Kraken: Leveraging Live Traffic Tests to Identify and Resolve Resource Utilization Bottlenecks in Large Scale Web Services

Kaushik Veeraraghavan Justin Meza David Chou Wonho Kim Sonia Margulis
Scott Michelson Rajesh Nishtala Daniel Obenshain Dmitri Perelman
Yee Jiun Song

Facebook Inc.

Abstract

Modern web services such as Facebook are made up of hundreds of systems running in geographically-distributed data centers. Each system needs to be allocated capacity, configured, and tuned to use data center resources efficiently. Keeping a model of capacity allocation current is challenging given that user behavior and software components evolve constantly.

Three insights motivate our work: (1) the live user traffic accessing a web service provides the most current target workload possible, (2) we can empirically test the system to identify its scalability limits, and (3) the user impact and operational overhead of empirical testing can be largely eliminated by building automation which adjusts live traffic based on feedback.

We build on these insights in Kraken, a new system that runs load tests by continually shifting live user traffic to one or more data centers. Kraken enables empirical testing by monitoring user experience (e.g., latency) and system health (e.g., error rate) in a feedback loop between traffic shifts. We analyze the behavior of individual systems and groups of systems to identify resource utilization bottlenecks such as capacity, load balancing, software regressions, performance tuning, and so on, which can be iteratively fixed and verified in subsequent load tests. Kraken, which manages the traffic generated by 1.7 billion users, has been in production at Facebook for three years and has allowed us to improve our hardware utilization by over 20%.

1 Introduction

Modern web services comprise software systems running in multiple data centers that cater to a global user base. At this scale, it is important to use all available data center resources as efficiently as possible. Effective resource utilization is challenging because:

- **Evolving workload**: The workload of a web service is constantly changing as its user base grows and new products are launched. Further, individual software systems might be updated several times a day [35] or even continually [27]. While modeling tools [20, 24, 46, 51] can estimate the initial capacity needs of a system, an evolving workload can quickly render models obsolete.
- **Infrastructure heterogeneity**: Constructing data centers at different points in time leads to a variety of networking topologies, different generations of hardware, and other physical constraints in each location that each affect how systems scale.
- **Changing bottlenecks**: Each data center runs hundreds of software systems with complex interactions that exhibit *resource utilization bottlenecks*, including performance regressions, load imbalance, and resource exhaustion, at a variety of scales from single servers to entire data centers. The sheer size of the system makes it impossible to understand all the components. In addition, these systems change over time, leading to different bottlenecks presenting themselves. Thus, we need a way to continually identify and fix bottlenecks to ensure that the systems scale efficiently.

Our key insight is that the live user traffic accessing a web service provides the most current workload possible, with natural phenomena like non-uniform request arrival rates. As a baseline, the web service must be capable of executing its workload within a preset latency and error threshold. Ideally, the web service should be capable of handling peak load (e.g., during Super Bowl) when unexpected bottlenecks arise, and still achieve good performance.

We propose Kraken, a new system that allows us to run live traffic load tests to accurately assess the capacity of a complex system. In building Kraken, we found that reliably tracking system health is the most important requirement for live traffic testing. But how do we select among thousands of candidate metrics? We aim to provide a good experience to all users by ensuring that a user served out of a data center undergoing a Kraken test has a comparable experience to users being served out of any other data center. We accomplish this goal with a light-weight and configurable monitoring component seeded with two topline metrics, the web server's 99^{th} percentile response time and HTTP fatal error rate, as reliable proxies for user experience.

We leverage Kraken as part of an iterative methodology to improve capacity utilization. We begin by running a load test that directs user traffic at a target cluster or region. A successful test concludes by hitting the utilization targets without crossing the latency or error rate thresholds. Tests can fail in two ways:

- We fail to hit the target utilization or exceed a preset threshold. This is the usual outcome following which we drill into test data to identify bottlenecks.
- Rarely, the load test results in a HTTP error spike or some similar unexpected failure. When this occurs, we analyze the test data to understand what monitoring or system understanding we were missing. In practice, sudden jumps in HTTP errors or other topline metrics almost never happen; the set of auxiliary metrics we add to our monitoring are few in number and are the result of small incidents rather than catastrophic failure.

Each test provides a probe into an initially uncharacterized system, allowing us to learn new things. An unsuccessful test provides data that allows us to either make the next test safer to run or increase capacity by removing a bottleneck. As tests are non-disruptive to users, we can run them regularly to determine both a data center's maximal load (e.g., requests per second, which we term the system's *capacity*), and continually identify and fix bottlenecks to improve system utilization.

Our first year of operating Kraken in production exposed several challenges. The first was that the systems often exhibited a non-linear response where a small traffic shift directed at a data center could trigger an error spike. Our follow-up was to check the health of all the major systems involved in serving user traffic to determine which ones were affected most during the test. This dovetailed into our second challenge which was that systems have complex dependencies so it was often unclear which system initially failed and then trigged errors in seemingly unrelated downstream subsystems.

We addressed both of these challenges by encouraging subsystem developers to identify system-specific counters for performance (e.g., response quality), error rate, and latency, that could be monitored during a test. We focused on these three metrics because they represent the contracts that clients of a service rely on—we have found that nearly every production system wishes to maintain or decrease its latency and error rate while maintaining or improving performance.

The third challenge was one of culture. Although we ensure the tests are safe for users, load tests put significant stress on many systems. Rather than treat load tests as painful events for systems to survive, we worked hard to create a collaborative environment where developers looked forward to load tests to better understand their systems. We planned tests ahead of time and communicated schedules widely to minimize surprise. We encouraged engineers to share tips on how to better monitor their systems, handle different failure scenarios, and build out levers to mitigate production issues. Further, we used a shared IRC channel to track tests and debug issues live. Often, we would use tests as an opportunity for developers to test or validate improvements, which made them an active part of the testing process. Engaging engineers was critical for success; engaged engineers improve their systems quickly and support a frequent test cadence, allowing us to iterate quickly.

Our initial remediations to resolve bottlenecks were mostly capacity allocations to the failing system. As our understanding of Facebook's systems has improved, our mitigation toolset has also expanded to include configuration and performance tuning, load balancing improvements, profiling-guided software changes and occasionally a system redesign. Our monitoring has also improved and as of October 2016, we have identified metrics from 23 critical systems that are bellwethers of non-linear behavior in our infrastructure. We use these metrics as inputs to control Kraken's behavior.

Kraken has been in use at Facebook for over three years and has run thousands of load tests on production systems using live user traffic. Our contributions:

- Kraken is the first live traffic load testing framework to test systems ranging in size from individual servers to entire data centers, to our knowledge.
- Kraken has allowed us to iteratively increase the capacity of Facebook's infrastructure with an empirical approach that improves the utilization of systems at every level of the stack.
- Kraken has allowed us to identify and remediate regressions, address load imbalance and resource exhaustion across Facebook's fleet. Our initial tests stopped at about 70% of theoretical capacity, but now routinely exceed 90%, providing a 20% increase in request serving capacity.

The Kraken methodology is not applicable to all services. These assumptions/caveats underpin our work:

- *Assumption: stateless servers.* We assume that *stateless* servers handle requests without using *sticky sessions* for server affinity. Stateless web servers provide high availability despite system or network failures by routing requests to any available machine. Note that stateless servers may still communicate with services that *are* stateful (such as a database) when handling a request.
- *Caveat: load must be controllable by re-routing requests.* Subsystems that are global in nature may be insensitive to where load enters the system. Such systems may include batch processors, message queues, storage, etc. Kraken is not designed as a comprehensive capacity assessment tool for every

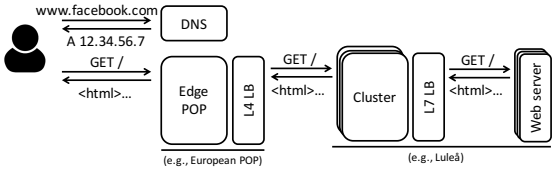


Figure 1: When a user sends a request to Facebook, a DNS resolver points the request at an edge point-of-presence (POP) close to the user. A L4 and L7 load balancer forward the request to a particular data center and a web server respectively.

subsystem. These systems often need more specialized handling to test their limits.

- *Assumption: downstream services respond to shifts in upstream service load.* Consider the case of a web server querying a database. A database with saturated disk bandwidth affects the number of requests served by the web server. This observation extends to other system architectures such as aggregator–leaf where a stateless aggregation server collects data from a pool of leaf servers to service a request.

2 The case for live traffic load tests

Broadly, two common approaches exist for identifying resource utilization bottlenecks: (1) *load modeling (or simulation)* and (2) *load testing (or benchmarking)*.

