Increasing resilience and improving customer experience by using chaos engineering on AWS

AWS Prescriptive Guidance

AWS Prescriptive Guidance: Increasing resilience and improving customer experience by using chaos engineering on AWS

Table of Contents

Increasing resilience and improving customer experience by using chaos engineering on AWS

Laurent Domb, Amazon Web Services

Chaos engineering is the discipline of experimenting on an application in order to build confidence in your organization's and application's capability to withstand turbulent conditions in production. It is a proactive approach to resilience, with the goal to verify if your application and organization are able to absorb, adapt to, and eventually recover from service impairments by introducing controlled failures across people, processes, and technology. The intent is also to identify and eliminate weaknesses before they can cause outages or other disruptions in production.

At Amazon, we understand that failure is inevitable in distributed systems, to the point that functioning despite the presence of failures is a normal mode of operation. Because interactions between services are bound to fail, you need to understand how your services react during various failure modes and build services that are resilient to key vulnerabilities such as dependency failures, retry storms, impaired Availability Zones, and host resource exhaustion.

Let's take the example of a retry storm. A localized failure in a client can impact multiple services significantly. This is commonly referred as the *butterfly effect*. A *retry storm* is a manifestation of the butterfly effect where a failing dependency triggers clients, and clients of those clients, to retry the failed operation, leading to an exponential growth in traffic. Services become overloaded because they must respond to regular traffic in addition to retry traffic while handling a degradation in performance.

Chaos engineering has emerged as a response to the increasing complexity of distributed systems. It is a multidisciplinary approach that combines principles from chaos theory, systems thinking, and engineering to design and manage complex systems that are resilient to unexpected events and behaviors. At its core, chaos engineering is concerned with understanding and managing the behavior of complex systems under conditions of uncertainty and unpredictability. It recognizes that traditional approaches to engineering, which rely on predicting and controlling outcomes, are often insufficient for dealing with the complex and dynamic nature of distributed systems. As these systems grow, they often exceed the scope of understanding of any single individual.

Chaos engineering provides concepts, techniques, and tools to intentionally inject failures into systems to uncover weaknesses before they manifest in production. This proactive approach allows organizations to build confidence that their systems will perform under stressful conditions.

Although chaos engineering is still an evolving practice, it represents a fundamental shift toward designing, managing, and operating modern computing systems to be resilient in the face of increasing complexity and interconnectedness.

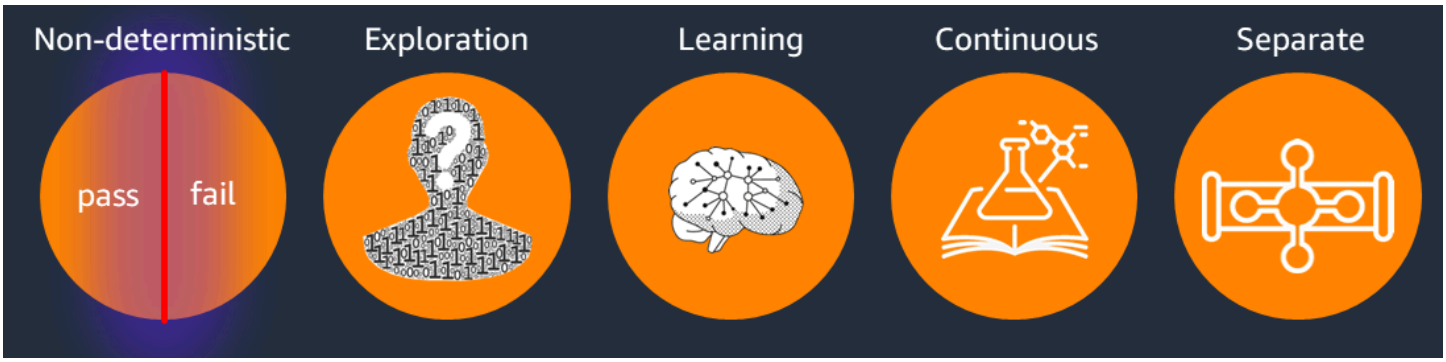
The following sections of this guide discuss the benefits of chaos engineering, explain how to conduct chaos engineering experiments, and describe the approaches you can take to implement chaos engineering at scale in your organization. Also included are sample experiment planning and experiment result documents that you can use as templates for your chaos engineering experiments.

The next section explores how the characteristics of chaos engineering differ from traditional resilience testing such as unit, smoke, or integration tests.

Overview

Comparing resilience testing with chaos engineering

Resilience testing is deterministic. That is, it validates known characteristics about resilience mechanisms, such as circuit breakers, retries, failovers or fallbacks, that have been implemented in your application. It confirms how these application components absorb controlled disruptions with minimal to no user impact. Therefore, resilience testing focuses on the validation of known failure modes that are injected into application components with the goal of producing pass/fail results. You should run resilience testing continuously as a step in your pipeline to ensure that you don't introduce regressions to your resilience posture. In resilience testing, you often do not run tests against real components, but mock APIs that simulate a certain component. This approach allows for consistent, reproducible testing of failure scenarios in a controlled environment, making it suitable for automated pipeline integration and regression testing.



In contrast, chaos engineering is non-deterministic. That is, it is hypothesis-based and verifies your mental model on how the application and its dependencies (people, process, and technology) absorb, adapt to, and eventually recover from unanticipated failure modes. Therefore, chaos engineering focuses on the end-to-end verification of unknown failure modes, with the goal of catching defects early, and remediating these before they turn into large-scale events. Chaos engineering fosters continuous learning and should be practiced through a separate chaos pipeline or ad-hoc experiments that enable you to run multiple experiments at any point in time without blocking your developer's productivity in deploying code.

The chaos engineering process often begins with a chaos game day, which is a dedicated event where teams intentionally inject controlled faults or failures into their application. The game day is progressive: It starts in lower-level environments (such as development or testing) and gradually advances to higher-level environments (such as staging and pre-production) as confidence builds. By systematically moving through these environments, teams can verify that their systems tolerate the injected faults properly before they reach production. This methodical progression ensures that by the time chaos experiments are conducted in production environments, teams have built substantial confidence in their system's resilience capabilities. The game day process is a proactive approach to identifying weaknesses and vulnerabilities in an application's architecture and operational practices, while eliminating the stress of learning during an unexpected production outage.

The value of chaos engineering

Complex systems are ubiquitous in today's world. They play a critical role in many aspects of our lives, from financial markets to healthcare. We expect these systems to be always operational. However, complex systems are often vulnerable to unexpected events and behaviors that can have significant consequences. Organizations need to plan for disruption instead of wondering whether it will happen. They can do that by applying scenario testing across their critical or mission-critical business services. This is where chaos engineering comes into play.

Chaos engineering offers an approach to manage complex systems that can help mitigate risks and improve resilience. The process of preparing for chaos experiments requires teams to develop hypotheses about their system's behavior, which deepens their understanding of how systems are built and how they operate. This preparation often reveals mental gaps, architectural insights, and operational knowledge that might otherwise remain undiscovered. By furthering the understanding of how complex systems react to failures, chaos engineering promotes greater transparency and accountability in system design and management. The more frequently your

organization practices chaos engineering, the better prepared they become operationally. Chaos engineering helps you establish best practices for designing resilient applications that can survive component failures with minimal to no user impact. This ensures that critical applications operate within expected service levels and impact tolerance, while continuously enhancing the team's knowledge of their own systems.

Preparing for adverse conditions

When you build on AWS, you use different types of services, including zonal services such as Amazon Elastic Compute Cloud (Amazon EC2), Regional services such as Amazon Simple Storage Service (Amazon S3), global services such as AWS Identity and Access Management (IAM), third-party software as a service (SaaS) services, and on-premises services. Each type of service exposes different failure domains that you need to account for. How do you prepare for self-inflicted events, or events that are caused by third parties that your organization has no control over?

