Frisbee: A Suite for Benchmarking Systems Recovery

Fotis Nikolaidis

Institute of Computer Science,
FORTH
Heraklion, Greece
fnikol@ics.forth.gr

Manolis Marazakis

Institute of Computer Science,
FORTH)
Heraklion, Greece
maraz@ics.forth.gr

Antony Chazapis

Institute of Computer Science,
FORTH
Heraklion, Greece
chazapis@ics.forth.gr

Angelos Bilas

Institute of Computer Science,
FORTH)
Heraklion, Greece
bilas@ics.forth.gr

Abstract

With failures being unavoidable, a system's ability to recover from failures quickly is a critical factor in the overall availability of the system. Although many systems exhibit self-healing properties, their behavior in the presence of failures is poorly understood. This is primarily due to the shortcomings of existing benchmarks, which cannot generate failures. For a more accurate systems evaluation, we argue that it is essential to create new suites that treat failures as first-class citizens. We present *Frisbee*, a benchmark suite and evaluation methodology for comparing the recovery behavior of highly available systems. *Frisbee* is built for the Kubernetes environment, leveraging several valuable tools in its stack, including Chaos tools for fault injection, Prometheus for distributed monitoring, and Grafana for visualization. We discuss a set of design requirements and present an initial prototype that makes *faultloads* as easy to run and characterize as traditional performance workloads. Furthermore, we define a core set of failure patterns against which systems can be compared.

1 Introduction

With the complexity of emerging systems rapidly increasing, the rectification of faults becomes a burdensome, labor-intensive, and error-prone task. These problems have sparked research in self-healing techniques that aim to take human out of the failure-recovery loop and reduce the recovery

time from human timescales to the significantly faster machine timescales [8, 27]. Self-healing is an elaborate process comprising: (a) data redundancy; (b) timely detection of system misbehavior [12]; (c) diagnosing the failure root cause; (d) remedy strategies to repair or replace faulty components; (e) prevent recurring failures [18].

However, research in this area is hampered by the lack of suitable benchmarks that can characterize remediation techniques (e.g. reactive vs. preventive), identify suboptimal (i.e. less effective) approaches, and compare the recoverability of different systems in the presence of failures. While existing benchmarks, such as YCSB [5], can help assess the system's performance during recovery, they are generally not built to trigger the faults themselves – this is left to the system evaluator. Contrarily, Chaos Engineering toolkits [3, 23] that provide fault injection (e.g. randomly triggering faults) focus on functional testing (i.e. can the system recover ?) rather than the performance and recovery characteristics of the evaluated system. Moreover, the design and evaluation of Chaos experiments is often based on the practitioner's experience and intuition rather than the systemic approach required by benchmarks.

Unfortunately, the fault space can be so extensive that it is impractical to measure each fault's repercussions. Moreover, today's systems run so many distributed and asynchronous tasks that it is almost impossible to achieve identical conditions across different runs. The challenge of automating the pipeline and making the experiments scalable, portable, and repeatable exacerbates the problem of reproducibility. All these sources of variance may lead to unfair comparisons and invalid conclusions [1].

In this paper we present a novel benchmark designed to characterize and compare systems with respect to failure handling. Our benchmark, called *Frisbee*, treats failures as first-class citizens and allows us to combine traditional client-based benchmark workloads with *faultloads* on system servers [6]. Featuring a microservice design, *Frisbee* can dynamically alter the testing environment at runtime, allowing us to create self-modifying experiments with arbitrary

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

HAOC'21, April 26, 2021, Online, United Kingdom

© 2021 Association for Computing Machinery.

ACM ISBN 978-1-4503-8336-3/21/04...\$15.00

<https://doi.org/10.1145/3447851.3458738>

topologies and failure modes. We have built a Kubernetes-based prototype, leveraging common stacks for failure injection (Chaos-Mesh [23]), service discovery (Kubernetes Services), distributed monitoring (Prometheus [26]), and visualization (Grafana [10]). Notably, this containerized approach enables seamless experimentation with a broad range of systems and application benchmarks.

