EasyChair Preprint

№ 15409

BugsInKube: a Collection of Reconciliation Bugs

Kabilan Mahathevan, Sivakajan Sivaparan, Tharsha Sivapalarajah, Sunimal Rathnayake and Ridwan Shariffdeen

November 12, 2024

BugsInKube: A Collection of Reconciliation Bugs

Kabilan Mahathevan
Dept. of Comp. Science and Eng.
University of Moratuwa
Sri Lanka
kabilan.19@cse.mrt.ac.lk

Sivakajan Sivaparan
Dept. of Comp. Science and Eng.
University of Moratuwa
Sri Lanka
sivakajan.19@cse.mrt.ac.lk

Tharsha Sivapalarajah
Dept. of Comp. Science and Eng.
University of Moratuwa
Sri Lanka
tharsha.19@cse.mrt.ac.lk

Sunimal Rathnayake
Dept. of Comp. Science and Eng.
University of Moratuwa
Sri Lanka
sunimal@cse.mrt.ac.lk

Ridwan Shariffdeen
School of Computing
National University of Singapore
Singapore
ridwan@comp.nus.edu.sg

***Abstract*—In the contemporary technological landscape, the widespread adoption of cloud systems and distributed resources has highlighted the need to overcome inherent limitations in achieving complete system dependability. This presents both significant opportunities and challenges in automating bug detection, bug fixing, and verification efforts in complex distributed systems, such as cloud infrastructure management tools like Kubernetes and Twine. Despite the importance of these efforts, there is a notable lack of data that can be used to study and analyze the types of challenges faced in developing and supporting these systems, as well as in building test automation and bug detection tools. To address this gap, we conducted an in-depth investigation into one of the most popular ecosystems: Kubernetes. We manually analyzed reported bugs and curated a comprehensive dataset comprising 311 developer-confirmed bugs. This dataset includes detailed information on bug categories, severity, affected versions, and reproducible steps when available. Through our analysis, we identified an emerging bug type in these systems, referred to as reconciliation bugs. To assist developers and researchers in creating new testing strategies for these platforms, we developed a bug-reproducing script that can reproduce 52 reconciliation bugs out of the 311 total bugs in Kubernetes. This tool provides valuable insights into these issues, facilitating the development of more robust testing and maintenance strategies. The dataset is publicly available and can be accessed at: https://github.com/EmInReLab/BugsInKube**

***Index Terms*—Container Orchestration, Kubernetes, Bug, Data Set**

I. Introduction

In today's digital landscape, cloud computing has significantly transformed how companies manage and maintain their applications and services [1]. There is a conspicuous trend toward the adoption and migration of workloads to the cloud, primarily driven by the pursuit of faster time-to-market, enhanced responsiveness, and substantial cost reductions. Cloud cluster management is a critical factor for both major tech companies and individuals in deploying and managing their applications in the cloud. Gartner predicts that by 2026, 75% of organizations will adopt a digital transformation model with cloud as the fundamental underlying platform [2].

However, this increased demand also brings potential challenges, including security issues, elasticity issues, high availability concerns, and multi-tenancy complications [3]. Any vulnerabilities or bugs in these platforms can have wide-ranging and severe consequences for the services they support, potentially impacting other applications using a multi-tenant cloud infrastructure. Although empirical studies are essential to understand the types of issues that emerge in cloud cluster management systems, enabling the development of better testing tools to detect these vulnerabilities before release, no significant bug analysis or bug dataset currently exists for cloud cluster management systems. Most empirical studies in cloud systems focus on cloud computing and big-data ecosystems [4]–[7], providing insights into general problems in the cloud. However, separate analyses of systems that manage the heterogeneous cloud hardware resources are required to understand the specific issues within these platforms.

To address these concerns, we analyzed reported bugs in one of the most prominent and widely used cloud cluster management systems, Kubernetes [8]. This container orchestration tool is commonly employed in constructing cloud services by efficiently managing hardware resources. We curated a comprehensive dataset comprising 311 developer-verified bugs sourced from the GitHub issue tracker. Through analyzing this dataset, we gleaned valuable insights into the bug categories, their severity, and the underlying problems within the system.

