Maelstrom: Mitigating Datacenter-level Disasters by Draining Interdependent Traffic Safely and Efficiently

Kaushik Veeraraghavan Justin Meza Scott Michelson Sankaralingam Panneerselvam
Alex Gyori David Chou Sonia Margulis Daniel Obenshain Ashish Shah
Yee Jiun Song Tianyin Xu*

{kaushikv, jjm, sdmich, sankarp, gyori, davidchou, frumious, danielo, ahish9, yj, tianyin}@fb.com

Facebook Inc. *UIUC

Abstract

We present Maelstrom, a new system for mitigating and recovering from datacenter-level disasters. Maelstrom provides a traffic management framework with modular, reusable primitives that can be composed to *safely* and *efficiently* drain the traffic of interdependent services from one or more failing datacenters to the healthy ones.

Maelstrom ensures safety by encoding inter-service dependencies and resource constraints. Maelstrom uses health monitoring to implement feedback control so that all specified constraints are satisfied by the traffic drains and recovery procedures executed during disaster mitigation. Maelstrom exploits parallelism to drain and restore independent traffic sources efficiently.

We verify the correctness of Maelstrom's disaster mitigation and recovery procedures by running large-scale tests that drain production traffic from entire datacenters and then retore the traffic back to the datacenters. These tests (termed drain tests) help us gain a deep understanding of our complex systems, and provide a venue for continually improving the reliability of our infrastructure.

Maelstrom has been in production at Facebook for more than four years, and has been successfully used to mitigate and recover from 100+ datacenter outages.

1 Introduction

1.1 Motivation

Modern Internet services are composed of hundreds of interdependent systems spanning dozens of geo-distributed datacenters [7, 23]. At this scale, seemingly rare natural disasters, such as hurricanes blowing down power lines and flooding [32, 46], occur regularly. Further, man-made incidents such as network fibercuts, software bugs and misconfiguration can also affect entire datacenters [25, 37, 41]. In our experience, outages that affect one or more datacenters cannot be addressed by traditional fault-tolerance mechanisms designed for individual machine failures and network faults as co-located redundant capacity is also likely impaired.

In a disaster scenario, *mitigation* is almost always the first response to reduce user-visible impact, before root causes are discerned and systems are recovered. In our experience, outages affecting physical infrastructure take a long time to repair as they often involve work by on-site maintenance personnel. Software failures can be hard to debug and fix, and thus it is hard to guarantee a resoluton time [9, 25, 26, 60].

The basic idea of disaster mitigation is to quickly *drain traffic*—redirect requests originally sent to failing datacenters and reroute them to healthy datacenters. Our assumption when draining traffic is that a disaster affects only a fraction of our overall infrastructure—this assumption is reasonable because most natural disasters (e.g., hurricanes and earthquakes) are locality based. For software failures caused by bugs and misconfiguration, we adopt a locality-based staged rollout strategy, partially driven by our ability to use Maelstrom to quickly drain traffic from an affected datacenter.

We find that most failures are not instantaneous and thus can be detected and mitigated in time. For instance, we had about one week of notice before Hurricane Florence made landfall in North Carolina on September 15, 2018. This advance notice allowed us to plan and execute mitigations were Facebook's Forest City datacenter to be affected. Further, it is far more likely that a failure affects parts of a datacenter or certain infrastructure components (e.g., several network backbone cables) than resulting in total loss of a physical datacenter. In all these cases, developing the mechanism to quickly redirect user traffic as well as inter-service traffic, which we term "draining traffic", is key to disaster readiness.

The conceptually simple idea of draining traffic turns out to be rather challenging in practice. In our experience, disasters often trigger failures that affect multiple interdependent systems simultaneously. Ideally, every system should be implemented with a multi-homed design [28], where any traffic can be sent to and served by any datacenter. However, we observe that most of today's Internet services are composed of a number

of heterogeneous systems including singly-homed and failover-based systems with complex, subtle dependencies, and distinct traffic characteristics [12, 32, 46, 53].

The most challenging aspect of mitigation is to ensure that dependencies among systems are not violated. For instance, in a distributed caching system, if we drain cache invalidation traffic before redirecting read traffic from clients, we risk serving stale data. Or, in a web service, if we drain intra-datacenter traffic between web servers and backend systems before redirecting user requests, we risk increasing response latency due to cross-datacenter requests. Hence, we need a disaster mitigation systems that can track dependencies among services, and also sequence operations in the right order.

Different systems may require customized mitigation procedures due to their distinct traffic characteristics, e.g., draining stateless web traffic requires a different procedure from draining stateful database traffic. Without unified, holistic tooling, each system might end up maintaining their own, incompatible disaster mitigation scripts that cannot be composed or tuned for scenarios with varying levels of urgency. As shown in §5.3, draining systems sequentially can significantly slow down the mitigation process, and prolong the impact of a disaster.

Disaster mitigation and recovery strategies also need to monitor shared resources, such as network bandwidth and datacenter capacity. Naïvely redirecting all traffic from one datacenter to another could overwhelm the network and trigger cascading failures.

1.2 Maelstrom for Disaster Mitigation & Recovery

We present Maelstrom, a system used for mitigating and recovering from datacenter-level disasters[1] at Facebook. Maelstrom *safely* and *efficiently* drains traffic of interdependent systems from one or more failing datacenters to the healthy ones to maintain availability during a disaster. Once the disaster is resolved, Maelstrom restores the datacenter to a healthy state.

Maelstrom offers a generic traffic management framework with modularized, reusable primitives (e.g., shifting traffic, reallocating containers, changing configurations, and moving data shards). Disaster mitigation and recovery procedures are implemented by customizing and composing these primitives. Inter-system dependencies specify the order of executing the primitives, and resource constraints control the pace of executing individual primitives. This design is driven by two observations: 1) while each system has its own procedures for mitigation and recovery, these procedures share a common set of primitives, and 2) different procedures share similar high-level flows—draining traffic while maintaining system health and SLAs. Therefore, it is feasible to build a generic system to satisfy the needs of a wide variety of systems with heterogeneous traffic characteristics.

To ensure safety, Maelstrom coordinates large-scale traffic shifts by respecting inter-system dependencies and resource constraints. Dependencies are rigorously encoded and maintained. We employ critical path analysis to identify bottlenecks and decrease time to mitigate disasters. Maelstrom implements a closed feedback loop to drain traffic as fast as possible without compromising system health. In order to mitigate disasters efficiently, Maelstrom exploits parallelism to drain independent traffic sources, which significantly speeds up execution of the mitigation and recovery procedures.

We find that Maelstrom makes disaster mitigation and recovery significantly easier to understand and reason about, in comparison to monolithic, opaque scripts. Maelstrom also incorporates extensive UI support to display the mitigation and recovery steps, and their runtime execution states, to assist human proctoring and intervention in disaster scenarios (cf. §3).

1.3 Drain Tests for Verifying Disaster Readiness

We employ Maelstrom to run different types of large-scale tests that simulate real-world disasters. We find that annual, multi-day failure drills such as DiRT [32] and GameDay [46] are useful to verify that entire datacenters can be shutdown and restarted. However, besides these annual tests, we desire a regimen of continuous tests that can be executed at daily and weekly frequencies to ensure that our mitigation and recovery keep up with rapidly-changing systems and infrastructure.

We present our practical approach, termed *drain tests*, to address the challenge. A drain test is a fully automated test that uses Maelstrom to drain user-facing and internal traffic from our datacenters in the same way as if these datacenters are failing. Running drain tests on a regular basis enables our systems to always be prepared for various disaster scenarios by maintaining and exercising the corresponding mitigation procedures. Drain tests also force us to gain a deep understanding of our complex, dynamic systems and infrastructure, and help us plan capacity for projected demand, audit utilization of shared resources, and discover dependencies (cf. §3).

Drain tests operate on live production traffic and thus could be disruptive to user-facing services, if not done carefully. It has taken us multiple years to reach our current state of safety and efficiency. Our original tests only targeted one stateless system: our web servers. The first set of drain tests were painful—they took more than 10 hours to run, experienced numerous interruptions as we uncovered dependencies or triggered failures that resulted in service-level issues. As we built Maelstrom

[1] Maelstrom does not target machine-level failures (which should be tolerated by any large-scale system), or software bugs and misconfiguration that can be immediately reverted.

and began using it to track and encode dependencies, drain tests gradually became smooth and efficient. After a year, we extended drain tests to two more services: a photo sharing service and a real-time messaging service. Currently, Maelstrom drains hundreds of services in a fully automated manner, with new systems being onboarded regularly. We can drain all user-facing traffic, across multiple product families, from any datacenter in less than 40 minutes.

1.4 Contributions

Maelstrom has been in operation at Facebook in the past 4 years, and has been used to run hundreds of drain tests and has helped mitigate 100+ disasters. The paper makes the following contributions:

- Maelstrom is the first generic framework that can drain heterogeneous traffic of interdependent systems safely and efficiently to mitigate datacenter-level disasters.
- We introduce drain tests as a novel reliability engineering practice for continuously testing and verifying the disaster-readiness of Internet services.
- We share the lessons and experience of running regular drain tests, as well as mitigating real disasters at a large-scale Internet service.

2 Background

This section provides an overview of Facebook's infrastructure and traffic management primitives which are similar to other major Internet services [10, 23, 36, 55].

2.1 Infrastructure Overview

As Figure 1 shows, user requests to `www.Facebook.com` are sent to an ISP which maps the URL to an IP address using a DNS resolver. This IP address points to one of the tens of edge locations (also known as Point-of-Presence or PoPs) distributed worldwide. A PoP consists of a small number of servers, typically co-located with a peering network [47, 59]. A PoP server terminates the user's SSL session and then forwards the request on to an L4 load balancer (Edge LB) which forwards the request on to a particular datacenter. A user request can be served from any of our datacenters.

We group machines in a datacenter into logical *clusters* such as frontend clusters composed of web servers, backend clusters of storage systems, and generic "service" clusters. We define a *service* as the set of subsystems that support a particular product.

