Available online www.jsaer.com

Journal of Scientific and Engineering Research, 2021, 8(8):181-185

Research Article

ISSN: 2394-2630
CODEN(USA): JSERBR

Resilience by Design: A Comprehensive Guide to Enhancing Resilience through Cloud Native Chaos Engineering

Savitha Raghunathan

Email: saveetha13@gmail.com

Abstract The necessity of chaos engineering in cloud native environments arises from the inherent complexities and dynamic nature of cloud computing. Organizations transitioning to microservices, containers, and serverless architectures encounter unprecedented scalability and flexibility. However, this evolution also increases system complexity and a greater potential for unpredictable failures. This whitepaper addresses this critical need by exploring how intentional fault injection can be a proactive tool for identifying vulnerabilities, enabling teams to address them before they lead to service degradation or outages. This paper delves into the foundational principles of chaos engineering, tailored methodologies for cloud native systems, the selection of appropriate tools, and the tangible benefits of adopting these practices. Organizations can significantly enhance system reliability and resilience by integrating chaos engineering into cloud native development and operations, ensuring their services can withstand and adapt to the digital ecosystem's evolving challenges. This guide aims to equip technology leaders and engineers with the knowledge to embed resilience into their cloud native architectures, making their systems robust against known challenges and adaptable to future disruptions.

Keywords Resilience, Reliability, Chaos Engineering, Cloud Native, Kubernetes Reliability

1. Introduction

The evolution of cloud computing has led to the widespread adoption of cloud native technologies, characterized by their scalability, flexibility, and resilience. However, these environments' complexity and dynamic nature also introduce new challenges in ensuring system reliability, making them react unexpectedly as shown in figure 1 below.

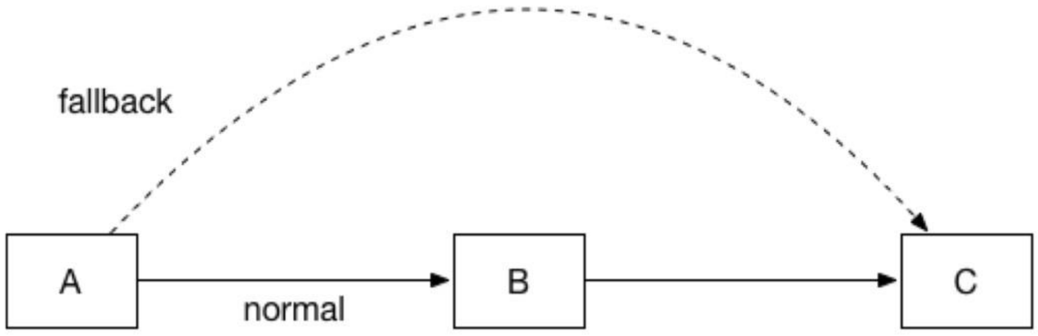

Figure 1: Unexpected system behavior [4]

Chaos engineering emerges as a proactive approach to identifying weaknesses in a system by intentionally injecting failure scenarios [2]. In cloud native ecosystems, where services are often ephemeral and distributed, chaos engineering is pivotal for maintaining system health and ensuring reliability.

2. Principles of Chaos Engineering

Chaos engineering is grounded in a set of core principles [2] designed to guide practitioners in systematically introducing and managing chaos:

2.1 Build a Hypothesis Around Steady State Behavior [3]: This foundational principle involves defining what normal operation (steady state) looks like for the system as shown in figure 2, including key performance indicators like latency, error rates, and throughput. Creating this hypothesis requires thoroughly understanding the system's capabilities and limitations under typical operating conditions. This involves collecting and analyzing historical data and then using this data to predict and quantify the impact of potential failures. This knowledge allows teams to identify when a system deviates from its steady state and the likely reasons behind such deviations.