[1]GitHub issue link: https://github.com/kubernetes/kubernetes/issues/72593

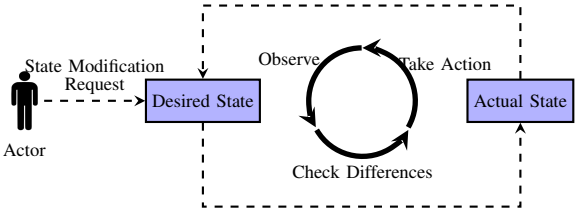
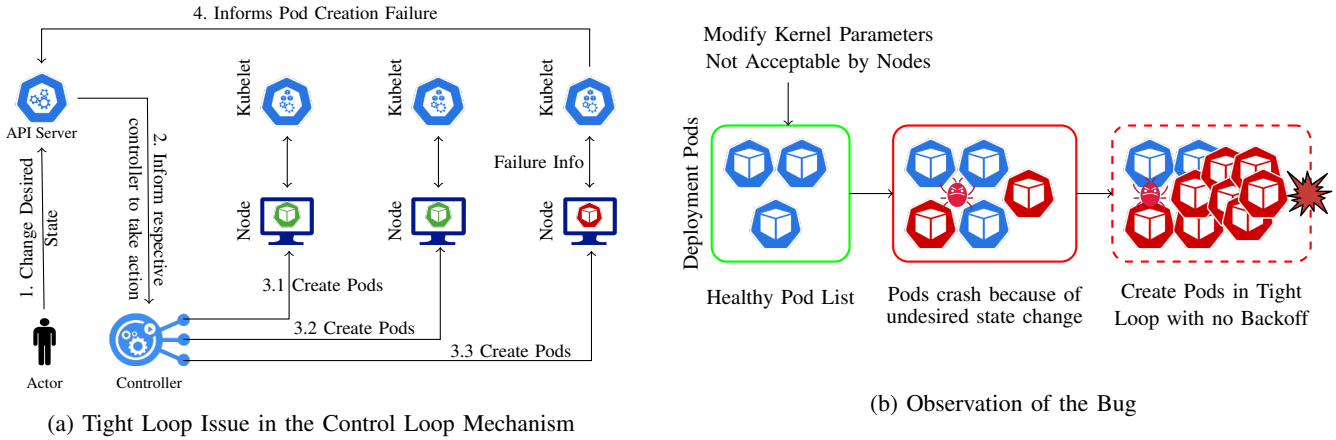


Fig. 1: Control Loop Mechanism of State Reconciliation



(a) Tight Loop Issue in the Control Loop Mechanism

(b) Observation of the Bug

Fig. 2: A Reconciliation Safety Bug in Kubernetes [1]

From the analysis of the manually curated dataset, we identified a category of bug called **reconciliation bug**, which constitutes the majority of the issues, with new bugs frequently falling into this category. More details about this bug type are explained in section II. We manually reproduced 52 reconciliation bugs in Kubernetes and developed a bug-reproduction tool, which requires the correct Kubernetes version to be installed beforehand. We believe that this dataset will support future work and studies related to building less vulnerable cloud cluster management systems, developing automated testing tools, and conducting software testing and maintenance activities for cloud platforms.

II. Background

The analysis of our dataset provided insights into an emerging category of bugs in cloud resource management systems, known as *reconciliation bugs* [9]–[11]. Cloud cluster management systems are designed with self-healing as a primary attribute, incorporating multiple components of the system that collaborate to restore the application in case of crashes and anomalies. Modern cloud cluster management systems [8], [12], [13] are built using state-reconciliation design pattern, in which the systems maintain a desired state to provide application services to public users and the actual state of the cloud cluster management systems is adjusted to reconcile with the desired state through the self-configuration, self-optimization, and self-healing properties of these systems [10] as shown in the Figure 1. The disparity between these two states can result in minor functional anomalies or failures that propagate to larger system failures similar to the safety bug in Kubernetes shown in Figure 2.