Within a datacenter, an L7 web load balancer (Web LB) forwards the user request to a web server in a frontend cluster. This web server may communicate with tens or hundreds of services, and these services typically need to further communicate with other services and backends, to gather the data needed to generate a response.

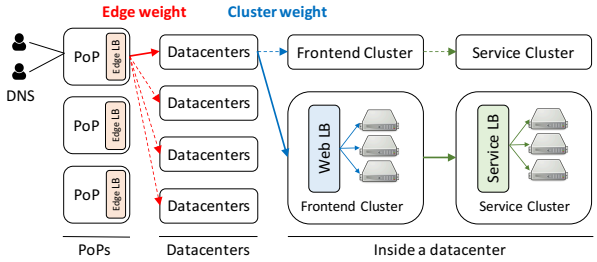

Figure 1: An overview of Facebook's infrastructure. The configurable edge and cluster weights determine how user requests are routed from PoPs to particular datacenters, and then on to particular clusters.

Traffic	Affinity	State	Strategy
Stateless	—	—	reroute
Sticky	✓	—	reroute → tear down
Replication	—	✓	customized
Stateful	✓	✓	master promotion

Table 1: Traffic type, property, and mitigation strategy (cf. §2.3).

We employ a set of service load balancers (Service LBs) to distribute requests amongst service and backend clusters. The web server handling the user request is also responsible for returning the response to the PoP which then forwards it on to the end user.

2.2 Traffic Management Primitives

The PoP server parses each request URI and maps it to a service. Our traffic management system assigns each service a virtual IP (VIP). Traffic for each VIP is controlled by two configurable values: *edge weight* and *cluster weight*. Edge weights specify the fraction of requests that the PoP should forward to each of the datacenters. Cluster weights specify the fraction of requests that each cluster is capable of handling.

Since PoPs and frontend clusters are stateless, a user request can be sent to any PoP and forwarded to any frontend web server. This property allows us to programmatically reconfigure edge and cluster weights to reroute traffic in disaster scenarios. For instance, if a network fiber-cut disconnects a datacenter from the rest, we push out a configuration change to all PoPs that sets the edge weight for the disconnected datacenter to 0; this results in the traffic originally sent to the failing datacenter being routed to the other datacenters.

Internal service traffic (e.g., RPC traffic) within and across datacenters are controlled by L7 service load balancers based on configurable knobs in a similar vein.

2.3 Traffic Types

Table 1 categorizes the traffic types of different systems based on affinity and state properties, as well as the common strategies for draining them during disasters.

- *Stateless.* The vast majority of web traffic is stateless, consisting of users' web requests directed from PoPs to one or more datacenters. Stateless traffic can be drained by rerouting it away from a failing datacenter, or from particular sets of clusters, racks, or machines.
- *Sticky.* Interactive services (e.g., messaging) improve user experience by pinning requests to particular machines that maintain the state for a user in a session. Sticky traffic can be drained by rerouting incoming session requests and tearing down the established sessions to force them reconnect to other machines.
- *Replication.* In a disaster, we may need to alter or even stop replication traffic from egressing or ingressing the failing datacenter for distributed storage systems. The replicas can be re-created in other datacenters to serve reads. This requires configuration changes or other heavyweight changes that influence resource sharing, such as intra- and inter-datacenter networks.
- *Stateful.* For master-slave replication based systems, the mitigation for a master failure is to promote a secondary to be the new master. This may require copying states from the failing datacenter to the new. The state copy requires careful control based on the network capacity to transfer data out to healthy machines.

3 Maelstrom Overview

Maelstrom is a disaster mitigation and recovery system. During a datacenter-level disaster, operators[2] use Maelstrom to execute a *runbook* that specifies the concrete procedure for mitigating the particular disaster scenario by draining traffic out of the datacenter; after the root causes are resolved, a corresponding recovery runbook is used to restore traffic back.

Maelstrom provides a generic traffic management framework. A runbook can be created via Maelstrom's UI by composing a set of *tasks*. A task is a specific operation, such as shifting a portion of traffic, migrating data shards, restarting container jobs, and changing configurations. Tasks can have *dependencies* that determine the order of execution—a task should not be started before its dependent tasks are completed. Figure 2 shows an example of a runbook and its corresponding tasks. We elaborate the runbook-based framework in §4.2.

Every service maintains its own *service-specific runbooks* for disaster mitigation. Taking our interactive messaging service as an example, the runbook for draining the service's sticky traffic (upon software failures in a datacenter) includes two tasks in order: 1) redirecting new session requests to the other datacenters, and 2) terminating established sessions in the failing datacenter to force them reconnect. A recovery runbook can be used to restore messaging traffic back to the datacenter.

[2]In this paper, we use "operators" as a general term for anyone helping with operations, including Software Engineers, Site Reliability Engineers, Production Engineers, and System Administrators.

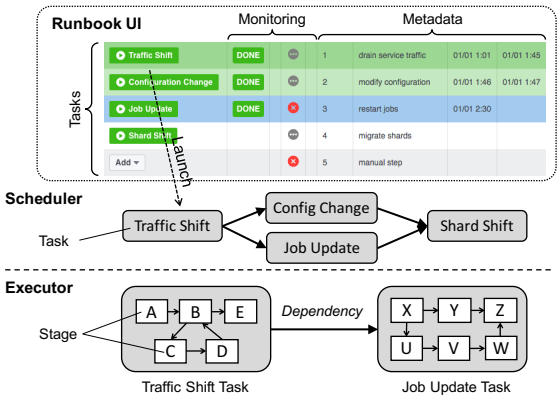

Figure 2: Maelstrom executes *runbooks,* each specifying the procedure for mitigating a particular disaster scenario. A runbook is composed of interdependent *tasks* (e.g., traffic shift and job updates). These tasks are scheduled by Maelstrom's *scheduler* based on their dependencies, and are executed in multiple stages by Maelstrom's *executor*. Maelstrom monitors and displays the runtime status tasks in the Runbook UI.

If an entire datacenter is down (e.g., due to network fibercuts that disconnect it from our infrastructure), a *datacenter evacuation runbook* will be used to drain traffic of all the services in the datacenter. A datacenter evacuation runbook is composed of service-specific runbooks, where each runbook drains or restores the traffic for a particular service deployed in a datacenter. These service-specific runbooks are aggregated through external dependencies that link tasks in different runbooks.

Runbooks are executed by Maelstrom's runtime engine consisting of two main components: 1) the *scheduler* that schedules tasks to execute based on the policy specified in the runbook (including dependencies and conditions), and 2) the *executor* that is responsible for executing each individual task. A task can be executed in multiple stages based on its implementation (cf. §4.3).

As shown in Figure 2, Maelstrom is equipped with a UI that monitors and visualizes the runtime information of a runbook, including the state of every task, their dependencies, and the associated health metrics. We keep improving the UI with an operator-centric methodology. Each disaster provides a learning opportunity for us to interact with the operators and to improve usability. Our UI design focuses on helping operators understand the mitigation and recovery status, and on efficiently controlling the runbook execution.

At Facebook, we use Maelstrom to run different types of tests with different frequencies. Besides weekly drain tests, we also run *storm tests* at a quarterly cadence. The primary difference between a *drain test* and a *storm test* is that a drain test is focused on draining user traffic out of a datacenter as fast as possible without user perceivable impact. In contrast, a storm test extends beyond user

traffic to drain all RPC traffic amongst services, stops data replication, and applies network ACLs to isolate the tested data center. Thus, a storm test is a more rigorous endeavor that verifies that all of Facebook's products and systems can function correctly despite the total loss of a datacenter. From our understanding, storm tests are akin to Google's DiRT [32] and Amazon's GameDay [46] exercises. In this paper, we focus on drain tests as a new type of large-scale, fully-automated test for production services, which can be run on a daily or weekly basis.

3.1 Drain Tests

Maelstrom requires runbooks to always keep updated with our rapidly-evolving software systems and physical infrastructure. However, maintaining up-to-date information (e.g., service dependencies) is challenging due to the complexity and dynamics of systems at scale, akin to the observations of other cloud-scale systems [5, 32, 37].

Drain tests are our practical solution to continuously verify and build trust in the runbooks. A drain test is a *fully automated* test that uses Maelstrom to drain user-facing and internal service traffic from our datacenters in the same way as if these datacenters are failing. Internal services include various asynchronous jobs, data processing pipelines, machine learning systems, software development tools, and many other services that are key components of our infrastructure.

We run multiple drain tests per week to simulate various types of disaster scenarios (e.g., those listed in [22]) on a least-recently-tested datacenter. Tests are scheduled at different times in a day to cover various traffic patterns (e.g., peak and off-peak time). We also vary the duration of each test to understand how the rest of our infrastructure serve user and service traffic when the disaster is in effect. Running drain tests brings many benefits:

- verifying that runbooks can effectively mitigate and recover from various types of disasters and meet our recovery objectives;
- aid planning by identifying capacity needs during various disaster scenarios;
- testing the pace at which a service can offload traffic without overwhelming its downstream systems;
- auditing how shared resources are utilized to identify resource bottlenecks;
- tease apart complex inter-system dependencies and continuously discover new dependencies.

A drain test is not expected to have any user-visible or service-level impact. If this expectation is not met, we follow up with the engineering teams to understand why a given disaster scenario was not handled well, and schedule followup tests to verify fixes.

3.2 Failure Mitigation

Maelstrom was initially built for mitigating disasters of physical infrastructure. We experience a handful of incidents each year that result in the temporary catastrophic loss of one or more datacenters, usually due to power or network outage. We mitigate and recover from these disasters using datacenter evacuation runbooks.

Over time, our practice of rigorously verifying runbooks via drain tests has resulted in its evolution as a trusted tool for handling a wide variety of failures, including service-level incidents caused by software errors including bugs and misconfiguration. These service-level incidents are an order of magnitude more frequent. Note that most service incidents are recovered by reverting the buggy code or configuration changes, so traffic drains are rare. We will discuss how Maelstrom is used to deal with various failure scenarios in §5.1.

