing Proxygen, Facebook's C++ HTTP framework. https://code.facebook.com/posts/1503205539947302.

[50] Chunqiang Tang, Thawan Kooburat, Pradeep Venkatachalam, Akshay Chander, Zhe Wen, Aravind Narayanan, Patrick Dowell, and Robert Karl. 2015. Holistic Configuration Management at Facebook. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP'15)*. Monterey, CA, USA.

[51] Tony Savor and Mitchell Douglas and Michael Gentili and Laurie Williams and Kent Beck and Michael Stumm. 2016. Continuous Deployment at Facebook and OANDA. In *Proceedings of the 38th IEEE/ACM International Conference on Software Engineering Companion (ICSE'16)*. Austin, TX, USA.

[52] Ben Treynor, Mike Dahlin, Vivek Rau, and Betsy Beyer. 2017. The Calculus of Service Availability. *Communications of the ACM (CACM)* 60, 9 (Sept. 2017), 42–47.

[53] Kaushik Veeraraghavan, Justin Meza, David Chou, Wonho Kim, Sonia Margulis, Scott Michelson, Rajesh Nishtala, Daniel Obenshain, Dmitri Perelman, and Yee Jiun Song. 2016. Kraken: Leveraging Live Traffic Tests to Identify and Resolve Resource Utilization Bottlenecks in Large Scale Web Services. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI'16)*. Savannah, GA, USA.

[54] Kaushik Veeraraghavan, Justin Meza, Scott Michelson, Sankaralingam Panneerselvam, Alex Gyori, David Chou, Sonia Margulis, Daniel Obenshain, Shruti Padmanabha, Ashish Shah, Yee Jiun Song, and Tianyin Xu. 2018. Maelstrom: Mitigating Datacenter-level Disasters by Draining Interdependent Traffic Safely and Efficiently. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI'18)*. Carlsbad, CA, USA.

[55] Matt Welsh and David Culler. 2003. Adaptive Overload Control for Busy Internet Servers. In *Proceedings of the 4th Conference on USENIX Symposium on Internet Technologies and Systems (USITS'03)*. Seattle, WA.

[56] Patrick Wendell, Joe Wenjie Jiang, Michael J. Freedman, and Jennifer Rexford. 2010. DONAR: Decentralized Server Selection for Cloud Services. In *Proceedings of the 2010 ACM SIGCOMM Conference (SIGCOMM'10)*. New Delhi, India.

[57] Hong Xu and Baochun Li. 2013. Joint Request Mapping and Response Routing for Geo-distributed Cloud Services. In *Proceedings of the 32nd IEEE International Conference on Computer Communications (INFOCOM'13)*. Turin, Italy.

[58] Tianyin Xu, Xinxin Jin, Peng Huang, Yuanyuan Zhou, Shan Lu, Long Jin, and Shankar Pasupathy. 2016. Early Detection of Configuration Errors to Reduce Failure Damage. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI'16)*. Savannah, GA, USA.

[59] Tianyin Xu, Jiaqi Zhang, Peng Huang, Jing Zheng, Tianwei Sheng, Ding Yuan, Yuanyuan Zhou, and Shankar Pasupathy. 2013. Do Not Blame Users for Misconfigurations. In *Proceedings of the 24th Symposium on Operating Systems Principles (SOSP'13)*. Farmington, PA, USA.

[60] 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. 2017. Taking the Edge off with Espresso: Scale, Reliability and Programmability for Global Internet Peering. In *Proceedings of the 2017 ACM SIGCOMM Conference (SIGCOMM'17)*. Los Angeles, CA, USA.

[61] Zheng Zhang, Ming Zhang, Albert Greenberg, Y. Charlie Hu, Ratul Mahajan, and Blaine Christian. 2010. Optimizing Cost and Performance in Online Service Provider Networks. In *Proceedings of the 7th USENIX Symposium on Networked Systems Design and Implementation (NSDI'10)*. San Jose, CA, USA.

[62] Hao Zhou, Ming Chen, Qian Lin, Yong Wang, Xiaobin She, Sifan Liu, Rui Gu, Beng Chin Ooi, and Junfeng Yang. 2018. Overload Control for Scaling WeChat Microservices. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC'18)*. Carlsbad, CA, USA.