For every state property, there can be one or more controller components that operate to reconcile that particular state, but all follow the same control loop mechanism as depicted in Figure 1. For example, in Kubernetes, the ReplicaSet Controller is responsible for maintaining a specified number of pod replicas running at any given time. If a pod fails, the ReplicaSet Controller automatically creates a new pod to replace it, ensuring continuous availability and adherence to the desired state [14]. Additionally, other components within the system collaborate with the controller to identify any disparities in the state and complete the deployment of pods as needed. Any defect in any of these components, including the controller component itself, can disrupt the expected behavior and result in failures within the control loop mechanism. We categorize these defects in the cloud PaaS system as reconciliation bugs.

III. Motivational Example

Figure 2 depicts an example of a reconciliation bug. Consider creating a deployment in Kubernetes using the deployment configuration file as in Listing 1, an actor requests Kubernetes to modify the kernel parameter of the nodes running the deployment pods. Kernel parameters are settings that configure the behavior of the Linux kernel, governing various aspects of system operation such as memory allocation, process scheduling, and network configuration. Listing 1 defines the maximum number of connections that can be queued for acceptance by the kernel. This is particularly relevant for network-intensive applications like web servers. With this configuration modification, such as adjusting the maximum number of connections or open files [15], the desired state is modified, therefore the control loop checks the difference between the desired state and the actual state and takes the necessary action.

Listing 1: Bug injecting Kubernetes configuration

```
...
    spec:
      securityContext:
        sysctls:
          - name: net.core.somaxcon
            value: "10000"
...
```

In a cluster, each node can be either a physical or a virtual machine, and typically, its kernel parameter configurations are

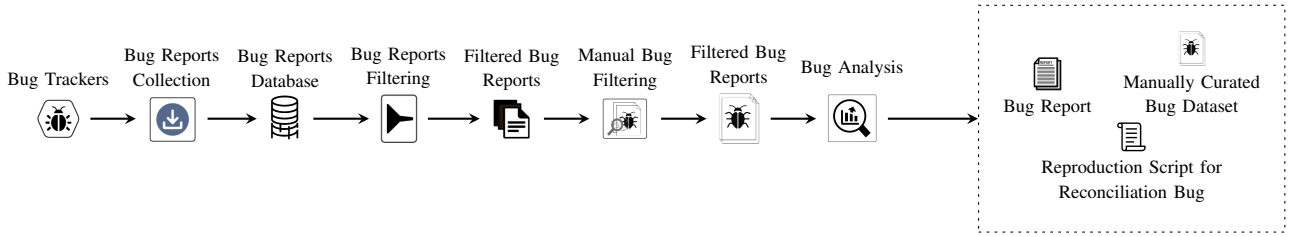


Fig. 3: High-level overview on the methodology

initially set to default values. For security reasons, modifications to kernel parameters of the nodes are restricted to the root user, preventing changes from any other user [15]. Hence, when attempting to align with the desired state, the deployed pod seeks to modify the kernel parameter of the node. However, the node rejects this modification, resulting in the failure of pod initialization. In such a scenario, if a node cannot accommodate a pod for execution, it fails. Subsequently, the ReplicaSet controller detects that the desired state is not met and attempts to create another pod in the cluster. However, this also fails, perpetuating a loop of unsuccessful attempts, as illustrated by the sequence $2 \rightarrow 3.3 \rightarrow 4 \rightarrow 2$ in the Figure 2a. Over time, these repeated attempts can consume significant CPU and memory, eventually leading to resource exhaustion and crashing the entire cluster as illustrated in Figure 2b.