Furthermore, we propose a core classification of failure patterns against which systems can be compared. We base our model on a largely overlooked trait: systems are designed to handle specific faults in a consistent manner [13]. For example, a fault-tolerant system will start a rebalancing process regardless if packets are dropped due to a node crash, increased load on the node, or a network partition that makes the node unreachable.

We believe that the systems community can benefit from *Frisbee* in many different ways. First, its automation pipelines would help researchers study what-if scenarios and choose the most appropriate algorithms while developing new systems or resiliency mechanisms. Second, the standardization of *faultloads* would help experimenting with prescribed fault occurrences to compare the continuity of different systems in the presence of failures. Lastly, its extensible design allows operators to create custom *faultloads* and understand how a system would behave if deployed on various real-world environments (e.g., IoT, Edge, Cloud).

The scientific contributions of this paper are as follows:

- (a) Design of a fully-automated benchmarking suite for evaluating systems in the presence of failures;
- (b) Fault classification based on their immediate effect on the system state;
- (c) An extensive discussion of related work and preliminary results from an early prototype.

The remainder of the paper is organized as follows: Section 2 explains the need for recovery benchmarks and how our approach complements related work, Section 3 introduces our proposed failure classification model, Section 4 discusses the design of the benchmark suite. Section 5 concludes the paper and briefly presents future goals.

2 Motivation and Related Work

Systems testing and benchmarking is a well-established and prolific field. However, our literature review on availability benchmarks revealed an important finding. Judging by the number of publications, availability was a hot topic until the dawn of Cloud computing. From 2006 onward, systems performance and scalability were extensively studied, while availability received much less attention. This finding also aligns with the fact that performance and scalability benchmarks exist in abundance, whereas availability benchmarks are rare. Recently though, there was an unexpected turn of events. With the rise of the Internet of Things (IoT) and Edge computing, an increasing number of system services are

(a) Conceptual recovery analysis

(b) iperf Recovery

(c) TiKV Recovery

Figure 1. Example of recovery analysis

moving from the reliable Cloud data-centers towards less reliable infrastructures, closer to the end-users. In this context, availability research is once again gaining popularity.

2.1 The need for recovery benchmarks

System availability is a metric that measures the probability of receiving service when requested [22, 25]. Commonly stated by the number of 9s, availability depends on three properties: rate of failures, resiliency to failures, and recovery [7, 31, 35]. Although trace analysis is frequently used to study the failure patterns of a specific infrastructure [9, 32], it is challenging to compile a generic set of representative failures since different environments have different hardware, and therefore different, unique failure modes. In this regard, the focus is on ‘what-if’ scenarios designed to study the system’s behavior in the presence of various failure patterns. A resilient system strives to minimize exposure to failures, and if a failure happens, the system reacts to resist service interruption. For example, a highly available system can redirect traffic to a replicated node if a primary node should fail. Recovery defines the system’s ability to regain normative operation levels after an interruption or outage occurs. Resiliency metrics may include the magnitude of deviation from the nominative performance levels. Recovery metrics may include (a) the sensitivity in detecting interruptions, (b) the time it takes to regain nominal performance levels, (c) and the completeness of recovery in a

finite amount of time. To better understand the difference between resiliency and recovery, we use *Frisbee* to study how a failing node affects the performance of the iperf benchmark and the TiKV [24] replicated key/value store (Figure 1). In this scenario, borrowed from a previous study [19], there is one client and two servers, with one of the servers being ungracefully restarted. The first observation is that iperf does not tolerate failures, thus any failure can lead directly to an outage. In contrast, TiKV is fault-tolerant and allows clients to run uninterrupted, albeit with degraded performance. The second and more important observation is that iperf regains its previous performance level immediately after the failure is repaired, whereas TiKV never fully recovers. Further experiments with the Redis key/value store showed that both systems could fully recover the tail latency, while the throughput (in terms of Queries per second) remained low. Regardless of the specific reason, this example demonstrates the need to integrate failures in the systems evaluation methodology.