The actual failures and disasters are mitigated using the same runbooks as drain tests. Drain tests are fully automated—operators are only paged when the test triggers unexpected issues. During a disaster, operators may choose to accelerate steps to speed up mitigation.

4 Design and Implementation

4.1 Design Principles

Composability. Disaster mitigation and recovery are rarely done by a single red button, but through procedures consisting of a series of interdependent tasks. In our experience, despite the heterogeneity of system-specific procedures, they share common structures and can be composed of a common set of primitives. Maelstrom enables services to implement their own runbooks by composing various primitives. Composability offers a number of benefits: 1) it allows Maelstrom to exploit parallelism among primitive tasks; 2) enforces modularity and reusability of mitigation- and recovery-related code, and 3) makes runbooks easy to understand and maintain.

Separation of policy and mechanism. Maelstrom separates *policies* that define how traffic should be drained and restored in a specific disaster scenario and the *mechanisms* for executing traffic shifts and other related operations (cf. §2.2).

Safety as a constraint. Disaster mitigation and recovery themselves must not create new outages—when shifting traffic, Maelstrom should avoid cascading failures that overload the remaining healthy datacenters. Further, drain tests as a regular operation should not have any user-visible, service-level impact.

Embracing human intervention. We have learned that it is critical for a disaster mitigation and recovery system to embrace human intervention, even with fully automated runbooks (cf. §6). Extensive visualization on top

Task Template	Parameter	Description
`TrafficShift`	{`vip_type`, `target`, `ratio`, ...}	Shift traffic into or out of a cluster or a datacenter (specified by `target`): `vip_type` specifies the traffic, `ratio` specifies the amount of traffic to shift;
`ShardShift`	{`service_id`, `target`, `ratio`, ...}	Move persistent data shards into or out of a cluster or a datacenter via our shard manager (`target` and `ratio` have same semantics as in `TrafficShift`)
`JobUpdate`	{`operation`, `job_ids`, ...}	Stop or restart jobs running in the containers
`ConfigChange`	{`path`, `rev_id`, `content`, ...}	Revert and/or update configurations in our distributed configuration store

Table 2: Several common templates used to materialize tasks in runbooks, and their descriptions. Note that we have omitted the optional parameters that provide fine-grained control (e.g., latency and stability optimization) from this table.

of continuous monitoring helps operators understand the context. The UI design should minimize tedious operations to let operators focus on critical decision making.

4.2 Runbook Framework

A runbook is created through the Maelstrom UI by specifying the following information:

- *Task specifications.* Tasks are materialized by applying parameters to a library of templates. Table 2 lists several task templates and their description.
- *Dependency graph.* We use a directed acyclic graph (DAG) to represent dependencies amongst the tasks in a runbook. Every node T_i in the DAG refers to a task in the runbook. A directed edge $T_1 \rightarrow T_2$ represents a *dependency*: Task T_1 *must precede* Task T_2, which means that T_2 can only be scheduled for execution after T_1 is completed.
- *Conditions.* A task can have pre-conditions (checking if it is safe to start) and post-conditions (determining if it completed successfully). Pre-conditions are typically used as safeguards to ensure that the service is in a healthy state, while a post-condition could check if the target traffic reaches zero.
- *Health metrics.* Each task is associated with a number of service-health metrics, which are visualized in Maelstrom's UI to help human operators monitor the status of task execution.

Each service maintains its *service-specific runbook* for disaster mitigation and recovery. We also maintain an *evacuation runbook* for each of our datacemters which aggregates service-specific runbooks. The aggregation is accomplished by adding dependencies between tasks from different service-specific runbooks. We run the evacuation runbooks during each drain test and thus exercise all the related service-specific runbooks. Therefore, every drain test covers hundreds of services—we run tests far more often than we experience real failures.

4.3 Runtime Engine

Maelstrom's runtime engine is responsible for executing a runbook. The runtime engine consists of two components: 1) a *scheduler* that determines the order of executing tasks by tracking their dependencies, and 2) an *executor* that executes each task and validates the results.

- *Scheduler.* The scheduler generates an *optimal* schedule of task execution by parallelizing independent tasks. The scheduler marks a task `ready` for execution and sends its specification to the executor, when and only when all the parent tasks that must precede this task are `completed` and all the pre-conditions are satisfied. Note that this schedule is generated dynamically based on the runtime status of each task, and it supports operator intervention (e.g., skipping and stopping tasks).
- *Executor.* The executor materializes each task based on the parameters in the specification sent by the scheduler, and then executes the task. A task is executed in multiple *steps*. For example, a `TrafficShift` task for draining 100% web traffic out of a datacenter can be done in one step, or in five steps (with wait time in between)—each one draining 20%—based on the desired pacing (cf. §4.7).

Maelstrom models a task as a nondeterministic finite state machine, with a set of *stages* $\mathbb{S}$, a set of runtime inputs, *transitions* between stages, an initial stage $I \in \mathbb{S}$, and a set of exit stages $\mathbb{E} \subseteq \mathbb{S}$. A stage accepts one or more inputs, performs the desired action, and then optionally invokes a *transition* to the next stage. A stage can have multiple outgoing and incoming transitions. Each stage can generate outputs, as inputs for other stages. The executor starts from I and continues executing subsequent stages following the transitions, until reaching an exit stage. This design allows us to reuse stages as the basic unit for implementing task templates.

We implement a library of stages that capture common operations like instructing load balancers to alter traffic allocation [6, 19, 40, 51], managing containers and jobs [12, 48, 56], changing system configurations [49, 52], migrating data shards [1, 13], etc. We also implement various helper stages for communication and coordination, such as `Barrier`, `TimedWait`, and `ChangeLog`.

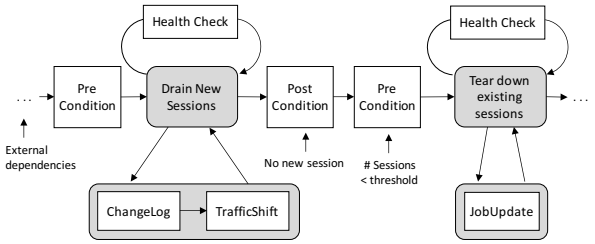

Figure 3: A runbook to drain a messaging service's sticky traffic. The runbook has two tasks: redirecting new sessions and tearing down existing sessions—both are executed in multiple steps.

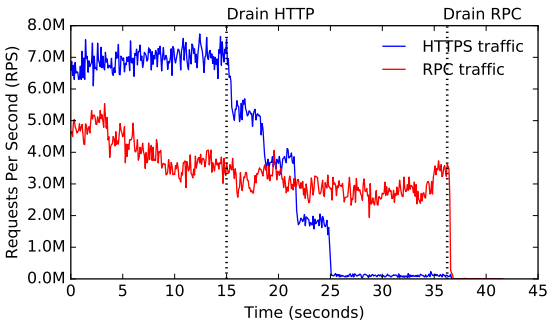

Figure 4: An example of a service dependency that determines the order of drains. A web service with HTTPS traffic communicates with backend services via RPC traffic (we say the web service *depends on* the backend services). So, the web service's HTTPS traffic must be drained *before* the backend service's RPC traffic is drained.

4.4 Executing Runbooks: Putting It All Together

Figure 3 illustrates how Maelstrom executes a service-specific runbook to drain traffic of a messaging service. Maelstrom executes two tasks in order: 1) redirecting new incoming session requests away, and 2) tearing down the remaining established sessions so clients can reconnect to machines in other datacenters. Maelstrom verifies that all of a task's parent dependencies are drained, and pre-conditions are satisfied. The second task uses its pre-condition as a safety check to confirm that the number of active sessions has dropped below a preset threshold to minimize the impact of tearing down all existing sessions. Maelstrom marks a task as `completed` when the post-condition is met.

Drains are blocked if the pre-/post-condition(s) are not met, because this signifies that the service is in an unexpected state. Maelstrom compares the duration of each task to the 75th percentile of the execution time of prior tests/incidents to determine whether the task is stuck in execution. If so, the operator will be alerted. We prioritize safety over speed, and stall subsequent operations until an operator intervenes—we find stalls to be a rare event that occurs only once every several dozen tests. When handling actual failures, Maelstrom allows human operators to override particular pre-/post-conditions if they wish—each of these overrides are logged and reviewed in postmortems to improve automation.

Maelstrom's traffic drains and restorations are guided by a variety of constraints:

- *physical and external constraints*, including network over-subscription within a datacenter, cache hit rate, I/O saturation in backend systems, etc.
- *service-specific constraints*—different types of traffic have distinct constraints. For example, draining sticky traffic is prone to a thundering-herd effect, because session establishment is resource intensive; draining stateful traffic leads to master promotion which requires the slave to have caught up with all updates, or requires state restoration from logs; restoring replication traffic requires syncing updates with volumes proportional to down time, and the sync speed is constrained by the available network bandwidth.

We show how Maelstrom respects these constraints when draining/restoring traffic of different types in §5.2.1.

4.5 Dependency Management

Broadly, we find that there are three common relationships amongst services that manifest as dependencies:

- *Bootstrapping*: A service depends on low-level system components to prepare the execution environment and setup configuration before serving traffic.
- *RPC*: A service makes RPC calls to fetch data from other services.
- *State*: Traffic can have states. For instance, a service with sticky traffic may depend on a proxy to coordinate and manage session establishment.

Discovery and sequencing. We employ a combination of methods to discover the aforementioned dependencies when onboarding a new service to Maelstrom. First, we design scenario-driven questionnaires to help service teams reason about their dependencies with upstream and downstream services under different types of failures and disasters. Moreover, we leverage our tracing systems [20, 29, 50] to analyze how the target service interacts with other services through RPC and network communications. Our analysis incorporates our service discovery systems [16, 18] to map the interactions to specific services. We further analyze the traces and logs of software components to reason about state-based dependencies based on the causal models of service behavior [17, 35]. Figure 4 illustrates the dependency between traffic of two services, which enforces the order of drains.