Load modeling relies on analytical models of data center resources to examine the trade-offs between performance, reliability, and energy-efficiency [11, 13, 18, 40]. We argue that it is infeasible to accurately model a large scale web service's capacity given their evolving workload, frequent software release cycles, and complexity of dependencies [27, 35], Alternatively, load test suites such as TPC-C [44], YCSB [49] and JouleSort [34] use system-level benchmarks to measure the load a system can sustain. Unfortunately, their synthetic workloads only cover very specific use cases, e.g., TPC-C performs a certain set of queries on a SQL database with a fixed set of inputs. An alternate design choice is to use *shadow* traffic where an incoming request is logged and replayed in a test environment. For the web server use case, most operations have side-effects that propagate deep into the system. Shadow tests must not trigger these side effects, as doing so can alter user state. Stubbing out side effects for shadow testing is not only impractical due to frequent changes in server logic, but also reduces the fidelity of the test by not stressing dependencies that would have otherwise been affected.

In contrast, Kraken uses *live* traffic to perform load tests. We prefer live traffic tests because:

- Live user traffic is a fully representative workload that consists of both read and write requests with a non-uniform arrival pattern, and traffic bursts.
- Live traffic tests can be run on production systems without requiring alternate test setups. Further, live traffic tests can expose bottlenecks that arise due to complex system dependencies, which are hard to reproduce in small scale test setups.
- Live traffic load tests on production systems have the implicit benefit of forcing teams to harden their systems to handle traffic bursts, overloads, etc., thus increasing the system's resilience to faults.

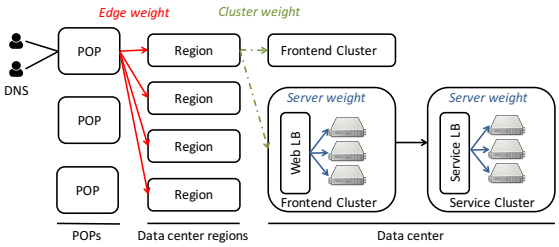


Figure 2: This figure provides an overview of the traffic management components at Facebook. User requests arrive at Edge POPs (points-of-presence). A series of weights defined at the edge, cluster, and server levels are used to route user requests from a POP to a web server in one of Facebook's data centers.

Safety is a key constraint when working with live traffic on a production system. We identify two problematic situations: (1) *internal* faults in system operation, such as violation of performance, reliability, or other constraints; and (2) *external* faults that result in capacity reduction due to, for example, network partitions, or power loss. Both situations require careful monitoring and fast traffic adjustment to safeguard the production system.

3 Design

Kraken is constructed as a feedback loop that shifts user traffic to evaluate the capacity of the system under test and identify resource utilization bottlenecks.

3.1 Traffic shifting

Figure 1 provides an overview of how a user request to Facebook is served. The user's request is sent to their ISP, which contacts a DNS resolver to map the URL to an IP address. This IP address maps to one of tens of edge point-of-presence (POP) locations distributed worldwide. A POP consists of a small number of servers on the edge of the network typically co-located with a local internet service provider. The user's SSL session is terminated in a POP at a L7 load balancer, which then forwards the request to one of the data centers.

At Facebook, we group 1–3 data centers in close proximity into a "region". Within each data center, we group machines into one or more logical "frontend" clusters of web servers, "backend" clusters of storage systems, and

multiple "service" clusters. In this paper, we define a "service" as a set of sub-systems that provide a particular product either internally within Facebook's infrastructure or externally to end users. Each cluster has a few thousand generally heterogeneous machines. Many services span clusters, but web server deployments are confined to a single cluster.

As Figure 2 shows, the particular frontend cluster that a request is routed to depends on two factors: (1) the *edge weight* from a POP to a region, and (2) the *cluster weight* assigned to each frontend cluster in a region. To understand why we need edge weights, consider a request from a user in Hamburg that is terminated at a hypothetical POP in Europe. This POP might prefer forwarding user requests to the Luleå, Sweden region rather than Forest City, North Carolina to minimize latency, implying that the Europen POP could assign a higher edge weight to Luleå than Forest City. A data center might house multiple frontend clusters with machines from different hardware generations. The capability of a Haswell cluster will exceed that of a Sandybridge cluster, resulting in differing cluster weights as well as individual servers being assigned different *server weights*.

When a request reaches a frontend cluster, an L7 load balancer forwards the request to one of many thousands of web servers. The web server may communicate with tens or hundreds of services residing in one or more service clusters to gather the data required to generate a response. Finally, the web server sends the response back to the POP, which forwards the response to the user.

Because web servers and some services in this architecture are stateless, any such server can handle any request bound for it or its peers in the same service. This implies that edge weights, cluster weights, and server weights can be programmed to easily and quickly change their destinations for requests. This allows us to programmatically shift different amounts of load to a particular region or cluster. We use live traffic shifting as the mechanism to manipulate load on different resources. Shifting live traffic allows us to gauge aggregate system capacity in a realistic environment.

3.2 Monitoring

The most important requirements in live traffic testing are reliable metrics that track the health of the system. At Facebook, we use Gorilla [33], a time series database that provides a fast way to store and query aggregated metrics. Our intuition when developing Kraken was that it could query Gorilla for the metrics of all important systems and use the results to compute the next traffic shift.

Our intuition proved impractical as Facebook is composed of thousands of systems, each of which export dozens of counters tracking their input request rate, internal state, and output. One of the goals of Kraken is to gauge the true capacity of the system. If we prioritized all systems equally and tried to ensure that every system operated within its ideal performance or reliability envelope, our focus would shift to constantly tuning individual systems rather than the overall user experience. This would hurt our ability to identify the real bottlenecks to system capacity, and instead give us the infeasible challenge of improving hundreds of systems at once.

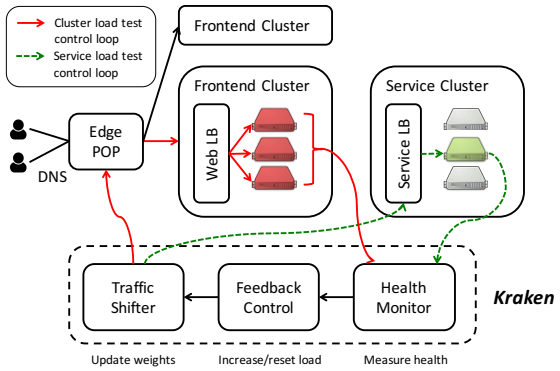


Figure 3: Kraken is a framework that allows us to load test large system units, such as clusters and entire data centers. To do so, Kraken shifts traffic from POPs to different frontend clusters while monitoring various health metrics to ensure they do not exceed allowable levels. The solid red line shows this control loop. In addition, Kraken can manage traffic in a more fine-grained manner to load test individual services composed of sets of servers, shown by the dotted green line.

Our insight was that Kraken running on a data center is equivalent to an operational issue affecting the site—in both cases our goal is to provide a good user experience. We use two metrics, the web servers' 99th percentile response time and HTTP fatal error rate, as proxies for the user experience, and determined in most cases this was adequate to avoid bad outcomes. Over time, we have added other metrics to improve safety such as the median queuing delay on web servers, the 99th percentile CPU utilization on cache machines, etc. Each metric has an explicit threshold demarcating the vitality of the system's health. Kraken stops the test when any metric reaches its limit, before the system becomes unhealthy.

3.3 Putting it all together: Kraken

Kraken employs a feedback loop where the traffic shifting module queries Gorilla for system health before determining the next traffic shift to the system under test. As Figure 3 shows, Kraken can shift traffic onto one or more frontend clusters by manipulating the edge and cluster weights. Notice that the generic nature of Kraken also allows it to function as a framework that can be applied to any subset of the overall system.

3.4 Capacity measurement methodology

Web servers in frontend clusters are the largest component of the infrastructure. Hence, a resource bottleneck such as a performance regression or a load imbalance in a smaller subsystem that limits the throughput of the web servers is highly undesirable. We can use Kraken to identify the peak utilization a single web server can achieve. A single web server is incapable of saturating the network or any backend services, so obtaining true capacity is not difficult. This empirical web server capacity multiplied by the number of web servers in a frontend cluster yields us a *theoretical frontend cluster capacity*.

At Facebook, we have found taking this theoretical number as truth does not yield good results. As we move to larger groups of webservers, complex system effects begin to dominate and our estimates tend to miss the true capacity, sometimes by a wide margin. Kraken allows us to continually run load tests on frontend clusters to obtain the empirical frontend cluster capacity and measure the deviation from the theoretical limit. We set ourselves the aggressive goal of operating our frontend clusters at (1) 93% of their theoretical capacity limits and (2) below a target pre-defined latency threshold, while concurrently keeping pace with product evolution, user growth and frequent software release cycles. On a less frequent basis, this methodology is also applied to larger units of capacity (regions comprising multiple data centers) with a similarly aggressive goal of 90%.

3.5 Identifying and fixing bottlenecks

We have learned that running live traffic load tests without compromising on system health is difficult. Succeeding at this approach has required us to invest heavily in instrumenting our software systems, using and building new debugging tools, and encouraging engineers to collaborate on investigating and resolving issues.