2.2 Chaos Engineering

Chaos Engineering is a production testing methodology pioneered by Netflix with the goal to build confidence in their geographically distributed video caches to withstand turbulent and unexpected conditions. The two key features of Chaos engineering [3, 11, 34] are to: (a) form a hypothesis about the expected behavior of the system in the presence of a given failure, (b) and perform the smallest possible experiment on production instances.

For example, Netflix hypothesizes that its platform can withstand arbitrarily failing services. To investigate this hypothesis, a ‘chaos experiment’ consists of randomly shutting down a virtual machine and checking that this perturbation has no significant influence on the main business metric of Netflix: the number of served video streams per second. Netflix assumes that confidence in past experiments’ results decreases as the system evolves, and has developed the ND-Bench [20] microbenchmark to automate the execution of long-running (infinite horizon) experiments with variable workloads, while ChaosMonkey (the Chaos suite of Netflix) causes random failures on the production services. Our vision with *Frisbee* is to develop a lab-scale equivalent of the Netflix ecosystem that provide researchers with a systematic way of studying complex experiments, without having to resort to scripting and adhoc solutions. To do so, *Frisbee* takes advantage of the Kubernetes environment, and enriches it with capabilities for fault injection, distributed monitoring, and dynamic workloads described as directed acyclic graphs.

3 Faultload

3.1 Faultload Definition

Performance benchmarks generate workloads and measure a system’s performance under these workloads to analyze the

system’s behavior and compare it to others. Recovery benchmarks may work in the same way [17]. Here the workload contains a set of faults, such as ungraceful node shutdowns, network partitions, disk corruptions. This type of ‘fault workload’ is called a *faultload* [6].

Like workloads, we can divide *faultloads* into two classes: macro and micro. Macro *faultloads* generate a complex mix of faults that represent adverse operating conditions on various classes of real-world environments (e.g., IoT, Edge, Cloud). Even though we have configured the *faultloads* with empirical values that make sense, building a macro *faultloads* require knowing the various failure probabilities for the given environment, which is out of the scope of *Frisbee*. Instead, *Frisbee* provides a pluggable API through which operators can integrate their custom failure patterns, either by contributing new *faultload* methods (implementing the *faultload* API), or by replacing the default values of an existing method with more realistic parameters. Micro *faultloads* generate a set of relatively simple faults that target a specific component of the system. As it is impractical to consider and evaluate all the possible faults, we propose a set of manageable classes that maintain essential properties required to standardize a benchmark. Our proposed classification adheres to the following properties:

- (a) be independent of any particular system,
- (b) be portable to every system,
- (c) allow fair comparison between architecturally different systems, and
- (d) be detailed but easy to understand.

3.2 Faultload Classification

The most apparent classification of *faultloads* is a two-class system based on a system’s availability: system available and system unavailable [2]. While this approach has the obvious advantage of having a direct relationship with interruptions perceived by the end-user, only a portion of fault events will result in prolonged outages. Most modern systems have varying degrees of in-built redundancy and fault tolerance mechanisms that can gracefully handle faults before they lead to an outage. With that in mind, we define fault events as falling into one of the following classes, which we summarize in Table 1:

- Class 1 (Non-recoverable) Fault: A permanent fault to a non-redundant component that causes a system outage and forces the system to remain unavailable. For example, a network partition to the majority of nodes will either bring the system to a halt or will force the system to run in a highly degraded mode.
- Class 2 (Recoverable) Fault: A permanent fault to a redundant component that causes an outage, but the fault can be circumvented to allow the system to return to service. For example, a crashed node in a replicated distributed system will cause some disturbance to the

Faultload	Recoverable	Outage	Duration	Recurring	Example
Non-Recoverable	No	Yes	Perm.	No	Network Partition
Recoverable	Yes	Yes	Perm.	No	Failing Node
Transparent	Yes	No	Perm.	No	Secondary Replica
Transient	Yes	No	Temp.	No	Load Spike
Intermittent	Yes	No	Temp.	Yes	Load Spike(s)

Table 1. Summary of *faultloads*

connected clients, but the fault can be circumvented by blacklisting the node and rebalance the load.

