



Chaos Engineering



“ *Chaos Engineering is a disciplined approach of identifying potential failures before they become outages.* ”

What is Chaos Engineering?

NOTE: The articles in this guide will use the term team to indicate a singular group that is responsible for an application that you are considering testing or actively testing using chaos experiments like those described in this guide.

Chaos Engineering is a disciplined approach of identifying potential failures before they become outages. Ultimately, the goal of Chaos Engineering is to enhance the stability and resiliency of our systems. There is some discussion in the community about proper terminology, but regardless of which side of the Chaos Engineering vs [Resilience Engineering](#) debate you come down on, most engineers agree that proper implementation is far more important than naming semantics.

Creating resilient software is a fundamental necessity for modern cloud applications and architectures. As systems are increasingly being distributed by design, the potential for unplanned failure and unexpected outages increases significantly. Thankfully, Chaos and Resilience Engineering techniques are quickly gaining traction within the community. [Many organizations](#) – both big and small – have embraced Chaos Engineering over the last few years.

What is This Guide?

Chaos Engineering is a new practice within the realm of DevOps and site reliability engineering. There are a variety of thoughts and opinions about what it is or what it should be, mostly from a high-level. Our goal is to help give some clarity about how to proceed beyond theories and concepts into practical steps.

This guide was created to give some specifics. Details. Concrete examples and direction for those who have bought into the idea and want to know what to actually do to get started.

Who is This Guide For?

This guide is for site reliability engineers (SREs), DevOps practitioners, Platform Engineers, and anyone else thinking about how to enhance the reliability of their computing systems, especially by enhancing those systems' abilities to stay up and running and providing a good experience for end users even when problems like component failures arise. It is specifically for people who want guidance without a lot of marketing and sales verbiage. At [Gremlin](#) we have created what we believe is a user-friendly and powerful means of implementing Chaos Engineering and we hope you will consider and ultimately use it. At the same time, we have intentionally written this content in a platform-agnostic way so that you can see the value of what Chaos Engineering offers.

Why Did We Create This Guide?

We want to build upon the introduction we have created in our Gremlin introductory content across our website and in our presentations. Those high-level views are intended to whet the appetite. Here we flesh out the idea with much greater detail, including implementation examples and a precise definition of what is needed before you start and while you implement in your current setting. Then, we follow that by including some extra ideas to spark your imagination for the future.

Contents

There are two main sections to our Chaos Engineering content. First, we start with a series outlining the stages of resiliency within an organization as you progress from first learning about Chaos Engineering to preparing to implement and on to increasing adoption. Second, we have a series of individual posts which each illustrate how one might choose to implement Chaos Engineering with specific technologies, architectures, and approaches that may be a part of your stack or something you are considering for the future.

Chaos Engineering Through Staged Resiliency

In his [ChaosConf 2018](#) talk titled [Practicing Chaos Engineering at Walmart](#), Walmart's Director of Engineering Vilas Veeraraghavan outlines how he and the hundreds of engineering teams at Walmart have implemented Resilience Engineering. By creating a robust series of "levels" or "stages" that each engineering team can work through, Walmart is able to progressively improve system resiliency while dramatically reducing support costs.

This series expands on this model by diving deep into the five Stages of Resiliency. Each post examines the necessary components of a stage, describes how those components are evaluated and assembled, and outlines the step-by-step process necessary to move from one stage to the next.

This series also digs into the specific implementation of each stage by progressing through the entire process with a real-world, fully-functional API application hosted on AWS. We'll go through everything from defining and executing disaster recovery playbook scenarios to improving system architecture and reducing RTO, RPO, and applicable support costs for this example app.

With a bit of adjustment for your own organizational needs, you and your team can implement similar practices to quickly add Chaos Engineering to your own systems with relative ease. After climbing through all five stages your system and its deployment will be almost entirely automated and will feature significant resiliency testing and robust disaster recovery failover.

Stage 0 - Preparing for Disaster

Stage 0 is all about implementing good site reliability engineering practices, laying the groundwork for Chaos Engineering. For the best results, these should be in place before you attempt anything in the later stages.

- Establish observability
- Define the critical dependencies
- Define the non-critical dependencies
- Create a disaster recovery failover playbook
- Create a critical dependency failover playbook
- Create a non-critical dependency failover playbook
- Publish the above and get team-wide agreement
- Manually execute a failover exercise
- Implementation example

Stage 1 - Injecting Chaos Internally

Stage 1 describes the early stages of implementing Chaos Engineering, where you begin to inject failure into non-production systems and establish good practices for documenting what you learn.

- Perform critical dependency failure tests in non-production
- Publish test results
- Implementation example

Stage 2 - Pushing the Envelope Outward

Stage 2 helps you take your first steps into automation and testing in production.

- Perform frequent, semi-automated tests
- Execute a resiliency experiment in production
- Publish test results
- Implementation example

Stage 3 - Automating Chaos Internally

Stage 3 is where you implement fully automated testing in your non-production systems and begin figuring out how to automate disaster recovery failover.

- Automate resiliency testing in non-production
- Semi-automate disaster recovery failover
- Implementation example

Stage 4 - Injecting Automated Chaos Everywhere

Stage 4 is a fully mature implementation of Chaos Engineering where you begin to have ideas of your own to add to and expand your testing plan.

- Integrate resiliency testing in CI/CD
- Automate resiliency and disaster recovery failover testing in production
- Implementation example

Chaos Engineering and Technology Options

This is a series covering interesting technologies, architectures, and approaches that companies are using today or considering for the future.

Chaos Engineering Tools Comparison

This article describes some of the common tools that the Chaos Engineering community considers when starting to implement the practice in an organization. The goal is to give a high level introduction to some frequently mentioned options and list some of the strengths of each using a brief table and then an annotated list.

Chaos Engineering: the history, principles, and practice

What is Chaos Engineering? This is a high-level overview of the concept, the history, and the practice of breaking things on purpose in the pursuit of knowledge that helps you improve the reliability of your systems.

What is Chaos Engineering? SREs and Leaders Define the Practice & Where It's Going

Because Chaos Engineering is a fairly new practice, there are a myriad of opinions about how to implement and when, about what the future holds. Here we interview industry leaders and practitioners to get thoughts about where the practice is heading.



Chaos Engineering Through Staged Resiliency - Stage 0

Prerequisites

Before you can begin moving through the resiliency stages there are a few prerequisite steps you'll need to complete. Most of these requirements are standard fare for a well-designed system, but ensuring each and every unique application team is fully prepared for the unknown is paramount to developing resilient systems.

1 Establish Observability

Microservice and clustered architectures favor the scalability and cost-efficiency of distributed computing, but also require a deep understanding of system behavior across a large pool of services and machines. Robust observability is a necessity for most modern software, which tends to be comprised of these complex distributed systems.

MONITORING: The act of collecting, processing, aggregating, and displaying quantitative data about a system. These data may be anything from query counts/

types and error counts/types to processing times and server lifetimes. Monitoring is a smaller subset of the overall measure of observability.

OBSERVABILITY: A measure of the ability to accurately infer what is happening internally within a system based solely on external information.

Continuous monitoring is critical to catch unexpected behavior that is difficult to reproduce, but at least historically, monitoring has largely focused on measuring “known unknowns.” By contrast, a highly-distributed system often requires tracking down, understanding, and preparing for a multitude of “unknown unknowns” – obscure issues that have never happened before, and may never happen again. A properly observable system is one that allows your team to answer new questions about the internals of the system without the need to deploy a new build. This kind of observability is often referred to as “black box monitoring,” as it allows your team to draw conclusions about unknowable events without using the internals of the system.

Most importantly, high observability is critically important when implementing Chaos Engineering techniques. As [Charity Majors](#), CEO of [Honeycomb](#), puts it, “Without observability, you don’t have ‘chaos engineering’. You just have chaos.”

2 Define the Critical Dependencies

Start by documenting every application *dependency* that is required for the application to function at all. This type of dependency is referred to as a **critical dependency**.

3 Define the Non-Critical Dependencies

Once all critical dependencies are identified then all remaining dependencies should be **non-critical dependencies**. If the core application can still function – even in a degraded state – when a dependency is missing, then that dependency is considered non-critical.

4 Create a Disaster Recovery Failover Playbook

Your team should create a disaster recovery plan specific to **failover**. A **disaster recovery failover playbook** should include the following information, at a minimum:

TIP: Not sure which failover scenarios to expect or plan for? Unable to determine if a dependency is critical vs non-critical? Consider running a GameDay to better prepare for and test specific scenarios in a controlled manner. Check out [How to Run a GameDay](#) for more info.

CONTACT INFORMATION: Explicitly document all relevant contact info for all team members. Identifying priority team members based on seniority, role, expertise, and the like will prove beneficial for later steps.

NOTIFICATION PROCEDURES: This should answer all the “Who/What/When/Why/How” questions for notifying relevant team members.

FAILOVER PROCEDURES: Deliberate, step-by-step instructions for handling each potential **failover scenario**.

5 Create a Critical Dependency Failover Playbook

A **critical dependency failover playbook** is a subset of the disaster recovery failover playbook and it should detail the step-by-step procedures for handling the potential failover scenarios for each critical dependency.

6 Create a Non-Critical Dependency Failover Playbook

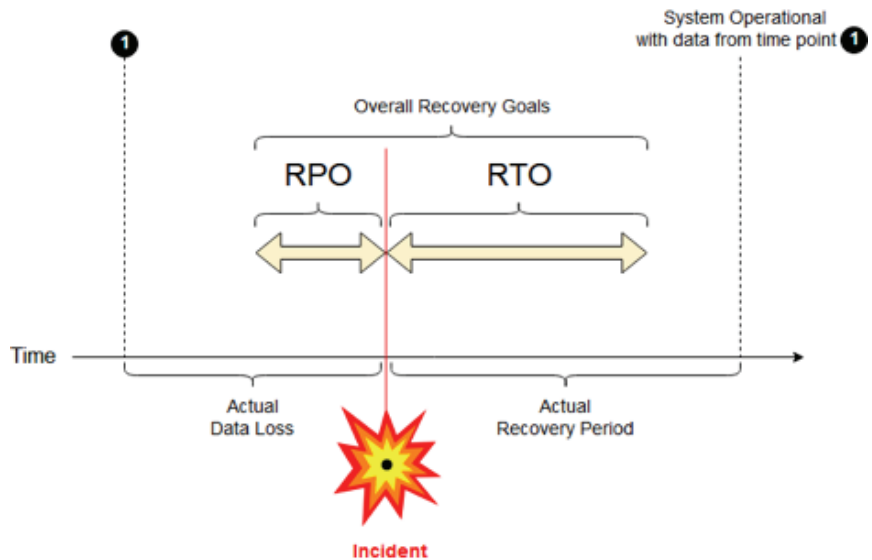
The final prerequisite is to determine how non-critical dependency failures will impact the system. Your team may not *necessarily* have failover procedures in place for non-critical dependencies, so this process can be as simple as testing and documenting what happens when each non-critical dependency is unavailable. Be sure to gauge the **severity** of the failure impact on the core application, which will provide the team with a better understanding of the system and its interactions (see Recovery Objectives).

Recovery Objectives

Most disaster recovery playbooks define the goals and allotted impact of a given failure using two common terms: **Recovery Time Objective** and **Recovery Point Objective**.

RECOVERY TIME OBJECTIVE (RTO): The maximum period of time in which the functionality of a failed service should be restored. For example, if a service with an RTO of twelve hours experiences an outage at 5:00 PM then functionality should be restored to the service by 5:00 AM the next morning.

RECOVERY POINT OBJECTIVE (RPO): The maximum period of time during which data can be lost during a service failure. For example, if a service with an RPO of two hours experiences an outage at 5:00 PM then only data generated between 3:00 PM and 5:00 PM should be lost – all existing data prior to 3:00 PM should still be intact.



RTO & RPO Diagrammed – Source: [Wikipedia](https://en.wikipedia.org/wiki/Recovery_time_objective)

Complete and Publish Prerequisites

Ensure that all [Prerequisites](#) have been met. All playbooks, dependency definitions, and other relevant documentation should be placed in a singular, globally accessible location so every single team member has immediate access to that information. Maintaining a single repository for the information also maintains consistency across the team, so there's never any confusion about the steps in a particular scenario or what is defined as a critical dependency.

Team-Wide Agreement on Playbooks

With unfettered access to all documentation, the next step is to ensure the entire team agrees with all documented information as its laid out. If there is disagreement about the best way to approach a given failover scenario, or about the risk and potential impact of a non-critical dependency failure, this is the best time to suss out those differences of opinion and come to a unanimous “best” solution. A healthy, active debate provides the team with a deeper understanding of the system and encourages the best ideas and techniques to bubble up to the surface.

While the goal is agreement on the playbooks currently laid out, documentation can (and should) be updated in the future as experiments shed new light on the system. The team should be encouraged and empowered to challenge the norms in order to create a system that is always adapting and evolving to be as resilient as possible.

This is where your team probably wants to consider Service-Level Objectives (SLOs) and Service-Level Agreements (SLAs). What promises have you made to your customers in contracts? What promises do you want to make internally? How will you keep those promises? Write your plans down and tailor your playbooks to ensure you know how you will meet your availability requirements and goals when unexpected failures happen.

Manually Execute a Failover Exercise

The last step is to manually perform a failover exercise. The goal of this exercise is to verify that the disaster recovery failover playbook works as expected. Therefore, the step-by-step process defined in the playbook should be followed exactly as documented.

WARNING: If an action or step is not *explicitly* documented within a playbook then it should be ignored. If the exercise fails or cannot be completed this likely indicates that the playbook needs to be updated.

Resiliency Stage 0: Implementation Example

Throughout this series, we'll take a simple yet real-world application through the entire staging process to illustrate how a team might progress their application through all five resiliency stages. While every application and system architecture is unique, this example illustrates the basics of implementing every step within a stage, to provide you with a jumping off point for staged resiliency within your own system.