Extensive data collection allows us to debug problems in situ during a test and iterate quickly. We encourage systems to leverage Scuba [1], an in-memory database that supports storing arbitrary values and querying them in real time. If a system displays a non-linear response or other unexpected behavior during a test, an engineer on call for the system can alert us and we can debug the problem using Scuba data. We use standard tools like `perf` as well as custom tracing and visualization tools [14] in our debugging.

Initially, when we identified bottlenecks we mitigated them by allocating additional capacity to the failing system. In addition to being costly, we started encountering more difficult problems, such as load imbalance and network saturation, where adding capacity had diminishing returns. We would sometimes see cases where a single shard of a backend system got orders of magnitude more traffic than its peers, causing system saturation that adding capacity could not resolve. As we developed expertise, we arrived at a more sustainable approach that includes system reconfiguration, creating and deploying new load balancing algorithms, performance tuning, and, in rare cases, system redesign. We verify the efficacy of these solutions in subsequent tests and keep iterating so we can keep pace with Facebook's evolving workload.

4 Implementation

We next describe the implementation of Kraken and how it responds to various faults.

4.1 Traffic shifting module

At Facebook, we run Proxygen, an open source software L4 and L7 load balancer. Rather than rely on a static configuration, Proxygen running on L4 load balancers in a POP reads configuration files from a distributed configuration store [41]. This configuration file lists customized edge and cluster weights for each POP as shown in Figure 2. Proxygen uses these weights to determine the fraction of user traffic to direct at each frontend cluster. By adjusting cluster weights, we can increase the relative fraction of traffic a cluster receives compared to its peers. Using edge weights we can perform the same adjustment for regions.

Kraken takes as input the target of the test and then updates the routing file stored in the configuration store with this change. The configuration store notifies the Proxygen load balancers in a remote POP of the existence of the new configuration file. In practice, we've found that it can take up to 60 seconds for Kraken to update weights, the updated configuration to be delivered to the POP, and for Proxygen to execute the requested traffic shift. Since the monitoring system, Gorilla, aggregates metrics in 60 second intervals, it takes about 120 seconds end-to-end for Kraken to initiate a traffic shift and then verify the impact of that change on system load and health.

4.2 Health monitoring module

The health monitoring system receives as input the system being tested. It then queries Gorilla [33] for metrics that can be compared to their thresholds. Gorilla stores metrics as tuples consisting of $\langle$entity, key, value, timestamp$\rangle$. These tuples are either readily available or aggregated once a minute. Thus, Kraken performs traffic shifting decisions only after waiting 60 seconds from the previous shift, but keeps querying the health monitoring system continuously to quickly react to any changes in health. Timestamps are only used to provide a monotonically increasing ordering for the data from each server; clocks do not have to be synchronized.

Table 1 provides some examples of the metrics the health monitoring module considers when judging the

Service type	Metrics
Web servers	CPU utilization, latency, error rate, fraction of operational servers
Aggregator–leaf	CPU utilization, error rate, response quality
Proxygen [39]	CPU utilization, latency, connections, retransmit rate, ethernet utilization, memory capacity utilization
Memcache [31]	Latency, object lease count
TAO [10]	CPU utilization, write success rate, read latency
Batch processor	Queue length, exception rate
Logging [23]	Error rate
Search	CPU utilization
Service discovery	CPU utilization
Message delivery	CPU utilization

Table 1: Health metrics for various systems that are affected by web load.

health of various systems. These systems are all significantly impacted by user traffic and are bellwethers of non-linear behavior. Note that the metrics in Table 1 are not intended to be comprehensive. For example, CPU utilization is the only health metric for several services, but as these services add more features and establish different constraints on performance and quality over time, other metrics may also become important in gauging their health. Observe that we monitor multiple metrics from lower-level systems like TAO, Memcache, and the Proxygen load balancers as they have a large fan-out and are critical to the health of higher-level systems.

We store the health metric definitions in a distributed configuration management system [41]. Figure 4 shows the definition for web service health in terms of error rate. We use five levels of severity when reporting metric health. `level_ranges` define the range of values for each of these levels. Metric values are sampled over the amount of time specified by `time_window`. To ensure high confidence in our assessment, we also stipulate that at least `sample_fraction` of the data points reside above a level range. We use the highest level with at least `sample_fraction` data points above that level range. If we do not receive samples, the health metric is marked as unavailable and the service is considered unhealthy. We define each of the health metrics in Table 1 in this way.

4.3 Feedback control

At the start of a test, Kraken aggressively increases load and maintains the step size while the system is healthy. We have observed a trade-off between *the rate of load increase* and *system health*. For systems that employ caching, rapid shifts in load can lead to large cache miss rates and lower system health than slow increases in load. In practice, we find that initial load increase increments of around 15% strike a good balance between load test speed and system health.

```
web_error_rate = {
  entity = 'cluster1.web',
  key = 'error.rate',
  level_ranges = [
    {BOLD => (0.0, 0.00035)},
    {MODERATE => (0.00035, 0.0004)},
    {CAUTIOUS => (0.0004, 0.00045)},
    {NOMORE => (0.00045, 0.0005)},
    {BACKOF => (0.0005, 0.001)},
  ],
  time_window = '4m',
  sample_fraction = 0.4
}
```

Figure 4: The health metric definition for web error rate.

As health metrics approach their thresholds, Kraken dynamically reduces the magnitude of traffic shifts to prevent the system from becoming overloaded. For example, when any health metric is within 10% of its threshold value, Kraken will decrease load increments to 1%. This behavior has the benefit of also allowing us to collect more precise capacity information at high load.

4.4 Handling external conditions

Kraken's feedback loop makes it responsive to any event that causes a system being load tested to be unhealthy, whether or not it was anticipated prior to the test. We have found that the infrastructure is robust enough for this mechanism alone to mitigate the majority of unexpected failures. For extreme events, we have a small set of additional remediations as described below:

- **Request spike.** Facebook experiences request spikes due to natural phenomena, national holidays, and social events, for example, we have experienced a 100% load increase in our News Feed system in the course of 30 seconds during the Super Bowl. Our simple mitigation strategy is to configure Kraken to not run a load test during national holidays or planned social and sporting events. In the event of unexpected spikes, Kraken will abort any running load tests and also distribute load to all data centers by explicitly controlling the routing configuration file published to POPs.
- **Major faults in system operation.** Sometimes, a critical low-level system might get overloaded or might have a significant reliability bug, such as a kernel crash, that is triggered during a load test. If

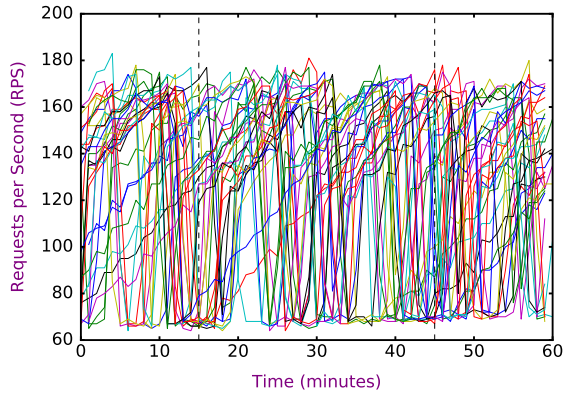


Figure 5: To measure web server capacity accurately, we continuously load test 32 web servers.

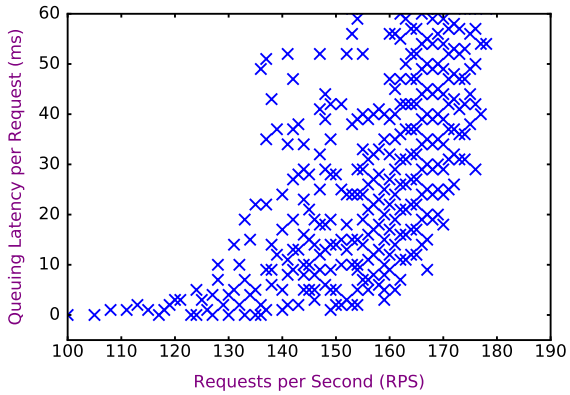


Figure 6: This plot displays the variance in the raw data from 32 load tested web servers.

this occurs, system health is bound to degrade significantly. Kraken polls Gorilla for system metrics every minute. If a fault is detected, Kraken immediately aborts the test and directs POPs to distribute load back to healthy clusters. We have recovered numerous times from issues of this sort, each time in about 2 minutes as it takes Kraken 60 seconds to detect the fault and then about 60 seconds to deliver an updated configuration to the POPs.

- **External faults such as a network partition and power loss.** As in the case above, Kraken will detect the problem and offload user traffic to other data centers in 2 minutes. If the cluster or region being tested does not recover, Kraken will decrease the load to 0, which will drain the target of all user traffic.

5 Evaluation

Kraken has been in use at Facebook for over three years and has run thousands of production load tests. As we mentioned in Section 4.2, we have augmented the set of metrics that Kraken monitors beyond the initial two, user perceivable latency and server error rate, to tens of other metrics that track the health and performance of our critical systems (cf. Table 1).