- Class 3 (Transparent) Fault: A permanent fault to a redundant component that does not cause a system outage. For example, a crashed disk on a RAID-1 configuration or a corrupted chunk in an erasure-coded system will most likely go unnoticed.
- Class 4 (Transient) Fault: A temporal fault that appears once and does not cause a system outage. An example of this fault is a spike on the network, disk, or CPU.
- Class 5 (Intermittent) Fault: A temporal fault that repeatedly appears and disappears without causing a system outage. Examples of this failure are recurring spikes on the network, disk, or CPU.

To understand the difference between Transient and Intermittent faults, consider the case of a *Circuit Breaker* design pattern aiming to detect failures and encapsulate the logic of preventing a failure from constantly recurring. The circuit breaker prevents the application from contacting a node known to cause timeouts (e.g., facing scheduled maintenance or temporal spikes in the load), thus saving communication overheads. In this context, by distinguishing the Transient class from the Intermittent class, we can separately evaluate the circuit breaker’s effectiveness and sensitivity.

A more refined categorization would also include crash failures, fail-stop failures, and propagation failures. Crash and fail-stop failures are detected almost instantaneously and do not corrupt the application state. In contrast, propagation may corrupt the state silently before their side-effect is detected. However, our focus with this work is on recovering from failures. Therefore, we assume that applications will detect faults and fail before generating incorrect output [4]. To ensure that systems can produce correct output in the presence of failures, one can use Jepsen [15] in the role of *Frisbee* clients, run a bunch of operations against the target system and verify the consistency of those operations.

4 Benchmark Design

4.1 Overview

As shown in Figure 2, we decouple the *Frisbee* runtime from the *Frisbee* controller. The runtime consists of lightweight containers in which operators can run images for databases,

Figure 2. *Frisbee*: Outline of architecture.

queuing systems, benchmarks, or any other executable program that runs in a distributed system. We classify these containers into system services, dependent services, and clients. The system services are server instances whose recovery mechanisms we want to investigate (e.g., key/value stores). Dependent services provide essential functions required by system services (e.g. Zookeeper for distributed coordination). The clients are performance benchmarks that continuously hammer system services with requests (e.g. YCSB).

The infrastructure interface layer is responsible for the creation, update, and termination of the containers. We presently use a Kubernetes-based driver for managing the containers. For operators that wish to run the benchmark in a more complex environment, including Kubernetes, Docker, vCenter, or any remote machine with ssh enabled, *Frisbee*’s powerful plugin model allows them to define their custom driver and run the benchmark without building code from scratch.

In the first step towards systems evaluation, the operator creates a YAML configuration that contains the initialization parameters for participating services and a workflow that expresses the runtime’s desired state as a function of time. The workflow contains inter-service dependencies (e.g. wait for execution or completion), failures that will happen to services (e.g. kill, partition, corruption), failure patterns (e.g. run a partition failure every 10 minutes), and callback functions that will be triggered upon certain events (e.g. push annotation to Grafana when a node has crashed).

After passing the YAML file to the *Frisbee* command-line utility, the *Frisbee* controller performs the necessary actions to maintain the runtime's present state in sync with the desired state. More specifically, the Deployment Manager launches the containerized instances in an isolated testing environment and employs distributed coordination to ensure that the system has reached a steady-state. Once the system is stable, the *faultload* executor takes over and drives failure injection into the running services, as prescribed by the given *faultload*. Both the deployment manager and *faultload* executor perform their tasks by making calls to the infrastructure abstraction layer.

While the clients run benchmarks against system services, both in the absence and presence of faults, the watcher interacts directly with the distributed clients and collects metrics through well-defined HTTP endpoints. We currently collect Queries per Second (QPS) and tail latency. Moreover, the watcher enriches the collected metrics with events from the deployment management and the *faultload* executor. This way, when operators use *Frisbee* for analyzing their systems, they will not have a simple visualization of the experiment's progress but an annotated representation of what exactly was happening at every different step.

