Event-Driven End-to-End Testing For Containerized Applications

Fotis Nikolaidis[1], Antony Chazapis[1],
Manolis Marazakis[1], and Angelos Bilas[1,2]

[1] Institute of Computer Science, FORTH
[2] Computer Science Department, University of Crete
{fnikol, chazapis, maraz, bilas}@ics.forth.gr

Abstract. With the complexity of emerging systems rapidly multiplying, it is important to evolve our testing infrastructures required to better understand how our distributed systems deal with scaling, failover and fatal tolerance. Compared to random testing the test of "deep" failure paths requires different methods for deriving test cases and for running the test. This paper introduces *Frisbee*, a platform for the automated testing of distributed applications over Kubernetes. *Frisbee* leverages static and dynamic runtime instrumentation to spin-up the dependency stack and perform execution-driven testing actions, while automating the collection of performance metrics and the assertion of system's behavior. This technique enables the controlled injection of realistic software faults while the target system executes, ensuring a predetermined fault load distribution throughout the experiment, regardless of the particular system or workload. Our evaluation demonstrates that *Frisbee* significantly enhances the precision and controllability of prior tools with only modest memory and performance overhead during fault-free execution.

Keywords: Cloud-Native testing · systems benchmarking · Event-driven Chaos testing

1 Introduction

In 2012, Netflix introduced the concept of Chaos Engineering to detect potential system failures before they resulted in outages. However, their initial approach of random testing, which involved subjecting complex cloud-deployed systems to negative behavior, had limitations. While it could reveal shallow bugs caused by independent faults, it was unlikely to expose "deep" failures resulting from combinations of different components and types of faults. These untested scenarios could be a problematic aspect of the system.

As Chaos Engineering has evolved, researchers have moved towards mimicking worst-case failure scenarios, allowing the system to be driven into unlikely but severe corner cases through intentional, measured, known failures. However, testing these "deep" paths can be challenging, given the inherent complexity and concurrency of distributed systems. Triggering faults using an external timer

This is a preprint version of the publication:
Nikolaidis, F., Chazapis, A., Marazakis, M., Bilas, A. (2023). Event-Driven Chaos Testing for Containerized Applications. In: Bienz, A., Weiland, M., Baboulin, M., Kruse, C. (eds) High Performance Computing. ISC High Performance 2023. Lecture Notes in Computer Science, vol 13999. Springer, Cham. https://doi.org/10.1007/978-3-031-40843-4_12

without accounting for the system's state is difficult, especially when attempting to inject faults at precise runtime conditions.

This paper introduces *Frisbee*, a Kubernetes platform that uses execution-driven fault injection, a new technique for dependable general-purpose Chaos Engineering experiments. Unlike existing tools, *Frisbee* injects a controlled and predetermined faultload distribution as the system executes at runtime, thanks to a combination of containerized workflows, Chaos toolkits, and a well-distributed monitoring baseline that provides insights into the system's state. As a result, *Frisbee* can deliver precise, controllable, and observable fault injection experiments with negligible impact on the system during normal execution.

The paper also introduces a comprehensive Chaos Engineering language that allows testers to create sophisticated Chaos scenarios by combining base scenarios without needing to write new code. Testers can define the entire dependency stack, bring the system into a steady state, schedule switching between fault-free and faulty execution, validate transitions for bad states or SLA violations, and use a broad range of containerized systems and application benchmarks. In this way, *Frisbee* provides a flexible and scalable solution for conducting reliable and reproducible Chaos Engineering experiments.