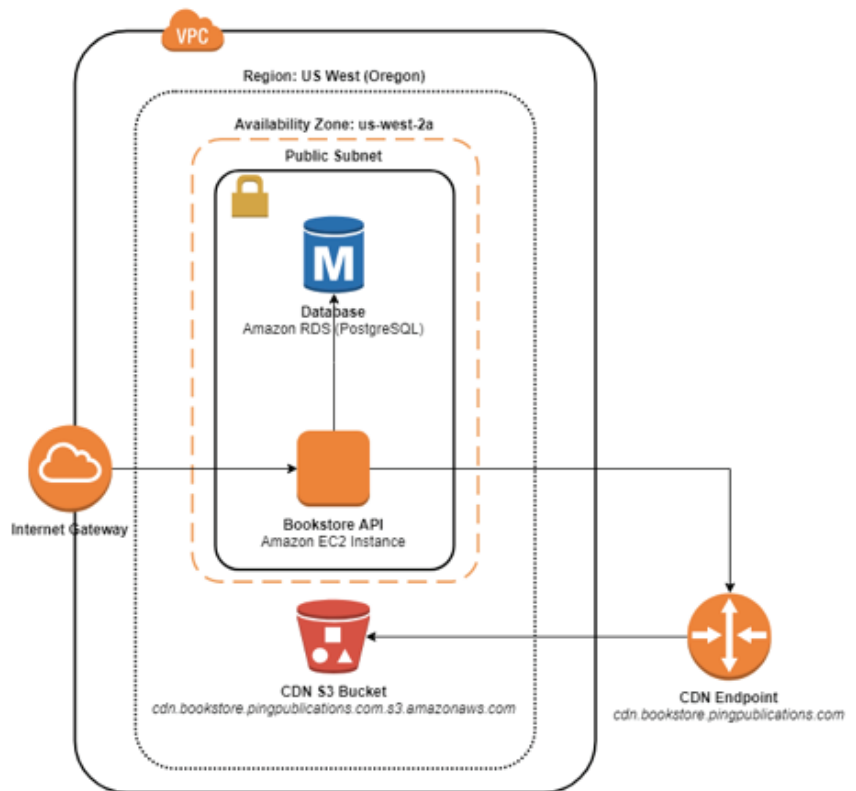
The **Bookstore example application** is a publicly accessible API for a virtual bookstore. The API includes two primary endpoints: `/authors/` and `/books/`, which can be used to add, update, or remove **Authors** and **Books**, respectively.

Bookstore's architecture consists of three core components, all of which are housed within Amazon Web Services.

API: The API is created with Django and the [Django REST Framework](#) and is hosted on an Amazon EC2 instance running NGINX.

DATABASE: A PostgreSQL database handles all data and uses Amazon RDS.

CDN: All static content is collected in and served from an Amazon S3 bucket.



Initial System Architecture

The web API is at the publicly accessible <http://bookstore.pingpublications.com> endpoint. The web API, database, and CDN endpoints are DNS-routed via Amazon Route53 to the underlying Amazon EC2, RDS, and Amazon S3 buckets, respectively.

Here's a simple request to the `/books/` API endpoint.

```
$ curl http://bookstore.pingpublications.com/books/ | jq
[
  {
    "url": "http://bookstore.pingpublications.com/books/1/",
    "authors": [
      {
        "url": "http://bookstore.pingpublications.com/authors/1/",
        "birth_date": "1947-09-21",
        "first_name": "Stephen",
        "last_name": "King"
      }
    ],
    "publication_date": "1978-09-01",
    "title": "The Stand"
  }
]
```

WARNING: The initial design and architecture for the **Bookstore** sample application is *intentionally* less resilient than a full production-ready system. This leaves room for improvement as progress is made through the resiliency stages throughout this series.

Prerequisites

We begin the example implementation by defining all prerequisites for the **Bookstore** app.

0. Define System Architecture

It may be useful to take a moment to define the basic components of the system, which can then be referenced throughout your playbooks. Below are the initial services for the **Bookstore** app.

Service	Platform	Technologies	AZ	VPC	Subnet	Endpoint
API	Amazon EC2	Django, Nginx	us-west-2a	bookstore-vpc	bookstore-subnet-2a	bookstore.pingpublications.com
Database	Amazon RDS	PostgreSQL 10.4	us-west-2a	bookstore-vpc	bookstore-subnet-2a	db.bookstore.pingpublications.com
CDN	Amazon S3	Amazon S3	N/A	N/A	N/A	cdn.bookstore.pingpublications.com

1 Define the Critical Dependencies

At this early stage of the application, all dependencies are critical.

Dependency	Criticality Period	Manual Workaround	RTO	RPO	Child Dependencies
API	Always	Manual Amazon EC2 Instance Restart	12	24	Database, CDN
Database	Always	Manual Amazon RDS Instance Restart	12	24	N/A
CDN	Always	Manual Amazon S3 Bucket Verification	24	24	N/A

A **criticality period** is useful for dependencies that are only considered “critical” during a specific period of time. For example, a database backup service that runs at 2:00 AM PST every night may have a criticality period of 2:00 AM - 3:00 AM PST. This is also a good time to evaluate initial acceptable RTO and RPO values. These values will decrease over time as resiliency improves, but setting a baseline goal provides a target to work toward.

2 Define the Non-Critical Dependencies

The **Bookstore** app is so simple that it doesn't have any non-critical dependencies – if a service fails, the entire application fails with it.

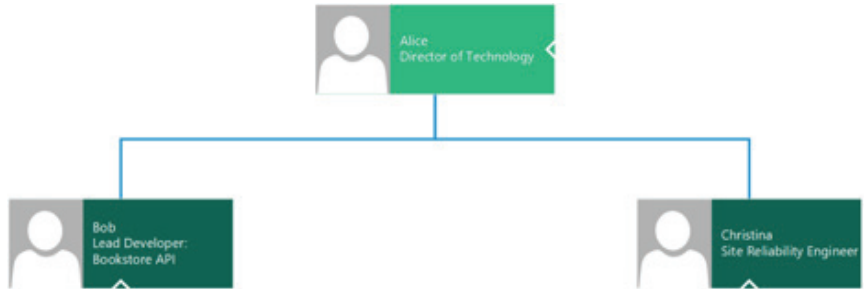
3 Create a Disaster Recovery Failover Playbook

The first part of a disaster recovery failover playbook should contain contact information for all relevant team members, including the services those members are related to and their availability.

Team Member	Position	Relevant Services	Email	Phone	Availability
Alice	Director of Technology	ALL	alice@example.com	555-555-5550	9 - 5, M - F
Bob	Lead Developer, Bookstore API	Bookstore API	bob@example.com	555-555-5551	9 - 5, M - F; 10 - 2, S & S
Christina	Site Reliability Engineer	ALL	christina@example.com	555-555-5552	On-call

To define the proper notification procedures it may help to add an organizational chart to the playbook.

Chaos Engineering



Organizational Chart

This can be used in conjunction with the contact information table to determine which team members should be contacted – and in what priority – when a given service fails.

The final part of the disaster recovery failover playbook is to explicitly document the step-by-step procedures for every failover scenario. For the **Bookstore** application, we'll just provide a single failover scenario plan for when the database fails, but this can be expanded as necessary for all other failover scenarios.

Scenario: Bookstore API Failure

The current architecture of the **Bookstore** app is limited to a manual *Backup & Restore* disaster recovery strategy.

Disaster Recovery Leads

- Primary: Bob, Lead Developer, Bookstore API
- Secondary: Alice, Director of Technology

Severity

- Critical

Recovery Procedure Overview

1. Manually verify if the API server has failed, or if the server is available but the Django API app failed.
 - If server failure: Manually restart API server.

PURPOSE: The severity level of this particular failover. Severity should be a general indicator of acceptable RTO/RPO metrics, as well as how critically dependent the service is.

Chaos Engineering

- If Django API app failure: Manually restart Django API app.
2. If neither restart solution works, propagate replacement server using prepared backup Amazon Machine Image (AMI).
 3. Verify backup instance is functional.
 4. Update DNS routing.

PURPOSE: Indicates the basic assumptions that can be made during the recovery process. Assumptions are typically factors outside of your control, such as third-party vendor availability.

Basic Assumptions

- Amazon S3, Amazon RDS, Amazon Route 53, and Amazon EC2 are all online and functional.
- Frequent AMI backups are generated for the application instance.
- Application code can be restored from code repository with minimal manual effort.

Recovery Time Objective

- 12 Hours

Recovery Point Objective

- 24 Hours

Recovery Platform

- Amazon EC2 `t2.micro` instance on `us-west-2a` Availability Zone with NGINX, Python, and Django **Bookstore** application configured and installed from the latest release.

PURPOSE: Indicate the specific technologies, platforms, and services that are necessary to complete the recovery procedure.

Recovery Procedure

1. Manually verify the `bookstore-api` instance availability on Amazon EC2.

```
$ curl http://bookstore.pingpublications.com
curl: (7) Failed to connect to bookstore.pingpublications.com
port 80: Connection refused
```

- If the `bookstore-api` instance is active but **Bookstore** Django application is failing then manually restart app from the terminal.

```
sudo systemctl restart gunicorn
```

- If **Bookstore** Django application remains offline then manually restart the instance and recheck application availability.
2. If the `bookstore-api` EC2 instance has completely failed and must be replaced then propagate a new Amazon EC2 instance from the `bookstore-api-ec2-image` AMI backup.
 3. Use the pre-defined `bookstore-api-ec2` launch template.

```
$ aws ec2 run-instances --launch-template
LaunchTemplateName=bookstore-api-ec2
532151327118 r-0e57eca4a2e78d479
...
```

4. Default values can be overridden as shown here.

```
aws ec2 run-instances \
  --image-id ami-087ff330c90e99ac5 \
  --count 1 \
  --instance-type t2.micro \
  --key-name gabe-ping-pub \
  --security-group-ids sg-25268a50 sg-0f818c22884a88694 \
  --subnet-id subnet-47ebaf0c
```

5. Confirm the instance has been launched and retrieve the public DNS and IPv4 address.

```
$ aws ec2 describe-instances --filters "Name=image-id,Values=ami-087ff330c90e99ac5,Name=instance-state-code,Values=16" --query "Reservations[*].Instances[*].[LaunchTime,PublicDnsName,PublicIpAddress]"
2018-11-09T04:37:32.000Z      ec2-54-188-3-235.us-west-2.
compute.amazonaws.com 54.188.3.235
```

NOTE: The `filters` used in the command above searched for **Running** instances based on the AMI `image-id`. If multiple instances match these filters then the `LaunchTime` value retrieved from the query will help determine which instance is the latest launched.

6. SSH into the new `bookstore-api` instance.

```
ssh ec2-54-188-3-235.us-west-2.compute.amazonaws.com
```

7. Pull latest **Bookstore** application code from the repository.

```
$ cd ~/apps/bookstore_api && git pull
Already up to date.
```

8. Restart application via `gunicorn`.

```
sudo systemctl restart gunicorn
```

9. On a local machine verify backup instance is functional, the public IPv4 address is available, and the **Bookstore** app is online.

```
$ curl ec2-54-188-3-235.us-west-2.compute.amazonaws.com | jq
{
  "authors": "http://ec2-54-188-3-235.us-west-2.compute.
amazonaws.com/authors/",
  "books": "http://ec2-54-188-3-235.us-west-2.compute.
amazonaws.com/books/"
}
```

10. Update the Amazon Route 53 DNS `A` record to point to the new `bookstore-api` EC2 instance IPv4 address.
11. Once DNS propagation completes then verify that the API endpoint is functional.

```
$ curl bookstore.pingpublications.com | jq
{
  "authors": "http://bookstore.pingpublications.com/
authors/",
  "books": "http://bookstore.pingpublications.com/books/"
}
```

PURPOSE: Indicates the test procedures necessary to ensure the system is functioning normally.

Test Procedure

1. Manually verify that `bookstore.pingpublications.com` is accessible and functional.
2. Confirm that critical dependencies are functional and also connected (database and CDN).

Resume Procedure

- Service is now fully restored.

PURPOSE: For more complex systems this final procedure should provide steps for resuming normal service.

4 Create a Critical Dependency Failover Playbook

Scenario: Database Failure

Disaster Recovery Leads

- Primary: Alice, Director of Technology
- Secondary: Bob, Lead Developer, Bookstore API

Severity

- Critical

Recovery Procedure Overview

1. Manually verify database availability through Amazon RDS monitoring.
2. If unavailable, restart.
3. If still unavailable, manually propagate replica.
4. If necessary, restore from the most recent snapshot.

Basic Assumptions

- Amazon RDS is online and functional.
- Database backups are available.
- AWS Support contact is available for additional assistance.

Recovery Time Objective

- 12 Hours

Recovery Point Objective

- 24 Hours

Recovery Platform

- PostgreSQL 10.4 database with identical configuration running on Amazon RDS with minimal `us-west-2a` Availability Zone.

Recovery Procedure

1. Disaster recovery team member should manually verify database availability through Amazon RDS monitoring.
2. Manually restart the `bookstore-db` instance.
If `bookstore-db` is back online, proceed to Resume Procedure.
3. If `bookstore-db` remains unavailable manual propagate a replica Amazon RDS PostgreSQL 10.4 instance.
4. If replacement created, update DNS routing on Amazon Route 53 for `db.bookstore.pingpublications.com` endpoint.
5. (Optional): Restore data from the most recent snapshot within acceptable RPO.

Test Procedure

1. Manually confirm a connection to public `bookstore-db` endpoint (`db.bookstore.pingpublications.com`).
2. Confirm that `bookstore-api` can access the `bookstore-db` instance.

Resume Procedure

- Service is now fully restored.
- If necessary, perform manual data recovery.

Scenario: CDN Failure

Disaster Recovery Leads

- Primary: Alice, Director of Technology
- Secondary: Christina, Site Reliability Engineer

Severity

- Critical

Recovery Procedure Overview

1. Manual verification of applicable Amazon S3 bucket.
2. If unavailable, manually recreate bucket and upload a backup snapshot of static data.

Basic Assumptions

- Amazon S3 is online and functional.
- Static asset backups are available.
- Static asset collection can be performed remotely from the EC2 `bookstore-api` server or locally via Django `manage.py collectstatic` command.
- AWS Support contact is available for additional assistance.

Recovery Time Objective

- 24 Hours

Recovery Point Objective

- 24 Hours

Recovery Platform

- Amazon S3 `private` bucket accessible by administrator AWS account.

Recovery Procedure

1. Team member manually verifies Amazon S3 `cdn.bookstore.pingpublications.com` bucket exists, is accessible, and contains all static content.
2. Manually recreate `cdn.bookstore.pingpublications.com` bucket.
3. Manually upload all static content to `cdn.bookstore.pingpublications.com` bucket.
4. If `cdn.bookstore.pingpublications.com` bucket exists but is non-functional, manually create the bucket, upload static content, and route the system to backup.