Additionally, the watcher performs statistical analysis on the client metrics to decide if the collected metrics have converged on a stable value. If not, it notifies the deployment manager to restore the testing environment and asks the executor to rerun the *faultload*. This feedback loop continues until the metrics have converged or the number of retries has exceeded a maximum threshold. At the end of the experiment, the aggregated reports from all iterations are available via the watcher dashboard.

4.2 Steady State

Many systems operate through distinct phases; setup, steady-state, and (optionally) shutdown. These phases have different performance characteristics. Setup may start with clean caches and store everything in memory. Steady-state may balance memory and I/O usage proportional to connected clients or request rate. Shutdown may perform I/O to persist data on the disk. The I/O can be proportional to the amount of processed data throughout the steady-state. For a fair comparison between systems, we need to ensure that evaluated systems have reached a steady-state before starting the *faultload*. Luckily, Cloud benchmarks like YCSB are also divided into two phases: load and transaction. Therefore, *faultloads* remain pending, waiting for the clients to complete the load phase and start the transaction phase. However, this requires coordination among the distributed clients [21].

The event watcher has two parameters to control distributed coordination: a barrier-sync variable and the client coordination group's size for the phase. The barrier-sync is a structure used to track the number of clients entering or leaving a phase. The watcher uses a hierarchical namespace

for synchronization and, for each client, it creates a corresponding directory in its namespace. Clients join a phase by contacting the watcher at the `/:phase/join/:clientID` and then wait for `/:phase/ready` to become True; if not, the client blocks and retries after a timeout. In parallel, the watcher creates a new entry with the client's identifier in the phase directory; phase barriers and the respective directories are created on the fly. If the total number of entries equals the group size, meaning all clients have joined the phase, the watcher changes the returned value of `/:phase/ready` to True. If not, it returns False, waiting for more clients to join. After clients have completed their tasks, they notify watcher about exiting the by calling `/:phase/done/:clientID`, and then join the next phase `/:next_phase/join/:clientID`.

4.3 Distributed Monitoring and Metrics

During the fault and recovery periods, the operator must observe the system performance and availability as a function of time. Using this model, operators can analyze the responsiveness of different design decisions against emerging failures and runtime conditions [19]. Fortunately, the need for monitoring is well understood, and there are numerous available solutions for funneling metrics from distributed applications into a single monitor pipeline. For our prototype, we use a combination of Prometheus [26], Grafana [10], and Telegraf [29]. Prometheus periodically collects metrics by scrapping HTTP endpoints from the distributed clients and store those metrics in an embedded database. Grafana is a front-end to Prometheus that provides statistical analysis tools, user-friendly dashboards, and alerts. It must be noted that the previously mentioned 'event watcher' refers collectively to the whole monitoring stack, which involves several Microservices. We follow this convention to keep Figure 2 comprehensive and straightforward.

A key design goal of *Frisbee* is to be application agnostic. However, most existing benchmarks do not expose their metrics directly to Prometheus, and it is also possible for their output formats to variate significantly. To address these challenges, clients run a Telegraf agent that converts the client's output into Prometheus metrics and exposes those metrics in HTTP endpoints which Prometheus scrapes asynchronously. Similarly, Telegraf agents running on the Kubernetes host node provide insights into the system's traffic, I/O patterns, and resource utilization. This information can be later used for comparing scalability factors (e.g. hotspots) and cost factors (e.g. I/O amplification).

4.4 Confidence Interval

Determinism is an essential property of an automated testing suite. Unfortunately, it is onerous to achieve identical conditions over different test runs [14, 16, 28]. Physical machines with different performance characteristics (e.g. disks, processors, or network interfaces), interference between co-located processes, or the execution of background tasks (e.g., garbage

collection, scrubbing), may lead to temporal variations that cause the obtained metrics to vary significantly.

To reduce the uncertainty caused by that variability, we need to run multiple test iterations, and at each iteration, to randomize the order of injected faults [1]. That being the case, the question is ‘how many iterations are needed?’. The short answer is ‘the minimum number to ensure that metrics have converged on a stable value’.

