Chaos Engineering and Testing for Availability and Resiliency in Cloud

Amith Shetty, Krithika Hegde, Atul Ahire

amith_shetty@mcafee.com, krithika_hegde@mcafee.com, atul_ahire@McAfee.com

Abstract

As enterprises embrace the cloud for large-scale distributed application deployments, it has become crucial to making sure that systems are always available and resilient to failures. We are quick to adopt practices that increase the flexibility of development and velocity of deployment. But as systems scale, we expect part of the infrastructure to fail ungracefully in random and unexpected ways. We must design cloud architecture where part of the system can fail and recover without affecting the availability of the entire system.

We make our architecture fault-tolerant and resilient by adopting multi-availability zone deployments, multi-region deployments, and other techniques. However, it is equally important to test these failure scenarios to be confident about surviving in these disruptions and recovering quickly. Functional specifications fail to describe distributed systems because we cannot characterize all possible inputs, thus the need to validate system availability in production with chaos engineering.

This paper explains how our teams can take small first steps in learning and building Chaos Engineering and Testing tools for cloud deployment in Amazon Web Services. If adopted and experimented in a controlled manner, we will be able to learn system behavior in a chaotic state and design preventive actions to recover from disaster situations and reduce the downtime.

Biography

Amith Shetty is a Technical Lead, at McAfee, based in Bangalore, India. He has over 10 years of experience in building applications in Microsoft .NET technology stack. He is experienced working on highly scalable, fault tolerant cloud applications with Amazon Web Services.

Krithika Hegde is a Senior Software Development Engineer at McAfee, with over 8 years of experience in product development and Integration Activities She has participated in various product life cycles and been a key contributor to various releases within McAfee. She is passionate about leveraging technology to build innovative and effective software security solutions.

Atul Ahire is an SDET at McAfee, with more than 6 years of experience in Software Development, Test Automation, Build Release and Deployment. His areas of interest include Cloud solution design, deployment, and DevOps.

1 Introduction

Over the years we have seen drastic changes in the way systems are built and the scale at which they operate. With the scale comes the complexity and there are so many ways these large-scale distributed systems can fail. Modern systems built on cloud technologies and microservices architecture have a lot of dependencies on the internet, infrastructure, and services which you do not have control over. It has become difficult to ensure that these complex systems are tested to be always on, resilient and fault tolerant. Confidence in distributed system behavior can be determined by experimenting with worst-case failure scenarios in production or close to production.

When we develop new or existing software, we toughen our implementation through various forms of tests. We often refer to this as a test pyramid that illustrates what kinds of tests we should write and to what extent. This pyramid at a minimum consists of Unit Tests, Integration Tests, and System Tests.

When creating unit tests, we write test cases to check the expected behavior. The component we are testing is free of all its dependencies and we keep their behavior under control with the help of mocks. These types of tests cannot guarantee that the system as a whole is free of errors.

Integration tests test the interaction of individual components. These are ideally run automatically after the successfully tested unit tests and test-interdependent components. We achieve a very stable state of our application with these tests, but only under real conditions of production environments, we see how all the individual components of the overall architecture behave. This uncertainty is increased with the adoption of modern microservice architectures, where set of possible system states grows exponentially over time.

Chaos Engineering is the discipline of experimenting on a distributed system to build confidence in the system's capability to withstand turbulent conditions in production. Chaos Testing is a deliberate introduction of disaster scenarios into our infrastructure in a controlled manner, to test the system's ability to respond to it. This is an effective method to practice, prepare, and prevent/minimize downtime and outages before they occur. Traditional testing methods mentioned above are necessary for every system. Most of these testing methods determine the validity of a known scenario under a given condition, whereas Chaos testing is one of the ways to generate new knowledge by experimenting complicated scenarios which these tests are not expected to cover.

2 Principles of Chaos Engineering

The term Chaos Engineering was coined by Engineers at Netflix. The idea of Chaos engineering is instead of waiting for systems to fail in production, perform experiments and proactively inject failures to be ready when disaster strikes. Chaos Engineering is all about running experiments and these experiments are designed based on the following principles (Principles of Chaos Engineering 2018).

Figure 1 Principles of Chaos Engineering

1. Define system's normal behavior: The Steady state can be defined as some measurable output like overall throughput, error rates, the latency of a system that indicates normal behavior.
2. Hypothesize about the steady state: Make assumptions about the behavior of an experimental group, as compared to a stable control group.
3. Vary real-world events: Expose experimental group to experiments by introducing real-world events like terminating servers, network failures, latency, etc.
4. Observer Results: Test the hypothesis by comparing the steady state of the control group and experimental group. The smaller the difference, the more confidence we have in the system.

3 Real-world example of chaos experiment

Let us look at some examples that can be considered for experiments. Injecting failures such that individual pieces of the infrastructure become unavailable is a good way to learn about decoupling in systems.

1. Let us assume that our application performs compute-intensive operations like mathematical calculations. Resources like CPU and memory are limited on standalone machines. When number of computations increases, the application will be forced to handle lack of resource. We can test the precautions taken by the development team to handle the resource exhaustion verify that steady state remains unaltered throughout the experiment.
2. Let us say we are building a music streaming service with microservice architecture. This application has a recommendation service which recommends music to users based on history. When these services are built as loosely coupled independent services, deployed separately, it is a common scenario for services to go down for shorter or longer duration of time, or not reachable over the network. As a common practice, we can fall back to graceful behavior, subtlety return recommendations from a cache or a list of static content to the user in this scenario. We can use chaos experiments to simulate this failure of network or instance and find out if the system behaves as expected.