Test Procedure

1. Confirm all static content exists in `cdn.bookstore.pingpublications.com` Amazon S3 bucket.
2. Confirm public endpoint (`cdn.bookstore.pingpublications.com/static`) is accessible for static content.
3. Confirm that `bookstore-api` can access `cdn.bookstore.pingpublications.com` bucket and content.

Resume Procedure

- Service is now fully restored.

5 Create a Non-Critical Dependency Failover Playbook

The **Bookstore** example app doesn't have any non-critical dependencies at the moment given its simple architecture (CDN > API Server < Database). However, progressing through each resiliency stage will require additional systems and services to maintain failure resilience, which will inherently add non-critical dependencies.

Complete and Available Prerequisites

- Status: **Complete**

All prerequisites for the **Bookstore** app have been met and all documentation has been dispersed among every member of the team.

Team-Wide Agreement on Playbooks

- Status: **Complete**

Every team member has agreed on the playbook/scenarios defined above.

Manually Execute a Failover Exercise

- Status: **Complete**

For this stage of the **Bookstore** app, we've manually performed the [Scenario: Bookstore API Failure](#) exercise.

Full manual restoration of the `bookstore-api` EC2 instance and the **Bookstore** app resulted in approximately 30 minutes of downtime. This is well under the initial RTO/RPO goals so we can reasonably update the playbooks. However, this manual process is still clunky and prone to errors, so there's plenty of room for improvement.

Resiliency Stage 0: Completion

This post laid the groundwork for how to implement resilience engineering practices through thoughtfully-designed dependency identification and disaster recovery playbooks. We also broke down the requirements and steps of **Resiliency Stage 0**, which empowers your team to begin the journey toward a highly-resilient system. Stay tuned to the [Gremlin Blog](#) for additional posts that will break down the four remaining **Stages of Resiliency**!



Chaos Engineering Through Staged Resiliency - Stage 1

In Chaos Engineering Through Staged Resiliency - Stage 0 we explored how engineering teams at Walmart successfully identify and combat unexpected system failure using a series of “resiliency stages.” We also demonstrated the process of going through **Stage 0** for the **Bookstore** API sample application. We showed that completing **Resiliency Stage 0** requires the definition of a disaster recovery failover playbook, dependency failover playbooks, team-wide agreement on said playbooks, and the execution of a manual failover exercise.

In this second part, we’ll dive into the components of **Resiliency Stage 1**, which focuses on critical dependency failure testing in non-production environments.

Prerequisites

- Creation and agreement on [Disaster Recovery and Dependency Failover Playbooks](#).
- Completion of Resiliency Stage 0.

Perform Critical Dependency Failure Tests in Non- Production

The most important aspect of **Resiliency Stage 1** is testing the resilience of your systems when critical dependencies fail. However, this early in the process the system can't be expected to handle such failures in a production environment, so these tests should be performed in a non-production setting.

Manual Testing Is Okay

Eventually, all experiments should be executed automatically, with little to no human intervention required. However, during **Resiliency Stage 1** the team should feel comfortable performing manual tests. The goal here isn't to test the automation of the system, but rather to determine the outcome and system-wide impact whenever a critical dependency fails.

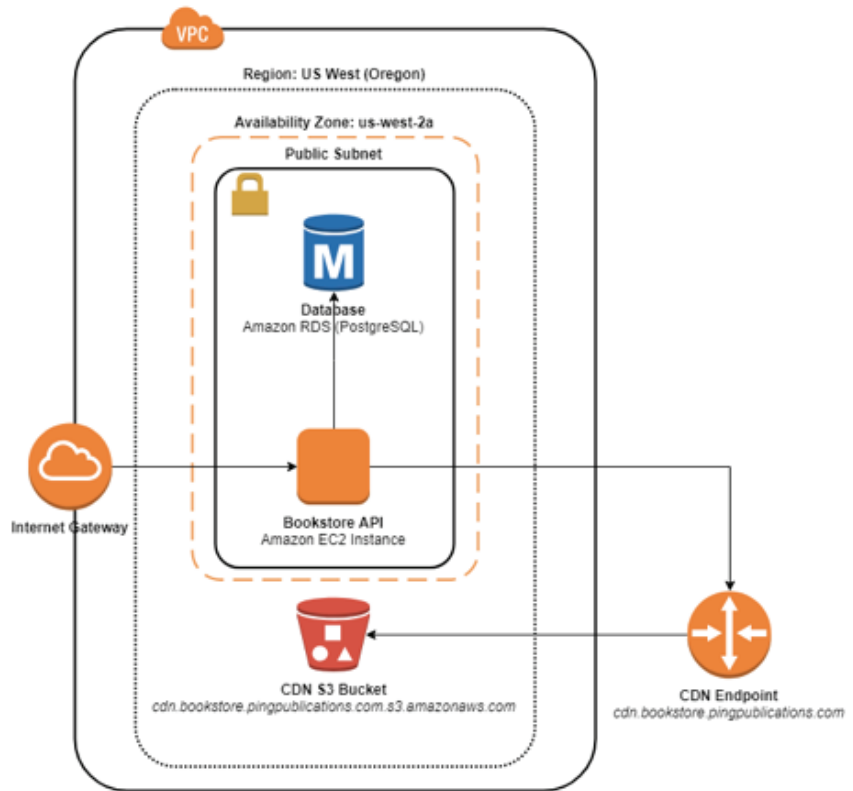
Publish Test Results

After every critical dependency failure test is performed the results should be globally published to the entire team. This allows every team member to closely scrutinize the outcome of a given test. This encourages feedback and communication, which naturally provides an insightful evaluation from the members that are best equipped to analyze the test results.

Resiliency

Stage 1: Implementation Example

At present the **Bookstore** sample app has no real concept of failover or resiliency – if the application or a critical dependency fails, the entire system fails along with it.



Bookstore App Architecture

Completing **Resiliency Stage 1** requires testing in a non-production environment, so this forces us to improve deployment and system resiliency by moving beyond a single point of failure. Since we're taking advantage of AWS we *could* jump straight into implementing all the safety net features the AWS platform brings with it (e.g. auto-scaling, elastic load balancing, traffic manipulation, etc). However, going through each resiliency stage by meeting the minimal requirements provides more robust examples and lets us learn the intricacies of all components within the system.

Therefore, before we go through this stage we'll be modifying the **Bookstore** architecture by adding a secondary staging

environment. In fact, this is a good opportunity to use a blue/green deployment strategy. A blue/green configuration requires creating two parallel production environments.

As an example scenario, the `blue` environment is “active” and is handling production traffic. Meanwhile, the other environment (`green`) is used for staging application changes. Once the `green` environment has been fully tested then traffic is routed to the `green` environment. `Green` now becomes the live production environment while `blue` is unused and ready for the next staging phase. If something goes wrong during deployment to `green`, traffic can be quickly routed back to the still-active `blue` environment.

Adding a Staging App Environment

1. Create a new EC2 instance from the most recent launch template version. Optionally, we can specify tags such as `Name` to help differentiate between the two environments.

```
aws ec2 run-instances --launch-template
LaunchTemplateName=bookstore-api-ec2 \
--tag-specifications "ResourceType=instance,Tags=[{Key=N
ame,Value=bookstore-api-green}]"
```

2. Verify the instance has been created by finding active instances generated from the same AMI.

```
$ aws ec2 describe-instances --filters "Name=image-
id,Values=ami-087ff330c90e99ac5,Name=instance-state-
code,Values=16" --query "Reservations[*].Instances[*].
[LaunchTime,PublicDnsName,PublicIpAddress]"
2018-11-09T04:37:32.000Z      ec2-54-188-3-235.us-west-2.
compute.amazonaws.com 54.188.3.235
2018-11-09T07:44:38.000Z      ec2-54-191-29-110.us-west-2.
compute.amazonaws.com 54.191.29.110
```


Chaos Engineering

3. SSH into new environment.

```
ssh ec2-54-191-29-110.us-west-2.compute.amazonaws.com
```

4. Pull any new application changes and restart the application.

```
$ cd ~/apps/bookstore_api && git pull && sudo systemctl  
restart gunicorn  
From github.com:GabeStah/bookstore_api  
0f1b811..ee6f15c master -> origin/master
```

5. Verify that each instance environment is serving a different version of the application.
 - Here we check the app version for the current production endpoint.

```
$ curl bookstore.pingpublications.com/version | jq  
{  
  "version": 3  
}
```

And for the `bookstore-api-green` environment.

```
$ curl ec2-54-191-29-110.us-west-2.compute.amazonaws.com/  
version | jq  
{  
  "version": 4  
}
```

Configuring Multi-AZ RDS

The PostgreSQL database on Amazon RDS is not resilient – if the single instance fails, the entire application goes down. RDS provides a [Multi-AZ Deployment](#) solution, which automatically creates a synchronous database replica on another Availability Zone. In the event of a primary failure, the secondary performs an automatic failover so there is little to no downtime.

Modifying the existing `bookstore-db` RDS instance so it uses a Multi-AZ Deployment is fairly simple.

1. Confirm the instance is not using Multi-AZ.

```
$ aws rds describe-db-instances --db-instance-identifier bookstore-db --query "DBInstances[*].[AvailabilityZone,MultiAZ]"
us-west-2a    False
```

2. Modify the instance by passing the `multi-az` flag. In this case, we always want to apply changes immediately (rather than wait for the next scheduled maintenance window).

```
aws rds modify-db-instance --db-instance-identifier bookstore-db --multi-az --apply-immediately
```

3. It may take a few minutes for the modification to take effect. We can check the status within the `PendingModifiedValues` key structure to confirm `MultiAZ` is waiting to be modified.

```
$ aws rds describe-db-instances --db-instance-identifier bookstore-db --query "DBInstances[*].PendingModifiedValues.[MultiAZ]"
True
```

After about 10 minutes we can check the status once again to confirm that a Multi-AZ Deployment is now active.

```
$ aws rds describe-db-instances --db-instance-identifier bookstore-db --query "DBInstances[*].[AvailabilityZone,MultiAZ]"
us-west-2a    True
```

Amazon S3 Cross-Region Replication

The last critical dependency for the **Bookstore** app is the CDN, which relies on Amazon S3. While Amazon S3 is an extremely resilient system and not prone to downtime, there is still some precedent for Amazon S3 failure. Luckily, the service has the ability to automatically replicate objects between buckets within different AWS regions.

To accomplish this we're working around a quirk with Amazon S3 bucket naming and DNS routing. Specifically, a CNAME DNS name must exactly match a referenced Amazon S3 bucket name. Therefore, we'll continue using the `cdn.bookstore.pingpublications.com.s3.amazonaws.com` Amazon S3 bucket (and Route53 CNAME record), but this bucket is now the secondary CDN. A new Amazon S3 bucket named `cdn-primary.bookstore.pingpublications.com` is now the primary CDN bucket.

We then create a CloudFront endpoint that points to the `cdn-primary` bucket but uses the DNS CNAME `cdn.bookstore.pingpublications.com`. From there, Amazon Route53 failover properly handles requests based on whichever Amazon S3 bucket is available. This automates the CDN failover process.

1. Start by creating an Amazon CloudFront Distribution. The fields we need to set are as follows.

ORIGIN DOMAIN NAME: This must be set the primary Amazon S3 bucket (`cdn-primary.bookstore.pingpublications.com.s3-us-west-2.amazonaws.com`).

ALTERNATE DOMAIN NAMES (CNAMES): This should point to the root CDN endpoint (`cdn.bookstore.pingpublications.com`).

2. Create an Amazon Route53 Health Check to check the health of the primary Amazon S3 bucket, which is behind the CloudFront endpoint created above.

```
$ aws route53 create-health-check --caller-reference BookstoreCloudfront --output json \
  --health-check-config '{
    "FullyQualifiedDomainName": "dqno94z16p2mg.cloudfront.net",
    "ResourcePath": "/static/status.json",
    "Type": "HTTP"
  }' \
  --query "{Id: HealthCheck.Id}"
{
  "Id": "352ef27f-213e-4dae-983e-62fd048ea7be"
}
```

TIP: We've added a `status.json` file to our static asset collection, which we'll use to confirm the online status of both CDN buckets, as well as determine which bucket is currently serving our content.

3. Create another Amazon Route53 Health Check to check the status of the secondary Amazon S3 bucket (`cdn.bookstore.pingpublications.com.s3.amazonaws.com`).

```
$ aws route53 create-health-check --caller-reference BookstoreS3 --output json \
  --health-check-config '{
    "FullyQualifiedDomainName": "cdn.bookstore.pingpublications.com.s3.amazonaws.com",
    "ResourcePath": "/static/status.json",
    "Type": "HTTP"
  }' \
  --query "{Id: HealthCheck.Id}"
{
  "Id": "e7fce91f-c82a-460f-b247-58c35790e39d"
}
```

4. Create a third Amazon Route53 Health Check that combines the two previous health checks. This combined health check is monitored by Amazon Route53 and automatically triggers failover from the primary to secondary CDN when the check fails. We need to pass the two health check `Ids` created above into this combined health check configuration.

```
$ aws route53 create-health-check --caller-reference
BookstoreCDN --output json --health-check-config '{
    "HealthThreshold": 2,
    "Type": "CALCULATED",
    "ChildHealthChecks": [
        "e7fce91f-c82a-460f-b247-58c35790e39d",
        "352ef27f-213e-4dae-983e-62fd048ea7be"
    ]
}'
{
    "HealthCheck": {
        "HealthCheckConfig": {
            "HealthThreshold": 2,
            "Type": "CALCULATED",
            "ChildHealthChecks": [
                "352ef27f-213e-4dae-983e-62fd048ea7be",
                "e7fce91f-c82a-460f-b247-58c35790e39d"
            ],
            "Inverted": false
        },
        "CallerReference": "BookstoreCDN",
        "HealthCheckVersion": 1,
        "Id": "99d5b3a0-7709-4553-934f-d0ac7bcd2eb3"
    },
}
```

Finally, verify all CDN endpoints behave correctly and point to the proper Amazon S3 bucket.