The long answer is that we need to perform some fundamental statistical analysis to calculate the confidence interval and the margin of error for the collected sample (of metrics). Suppose the error is below a given threshold. In that case, we are confident that the collected sample reflects what we would expect to find had we run infinite amounts of tests. Thus, no further tests are needed, as they would only increase the overhead without compensating for additional knowledge.

However, confidence issues are orthogonal to the *faultload*, and should not bog *faultload* developers. *Frisbee* provides a Benchmark function to wrap the *faultload* and transparently run the Benchmark functions several times. Our Benchmark function is quite simple, being very similar to a unit test.

```
func BenchmarkFaultload(b *frisbee.B) {
    for i := 0; i < b.N; i++ {
        Faultload()
    }
}
```

B is a struct passed to Benchmark functions to manage the number of iterations and the fault order in each iteration. Each benchmark is run three times for a minimum of 1-minute per run by default, with 5 percentage points being the expected margin of error. If the error is greater than 5 after three iterations, *Frisbee* keeps increasing the b.N in the sequence 1, 2, 4, 8, 16, and the *faultload* method runs again. If the error remains high after 16 iterations, we regard the system as highly unstable and incomparable. This hard threshold helps us avoid a situation where b.N keeps growing indefinitely, and the benchmark never completes.

4.5 Portability

Benchmark portability [30, 33] is a critical topic that until now remains largely overlooked. It is not uncommon to spot two different papers in the literature whose authors use the same benchmark, the same deployment, and yet find that the published result variate significantly. The reason typically has to do with the experimentation platforms they use. Although the experimentation platforms can be documented and compared qualitatively, it is much harder to achieve repeatable results across different infrastructures. Since important decisions depend on reliable experimental data, it is necessary to enable reproducible experiments and verifiable results. Using Kubernetes as an abstraction layer, it is now

possible to create a new generation of portable benchmarks. Some of the merits of this approach include:

- Physical resources are replaced by virtual counterparts whose capabilities are well specified and therefore recreated amongst different Kubernetes deployments.
- There are several Chaos Testing tools, like Chaos-Mesh, which we can use for fault injection. These tools typically consist of a high-level part for selecting fault parameters and a low-level part designed to cause (or simulate) faults at the targeted components.
- Experiments can seamlessly scale from a single workstation to clusters of commodity servers or even larger enterprise-grade servers. Equally, they may run on-premise or in the Cloud.

5 Conclusions

With the increasing popularity of IoT and Edge computing, availability benchmarks are becoming an impending need. We have presented *Frisbee*, a benchmarking suite designed to facilitate the study of systems behavior under the presence of failures. *Frisbee* is designed to harness the collective power of performance benchmarks, Chaos engineering, distributed monitoring, and visualization tools. With minimal human configuration, *Frisbee* automatically launches the target system in an isolated environment and runs performance benchmarks against the system services both in the absence and in the presence of faults. For an apples-to-apples comparison between different systems, we reduce the fault-space by proposing a novel classification of failure patterns against which systems can be compared. To ensure reproducible results, *Frisbee* automatically runs as many tests as needed to ensure that collected metrics converge to a stable state. At the end of the experiment series, the collected metrics are available in a user-friendly dashboard from which operators can retrieve the aggregated reports or drill-into intermediate results.

Albeit being in the early stages of development, the design of *Frisbee* is well-grounded and mainly focused on extensibility. Built on top of Kubernetes, *Frisbee* has no constraints regarding what systems to run on or against which workloads. Henceforth, our primary focus is to use *Frisbee* to compare a set of high-available systems in adverse scenarios and collect failure traces from production environments to enrich our proposed classification further. We hope to foster the development of additional failure patterns that represent real-world environments (e.g. IoT, Edge, Cloud) by making our framework available as open source.

Acknowledgement. This work is supported by the European Commission within the scope of the ETHER (H2020-MSCA-IF-2019 / Grant Agreement ID: 894204) project.