With such cost-effective experiments, we will be able to disprove our hypothesis and find weaknesses in system behavior.

4 Chaos Engineering in practice

Chaos experiments should be designed by considering a collection of services as a single system to understand the behavior of the system rather than testing individual components in the system. Following is the overview of the process involved in designing chaos experiments (Casey, et al. 2017).

4.1 Prerequisites

1. As stated in principles, chaos engineering is all about discovering the unknowns, if you know that your system cannot withstand failures, fix those problems first.
2. Have monitors in place which can provide visibility into the steady state of the system and help in making conclusions about experiments. There are two classes of metrics which can determine your steady state.
    a. Business metrics: Business metrics are the numbers that indicate activities in the system, that have a direct impact on the revenue. For example, in e-Commerce systems, how many checkouts are successful over time? Metrics can also be answers to questions like, is each customer happy with response time of the application and not quitting before checkout? Typically, these metrics are harder to implement than system metrics. These metrics should be a matter of continuous monitoring in contrast to the calculations at end of the month.

b. System metrics: System metrics measure and monitor the overall health of the system. For example, CPU usage, memory, disk usage, response time, error rate of the APIs etc. These can help to troubleshoot performance issues and sometimes functional bugs.

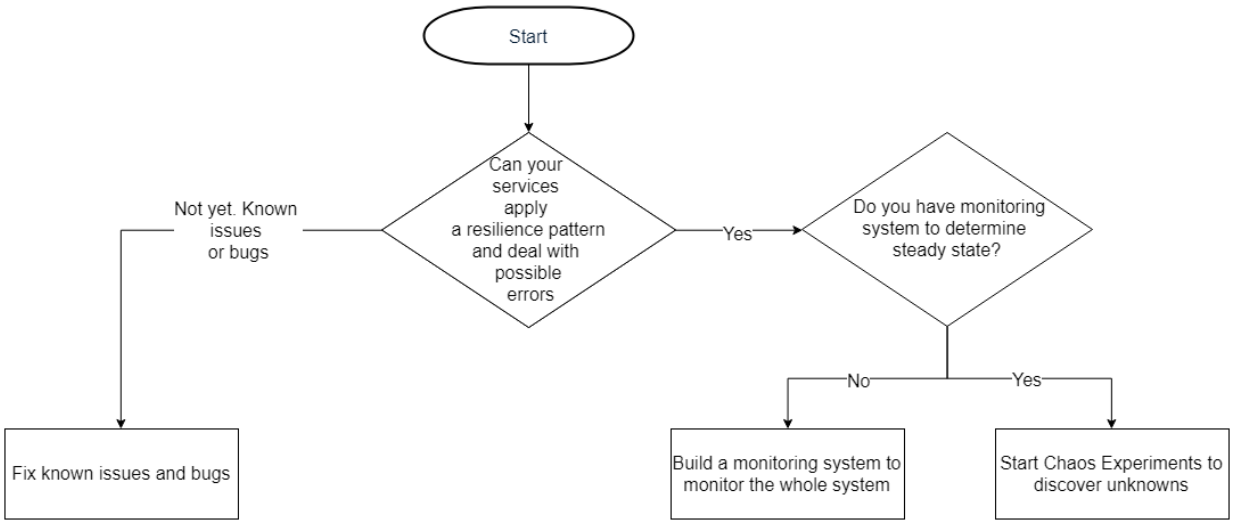

Figure 2 Prerequisites

4.2 Designing experiments

Let us look at the process involved in designing chaos experiments in general.

4.2.1 Pick a Hypothesis :

System behaviors are generally categorized as acceptable behaviors and unexpected behaviors. The normal state of the system should be considered as the steady state. The goal of this step is to develop a model that characterizes the steady state of the system based on business metrics. If a system is new, steady state is determined by regular testing of the application and collecting metrics over time which can portray the healthiest state of the system.

Once we have the steady state behavior and metrics, we can derive Hypothesis for the experiment. Hypothesis chosen here will be believed as the output of the experiment. If we have a strong hypothesis about the steady state, designing experiments becomes easy as we precisely know what experiments to create instead of creating uncontrolled Chaos. Since Chaos Engineering is to ensure the reliability or the graceful degradation of systems, the hypothesis for the tests should be in line with the statement "the events we will inject into the system will not result in a change from the steady state."

4.2.2 Choose the scope of the experiment:

While planning the experiments we need to carefully consider how far we need to go to learn something useful about the system. We define the area which can get affected by the experiment as its blast radius and it should be as minimal as possible. The scope of the experiment should be determined to minimize the impact on real customers if running on production and in general. Although running tests closer to production will yield better results as the real events often happen to occur in production, not containing the blast radius can cause customer pain. Start experiments in test environments initially, and start with the smallest possible tests, until you gain confidence and then simulate bigger events.

4.2.3 Identify the system/ business metrics:

A central metric system could be capturing metrics for multiple services. Out of these metrics, identify the correct metrics which you can use to evaluate the results of the experiments chosen.