After identifying dependencies, the next challenge is to sequence drains amongst multiple interdependent systems in the right order. We tackle this problem by first organizing services into chains of parent-child dependencies in a disconnected dependency graph (we often do

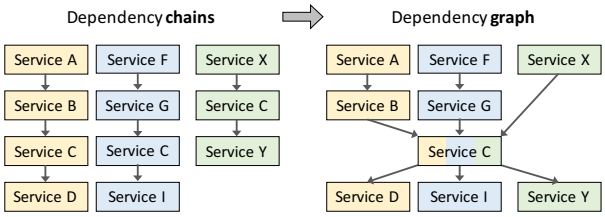

Figure 5: To build a runbook, we start with independent dependency *chains* (left). We identify *highly connected components (HCCs)*, like Service C, and merge the dependency chains at the HCCs to form a dependency *graph* (right).

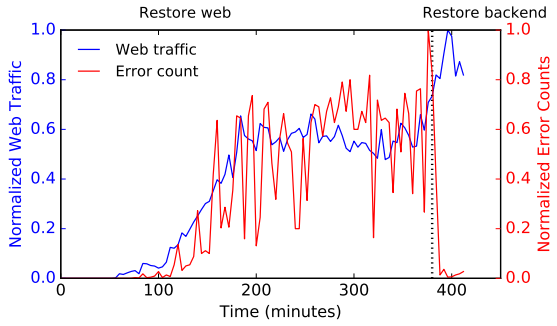

Figure 6: Restoring web traffic (left y axis) at minute 55 caused a proportional increase in errors (right y axis) until a backend service that the web traffic depended on was restored at minute 375.

not have one single complete graph at a time). Next, we identify common services across chains—the common services are often highly connected components (HCCs) that hold the dependency graph, as illustrated in Figure 5. Draining a HCC service will likely require us to drain its parents first; once the HCC service is drained, its children can likely be drained concurrently.

Continuous verification. We use Maelstrom to empirically verify that independent services can be drained and restored in parallel. For a new service, we use Maelstrom to cautiously validate the specified dependencies via small-scale drain tests, while closely monitoring the health of all involved services. We gradually enlarge the radius of the drain tests until all levels of traffic drains can be performed regularly. We find this process to be time consuming but worthwhile, as a principled way of verifying dependencies in a controlled, scalable manner.

Figure 6 illustrates how a previously unknown dependency was discerned in a drain test. This issue was caused by an out-of-order restore, where a backend service was drained while its dependent web traffic was restored. This made the error count proportional to the web traffic, as the web service was trying to query the unavailable backend. The error rate went down to zero after the operator also restored the backend service. After this test, the dependency was added into the runbook, together with improved monitoring of the error count.

Critical path analysis. Maelstrom performs automated critical path analysis after each drain test as well as each disaster mitigation and recovery event. Critical path analysis helps us optimize mitigation time by identifying bottlenecks in our dependency graphs.

When adding a new dependency into an existing runbook, we run drain tests to check if the new dependency is on the critical path or not. If it is, we engage with the service team responsible for that dependency to optimize drain and recovery performance as necessary. We also actively examine dependencies on slow, heavy tasks (e.g., data shard migration) to try to move these dependencies off the critical path. If a dependency lengthens the critical path, the service team evaluates whether the dependency is providing value given its failure mitigation cost.

Maelstrom allows a dependency to be tagged as *weak*, while by default all dependencies are *strong*. Strong dependencies affect correctness, and thus are expected to be respected in most failure scenarios. Weak dependencies affect a system's performance and reliability SLA. Drain tests respect both strong and weak dependencies. During a disaster, operators can override weak dependencies to stabilize a system or speed up mitigation. For instance, in case of a fibercut disconnecting a datacenter, an operator might move all user traffic to a different datacenter in a single step which minimizes user impact, but might affect the hit rate of any underlying caching systems, and possibly push backend storage systems to their limits. We curate weak dependencies by analyzing the dependencies on the critical path as discussed above. We also perform experiments that intentionally break weak dependencies in a controlled fashion to assess the corresponding service-level impact.

4.6 Preventing Resource Contention

Safely draining traffic with Maelstrom involves ensuring that shared resources (e.g., server compute capacity and network bandwidth) do not become overloaded during a drain. Our approach to reducing the effect of resource contention is guided by the following three principals:

- *Verifying capacity.* We verify that the shared infrastructure has enough capacity to absorb the spikes in utilization caused by draining with Maelstrom through regular testing. Since Facebook has full monitoring and control of its backbone network, we can observe how draining affects peak network capacity utilization. When bottlenecks arise during tests, we work with teams to update our routing policies, traffic tagging and prioritization schemes, or bandwidth reservation configuration so we can drain services safely. At the network level, we provision multiple diverse paths

both intra- and inter-datacenters, and plan 75% utilization for our switches [38].

- *Prioritizing important traffic.* To handle the event of a widespread failure where shared resources cannot support demand, we have a prioritization scheme for how we drain traffic from a datacenter. We prioritize draining user-facing traffic as soon as possible to limit the user-perceivable impact of a failure, and then drain stateful service traffic. This ensures that the effect of the drain on an end user are minimized and also minimizes the overhead of state migration.
- *Graceful degradation.* Finally, we plan for systems to degrade gracefully in the case of resource overload. Some systems employ PID controllers to reduce the complexity of serving requests (e.g., by incrementally turning off ranking algorithm complexity to reduce server compute capacity). Other systems are able to respond automatically to resource contention by performing large-scale traffic redirection, while safely accounting for the effect of traffic changes.

4.7 Pacing and Feedback Control

Maelstrom implements a closed feedback loop to pace the speed of traffic drains based on extensive health monitoring. The drain pace is determined by the `step_size` (traffic fraction to reduce) and `wait_time` before the next step. The runbook uses past drain parameters from tests/incidents as a starting value for step size and wait time. When running an actual drain, these parameters are further tuned to be more aggressive or conservative based on the health of underlying systems.

Our pacing mechanism seeks to balance safety and efficiency—we wish to drain as fast as possible without overloading other datacenters. Specifically, Maelstrom breaks down a drain operation into multiple steps, and for each step, tunes the weights such that no traffic shift breaches the health metrics of any datacenter. For instance, when draining web traffic from 100% to 0%, Maelstrom typically does not drain in one step (which could have ripple effect such as significant cache misses). Instead, the drain takes multiple steps (with specified `wait_time` in between), gradually increasing the traffic shift proportion, in order to allow cache and other systems to warm up with smoothly increasing load without getting overwhelmed. The health metrics are also displayed in Maelstrom's UI, so operators can audit operations and intervene as needed.

Maelstrom reads the health data maintained as time series. In our experience, a few key metrics from each service can provide good coverage of their health, and we infrequently need to add new metrics.

We use drain tests to experiment with various starting speeds. Based on the empirical mapping from speed to health metric impact, we tune the default value to the maximal speed without compromising health or safety.

4.8 Fault Tolerance

Maelstrom is designed to be highly fault tolerant in the presence of both component and infrastructure failures. We deploy multiple Maelstrom instances in geo-distributed datacenters so at least one instance is available even when one or more datacenters fail. We also have a minimal version of Maelstrom that can be built and run on any of our engineering development servers.

We verify the correctness of runbooks by leverage continuous tests that validate the invariants in every service's runbook including checking for circular dependencies, reachability (no inexistent dependencies), duplication, ordering (every drain step is undone with a restore), and configuration (mandatory parameters are always set). If a test fails, the service's oncall engineers will be notified to review the service's tests, dependencies, health indicators, etc. If all tests pass, but other correctness violations manifest during a drain test (e.g., due to insufficient tests), the disaster-recovery team will arbitrate between services to ensure that problems are fixed and continuous tests are updated. As discussed in §4.4, live locks (e.g., due to failures of task execution or condition checks) are prevented by tracking the execution time of tasks and comparing it with the 75th percentile of prior runtimes.

Maelstrom stores its metadata and runtime state in a highly-available, multi-homed database system. Both task and stage level state is recorded so both the scheduler and executor can be recovered in case of failure. Maelstrom also records the state of each task (`waiting`, `running`, or `completed`) into the database so it can resume at the last successful step of a drain or recovery procedure. For a running task, Maelstrom records the runtime state of each stage and transition in the database based on the state machine it generated. Hence, if there is a crash of Maelstrom (including both the scheduler and the executor), we can use standard recovery techniques to read the last committed state from the database to initialize Maelstrom and resume the execution.

Maelstrom also relies on a highly-available time-series database to provide it with health monitoring data [43]. Our monitoring database continuously provides data even in the presence of failures by varying the resolution of data points.

5 Evaluation

Maelstrom has been in use at Facebook for more than four years, where it has been used to run hundreds of drain tests, and helped mitigate and recover from 100+ datacenter-level failures and service-level incidents.

Our evaluation answers the following questions:

- Does Maelstrom enable us to mitigate and recover from real disasters safely and efficiently?
- Does Maelstrom provide a safe and efficient methodology for regular drain tests?
- How quickly does Maelstrom drain and restore different types of traffic?

5.1 Mitigating Disasters

Power and network outages. Maelstrom is one of our primary tools to mitigate and recover from disasters impacting physical infrastructure whether caused by power outages or backbone network failures, resulting in the total or partial unavailability of datacenters. Taking the network as an example, a single fibercut almost never disconnects a datacenter; usually, one link is lost, and network flows reroute over alternate paths. This rerouting procedure often takes tens of seconds, and could impact users. On the other hand, a single-point-of-failure link, such as a trans-Atlantic or trans-Pacific optical cable, can get cut occasionally [38]—these incidents are severe in both magnitude and duration-to-fix, thus requiring datacenter drains.