2 The *Frisbee* Framework

2.1 Features

Cloud application developers face numerous challenges when exploring the performance and reliability of their applications under different operational conditions. *Frisbee* offers several features to help developers overcome these challenges:
Extensibile and Portable Testbed The *Frisbee* framework is designed to be easily extensible, portable, and practical. The core features of *Frisbee* operate in Kubernetes to eliminate infrastructure-specific constraints, ensuring similar environments for dev, test, and production. This means that *Frisbee* scenarios are reproducible across a range of environments, including local workstations (e.g., minikube, microk8s), bare-metal cluster setups (e.g., k8s), and Cloud deployments (e.g., EKS).
Event-Driven Initialization, Workload and Faultload *Frisbee* provides direct access to the runtime environment to ensure controllability and precision of executed actions. This means that testers can build execution-driven scenarios, allowing them to spin up the dependency stack of the System Under Test (SUT) in a deterministic manner, build complex workloads with dynamically changing request patterns, and surgically inject faults into specific locations without introducing spurious faults that may compromise the validity of the results.
Failure-Aware Semantics Managing the SUT and the failure injection from within the same workflow allows testers to distinguish between an unexpected service failure (i.e., due to a bug in the application's code) and an expected service failure (i.e., part of a Chaos experiment). This is a highly required feature in resiliency testing [1]. In the first case, the test should abort immediately,

whereas in the second case, the fault does not constitute a culpable failure, and the test should continue.

Contextualized Visualizations: *Frisbee* provides additional interpretation and contextual data to help testers visually correlate the observed behavior with a root event. For example, an annotation marking a network fault's injection could help explain an otherwise enigmatic drop in the system's throughput. To achieve this, when *Frisbee* performs an action (e.g., create a service, inject a fault), it interacts with Grafana to annotate the action's beginning and ending.

Hierarchical Assertions *Frisbee* promotes the decomposition of complex properties into simpler assertions that are provable at the action level. This is important because testing workflows are defined hierarchically (e.g., clusters managing hundreds of services), making it natural to partition large verification problems into a set of properties associated locally with the entities of the workflow hierarchy [2].

2.2 System Overview

At a high level, *Frisbee* interfaces with Kubernetes to run the experiment, monitor system state and application metrics, and compare them against assertions. To do so, *Frisbee* implements a set of Actions that cover the full spectrum of operations needed to test a distributed system. Each action is defined on a separate CRD and is managed by a dedicated controller. The overall architecture is shown in Figure 1.

The experiment starts with the user specifying the testing resources in a YAML file. The most important resources are the templates for the System Under Test (SUT) and the testing scenario that will drive the testing process. Within the SUT template, the user specifies the properties of the SUT (e.g, listening ports) and the benchmarks to evaluate the SUT. Within the scenario, the user specifies the sequence to bootstrap the SUT, changes in the environment that will happen throughout the experiment (testing actions), as well as conditions that should be met in order for the test to be successful. Actions may include creating instances from SUT templates, deleting running instances, executing scripts in running instances, injecting or revoking faults, and more.

After the scenario is ready for deployment, the user submits it to Kubernetes API via either the standard kubectl tool or the *Frisbee* CLI. Subsequently, the Kubernetes API invokes the *Frisbee* admission controller, which performs a preliminary validation of the scenario to detect malformed dependencies, expression errors, and ensure that the scenario does not contain infinite loops which cause the scenario to run forver, thus preventing other test-cases from execution. After validation from the admission controller, the scenario is being forward to the Scenario controller for execution. In general, listed actions are executed immediately, unless the user has specified scheduling constraints or inter-action dependencies. In this case, the execution is postponed until the desired conditions are met. The conditions are described using *Frisbee* expressions, which can be time-driven, state-driven, metrics-driven, or tag-events. These events are summarized on Table 1