We should be very particular about evaluating the result of a test by relating it to a metric. For example, if our hypothesis states that an instance under our load balancer in auto scaling group is terminated, it should have no impact on the system. “No impact” is measured by checking the metrics. Number of failed requests for orders should be zero, the response time of the service falls in the expected range etc. Metrics will also help to identify when to abort the experiments if there is a larger impact.

4.2.4 Notify teams

It is important to discuss and notify the teams which handle the service about the experiments going to be carried out so that they are prepared to respond. This is required at initial stages of adapting Chaos Engineering. Once all teams are used to these types of experiments, teams gain confidence and they will start incorporating correct measures to withstand these tests in the development phase and they would start perceiving these exercises as normal events. If experiments are run in production, we should inform support and community staff which may start receiving questions from customers.

4.2.5 Run the experiment

At this step, we start executing our experiments and start watching metrics for abnormal behavior. We should have abort and rollback plan in place in case the experiment causes too much system damage or impact real customers. Alert mechanisms should be in place to act upon if there is too much variation in the critical metrics and handle customer impact. Our experiments should simulate real-world events. Experimenting and fixing events such as CPU exhaustion, Memory overload, network latency and failure, functional bugs, significant fluctuations in the input, retry storms, race conditions, dependency failures, and failures in communication between services can increase confidence in the reliability of the system.

4.2.6 Analyze the results

Analyze the metrics once the experiment is complete. Verify if the hypothesis was correct or there was a change to the system’s steady state behavior. Identity, whether there was any adverse effect on customers or the system, was resilient to the failures injected. Share result of the experiment with the service teams so that they can fix them. The harder it is to deviate from the system behavior with the experiments, the higher the trust in the reliability of the system. Look at recurring patterns in design or issues that need to be addressed and encourage teams to test for resilience early in the product lifecycle.

4.2.7 Increase the scope

As our experiments with smaller inputs and scope start succeeding, we can start increasing the scope. For example, smaller tests can target one host at a time under a load balancer. Start scaling it to take down two or more in further experiments. to test for resilience earlier in their lifecycle.

4.2.8 Automate

Initial experiments can be started with the manual process as it is safer to run manual tests than ruining a buggy script with unknown outcomes. As we gain confidence it the system we should start automating the tests so that we can save time and execute tests frequently.

4.3 Sample inputs for experiments

Experiments vary depending on the architecture of the systems being built. However, in a distributed system and microservices architecture deployed on cloud following are the most common experiments that can be automated with open source and commercial tools.

- Turning off instances randomly in availability zone (or data center)
- Simulating the failure of an entire region or availability zone.
- Resource exhaustion: Exhausting CPU and Memory on instances.
- Partially deleting stream of records/messages over a variety of instances to recreate an issue that occurred in production.
- Injecting latency between services for a select percentage of traffic over a predetermined period.
- Function-based chaos (runtime injection): randomly causing functions to throw exceptions.
- Code insertion: Adding instructions to the target program and allowing fault injection to occur prior to certain instructions.

The opportunities for chaos experiments are boundless and may vary based on the architecture of your distributed system and your organization's core business value.

5 How we got started

In this section, we will focus on a few applications which our teams are responsible for and the chaos experiments carried out to find weaknesses.

5.1 Identity Service

Application: Identity service is one of the key services which is part of our Management Platform. This acts as central service to provide Authentication and authorization to all other services so that each application need not build its own authentication service. This service generates OAuth based identity and access tokens. This service becomes critical service as the other services depend on the token generated by this. Thus, it was crucial for us to make sure this service is always available and resilient to failures.

Architecture: To make this service highly available, we chose to deploy this on AWS and make use of managed services to reduce the operating cost.

1. Stateless application servers containing microservice were deployed on EC2 instances in 3 availability zones (AZ-#).
2. Application servers are placed under Elastic Load Balancer.
3. Auto-scaling enabled with Minimum one system in each availability zone and auto-scaling rules set to scale based on a number of parameters.
4. Cold stand-by set up in a secondary region to failover in case of region failure.
5. 6 node Cassandra cluster across 3 availability zones with Quorum consistency.

Deployments: Following is a typical deployment model for high availability in AWS