To help understand how your application might respond to adverse conditions, you can use AWS Fault Injection Service (AWS FIS). AWS FIS is a fully managed service for running fault injection experiments in a controlled way. You can use this service to inject AWS-provided scenarios such as Availability Zone power interruptions and cross-Region connectivity issues, or build your own experiments by chaining together a wide variety of fault actions that are provided by the service. AWS FIS enables your teams to continuously practice and learn how their application would react to common faults and remediate defects as they detect them.

Practicing controlled chaos engineering

The key principles of controlled chaos experiments are:

- Start with an environment that's similar to your production environment.
- Establish a hypothesis and stop conditions for your experiment.
- Start small.
- Exercise control over your chaos experiments.
- Set the scope of impact.
- Know your service baseline.
- Schedule experiments.
- Remediate first, and then experiment.

- Monitor the experiment closely.
- Learn from your results.
- Prioritize findings, remediate, and verify.
- Propagate the learnings across your organization.

To successfully scale chaos engineering, you must implement chaos experiments in a controlled way. When you use AWS FIS, you can create stop conditions by using Amazon CloudWatch alarms. You can incorporate these conditions into an experiment template to ensure that experiments are stopped if out of bounds and rolled back to their last known state. AWS FIS also provides safety levers. When you engage these levers, AWS FIS stops and rolls back all running experiments in the account in the AWS Region, including multi-account experiments, and prevents new experiments from starting. This prevents fault injection during certain time periods, such as trading hours, sales events, or product launches, or in response to application health alarms. The safety lever remains engaged until it's manually disengaged.

When you conduct a chaos experiment, you should define safeguards to prevent undesirable side-effects in the environment, especially if there is a possibility that the experiment will affect applications that are in production. When you plan the experiment, anticipate any adverse effects it might have on other applications in the environment. For example, other applications could receive erroneous messages from the application that is part of the experiment, experience high request volumes, or encounter resource contention if they share infrastructure. Document these risks and address any known or unacceptable issues before you run the experiment.

Getting started with chaos engineering

Before you conduct an experiment, we recommend that you put a few essentials in place to make the most of your chaos engineering practices. These essentials include:

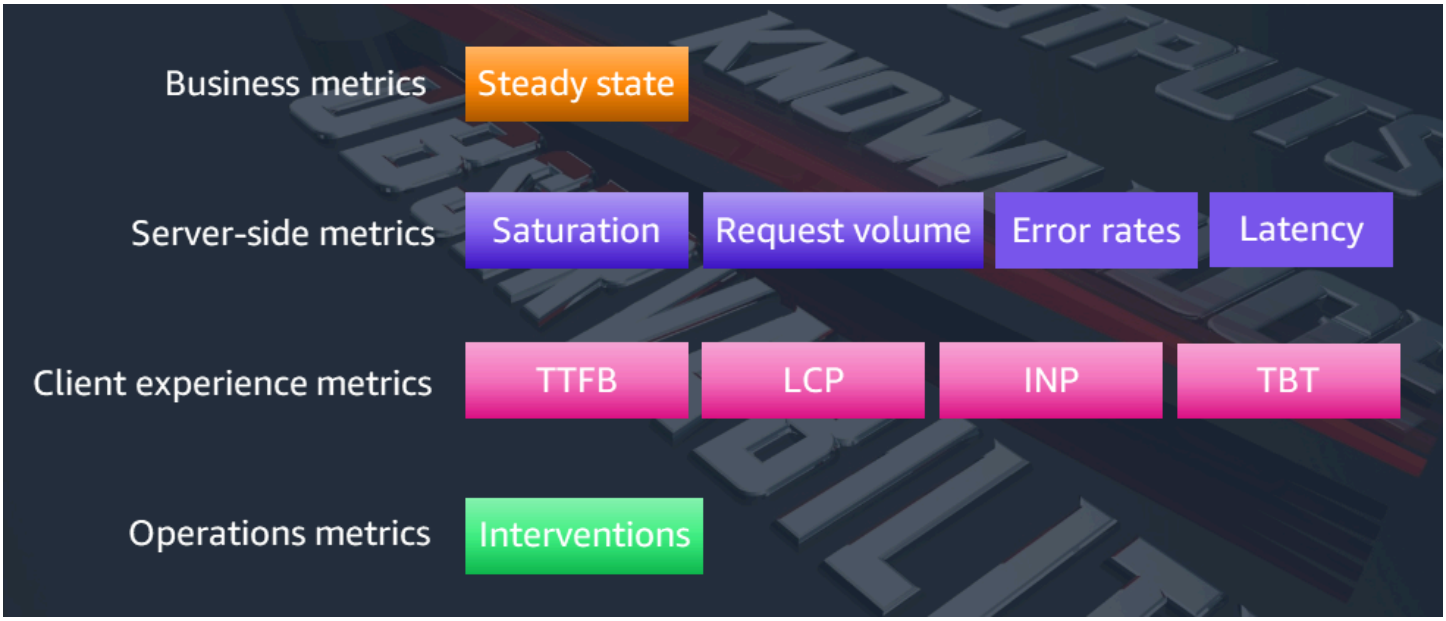
- Observability (metrics, logging, request tracing)
- A list of real-world events or faults that you would like to explore
- Organizational resilience sponsorship through leadership buy-in
- Prioritization of critical findings based on potential business impact over new features that are discovered when running chaos experiments

Observability for chaos experiments

Observability, which comprises metrics, logging, and request tracing, plays a key role in chaos engineering. You will want to understand business metrics, server-side metrics, client experience metrics, and operations metrics when you run an experiment. Without observability, you won't be able to define steady-state behavior or create a meaningful experiment to verify if your hypothesis about your application holds true.

Metrics

The following diagram shows the types of metrics that you can track for chaos experiments for different types of applications.



- **Business metrics** – *Steady state* indicates the normal operation of your system and is defined by your business metrics. It can be represented by transactions per second (TPS), click streams per second, orders per second, or a similar measurement. Your application exhibits steady state when it is operating as expected. Therefore, verify that your application is healthy before you run experiments. Steady state doesn't necessarily mean that there will be no impact to the application when a fault occurs, because a percentage of faults could be within acceptable limits. The steady state is your baseline. For example, the steady state of a payments system might be defined as the processing of 300 TPS with a success rate of 99 percent and round-trip time of 500 ms. Visually, think of steady state as an electrocardiogram (EKG). If the steady state of your system suddenly fluctuates, you know that there is a problem with your service.
- **Server-side metrics** – To understand how your resources perform during the experiment, you need insights into their performance before, during, and after the experiment. To measure the impact of your resources on AWS, you can use Amazon CloudWatch. CloudWatchis a service that monitors applications, responds to performance changes, optimizes resource use, and provides insights into operational health. During your experiments, you will want to capture server-side metrics such as saturation, request volumes, error rates, and latency.
- **Customer experience metrics** – On AWS, you can capture real user metrics by using CloudWatch RUM to simulate user requests through tools such as Locust, Grafana k6, Selenium, or Puppeteer. Real user metrics are crucial for organizations that conduct chaos engineering experiments. By monitoring how real users are impacted during an experiment, teams can get an accurate picture of how faults and disruptions will affect customers in production. Key client experience metrics

are Time to First Byte (TTFB), Largest Contentful Paint (LCP), Interaction to Next Paint (INP), and Total Blocking Time (TBT).

- **Operations metrics** – Interventions measure how successfully you mitigate faults in an automated way—for example, successful client request latency during a reboot of pods, tasks, or EC2 instances with mechanisms such as replication controller or automatic scaling. Being able to automatically intervene during a fault directly correlates with a good user experience. Understanding if there is any drift in these mitigation mechanisms over time is crucial. By defining metrics for both successful and failed automated mitigations, you create guideposts that help identify potential regressions throughout your system.