A recent incident caused by fibercuts led to the loss of over 85% of the capacity of the backbone network that connects a datacenter to our infrastructure. This incident was immediately detected as we experienced a dip in site egress traffic. The disaster was mitigated by the site operators using Maelstrom to drain all user and service traffic out of the datacenter in about 1.5 hours, with most user-facing traffic drained in about 17 minutes. The remaining network capacity was used to replicate data to the storage systems resident in that datacenter (which helps efficiently redirect user traffic back, once the fiber is repaired). It took several hours to repair the backbone network, at which point we used Maelstrom to restore all traffic back to the datacenter.

Note: when a datacenter is drained, users may experience higher latency, as they are redirected to a remote datacenter, or experience reconnection (only for sticky services). Draining faster could reduce the amount of time during which users experience increased latency. We are continually working to decrease dependencies and optimize constraints to enable faster drains.

Software failures. Maelstrom is also used to respond to service-level incidents caused by software errors, including bugs and misconfiguration [25, 26, 37, 57, 58]. These incidents are typically triggered in two ways:

- *software errors in ongoing rollouts.* Despite the wide adoption of end-to-end testing, canaries, and staged rollout, bugs or misconfiguration can still make their way into production systems.
- *latent software errors.* A software release or configuration change might trigger latent bugs or misconfiguration residing in production systems.

In both cases, any error or disruption ought to trigger an alert and inform operators. The operators need to decide between two options: reverting the offending change(s), or fixing forward after diagnosing the problem.

Unfortunately, neither of these options is trivial. First, it may take time to identify the offending change (or changes) due to the challenge of debugging large-scale distributed systems. Second, rollback to an early version may cause other issues such as version incompatibility, which can result in other failures. Third, it takes time to diagnose, code up a fix, test it thoroughly, and then deploy it into production [60]. During this time, the error continues to manifest in production.

Maelstrom provides a pragmatic solution for reducing the impact of failures by moving diagnosis and recovery out of the critical path—it simply drains service-specific traffic from the failing datacenters when failures are detected. We find that this mitigation approach is robust when paired with a locality-based, staged rollout strategy for all software and configuration changes.

Maelstrom was used to mitigate a recent service incident where a configuration change was rolled out to all instances of our search aggregator deployed in one of our datacenters. The configuration change inadvertently triggered a new code path and exposed a latent bug in the aggregator code (a dangling pointer). All the instances in the datacenter immediately crashed due to segfaults.

This incident was mitigated by draining service traffic from the datacenter where the misconfigured search aggregators were deployed. Detecting the incident took only 2 minutes as it immediately triggered alerts. It took 7 minutes to drain service requests out of the affected datacenter using Maelstrom. Failure diagnosis (identifying the root cause) took 20 minutes. Thus, Maelstrom reduced the duration of service-level impact by about 60%.

5.2 Draining Different Types of Traffic

5.2.1 Service-specific Traffic

Stateless traffic. Figure 7 shows how Maelstrom drains stateless web traffic of one of our services out of a target datacenter in a recent drain test. We normalize the data, because different datacenters have different sizes ((in terms of the magnitude of traffic served) and we want to highlight the relative trends of each datacenter during the drain. The runbook for draining web traffic includes a `TrafficShift` task which manipulates the edge weights of the target datacenter by applying a *drain multiplier* between [0.0, 1.0]. The drain was executed in multiple steps indicated by the drain multiplier changes in Figure 7. Splitting the drain into multiple steps prevents traffic from shifting too fast and overloading the other datacenters (cf. §4.7).

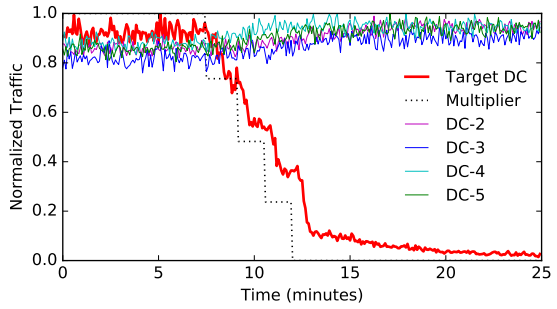

Figure 7: Draining stateless traffic. We apply a multiplier (dotted line) to the edge weight (cf. §2) of stateless traffic in a Target DC to drain stateless traffic. The long tail of Target DC traffic is from DC-internal requests that are controlled separately from the edge weight.

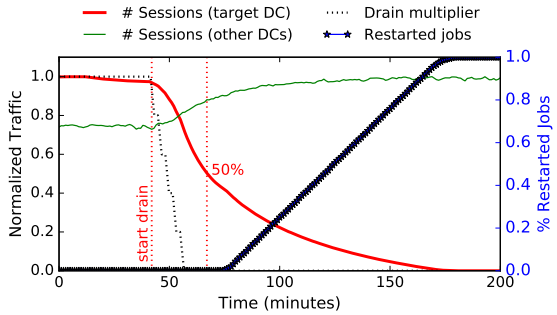

Figure 8: Draining sticky traffic. We drain sticky traffic by first applying a multiplier (dotted line) to the edge weight of a target DC (similar to stateless traffic). We then restart the jobs in the target DC to force already-established sessions in the target DC to reconnect in other DCs.

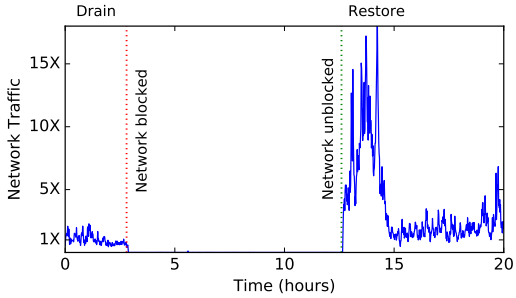

Figure 9: Draining and restoring replication traffic (normalized by the average traffic volume before the network block). We drained replication traffic before the network block, and a surge of replicated data saturated the network during restoration. This problem has been fixed by network-aware recovery.

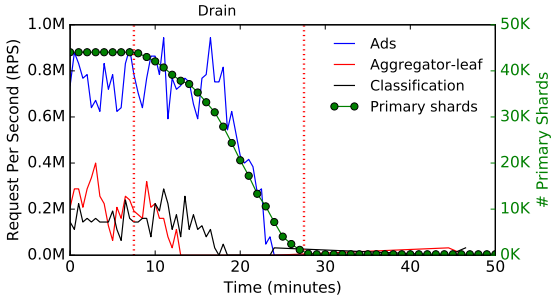

Figure 10: Draining stateful traffic. Maelstrom moves primary data shards from storage system and simultaneously drains traffic from the services that access the storage system.

As shown in Figure 7, the traffic follows the changes of the drain multiplier instantly. *Maelstrom can drain stateless traffic fast*. Maelstrom can drain traffic of most of our web services out of a datacenter in less than 10 minutes without any user-visible, service-level impact. The 10-minute duration is used as a baseline for draining web traffic during real disasters (cf. §5.1).

Sticky traffic. Figure 8 shows how Maelstrom drains sticky traffic for a messaging service. This runbook contains two tasks as described in §4.4: (1) changing edge weights to redirect new, incoming session requests (at the 42nd minute), and then (2) tearing down established sessions by restarting container jobs, if the client can still connect to the datacenter (at the 75th minute). Figure 8 shows the effects of these two tasks—it took about 25 minutes to reduce the number of sessions down to 50%, and the remaining time to restart jobs and reset connections. Note that we need to pace job restarts to avoid thundering-herd effects caused by computationally expensive session establishment. During real disasters, we find that clients' connections are severed due to network disconnections or server crashes, so drains are faster.

Replication traffic. Figure 9 shows how Maelstrom drains and restores replication traffic of a multi-tenant storage system when the target datacenter was subject to a network partition. Maelstrom drains replication traffic by disabling the replication to the storage nodes in the target datacenter after first pointing read traffic away from the replicas. The drain was smooth, but the restoration caused an incident. Upon enabling replication for recovery, the system attempted to resolve its stale state as quickly as possible, transferring data from other datacenters at about 10× the steady-state rate. This saturated network capacity at multiple layers in our backbone and datacenter network fabric and triggered production issues in other systems that shared the network. In this case, we mitigated the issue by draining user-facing traffic depending on these systems out of the datacenter, and subsequently adopting a network-aware transfer strategy.

Stateful traffic. Figure 10 shows how Maelstrom drained stateful traffic of three services: an "ads", an "aggregator–leaf", and a "classification" service. All three services store their data in a multi-tenant stateful storage system where the data are sharded. The storage system distributes replicas of each shard in multiple datacenters to ensure high availability. During the drain, Maelstrom promotes a replica outside of the datacenter to be the new primary shard, and then shifts traffic to it.

Figure 10 plots the fraction of primary shards in the

Traffic	Service	# Tasks	# Steps	Drain Time
Stateless	Web service	1	10	10 min
Sticky	Messaging	2	1 → 5	3 min → 61 min
Replication	KV store (replica)	1	1	3 min
Stateful	KV store (master)	1	24	18 min

Table 3: Time for draining different types of traffic of representative services at Facebook. The time is collected from our recent drain tests. For sticky traffic, → denotes the two tasks for draining the traffic.

Runbook	# Tasks	# Dep.	# Tasks on CP
Mitigation (drain)	79	109	8 (9.6%)
Recovery (restore)	68	93	5 (7.4%)

Table 4: Aggregate statistics of the runbooks that drain and restore all user-facing traffic in one of our datacenters, respectively. "CP" is an abbreviation of critical path.

datacenter being drained. From the perspective of the storage system, each of these services is independent of the others because their data are sharded separately. This allows Maelstrom to drain writes and promote their masters in parallel while respecting shared resource constraints. Note that Figure 10 only highlights three services for clarity—Maelstrom drains tens of thousands of shards for hundreds of services in the datacenter.

Timing. Table 3 shows the time for Maelstrom to drain one representative service that displays each type of traffic pattern. Note that they vary significantly in duration from 3 minutes for a replication system where a drain is as simple as redirecting read requests and shutting the instance down, to a sticky service that takes 61 minutes to drain at its natural pace. Note that we encourage services to plan for different disaster scenarios but do not force particular policies for timing or reliability unless the service is in the critical path for evacuating a datacenter.