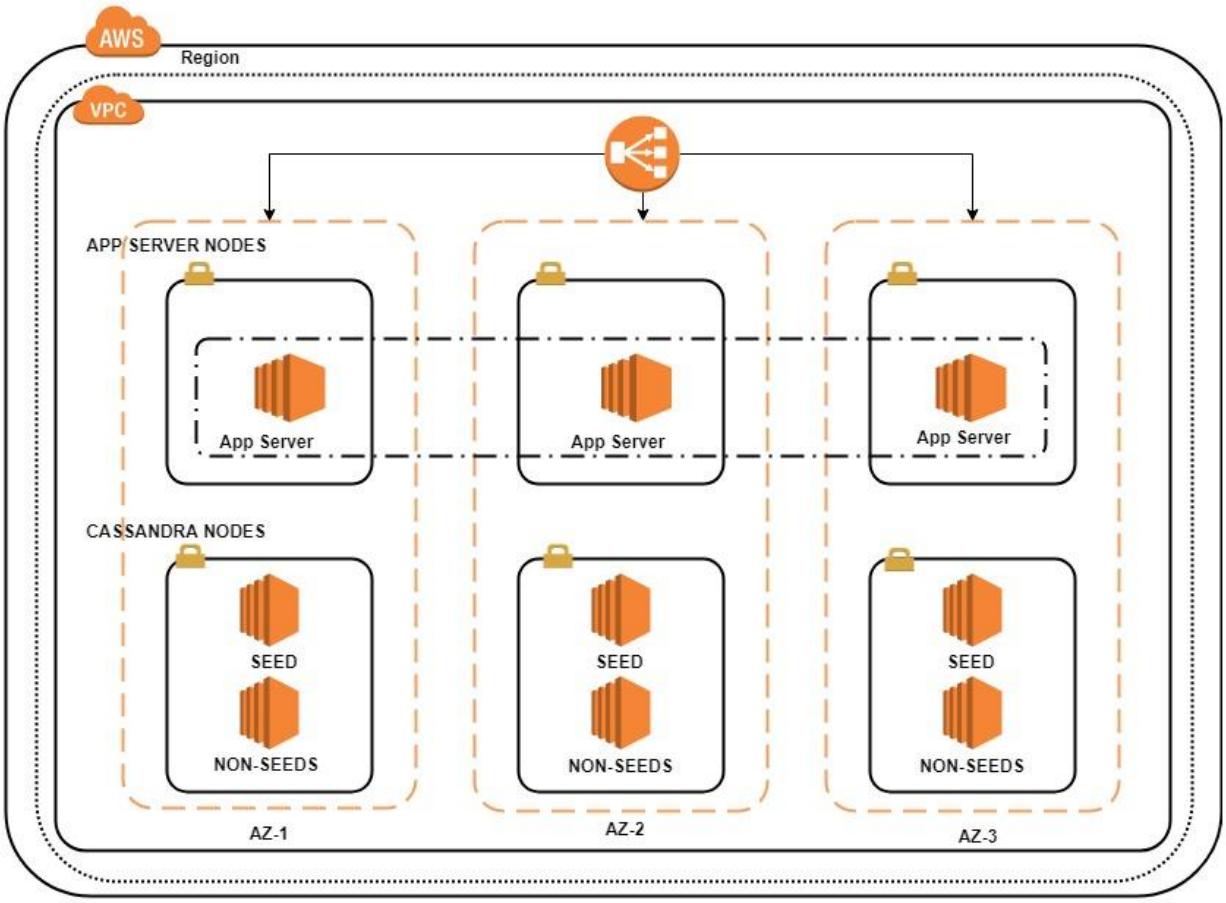

Figure 3 AWS Deployment Diagram

5.1.1 Experiment – Host failures

Target: Identity Service

Experiment Type: EC2 instance failure

Steady State: All requests to token endpoint (API serving tokens) return tokens with no failures

Hypothesis: If a single instance of EC2 hosting the identity service application fails, rest of the services should be able to handle the request. There will be no impact on the tokens generated. A new EC2 instance should come up with the application deployed on it and join the load balancer. While the new instance is coming, we might see an increase in response time for the requests.

Blast radius: Contained to a Single instance

Metrics: Kibana dashboard showing number of failed requests to token endpoints

Simulation: During the initial days of the experiment we started this in manual mode. The manual test was as simple as terminating one of the EC2 instances from AWS console. Our API test automation suite (Gatling tests) was continuously sending requests to token endpoint throughout the duration of the experiment.

Results and Findings:

When the EC2 instance was shut down, auto-scaling group configuration and scaling policies determined the change and started the process of launching a new instance. Requests coming in during this duration were handled by other two instances and we did not see any failed requests, which proved our hypothesis correct. As you can notice in the below screenshot from Kibana dashboard, all requests results returned 200 and none of the requests returned response error codes such as 500.