This issue arises due to a lack of awareness in the scheduler component regarding the node's capacity to accommodate the pod during deployment. Alternatively, the controller may fail to recognize the repetitive creation of the same pod, neglecting to halt the creation process and provide pertinent information to the user. This issue has been recurrently reported in Kubernetes GitHub issues [1], with developers currently offering only a temporary solution. The frequent reporting of a bug by different individuals and organizations highlights its impact on the platform. Notably, 16% (52) of the 311 collected bugs in our dataset are reconciliation bugs, underscoring the significance of addressing these bugs in future development efforts. This also emphasizes the need to create specialized bug detection tools tailored to identifying and managing reconciliation bugs.

To facilitate further studies on **reconciliation bugs** [9] we extracted 52 reconciliation bugs in Kubernetes, in a reproducible data-set. We created the data-set to reproduce these bugs in an installed Kubernetes system, along with step-by-step instructions on what happens when each bug manifests. This contribution aims to improve the understanding of reconciliation bugs and motivate further research on program analysis to detect such bugs.

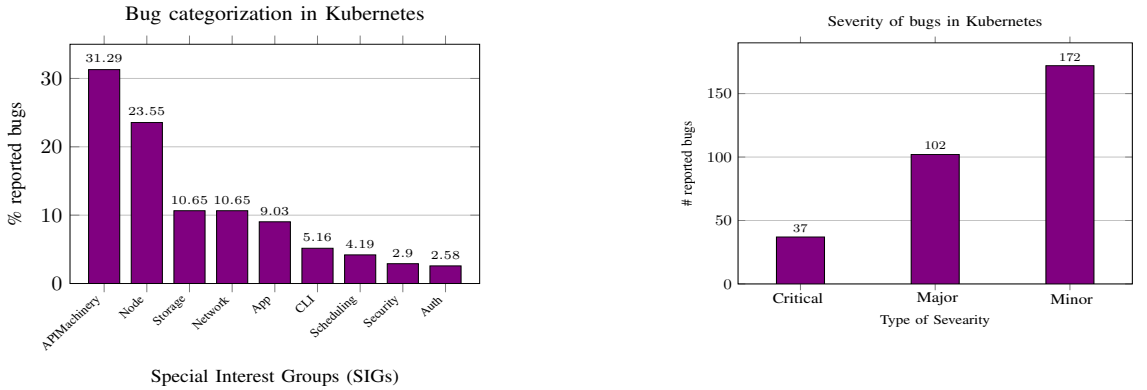
IV. Dataset

The significant growth of the open-source movement has led to a substantial increase in open-source cloud systems with publicly accessible issue repositories. These repositories contain valuable data, including bug reports, patches, and in-depth discussions among developers. This readily available information provides a valuable resource for understanding, analyzing, and categorizing bugs more effectively [16]. However, there can also be numerous unimportant issues, feature requests, incorrectly reported bugs, and issues created during the development process because of the open-source nature of these repositories. Therefore, the comprehensive analysis of reported bugs on these platforms can be a time-intensive process.

As indicated by Figure 3, we initially collected all issues from the Kubernetes issue tracker on GitHub. We then applied the following label filters: *is:issue label:kind/bug label:triage/accepted created:¿2014-10-15 -label:kind/failing-test -label:area/test -label:kind/documentation*. This filtering process reduced the number of issues from 44,321 to 1,770. Our filtering criteria focuses exclusively on issues designated as bugs by developers and restricts its analysis to issues submitted after the initial developer package release. This enabled the exclusion of irrelevant issues generated during software development. Additionally, we refined our analysis by excluding documentation-related issues and test failures. This reduced the total issues that needed manual inspection by 89%. Even among the filtered issues, not all proved to be valid bugs; some were related to performance optimization, feature requests, specific applications, or cloud provider support. To address this, we conducted a manual review of 1000+ issues and their corresponding developer conversations, ultimately disregarding those that did not pertain to actual bugs.

We identified 311 authentic bugs in Kubernetes, categorizing them according to the Special Interest Group (SIG) responsible for the affected component and the severity of each bug. For bugs with available information in the issue conversations, we created reproducible steps. Figure 4 illustrates this categorization, which facilitates the identification of bug-prone areas within the system. This can aid in formulating more effective testing strategies that prioritize these areas, leading to efficient detection of existing bugs and preventative measures against future occurrences.