Further, we have developed a methodology around Kraken load tests that allows us to identify and resolve blockers limiting system utilization. By maintaining the pace and regularity of large scale load tests, we have incentivized teams to build instrumentation and tooling around collecting detailed data on system behavior under load. When a blocker is identified, Kraken's large scale load tests provide a structured mechanism for iteration so teams can experiment with different ideas for resolving the bottleneck and also continually improve their system performance and utilization.

Our evaluation answers the following questions:

1. Does Kraken allow us to validate capacity measurements at various scales?
2. Does Kraken provide a useful methodology for increasing utilization?

5.1 Does Kraken allow us to validate capacity measurements at various scales?

We evaluate this claim by first examining how Kraken allows us to measure the capacity of an individual server despite a continually changing workload, and then demonstrating how Kraken allows us to measure cluster and regional capacity.

5.1.1 Measuring an individual web server's capacity

Apache JMeter [4] and other load testing tools are widely employed to evaluate the capacity of an individual web server. While existing systems use a synthetic workload, Kraken directs live user traffic to measure system capacity for complex and continually changing workloads. In keeping with existing load testing systems, when measuring the capacity of an individual web server in isolation, Kraken assumes that all backend services and the network are infinite in size and cannot be saturated.

Each cluster of web servers might run a different generation of server hardware and experience a different composition of requests, so we need to run tests on web servers in each production cluster to obtain the baseline capacity for that cluster. As in a cluster load test, we use a preset error rate and latency metric as thresholds for when the system is operating at peak capacity. However, as we are testing a less complex system, we can be more exact. We use queuing latency on the server as our primary metric to gauge capacity—if the server begins to queue consistently, it is no longer able to keep up with the workload we are sending to it, and we have reached the individual server's max capacity.

We found that using a single server results in too much variance in the output capacity number. Through experi-

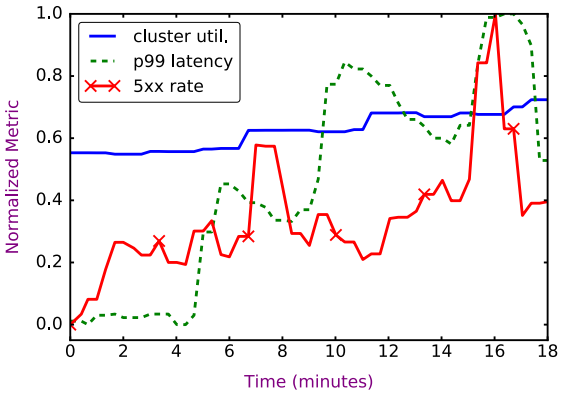
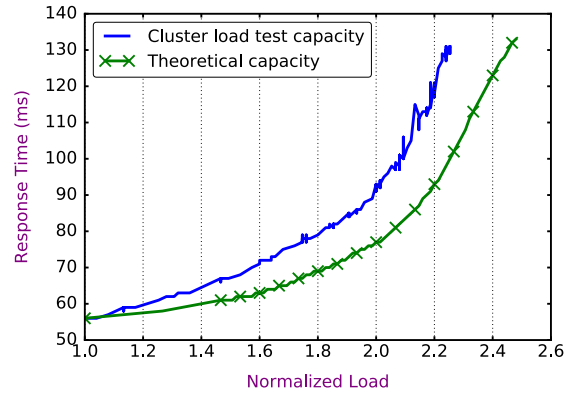
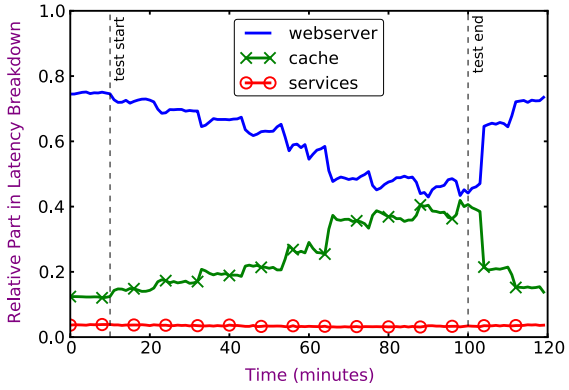


Figure 7: Kraken runs a cluster load test by directing increasing amounts of user traffic at a cluster. A load test affects both the cluster's CPU utilization and health metrics such as the HTTP 5xx error rate and response latency.



(a) Performance gap between cluster load test capacity and theoretical capacity.



(b) Latency breakdown in cluster load test.

Figure 8: (a) Demonstrates the performance gap between the cluster load test capacity and theoretical capacity. (b) Shows the latency breakdown that allows us to identify which of the caching systems or miscellaneous services contribute most to the performance gap. We observe an increase in time spent waiting for cache response as the cluster load increases, indicating that cache is the bottleneck.

mentation, we have found 32 servers to be an ideal number for our workload. Further increases in machine count do not significantly reduce variance. Figure 5 depicts a load test run on 32 independent web servers in a 60 minute interval. At the time this test was performed the servers were able to perform about 175 requests per second before the queuing latency threshold was reached.

Figure 6 shows the data points collected in a 30 minute window during the load test shown in Figure 5. Each data point plots the requests per second against CPU delay per request in milliseconds for a single web server, averaged over one minute. Our initial load test was run with a relaxed threshold for the sake of illustration. Once queuing begins, increases in allowed queuing result in quickly diminishing returns on throughput. In practice, we apply a more conservative limit of 20 ms queuing delay. To get our baseline server capacity, we simply take the average of the 32 servers. We use pick-2 load balancing [28] to ensure we evenly utilize servers, so we do not need to worry about the variance in hardware between servers. We can then derive the theoretical cluster capacity by multiplying the per-server capacity by the number of servers in a cluster.

5.1.2 Measuring a cluster's capacity

Figure 7 shows the execution of a cluster load test with Kraken. The line labeled `cluster util.` shows the current requests per second coming in to the cluster normalized by the cluster's theoretical max requests per second. Kraken initiates the cluster load test at minute 0 and the test concludes at minute 18. Every 5 minutes, Kraken inspects the health of the cluster and makes a decision for how to shift traffic. As described in Section 4.1, it takes Kraken about 2 minutes to execute a load shift, which is evident as cluster utilization changes around minute 7 for a decision made at minute 5. Notice that as the test proceeds, the cluster's health begins to degrade so Kraken decreases the magnitude of the traffic shifts until a peak utilization of 75% is hit. Kraken resets the load of the cluster in two traffic shifts over the course of 10 minutes (not shown in Figure 7).

Kraken closely monitors the health of the system when running this load test. The lines labeled `p99 latency` and `5xx rate` in Figure 7 correspond to the two initial health metrics we monitor: user perceivable latency and server error rate, respectively. Both of these metrics are normalized to the minimum and maximum values measured during the test. The three spikes in latency are a direct result of Kraken directing new users at this cluster—requests from the new users cause cache misses and require new data to be fetched. This test stopped due to the p99 latency, which is initially low but sustains above the threshold level for too long after the third traffic shift.

The final utilization of this cluster was 75% of the the-

oretical maximal. This is below our target utilization of 93% of the theoretical maximum, and we consider this load test unsuccessful. We next turn to how we identify and fix the issues that prevent a cluster from getting close to its theoretical maximal utilization.

Why did the cluster not hit its theoretical max utilization? Figure 8(a) depicts a different load test that hit a latency threshold. Here, the cluster performance (solid line) diverges from the theoretical web server performance (crossed line). To identify the utilization bottleneck, we drill into the data for other subsystems involved in serving a web request: the web server, cache, and other services. Figure 8(b) breaks down the web server response time between these three components. As load increases, the proportion of web server latency (the unmarked line) decreases while the proportion of cache latency (the crossed line) increases from around 15% at the point labeled `test start` to around 40% at the point labeled `test end` while service latency (the ○ line) remains unaffected, identifying cache as the bottleneck.

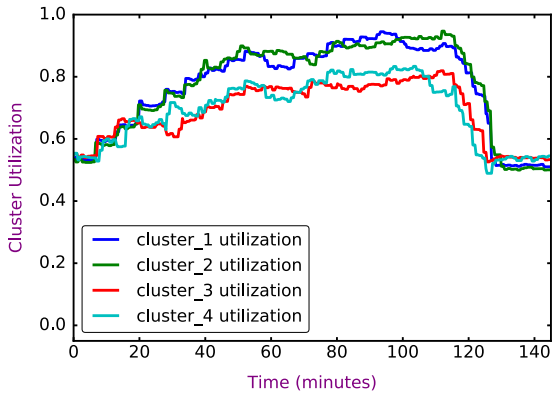
5.1.3 Measuring a region's capacity

Figure 9(a) shows that we can use Kraken to direct user traffic at multiple clusters simultaneously in a regional load test to stress systems that span clusters. Kraken reacts to variances in system health (for example, 5xx error rate, shown in Figure 9(b)) by decreasing user traffic to the tested clusters. Kraken maintains high load at about 90% utilization for about an hour—we intentionally hold the region under high load for an extended period of time to allow latent effects to surface. Kraken stops the test at minute 111 and quickly resets the load to normal levels in 15 minutes.