Logging

Centralized logging is key to understanding your application's components' states before, during, and after a chaos experiment. We recommend that you collect logs from all your application components to build a consolidated view of what each component was doing at the time the experiment was injected. This provides a clear picture of the end-to-end experiment flow.

Request tracing

Request tracing enables you to observe the flow of any single request across the components in your application to gain a comprehensive understanding of the impact that the injected failure has on the system and its dependencies. By tracing the requests, you can see how the failure propagates through different services and components, so you can better assess the scope of the disruption. To trace your requests on AWS, you can use AWS X-Ray.

Failure scenarios to inject in chaos experiments

The goal of injecting common faults into your application is to understand how the application reacts to these unexpected events, so you can create mitigation mechanisms and make your system resilient to such faults. Additionally, you should use chaos engineering to replay historical failure scenarios to verify that your mitigation mechanisms are still functioning as expected and did not drift over time.

Consider the following events when you plan your chaos engineering experiments.

Failure mode	Description

Server impairment	Reboot EC2 instances, delete Kubernetes pods or Amazon Elastic Container Service (Amazon ECS) tasks to understand how your application reacts to such crashes.
API errors	Inject faults into AWS and your own service APIs to understand application behavior.
Network issues	Introduce latency or congestion, or block connections to mimic real-world network problems.
AWS Availability Zone impairment	Replay the impairment of an entire Availability Zone to verify recovery across zones.
AWS Region connectivity impairment	Replay a network impairment across AWS Regions to verify how resources in the secondary Region react to such an event.
Database failures	Fail over database replicas or corrupt data, or make database instances unreachable, to understand impact to your application and recovery strategies.
Pause in database and Amazon S3 replication	Pause database or Amazon Simple Storage Service (Amazon S3) replication across Availability Zones or AWS Regions to understand downstream application impact.
Storage degradation	Pause I/O, remove Amazon Elastic Block Store (Amazon EBS) volumes, or delete files to verify data durability and recovery.
Dependency impairment	Take down or degrade the performance of the downstream or upstream services that you depend on, including third-party services, to understand the end-to-end flow and impact to your customers.

Traffic surges	Generate spikes in user traffic to test automatic scaling capabilities, and see how cold boot time might impact your overall application state.
Resource exhaustion	Max out CPU, memory, and disk space to verify the graceful degradation of your application.
Cascading failures	Initiate primary failures that cascade to downstream applications and components.
Bad deployments	Roll out problematic changes or configura tions to verify rollback mechanisms.

Organizational resilience sponsorship

Chaos engineering provides the most value when it's applied across your organization. We recommend that you work with an executive sponsor who can help set resilience goals across your organization, remove the fear, uncertainty, and doubt about the domain, and start the transformation process to make resilience everyone's responsibility.

To support the business case of building a chaos engineering practice, attach chaos engineering efforts to your critical business projects. Making resilience an asset and driver for acceleration will help you track success over time. Start with a count of critical incidents per month or per quarter, the average time to recover, and the impact that these incidents caused to your customers and organization. Set a goal with your teams to reduce the number of incidents over a 6-month to 12-month period as improvements are made across your application stacks in response to chaos engineering experiments.

Measure whether incidents are highly repetitive. For example, let's say an expired TLS certificate leads to downtime because clients cannot establish a trusted connection. If multiple incidents occur in a year because of multiple TLS certificate expirations, you can run an experiment of a TLS certificate expiration and verify that your teams get alerts or are able to automatically mitigate such issues. This will help ensure that you become resilient to such faults.

To track progress in chaos engineering over time, capture the following metrics to help highlight the value of chaos engineering across an application's lifecycle:

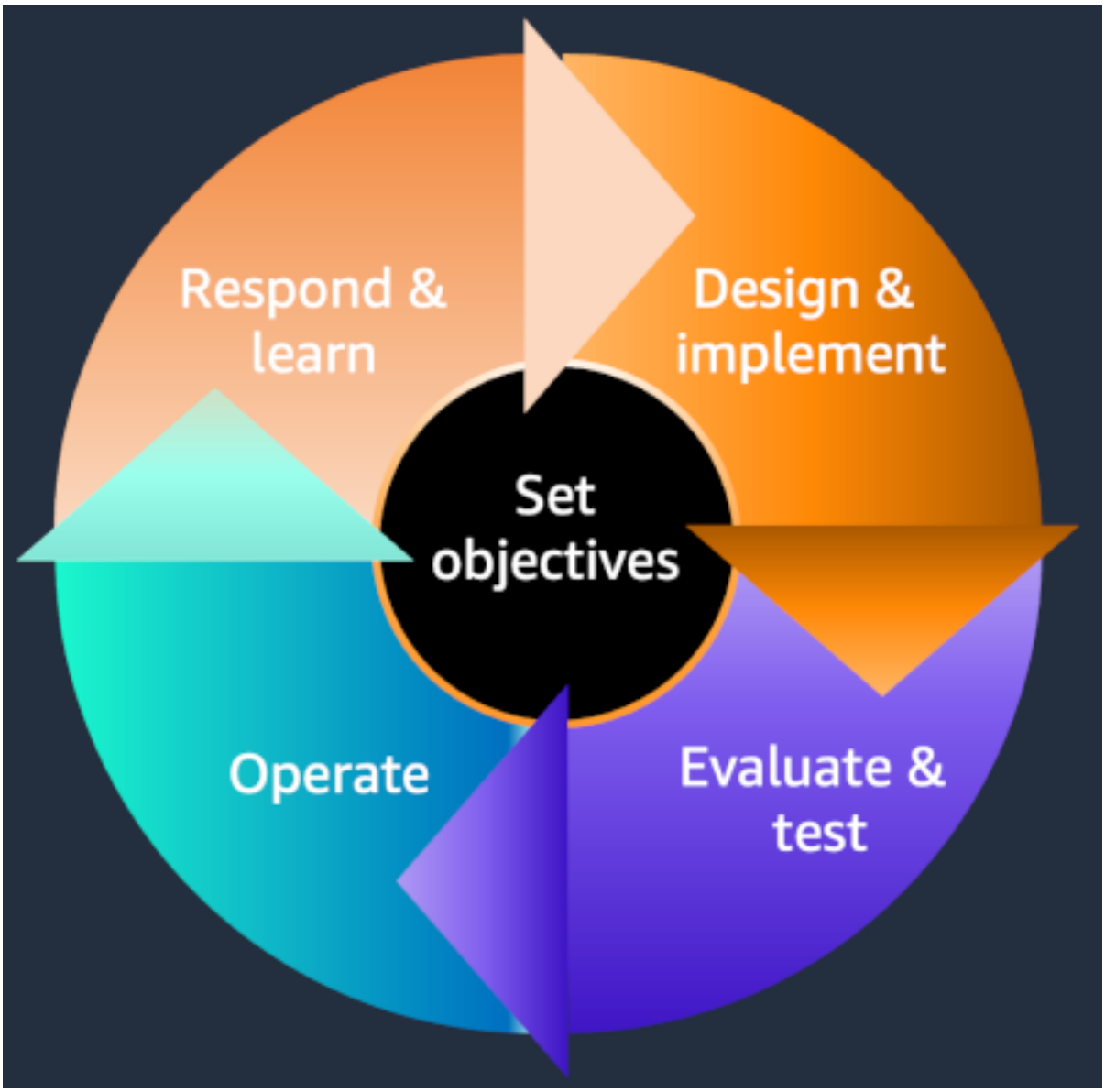
- **Reduced incident rate** – Track the number of production incidents over time and correlate this number with the adoption of chaos engineering. The expectation is that the rate of severe incidents will decline.
- **Improved mean time to resolution (MTTR)** – Calculate the average time it takes to resolve incidents and track this data to see if it improves with chaos engineering over time.
- **Increased application availability** – Use service-level metrics to show availability improvements as application resilience increases through chaos experiments.
- **Faster time to market** – Chaos engineering can provide the confidence to launch innovative offerings faster, because you know that your applications are resilient. Track increases in product release velocity.
- **Operational cost reduction** – Quantify if indicators such as alert noise, operational load, and manual effort to manage applications decrease with chaos practices in place.
- **Boosting confidence** – Survey developers, site reliability engineers (SREs), and other technical staff to gauge if chaos engineering boosted their confidence in application resilience. Perceptions matter.
- **Improved customer experiences** – Connect chaos engineering to positive outcomes for customers, such as fewer service disruptions, rollbacks, and outages.