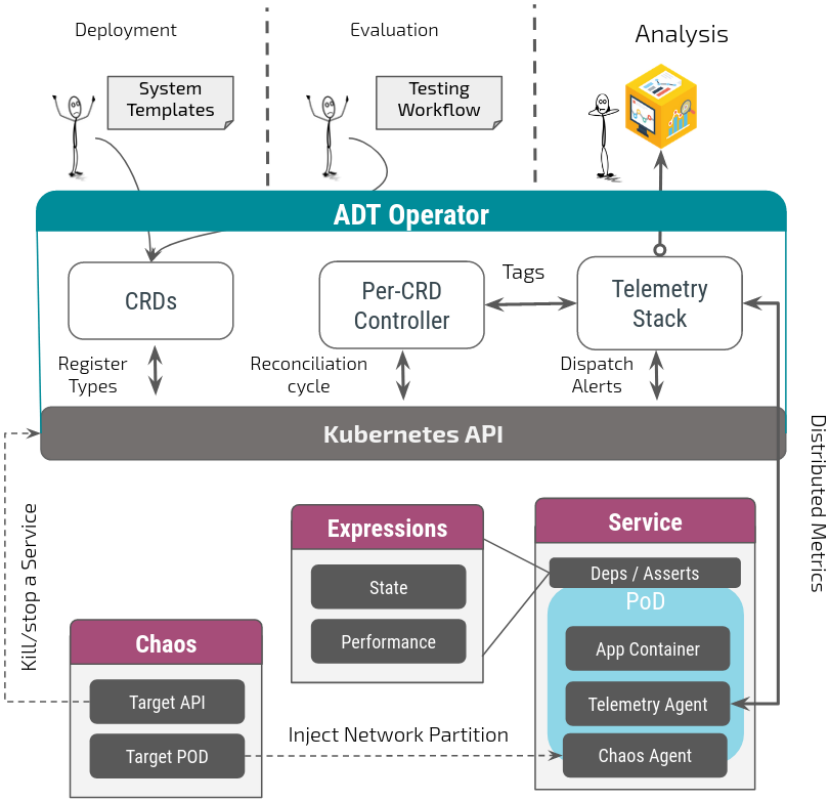


Fig. 1: *Frisbee* architecture. Given a template describing the system under test and a workflow describing the experiment, *Frisbee* interfaces with Kubernetes to run the experiment. Notice the loop among controllers, telemetry stack, and kubernetes API.

During scenario execution, *Frisbee* collects system and application metrics from distributed components using a monitoring sidecar attached to the same pod as the target application [3,4]. Prometheus scrapes the metrics periodically and stores them in a time-series database for analysis using Grafana, which provides aggregation functions, real-time visualizations, and alerting capabilities. Reporting is emphasized, and the controller pushes a descriptive annotation to Grafana when creating or deleting an object to provide context for visually correlating observed behavior with a root event. The controller maintains a bidirectional link with Grafana, capturing alerts raised by it, which can be used in evaluating scheduling policies and testing assertions.

Finally, it worth mentioning that *Frisbee* is shipped with a cli which simplifies testing-related tasks, such as running tests on isolated namespaces, providing an overview of the test status, facilitate inspection of the testing environment, and save testing results for further processing.

Event	**Description**
Time-driven	Fired after an elapsed time measured by the controller.
State-driven	Fired by the Kubernetes API when managed objects have state changes, errors, or other messages that should be broadcast to the system.
Metrics-driven	Fired by the telemetry stack when the outcome of statistical analysis on the collected metrics matches a given rule.
Tag-events	Fired when one controller passes contextual information to another controller.

Table 1: Events used to drive assertions, conditional loops, and other places involving execution-driven knowledge.

2.3 Scenario Modeling

An *Frisbee* scenario describes a set of dependent actions that collectively describe the testing process. The scenario defines three important properties: 1. a list of actions that drive the testing process, 2. the preconditions of the runtime before each action, and 3. the desired state of the runtime after each action.

Actions covers a full range of operations required to test a distributed system, and consists of five comprehensive and fine-grained abstractions: services, clusters, faults, calls, and checkpoints. Descriptions are in Table 2. Under the hood, actions invoke templates that provide solid definitions of the SUT. This strategy allows us to create a library of frequently-used specifications and use them to generate objects on-demand throughout the experiment. When called without parameters, templates generate services initialized with reasonable defaults. With parameters, templates generate the customized configuration by replacing the placeholders (denoted as ...) with the given input. Due to the lim-

ited space, we only provide the definition of a scenario 1.1, not the underlying templates.

Action	Description
Service	Standard representation of a containerized application.
Cluster	Abstraction for managing multiple services in a shared execution context.
Fault	Condition that disturbs a running service, network, or storage.
Call	Command executed by the controller to a running service.
Checkpoint	Store the status and the metrics of the SUT at the given time.

Table 2: Using just 5 basic primitives, *Frisbee* covers a full range of operations required to test a distributed system.

Code 1.1: Scenario demonstrating action with logical dependencies (depends), parameterization (inputs), addressing macros (.cluster), and injection of templates faults.