- Start by checking the primary Amazon S3 bucket.

```
$ curl http://cdn-primary.bookstore.pingpublications.com.s3-
us-west-2.amazonaws.com/static/status.json
{
    "online": true,
    "cdn": "primary"
}
```

- The CloudFront endpoint routes to the primary bucket above.

```
$ curl http://dqno94z16p2mg.cloudfront.net/static/status.json
{
    "online": true,
    "cdn": "primary"
}
```

Chaos Engineering

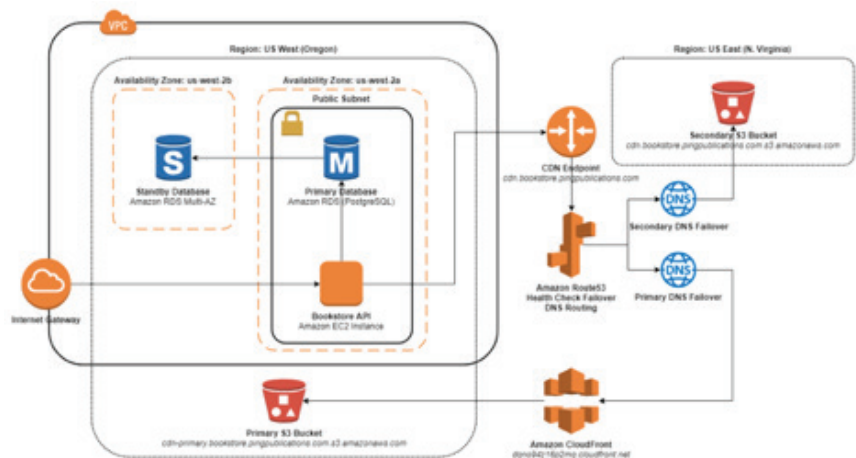
- The last part of the chain is the primary CDN endpoint, which routes to the CloudFront endpoint, and then onto the primary Amazon S3 bucket itself.

```
$ curl http://cdn.bookstore.pingpublications.com/static/status.json
{
  "online": true,
  "cdn": "primary"
}
```

- Finally, the secondary Amazon S3 bucket is properly retrieving the secondary static content.

```
$ curl http://cdn.bookstore.pingpublications.com.s3.amazonaws.com/static/status.json
{
  "online": true,
  "cdn": "secondary"
}
```

The app is now ready for failure testing. Here is the updated diagram showing the **Bookstore** app architecture for **Resiliency Stage 1**.



Bookstore App Architecture

Database Failure Test

With Multi-AZ configured on the Amazon RDS instance we are now ready to perform a failover test for this critical dependency.

1. Confirm the status and current Availability Zone of the `bookstore-db` instance with the `describe-db-instances` CLI command.

```
$ aws rds describe-db-instances --db-instance-identifier bookstore-db --output json --query "DBInstances[*].{Id:DBInstanceIdentifier, AZ:AvailabilityZone, SecondaryAZ:SecondaryAvailabilityZone, Endpoint:Endpoint.Address, Status:DBInstanceStatus}"
[
  {
    "SecondaryAZ": "us-west-2b",
    "Status": "available",
    "AZ": "us-west-2a",
    "Id": "bookstore-db",
    "Endpoint": "bookstore-db.cvajeopcvda.us-west-2.rds.amazonaws.com"
  }
]
```

2. Perform a manual reboot via `reboot-db-instance`. We're also forcing a failover with the `--force-failover` flag.

```
$ aws rds reboot-db-instance --db-instance-identifier bookstore-db --force-failover --output json --query "DBInstance.{Id:DBInstanceIdentifier, AZ:AvailabilityZone, SecondaryAZ:SecondaryAvailabilityZone, Endpoint:Endpoint.Address, Status:DBInstanceStatus}"
{
  "SecondaryAZ": "us-west-2b",
  "Status": "rebooting",
  "AZ": "us-west-2a",
  "Id": "bookstore-db",
  "Endpoint": "bookstore-db.cvajeopcvda.us-west-2.rds.amazonaws.com"
}
```

3. Even though the primary instance is currently `Status: rebooting`, both the production and secondary **Bookstore** environments are able to connect to `db.bookstore.pingpublications.com`, which is pointed to the `bookstore-db.cvajeopcvda.us-west-2.rds.amazonaws.com` endpoint.

```
$ curl bookstore.pingpublications.com/books/ | jq
[
  {
    "url": "http://bookstore.pingpublications.com/
books/1/",
    "authors": [
      {
        "url": "http://bookstore.pingpublications.
com/authors/1/",
        "birth_date": "1947-09-21",
        "first_name": "Stephen",
        "last_name": "King"
      }
    ],
    "publication_date": "1978-09-01",
    "title": "The Stand"
  }
]

$ curl ec2-54-191-29-110.us-west-2.compute.amazonaws.com/
books/ | jq
[
  {
    "url": "http://ec2-54-191-29-110.us-west-2.compute.
amazonaws.com/books/1/",
    "authors": [
      {
        "url": "http://ec2-54-191-29-110.us-west-2.
compute.amazonaws.com/authors/1/",
        "birth_date": "1947-09-21",
        "first_name": "Stephen",
        "last_name": "King"
      }
    ],
    "publication_date": "1978-09-01",
    "title": "The Stand"
  }
]
```

4. Amazon RDS Multi-AZ [automatically](#) switches to the secondary replica on the other Availability Zone and reroutes the endpoint for us.
5. Once the reboot completes, query the `bookstore-db` instance to confirm that it has swapped the active instance to the other Availability Zone.


```
$ aws rds describe-db-instances --db-instance-identifier bookstore-db --output json --query "DBInstances[*]. {Id:DBInstanceIdentifier, AZ:AvailabilityZone, SecondaryAZ:SecondaryAvailabilityZone, Endpoint:Endpoint.Address, Status:DBInstanceStatus}"
[
  {
    "SecondaryAZ": "us-west-2a",
    "Status": "available",
    "AZ": "us-west-2b",
    "Id": "bookstore-db",
    "Endpoint": "bookstore-db.cvajeopcjvda.us-west-2.rds.amazonaws.com"
  }
]
```

6. Manually confirm the **Bookstore** API remains connected to the bookstore-db instance replica.

```
$ curl bookstore.pingpublications.com/books/ | jq
[
  {
    "url": "http://bookstore.pingpublications.com/books/1/",
    "authors": [
      {
        "url": "http://bookstore.pingpublications.com/authors/1/",
        "birth_date": "1947-09-21",
        "first_name": "Stephen",
        "last_name": "King"
      }
    ],
    "publication_date": "1978-09-01",
    "title": "The Stand"
  }
]
```

CDN Failure Test

This test ensures that the primary CDN endpoint (cdn.bookstore.pingpublications.com) performs an automatic failover to the secondary Amazon S3 bucket when the Route53 Health Check fails.

1. Retrieve appropriate Health Check `Ids`.

```
$ aws route53 list-health-checks --output json
{
  "HealthChecks": [
    {
      "HealthCheckConfig": {
        "HealthThreshold": 2,
        "Type": "CALCULATED",
        "ChildHealthChecks": [
          "57a1606e-df1e-43d0-b6d5-e04bf3d789ff",
          "e4a34329-26c3-4dbe-bec7-4afba94bd487"
        ],
        "Inverted": false
      },
      "HealthCheckVersion": 2,
      "Id": "1b01881d-a658-4d8e-86c9-8d5329d0bfcc"
    }
  ]
}
```

2. Invert the Health Check, so that a normally healthy value is now considered unhealthy.

```
$ aws route53 update-health-check --output json --health-check-id 1b01881d-a658-4d8e-86c9-8d5329d0bfcc --inverted
{
  "HealthCheck": {
    "HealthCheckConfig": {
      "HealthThreshold": 2,
      "Type": "CALCULATED",
      "ChildHealthChecks": [
        "57a1606e-df1e-43d0-b6d5-e04bf3d789ff",
        "e4a34329-26c3-4dbe-bec7-4afba94bd487"
      ],
      "Inverted": true
    },
    "HealthCheckVersion": 3,
    "Id": "1b01881d-a658-4d8e-86c9-8d5329d0bfcc"
  }
}
```

3. After a moment for the Health Check status to update, check the primary CDN endpoint to ensure that it has automatically failed over to the secondary CDN Amazon S3 bucket.

```
$ curl http://cdn.bookstore.pingpublications.com/static/status.json
{
  "online": true,
  "cdn": "secondary"
}
```

4. Revert the previous Health Check inversion.

```
$ aws route53 update-health-check --output json --health-check-id 1b01881d-a658-4d8e-86c9-8d5329d0bfcc --no-inverted
{
  "HealthCheck": {
    "HealthCheckConfig": {
      "HealthThreshold": 2,
      "Type": "CALCULATED",
      "ChildHealthChecks": [
        "57a1606e-df1e-43d0-b6d5-e04bf3d789ff",
        "e4a34329-26c3-4dbe-bec7-4afba94bd487"
      ],
      "Inverted": false
    },
    "HealthCheckVersion": 4,
    "Id": "1b01881d-a658-4d8e-86c9-8d5329d0bfcc"
  }
}
```

5. Verify that the primary CDN endpoint points back to the primary Amazon S3 bucket.

```
$ curl http://cdn.bookstore.pingpublications.com/static/status.json
{
  "online": true,
  "cdn": "primary"
}
```

Publish Test Results

Now that both critical dependencies have been tested we'll publish the results to the entire team, so everyone can keep tabs on the resiliency progress of the project.

In this case, both the [Database Failure Test](#) and [CDN Failure Test](#) were performed manually within less than 15 minutes. Therefore, we can safely update some of the playbook information created during the prerequisite phase of [Resiliency Stage 0](#) and reduce the critical dependency RTO/RPO targets significantly. Even a conservative estimate of 1 hour is a massive improvement to resiliency and reduction to potential support costs, and we're only through the second Stage!

Dependency	Criticality Period	Manual Workaround	RTO	RPO
Database	Always	Manual verification of Amazon RDS Multi-AZ secondary instance failover.	1	1
CDN	Always	Manual verification of secondary Amazon S3 bucket failover DNS routing.	1	1

Resiliency Stage 1: Completion

Once all critical dependencies have been failure tested and those test results have been disseminated throughout the team then Resiliency Stage 1 is complete! Your system should now have well-defined recovery playbooks and been manually tested for both failover and critical dependency failures. In [Chaos Engineering Through Staged Resiliency - Stage 2](#) we'll look at the transition into automating some of these tests and performing them at regular intervals.



Chaos Engineering Through Staged Resiliency - Stage 2

Performing occasional, manual resiliency testing is useful, but your system must be automatically and frequently tested to provide any real sense of stability. In [Chaos Engineering Through Staged Resiliency - Stage 1](#) we focused on critical dependency failure testing in non-production environments. To work through **Resiliency Stage 2** your team will need to begin automating these tests and experiments. This allows the testing frequency to improve dramatically and reduces the reliance on manual processes.

Prerequisites

- Creation and agreement on [Disaster Recovery and Dependency Failover Playbooks](#).
- Completion of [Resiliency Stage 0](#).
- Completion of [Resiliency Stage 1](#).

Perform Frequent, Semi-Automated Tests

There's no more putting it off – it's time to begin automating your testing procedures. During this third **Resiliency Stage**, the team should aim to automate as much of the resiliency testing process as reasonably possible. The overall goal of this stage isn't to finalize automation, but to work toward a regular cadence of testing. The frequency of each test is up to the team, but once established it's critical that the schedule is maintained and automation handles at least some of the testing process.

If the team isn't already doing so, this is a prime opportunity to introduce Chaos Engineering tools. These tools empower the team to intelligently create controlled experiments that can be executed precisely when necessary. For example, Gremlin attacks can be [scheduled](#) to execute on certain days of the week and within a specified window of time.

Execute a Resiliency Experiment in Production

Resiliency experiments in non-production are beneficial, but a system is never truly being properly tested unless you're willing to perform experiments in production. Use of internal or third-party Chaos Engineering tools can help with the process of executing a resiliency experiment in production. A tool that configures and executes a given experiment can be used to repeat that experiment over and over, without introducing any issues that may normally be introduced due to human intervention. As experiments are performed and the results evaluated, observability will improve, playbooks are updated, and overall support costs are reduced.

Publish Test Results

As you've been doing thus far, you must continue publishing test results to the entire team. This step is especially critical now that experiments are being performed in production.

Resiliency Stage 2: Implementation Example

How to Automate Blue/Green Instance Failover in AWS

The blue/green deployment for the Bookstore application provides two identical production environment instances of the system. However, if the currently active instance fails we still have to manually swap DNS records from blue to green (or vice versa). In order to meet the requirement of executing a resiliency experiment in production, we need to automate this failover process.