Prioritizing remediation

As you perform chaos experiments, you are likely to identify areas for improvement where the application does not perform as intended. Remediation of such items will become work in your backlog that will have to be prioritized along with other work such as feature development. We recommend that you make time for these enhancements to avoid future failure. Consider prioritizing these learnings and remediation tasks based on the level of impact they might cause. Findings that directly impact the resilience or security of your application should have priority over new features, to avoid customer impact. If the team struggles to prioritize remediation work over feature development, consider reaching out to your executive sponsor to ensure that priorities are set based on business risk tolerance.

Implementing chaos engineering on AWS

Chaos engineering is part of the evaluate and test stage of the AWS resilience lifecycle, as illustrated in the following diagram. Distributed applications do not operate in isolation from other applications or clients, so we recommend that you review the entire resilience lifecycle. Change is constant for distributed applications as the network evolves, upstream and downstream applications undergo shifts, and client usage changes over time.



To understand how these changes to your application might impact its resilience, make chaos engineering a part of your day-to-day operations. You can implement chaos experiments in different ways:

- **Ad hoc** – You can perform chaos experiments as one-time experiments to address a specific issue or question.
- **Chaos game days** – These are structured and recurring events that are designed to verify the reliability and resilience of an application. The purpose of a chaos game day is to identify potential resilience issues or deficiencies across people, processes, and technologies, and to practice the processes and procedures for identifying, mitigating, and responding to incidents.
- **Chaos pipeline** - Continuous integration and continuous delivery (CI/CD) is about building new features and deploying them safely throughout the environments. To implement chaos engineering experiments, create a chaos pipeline that's separate from your CI/CD pipeline. To understand why, let's assume that you want to add a single chaos engineering experiment to your CI/CD pipeline that injects increasing packet loss for downstream components. That experiment runs 3 times and takes 5 minutes to finish each time. Packet loss increases from 10 percent to 20 percent to 30 percent with each run, and the experiment takes 15 minutes overall to complete. If you have 100 parallel deployments, you'll have to wait 1500 minutes for a single experiment to complete. If you have 10 experiments to run, the impact to your developers would be unbearable. At scale, chaos engineering needs its own pipeline that allows you to run experiments in parallel to your software development lifecycle (SDLC) process.
- **Canary deployments** – Canaries provide a testing environment for chaos experiments. By directing a small percentage of traffic to a canary service or using methods such as traffic mirroring or replay, you can verify new infrastructure or code changes with zero impact to your stable production system. You can run experiments against the canary and inject faults as necessary, because you can limit the scope of impact to the end-user.
- **Scheduled experiments** – You can schedule experiments to verify predictable recovery mechanisms for your application. Use scheduled experiments to replay commonly known events to capture how your systems can recover from events such as terminating an EC2 instance behind an automatic scaling group, removing a Kubernetes pod, or deleting an Amazon ECS task.

Continuous chaos engineering experiment lifecycle

As discussed in the previous section, you can implement chaos engineering experiments in different ways. In all cases, the key to building a meaningful chaos experiment is to understand the application, historical incidents, and implemented remediations, and to clearly understand the areas to investigate, such as resilience or security. Your knowledge about the application helps you formulate a hypothesis on the application's potential weaknesses and understand how it will detect, remediate, and recover when the fault is injected.

The chaos experiment lifecycle includes these steps:

1. Define the objective of the experiment.

2. Select the target application.

3. Align mental maps.

4. Address the known issues with your application.

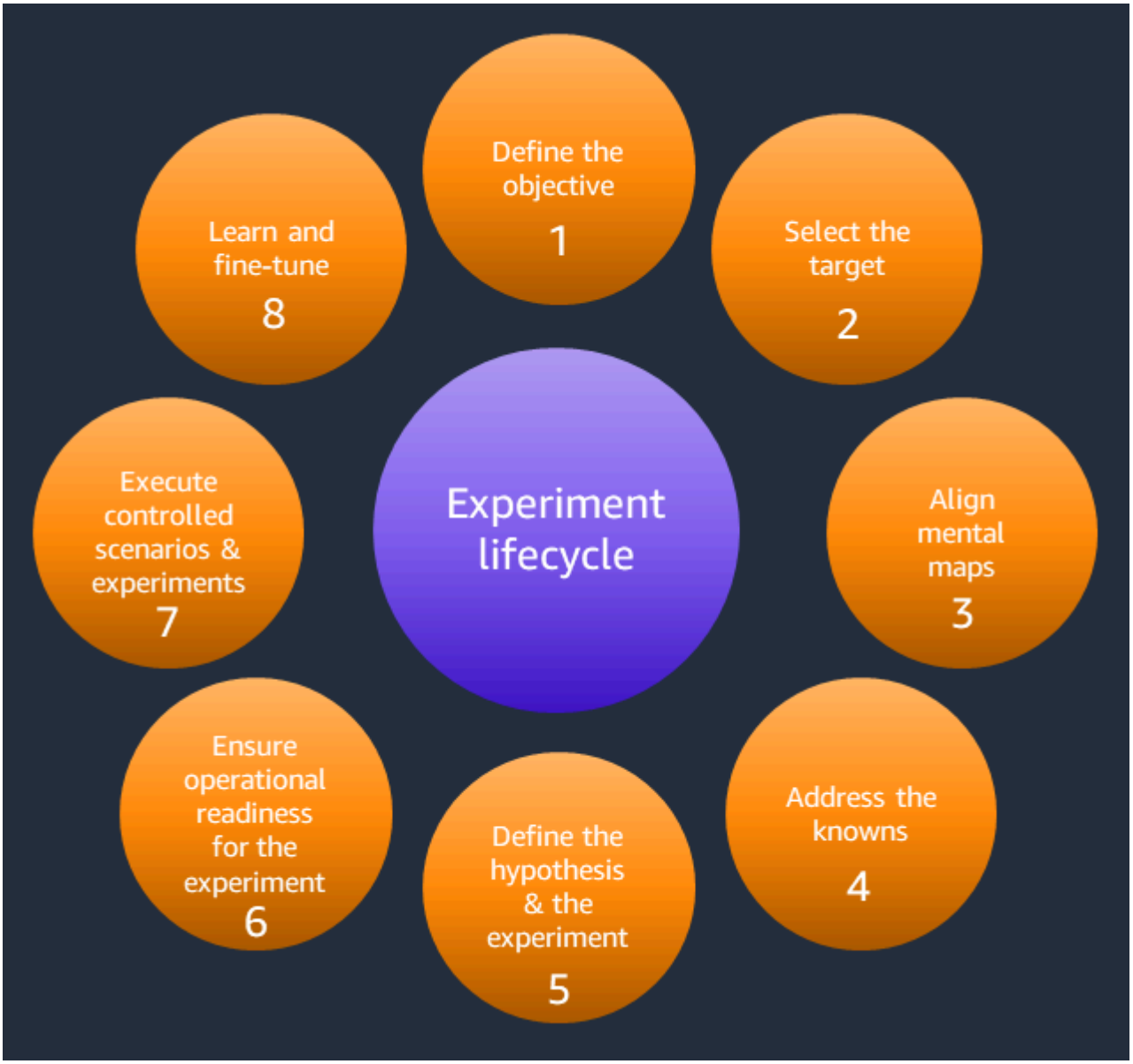
5. Define the hypothesis and the experiment.