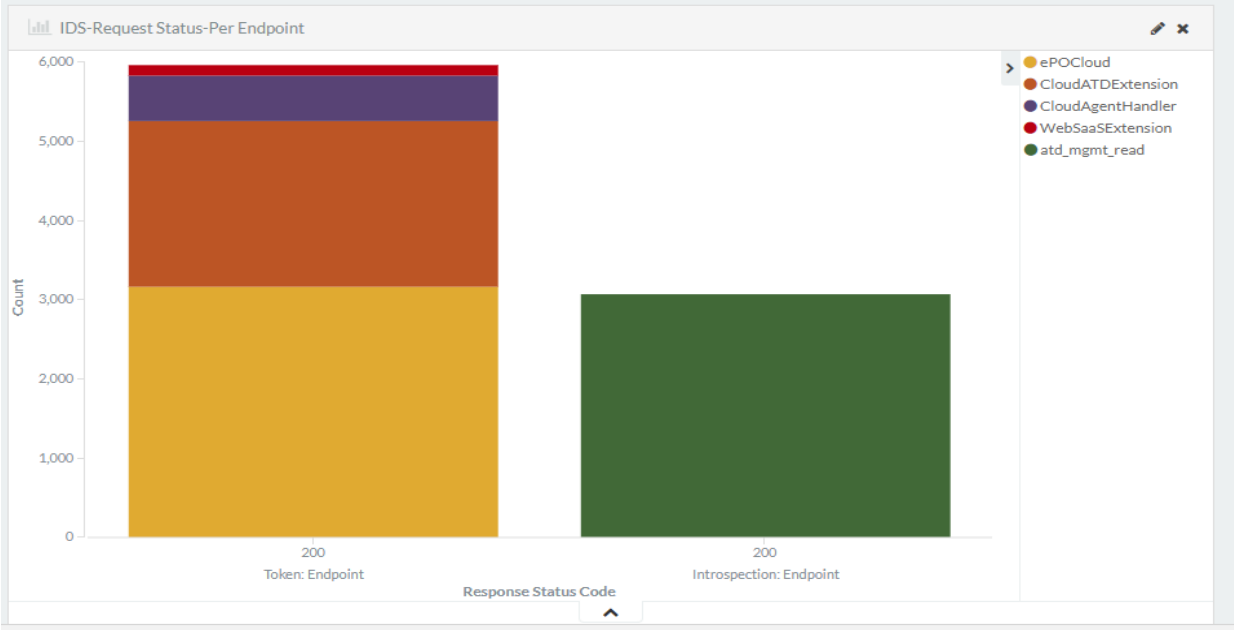

Figure 4 Steady State of requests from Kibana

It can be observed from the below reports that more than 2000 requests show response times greater than 800 ms. This increase in response time was observed at the time when one instance was terminated.

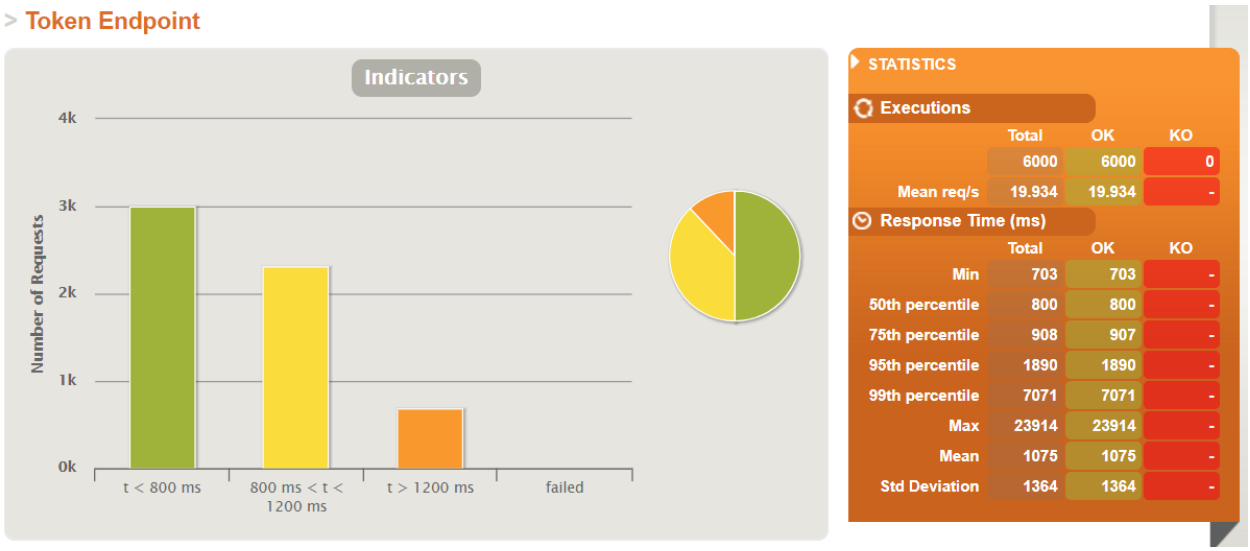

Figure 5 Status of requests

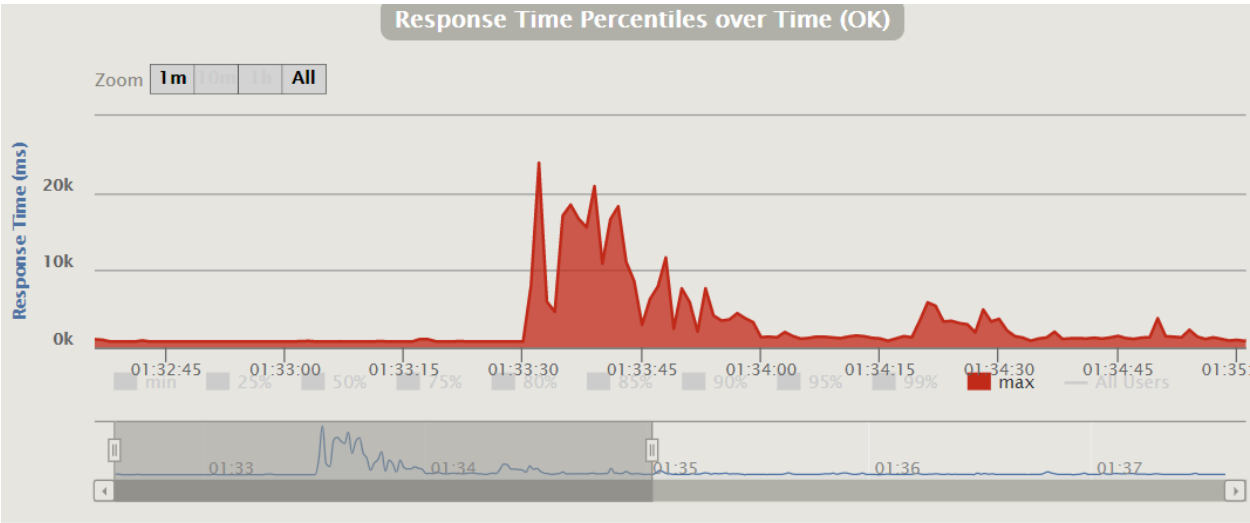

Figure 6 Increase in response time with a single machine

Once we gained confidence in such manual tests, we moved on to use Chaos Monkey for terminating random instances in the setup. Once the experiments are stable we started scaling up the scenario by increasing the number of instances that fail and repeat the experiment.