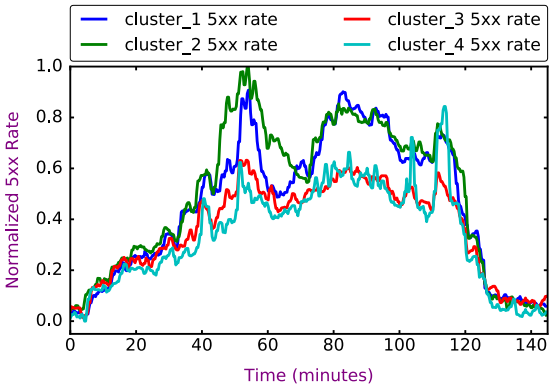
5.2 Does Kraken provide a useful methodology for increasing utilization?

Figure 10 depicts a load test from May 2015 where one of Kraken's regional load tests hit 75% utilization before encountering bottlenecks. Test outcomes of this form were the norm in 2014 and early 2015 as we were still developing the Kraken methodology for identifying and resolving bottlenecks encountered in cluster and regional load tests. Figure 9 depicts our current status where our regional load tests almost always hit their target utilization of 90% theoretical max resulting in a 20% system utilization improvement.

In this section, we describe how Kraken allowed us to surface many bottlenecks that were hidden until the systems were under load. We identified problems, experimented with remedies, and iterated on our solutions over successive tests. Further, this process of continually testing and fixing allowed us to develop a library of solutions and verify health without permitting regressions.



(a) Cluster utilization



(b) Cluster 5xx error rate

Figure 9: Kraken measures the capacity of a region by directing user traffic to all of the frontend clusters in the region simultaneously. (a) shows the effects on the data center clusters' utilization. (b) shows how the 5xx rate of the clusters changed during this time. Notice that when a cluster's HTTP fatals (5xx errors) increase, Kraken reduces the load on the cluster so it operates within a healthy range.

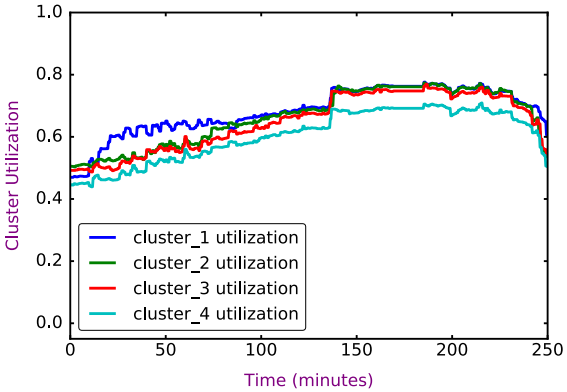


Figure 10: In this May 2015 test, Kraken pushes the frontend clusters in a region to an average of 75.7% utilization of their theoretical max before hitting pre-set thresholds. This load test was unsuccessful.

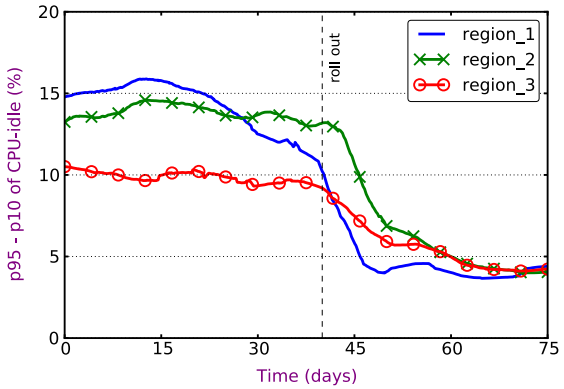


Figure 11: The spread in CPU utilization as measured by the difference between the 95th and 10th percentile CPU utilization in a cluster in different geographic regions. After using Kraken to identify a bottleneck in the shared cache resource, we deployed a technique to alleviate the bottleneck, resulting in a lower CPU utilization spread.

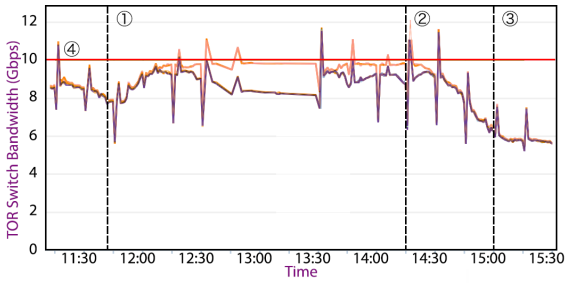


Figure 12: A spike in network traffic during a Kraken load test saturates two top-of-rack network switches. Alleviating this issue required relocating services running in the racks.

5.2.1 Hash weights for cache

The cache bottleneck depicted in Figure 8(b) was the first major issue that Kraken helped identify and resolve.

How did Kraken expose the cache bottleneck? Several cluster load tests were blocked by latency increases. Further inspection revealed that during the load test there was a disproportionate increase in the fraction of time spent retrieving data from TAO [10], a write-through cache that stores heavily accessed data. We engaged with the TAO engineers and after some instrumentation, learned that the latency increase was due to just a few cache machines that had significantly higher than average load during the test. It took several load tests to gather sufficient data to diagnose the bottleneck.

What was the cache bottleneck? TAO scales by logically partitioning its data into shards and randomly assigning these shards to individual cache servers. When constructing the response to a request, the web server might have to issue multiple rounds of data fetches, which might access hundreds of cache servers. TAO's hashing algorithm had been designed 4 years prior to the test. At that time, the ratio of shards to TAO servers was larger, and the shards were more uniform in request rate. Over time these assumptions changed, causing imbalance among the servers. Kraken's cluster load tests revealed that a small number of cache servers were driven out of CPU because they stored a significant fraction of all frequently-accessed data (e.g., popular content); this adversely affected all web servers that accessed that data. Instead of assigning shards to cache servers with a uniform hash, the solution was to assign each server a tunable *hash weight* based on the frequency of access, giving us finer control to even out the distribution of shards and balance load.

How did we validate the cache bottleneck fix? We leveraged Kraken to run multiple cluster load tests in different regions to validate the fix. Figure 11 depicts the outcome of hash weights—the x-axis shows time in days and the y-axis is the *difference* between the 95th and 10th percentile CPU utilization (i.e., the CPU utilization *spread* between highly-loaded cache servers and lowly-loaded cache servers) across the cache servers in clusters spread among three regions. Notice that after hash weights were rolled out, the spread in CPU utilization was reduced from an initial 10–15% to less than 5%. In tests following the fix, we were able to verify that this solution maintained its effectiveness at high load, and that we were able to apply more load to the clusters being tested. This solution addressed our short term capacity bottleneck and also significantly improved the reliability and scalability of TAO.

5.2.2 Network saturation

Most of Kraken's load tests in its first year of production were blocked by issues and bottlenecks in our higher-level software systems. As these bottlenecks were resolved, latent system issues at lower levels of the stack were revealed. During one test, which failed due to high latency, the latency effect was attributed to several disconnected services rathan than a single service. Further analysis revealed that the effect was localized to particular data center racks, which are often composed of machines from different services.

Figure 12 depicts the bandwidth utilization of two top-of-rack network switches during a Kraken load test. In this figure (and in the other case study figures in this section), we have labeled the phases of a load test: ① load test begins, ② load test starts decreasing load, and ③ load test ends. The nominal switch bandwidth limit of 10 Gbps, beyond which retransmissions or packet loss may occur, has been labeled ④ (specific to Figure 12).

Notice that the load test results in rack switches experiencing higher load due to the additional requests and

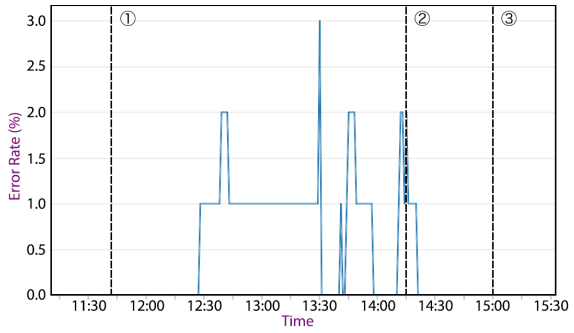


Figure 13: Saturated top-of-rack network switches result in services in the rack experiencing a 3% error rate during a Kraken load test.

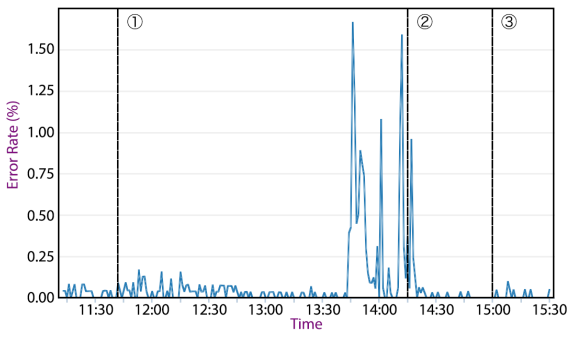


Figure 14: A Kraken load test triggers a 1.75% error rate in a critical classification service.