6. Ensure operational readiness for the experiment.

7. Run controlled scenarios and experiments.

8. Learn from and fine-tune the experiment.

These steps are illustrated in the diagram and discussed in the following sections.



Define objectives and set expectations

Before each experiment, make sure that your objectives and expectations are specific, measurable, achievable, relevant, and time-bound. Clearly define the following:

- **Identify** potential failures or weaknesses in systems and services, to understand how they might impact users. This includes identifying possible failure modes, such as server crashes, network failures, or software bugs, and assessing their potential impact on the system's overall performance and reliability.

- **Quantify** the impact of failures by defining key risk indicators (KRIs) on your systems and services. This includes measuring the effect of failures when metrics such as latency, throughput, and error rates deviate from their steady state. By understanding the impact of such deviations, you can prioritize efforts to mitigate failures based on business risks.
- **Develop** and **verify** strategies for mitigating or preventing failures. This includes identifying potential solutions, such as redundancy, error correction, or fallback strategies, and testing their effectiveness in a controlled environment. By verifying these strategies, you can ensure that you are effective in preventing or mitigating failures, and can deploy them in your production systems with confidence.
- **Improve** incident response and disaster recovery processes. By replaying failures in a controlled environment, you can test incident response processes, identify potential bottlenecks or gaps, and refine disaster recovery procedures. This helps ensure that you are prepared to respond quickly and effectively in the event of unexpected failures.

Select the target application

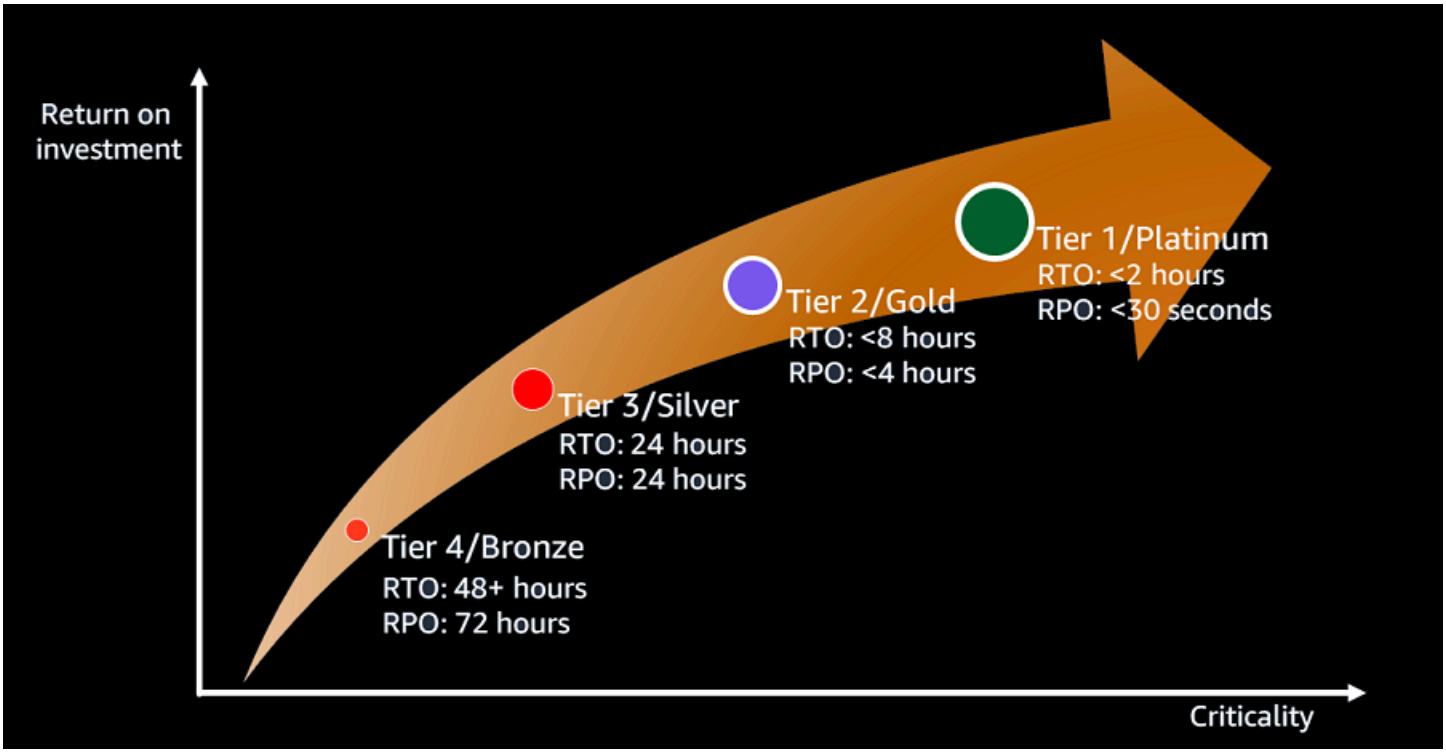
Chaos engineering is a powerful technique but requires thoughtful prioritization to maximize value. When deciding where to focus chaos engineering efforts, start by considering your business's critical services. Ask your teams to iterate through the software development lifecycle stages, and start to inject faults in testing environments first. Business-critical applications are directly tied to revenue, customer experience, and core operations. Chaos experiments on these services can uncover vulnerabilities that can severely impact the organization—and potentially entire markets—if they aren't addressed. For example, focus on customer-facing services such as trading systems or order systems first. Prioritizing these central services provides the most protection per investment of time.

After critical services, look at foundational components such as databases, message queues, networks, and shared services APIs. These might be used as shared components or services across your organization, so their failure will cause widespread problems. Confirming the resilience of infrastructure services provides confidence that they won't cripple the dependent applications above them. For example, a chaos engineering experiment that takes down a Kafka cluster reveals a lot about the fault tolerance of downstream applications. Although system infrastructure isn't directly customer-facing, it is a prime chaos engineering target.

Don't forget to map the mental gaps of people, processes, facilities information and third-party dependencies, because these can cause major disruptions if they aren't aligned with your

organization's impact tolerance objectives. For more information about measuring the ROI of chaos engineering, read Quantifying the ROI of chaos engineering in the strategy document *Investing in chaos engineering as a strategic necessity*.

The following diagram shows the return on investment for running chaos experiments on different tiers of services.



Align mental maps (application discovery)

When you run ad-hoc experiments or game days, you will begin the application discovery process by holding a whiteboarding session that focuses on mapping out the details of your application. (If you run the experiments in the chaos pipeline, you will have already aligned that mental map, by defining the target application.) A good approach to understanding mental gaps is to have the most junior team member draw a diagram of the application first, and then ask more senior staff members to add to the diagram progressively. This will uncover any gaps in understanding across experience levels.

The diagram should depict both direct upstream and downstream dependencies of the application, as well as any critical third-party integrations. Make sure that there is alignment on the expected flow of a request through the application. Map out the key workflows and user journeys to gain

clarity on how customers use the application. Consider using a sequence diagram to capture this information.

After this collaborative session, the team should have a shared mental model of the application, its critical dependencies, and its monitoring capabilities, and an understanding of the risks to make an informed decision to proceed with, or cancel, a potential chaos experiment.

Address the known issues with your application

Chaos engineering experiments are designed to proactively surface defects in an application. By injecting failures such as latency increases, server reboots, or Availability Zone power impairments, you can verify your application's ability to tolerate realistic disruptions. However, this process assumes an underlying level of stability and health in the target application. Running chaos experiments on an already problematic application risks masking deeper issues.

Before undertaking chaos engineering, teams should resolve any known defects, bugs, and performance problems in their application.