5.2.2 Draining All the Traffic of a Datacenter

Figure 11 shows a drain test that drains a wide variety of service traffic hosted in the datacenter, including both user traffic as well as internal service traffic. We see that no single service constitutes a majority of the traffic. Maelstrom achieved a high degree of parallelism while maintaining safety by ordering drains according to the dependency graph encoded in the runbook (cf. §4.5).

Figure 12 is a complement of Figure 11 that depicts the normalized utilization of both the target datacenter that is being drained of traffic, and the other datacenters that the traffic is being redirected to and then restored from. We see that once all traffic is drained, the utilization of the target datacenter drops to zero. Meanwhile, the utilization of the other datacenters increase as they need to serve more traffic. None of the remaining datacenters were overloaded—traffic is evenly distributed to the available datacenters.

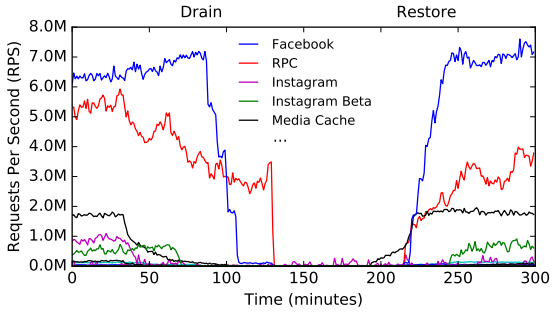

Figure 11: Draining and restoring traffic for 100+ production systems in a datacenter. Each line corresponds to the traffic of a specific service. "Facebook" refers to the traffic of Facebook's main web service.

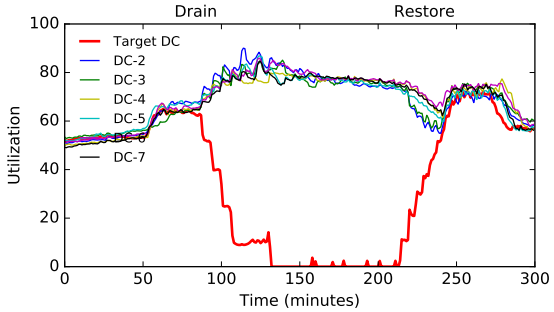

Figure 12: Datacenter utilization when the traffic of an entire datacenter is drained and restored.

Aggregate statistics. Table 4 analyzes the runbooks used to drain and restore all user-facing traffic and their dependent services from one of our datacenters. Our goal is to provide insight into the policies we encode in Maelstrom. The mitigation runbook consists of 79 tasks with 109 dependencies, of which less than 10% of the tasks are on the critical path. Note that this minimal critical path is not an organic outcome but rather the result of continually optimizing and pruning dependencies over four years, based on the critical path analysis described in §4.5. The recovery component has fewer tasks on the critical path implying that there is higher parallelism during recovery than mitigation.

The histogram displayed in Figure 13 shows that there are 13 different template types in use in this runbook. Further, we find that `TrafficShift` is the most frequently used tempate. This is because most user traffic is delivered over HTTP to our web servers, and hence manipulated by tuning weighs in our software load balancers.

Figure 14 plots the number of steps per tasks—observe that most tasks were executed in more than one step, and several were paced in more than 10 steps, during both the mitigation and recovery phases.

5.3 Efficiency

We leverage drain tests to estimate how fast we are able to mitigate a real disaster. In this section, we focus on

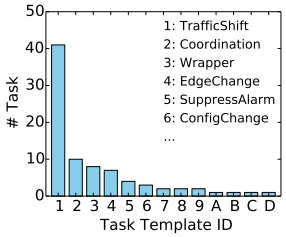

Figure 13: Histogram of number of tasks to drain and restore user-facing traffic.

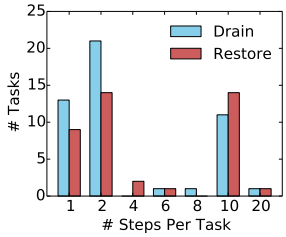

Figure 14: Histogram of number of steps per task at runtime when draining and restoring user-facing traffic.

Phases (Traffic Shift)	Time Duration	
	Maelstrom	Sequential
Drain web traffic	10 min	×1
Drain all user-facing traffic	40 min	×6.6
Drain all service traffic	110 min	×6.3
Restore web traffic	50 min	×1
Restore internal service traffic	1.5 hour	×4.3
Restore all service traffic	2 hour	×5.6

Table 5: Time duration of draining and restoring traffic of a datacenter. The data are collected from a real disaster for which we drained all the traffic out of the entire datacenter (and restored it after repair).

the scenario where an entire datacenter fails.

Table 5 shows the time taken by Maelstrom to drain and restore traffic in different phases from and back into a datacenter. The traffic of Facebook's main web service, referred to as *web traffic* in Figure 11 and Table 5, is used as the baseline. It takes less than a minute to propagate a change of drain multiplier (cf. §5.2.1) to Edge LBs when draining web traffic. Maelstrom typically does not drain web traffic in one step but gradually adjusts the speed based on health monitoring. Currently, it takes about 10 minutes to fully drain the web traffic, and 50 minutes to restore it. Restoration is slow as we wish to minimize backend overload due to cold caches. Overall, it takes 40 minutes to drain all the user-facing traffic, and 110 minutes to drain all service traffic including the traffic of internal systems.

We next evaluate whether Maelstrom provides an efficient methodology to mitigate and recover from a datacenter failure. We calculate the time needed to drain and restore the datacenter sequentially by summing up the time used by the traffic drain and restoration of every service in a runbook. As shown in Table 5, sequential drain and restoration would take up to 6× longer than Maelstrom. The results verify the efficiency of Maelstrom and demonstrate the importance of parallelizing operations.

Note that restoring traffic back to a datacenter encounters a narrower bottleneck where a single target datacenter is receiving more load, in comparison to draining traffic from a datacenter to many others. We prioritize restoring user-facing traffic back into the datacenter as this minimizes the risk of exposing users to multiple independent datacenter failures.

6 Experience

Drain tests help us understand interactions amongst systems in our complex infrastructure. We find drain tests to be one of the most efficient ways to understand how a system fits into our infrastructure. A successful drain test is a validation of our tooling which tracks inter-system dependencies and health monitoring, while a failed drain test reveals gaps in our understanding. We find that drain tests are truer validators of inter-service dependencies than other methods we have experimented with, such as methods based on log and trace analysis.

Drain tests help us prepare for disaster. Prior to running regular drain tests, we often encountered delays in disaster mitigation due to our tools having atrophied as they did not account for evolving software, configuration and shared infrastructure components. Drain tests exercise our tooling continuously and confirm operational behavior in a controlled manner.

Drain tests are challenging to run. We observe that infrastructure changes, new dependencies, software regressions, bugs and various other dynamic variables inevitably trigger unexpected issues during a drain test. We strive to continually tune and improve our monitoring systems to quickly assess impact and remediate issues. Further, we have focused on communication and continually educate other teams on our tests and their utility so our systems are prepared.

Automating disaster mitigation completely is not a goal. Our initial aspiration was to take humans out of the loop when mitigating disasters. However, we have learned that it is prohibitively difficult to encode the analytical and decision making skills of human operators without introducing tremendous complexity. The current design of Maelstrom is centered around helping operators triage a disaster and efficiently mitigate it using our tools and well-tested strategies. We intentionally expose runtime states of each task and allow human operators to override operations. This strategy has proved simpler and more reliable than attempting to automate everything. Our experience with operators confirms that Maelstrom significantly reduces operational overhead and the errors that are inevitable in a manual mitigation strategy.

Building the right abstractions to handle failures is important, but takes time and iteration. We have evolved Maelstrom's abstractions to match the mental model of the teams whose systems are manipulated by Maelstrom. We find that our separation of runbooks and tasks allows each service team to focus on maintaining their own service-specific policies without the need to (re-)build mechanisms. This separation also allows us to

efficiently onboard new services, and ensure a high quality bar for task implementation. Lastly, we find that as new systems are onboarded, we need to create new task templates and other supporting extensions to satisfy their needs.

7 Limitation and Discussion

Maelstrom, and draining traffic in general to respond to outages, is not a panacea. In fact, we find that there is no single approach or mechanism that can mitigate all the failures that might affect a large scale Internet service.

Capacity planning is critical to ensure that healthy datacenters and shared infrastructure like backbone and the datacenter network fabric have sufficient headroom to serve traffic from a failing datacenter. Drain tests can help validate capacity plans but shortfalls can still be difficult to address as it takes time to purchase, deliver, and turn-up machines and network capacity. If a capacity shortfall were to exist, it is wholly possible that draining traffic from a failing datacenter might overwhelm healthy datacenters and trigger cascading failures. Our strategy is to work in lockstep with capacity planning, and also regularly perform drills (storm tests) that isolate one or more datacenters and confirm that the remaining capacity can serve all our user and service needs.

If an outage is triggered by malformed client requests, or a malicious payload, redirecting traffic away from a failing datacenter to healthy ones will spread the failure. We handle this scenario by applying traffic shifts in multiple steps; the first step is intentionally small so we can monitor all systems in the target datacenter and confirm their health before initiating a large-scale drain.

Traffic drains may not always be the fastest mitigation strategy. Specifically, outages triggered by buggy software or configuration changes might be mitigated faster by reverting suspect changes. We expect operators to decide which mitigation strategy to use.

8 Related Work

Many prior papers study failures and outages in large scale systems running on cloud infrastructure [21,25–27, 30, 37, 39, 41, 42, 61]. These papers share several common conclusions: (1) outage is inevitable at scale when systems are exposed to a myriad set of failure scenarios, (2) large-scale, complex systems cannot be completely modeled for reliability analysis, and thus failure response cannot be predicted in advance; and (3) the philosophy of building and operating highly-available services is to anticipate disasters and proactively prepare for them. We built Maelstrom to mitigate and recover from failures.