5.1.2 Experiment 2

Target: Cassandra DB Nodes

Experiment Type: Un-reachable DB nodes and DB node restarts

Steady State: All requests to token endpoint (API serving tokens) return tokens with no failures

Hypothesis: Our Cassandra data store runs with a setup of 3 nodes. Key-space is initialized with replication-factor of 3. Read and write operations are processed on a QUORUM consistency level, i.e saving a new record will succeed only if at least two of the 3 nodes receive the new record. A select query will return record only if at least two of the three nodes return the record. If there is a problem with one of the nodes and it is not reachable by other DB nodes or the application server, service should continue to work as the consistency level is set to QUORUM.

Blast radius: Single Cassandra DB node

Metrics: Kibana dashboard showing number of failed requests to token endpoints

Simulation: We started this experiment in manual mode by blocking access to Cassandra Port – 7000 through AWS security groups. This made the node unable to communicate with other nodes in the ring. API test tool was continuously sending token requests.

Results and Findings: With QUORUM consistency set, the application can perform queries on DB without error when one node is not available. No failures were observed on the dashboard which proved our hypothesis.

Automation: We started using a tool like Blockade and Muxy to simulate these failures. Blockade is a utility for testing network failures and partitions in distributed applications. Blockade uses Docker containers to run application processes and manages the network from the host system to create various failure scenarios.

5.2 Order Management System

Order management system is a collection of multiple microservices facilitating customer orders for Endpoint solutions. This solution consists of a collection of REST APIs, built as microservices communicating with each other over HTTP. Order Data Service and Product Catalog services are two among other microservices in this system. Order Data service makes a call to Product catalog service to fetch information about Product SKU being placed.

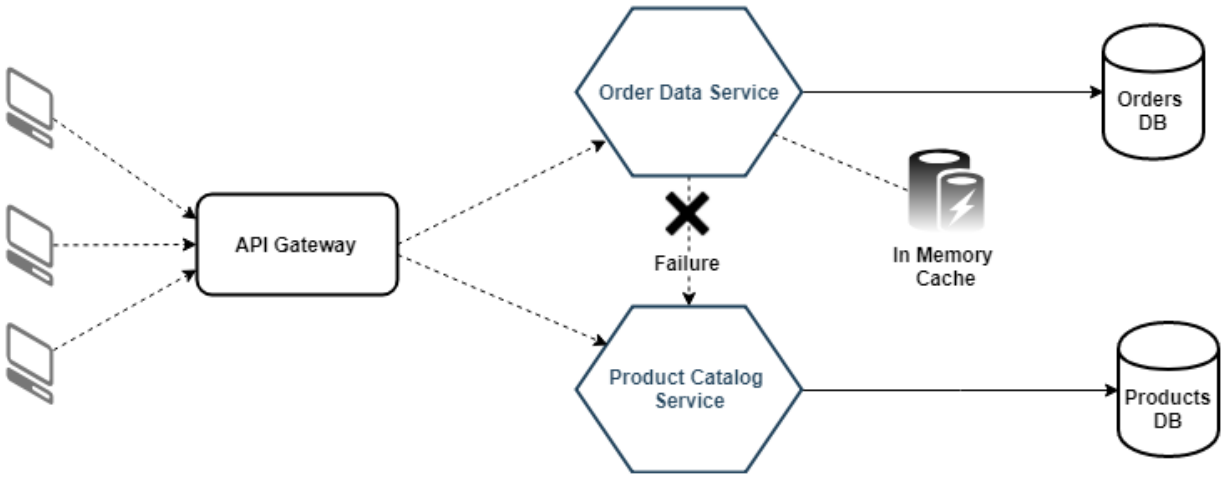

Figure 7 Services interaction

5.2.1 Experiment – Network Failure, Retry Storms

Target: Product Catalog Service

Experiment Type: Network Failure, Retry Storms

Steady State: All orders received by order data service are successfully accepted for processing.

Hypothesis: If product catalog service goes down because of a node failure, or order data service is not able to connect to product catalog service, a retry mechanism will try again for 3 times. If 3 retries fail, order data service should call In-memory cache to get details of the product which is being placed.

Blast radius: Contained to product catalog service

Metrics: Dashboard showing number of orders failed to process

Simulation: To begin with, we took manual steps. We blocked access to port 8443 on the instance where product catalog service was hosted. Test tools started sending requests to order data service. In later stages, we used Blockade and Latency Monkey for this experiment.

Results: When the number of requests were low, order data service was able to handle the retries and return the cache for product catalog response after 3 retries. However, as the number the requests started increasing we started seeing an interesting failure scenario. Our CPU utilization on the order data service started spiking up and requests started to fail.