Define the hypothesis and the experiment

Past incidents that have caused disruptions to your application or other applications within your organization can serve as excellent sources for chaos experiment ideas. For example, were previous outages triggered by configuration errors or missing resilience patterns? Reviewing incident histories and replaying the root causes of those real-world failures through chaos experiments is an effective way to develop resilience against similar issues in the future.

Another valuable source of experiment concepts can come directly from the engineers, architects, and operators who are most familiar with an application. Allowing team members to submit hypothetical failure scenarios that they believe could significantly disrupt the application enables you to collect ideas based on insider knowledge. The application team can then evaluate which of these proposed scenarios might have the largest potential impact or expose the biggest unknown risks. Targeting chaos experiments for such high-risk, lesser understood scenarios can generate important learnings and prevent problems in the future.

A third source of ideas comes from performing resilience modeling to anticipate the conditions that would lead to identified business losses. Some resilience modeling exercises have a component-based approach to building a resilience model whereas others have a systems-based

approach. A component-based approach asks the question, "What happens when component *x* is under extreme load or has failed?" The team that develops the resilience model then speculates the effect of such a scenario on the wider application, and identifies the monitoring and preventative controls currently in place to detect and mitigate the effects of the scenario. Alternatively, a systems-based approach follows a top-down process to highlight an undesirable state of the application—such as, "The web storefront is showing stale inventory levels"—and invites the application team to anticipate which condition or conditions would cause the application to behave in this way.

Ensure operational readiness for the experiment

You need quantifiable indicators to measure the impact of adverse conditions on the application and its behavior, as described previously in the section on observability. Being able to measure the application's behavior enables you to determine whether the adverse conditions impacted the application and to what magnitude.

The best way to understand whether there is an impact to your application is to measure its steady state. Steady state measures what normal operation looks like and typically aligns with the business and client experience indicators for a given application. Before you move to the next step, make sure that you have the observability in place to understand impact, and rollback mechanisms ready in case the experiment doesn't turn out as expected.

Run controlled experiments and scenarios

At AWS, we do not recommend conducting an initial chaos experiment on an application that is in production. The purpose of a chaos experiment is to learn something new about how the application behaves under stressful conditions. The application's behavior might be unpredictable during the experiment, so performing an experiment for the first time in production could have customer-impacting consequences. Therefore, you should always run an initial chaos experiment in a lower-level environment that has minimal potential for affecting real-world users, and then iterate through your environments after you verify and are confident that your application can absorb, adapt to, and recover from the injected actions.

Plan each experiment thoroughly by using a document that captures key details, similar to the experiment planning document provided in the appendix. Some of the critical fields to include are the steady state definition, hypothesis, and method of failure injection. The planning, execution, and analysis of a chaos experiment can be covered in a single artifact.

After you finalize the written plan for the experiment, prepare any necessary code to inject the planned disruptions that are outlined in the document.

To capture potential impact during the experiment, make sure that observability mechanisms are in place. If you do not yet have an automated way to capture experiment outcomes, such as AWS FIS experiment reports, identify the team members who will take notes during execution, capture screenshots of dashboards, and lead the team through the experiment.

Learn and fine-tune

After each experiment, get together as a team to review and reflect on the chaos experiment. Make a conscious effort to maintain a blameless mindset. Your goal should be to have an open, constructive dialogue that focuses entirely on deriving maximum learning, not assigning blame.

Start by reviewing the steady state definition and hypothesis for the experiment. Did the application behave as expected? Were there any surprises that invalidated assumptions? Discuss observations of how the application reacted during the experiment, both good and bad. The data collected—metrics, logs, screenshots, and so on—should tell the story of exactly what happened.

Approach this data review with curiosity instead of judgment, and identify areas where improvements can be made to application design, documentation, monitoring, or other capabilities based on the learnings. These action items are captured as follow-up projects to make the application more resilient.

Through this blameless approach, you can have candid conversations about what went wrong and how you can fix it. Assume positive intent from everyone who is involved, and trust that they were working toward good outcomes. Your shared goal is organizational growth and progression through continuous learning and adapting. Chaos experiment reviews that are conducted in a constructive, blameless manner provide a safe space for your team to gain valuable insights that make your applications and organization more reliable and resilient in the long term. The focus stays on the learnings, not the people. To spread the learnings across your teams, publish the experiment report in a central place and advertise the findings so that others can learn from it.

Scaling chaos engineering across your organization

As your organization adopts chaos engineering, standardizing and implementing it will present challenges. In the early stages of maturity, different teams are likely to use different tooling and variations of the chaos engineering process described in the previous sections. At the same time, some teams might not prioritize or adopt chaos engineering at all, despite its potential benefits. The following sections provide guidance on how to overcome these challenges.

Overall, your approach to chaos engineering should be designed to strike a balance between centralized leadership and decentralized participation. This balance helps ensure that chaos engineering is integrated into the development process and that learnings are shared across your organization.

Establishing a chaos engineering practice

Standardizing the practice of chaos engineering can accelerate its adoption. Sharing the learnings from experiments across teams can magnify the return on chaos engineering investments.

Build a centralized center of excellence, or assemble a group of subject matter experts, as part of your chaos engineering practice. As a small, centralized function, this team can function across software development, infrastructure, security, and business teams and maintain standards that are used by those teams. For simplicity, the center of excellence is called the *centralized practice team*, and groups that apply chaos engineering are called *practicing teams* in the remainder of this guide.

Role of the centralized practice team

The centralized practice team is responsible for developing and implementing chaos engineering practices across the organization. They work closely with practicing teams to guide them in designing and conducting experiments, and ensuring that the experiments are valuable to the business. The centralized practice team also provides guidance and support to the development, infrastructure, and security teams to help them integrate chaos engineering into their development processes.

The key responsibilities of a centralized chaos engineering practice team include the following:

- **Enablement** – A centralized chaos engineering function acts as a facilitator to introduce the practice of chaos engineering through game days and workshops. They guide teams in the

process of chaos engineering, including selecting failure scenarios, defining hypotheses, and producing reports to be shared with the wider organization. The centralized practice team should own training materials and work to upskill the practicing teams in their use of chaos engineering.

- **Advisory** – The centralized practice team can also act in an advisory role to oversee experiments that are conducted by the practicing teams. Their experience and knowledge can ensure that experiments deliver value to the business and are conducted in a safe manner. Similarly, the team can oversee the execution and debrief of an experiment to guide people who are new to chaos engineering.
- **Marketing and value tracking** – Communicating the business value of chaos engineering is key to the success of such a program. Each team that participates in chaos engineering experiments should collect data from the experiments across the business and demonstrate the value of the organization's investment into chaos engineering. This includes quantifying and celebrating the number of incidents that were avoided during each experiment, the downtime that would have been incurred if the experiment had failed, and the overall impact to the business if the failure scenarios had occurred in production. By gathering and centralizing such data from across the teams, and making the data available across the organization, the centralized practice team can track and influence the value derived from the adoption of chaos engineering throughout the organization.
- **Standards** – The centralized practice team should own and maintain the process for conducting chaos experiments, the templates for planning and reporting on experiments, and the tooling used to conduct experiments.

  The central team should own and manage experiment planning templates, experiment report templates, process documentation, and enablement materials. Best practice documentation and enablement materials provide guidance to practicing teams on topics such as the guardrails they can use to limit the impact of an experiment, when to conduct an experiment in production, and how to evolve their use of chaos engineering over time. For examples of templates and outputs, see the appendix.

  The centralized practice team should also own the process for conducting an experiment, including communications and escalation, and when and how to communicate with other teams in the organization before or during an experiment. The process should also outline when guardrails are required.

  The centralized practice team should also select and own the core tools for conducting chaos experiments (for example, tools such as AWS FIS). The selection and implementation of

supplementary tools, such as load generation tools, should be left to the practicing teams to decide. Practicing teams should be able to adapt the overall process and tooling to best suit their needs.