Many prior studies have focused on fast recovery [11, 14, 42, 44, 45] and efficient diagnosis [9, 15, 34, 62, 63]. While these studies help resolve the root cause of failures and outages in a timely manner, our experience shows that even this speedy resolution exposes users to a frustrating experience. We use Maelstrom to mitigate failure and reduce user-visible impact, which buys us time for thorough diagnosis and recovery.

Fault-injection testing has been widely adopted to continuously exercise the fault tolerance of large-scale systems [2–5,8,24,33,54]. Maelstrom is *not* a fault-injection tool like Chaos Monkey [8, 54]. Specifically, Maelstrom is not designed for simulating machine- or component-level failures, but rather for responding to disastrous failures at the datacenter level.

Drain tests are different from annual, multi-day testing drills such as DiRT [32] and GameDay [46]. Fundamentally, drain tests focus on testing mitigation and recovery for user traffic and services without fully isolating or shutting down a datacenter. Drain tests are fully automated and run frequently. In contrast, DiRT and GameDay intentionally disconnect or shutdown one or more datacenters fully and exercise the entire technical and operational spectrum, including detection, mitigation, escalation, and recovery components of a response strategy. Aside: we also use Maelstrom in our own periodic large-scale, DiRT-like drills to verify the capability and end-to-end effectiveness of our disaster response strategies.

Kraken and TrafficShifter [36, 55] leverage live traffic for load testing to identify resource utilization bottlenecks; TrafficShift [31] can also drain stateless web traffic. Maelstrom uses similar underlying traffic management primitives to Kraken (cf. §2), and goes beyond TrafficShift in its capability to drain different traffic types, track dependencies, and order operations.

Traffic draining has been anecdotally mentioned as a method for mitigating failures for site reliability [10, 31, 32]. To our knowledge, existing systems only work with one service or one type of traffic, and cannot drain different types of traffic of heterogenous services. Maelstrom serves as the sole system for draining and restoring *all* the services in *all* the datacenters at Facebook.

9 Conclusion

As our infrastructure grows, we have learned that it is critical to develop trusted tools and mechanisms to prepare for and respond to failure. We describe Maelstrom which we have built and improved over the past four years to handle datacenter-level disasters. Maelstrom tracks dependencies amongst services and uses a closed feedback loop to handle outages safely and efficiently. We propose drain tests as a new testing strategy to identify dependencies amongst services, and ensure that tools and procedures for handling failure are always up to date. Maelstrom has been in production at Facebook for over four years. It has run hundreds of drain tests and helped mitigate and recover from more than 100 disasters.

Much of the focus of Maelstrom has been around ensuring that Facebook stays available when an incident affects an entire datacenter. In practice, we find that many incidents affect only a subset of hardware and software systems rather than entire datacenters. Our next focus is on building tools to isolate outages to the minimal subset of the systems they affect.

Acknowledgments

We thank the reviewers and our shepherd, Justine Sherry, for comments that improved this paper. We thank Chunqiang Tang for insightful feedback on an early draft. Much of our work on Disaster Readiness, and drain tests in particular, would not be possible without the support of engineering teams across Facebook. We thank the numerous engineers who have helped us understand various systems, given us feedback on the tooling, monitoring, and methodology of Maelstrom, and helped us improve the reliability of our infrastructure.

References

[1] ADYA, A., MYERS, D., HOWELL, J., ELSON, J., MEEK, C., KHEMANI, V., FULGER, S., GU, P., BHUVANAGIRI, L., HUNTER, J., PEON, R., KAI, L., SHRAER, A., MERCHANT, A., AND LEV-ARI, K. Slicer: Auto-Sharding for Datacenter Applications. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI'16)* (Savannah, GA, USA, Nov. 2016).

[2] ALLSPAW, J. Fault Injection in Production: Making the case for resilience testing. *Communications of the ACM (CACM) 55*, 10 (Oct. 2012), 48–52.

[3] ALVARO, P., ANDRUS, K., SANDEN, C., ROSENTHAL, C., BASIRI, A., AND HOCHSTEIN, L. Automating Failure Testing Research at Internet Scale. In *Proceedings of the 7th ACM Symposium on Cloud Computing (SoCC'16)* (Santa Clara, CA, USA, Oct. 2016).

[4] ALVARO, P., ROSEN, J., AND HELLERSTEIN, J. M. Lineage-driven Fault Injection. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (SIGMOD'15)* (Melbourne, Victoria, Australia, May 2015).

[5] ALVARO, P., AND TYMON, S. Abstracting the Geniuses Away from Failure Testing. *Communications of the ACM (CACM) 61*, 1 (Jan. 2018), 54–61.

[6] ARAÚJO, J. T., SAINO, L., BUYTENHEK, L., AND LANDA, R. 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, Apr. 2018).

[7] BARROSO, L. A., CLIDARAS, J., AND HÖLZLE, U. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-scale Machines*, 2 ed. Morgan and Claypool Publishers, 2013.

[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] BESCHASTNIKH, I., WANG, P., BRUN, Y., AND ERNST, M. D. Debugging Distributed Systems. *Communications of the ACM (CACM) 59*, 8 (Aug. 2016), 32–37.

[10] BEYER, B., JONES, C., PETOFF, J., AND MURPHY, N. R. *Site Reliability Engineering: How Google Runs Production Systems*. O'Reilly Media Inc., 2016.

[11] BROWN, A. B., AND PATTERSON, D. A. Undo for Operators: Building an Undoable E-mail Store. In *Proceedings of the 2003 USENIX Annual Technical Conference (USENIX ATC'03)* (San Antonio, TX, USA, June 2003).

[12] BURNS, B., GRANT, B., OPPENHEIMER, D., BREWER, E., AND WILKES, J. Borg, Omega, and Kubernetes: Lessons learned from three container-management systems over a decade. *Communications of the ACM (CACM) 59*, 5 (May 2016), 50–57.

[13] BYKOV, S., GELLER, A., KLIOT, G., LARUS, J. R., PANDYA, R., AND THELIN, J. Orleans: Cloud Computing for Everyone. In *Proceedings of the 2nd ACM Symposium on Cloud Computing (SoCC'11)* (Cascais, Portugal, Oct. 2011).

[14] CANDEA, G., KAWAMOTO, S., FUJIKI, Y., FRIEDMAN, G., AND FOX, A. Microreboot – A Technique for Cheap Recovery. In *Proceedings of the 6th USENIX Conference on Operating Systems Design and Implementation (OSDI'04)* (San Francisco, CA, USA, Dec. 2004).

[15] CHEN, A., HAEBERLEN, A., ZHOU, W., AND LOO, B. T. One Primitive to Diagnose Them All: Architectural Support for Internet Diagnostics. In *Proceedings of the 12th European Conference on Computer Systems (EuroSys'17)* (Belgrade, Serbia, Apr. 2017).

[16] CHESHIRE, S., AND KROCHMAL, M. Dns-based service discovery. *Internet Engineering Task Force (IETF)*, 6763 (Feb. 2013).

[17] 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 (OSDI'14)* (Broomfield, CO, USA, Oct. 2014).

[18] DINESH, S. Service Discovery and Configuration on Google Cloud Platform, Jan. 2016. https://medium.com/google-cloud/service-discovery-and-configuration-on-google-cloud-platform-spoiler-it-s-built-in-c741eef6fec2.

[19] EISENBUD, D. E., YI, C., CONTAVALLI, C., SMITH, C., KONONOV, R., MANN-HIELSCHER, E., CILINGIROGLU, A., CHEYNEY, B., SHANG, W., AND HOSEIN, J. D. 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, Mar. 2016).

[20] FONSECA, R., PORTER, G., KATZ, R. H., SHENKER, S., AND STOICA, I. X-Trace: A Pervasive Network Tracing Framework. In *Proceedings of the 4th USENIX Symposium on Networked Systems Design and Implementation (NSDI'07)* (Cambridge, MA, USA, Apr. 2007).

[21] FORD, D., LABELLE, F., POPOVICI, F. I., STOKELY, M., TRUONG, V.-A., BARROSO, L., GRIMES, C., AND QUINLAN, S. Availability in Globally Distributed Storage Systems. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation (OSDI'10)* (Vancouver, BC, Canada, Oct. 2010).

[22] GOOGLE CLOUD. Disaster Recovery Cookbook, 2017. https://cloud.google.com/solutions/disaster-recovery-cookbook.

[23] GOVINDAN, R., MINEI, I., KALLAHALLA, M., KOLEY, B., AND VAHDAT, A. Evolve or Die: High-Availability Design Principles Drawn from Googles Network Infrastructure. In *Proceedings of the 2016 ACM SIGCOMM Conference (SIGCOMM'16)* (Florianópolis, Brazil, Aug. 2016).

[24] GUNAWI, H. S., DO, T., JOSHI, P., ALVARO, P., HELLERSTEIN, J. M., ARPACI-DUSSEAU, A. C., ARPACI-DUSSEAU, R. H., SEN, K., AND BORTHAKUR, D. FATE and DESTINI: A Framework for Cloud Recovery Testing. In *Proceedings of the 8th USENIX Symposium on Networked Systems Design and Implementation (NSDI'11)* (Boston, MA, USA, Mar. 2011).

[25] GUNAWI, H. S., HAO, M., LEESATAPORNWONGSA, T., PATANA-ANAKE, T., DO, T., ADITYATAMA, J., ELIAZAR, K. J., LAKSONO, A., LUKMAN, J. F., MARTIN, V., AND SATRIA, A. D. 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, Nov. 2014).

[26] GUNAWI, H. S., HAO, M., SUMINTO, R. O., LAKSONO, A., SATRIA, A. D., ADITYATAMA, J., AND ELIAZAR, K. J. 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, Oct. 2016).