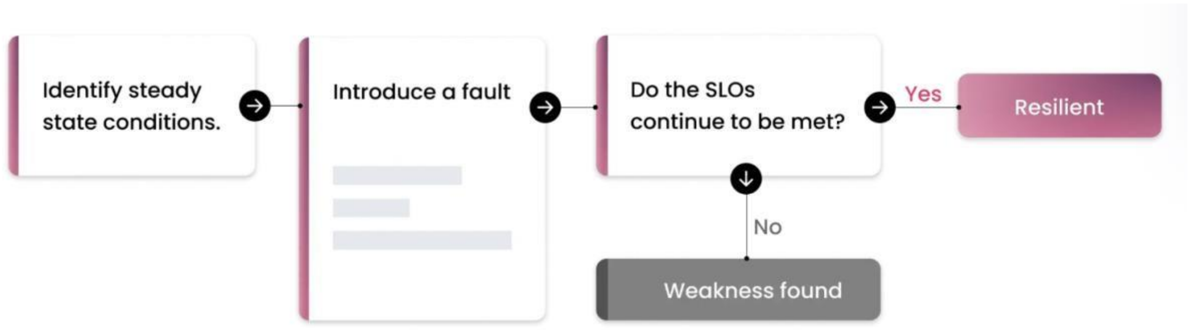
Identify steady state conditions.
Introduce a fault
Do the SLOs continue to be met?
Yes
Resilient
No
Weakness found

Figure 2: Chaos engineering process [6]

2.2 Vary Real-World Events [3]: The complexity of modern systems means that they can be affected by a wide range of external factors. This principle is about simulating real-world events, such as spikes in traffic, server failures, network partitions, and resource exhaustion, to see how they affect the system's operations. By varying these inputs and observing the outcomes, teams can better prepare for unexpected disruptions. For example, by intentionally introducing a network delay [7], you can assess how well your service degrades in performance when latency increases, thereby understanding the resilience of your system's communications. The objective is to reveal hidden bugs and monitor blind spots and performance bottlenecks that could lead to outages or service degradation in the face of such real-world events.

2.3 Run Experiments in Production [3]: Testing in an environment that closely resembles real-world usage is important, and there is no substitute for the production environment. By conducting experiments in production, it is possible to encounter the actual state of the system under current loads and operational conditions. While this approach carries risk, the principle is that it is better to test under controlled circumstances rather than waiting for an uncontrolled real incident. This allows teams to understand the system's actual behavior, not just its theoretical performance.

2.4 Automate Experiments to Run Continuously [3]: Automation is key to executing chaos experiments consistently and scalably [7]. Automated chaos experiments can be run continuously as part of the CI/CD pipeline, during off-hours, or in response to specific triggers. It ensures that resilience testing is not a one-off event but an integral part of the development and deployment lifecycle. Tools have evolved to provide sophisticated scheduling, execution, and analysis of chaos experiments, enabling organizations to integrate chaos experiments into their automated testing suites. As systems grow and evolve, automation ensures that resilience is continuously assessed and validated, even as new features are rolled out and changes are made to the system.

3. Chaos Engineering in Cloud Native

The diagram outlines the application's resilience hierarchy, based upon various layers from infrastructure to Kubernetes and cloud native services. Each layer is subject to frequent updates. Over 90% of an application's resilience is tied to this underlying stack rather than the application code, underscoring the need for continuous validation of resilience, especially post-updates, through regular chaos engineering practices [1].