While analyzing and categorizing these bugs, we noticed that a significant portion—over 35%—of the dataset is related to components responsible for scheduling, tagged with the API Machinery and Scheduling SIGs. The API Machinery SIG focuses on the development and improvement of the Kubernetes cluster control plane, including the API server, persistence layer (etcd), controller manager, cloud controller manager, CustomResourceDefinition, and webhooks. The API Machinery and Scheduling SIGs collaborate to manage con-



(a) Special Interest Group categorization of bugs in Kubernetes (b) Bug Categorization based on Severity in Kubernetes

Fig. 4: Different Bug Categorization

trollers that maintain consistency between the desired and current states of the Kubernetes cluster configuration [17]. This prompted us to create reproducible scripts for the emerging reconciliation bug category.

The bug-reproducing script is designed to first deploy the correct Kubernetes configuration and verify that the control loop mechanism functions as expected. After a short waiting period, it introduces a deployment configuration that triggers reconciliation issues. The tool also logs the entire process, providing users with comprehensive insights into the tool's behaviour during execution.

V. Related Work

Heterogeneous cloud container orchestration systems, such as Kubernetes, are inherently complex due to their integration of multiple individual programs. These systems enable cloud and application providers to define the selection, deployment, monitoring, and dynamic control of multi-container applications in the cloud. Unlike single-server systems, their complexity arises from the need to manage diverse components and ensure seamless operation across distributed environments. Due to the presence of numerous distributed components, hardware failures, diverse user interactions, and deployment scenarios, occasional downtimes are anticipated. There is much room for improving cloud systems' dependability [7], [18], [19]. There is no significant bug dataset specifically focusing on reconciliation bugs, but there are a few bug analyses on general cloud platforms.

The most extensive bug hunting in distributed systems has been conducted in CbsDB [7] on six scale-out systems: Cassandra, Flume, HBase, HDFS, MapReduce, and ZooKeeper. They analyzed over 3000 issues in the issue tracking systems of open-source cloud systems, originating from developers' code reviews, in-house testing, and user reports between 2011 and 2014. This detailed analysis led to the categorization of issues into reliability (45%), performance (22%), availability (16%), data consistency (5%), scalability (1%), and QoS (1%). These insights are invaluable for system developers and operators, systems researchers, and tool builders aiming to enhance the reliability of future scale-out systems [4], [20], [21].

Haopeng Liu et al. systematically studied all high-severity production-run incidents over six months in Microsoft Azure services [22]. It identifies software bugs as the most common cause of cloud incidents, accounting for nearly 40%. The study provides insights into the types of software bugs that lead to production incidents, their resolution methods, and differences from failures in single-machine systems. This research enhances the understanding of cloud service reliability and potential automation in incident resolution.

Recent research on building end-to-end testing for Kubernetes CustomResourceDefinition [9] signifies the importance of test automation in identifying reconciliation bugs. However, these end-to-end tests require extensive manual effort and are limited by their inability to cover all possible system states and transitions. This limitation makes comprehensive validation prohibitively expensive and may result in missed bugs and false alarms.

VI. Limitations and Future Work

Our focus is on identifying reconciliation bugs across various cloud cluster management systems, not just Kubernetes. Due to the active open-source community around Kubernetes, we initially curated a dataset of reconciliation bugs specific to this platform. However, we plan to expand our dataset to include other cloud cluster management systems, such as Twine [12] and Apache Mesos [23], to broaden our understanding and generalize the concept of reconciliation bugs across different platforms.

The full dataset consists only of developer-confirmed reported bugs in Kubernetes, including both open and closed issues. Since the developer community is smaller compared to the user community, many bug reports have not been confirmed by developers. Therefore, those unconfirmed bug reports, despite appearing legitimate, are not included in our dataset. Additionally, many of these bugs are reported by various cloud administrators and individuals across different

organizations, which often obscures the true impact of these bugs in production environments due to the anonymity maintained in the reporting process.