Findings: So, what went wrong with the retry mechanism? This unavailability or failure to connect is not due to any problem in the service, but due to some reasons like network failure or server overload. Such issues are ephemeral. If we call the service again, chances are that our call will succeed.

Such failures are called transient failures. Simple retry mechanisms work when transient failures are of short duration. However, if the duration is longer, our application will end up retrying, waiting and retrying till the maximum retry count is hit and thus wasting resources. So, we should have some mechanism to identify long-lasting transient faults and let this retry mechanism not come into play when we have identified the fault as a long-lasting transient fault.

Fixing: Circuit Breaker pattern is a better approach to handle the long-lasting transient faults. In this pattern, we wrap the service calls in a circuit breaker. When the number of retry failures reaches above a certain threshold value, we change the status of the circuit to OPEN, which means the service is not available at the time. Once the circuit reaches the OPEN state, further calls to the service will immediately return failure to the caller instead of executing retry logic. This circuit breaker will have some timeout period after which it will move to HALF-OPEN state. In this state, it will allow a service call which will determine if the service has become available or not. If the service is not available, it goes back to OPEN state. If the service becomes available after this timeout, the circuit breaker moves to the CLOSED state. The callers will be able to call the service and the usual retry mechanism will come into play again.

This helped in reducing all the retry execution in case the fault is long lasting and thus saving resource and providing more immediate feedback to the order data service.

6 Benefits of Chaos Engineering

1. **Customer**:

    - Benefits of chaos engineering at the top level can be measured by overall system availability. Companies pioneered in Chaos Engineering like Netflix, Amazon defines their availability for most of the services in terms of 4 to 6 9s or even more. Four nines availability mean that a system is available 99.99% of the time (or less than 1 hour a year). Translating these numbers into actual outage time can indicate why availability matters the most. Frequent chaos experiments make systems more resilient to failures and increase these numbers which makes happy customers.

2. **Business:**

    - Can help prevent extremely large losses in revenue and maintenance costs. Outages can cost companies millions of dollars in revenue depending on the usage of the system and the duration of the outage (Chaos Engineering : Breaking Your Systems for Fun and Profit 2017).
    - Increases confidence in carrying out disaster recovery methods. Most teams do not have enough confidence in full-scale disaster recovery as they perform these tasks only in extreme disaster cases. If the whole system has adopted principles of chaos engineering, disaster recovery practices can be performed more often to gain confidence.

3. **Technical**:

    - The insights from chaos experiments can mean a reduction in incidents.
    - Helps teams to see how systems behave on failure and prove that their hypothesis is valid
    - Chaos Engineering also tests the human involvement in the system like the response of engineers to the incidents. It can verify when an outage occurs if the right alert was raised and the appropriate engineer was notified to take appropriate action.
    - Increases engagement of engineers to make applications highly reliable as they put more effort to make application reliable to sustain the chaos tests from the early stages of the development.

7 Tools & Techniques

Following are various open source and commercial products available for performing chaos experiments:

1. The Netflix Simian Army (The Netflix Simian Army 2011): Simian army consists of several services in the cloud for generating various kinds of failure.
    a. Chaos Monkey: Randomly terminates virtual machine instances and containers that run inside of your environment. The service operates at a controlled time (does not run on weekends and holidays) and interval (only operates during business hours).
    b. Chaos Kong: Simulates AWS region failure by injecting failures for all resources in a region. Best suited for disaster recovery testing to determine Recovery Time Objective (RTO) & Recovery Point Object
2. Chaos toolkit: This tool kit aims to be the simplest and easiest way to explore building your own Chaos experiments. Built as a vendor and technology independent way of specifying chaos experiments by providing Open API. https://docs.chaostoolkit.org/
3. Kube-Monkey: This is an implementation of Chaos Monkey for Kubernetes cluster. It randomly deletes Kubernetes pods to experiment with the resiliency of cluster. kube-monkey runs at a pre-configured hour on weekdays and builds a schedule of deployments that will face a random Pod failure sometime during the same day.
4. Pumba : Chaos testing and network emulation tool for Docker containers and clusters.
5. Gremlin; Gremlin offers a failure-as-a-service tool to make chaos engineering easier to deploy. It includes a variety of safeguards built in to break infrastructure responsibly
6. Chaos lambda: Chaos Lambda increases the rate at which EC2 instance failures occur during business hours, helping teams to build services that handle them gracefully. Every time the lambda triggers it examines all the Auto Scaling Groups in the region and potentially terminates one instance in each. The probability of termination can be changed at the ASG level with a tag, and at a global level with the Default Probability stack parameter.
7. Chaos monkey for spring boot: This project provides a Chaos Monkey for Spring Boot application and will try to attack your running Spring Boot App.
8. Blockade: Blockade is a utility for testing network failures and partitions in distributed applications. Blockade uses Docker containers to run application processes and manages the network from the host system to create various failure scenarios. A common use is to run a distributed application such as a database or cluster and create network partitions, then observe the behavior of the nodes. For example, in a leader election system, you could partition the leader away from the other nodes and ensure that the leader steps down and that another node emerges as leader.
9. WireMock : API mocking (Service Virtualization) as a service which enables modeling real-world faults and delays.
10. ChaoSlingr : ChaoSlingr is a Security Chaos Engineering Tool focused primarily on the experimentation on AWS Infrastructure to bring system security weaknesses to the forefront.
11. Muxy: Muxy is a proxy that mucks with your system and application context, operating at Layers 4, 5 and 7, allowing you to simulate common failure scenarios from the perspective of an application under test; such as an API or a web application. https://github.com/mefellows/muxy

8 Best Practices

Many organizations are reluctant to break the systems on purpose considering the risks involved as the experiments are run in production. We can follow some of the best practices that have evolved from the experience of people in the chaos community to mitigate the risks.