2.4 Extensibility and Integration with Chaos Controllers

Every action in *Frisbee* is represented by a different CRD and is managed by a separate controller, e.g, the *cluster* action will handled by the *Cluster controller*. Practically, this architecture builds a tree of domain-specific controllers with a top-level controller (the scenario) that dispatch requests based on the type of action. Controllers interact by submitting and receiving requests recorded in the Kubernetes API. In the previous example, the *Scenario controller* will create a *Cluster resource* to the Kubernetes API, that will be subsequently captured by the *Cluster controller*. After handling the request, the *Cluster controller* will modify the *Cluster resource* with the updated status, and the Kubernetes API will notify the *Scenario controller* about the change.

2.5 Manage Dependencies Between Actions

To manage the logical dependencies between actions, the *scenario* controller should inspect the status of the various actions (stored in the Kubernetes API) and, once it comes to the desired state, trigger the following action. This, however, requires the *Scenario* controller to know how to parse the various statuses and understand their semantics. One solution is to let the user specify parsing rules for each action and explicitly model the reaction to every possible outcome. Instead of this laborious task, we have set a common phase-field so that all objects managed by *Frisbee* can consistently communicate where they are in their lifecycle. The values and their meaning are tightly guarded, as shown in Table 3.

Phase	Description
Uninitialized	the request has been accepted by the Kubernetes API, but it is not yet dispatched to the *Frisbee* controller.
Pending	the request has been accepted by a *Frisbee* controller, but at least one of the constituent jobs has not been created.
Running	all constituent jobs of the request have been created, and at least one job has not been completed yet.
Success	all the constituent jobs of a request have been successfully completed.
Failed	at least one job of the request has terminated in a failure.

Table 3: Lifecycle Semantics. Logical Dependencies are built upon them.

2.6 Expected versus Unexpected Failures

Frisbee features a holistic approach to failure detection, propagation, and recovery [5], which provide testers the ability to distinguish between expected and unexpected failure. The basis of our mechanism is that almost all fault-injections in Kubernetes happen at a Pod level. Whenever the workflow controller encounters a Chaos action, it traverses the ownership semantics for finding the service responsible for the affected pod and places a *metadata.Chaos* tag on it. This tag describes the type of impending fault (e.g., partition, kill). It then applies the action. If the pod crashes, the failure is propagated to the service via the status updating mechanisms, making the service fail as well. When this happens, the service owner (i.e., cluster) can conclude whether the failure is expected or not by inspecting the service's metadata for the presence of a chaos tag. If there is no tag, the failure is unexpected, and the cluster fails immediately to cut losses and quickly try something else. If there is a tag, the failure is expected, and the behavior is tunable according to the clusters' semantics, e.g., the number of tolerated failures.

2.7 Extract State and Performance Metrics

Expressions in *Frisbee* can be state-driven or metrics-driven. **State Expressions** are used to describe dependencies on the lifecycle of a Kubernetes object (i.e when a service has failed). They consume state events fired by the Kubernetes API. For simplicity, we provide a set of aggregation functions that are callable from within the expressions, e.g., '$.state.failed() > 4$'. Expressions can also be combined into complex assertions via logical/arithmetic/string comparators.

Metrics Expressions are used to describe dependencies on the performance of a running service (i.e when percentile latency exceeds a given limit). They involve an initial step for setting an alert to Grafana and then checking if the alert is fired. For interoperability, we have adopted the Grafana syntax for writing alerts. For example, '*MAX() QUERY(metric, 1m, now) IS ABOVE(70000)*' roughly translates as 'raise an alert if the max value of the given metric for the period between not and 1 minute ago has been above 70000'.

The expressions are well-integrated into the *Frisbee* language and are reusable across assertion, conditional loops, and other places involving execution-driven knowledge. Additionally, the extracted output of the expressions can be stored using the *Checkpoint* action. As demonstrated in the Evaluation Section, the *Checkpoint* allows us to take snapshots and use this information in assertions to compare performance metrics before and after an action (i.e., network partition).