Bug reports involve extensive discussions between the reporter and developers, making them quite lengthy. After reading through these conversations, we extracted and included reproducible steps from the discussions, wherever possible, into the full dataset. However, we did not manually reproduce all the bugs in the full dataset. In future work, manually reproducing all the bugs from the full dataset and providing reproducible steps for each bug would be beneficial for understanding the bugs and building better testing tools.

To reduce the manual effort in testing these platforms while ensuring the non-existence of reconciliation bugs, it is essential to develop a testing framework specifically catered to this need. Techniques like chaos testing [24]–[26], while effective for identifying reconciliation bugs, often lack determinism. However, integrating deterministic approaches similar to Mallory [27] for distributed systems into chaos testing could help bring more predictability to the process. This integration could pave the way for creating a fuzz testing tool specifically designed to identify reconciliation bugs in cloud cluster management tools.

VII. Conclusion

This paper presents a collection of developer-confirmed bugs in Kubernetes, a container orchestration tool. Additionally, it includes reproduction scripts for an emerging category of bugs known as reconciliation bugs. We believe that this dataset will support open, reproducible future research in automated bug report management, software testing, and software maintenance more broadly. A demonstration video showcasing the tool in action can be accessed here: https://www.youtube.com/watch?v=q0cPP-GHJUc

Bug Data Set: Our dataset can be accessed via the following link: https://github.com/EmInReLab/BugsInKube

References

[1] "Why developers like Kubernetes," Oct. 2023, [Online; accessed 2. Oct. 2023]. [Online]. Available: https://stackshare.io/kubernetes

[2] "Gartner Forecasts Worldwide Public Cloud End-User Spending to Reach Nearly $600 Billion in 2023," Oct. 2023, [Online; accessed 18. Oct. 2023]. [Online]. Available: https://shorturl.at/jhR5o

[3] G. N. Iyer, J. Pasimuthu, and R. Loganathan, "PCTF: An integrated, extensible cloud test framework for testing cloud platforms and applications," in *2013 13th International Conference on Quality Software*. IEEE, Jul. 2013. [Online]. Available: https://doi.org/10.1109/qsic.2013.65

[4] H. Liu, G. Li, J. F. Lukman, J. Li, S. Lu, H. S. Gunawi, and C. Tian, "Dcatch: Automatically detecting distributed concurrency bugs in cloud systems," *ACM SIGARCH Computer Architecture News*, vol. 45, no. 1, pp. 677–691, 2017.

[5] H. Chen, W. Dou, Y. Jiang, and F. Qin, "Understanding exception-related bugs in large-scale cloud systems," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2019, pp. 339–351.

[6] D. J. Dean, H. Nguyen, X. Gu, H. Zhang, J. Rhee, N. Arora, and G. Jiang, "Perfscope: Practical online server performance bug inference in production cloud computing infrastructures," in *Proceedings of the ACM Symposium on Cloud Computing*, 2014, pp. 1–13.

[7] H. S. Gunawi, M. Hao, T. Leesatapornwongsa, T. Patana-anake, T. Do, J. Adityatama, K. J. Eliazar, A. Laksono, J. F. Lukman, V. Martin, and A. D. Satria, "What bugs live in the cloud? a study of 3000+ issues in cloud systems," ser. SOCC '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 1–14. [Online]. Available: https://doi.org/10.1145/2670979.2670986

[8] "Production-Grade Container Orchestration," Mar. 2024, [Online; accessed 21. Mar. 2024]. [Online]. Available: https://kubernetes.io

[9] J. T. Gu, X. Sun, W. Zhang, Y. Jiang, C. Wang, M. Vaziri, O. Legunsen, and T. Xu, "Acto: Automatic end-to-end testing for operation correctness of cloud system management," in *Proceedings of the 29th Symposium on Operating Systems Principles*, 2023, pp. 96–112.