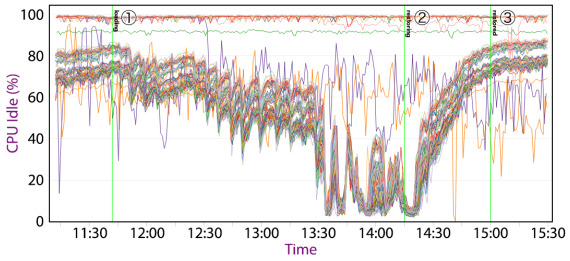


Figure 15: The multiple bands of CPU utilization clearly reveal a load imbalance in the machines running the classification service. This imbalance worsens under load and becomes a bottleneck that prevents us from fully utilizing the data center.

responses being sent to/from servers located in the racks, with peak bandwidth nearing 12 Gbps. These bandwidth spikes resulted in response errors, shown in Figure 13. Notice that around 13:30, nearly 3% of all requests experience an error.

An investigation revealed that these two racks housed a large number of machines belonging to a high-volume storage service that is used to retrieve posts when a user loads News Feed. During the load test, these machines received a higher volume of requests than normal due to the increased user traffic directed at this cluster and their responses saturated the uplink bandwidth of the top-of-rack network switch. We mitigated this problem by distributing this service's machines evenly across multiple racks in the data center.

Prior to surfacing this issue, we were aware of network saturation as a potential issue, but did not anticipate it as a bottleneck. This demonstrated the ability of Kraken to identify unexpected bottlenecks in the system. As a result we built a new network monitoring tool that identifies network utilization bottlenecks during Kraken's load tests and alerts service developers.

5.2.3 Poor load balancing

During a Kraken load test, a critical classification service in the data center under test experienced a 1.75% error rate in serving its requests (Figure 14). Note that this error rate is considered too high for the service.

Investigating uncovered the fact that the load was not evenly distributed among instances of the service as shown in Figure 15. Note that this load imbalance is clearly visible at time 11:30 (before the test commences) but it is only at time 13:30 that it is evident that imbalanced load imperils the service, as some fraction of machines are out of CPU. We can also observe an oscillation in CPU utilization between 13:30 and 14:15, due to the process hitting a failure condition because it is unable to properly handle the excess load. The over-utilization of a small handful of machines bottlenecks the entire region and results in a degraded performance.

In this case, pick-2 load balancing [28] was applied to even the load between servers. Failing under heavy load was also a suboptimal behavior we uncovered. For some services, techniques such as adaptive load shedding [17, 48] can be effective at reducing resource load while still servicing requests. In the case of this service, failures are difficult to tolerate, so a proportional-integral-derivative controller [37] was implemented to help continue to serve requests under high load.

5.2.4 Misconfiguration

Figure 16 shows an example of Kraken's health monitoring. One of the metrics that Kraken monitors is the error rate of a story ranking and delivery service. Recall that we have defined thresholds corresponding to how healthy a metric is: `MODERATE` (labeled A), `CAUTIOUS` (labeled B), `NOMORE` (labeled C), and `BACKOFF` (labeled D) (note that `BOLD` is not labeled as it corresponds to the range below `MODERATE`). Kraken uses the indicated thresholds for this health metric when making load change decisions.

We then worked with the ranking service's developers to instrument their system. Figure 17 shows that the service was experiencing multiple types of errors under load, with the most severe impact from "connection gating" where the number of concurrent threads requesting

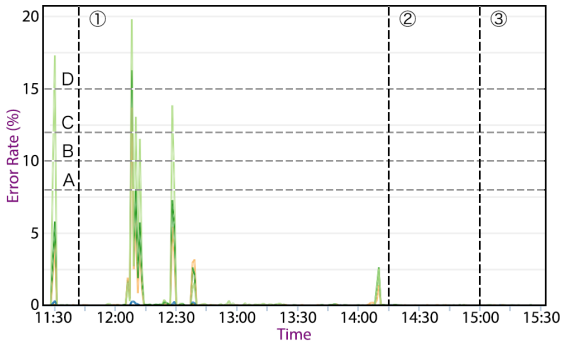


Figure 16: A load test triggers error spikes in a misconfigured service.

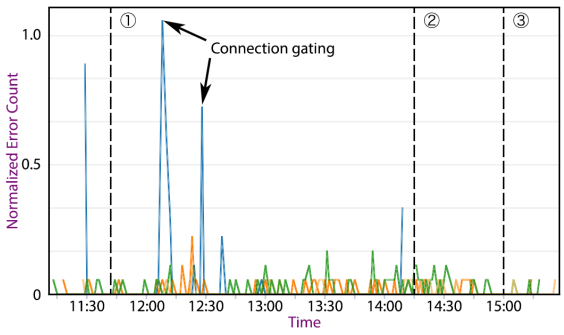


Figure 17: Failed responses in a misconfigured service.

data from an external database was limited to a fixed amount. While the intent of connection gating is to avoid exacerbating the effects of requesting data from a slow database instance, the actual gating value configured for this service was too low for the request rate it was servicing.

This bottleneck was resolved by increasing the number of concurrent threads allowed to request data from the database, and then running a follow-up Kraken load test to verify that no failed responses were returned by the ranking service, and that we could utilize the region more effectively as a result.

5.2.5 Insufficient capacity

Well-tuned services without any hardware or software bottlenecks may still lack sufficient capacity. This issue arises due to organic increases in service usage that are sometimes difficult for service developers to predict.

One surprising revelation was a Kraken load test that exposed a capacity issue in Gorilla [33], the service that Kraken relies on for monitoring and data collection. During a routine regional load test, Gorilla started experiencing increasing queuing, ultimately causing it to drop some requests as they began timing out. Figure 18 depicts the decrease in Enqueued requests and the increase in Dropped requests. Kraken aborted the load test as its monitoring component registered incomplete metrics.

We followed up by analyzing Gorilla. It turned out that the data center had recently acquired a new frontend cluster. As a result, when Kraken ran a regional load test, a much larger set of users were directed at the region and overloaded the existing Gorilla machines. The link between web traffic and Gorilla traffic is indirect, as Gorilla is accessed primarily by services rather than the web servers themselves, so the developers of Gorilla did not realize additional capacity would be required.

The mitigation was to allocate additional capacity to Gorilla to keep pace with increased demand. Note that these capacity allocations do not impact the service's efficiency requirements, which are benchmarked by regular load tests to ensure that performance regressions do not creep in and decrease throughput under high load.

5.3 Service-level load testing

Regional load tests are effective at finding service-level issues. However, service regressions and misconfigurations can be identified without requiring large scale tests, whereas problems like TAO load imbalance (see Section 5.2.1) are system effects that only manifest as bottlenecks at scale. Providing service owners a way to test their systems independently allows us to focus on identifying bottlenecks due to cross-system dependencies during regional tests.

Encouraged by Kraken's utility in characterizing system performance, we extended it to support load tests on individual services. When testing a service, Kraken dynamically updates the service's load balancing configuration to gradually concentrate traffic onto the designated set of target machines. Kraken monitors service-specific metrics that track resource usage, latency, and errors and measures both the service capacity and its behavior under load. Figure 3 illustrates how Kraken performs a service-level load test.

The goal of service-level load tests is to enable developers to quickly identify performance issues in their services without needing to wait until the next regional test. In addition to identifying regressions, resource exhaustion, and other utilization bottlenecks, service-specific load tests allow developers to evaluate different optimizations in production on a subset of their fleet.

For example, Figure 19 shows a load test performed on a set of News Feed's ranking aggregators. Load is measured in queries per second (QPS). As the test proceeds, QPS increases and the fraction of idle CPU time decreases. Notice that when the fraction of idle CPU time reaches 25%, the News Feed aggregator dynamically adjusts result ranking complexity. This ensures that the News Feed aggregator can continue to serve all incoming requests without exhausting CPU resources. This continues over the range of load labeled QoS control until,

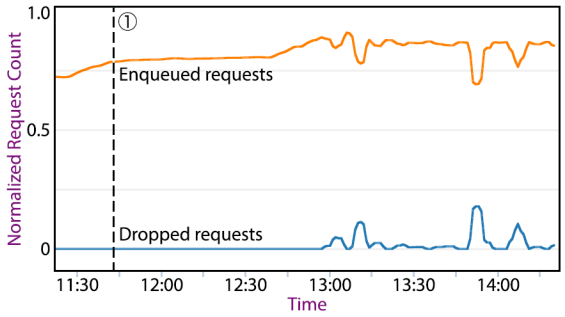


Figure 18: Gorilla drops requests due to insufficient capacity.