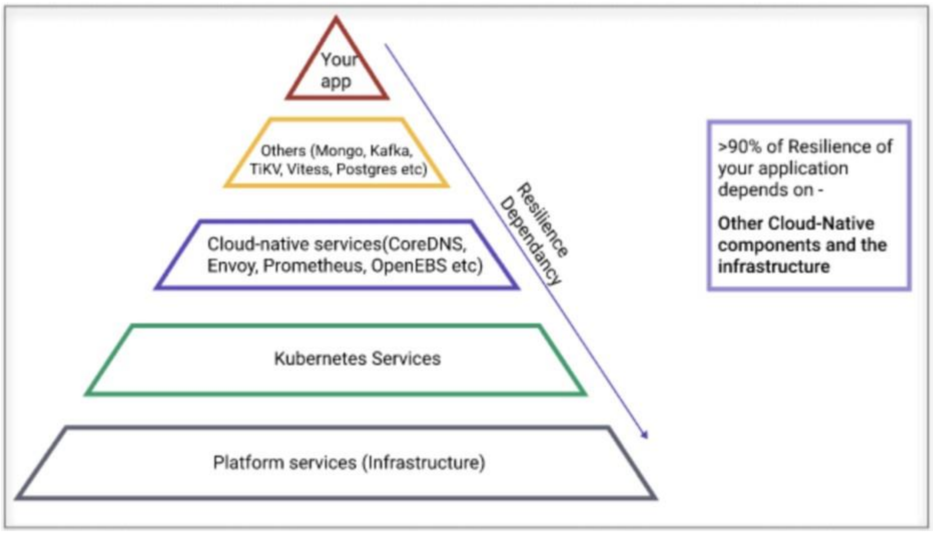
Your app
Others (Mongo, Kafka, TiKV, Vitess, Postgres etc)
Cloud-native services(CoreDNS, Envoy, Prometheus, OpenEBS etc)
Kubernetes Services
Platform services (Infrastructure)
Resilience Dependancy
>90% of Resilience of your application depends on -
Other Cloud-Native components and the infrastructure

Figure 3: Application's Resilience dependency [1]

In cloud native environments, chaos engineering methodologies need to adapt to the unique characteristics of these systems, spanning across multiple layers, from the container level—where disruptions like container termination or induced network latencies test service recovery protocols—to the orchestration layer, where the robustness of systems like Kubernetes is evaluated against simulated scheduler malfunctions and node failures. At the infrastructure level, underlying resource stability is assessed by intentionally causing outages of virtual machines or entire availability zones. Finally, at the application level, fault injection into the application's processes provides insights into the application's resilience, ensuring that it can withstand and gracefully recover from various failure modes.

4. Benefits of Cloud Native Chaos Engineering

Implementing chaos engineering practices in cloud native environments brings several benefits:

- **Improved Reliability:** Systems become more reliable and robust by proactively identifying and addressing potential failures [9].
- **Enhanced Resilience:** Systems are better prepared to handle unexpected disruptions, minimizing downtime and ensuring continuous service availability [9].
- **Increased Confidence:** Teams gain confidence in their system's ability to withstand real-world events, reducing fear of deployments and changes.
- **Cost Savings:** Identifying and addressing issues early can save significant costs by avoiding downtime and business loss [8].

5. Implementing Cloud Native Chaos Engineering

5.1 Understand the System's Architecture

- Document the architecture: Have a clear understanding of the system's architecture, including services, dependencies, and communication patterns.
- Identify critical components: Determine which parts of the system are critical for the business operations and should be prioritized for chaos experiments.

5.2 Establish a Baseline of Normal Operation [5]

- Define steady state: Establish the system's normal operation, including key metrics that indicate its health (e.g., latency, throughput, error rates).
- Monitoring and observability: Ensure that robust monitoring and observability tools are used to measure the system's steady state and detect deviations.

5.3 Start with a Chaos Engineering Plan [5]

- Formulate hypotheses: Based on the understanding of the system, hypothesize what could go wrong and how the system might respond to different failure scenarios.
- Define experiments: Create detailed plans for chaos experiments that will test these hypotheses, including the scope, methods of failure injection, and expected outcomes.

5.4 Choose the Right Tools

Select tools and platforms that fit the organization's cloud native ecosystem and can effectively inject failures. Popular tools for cloud native chaos engineering include:

- **Chaos Monkey**: For randomly terminating instances to ensure resilience [5].
- **LitmusChaos** [11]: A Cloud Native Computing Foundation sandbox project tailored for Kubernetes environments.
- **Chaos Mesh** [10]: Offers comprehensive chaos testing on Kubernetes, from infrastructure to application-level chaos.
- **Gremlin** [11]: Provides a "Failure as a Service" platform, allowing controlled chaos experiments across your stack.