[10] "Cloud native computing foundation operator white paper," https://www.cncf.io/wp-content/uploads/2021/07/CNCF_Operator_WhitePaper.pdf.

[11] J. Bowes, "Level Triggering and Reconciliation in Kubernetes," Jan. 2018, [Online; accessed 26. Mar. 2024]. [Online]. Available: https://hackernoon.com/level-triggering-and-reconciliation-in-kubernetes-1f17fe30333d

[12] C. Tang, K. Yu, K. Veeraraghavan, J. Kaldor, S. Michelson, T. Kooburat, A. Anbudurai, M. Clark, K. Gogia, L. Cheng *et al.*, "Twine: A unified cluster management system for shared infrastructure," in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, 2020, pp. 787–803.

[13] T. Melissaris, K. Nabar, R. Radut, S. Rehmtulla, A. Shi, S. Chandrashekar, and I. Papapanagiotou, "Elastic cloud services: scaling snowflake's control plane," in *Proceedings of the 13th Symposium on Cloud Computing*, 2022, pp. 142–157.

[14] "Controllers," Nov. 2023, [Online; accessed 3. Jun. 2024]. [Online]. Available: https://kubernetes.io/docs/concepts/architecture/controller

[15] "Chapter 5. Configuring kernel parameters at runtime Red Hat Enterprise Linux 8 | Red Hat Customer Portal," Jun. 2024, [Online; accessed 4. Jun. 2024]. [Online]. Available: https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/8/html/managing_monitoring_and_updating_the_kernel/configuring-kernel-parameters-at-runtime_managing-monitoring-and-updating-the-kernel

[16] T. Diamantopoulos, D.-N. Nastos, and A. Symeonidis, "Semantically-enriched jira issue tracking data," in *2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR)*. IEEE, 2023, pp. 218–222.

[17] "Kubernetes github issues." [Online]. Available: https://github.com/kubernetes/kubernetes

[18] A. Velimirovic, "Cloud Outage: Why and How Does It Happen?" *phoenixNAP Blog*, Sep. 2023. [Online]. Available: https://phoenixnap.com/blog/cloud-outage

[19] E. Casalicchio, *Container Orchestration: A Survey*. Cham: Springer International Publishing, 2019, pp. 221–235. [Online]. Available: https://doi.org/10.1007/978-3-319-92378-9_14

[20] T. Leesatapornwongsa, M. Hao, P. Joshi, J. F. Lukman, and H. S. Gunawi, "{SAMC}:{Semantic-Aware} model checking for fast discovery of deep bugs in cloud systems," in *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, 2014, pp. 399–414.

[21] C. Zhang, J. Li, D. Li, and X. Lu, "Understanding and statically detecting synchronization performance bugs in distributed cloud systems," *IEEE Access*, vol. 7, pp. 99 123–99 135, 2019.

[22] H. Liu, S. Lu, M. Musuvathi, and S. Nath, "What bugs cause production cloud incidents?" in *Proceedings of the Workshop on Hot Topics in Operating Systems*, 2019, pp. 155–162.

[23] "Apache Mesos," Dec. 2022, [Online; accessed 21. Mar. 2024]. [Online]. Available: https://mesos.apache.org

[24] "ChaosBlade · Help companies solve the high availability problems in the process of migrating to cloud-native systems through chaos engineering | ChaosBlade," Jul. 2024, [Online; accessed 13. Aug. 2024]. [Online]. Available: https://chaosblade.io/en

[25] "chaosmonkey," Mar. 2024, [Online; accessed 24. Mar. 2024]. [Online]. Available: https://github.com/Netflix/chaosmonkey

[26] "Chaos Mesh Overview | Chaos Mesh," Mar. 2024, [Online; accessed 24. Mar. 2024]. [Online]. Available: https://chaos-mesh.org/docs

[27] R. Meng, G. Pîrlea, A. Roychoudhury, and I. Sergey, "Distributed system fuzzing," *arXiv preprint arXiv:2305.02601*, 2023.