References

- [1] Ali Abedi and Tim Brecht. 2017. Conducting repeatable experiments in highly variable cloud computing environments. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering*. 287–292.
- [2] Algirdas Avizienis, Jean-Claude Laprie, Brian Randell, et al. 2001. *Fundamental concepts of dependability*. University of Newcastle upon Tyne, Computing Science.
- [3] Ali Basiri, Niosha Behnam, Ruud De Rooij, Lorin Hochstein, Luke Kosewski, Justin Reynolds, and Casey Rosenthal. 2016. Chaos engineering. *IEEE Software* 33, 3 (2016), 35–41.
- [4] Andera Bondavalli and Luca Simoncini. 1990. Failure classification with respect to detection. In *[1990] Proceedings. Second IEEE Workshop on Future Trends of Distributed Computing Systems*. IEEE, 47–53.
- [5] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM symposium on Cloud computing*. 143–154.
- [6] Joao Duraes and Henrique Madeira. 2004. Generic Faultloads Based on Software Faults for Dependability Benchmarking. In *Proceedings of the 2004 International Conference on Dependable Systems and Networks (DSN '04)*. IEEE Computer Society, USA, 285.
- [7] Elsayed Elsayed. 2021. *Reliability engineering*. Wiley, Hoboken, NJ.
- [8] Debanjan Ghosh, Raj Sharman, H. Raghav Rao, and Shambhu Upadhyaya. 2007. Self-healing systems — survey and synthesis. *Decision Support Systems* 42, 4 (2007), 2164 – 2185. <https://doi.org/10.1016/j.dss.2006.06.011> Decision Support Systems in Emerging Economies.
- [9] Ramesh Govindan, Ina Minei, Mahesh Kallahalla, Bikash Koley, and Amin Vahdat. 2016. Evolve or die: High-availability design principles drawn from googles network infrastructure. In *Proceedings of the 2016 ACM SIGCOMM Conference*. 58–72.
- [10] grafana. 2020. The open and composable observability and data visualization platform. <https://github.com/grafana/grafana>
- [11] Haryadi S Gunawi, Thanh Do, Joseph M Hellerstein, Ion Stoica, Dhruba Borthakur, and Jesse Robbins. 2011. Failure as a service (faas): A cloud service for large-scale, online failure drills. *University of California, Berkeley, Berkeley* 3 (2011).
- [12] Trinabh Gupta, Joshua B. Leners, Marcos K. Aguilera, and Michael Walfish. 2013. Improving Availability in Distributed Systems with Failure Informers. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*. USENIX Association, Lombard, IL, 427–441. <https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/leners>
- [13] Walter L Heimerdinger and Charles B Weinstock. 1992. *A conceptual framework for system fault tolerance*. Technical Report. CARNEGIE-MELLON UNIV PITTSBURGH PA SOFTWARE ENGINEERING INST.
- [14] Nikolas Herbst, André Bauer, Samuel Kounev, Giorgos Oikonomou, Erwin Van Eyk, George Kousiouris, Athanasia Evangelinou, Rouven Krebs, Tim Brecht, Cristina L Abad, et al. 2018. Quantifying cloud performance and dependability: Taxonomy, metric design, and emerging challenges. *ACM Transactions on Modeling and Performance Evaluation of Computing Systems (ToMPECS)* 3, 4 (2018), 1–36.
- [15] Jepsen. 2020. A framework for distributed systems verification, with fault injection. <https://github.com/jepsen-io/jepsen>
- [16] Philipp Leitner and Jürgen Cito. 2016. Patterns in the chaos—a study of performance variation and predictability in public iaas clouds. *ACM Transactions on Internet Technology (TOIT)* 16, 3 (2016), 1–23.
- [17] Henrique Madeira and Philip Koopman. 2001. Dependability Benchmarking: making choices in an n-dimensional problem space. (2001).