1. Create a metric alarm within Amazon CloudWatch.

```
$ aws cloudwatch put-metric-alarm --output json --cli-input-
json '{
  "ActionsEnabled": true,
  "AlarmActions": [
    "arn:aws:sns:us-west-2:532151327118:PingPublications
SNS"
  ],
  "AlarmDescription": "Bookstore API (Blue) instance
failure.",
  "AlarmName": "bookstore-api-blue-StatusCheckFailed",
  "ComparisonOperator": "GreaterThanOrEqualToThreshold",
  "DatapointsToAlarm": 1,
  "Dimensions": [
    {
      "Name": "InstanceId",
      "Value": "i-0ab55a49288f1da24"
    }
  ],
  "EvaluationPeriods": 1,
  "MetricName": "StatusCheckFailed",
  "Namespace": "AWS/EC2",
  "Period": 60,
  "Statistic": "Maximum",
  "Threshold": 1.0,
  "TreatMissingData": "breaching"
}'
```

2. The above configuration checks the `StatusCheckFailed` metric once a minute, which determines if either the Amazon EC2 system or the specific EC2 instance has failed. We have to set `TreatMissingData` to `breaching` so that the absence of data will trigger an `ALARM` state (otherwise, the instance can be `stopped` or `terminated`, but the `StatusCheck` will succeed).
3. The `Dimensions` array ensures this `bookstore-api-blue-StatusCheckFailed` alarm only checks against the `bookstore-api-blue` instance, and the `AlarmActions` specifies an Amazon SNS ARN to push an alert to when the alarm is triggered.
4. Next, create an Amazon Route53 Health Check that uses the `bookstore-api-blue-StatusCheckFailed` Amazon CloudWatch alarm for its `healthy/unhealthy` determination.

```
$ aws route53 create-health-check --caller-reference bookstore-api-blue-health-check --output json --health-check-config '{
  "InsufficientDataHealthStatus": "Unhealthy",
  "Type": "CLOUDWATCH_METRIC",
  "AlarmIdentifier": {
    "Region": "us-west-2",
    "Name": "bookstore-api-blue-StatusCheckFailed"
  },
  "Inverted": false
}'
{
  "HealthCheck": {
    "HealthCheckConfig": {
      "InsufficientDataHealthStatus": "Unhealthy",
      "Type": "CLOUDWATCH_METRIC",
      "AlarmIdentifier": {
        "Region": "us-west-2",
        "Name": "bookstore-api-blue-StatusCheckFailed"
      },
      "Inverted": false
    },
    "CallerReference": "bookstore-api-blue-health-check",
    "HealthCheckVersion": 1,
    "Id": "119e6884-3685-4e5a-9520-44cebae6c734",
    "CloudWatchAlarmConfiguration": {
      "EvaluationPeriods": 1,
      "Dimensions": [
        {
          "Name": "InstanceId",
          "Value": "i-0ab55a49288f1da24"
        }
      ]
    }
  }
}
```



```

    }
  ],
  "Namespace": "AWS/EC2",
  "Period": 60,
  "ComparisonOperator":
    "GreaterThanOrEqualToThreshold",
  "Statistic": "Maximum",
  "Threshold": 1.0,
  "MetricName": "StatusCheckFailed"
},
{
  "Location": "https://route53.amazonaws.com/2013-04-01/healthcheck/119e6884-3685-4e5a-9520-44cebae6c734"
}
}

```

5. Retrieve the `Id` for the `pingpublications.com` Amazon Route53 hosted zone.

TIP: To reduce AWS costs we're using a 60-second metric evaluation interval here, but in a more demanding application this `Period` should be significantly shorter.

```

$ aws route53 list-hosted-zones --output json
{
  "HostedZones": [
    {
      "ResourceRecordSetCount": 5,
      "CallerReference": "303EC092-EDA3-C2B7-822E-5E609CA718A3",
      "Config": {
        "PrivateZone": false
      },
      "Id": "/hostedzone/Z2EHK3FAEUNRMI",
      "Name": "pingpublications.com."
    }
  ]
}

```

6. Create a pair of Amazon Route53 `Type: A` DNS record sets that will perform a `Failover` routing policy from the `bookstore-api-blue` to the `bookstore-api-green` production environment. Therefore, even in the event that the blue instance fails the above health check, the `bookstore.pingpublications.com` endpoint will always point to an active production instance.

```
$ aws route53 change-resource-record-sets --hosted-zone-id /
hostedzone/Z2EHK3FAEUNRMI --output json --change-batch '{
  "Comment": "Adding bookstore.pingpublications.com
failover.",
  "Changes": [
    {
      "Action": "CREATE",
      "ResourceRecordSet": {
        "Name": "bookstore.pingpublications.com.",
        "Type": "A",
        "SetIdentifier": "bookstore-Primary",
        "Failover": "PRIMARY",
        "TTL": 60,
        "ResourceRecords": [
          {
            "Value": "54.213.54.171"
          }
        ],
        "HealthCheckId": "119e6884-3685-4e5a-9520-
44cebae6c734"
      }
    },
    {
      "Action": "CREATE",
      "ResourceRecordSet": {
        "Name": "bookstore.pingpublications.com.",
        "Type": "A",
        "SetIdentifier": "bookstore-Secondary",
        "Failover": "SECONDARY",
        "TTL": 60,
        "ResourceRecords": [
          {
            "Value": "52.11.79.9"
          }
        ]
      }
    }
  ]
}'
{
  "ChangeInfo": {
    "Status": "PENDING",
    "Comment": "Adding bookstore.pingpublications.com
failover.",
    "SubmittedAt": "2018-11-15T01:54:15.785Z",
    "Id": "/change/C1XJ0G2FTPLYD0"
  }
}
```

7. Double-check that the two failover record sets have been created.

```
$ aws route53 list-resource-record-sets --hosted-zone-id /
hostedzone/Z2EHK3FAEUNRMI --output json
{
  "ResourceRecordSets": [
    {
      "HealthCheckId": "119e6884-3685-4e5a-9520-
44cebae6c734",
      "Name": "bookstore.pingpublications.com.",
      "Type": "A",
      "Failover": "PRIMARY",
      "ResourceRecords": [
        {
          "Value": "54.213.54.171"
        }
      ],
      "TTL": 60,
      "SetIdentifier": "bookstore-Primary"
    },
    {
      "Name": "bookstore.pingpublications.com.",
      "Type": "A",
      "Failover": "SECONDARY",
      "ResourceRecords": [
        {
          "Value": "52.11.79.9"
        }
      ],
      "TTL": 60,
      "SetIdentifier": "bookstore-Secondary"
    }
  ]
}
```

Verifying Automated Instance Failover

To test the failover process for the Bookstore API application instances we'll use a Gremlin [Shutdown Attack](#). This attack will temporarily shutdown the `bookstore-api-blue` Amazon EC2 instance.

1. Verify the current status of the Bookstore API by checking the `bookstore.pingpublications.com/version/` endpoint.

```
$ curl bookstore.pingpublications.com/version/ | jq
{
  "environment": "bookstore-api-blue",
  "version": 7
}
```

Chaos Engineering

2. This shows what app version is running and confirms that the endpoint is pointing to the `bookstore-api-blue` Amazon EC2 instance.
3. Run a Gremlin Shutdown Attack targeting the `bookstore-api-blue` Client. Check out the [Gremlin Help](#) documentation for more details on creating attacks with Gremlin's Chaos Engineering tools.
4. After a moment the Amazon CloudWatch `bookstore-api-blue-StatusCheckFailed` Alarm triggers, thereby causing the `bookstore-api-blue-healthy` Amazon Route53 health check to fail. This automatically triggers the DNS failover created previously, which points the primary `bookstore.pingpublications.com` endpoint to the `green` EC2 production environment. Verify this failover has automatically propagated by checking the `/version/` endpoint once again.

```
$ curl bookstore.pingpublications.com/version/ | jq
{
  "environment": "bookstore-api-blue",
  "version": 7
}
```

How to Generate Custom AWS Metrics

Automating critical dependency failure tests often requires a way to simulate the failure of critical dependencies if those dependencies are third-party and therefore out of our control. This is the case with the **Bookstore** app that relies on Amazon S3 and Amazon RDS for its CDN and database services, respectively. Tools like Gremlin can help with this task. Running a Gremlin **Blackhole Attack** on the **Bookstore** API environment can block all network communication between the API instance and specified endpoints, including those for the CDN and database.

However, AWS EC2 instances don't have any built-in capability to monitor the connectivity between said instance and another arbitrary endpoint. Therefore, the solution is to create some custom metrics that will effectively measure the "health" of the connection between the **Bookstore** API instance and its critical dependencies.

While AWS provides many built-in metrics, creating [custom metrics](#) requires that you explicitly generate all the relevant data for said metric. For this reason, a custom metric is little more than a combination of a metric name and list of [dimensions](#), which are used as [key/value](#) pairs to distinguish and categorize metrics. For our purposes here we're creating simple metrics that act as a health monitor for each critical dependency.

Chaos Engineering

1. We start by creating a couple of bash scripts within our application code repository. These scripts are executed from the **Bookstore** API instances.

```
#!/bin/bash
# ~/apps/bookstore_api/metrics/db-connectivity.sh
INSTANCE_ID="$(ec2metadata --instance-id)"
TAG_NAME="$(/home/ubuntu/.local/bin/aws ec2 describe-
tags --filter "Name=resource-id,Values=$INSTANCE_ID"
"Name=key,Values=Name" --query "Tags[*].Value" --output text)"

while sleep 10
do
    if nc -zv -w 5 db.bookstore.pingpublications.com 5432
2>&1 | grep --line-buffered -i succeeded; then
        /home/ubuntu/.local/bin/aws cloudwatch put-metric-
data --metric-name db-connectivity --namespace Bookstore
--dimensions Host=$TAG_NAME --value 1
    else
        echo "Connection to db.bookstore.pingpublications.com
5432 port [tcp/postgresql] failed!"
        /home/ubuntu/.local/bin/aws cloudwatch put-metric-
data --metric-name db-connectivity --namespace Bookstore
--dimensions Host=$TAG_NAME --value 0
    fi
done

#!/bin/bash
# ~/apps/bookstore_api/metrics/cdn-connectivity.sh
INSTANCE_ID="$(ec2metadata --instance-id)"
TAG_NAME="$(/home/ubuntu/.local/bin/aws ec2 describe-
tags --filter "Name=resource-id,Values=$INSTANCE_ID"
"Name=key,Values=Name" --query "Tags[*].Value" --output text)"

while sleep 10
do
    if nc -zv -w 5 cdn.bookstore.pingpublications.com 80 2>&1
| grep --line-buffered -i succeeded; then
        /home/ubuntu/.local/bin/aws cloudwatch put-metric-
data --metric-name cdn-connectivity --namespace Bookstore
--dimensions Host=$TAG_NAME --value 1
    else
        echo "Connection to cdn.bookstore.pingpublications.
com 80 port [tcp/http] failed!"
        /home/ubuntu/.local/bin/aws cloudwatch put-metric-
data --metric-name cdn-connectivity --namespace Bookstore
--dimensions Host=$TAG_NAME --value 0
    fi
done
```

These scripts use [netcat](#) to check the current network connectivity between the instance and the primary critical dependency endpoint. This check is performed every **10 seconds**. The AWS CLI then generates metric data for the **db-connectivity** and **cdn-connectivity** metrics, passing a value of **1** if the connection succeeded and **0** for a failure.

2. To check the connectivity for the database or CDN endpoint we first retrieve the Amazon EC2 instance Id and, from that, get the **Name** tag value, which is passed as the **Host** dimension when creating our metric. This **Host** value specifies which environment (blue/green) is being evaluated.
3. Create **systemd** service configuration files so the bash scripts will be executed and remain active.
 - Start by creating the **bookstore-db-connectivity** service and pointing the **ExecStart** command to the **db-connectivity.sh** file.

```
sudo nano /etc/systemd/system/bookstore-db-connectivity.service

[Unit]
Description=Bookstore Database Connectivity Service
After=network.target

[Service]
Type=simple
User=ubuntu
Group=ubuntu
WorkingDirectory=/home/ubuntu
ExecStart=/home/ubuntu/apps/bookstore_api/metrics/db-connectivity.sh
Restart=always

[Install]
WantedBy=multi-user.target
```

- Do the same for the CDN.

```
sudo nano /etc/systemd/system/bookstore-db-connectivity.service

[Unit]
Description=Bookstore Database Connectivity Service
After=network.target

[Service]
Type=simple
User=ubuntu
Group=ubuntu
WorkingDirectory=/home/ubuntu
ExecStart=/home/ubuntu/apps/bookstore_api/metrics/cdn-connectivity.sh
Restart=always

[Install]
WantedBy=multi-user.target
```

Chaos Engineering

4. Set permissions such that both service target scripts are executable.

```
chmod +x /home/ubuntu/apps/bookstore_api/metrics/db-  
connectivity.sh && chmod +x /home/ubuntu/apps/bookstore_api/  
metrics/cdn-connectivity.sh
```

5. Enable both services and either reboot the instance or manually start the services.

```
sudo systemctl enable bookstore-db-connectivity && sudo  
systemctl enable bookstore-cdn-connectivity  
  
sudo systemctl start bookstore-db-connectivity && sudo  
systemctl start bookstore-cdn-connectivity
```

6. Verify that both services are active.

```
$ sudo systemctl status bookstore-db-connectivity.service  
Loaded: loaded (/etc/systemd/system/bookstore-db-  
connectivity.service; disabled; vendor preset: enabled)  
Active: active (running) since Mon 2018-11-17 12:02:19 UTC;  
2min 3s ago  
  
Nov 17 12:02:42 bookstore-api-green systemd[1]: Started  
Bookstore Database Connectivity Service.  
Nov 17 12:02:42 bookstore-api-green db-connectivity.  
sh[24683]: Connection to db.bookstore.pingpublications.com  
5432 port [tcp/postgresql] succeeded!  
  
$ sudo systemctl status bookstore-cdn-connectivity  
Loaded: loaded (/etc/systemd/system/bookstore-cdn-  
connectivity.service; disabled; vendor preset: enabled)  
Active: active (running) since Mon 2018-11-17 12:02:19 UTC;  
20s ago  
  
Nov 17 12:02:19 bookstore-api-green systemd[1]: Started  
Bookstore CDN Connectivity Service.  
Nov 17 12:02:31 bookstore-api-green cdn-connectivity.  
sh[24695]: Connection to cdn.bookstore.pingpublications.com 80  
port [tcp/http] succeeded!
```