[27] GUNAWI, H. S., SUMINTO, R. O., SEARS, R., GOLLIHER, C., SUNDARARAMAN, S., LIN, X., EMAMI, T., SHENG, W., BIDOKHTI, N., MCCAFFREY, C., GRIDER, G., FIELDS, P. M., HARMS, K., ROSS, R. B., JACOBSON, A., RICCI, R., WEBB, K., ALVARO, P., RUNESHA, H. B., HAO, M., AND LI, H. Fail-Slow at Scale: Evidence of Hardware Performance Faults in Large Production Systems. In *Proceedings of the 16th USENIX Conference on File and Storage Technologies (FAST'18)* (Oakland, CA, USA, Feb. 2018).

[28] GUPTA, A., AND SHUTE, J. High-Availability at Massive Scale: Building Google's Data Infrastructure for Ads. In *Proceedings of the 9th Workshop on Business Intelligence for the Real Time Enterprise (BIRTE'15)* (Kohala Coast, HI, USA, Aug. 2015).

[29] KALDOR, J., MACE, J., BEJDA, M., GAO, E., KUROPATWA, W., O'NEILL, J., ONG, K. W., SCHALLER, B., SHAN, P., VISCOMI, B., VENKATARAMAN, V., VEERARAGHAVAN, K., AND SONG, Y. J. Canopy: An End-to-End Performance Tracing And Analysis System. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP'17)* (Shanghai, China, Oct. 2017).

[30] KEETON, K., SANTOS, C., BEYER, D., CHASE, J., AND WILKES, J. Designing for Disasters. In *Proceedings of the 3rd USENIX Conference on File and Storage Technologies (FAST'04)* (San Francisco, CA, USA, Mar. 2002).

[31] KEHOE, M., AND MALLAPUR, A. TrafficShift: Avoiding Disasters at Scale. In *USENIX SRECon'17 Americas* (San Francisco, CA, USA, Mar. 2017).

[32] KRISHNAN, K. Weathering the Unexpected. *Communications of the ACM (CACM) 55*, 11 (Nov. 2012), 48–52.

[33] LEESATAPORNWONGSA, T., AND GUNAWI, H. S. The Case for Drill-Ready Cloud Computing. In *Proceedings of the 5th ACM Symposium on Cloud Computing (SoCC'14)* (Seattle, WA, USA, Nov. 2014).

[34] LIU, X., GUO, Z., WANG, X., CHEN, F., LIAN, X., TANG, J., WU, M., KAASHOEK, M. F., AND ZHANG, Z. D3S: Debugging Deployed Distributed Systems. In *Proceedings of the 5th Conference on Symposium on Networked Systems Design and Implementation (NSDI'08)* (San Francisco, CA, USA, Apr. 2008).

[35] MACE, J., ROELKE, R., AND FONSECA, R. Pivot Tracing: Dynamic Causal Monitoring for Distributed Systems. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP'15)* (Monterey, CA, USA, Oct. 2015).

[36] MALLAPUR, A., AND KEHOE, M. TrafficShift: Load Testing at Scale, May 2017. https://engineering.linkedin.com/blog/2017/05/trafficshift--load-testing-at-scale.

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

[38] MEZA, J., XU, T., VEERARAGHAVAN, K., AND SONG, Y. J. A Large Scale Study of Data Center Network Reliability. In *Proceedings of the 2018 ACM Internet Measurement Conference (IMC'18)* (Boston, MA, USA, Oct. 2018).

[39] MOGUL, J. C., ISAACS, R., AND WELCH, B. Thinking about Availability in Large Service Infrastructures. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems (HotOS XVI)* (Whistler, BC, Canada, May 2017).

[40] OLTEANU, V., AGACHE, A., VOINESCU, A., AND RAICIU, C. Stateless Datacenter Load-balancing with Beamer. In *Proceedings of the 15th USENIX Symposium on Networked Systems Design and Implementation (NSDI'18)* (Renton, WA, USA, Apr. 2018).

[41] OPPENHEIMER, D., GANAPATHI, A., AND PATTERSON, D. A. 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, Mar. 2003).

[42] PATTERSON, D., BROWN, A., BROADWELL, P., CANDEA, G., CHEN, M., CUTLER, J., ENRIQUEZ, P., FOX, A., KICIMAN, E., MERZBACHER, M., OPPENHEIMER, D., SASTRY, N., TETZLAFF, W., TRAUPMAN, J., AND TREUHAFT, N. Recovery-Oriented Computing (ROC): Motivation, Definition, Techniques, and Case Studies. Tech. Rep. UCB//CSD-02-1175, University of California Berkeley, Mar. 2002.

[43] 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 (VLDB'15)* (Kohala Coast, HI, USA, Aug. 2015).

[44] QIN, F., TUCEK, J., SUNDARESAN, J., AND ZHOU, Y. Rx: Treating Bugs As Allergies — A Safe Method to Survive Software Failure. In *Proceedings of the 20th Symposium on Operating System Principles (SOSP'05)* (Brighton, United Kingdom, Oct. 2005).

[45] RINARD, M., CADAR, C., DUMITRAN, D., ROY, D. M., LEU, T., AND WILLIAM S. BEEBEE, J. Enhancing Server Availability and Security Through Failure-Oblivious Computing. In *Proceedings of the 6th USENIX Conference on Operating Systems Design and Implementation (OSDI'04)* (San Francisco, CA, USA, Dec. 2004).

[46] ROBBINS, J., KRISHNAN, K., ALLSPAW, J., AND LIMONCELLI, T. Resilience Engineering: Learning to Embrace Failure. *ACM Queue 10*, 9 (Sept. 2012), 1–9.

[47] SCHLINKER, B., KIM, H., CUI, T., KATZ-BASSETT, E., MADHYASTHA, H. V., CUNHA, I., QUINN, J., HASAN, S., LAPUKHOV, P., AND ZENG, H. 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, Aug. 2017).

[48] SCHWARZKOPF, M., KONWINSKI, A., ABD-EL-MALEK, M., AND WILKES, J. Omega: Flexible, Scalable Schedulers for Large Compute Clusters. In *Proceedings of the 8th ACM European Conference on Computer Systems (EuroSys'13)* (Prague, Czech Republic, Apr. 2013).

[49] SHERMAN, A., LISIECKI, P. A., BERKHEIMER, A., AND WEIN, J. 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, May 2005).

[50] SIGELMAN, B. H., BARROSO, L. A., BURROWS, M., STEPHENSON, P., PLAKAL, M., BEAVER, D., JASPAN, S., AND SHANBHAG, C. Dapper, a Large-Scale Distributed Systems Tracing Infrastructure. Tech. Rep. dapper-2010-1, Google, Inc., Apr. 2010.

[51] SOMMERMANN, D., AND FRINDELL, A. Introducing Proxygen, Facebook's C++ HTTP framework, Nov. 2014. https://code.facebook.com/posts/1503205539947302.

[52] 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 (SOSP'15)* (Monterey, CA, USA, Oct. 2015).

[53] TREYNOR, B., DAHLIN, M., RAU, V., AND BEYER, B. The Calculus of Service Availability. *Communications of the ACM (CACM) 60*, 9 (Sept. 2017), 42–47.

[54] TSEITLIN, A. The Antifragile Organization. *Communications of the ACM (CACM) 56*, 8 (August 2013), 40–44.

[55] VEERARAGHAVAN, K., MEZA, J., CHOU, D., KIM, W., MARGULIS, S., MICHELSON, S., NISHTALA, R., OBENSHAIN, D., PERELMAN, D., AND SONG, Y. J. 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, Nov. 2016).

[56] VERMA, A., PEDROSA, L., KORUPOLU, M., OPPENHEIMER, D., TUNE, E., AND WILKES, J. Large-Scale Cluster Management at Google with Borg. In *Proceedings of the 10th European Conference on Computer Systems (EuroSys'15)* (Bordeaux, France, Apr. 2015).

[57] XU, T., JIN, X., HUANG, P., ZHOU, Y., LU, S., JIN, L., AND PASUPATHY, S. 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, Nov. 2016).

[58] XU, T., ZHANG, J., HUANG, P., ZHENG, J., SHENG, T., YUAN, D., ZHOU, Y., AND PASUPATHY, S. Do Not Blame Users for Misconfigurations. In *Proceedings of the 24th Symposium on Operating Systems Principles (SOSP'13)* (Farmington, PA, USA, Nov. 2013).

[59] YAP, K.-K., MOTIWALA, M., RAHE, J., PADGETT, S., HOLLIMAN, M., BALDUS, G., HINES, M., KIM, T., NARAYANAN, A., JAIN, A., LIN, V., RICE, C., ROGAN, B., SINGH, A., TANAKA, B., VERMA, M., SOOD, P., TARIQ, M., TIERNEY, M., TRUMIC, D., VALANCIUS, V., YING, C., KALLAHALLA, M., KOLEY, B., AND VAHDAT, A. 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, Aug. 2017).

[60] YIN, Z., YUAN, D., ZHOU, Y., PASUPATHY, S., AND BAIRAVASUNDARAM, L. N. How Do Fixes Become Bugs? – A Comprehensive Characteristic Study on Incorrect Fixes in Commercial and Open Source Operating Systems. In *Proceedings of the 19th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE'11)* (Szeged, Hungary, Sept. 2011).

[61] YUAN, D., LUO, Y., ZHUANG, X., RODRIGUES, G., ZHAO, X., ZHANG, Y., JAIN, P. U., AND STUMM, M. Simple Testing Can Prevent Most Critical Failures: An Analysis of Production Failures in Distributed Data-intensive Systems. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI'14)* (Broomfield, CO, USA, Oct. 2014).

[62] YUAN, D., PARK, S., HUANG, P., LIU, Y., LEE, M. M., TANG, X., ZHOU, Y., AND SAVAGE, S. Be Conservative: Enhancing Failure Diagnosis with Proactive Logging. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation (OSDI'12)* (Hollywood, CA, USA, Oct. 2012).

[63] ZHANG, Y., MAKAROV, S., REN, X., LION, D., AND YUAN, D. Pensieve: Non-Intrusive Failure Reproduction for Distributed Systems Using the Event Chaining Approach. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP'17)* (Shanghai, China, Oct. 2017).