- **Minimize the blast radius:** Begin with small experiments to learn about unknowns. Only when you gain confidence, scale out the experiments. Start with a single instance, container or microservice to reduce the potential side effects.

- **Build v/s Buy:** Perform a study of on the tools available. Compare features available and time and effort required to build your own tools. Do not pick tools which perform random experiments as it would become difficult to measure the outcome. Use tools which perform thoughtful, planned, controlled, safe and secure experiments.

- **Start in the staging environment:** To be safe and get confidence in tests, start with staging or development environment. Once the tests in these environments are completely successful, move up to production.

- **Prioritizing Experiments:** Chaos experiments while running in production can have an impact on core business functionality. Several factors need to be considered while prioritizing the experiments.
    - Categories your services as Tier1 (critical) and Tier 2(Non-Critical). This can be determined by factors such as the percentage of traffic a service receives, ability function with or without a fallback path etc. Start experiments with Tier 2 services to verify if the unavailability of these services is handled gracefully and core business functionalities are not affected. If an attack on Tier 2 service brings the system down, then this service need to be badged as a critical service.
    - If your system is resilient to instance failure, start those tests firsts and then move on to experiments like latency injection and network failures which may require special tooling.
    - Among the categories chosen above, prioritize the experiments which can be executed safely without causing business impact.
- **Be ready to abort and revert:** Make sure you have done enough work to stop any experiment immediately and revert the system back to a normal state. If experiment by any chance causes a severe outage, track it carefully and do an analysis to avoid it happening again.

- **Calculate cost and Return on Investments**: Business impact of the outages can vary based on the nature of the business. Impact on revenue in case of outages can be calculated by number of incidents, outages, and their severity. Compare this with the cost of chaos experiments and other alternate ways of reducing outages to arrive at a conclusion.

- **Don’t do the experiments if the problem is known to exist:** Brainstorm with the team to understand systems known weaknesses. If you know that an experiment can result in failures or cause outage to the customer, do not perform it. Fix the known issues and then experiment. Perform your experiments in the following order.

    1. Known Knowns - Things you are aware of and understand
    2. Known Unknowns - Things you are aware of but don’t fully understand
    3. Unknown Knowns - Things you understand but are not aware of
    4. Unknown Unknowns - Things you are neither aware of nor fully understand

9 Limitations

1. Unplanned, careless chaos can cause damage to applications and impact the customer experience.
2. Executing large-scale experiments in the cloud can become expensive
3. Failures in abort conditions and strategy can lead to downtime.

10 Conclusion

In today's interconnected, internet-based world, no system is safe from system failure. Most of the Organization's experience with the cloud and the distributed system has not been without glitches and some of them are major. Infrastructure can be volatile and can fail in ways dissimilar to the ones we had learned to expect using a data center. Apart from Infrastructure failures, APIs and services in large micros services ecosystem can fail for several reasons and it is important to make this system resilient and highly available. The key takeaway is that we need to plan and design around failures. Resiliency must be baked into our designs and plans.

One of the best ways for the failures to not impact customers, employees, partners, and your reputation, is to proactively find it and address it before they occur. Chaos engineering is one of the optimal and cost-effective ways to do this. Organizations that build distributed internet applications should consider chaos engineering as part of their Resiliency strategy.

The opportunities for chaos experiments are boundless and may vary based on the architecture of your distributed system and your organization's core business value. Cost of deploying chaos experiments can be significantly lesser than hiring several system admins to fix problems when they occur. Running chaos experiments on a regular basis is one of many things you can do to begin measuring the resiliency of your APIs. Making sure you have good visibility (monitoring) and increasing your fallback coverage will all help strengthen your own systems.

References

Casey, Rosenthal, Hochstein Lorin, Blohowiak Aaron, Jones Nora, and Basiri Ali. 2017. *Chaos Engineering. Building Confidence in System Behavior through Experiments.* O'REILLY.

2017. "Chaos Engineering : Breaking Your Systems for Fun and Profit." *Gremlin.* 12. https://www.gremlin.com/media/20171210%20%E2%80%93%20Chaos%20Engineering%20White%20Paper.pdf.

2018. *Chaos Engineering: the history, principles, and practice.* 06. https://www.gremlin.com/community/tutorials/chaos-engineering-the-history-principles-and-practice/.

2017. *Netflix Technology Blog - Chaos Automation Platform.* https://medium.com/netflix-techblog.

2018. *Principles of Chaos Engineering.* 05. http://principlesofchaos.org/.

Ratis, Pavlos. 2017. *awesome-chaos-engineering.* https://github.com/dastergon/awesome-chaos-engineering.

2011. *The Netflix Simian Army.* 07. https://medium.com/netflix-techblog/the-netflix-simian-army-16e57fbab116.