Role of the practicing teams

The centralized team is responsible for driving the overall chaos engineering strategy, whereas the practicing teams participate in the process and own the development and execution of experiments. This helps to ensure that the experiments are relevant to each specific product or service, and that the learnings are actionable and can be applied to improve the product's reliability and resilience. The centralized practice team acts as a mentor and owner of the organization's chaos engineering standards and process. However, in order to prevent the centralized team from becoming a bottleneck, individual practicing teams will need to learn from the central practice to perform chaos experiments for themselves.

Establishing a community of practice

In addition to creating a centralized team, we recommend that you establish an informal community of practitioners who are interested in chaos engineering. This community provides a platform for sharing knowledge, best practices, and experiences across practicing teams and the wider organization.

The community of practice can be operated by the centralized chaos engineering practice team, but anyone within the organization can become a member of the community. The centralized team can leverage the community of practice to broadcast updates and source learnings, and to collect feedback from practicing teams who are using the standards and process managed by the centralized team. The community will act as a feedback loop to inform the centralized team of the effectiveness of chaos engineering practices across the practicing teams. The centralized practice team can then adjust their documentation and supporting artifacts to best support the product teams.

Incorporating chaos engineering into your operational resilience

A chaos experiment is an investment by your business to prevent incidents in production. It will be necessary to determine where the business can realize the greatest return on this investment. The

organization can work with the centralized chaos engineering practice team to update its standards and determine which products are critical enough to require chaos experimentation.

Systems development process

Chaos engineering and chaos experiments should be performed repeatedly as part of an application's lifecycle. Similar to how teams regularly perform disaster recovery tests, they should conduct chaos experiments and game days continuously and periodically throughout the year. This approach improves how an organization anticipates, observes, and responds to incidents.

Conclusion

The discipline of chaos engineering has come a long way over the last decade. It has been adopted across a variety of industries, and has helped organizations create resilient services and increase customer satisfaction. (For examples of how organizations have implemented these practices, see Chaos Engineering Stories.) Chaos engineering is enabling organizations to reduce risks for their mission-critical applications by injecting controlled faults at all levels of their application stack, including cloud provider services. Having the capability to impact an entire application stack in a controlled way allows for continuous resilience, improvement in operational excellence, observability, and recovery-oriented architecture without the stress of production outages. This approach also leads to better resilience testing practices across the organization. To start embracing chaos engineering, run a chaos game day or workshop to showcase the value that chaos engineering can provide to your organization.

Resources

- Investing in chaos engineering as a strategic necessity
- Chaos Engineering Stories
- Amazon CloudWatch documentation
- Amazon CloudWatch RUM documentation
- AWS X-Ray documentation
- AWS FIS documentation
- AWS FIS failure model implementations:
  - Use AWS Fault Injection Service to demonstrate multi-region and multi-AZ application resilience (AWS blog post)
  - AWS Fault Injection Simulator supports chaos engineering experiments on Amazon EKS Pods (AWS blog post)
  - Announcing AWS Fault Injection Simulator new features for Amazon ECS workloads (AWS blog post)
  - Improve application resiliency with Amazon EBS volume metrics and AWS Fault Injection Simulator (AWS blog post)
  - Chaos engineering leveraging AWS Fault Injection Simulator in a multi-account AWS environment (AWS blog post)
  - Chaos experiments on Amazon RDS using AWS Fault Injection Simulator (AWS blog post)
  - Building resilient serverless applications using chaos engineering (AWS blog post)
  - Use FIS to interrupt a spot instance (AWS workshop)
  - Automating Chaos Engineering in Your Delivery Pipelines (AWS Community post)

Appendix: Sample documents

The sample documents provided in this section use a fictitious pet adoption site (PetSite) as a target application for a chaos experiment.

Experiment planning document

Steady state

Process name	Pet adoption site
Physical architecture	(Link to architecture diagram.)
Logical architecture	(Link to logical diagram.)
Define steady state	Average page load time, measured by using Largest Contentful Paint (LCP), for the pet adoption site is 2.5 seconds or less with a 99 percentile latency (P99) of 4.0 seconds or less with a baseline of 5000 concurrent users.
Steady state metrics	LCP metric captured across user base, and golden metrics (latency, throughput, error rates, saturation).
Steady state observability	LCP will be captured by the user's browser, sent to Amazon CloudWatch, and inspected with CloudWatch RUM. Over a 60 second period, the average and P99 LCP time will be aggregated for all requests in that period. Top-level golden metrics are captured by using CloudWatch.
Process to achieve steady state	Grafana K6 will be used to create a load that simulates normal production traffic levels of approximately 5K concurrent users.

Observability requirements

Teams should be able to view the following:

- **Steady state**: What will be observed to verify that the application is under normal conditions?
- **Failure condition**: How will the failure condition appear in the dashboard? For example:
  - Alarms that should be triggered
  - Logs that should be generated
- **Failure impact**: What should be observed to view components that are expected to be impacted (scope of impact)?
- **Recovery**: How will the recovery be viewed and measured to capture MTTR?
- **Debug**: Troubleshooting details on experiment failures.

The following table provides suggestions and examples for an observability requirements chart. You should define what should be observed based on your specific experiment.

What needs to be observed	Link to observability tool	What is being observed
Source of input	Grafana K6 dashboard	• Running container count • Requests per second
Overall application health	• Pet adoption CloudWatch dashboard • Pet adoption user experienc e dashboard (RUM)	• Amazon EKS healthy node count • Amazon EKS node CPU utilization
Workflow health	Pet adoption CloudWatch dashboard	LCP time, golden metrics
Traces	Pet adoption X-Ray dashboard	• Request latency • Request count

		• Failure count
Logs	Pet adoption CloudWatch Logs	Any errors encountered by the pods will be issued to CloudWatch Logs.

Experiment definition

Experiment name	Amazon EKS PetSite pod CPU stress
Experiment source code	(Link to experiment source repository.)
Experiment description	This experiment explores how an increase in CPU usage of the PetSite application pod would impact overall customer experience. By injecting CPU stress into each running PetSite pod, we will be able to understand if there is impact to customers and the extent of that impact.
Experiment requirements or parameters	Application load: Production average Pod label selector: `app=petsite`
Experiment duration	10 minutes
Environment	Alpha test environment
Experiment target resources	PetSite application pods
Experiment baseline that is introduced through the load generating tool	• 54% of requests have an LCP of <2.5 seconds. • 46% of requests have an LCP of <4 seconds. • No errors are observed.
Backoff condition	None

Hypothesis

What if	Impact	Recovery
What would happen to steady state if the PetSite application pods experienced or caused more than 60% CPU utilizati on for 10 minutes under normal production-level traffic?	LCP times will remain under 2.5 seconds for P50 of users with P99 of 4.0 seconds or less. The consumer should be able to load the PetSite landing page.	Detection: • CPU stress will be detected by alarms that are configured in CloudWatch. • LCP metrics will also generate alarms for the degradation of user experience. Self-healing: • The distributed nature of the microservice architect ure means that many instances of pods are running across multiple Availability Zones. • The EKS cluster control plane will shift traffic away from the affected pods, and will launch new pods on worker nodes. Recovery: When CPU utilization returns to normal, the LCP should recover automatically.