7. Create Amazon CloudWatch Alarms based on the four generated metrics. These alarms check for a connectivity failure within the last two minutes.

```
aws cloudwatch put-metric-alarm --output json --cli-input-json
'{
  "ActionsEnabled": true,
  "AlarmActions": [
    "arn:aws:sns:us-west-2:532151327118:PingPublications
SNS"
  ],
  "AlarmDescription": "Database connectivity failure for
Bookstore API (Blue).",
  "AlarmName": "bookstore-api-blue-db-connectivity-failed",
  "ComparisonOperator": "LessThanThreshold",
  "DatapointsToAlarm": 2,
  "Dimensions": [
    {
      "Name": "Host",
      "Value": "bookstore-api-blue"
    }
  ],
  "EvaluationPeriods": 2,
  "MetricName": "db-connectivity",
  "Namespace": "Bookstore",
  "Period": 60,
  "Statistic": "Average",
  "Threshold": 1.0
}'
```

```
aws cloudwatch put-metric-alarm --output json --cli-input-json
'{
  "ActionsEnabled": true,
  "AlarmActions": [
    "arn:aws:sns:us-west-2:532151327118:PingPublications
SNS"
  ],
  "AlarmDescription": "CDN connectivity failure for
Bookstore API (Blue).",
  "AlarmName": "bookstore-api-blue-cdn-connectivity-
failed",
  "ComparisonOperator": "LessThanThreshold",
  "DatapointsToAlarm": 2,
  "Dimensions": [
    {
      "Name": "Host",
      "Value": "bookstore-api-blue"
    }
  ],
  "EvaluationPeriods": 2,
  "MetricName": "cdn-connectivity",
  "Namespace": "Bookstore",
  "Period": 60,
  "Statistic": "Average",
  "Threshold": 1.0
}'
```

```

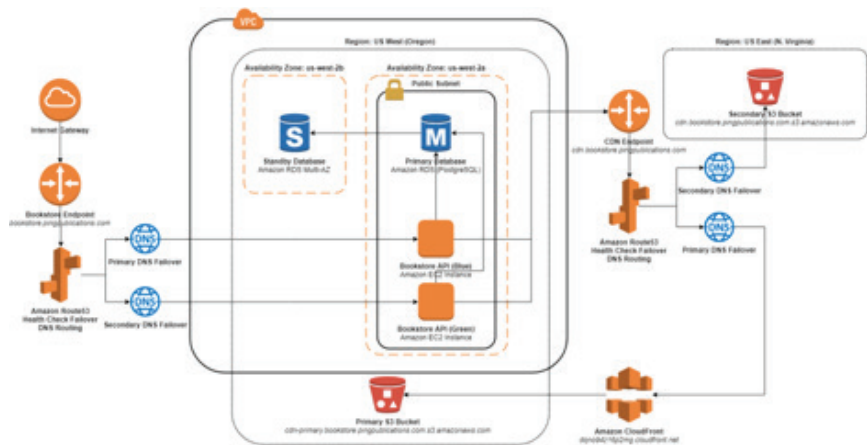
        "Threshold": 1.0
    }'

aws cloudwatch put-metric-alarm --output json --cli-input-json
'{
    "ActionsEnabled": true,
    "AlarmActions": [
        "arn:aws:sns:us-west-2:532151327118:PingPublications
SNS"
    ],
    "AlarmDescription": "Database connectivity failure for
Bookstore API (Green).",
    "AlarmName": "bookstore-api-green-db-connectivity-
failed",
    "ComparisonOperator": "LessThanThreshold",
    "DatapointsToAlarm": 2,
    "Dimensions": [
        {
            "Name": "Host",
            "Value": "bookstore-api-green"
        }
    ],
    "EvaluationPeriods": 2,
    "MetricName": "db-connectivity",
    "Namespace": "Bookstore",
    "Period": 60,
    "Statistic": "Average",
    "Threshold": 1.0
}'

aws cloudwatch put-metric-alarm --output json --cli-input-json
'{
    "ActionsEnabled": true,
    "AlarmActions": [
        "arn:aws:sns:us-west-2:532151327118:PingPublications
SNS"
    ],
    "AlarmDescription": "CDN connectivity failure for
Bookstore API (Green).",
    "AlarmName": "bookstore-api-green-cdn-connectivity-
failed",
    "ComparisonOperator": "LessThanThreshold",
    "DatapointsToAlarm": 2,
    "Dimensions": [
        {
            "Name": "Host",
            "Value": "bookstore-api-green"
        }
    ],
    "EvaluationPeriods": 2,
    "MetricName": "cdn-connectivity",
    "Namespace": "Bookstore",
    "Period": 60,
    "Statistic": "Average",
    "Threshold": 1.0
}'

```

Here is the updated architecture diagram for the **Bookstore** app at the end of **Resiliency Stage 2**.



Bookstore App Architecture

Performing a DB Failure Simulation Test

To meet the **Stage 2** criteria of performing “frequent, semi-automated tests” we can use Chaos Engineering tools like Gremlin to easily schedule automated attacks on relevant services and machines. This allows your team to properly prepare for, evaluate, and respond to the outcome of these tests.

For the **Bookstore** example application, we’re simulating a failure of the database by preventing the blue environment from establishing a database connection.

1. Schedule a Gremlin **Blackhole** Attack targeting the **bookstore-api-blue** environment that specifies the primary database endpoint (**db.bookstore.pingpublications.com**) as a **Hostname**. Check out the [Gremlin API](#) or [documentation](#) for more details on creating Gremlin Attacks.
2. This will automatically simulate a failure of the **bookstore-api-blue** instance’s ability to connect to the primary database endpoint, which causes the custom **db-connectivity** metrics to fail and trigger the subsequent Amazon Cloudwatch alarm.

3. Check the status of the `bookstore-db-connectivity` service to confirm that the health check is now failing.

```
$ sudo systemctl status bookstore-db-connectivity.service
[...]
Nov 25 21:18:02 bookstore-api-blue db-connectivity.sh[785]:
Connection to db.bookstore.pingpublications.com 5432 port
[tcp/postgresql] succeeded!
Nov 25 21:18:18 bookstore-api-blue db-connectivity.sh[785]:
Connection to db.bookstore.pingpublications.com 5432 port
[tcp/postgresql] failed!
Nov 25 21:18:49 bookstore-api-blue db-connectivity.sh[785]:
Connection to db.bookstore.pingpublications.com 5432 port
[tcp/postgresql] failed!
```

4. Use `aws cloudwatch get-metric-statistics` to check the `db-connectivity` metric around this time to see the actual metric values reported in AWS.

```
$ aws cloudwatch get-metric-statistics --metric-name db-
connectivity --start-time 2018-11-25T21:17:00Z --end-
time 2018-11-25T21:24:00Z --period 60 --namespace
Bookstore --statistics Average --dimensions
Name=Host,Value=bookstore-api-blue --query "Datapoints[*].
{Timestamp:Timestamp,Average:Average}" --output json
[
  {
    "Timestamp": "2018-11-25T21:20:00Z",
    "Average": 0.0
  },
  {
    "Timestamp": "2018-11-25T21:18:00Z",
    "Average": 0.25
  },
  {
    "Timestamp": "2018-11-25T21:19:00Z",
    "Average": 0.0
  },
  {
    "Timestamp": "2018-11-25T21:23:00Z",
    "Average": 1.0
  },
  {
    "Timestamp": "2018-11-25T21:17:00Z",
    "Average": 1.0
  },
  {
    "Timestamp": "2018-11-25T21:22:00Z",
    "Average": 0.8
  },
  {
    "Timestamp": "2018-11-25T21:21:00Z",
    "Average": 0.0
  }
]
```

Since an *actual* database failure is handled by the Multi-AZ Amazon RDS configuration created in [Stage 1 - Database Failure Test](#), this automated critical dependency failure test doesn't currently trigger any actual failover actions. However, alarms are now configured to easily hook into disaster recovery actions as progress is made through the final **Resiliency Stages**.

- Here we see the `db-connectivity` health is acceptable (1.0) until it starts to drop around `21:18:00`, then remains at 0.0 for a few minutes before returning to healthy status.
- This metric failure triggered the `bookstore-api-blue-db-connectivity-failed` Amazon Cloudwatch alarm that we also created. This can be confirmed by checking the alarm history status around that same time period.

```
$ aws cloudwatch describe-alarm-history --alarm-name bookstore-api-blue-db-connectivity-failed --history-item-type StateUpdate --start-date 2018-11-25T21:17:00Z --end-date 2018-11-25T21:28:00Z --output json --query "AlarmHistoryItems[*].{Timestamp:Timestamp,Summary:HistorySummary}"
[
  {
    "Timestamp": "2018-11-25T21:24:09.620Z",
    "Summary": "Alarm updated from ALARM to OK"
  },
  {
    "Timestamp": "2018-11-25T21:20:09.620Z",
    "Summary": "Alarm updated from OK to ALARM"
  }
]
```

- Here we see that the alarm was triggered at `21:20:09.620` and remained for four minutes before returning to OK status.

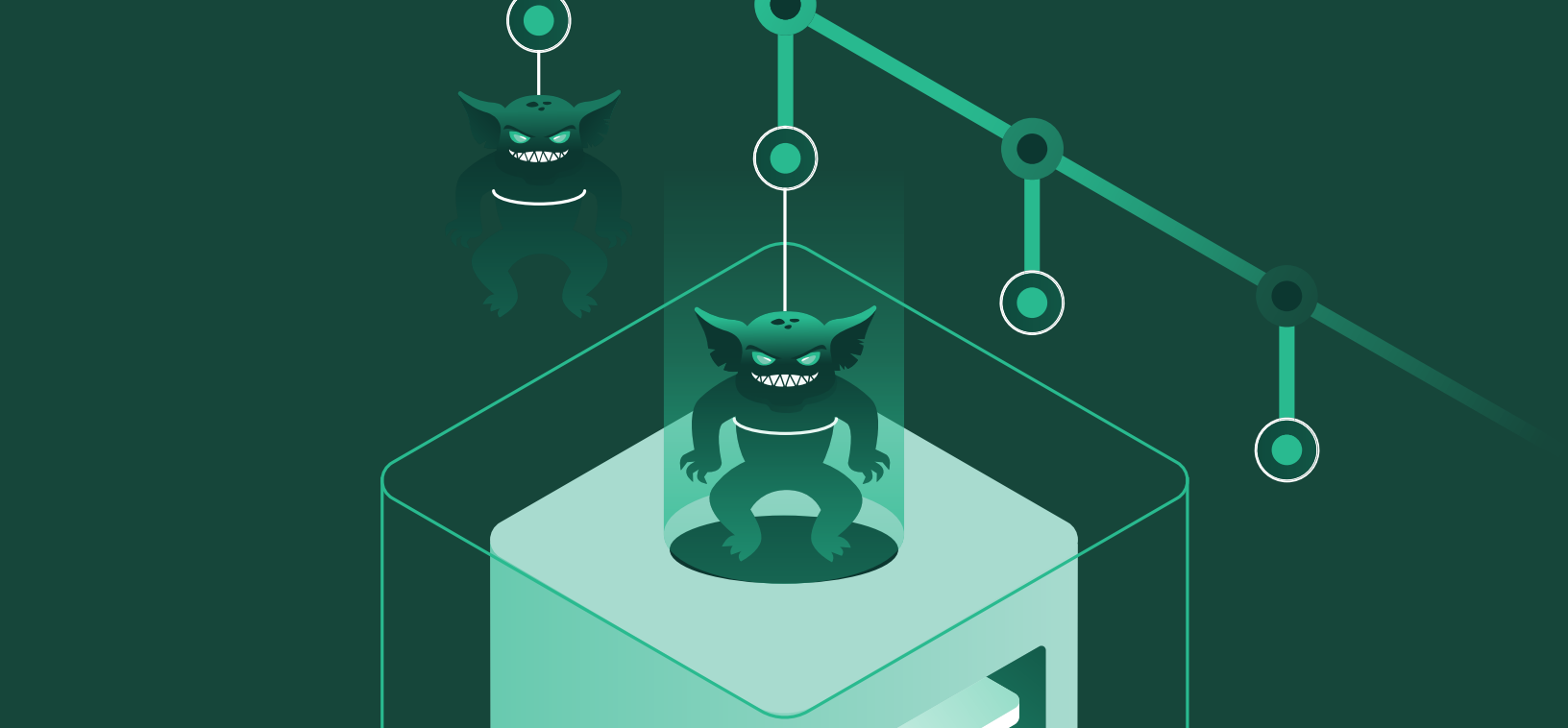
Performing a CDN Failure Simulation Test

We use the same steps as above to perform a scheduled, automated Gremlin `BlackholeAttack` to simulate failure of the CDN for our `bookstore-api-blue` environment. Simply changing the `db-` references to `cdn-` and the relevant endpoints will do the trick. However, for brevity's sake, we won't include the step-by-step instructions for doing so within this section.

Resiliency

Stage 3: Completion

Your team has been performing semi-automated tests at a regular cadence, has executed at least one resiliency experiment in production, and has disseminated all test results to the entire team. With that, **Resiliency Stage 2** is now complete. Revenue loss and support costs should finally be dramatically dropping as fewer failures occur, and those that do happen last for much shorter periods. In [Chaos Engineering Through Staged Resiliency - Stage 3](#) we'll explore automating resiliency testing in non-production, along with semi-automation of our disaster recovery failovers.



Chaos Engineering Through Staged Resiliency - Stage 3

In [Chaos Engineering Through Staged Resiliency - Stage 2](#) we examined partial automation by implementing a handful of automated resiliency tests. Now, as we progress through **Resiliency Stage 3** it's time to fully embrace automation: All resiliency testing in non-production environments should be completely automated, requiring little to no manual interaction. After completing the entirety of this **Resiliency Stage** your application will be quite resilient and – at least outside of production – should require minimal supervision and support costs. Let's get to it!

Prerequisites

- Creation and agreement on [Disaster Recovery and Dependency Failover Playbooks](#).
- Completion of [Resiliency Stage 0](#).
- Completion of [Resiliency Stage 1](#).
- Completion of [Resiliency Stage 2](#).

Automate Resiliency Testing in Non-Production

After progressing through [Resiliency Stage 2](#) your team implemented at least some semi-automated resiliency testing. However, this fourth stage is where all non-automated resiliency tests must also be integrated into your automated testing suite. If your application features a development or other non-production environment, you can opt to integrate these automated resilience tests in that non-production environment, as full-blown production testing isn't required until [Stage 4](#). However, the earlier the team starts thinking about and practicing implementation within production systems, the smoother the transition will be and the sooner you'll see that dramatic increase in resilience and drop in support costs that staged resiliency aims to provide.

Semi-Automate Disaster Recovery Failover

In spite of everyone's best efforts, not all disasters can be avoided, so it's critical that the team implement at least a *semi*-automated disaster recovery failover script. As with resiliency testing, it's best to automate as much of the disaster recovery failover process as possible, requiring as little human intervention as feasible. However, depending on the breadth of the system and initial planning throughout the earlier **Resiliency Stages**, it's entirely possible your disaster recovery failover will require at least a modicum of human supervision.

As the team progresses through this stage make sure you follow the playbooks that have been previously established. If something needs to be changed in a process or playbook, this is the time to suss that out and make those updates.