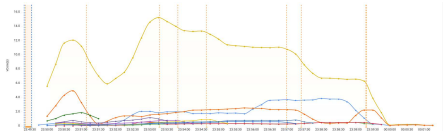
2.8 Scoped Assertions

Every resource managed by *Frisbee* (e.g., services, faults, clusters, cascade) has its own assertions. The main benefit is that we can isolate failed services while allowing the workflow components to synchronize their knowledge and correlate individual failures to top-level events [6, 7]. For state-based expressions, the assertions only consider jobs in the local scope – those created by the entity. This way, we prevent cross-references on jobs belonging to a different resource, thus saving the user from never-ending loops. Nonetheless, we cannot wholly exclude cross-references, given that metrics-driven expressions have access to the common telemetry stack. Therefore, we prevent cross-references to a job but not to a job's performance metrics. Regardless of this peculiar case, we still permit top-level assertions to use available assertions at lower hierarchy levels.

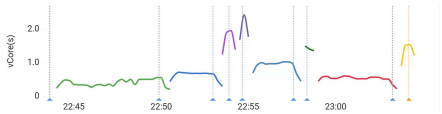
3 Evaluation

We evaluate *Frisbee* using the CockroachDB. The decision was based on the fact that CockroachDB have open-sourced their own tool (roachtest) for performing large-scale (multi-machine) automated tests, and have a large collection of testing scenarios that we replicate in *Frisbee*. The evaluation was conducted on a Kubernetes cluster with 4 server-grade nodes, and the default settings were used to configure CockroachDB and YCSB benchmarks. Bitnami containers Telegraf, Prometheus, and Grafana were used for monitoring, while Chaos-Mesh was used for fault injection [8].

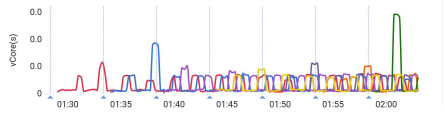
Load testing experiments were conducted using a sequence of YCSB workloads to evaluate the precision of *Frisbee*'s event-driven strategy. The scenario consisted of creating a cockroach cluster, preloading the server with keys, and running YCSB workloads A-F and E, while using different invocation strategies (Kubernetes Vanilla, Time-Driven, State-Driven) and collecting statistics on system utilization and application-specific metrics (number of Go routines). As shown in Figure 2, the vanilla Kubernetes policy was found to be unreliable due to its expectation of a final state and lack of logical dependencies. The time-driven policies is also found to be flaky. If the sleeping time is small, subsequent stages of the workflow will overlap, leading to erroneous results. If the sleeping time is large, there is a huge portion of idle time, that is translated to long-running experiments with increased time and energy costs. In contrast, state-driven execution was found to respect job ordering regardless of the number of ingested keys.



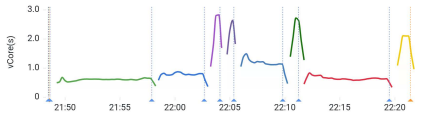
(a) Kubernetes Scheduling, 500 keys



(b) State-Driven Scheduling, 500 Keys



(c) Time-Driven Scheduling, 500 Keys



(d) State-Driven Scheduling, 1M keys

Fig. 2: Scheduling Policies for the standard sequence of YCSB workloads (A,B,C,F,D,E)

SSTable Corruption The purpose of this scenario is to test whether the storage layer (pebble) can detect corrupted SST files and fail fast [9]. The scenario involves provisioning a 3-node cluster, importing TPC-C data from the workload node, corrupting six random SST files on each node, starting each node, and verifying that each node panics. If the nodes do not panic, the TPC-C workload is run for up to 10 minutes on node 1, and the cluster is verified to panic. The experiment is aborted if the workload finished but no panic occurred. As can be seen in Figure 3, *Frisbee* can successfully detect the panicked service and abort the experiment.

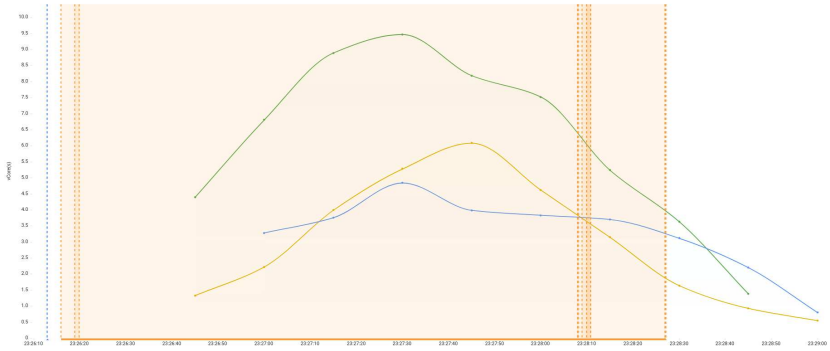


Fig. 3: SSTable corruption, with assertions

Goroutine Leak This scenario is adopted from the roachtest suite that tests for leaking goroutines after recovering from a network partition [10]. The scenario involves provisioning a 4-node cluster, importing TPC-C data, waiting for replication, running a workload, partitioning node 1 from the rest of the