- [18] Fabrizio Montesi and Janine Weber. 2016. Circuit Breakers, Discovery, and API Gateways in Microservices. *arXiv:1609.05830 [cs.SE]*
- [19] Kiran Nagaraja, Xiaoyan Li, Ricardo Bianchini, Richard P. Martin, and Thu D. Nguyen. 2003. Using Fault Injection and Modeling to Evaluate the Performability of Cluster-Based Services. In *Proceedings of the 4th Conference on USENIX Symposium on Internet Technologies and Systems - Volume 4 (Seattle, WA) (USITS'03)*. USENIX Association, USA, 2.
- [20] Ioannis Papapanagiotou and Vinay Chella. 2018. Ndbench: Benchmarking microservices at scale. *arXiv preprint arXiv:1807.10792* (2018).
- [21] Swapnil Patil, Milo Polte, Kai Ren, Wittawat Tantisiriroj, Lin Xiao, Julio López, Garth Gibson, Adam Fuchs, and Billie Rinaldi. 2011. YCSB++: Benchmarking and Performance Debugging Advanced Features in Scalable Table Stores. In *Proceedings of the 2nd ACM Symposium on Cloud Computing (Cascas, Portugal) (SOCC '11)*. Association for Computing Machinery, New York, NY, USA, Article 9, 14 pages. <https://doi.org/10.1145/2038916.2038925>
- [22] David A. Patterson. 2002. A Simple Way to Estimate the Cost of Downtime. In *Proceedings of the 16th USENIX Conference on System Administration (Philadelphia, PA) (LISA '02)*. USENIX Association, USA, 185–188.
- [23] PingCAP. 2020. *A Chaos Engineering Platform for Kubernetes*. <https://github.com/chaos-mesh/chaos-mesh>
- [24] PingCAP. 2020. TiKV: A distributed transactional key-value database. <https://tikv.org/>
- [25] HBM Prensca. 2007. *Availability and the Different Ways to Calculate It*. <https://www.weibull.com/hotwire/issue79/rebasics79.htm>
- [26] prometheus. 2020. The Prometheus monitoring system and time series database. <https://github.com/prometheus/prometheus>
- [27] Harald Psailer and Schahram Dustdar. 2011. A Survey on Self-Healing Systems: Approaches and Systems. *Computing* 91, 1 (Jan. 2011), 43–73. <https://doi.org/10.1007/s00607-010-0107-y>
- [28] Jörg Schad, Jens Dittrich, and Jorge-Arnulfo Quiané-Ruiz. 2010. Run-time measurements in the cloud: observing, analyzing, and reducing variance. *Proceedings of the VLDB Endowment* 3, 1-2 (2010), 460–471.
- [29] telegraf. 2020. The plugin-driven server agent for collecting & reporting metrics. <https://github.com/influxdata/telegraf>
- [30] Timothy K Tsai, Ravishankar K Iyer, and Doug Jewitt. 1996. An approach towards benchmarking of fault-tolerant commercial systems. In *Proceedings of Annual Symposium on Fault Tolerant Computing*. IEEE, 314–323.
- [31] Enrique Vargas and Sun BluePrints. 2000. High availability fundamentals. *Sun Blueprints series* (2000), 1–7.
- [32] G. Wang, L. Zhang, and W. Xu. 2017. What Can We Learn from Four Years of Data Center Hardware Failures?. In *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE Computer Society, Los Alamitos, CA, USA, 25–36. <https://doi.org/10.1109/DSN.2017.26>
- [33] Don Wilson, Brendan Murphy, and Lisa Spainhower. 2002. Progress on defining standardized classes for comparing the dependability of computer systems. In *Proc. DSN 2002 Workshop on Dependability Benchmarking*. IEEE Computer Society, Los Alamitos, CA, USA.
- [34] Long Zhang, Brice Morin, Philipp Haller, Benoit Baudry, and Martin Monperrus. 2019. A chaos engineering system for live analysis and falsification of exception-handling in the JVM. *IEEE Transactions on Software Engineering* (2019).
- [35] Ji Zhu, James Mauro, and Ira Pramanick. 2002. *R-Cubed (R3): Rate, Robustness, and Recovery - an Availability Benchmark Framework*. Technical Report. USA.