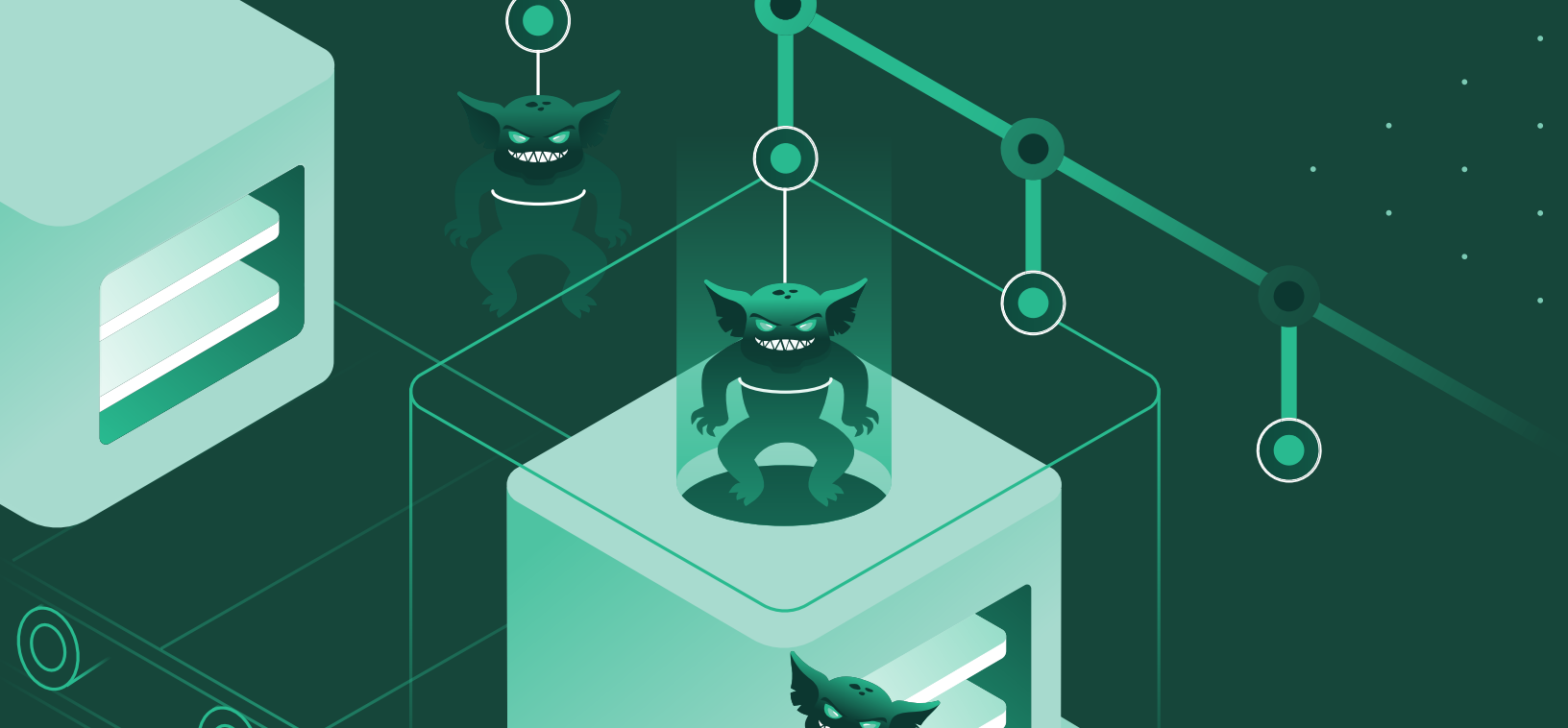
Resiliency Stage 3: Implementation Example

As will sometimes be the case when your own team is working through each **Stage of Resiliency**, the **Bookstore** application has already been configured to automatically perform resiliency testing in non-production environments. In **Stage 2** we explored [Performing a CDN Failure Simulation Test](#) and [Performing a DB Failure Simulation Test](#), which handles the major resiliency tests for the system by creating Gremlin attacks to sever the connection between the `bookstore-api` instances and the respective CDN/DB endpoints.

To ensure these tests are performed automatically, we can use the [Gremlin API](#) or [web front-end](#) to *automatically schedule* attacks for our given testing schedule. Similarly, we'd want to schedule an automatic disaster recovery failover test using a Gremlin [Shutdown Attack](#), as illustrated in [Verifying Automated Instance Failover](#) in **Stage 2**. Check out the [Gremlin documentation](#) for more details on creating attacks with Gremlin.

Resiliency Stage 3: Completion

You've automated resiliency testing in a non-production environment (and, ideally, even a bit in production). Your team has also semi-automated disaster recovery failover procedures to ensure your service can moderately recover itself after a failure, with minimal human intervention. In the last chapter of this series, [Chaos Engineering Through Staged Resiliency - Stage 4](#), we'll explore the final steps of fully automating resiliency testing in production, along with CI/CD integration to ensure your service maintains stability throughout every step of the software development life cycle.



Chaos Engineering Through Staged Resiliency - Stage 4

We've finally made it to the last **Resiliency Stage**, where your team will be working toward full automation of both resiliency testing and disaster recovery failover procedures. [Chaos Engineering Through Staged Resiliency - Stage 3](#) emphasized the importance of looking toward the production environment for automation, but this final **Resiliency Stage 4** is where all remaining non-automated tests and processes must be automated within production. That's not to say you *cannot* have any human interactions involved, but rather, that a given resiliency testing or disaster recovery process must not *rely* on human intervention to succeed.

Once your team has progressed through this final stage your system will be *demonstrably* more resilient and will require *far* fewer support hours, including reduced support costs in the event of a failure.

Prerequisites

- Creation and agreement on [Disaster Recovery and Dependency Failover Playbooks](#).
- Completion of [Resiliency Stage 0](#).
- Completion of [Resiliency Stage 1](#).
- Completion of [Resiliency Stage 2](#).
- Completion of [Resiliency Stage 3](#).

Integrate Automatic Resiliency Testing in CI/CD

For systems relying on continuous integration and continuous deployment, this final stage is when you should integrate automatic resiliency testing within the CI/CD process.

Additionally, a failed resiliency test should also result in a build and/or deployment failure. Just as with application-level unit testing, resiliency testing should always be 100% successful prior to releasing a given build, so this prerequisite helps maintain system stability.

Automate Resiliency and Disaster Recovery Failover Testing in Production

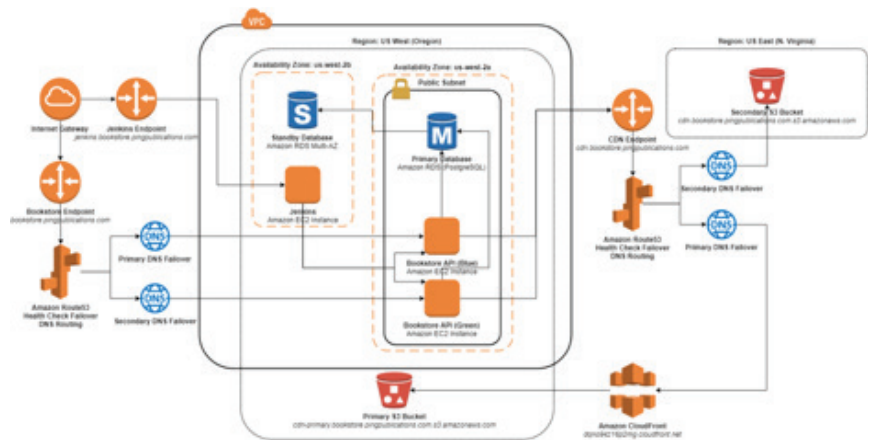
The final and most critical requirement for **Resiliency Stage 4** is the nearly-total automation of both resiliency and disaster recovery failover testing in the **production** environment. An SRE can be involved, but everything should be so automated that running tests merely requires a minimal amount of support time to get the process started.

Resiliency Stage 4: Implementation Example

To keep things simple the **Bookstore** application uses [Jenkins](#) to handle CI/CD. We've propagated a new Amazon EC2 instance onto which Jenkins is installed. An Amazon Route53 DNS record points the `jenkins.bookstore.pingpublications.com` endpoint to the EC2 instance, so accessing the Jenkins front end is done through `http://jenkins.bookstore.pingpublications.com:8080`.

The steps for installing and running Jenkins on an Amazon EC2 instance are beyond the scope of this article, but a [traditional deployment](#) is easy to accomplish.

This is now what the final **Bookstore** architecture looks like.



Bookstore App Architecture

Jenkins Configuration

With Jenkins installed we need to create a new **Project** for the **Bookstore** application. We're using the [Publish Over SSH](#) plugin to simplify deployment to both the blue/green Amazon EC2 instances. Once the plugin is installed it must be configured by adding each SSH server we'll be interacting with.

1. Navigate to **Manage Jenkins > Configure System** and scroll down to the **Publish over SSH section**.
2. Enter the appropriate private key allowing connection to both **Bookstore API** Amazon EC2 instances.
3. Under **SSH Servers** click **Add**.
4. Enter the appropriate details for the `bookstore-api-blue` server.

Name: `bookstore-api-blue`

Hostname: `54.213.54.171`

Username: `ubuntu`

Remote Directory: `/home/ubuntu/apps`

5. Add another entry for the `bookstore-api-green` server.

Name: `bookstore-api-green`

Hostname: `52.11.79.9`

Username: `ubuntu`

Remote Directory: `/home/ubuntu/apps`

6. Click **Save**.

With the SSH connections configured it's time to create the **Project** to handle actual deployment.

1. Click **New Item**.
2. Input `bookstore-deploy` in the **Item name** field.
3. Select **Freestyle Project** type and click **OK**.

4. Under **Source Code Management** select **Git** and enter the GitHub endpoint (e.g. `git@github.com:GabeStah/bookstore_api.git`).
5. Add the appropriate GitHub credentials.

Deploying the **Bookstore** application code is just a matter of creating an **SSH Publisher** for each environment.

1. Under **Build** click **Add build step** and select **Send files or execute commands over SSH**.
2. Under **SSH Server > Name** select `bookstore-api-blue`.
3. In the **Source files** field input `**/*`. This field uses the [Apache Ant](#) pattern format, so here we're merely ensuring all files in the project directory are published.
4. For **Remote directory** input `bookstore_api`.
5. Since the application uses Unicorn for the web server we need to add a command to restart it after deployment. Input `sudo systemctl restart gunicorn` in the **Exec command** field.
6. Click **Add Server** and repeat steps 2 through 5 for the `bookstore-api-green` server.
7. Click **Save** to finalize the settings. The `bookstore-deploy` **Project** will now deploy new builds of the **Bookstore** to each production environment!

Automating Resiliency Testing in Jenkins

To meet the requirements of **Resiliency Stage 5** we need to integrate automated resiliency testing into the CI/CD pipeline. For the **Bookstore** application, we'll use a Gremlin [Blackhole Attack](#). This attack blocks all network communication

between the targeted instance and specified endpoints. Just as we saw in [Stage 2](#) we'll be using this attack to perform a resiliency test that interrupts communication and triggers an Amazon CloudWatch alarm. Check out the [Gremlin Help](#) documentation for more details on creating attacks with Gremlin's Chaos Engineering tools.

Within Jenkins, we can initiate the resiliency test by executing over SSH.

1. Navigate to the **Configure** section of the `bookstore-deploy` **Project**.
2. Under **Build** click **Add build step** and select **Send files or execute commands over SSH**.
3. Under **SSH Server > Name** select `bookstore-api-blue`.
4. In the **Exec command** field input the following:

```
gremlin attack blackhole -l 240 -h ^api.gremlin.com,cdn.
bookstore.pingpublications.com,db.bookstore.pingpublications.
com
```

TIP: This Gremlin Blackhole attack tests resiliency against the simulated failure of our critical dependencies (database and CDN). Consequently, this will trigger the respective `bookstore-api-{blue/green}-{db/cdn}-connectivity-failed` Amazon CloudWatch Alarms that we configured during [Resiliency Stage 2](#). This causes a DNS failover, which can be confirmed just as it was in **Stage 3**.

5. Click **Add Server** and repeat steps 3 and 4 for the `bookstore-api-green` server.
6. Click **Save**.

TIP: This Gremlin Shutdown attack performs a disaster recovery failover test by terminating the `bookstore-api-blue` instance. We can also optionally add the `-r` flag to have the server restart itself after shutdown. This triggers the `bookstore-api-blue-StatusCheckFailed` Amazon CloudWatch alarm that was created within [Resiliency Stage 2](#). This automatically triggers Amazon Route53 DNS failover to reroute the `bookstore.pingpublications.com` endpoint to the `bookstore-api-green` environment.

Automating a Disaster Recovery Failover Test in Jenkins

We also want to automate disaster recovery failover testing within the CI/CD pipeline. This time we'll use a Gremlin [Shutdown Attack](#), which shuts down the targeted instance.

1. Navigate to the **Configure** section of the `bookstore-deploy` **Project**.
2. Under **Build** click **Add build step** and select **Send files or execute commands over SSH**.
3. Under **SSH Server > Name** select `bookstore-api-blue`.
4. In the **Exec command** field input the following: `gremlin attack shutdown -d 1`.
5. Click **Save**.

Performing a Jenkins Build

Everything is now configured for our simple **Bookstore** application to be automatically deployed via Jenkins, during which resiliency testing and disaster recovery failover testing is performed. It may be ideal to further automate the build process with Jenkins by configuring a **Build Trigger**, but we can also manually perform a build to confirm it works properly.