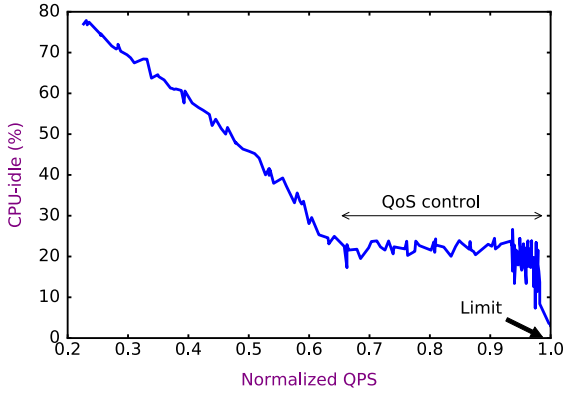


Figure 19: Kraken load testing News Feed. The range labeled QoS control shows the News Feed aggregators dynamically adjusting request quality to be able to serve more requests.

eventually, the fraction of CPU idle time drops below a pre-determined threshold and the load test ends.

6 Related Work

Large scale web services such as Amazon, Facebook, Google, Microsoft, and others serve a global audience that expect these services to be highly available. To ensure high availability in the face of disasters, these services often operate out of multiple geo-replicated data centers [3, 21]. Additionally, these services rely on modeling for capacity planning and resource allocation.

Capacity management based on theoretical system modeling is a widely studied topic. Some recent works in this field [20, 24, 46, 51], recognize the challenges of modeling multi-tier systems with changing workloads and propose various schemes for dynamic workload-driven adjustments to the theoretical allocations. There is an understanding that large scale distributed systems often times demonstrate *emergent behavior*, which cannot be predicted through analysis at any level simpler than that of the system as a whole [22, 29, 43]. These prior systems support Kraken's motivation for load testing to glean empirical data for dynamic capacity management.

Many prior systems such as Muse [11] and others [5, 32, 38, 47] automate resource allocation using instrumentation to provide prior history, offline and online analysis or simple feedback control to observe the system under testing, and iteratively apply an optimization policy until their objective function is satisfied. Doyle et. al. [18] build on the Muse system and propose a model-based approach to provisioning resources. Other systems utilize queuing [24, 30, 40] or energy [13, 19] models to predict how large systems will behave under load and how much power they will consume, to aid capacity management or maximize resource utilization.

Rather than derive analytical models for managing data centers, several recent systems propose experimentation on production systems [6, 50]. JustRunIt [50] proposes the use of sandboxed deployments that can execute shadow traffic from a real world deployment to answer various "what-if" questions. We make the same observation as JustRunIt but take their idea further as we wish to evaluate the performance and capacity limits of a non-virtualized production deployment.

Our strategy is to leverage load testing to evaluate and manage our systems. There are many open source and commercial load testing systems including Apache JMeter [4] and its variants such as BlazeMeter [9] as well as alternate frameworks such as Loader [25] that provide load testing as a service. We were unable to leverage these tools in our work, as they were limited in their scope. For instance, JMeter does not execute JavaScript while Loader only simulates connections to a web service. Instead, Kraken allows us to load test production systems with live user traffic.

Tang et. al. [42] leverage load testing to profile NUMA usage at Google but do not describe how their technique can be applied to identify higher-level bottlenecks or resource misallocation.

Kraken's analytics share ideas with the deeply developed field of performance analysis, which has always been crucial for detecting regressions and discovering bottlenecks in large scale distributed systems. While some previous performance analysis systems leverage fine-grained instrumentation such as Spectroscope [36], Magpie [7] and Pinpoint [12], others rely on passive analytics, investigating network flows [2], or logging information [14]. Kraken infers the dominant causal dependencies and bottlenecks by relying on the existing monitoring mechanisms provided by individual services. This allows for flexibility in the analysis of large heterogeneous systems. Various machine learning and statistics techniques have been developed for improving performance analysis of distributed systems [15, 16, 26]; Kraken's algorithms are much simpler so that its operations can be easily reasoned about by a human.

Kraken is not the first large scale test framework that

works with live user traffic. Netflix's Chaos Monkey [8, 45] induces failure in some component of an Amazon Web Services deployment and empirically confirms that the service is capable of tolerating faults and degrading gracefully under duress.

7 Experience

Since deploying Kraken to production, we have run over 50 regional load tests and thousands of cluster load tests. We list some lessons learned below:

- Generating capacity and utilization models with a continually changing workload is difficult. A competing approach, empirical testing, is a simpler alternative.
- Simplicity is key to Kraken's success. When load causes multiple systems to fail in unexpected ways, we need the stability of simple systems to debug complex issues.
- Identifying the right metrics that capture a complex system's performance, error rate, and latency is difficult. We have found it useful to identify several candidate metrics and then observe their behavior over tens to hundreds of tests to determine which ones provide the highest signal. However, once we identify stable metrics, their thresholds are easy to configure and almost never change once set.
- We find that specialized error handling mechanisms such as automatic failover and fallback mechanism can make systems harder to debug and lead to unexpected costs. These mechanisms have the ability to hide problems without resolving root causes, often leading small problems to snowball into bigger issues before detection and resolution. We find that such mitigations need to be well-instrumented to be effective in the long run, and prefer more direct methods such as graceful degradation.
- Bottleneck resolutions such as allocating capacity to needy services, changing system configuration or selecting different load balancing strategies have been critical for fixing production issues fast. We turn to heavy-weight resolutions like profiling, performance tuning, and system redesign only if the benefit justifies the engineering and capacity cost.
- While some systems prefer running on non-standard hardware or prefer non-uniform deployments in data centers, we have found that trading off some amount of efficiency and performance for simplicity makes systems much easier to operate and debug.

8 Conclusion

Large scale web services are difficult to accurately model because they are composed of hundreds of rapidly evolving software systems, are distributed across geo-replicated data centers, and have constantly changing workloads. We propose Kraken, a system that leverages live user traffic to empirically load test every level of the infrastructure stack to measure capacity. Further, we describe a methodology for identifying bottlenecks and iterating over solutions with successive Kraken load tests to continually improve infrastructure utilization. Kraken has been in production for the past three years. In that time, Kraken has run thousands of load tests and allowed us to increase Facebook's capacity to serve users by over 20% using the same hardware.

9 Acknowledgments

We thank the anonymous reviewers and our shepherd, Rebecca Isaacs, for comments that greatly improved this paper. We also thank Théo Dubourg, Constantin Dumitrascu, David Felty, Calvin Fu, Yun Jin, Silvio Soares Ribeiro Junior, Daniel Peek, and the numerous developers at Facebook who have given us feedback on the tooling, monitoring, and methodology of Kraken and helped us improve our infrastructure utilization.

References

[1] ABRAHAM, L., ALLEN, J., BARYKIN, O., BORKAR, V., CHOPRA, B., GEREA, C., MERL, D., METZLER, J., REISS, D., SUBRAMANIAN, S., WIENER, J., AND ZED, O. Scuba: Diving into data at Facebook. In *Proceedings of the 39th International Conference on Very Large Data Bases* (August 2013), VLDB '13.

[2] AGUILERA, M. K., MOGUL, J. C., WIENER, J. L., REYNOLDS, P., AND MUTHITACHAROEN, A. Performance debugging for distributed systems of black boxes. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles* (2003), SOSP '03.

[3] AMAZON. Regions and availability zones. `http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/using-regions-availability-zones.html`.

[4] Apache JMeter. `http://jmeter.apache.org/`.

[5] ARON, M., DRUSCHEL, P., AND ZWAENEPOEL, W. Cluster reserves: A mechanism for resource management in cluster-based network servers. In *Proceedings of the 2000 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems* (2000), SIGMETRICS '00.

[6] BABU, S., BORISOV, N., DUAN, S., HERODOTOU, H., AND THUMMALA, V. Automated experiment-driven management of (database) systems. In *Proceedings of the 12th*

Conference on Hot Topics in Operating Systems (2009), HotOS '09.

[7] BARHAM, P., DONNELLY, A., ISAACS, R., AND MORTIER, R. Using Magpie for request extraction and workload modelling. In *Proceedings of the 6th USENIX Symposium on Operating Systems Design and Implementation* (2004), OSDI '04.

[8] BASIRI, A., BEHNAM, N., DE ROOIJ, R., HOCHSTEIN, L., KOSEWSKI, L., REYNOLDS, J., AND ROSENTHAL, C. Chaos engineering. *IEEE Software 33*, 3 (May 2016), 35–41.

[9] BlazeMeter. `http://blazemeter.com/`.

[10] BRONSON, N., AMSDEN, Z., CABRERA, G., CHAKKA, P., DIMOV, P., DING, H., FERRIS, J., GIARDULLO, A., KULKARNI, S., LI, H. C., MARCHUKOV, M., PETROV, D., PUZAR, L., SONG, Y. J., AND VENKATARAMANI, V. TAO: Facebook's distributed data store for the social graph. In *Proceedings of the 2013 USENIX Annual Technical Conference* (2013), USENIX ATC '13.

[11] CHASE, J. S., ANDERSON, D. C., THAKAR, P. N., VAHDAT, A. M., AND DOYLE, R. P. Managing energy and server resources in hosting centers. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles* (2001), SOSP '01.