5.5 Perform Chaos Experiments in a Controlled Environment [12]

- Minimize blast radius: Start with the smallest possible scope and gradually increase the scale of the experiments.
- Conduct experiments in staging: Before moving to production, test the chaos experiments in a staging environment that closely mirrors production.

5.6 Automate and Integrate into CI/CD [3]

- Automation: Automate chaos experiments to run periodically or as part of the CI/CD pipeline to continuously test and improve resilience.
- Integration: Integrate chaos engineering into the development lifecycle, allowing continuous feedback and improvement.

5.7 Analyze Results and Learn

- Measure impact: Analyze the impact of chaos experiments on the system, comparing against the baseline to identify deviations.
- Document findings: Keep detailed records of each experiment, including the hypothesis, method, results, and any system improvements made.
- Iterate: Use the insights from experiments to iterate on the system design and operation, enhancing resilience.

5.8 Cultivate a Culture of Reliability

- Promote understanding: Ensure that the team understands the value of chaos engineering and its role in improving system reliability.
- Encourage participation: Engage the whole team in planning, executing, and learning from chaos experiments.

6. Conclusion

Cloud native chaos engineering represents a proactive and systematic approach to enhancing system reliability and resilience in cloud native environments. By embracing failure as an opportunity for improvement, organizations can ensure their services remain robust, scalable, and resilient in the face of ever-evolving challenges. As cloud native technologies continue to evolve, the principles and methodologies of chaos engineering will remain vital for maintaining the health and reliability of these complex systems.

References

[1]. U. Mukkara, “Introduction to LitmusChaos,” CNCF, Aug. 28, 2020. https://www.cncf.io/blog/2020/08/28/introduction-to-litmuschaos/

[2]. A. Basiri et al., “Chaos Engineering,” IEEE Software, vol. 33, no. 3, pp. 35–41, May 2016, doi: https://doi.org/10.1109/MS.2016.60.

[3]. T. L. Tran, “Chaos Engineering for Databases,” Master Thesis, Universiteit van Amsterdam-Vrije Universiteit Amsterdam, 2020. Available: https://homepages.cwi.nl/~boncz/msc/2020-LongTran.pdf

[4]. A. Blohowiak, A. Basiri, L. Hochstein, and C. Rosenthal, “A Platform for Automating Chaos Experiments,” IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW), pp. 5–8, Oct. 2016, doi: https://doi.org/10.1109/ISSREW.2016.52.

[5]. M. Lafeldt, “Chaos Engineering 101,” Sharpend, Feb. 10, 2016. https://sharpend.io/chaos-engineering-101/

[6]. K. S, "Principles of Cloud Native Chaos Engineering," ChaosNative, May 06, 2021. https://www.chaosnative.com/blog/principles_of_cloud_native

[7]. K. Torkura, "From Resilience to Dependability: Security Chaos Engineering for Cloud Services," Medium, Nov. 02, 2019. https://run2obtain.medium.com/from-resilience-to-dependability-security-chaos-engineering-for-cloud- services-9c6d6d152ed2

[8]. S. Bocetta, "How to Use Chaos Engineering to Break Things Productively," InfoQ, Sep. 02, 2019. https://www.infoq.com/articles/chaos-engineering-security-networking/

[9]. The Chief I/O, "Introduction to Chaos Engineering," TheChief. https://thechief.io/c/editorial/introduction-to- chaos-engineering/

[10]. C. Yin, "Chaos Mesh - Your Chaos Engineering Solution for System Resiliency on Kubernetes," Chaos Mesh, Jan. 15, 2020. https://chaos-mesh.org/blog/chaos_mesh_your_chaos_engineering_solution/

[11]. Apexon, "What Is Chaos Engineering? Approaches, Best Practices & Case Studies," Apexon. https://www.apexon.com/insights/white-papers/Apexon_White_Paper_Chaos_Engineering.pdf

[12]. K. Andrus, "3 Key Steps for Running Chaos Engineering Experiments," *InfoWorld*, Apr. 11, 2018. https://www.infoworld.com/article/3268017/3-key-steps-for-running-chaos-engineering-experiments.html