1. Navigate to the `bookstore-deploy` **Project**.
2. Click **Build Now**.
3. Click **Console Output** to view the output generated by Jenkins. It will look something like the following.


```

Started by user Gabe
  Building in workspace /var/lib/jenkins/workspace/bookstore-deploy
  > git rev-parse --is-inside-work-tree # timeout=10
  Fetching changes from the remote Git repository
  > git config remote.origin.url https://github.com/GabeStah/bookstore_api # timeout=10
  Fetching upstream changes from https://github.com/GabeStah/bookstore_api
  > git --version # timeout=10
  using GIT_ASKPASS to set credentials gabestah@github.com
  > git fetch --tags --progress https://github.com/GabeStah/bookstore_api +refs/heads/*:refs/remotes/origin/*
  > git rev-parse refs/remotes/origin/master^{commit} # timeout=10
  > git rev-parse refs/remotes/origin/origin/master^{commit} # timeout=10
  Checking out Revision
  d7172265ec23ab20d9eaabc6f3dd37a6c741f2cc (refs/remotes/origin/master)
  > git config core.sparsecheckout # timeout=10
  > git checkout -f d7172265ec23ab20d9eaabc6f3dd37a6c741f2cc
  Commit message: "bumped"
  > git rev-list --no-walk
  d7172265ec23ab20d9eaabc6f3dd37a6c741f2cc # timeout=10
  SSH: Connecting from host [ip-172-31-43-207.us-west-2.compute.internal]
  SSH: Connecting with configuration [bookstore-api-blue] ...
  SSH: EXEC: STDOUT/STDERR from command [sudo systemctl restart gunicorn] ...
  SSH: EXEC: completed after 401 ms
  SSH: Disconnecting configuration [bookstore-api-blue] ...
  SSH: Transferred 20 file(s)
  SSH: Connecting from host [ip-172-31-43-207.us-west-2.compute.internal]
  SSH: Connecting with configuration [bookstore-api-green] ...
  SSH: EXEC: STDOUT/STDERR from command [sudo systemctl restart gunicorn] ...
  SSH: EXEC: completed after 200 ms
  SSH: Disconnecting configuration [bookstore-api-green] ...
  SSH: Transferred 20 file(s)
  Build step 'Send files or execute commands over SSH' changed build result to SUCCESS
  SSH: Connecting from host [ip-172-31-43-207.us-west-2.compute.internal]
  SSH: Connecting with configuration [bookstore-api-blue] ...
  SSH: EXEC: STDOUT/STDERR from command [gremlin attack blackhole -l 240 -h ^api.gremlin.com,cdn.bookstore.pingpublications.com,db.bookstore.pingpublications.com] ...
  Setting up blackhole gremlin with guid '0367cfe2-f9fc-11e8-acea-0242db4d1180' for 240 seconds

```

```

Setup successfully completed
Running blackhole gremlin with guid '0367cfe2-f9fc-11e8-acea-
0242db4d1180' for 240 seconds
Whitelisting all egress traffic to 54.186.219.32
Whitelisting all egress traffic to 54.68.250.40
Dropping all egress traffic to 52.84.25.207
Dropping all egress traffic to 52.84.25.199
Dropping all egress traffic to 52.84.25.64
Dropping all egress traffic to 52.84.25.16
Dropping all egress traffic to 172.31.22.69
Whitelisting all ingress traffic from 54.186.219.32
Whitelisting all ingress traffic from 54.68.250.40
Dropping all ingress traffic from 52.84.25.207
Dropping all ingress traffic from 52.84.25.199
Dropping all ingress traffic from 52.84.25.64
Dropping all ingress traffic from 52.84.25.16
Dropping all ingress traffic from 172.31.22.69
Dropping all egress traffic to 13.33.147.150
Dropping all egress traffic to 13.33.147.73
Dropping all egress traffic to 13.33.147.19
Dropping all egress traffic to 13.33.147.18
Dropping all ingress traffic from 13.33.147.150
Dropping all ingress traffic from 13.33.147.73
Dropping all ingress traffic from 13.33.147.19
Dropping all ingress traffic from 13.33.147.18
Dropping all egress traffic to 13.32.253.5
Dropping all egress traffic to 13.32.253.244
Dropping all egress traffic to 13.32.253.105
Dropping all egress traffic to 13.32.253.26
Dropping all ingress traffic from 13.32.253.5
Dropping all ingress traffic from 13.32.253.244
Dropping all ingress traffic from 13.32.253.105
Dropping all ingress traffic from 13.32.253.26
Reverting impact!
SSH: EXEC: completed after 240,397 ms
SSH: Disconnecting configuration [bookstore-api-blue] ...
SSH: Transferred 0 file(s)
SSH: Connecting from host [ip-172-31-43-207.us-west-2.
compute.internal]
SSH: Connecting with configuration [bookstore-api-green] ...
SSH: EXEC: STDOUT/STDERR from command [gremlin attack
blackhole -l 240 -h ^api.gremlin.com,cdn.bookstore.
pingpublications.com,db.bookstore.pingpublications.com] ...
Setting up blackhole gremlin with guid '932c78b3-f9fc-11e8-
9244-024280df5a87' for 240 seconds
Setup successfully completed
Running blackhole gremlin with guid '932c78b3-f9fc-11e8-9244-
024280df5a87' for 240 seconds

```

```
Whitelisting all egress traffic to 54.186.219.32
Whitelisting all egress traffic to 54.68.250.40
Dropping all egress traffic to 13.33.147.150
Dropping all egress traffic to 13.33.147.73
Dropping all egress traffic to 13.33.147.19
Dropping all egress traffic to 13.33.147.18
Dropping all egress traffic to 172.31.22.69
Whitelisting all ingress traffic from 54.186.219.32
Whitelisting all ingress traffic from 54.68.250.40
Dropping all ingress traffic from 13.33.147.150
Dropping all ingress traffic from 13.33.147.73
Dropping all ingress traffic from 13.33.147.19
Dropping all ingress traffic from 13.33.147.18
Dropping all ingress traffic from 172.31.22.69
Dropping all egress traffic to 52.84.25.199
Dropping all egress traffic to 52.84.25.207
Dropping all egress traffic to 52.84.25.16
Dropping all egress traffic to 52.84.25.64
Dropping all ingress traffic from 52.84.25.64
Dropping all ingress traffic from 52.84.25.16
Dropping all ingress traffic from 52.84.25.207
Dropping all ingress traffic from 52.84.25.199
Reverting impact!
SSH: EXEC: completed after 240,395 ms
SSH: Disconnecting configuration [bookstore-api-green] ...
SSH: Transferred 0 file(s)
SSH: Connecting from host [ip-172-31-43-207.us-west-2.
compute.internal]
SSH: Connecting with configuration [bookstore-api-blue] ...
SSH: EXEC: STDOUT/STDERR from command [gremlin attack
shutdown -d 1] ...
Setting up shutdown gremlin with guid '22cdba53-f9fd-11e8-
8292-024230fbb3f0' after 1 minute
Setup successfully completed
Running shutdown gremlin with guid '22cdba53-f9fd-11e8-8292-
024230fbb3f0' after 1 minute
SSH: Disconnecting configuration [bookstore-api-blue] ...
SSH: EXEC: completed after 60,450 ms
Finished: SUCCESS
```

WARNING: The above Jenkins deployment configuration for the **Bookstore** example app is merely a functional proof of concept. For an actual production configuration, we'd want to add many more safeguards, such as automating the alternation of deployment between blue/green environments, to ensure one environment is always ready in the event of a rollback.

The initial build steps grab the latest version via Git, deploy the new app version to both the blue/green environments via SSH, then perform the resiliency and disaster recovery tests via Gremlin. As expected, all relevant `bookstore-api-{blue/green}-{db/cdn}-connectivity-failed` Amazon CloudWatch Alarms are triggered, forcing Amazon Route53 and Amazon RDS to engage its automated failover policies that we established in previous **Resiliency Stages**.

Resiliency Stage 4: Completion

With **Resiliency Stage 4** finished you and your team have reached the end of the current journey, but site reliability engineering is an ongoing process. Your system should now be fully monitored, provide high observability, and automatically perform resiliency and disaster recovery testing in the production environment at regular intervals.

Working through all the **Stages of Resiliency** takes time and will invariably be more difficult for some teams, so do not be discouraged. For organizations with multiple teams, the integration of this staged process allows for teams, as well as individuals within said teams, to be empowered to make improvements and be responsible for the services under their purview. As teams progress further through the stages, overall support costs will drop dramatically, while system stability and resiliency will increase.

Chaos Engineering Tools Comparison

This article describes some of the common tools that the Chaos Engineering community considers when starting to implement the practice in an organization. The goal is to give a high level introduction to some frequently mentioned options and list some of the strengths of each.

Our team at Gremlin has a combined decades of experience implementing Chaos Engineering at companies like Netflix, Amazon, Google, Salesforce, and Dropbox. We have created and implemented a variety of these tools multiple times, and we have found that the features listed in this table are critical for widespread adoption of a Chaos Engineering practice in most enterprise engineering organizations. Following the table is a brief description of each entry.

These features center over the common areas like the variety of chaos experiments and the portability of the code, but also critical safety and security features. In particular, safety and security features help adoption with IT, and are difficult to build yourself. Anything that makes the lives of DevOps and site reliability engineers easier is worth consideration.

Chaos Engineering

Tool	(N)umber of attack types available	Enterprise support	Can halt attack in progress	GUI	Open source	Software as a service (SaaS)	Cloud agnostic	Can attack container / pod	Can attack serverless	Has API
Bloomberg Powerful Seal	1			X	X			X		X
ChaosIQ Chaos Toolkit	3	Via vendor			X	X	X	X		
Gremlin Free	2		X	X		X	X	X	X	X
Gremlin ILFI	11	X	X	X		X	X	X	X	X
Gremlin ALFI	1	X	X	X		X	X	X	X	X
Istio	2				X		X	X		
Mesosphere DRAX	1						X			
Netflix Chaos Monkey	1				X					
Netflix Simian Army	2				X					
New Relic Chaos Panda	2			X						
Nutanix X-Ray	1				X	X				
Pumba	6				X			X		
Shopify ToxiProxy	5				X		X			
T-Mobile Turbulence	5				X		X			

Bloomberg Powerful Seal

From Bloomberg, [Powerful Seal](#) is an open source tool written in Python designed for testing Kubernetes clusters. It works on nodes, pods, containers, and namespaces. It natively works with AWS, OpenStack, and local machines. You can run experiments directly or through automation. No official support is available, but documentation is available and development is active.

ChaosIQ Chaos Toolkit

[Chaos Toolkit](#) is an open source project written in Python that defines an API to help you run chaos experiments that you define. The project uses a system of drivers, plugins, and extensions to allow you to customize and automate the experiments you design. Designing your own experiments and assembling the right set of pieces gives you great flexibility, but with the risk of complexity and engineering time to properly implement. Commercial support is available from ChaosIQ.

Gremlin

[Gremlin](#) is a commercial software as a service (SaaS) offering focused on enterprise customers and others with large-scale deployments. It includes multiple attack possibilities and the ability to halt an attack in progress and rollback should problems occur. You can test in development, in testing, in your CI/CD process, and in production. Gremlin is controlled via either an API or a clear and easy to understand web-accessible graphical user interface. Either method includes significant security features including encryption and granular user account permissions. The latter feature enables teams to designate who has administrator access to control Gremlin attacks and also limit some user accounts to specific functions, thereby limiting risk to the old only give the level of permissions necessary to complete a task standard. Gremlin also has the ability to control application-level fault

injection (ALFI) attacks, making it possible to use request-level metadata to construct your own attacks to be scheduled and controlled by Gremlin.

Istio

Istio is a service mesh that includes some features that you can use for chaos experiments, because the istio-proxy is already intercepting all network traffic. That means the proxy can be used to change the responses or delay responses to simulate latency, provided the request you want to target is a part of your service mesh.

Mesosphere DRAX

From Mesosphere, [DRAX](#), a container-level, DC/OS-specific resilience testing tool inspired by Netflix's Chaos Monkey. DC/OS is Mesosphere's datacenter operating system and cloud automation platform. DRAX runs as a Marathon app, killing off random tasks of any (non-framework) or specific (non-framework) application running in Marathon, which is Mesosphere's container orchestration platform.

Netflix Chaos Monkey

From Netflix, [Chaos Monkey](#) is the first of all Chaos Engineering tools, the one that started it all. If a person has only heard of one tool, this is the most likely candidate. Netflix created Chaos Monkey as they were moving from an on-site to an AWS cloud deployment. Once Netflix realized that their servers and nodes were no longer under their complete control, but rather they were choosing to trust a vendor, they decided it would be a good idea to figure out what would happen if one of those nodes suddenly went down, so they could mitigate against any customer-facing problems that might arise. Chaos Monkey does just that, in production. It pseudorandomly reboots hosts to suss out weaknesses and/or validate that automated

remediation worked correctly. The Chaos Monkey code was released as open source, but is essentially unmaintained and unsupported, so if you choose to use it you are taking on those responsibilities yourself. In addition, it requires that you deploy using Spinnaker, which is not a bad thing, but simply a limitation to consider.

Netflix Simian Army

Also from Netflix, [Simian Army](#) is the logical evolution from Chaos Monkey. It is a suite of Chaos Engineering tools designed to expand the types of failure that can be induced while experimenting to find weaknesses en route to enhancing resilience. Parts of Simian Army have been rolled into Spinnaker while other parts have been either released as somewhat-maintained, standalone open source projects or deprecated and never released because they were written as internal, private tools.

New Relic Chaos Panda

From New Relic, [Chaos Panda](#) is designed to help you implement Chaos Engineering with New Relic's GraphQL API. Internal teams can configure GraphQL in pre-production testing to do things like add latency to query responses or to cause certain fields to fail at a specific failure rate. This tool appears to be limited currently to internal New Relic teams, but is interesting enough to warrant a mention here.

Nutanix X-Ray

From Nutanix, [X-Ray](#) is a Chaos Engineering tool designed to test infrastructures based on their Acropolis Hypervisor (AHV) or VMware's ESXi hypervisor. It can test for node availability, performance, provisioning, and data integrity. Test scenarios can also be customized, shared and imported into new X-Ray deployments. X-Ray is packaged as a VM and uses workloads

paired with real-world scenarios to simulate typical workflows and events for their platform. Usage requires that you are already using Nutanix and an additional registration.

Pumba

[Pumba](#) is an open source project written in Go focused on chaos testing for Docker. It simulates failures related to processes, containers that stop or disappear, and various network and performance issues. The code and binaries are available from GitHub. No official support is available, but documentation is available and development is active.

Shopify ToxiProxy

From Shopify, [ToxiProxy](#) is a TCP proxy written in Go and created to simulate network connections and conditions while your application is in development, testing, and CI environments. It has multiple attack vectors available within those constraints and the code is released as an open source project. There is no official support available, so implementation is up to you, however, the documentation seems pretty clear and complete. [UnderArmor](#) started their Chaos Engineering practice using ToxiProxy, but has since moved to Gremlin.

T-Mobile Turbulence

From T-Mobile, [Turbulence](#) is an extension of Chaos Toolkit that, like T-Mobile's microservices, runs in a Cloud Foundry BOSH environment and can take an organization name, space name, and application name in and block either access to that application, or that application's access to one or more of its bound services. Turbulence and its drivers have been released as open source projects by T-Mobile. No official support is available, but documentation is available and development is active.

Others

There are [other tools](#) that you can use to perform chaos experiments. Any technology component in your stack can theoretically be targeted with some form of intentional failure to see how the rest of your stack handles that failure. You can find ways to attack that are minimal, well-bounded, and carefully limited to minimize the blast radius. The main thing to keep in mind at all times is that you are seeking resilience, looking for areas of weakness that you can work to strengthen. Poke. Prod. Experiment. Learn. Adapt. Grow. Some paths that follow this pattern are easier, safer, or more secure than other paths. Find the one that works best for your company, your technology stack, and your budget and personnel constraints.