nodes, and verifying that the maximum number of goroutines has not spiked after removing the network partition. As can be seen in Figure 4, *Frisbee* can successfully inject and repair a network partition fault. However, the number of goroutines after the partition have not spiked, and *Frisbee* correctly continues the execution of the test.

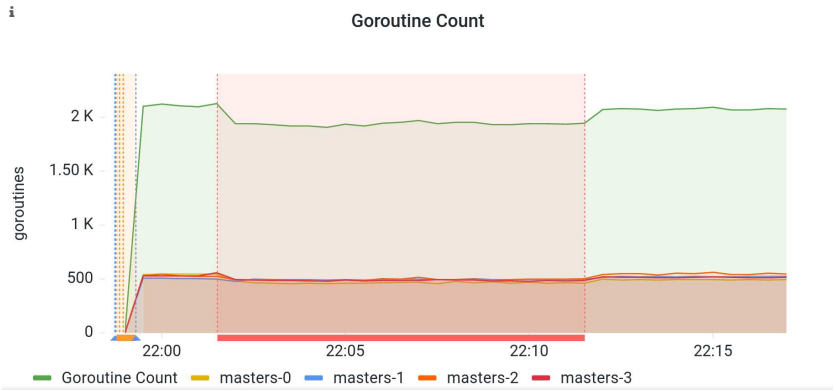


Fig. 4: Network Partition

4 Related Work

We compare the related work concerning these two rules: (i) automation of the testing process and (ii) creation of realistic testing environments.

Test source code: Frameworks like Ginkgo for Go, Robot for Python, or Serenity for Java, provide a rich and expressive Domain-specific Language (DSL) for writing test scenarios. These frameworks implement the scenarios using thin threads [11,12], each representing a single service, which makes them impractical for systems whose components are written in different languages.

Test processes in physical infrastructure: Another approach is to use Chef and Puppet, which provide a declarative way to automate the deployment and configuration of testing objects like servers and benchmarks. However, since these tools are designed for deployment and configuration, they are not well-suited for testing, which typically involves multiple intermediate steps with logical dependencies between them.

Test containers in Docker The rise of Docker led to a shift in the testing community towards using containers instead of virtual machines. Initially, containers were managed manually with scripts, which mixed the test case with the testing mechanism. Later, the Docker-native approach was to use the Docker-Compose language for running multi-container tests, as demonstrated by BenchPilot [13], Fogify [14], and IOTier [15]. However, Docker-Compose lacks assertions in the language, and it only works for single-node deployments.

Test containers natively in Kubernetes While some tools like Cilium and Testground create a small multi-node Kubernetes environment for testing features beyond a single host, this solution incurs significant state-management concerns, since the test suite runs externally to Kubernetes. Instead, KUTTL [16] and Iter8 [17] are Kubernetes-native testing tools, but they differ from our goals. KUTTL is a toolkit for testing a Kubernetes controller, and Iter8 aims at testing the progressive rollout of microservices.

5 Conclusion

Because our systems and system designs have evolved in such an unprecedented way, we must also evolve our testing methods to better understand how our systems perform under normal and adverse operating conditions. We believe the answer lies in building a new generation testing tools that can support truly precise, controllable, observable, and language-agnostic experiments. With *Frisbee*, our ultimate goal is to foster the development of a common Cloud-Native Testing Framework [18] for systems researchers and practitioners, in order to support dependable, reproducible, and comparable experiments in Chaos campaigns. The *Frisbee* and the experiments are made available open-source. The link is disclosed for blindness purposes.