[12] CHEN, M. Y., KICIMAN, E., FRATKIN, E., FOX, A., AND BREWER, E. Pinpoint: Problem determination in large, dynamic internet services. In *Proceedings of the 2002 International Conference on Dependable Systems and Networks* (2002), DSN '02.

[13] CHEN, Y., DAS, A., QIN, W., SIVASUBRAMANIAM, A., WANG, Q., AND GAUTAM, N. Managing server energy and operational costs in hosting centers. In *Proceedings of the 2005 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems* (2005), SIGMETRICS '05.

[14] CHOW, M., MEISNER, D., FLINN, J., PEEK, D., AND WENISCH, T. F. The mystery machine: End-to-end performance analysis of large-scale internet services. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation* (2014), OSDI '14.

[15] COHEN, I., GOLDSZMIDT, M., KELLY, T., SYMONS, J., AND CHASE, J. S. Correlating instrumentation data to system states: A building block for automated diagnosis and control. In *Proceedings of the 6th USENIX Symposium on Operating Systems Design and Implementation* (2004), OSDI '04.

[16] COHEN, I., ZHANG, S., GOLDSZMIDT, M., SYMONS, J., KELLY, T., AND FOX, A. Capturing, indexing, clustering, and retrieving system history. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles* (2005), SOSP '05.

[17] DAS, T., ZHONG, Y., STOICA, I., AND SHENKER, S. Adaptive stream processing using dynamic batch sizing. In *Proceedings of the ACM Symposium on Cloud Computing* (2014), SOCC '14.

[18] DOYLE, R. P., CHASE, J. S., ASAD, O. M., JIN, W., AND VAHDAT, A. M. Model-based resource provisioning in a web service utility. In *Proceedings of the 4th USENIX Symposium on Internet Technologies and Systems* (2003), USITS '03.

[19] FAN, X., WEBER, W.-D., AND BARROSO, L. A. Power provisioning for a warehouse-sized computer. In *Proceedings of the 34th Annual International Symposium on Computer Architecture* (2007), ISCA '07.

[20] GMACH, D., ROLIA, J., CHERKASOVA, L., AND KEMPER, A. Resource pool management: Reactive versus proactive or let's be friends. *Computer Networks: The International Journal of Computer and Telecommunications Networking 53*, 17 (December 2009), 2905–2922.

[21] GOOGLE. Google compute engine: Regions and zones. `https://cloud.google.com/compute/docs/zones`.

[22] GRIBBLE, S. D. Robustness in complex systems. In *Proceedings of the 8th Workshop on Hot Topics in Operating Systems* (2001), HOTOS '01.

[23] JAIN, N., BORTHAKUR, D., MURTHY, R., SHAO, Z., ANTONY, S., THUSOO, A., SARMA, J., AND LIU, H. Data Warehousing and Analytics Infrastructure at Facebook. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data* (2010), SIGMOD '10.

[24] LIU, X., HEO, J., AND SHA, L. Modeling 3-tiered web applications. In *Proceedings of the 13th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems* (2005), MASCOTS '16.

[25] Loader: simple cloud-based load testing. `http://loader.io`.

[26] MALIK, H. A methodology to support load test analysis. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering* (2010), ICSE '10.

[27] MICHELSEN, M. Continuous deployment at Quora. `http://engineering.quora.com/Continuous-Deployment-at-Quora`.

[28] MITZENMACHER, M. The power of two choices in randomized load balancing. *IEEE Transactions on Parallel and Distributed Systems 12* (October 2001), 1094–1104.

[29] MOGUL, J. C. Emergent (mis)behavior vs. complex software systems. In *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems* (2006), EuroSys '06.

[30] NAIR, J., WIERMAN, A., AND ZWART, B. Provisioning of large-scale systems: The interplay between network effects and strategic behavior in the user base. *Management Science 62*, 6 (November 2016), 1830–1841.

[31] NISHTALA, R., FUGAL, H., GRIMM, S., KWIATKOWSKI, M., LEE, H., LI, H., MCELROY, R., PALECZNY, M., PEEK, D., SAAB, P., STAFFORD, D., TUNG, T., AND VENKATARAMANI, V. Scaling Memcache at Facebook. In *Proceedings of the 10th USENIX conference on Networked Systems Design and Implementation* (2013), NSDI '13.

[32] PADALA, P., SHIN, K. G., ZHU, X., UYSAL, M., WANG, Z., SINGHAL, S., MERCHANT, A., AND SALEM, K. Adaptive control of virtualized resources in utility computing environments. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems* (2007), EuroSys '07.

[33] PELKONEN, T., FRANKLIN, S., TELLER, J., CAVALLARO, P., HUANG, Q., MEZA, J., AND VEERARAGHAVAN, K. Gorilla: A fast, scalable, in-memory time series database. In *Proceedings of the 41st International Conference on Very Large Data Bases* (2015), VLDB '15.

[34] RIVIORE, S., SHAH, M. A., RANGANATHAN, P., AND KOZYRAKIS, C. Joulesort: A balanced energy-efficiency benchmark. In *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data* (2007), SIGMOD '07.

[35] ROSSI, C. Ship early and ship twice as often, 2012. `https://www.facebook.com/notes/facebook-engineering/ship-early-and-ship-twice-as-often/10150985860363920`.

[36] SAMBASIVAN, R. R., ZHENG, A. X., DE ROSA, M., KREVAT, E., WHITMAN, S., STROUCKEN, M., WANG, W., XU, L., AND GANGER, G. R. Diagnosing performance changes by comparing request flows. In *Proceedings of the 8th USENIX Networked Systems Design and Implementation* (2011), NSDI '11.

[37] SELLERS, D. An overview of proportional plus integral plus derivative control and suggestions for its successful application and implementation. In *Proceedings of the 1st International Conference for Enhanced Building Operations* (2001), ICEBO '01.

[38] SHEN, K., TANG, H., YANG, T., AND CHU, L. Integrated resource management for cluster-based internet services. In *Proceedings of the 5th USENIX Symposium on Operating Systems Design and implementation* (2002), OSDI '02.

[39] SOMMERMANN, D., AND FRINDELL, A. Introducing Proxygen, Facebook's C++ HTTP framework, 2014. `https://code.facebook.com/posts/1503205539947302/introducing-proxygen-facebook-s-c-http-framework/`.

[40] STEWART, C., AND SHEN, K. Performance modeling and system management for multi-component online services. In *Proceedings of the 2nd USENIX Networked Systems Design and Implementation* (2005), NSDI '05.

[41] TANG, C., KOOBURAT, T., VENKATACHALAM, P., CHANDER, A., WEN, Z., NARAYANAN, A., DOWELL, P., AND KARL, R. Holistic configuration management at Facebook. In *Proceedings of the 25th Symposium on Operating Systems Principles* (October 2015), SOSP '15.

[42] TANG, L., MARS, J., ZHANG, X., HAGMANN, R., HUNDT, R., AND TUNE, E. Optimizing Google's warehouse scale computers: The NUMA experience. In *Proceedings of the 19th International Symposium on High-Performance Computer Architecture* (2013), HPCA '13.

[43] THERESKA, E., AND GANGER, G. R. IRONModel: Robust performance models in the wild. In *Proceedings of the 2008 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems* (2008), SIGMETRICS '08.

[44] TPC-C. `http://www.tpc.org/tpcc/`.

[45] TSEITLIN, A. The antifragile organization. *Communications of the ACM 56*, 8 (August 2013), 40–44.

[46] URGAONKAR, B., SHENOY, P., CHANDRA, A., GOYAL, P., AND WOOD, T. Agile dynamic provisioning of multi-tier internet applications. *ACM Transactions on Autonomous and Adaptive Systems (TAAS) 3*, 1 (March 2008), 1–39.

[47] URGAONKAR, B., SHENOY, P., AND ROSCOE, T. Resource overbooking and application profiling in shared hosting platforms. In *Proceedings of the 5th USENIX Symposium on Operating Systems Design and implementation* (2002), OSDI '02.

[48] WELSH, M., AND CULLER, D. Adaptive overload control for busy internet servers. In *USENIX Symposium on Internet Technologies and Systems* (2003), USITS '03.

[49] Yahoo! cloud serving benchmark (YCSB). `https://github.com/brianfrankcooper/YCSB/wiki`.

[50] ZHENG, W., BIANCHINI, R., JANAKIRAMAN, G. J., SANTOS, J. R., AND TURNER, Y. JustRunIt: Experiment-based management of virtualized data centers. In *Proceedings of the 2009 USENIX Annual Technical Conference* (2009), USENIX '09.

[51] ZHU, X., YOUNG, D., WATSON, B. J., WANG, Z., ROLIA, J., SINGHAL, S., MCKEE, B., HYSER, C., GMACH, D., GARDNER, R., CHRISTIAN, T., AND CHERKASOVA, L. 1000 islands: Integrated capacity and workload management for the next generation data center. In *Proceedings of the 2008 International Conference on Autonomic Computing* (2008), ICAC '08.