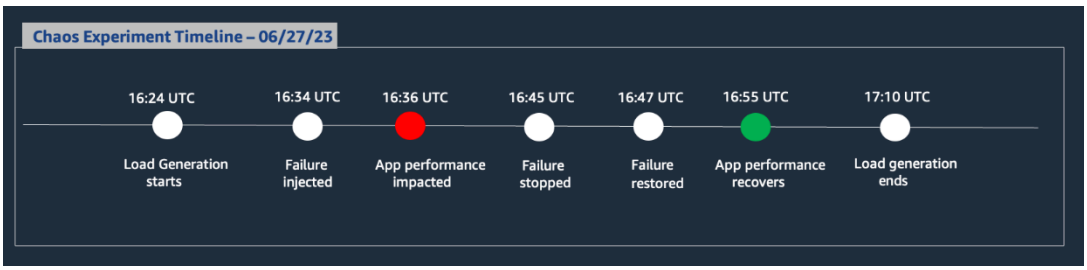
Experiment process

Tailor the following example step-by-step process to your specific experiment:

1. Validate access to, and functionality of, all Amazon CloudWatch, CloudWatch RUM, and AWS X-Ray dashboards.
2. Validate the health of the application environment:
   a. Confirm that the EKS cluster is healthy by using the CloudWatch dashboard.
   b. Visit the test pet adoption site application deployment at the example URL.
3. Initiate a load to achieve steady state:
   a. Confirm that the load generator is running and sending 5000 requests per second.
   b. Allow 5 minutes for the application to reach its steady state.
   c. Confirm the steady state of the application by using the CloudWatch RUM dashboard.
4. Initiate a fault (experiment):
   a. Open the AWS FIS console.
   b. Run the pet-adoption-pod-stress experiment.
   c. Confirm that the experiment is running in the console.
5. Observe the impact of the fault on your application:
   a. Capture screenshots from the CloudWatch RUM and CloudWatch dashboards, and note any anomalous data points.
   b. After the experiment has completed in AWS FIS, capture additional screenshots to record if the application returns to steady state in the absence of stress, and note any anomalies in the data points.
   c. If the steady state doesn't resume, take steps to recover the application and record the steps taken.
6. Validate that the environment has returned to normal:
   - Review all business, user experience, application, and infrastructure metrics to verify that the system has returned to a known state. Capture dashboard screenshots if helpful.

Experiment timeline

Make sure that you capture the timeline of the end-to-end experiment, starting with load generation, injection of the fault, observation of impact, and recovery of the application, and ending when you stop the load generation. This is illustrated in the following example.



Experiment results

Experiment run ID	Experiment results
PET-ADOPT-EXP-23	(Link to experiment results.)

Identified defects

- The Kubernetes cluster didn't detect the CPU impairment of the PetSite pods, so it didn't schedule additional deployments.
- There was no increase in 4XX or 5XX error rates as a result of this experiment.
- We need to adjust the health check of the pod to account for impact to LCP when there are resource constraints.

Experiment result document

Configuration

Document the specific configurations for the experiment. For example:

- Load generation set to simulate 5K users issuing a total of 85 requests per second.

Prerequisites

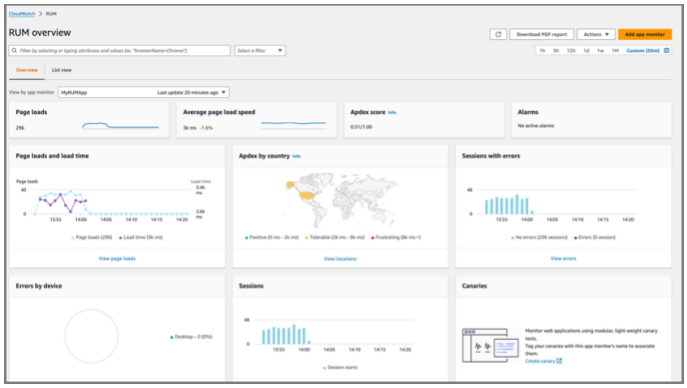
- Verified that the pet adoption site was running in the alpha test environment.
- Verified that the experiment template was configured to apply CPU stress to the PetSite application pods that are running in the EKS cluster. Application pods were identified by the Kubernetes label `app=petsite`.
- Load was confirmed to be running and generating 85 requests per second.

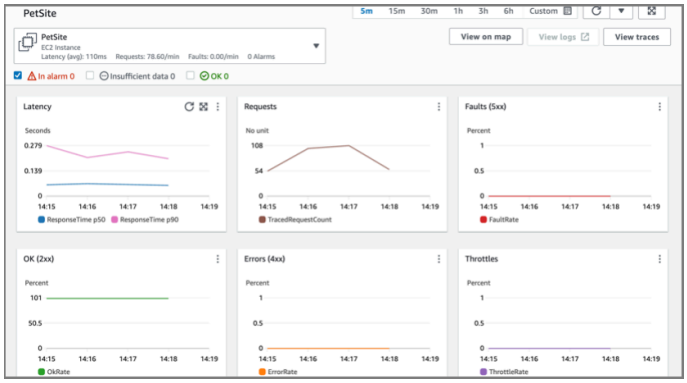
Steady state

Document the steps taken to achieve the steady state and how you verified it. For example:

For the test deployment of pet adoption site, a load of 85 RPS is being generated to simulate steady state. The CloudWatch RUM and CloudWatch dashboards were reviewed to verify that all business and application metrics were within normal ranges previous to the execution of the experiment.

Observability data:

Expected	Observed (screenshot)
• LCP is less than 4 seconds for P99 of requests. • Response latency is less than 500 ms. • There are no 4XX or 5XX errors.	



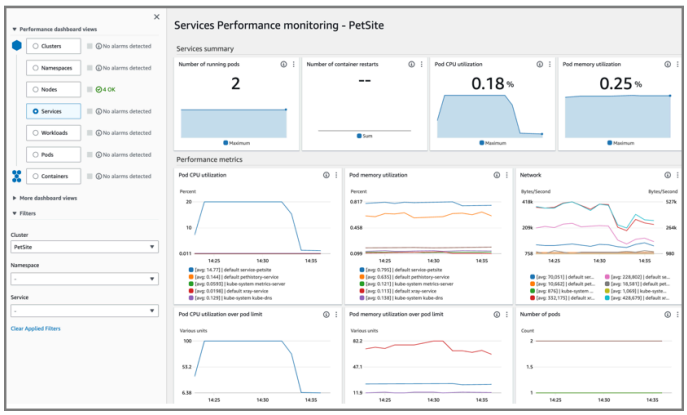
Fault injection

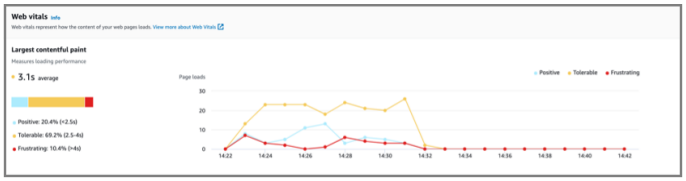
AWS FIS was used to inject faults by using the experiment template (provide link). The experiment was set to run for 10 minutes, and a rollback was configured if the worker nodes experienced CPU stress over 60 percent.

Fault observation

The CloudWatch RUM and CloudWatch dashboards were reviewed to track the steady state of the application (defined by using LCP metrics). Screenshots were captured in the following table.

Observability data:

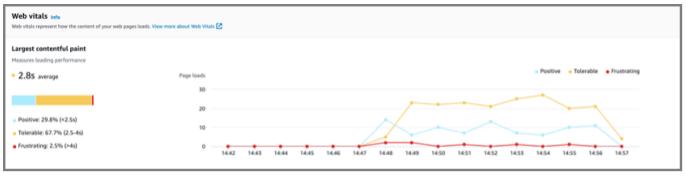
Expected	Observed (screenshot)
• LCP should remain under 4 seconds for P99. • Response time should remain under 500 ms. • No 4XX or 5XX errors should be encounter ed.	



Recovery

After the stress has been removed (the AWS FIS experiment has completed and removed the CPU stress from the pods), the application should resume its normal steady state. No manual intervention should be required.

Observability data:

Expected	Observed (screenshot)
LCP P99 should be under 4 seconds with the average under 2.5 seconds.	

Document history

The following table describes significant changes to this guide.

Change	Description	Date
Initial publication	—	April 4, 2025