Acknowledgement

We thankfully acknowledge the support of the European Commission and the Greek General Secretariat for Research and Innovation under the EuroHPC Programme through project EUPEX (GA-101033975). National contributions from the involved state members (including the Greek General Secretariat for Research and Innovation) match the EuroHPC funding.

References

1. R. Tolosana-Calasanz, J. A. Banares, O. F. Rana, P. Álvarez, J. Ezpeleta, and A. Hoheisel, "Adaptive exception handling for scientific workflows," *Concurrency and computation: Practice and experience*, vol. 22, no. 5, pp. 617–642, 2010.
2. A. Kasuya and T. Tesfaye, "Verification methodologies in a tlm-to-rtl design flow," in *2007 44th ACM/IEEE Design Automation Conference*. IEEE, 2007, pp. 199–204.
3. B. Burns, *Designing Distributed Systems: Patterns and Paradigms for Scalable, Reliable Services*. " O'Reilly Media, Inc.", 2018.
4. B. Burns and D. Oppenheimer, "Design patterns for container-based distributed systems," in *8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 16)*. Denver, CO: USENIX Association, Jun. 2016. [Online]. Available: https://www.usenix.org/conference/hotcloud16/workshop-program/presentation/burns

5. S. Herbein, D. Domyancic, P. Minner, I. Laguna, R. da Silva, and D. Ahn, "Mcem: Multi-level cooperative exception model," Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States), Tech. Rep., 2019.
6. R. Gupta, P. Beckman, B.-H. Park, E. Lusk, P. Hargrove, A. Geist, D. Panda, A. Lumsdaine, and J. Dongarra, "Cifts: A coordinated infrastructure for fault-tolerant systems," in *2009 International Conference on Parallel Processing*. IEEE, 2009, pp. 237–245.
7. S. Di, R. Gupta, M. Snir, E. Pershey, and F. Cappello, "Logaider: A tool for mining potential correlations of hpc log events," in *2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*. IEEE, 2017, pp. 442–451.
8. PingCAP. (2020) A chaos engineering platform for kubernetes. [Online]. Available: https://github.com/chaos-mesh/chaos-mesh
9. CockroachDB. (2022, Mar.) Sstable corruption. [Online]. Available: https://github.com/cockroachdb/.../roachtest/tests/sstable_corruption.go
10. ——. (2022, Mar.) Sstable corruption. [Online]. Available: https://github.com/cockroachdb/.../roachtest/tests/network.go
11. X. Bai, W.-T. Tsai, R. Paul, T. Shen, and B. Li, "Distributed end-to-end testing management," in *Proceedings Fifth IEEE International Enterprise Distributed Object Computing Conference*. IEEE, 2001, pp. 140–151.
12. W.-T. Tsai, X. Bai, R. Paul, W. Shao, and V. Agarwal, "End-to-end integration testing design," in *25th Annual International Computer Software and Applications Conference. COMPSAC 2001*. IEEE, 2001, pp. 166–171.
13. J. Georgiou, M. Symeonides, M. Kasioulis, D. Trihinas, G. Pallis, and M. D. Dikaiakos, "Benchpilot: Repeatable & reproducible benchmarking for edge micro-dcs."
14. M. Symeonides, Z. Georgiou, D. Trihinas, G. Pallis, and M. D. Dikaiakos, "Fogify: A fog computing emulation framework," in *2020 IEEE/ACM Symposium on Edge Computing (SEC)*. IEEE, 2020, pp. 42–54.
15. F. Nikolaidis, M. Marazakis, and A. Bilas, "Iotier: A virtual testbed to evaluate systems for iot environments," in *2021 IEEE/ACM 21st International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*. IEEE, 2021, pp. 676–683.
16. KUTTL. (2022, Mar.) What is kuttl? [Online]. Available: The KUbernetes Test TooL
17. M. Toslali, S. Parthasarathy, F. Oliveira, H. Huang, and A. K. Coskun, *Iter8: Online Experimentation in the Cloud*. New York, NY, USA: Association for Computing Machinery, 2021, p. 289–304. [Online]. Available: https://doi.org/10.1145/3472883.3486984
18. F. Nikolaidis, A. Chazapis, M. Marazakis, and A. Bilas, "Frisbee: automated testing of cloud-native applications in kubernetes," *arXiv preprint arXiv:2109.10727*, 2021.