# Building confidence through chaos engineering on AWS

Naren Gakka (he/him)

Solutions Architect
AWS

# Agenda

1

Chaos engineering (CE) introduction

2

Continuous resilience (CR)

3

Building a CR/CE program

4

Chaos engineering resources

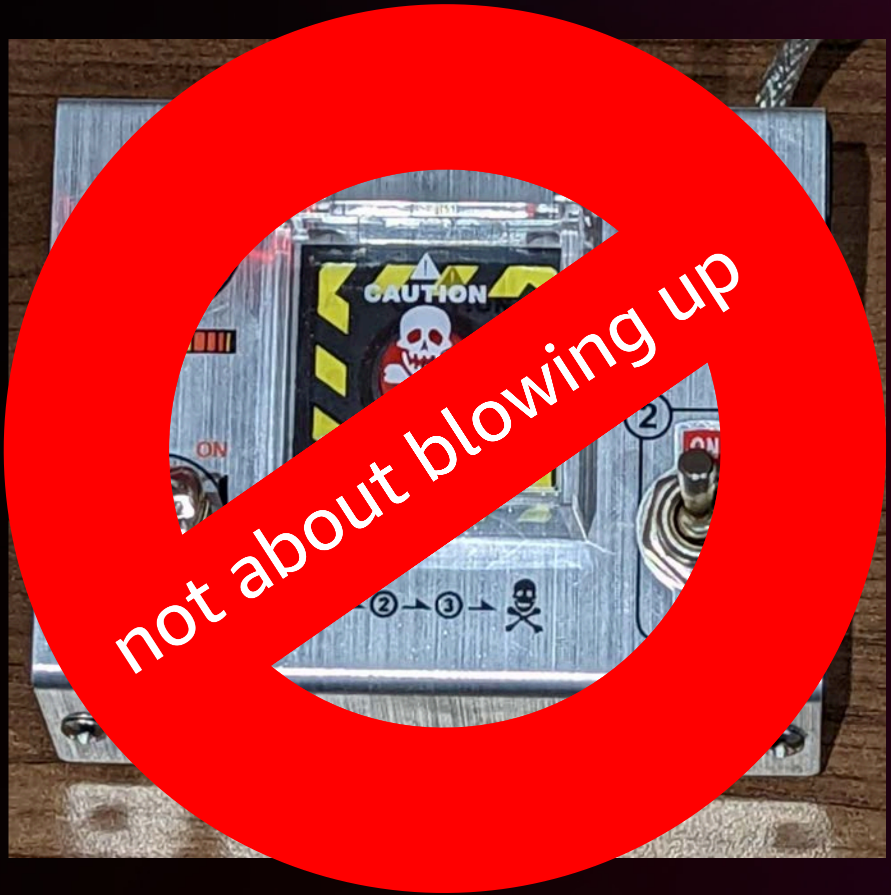
# Introduction to chaos engineering

> “Chaos engineering is about building a culture of resilience in the presence of unexpected system outcomes.”

**Principles of chaos engineering**

principlesofchaos.org

# Chaos engineering a different perspective

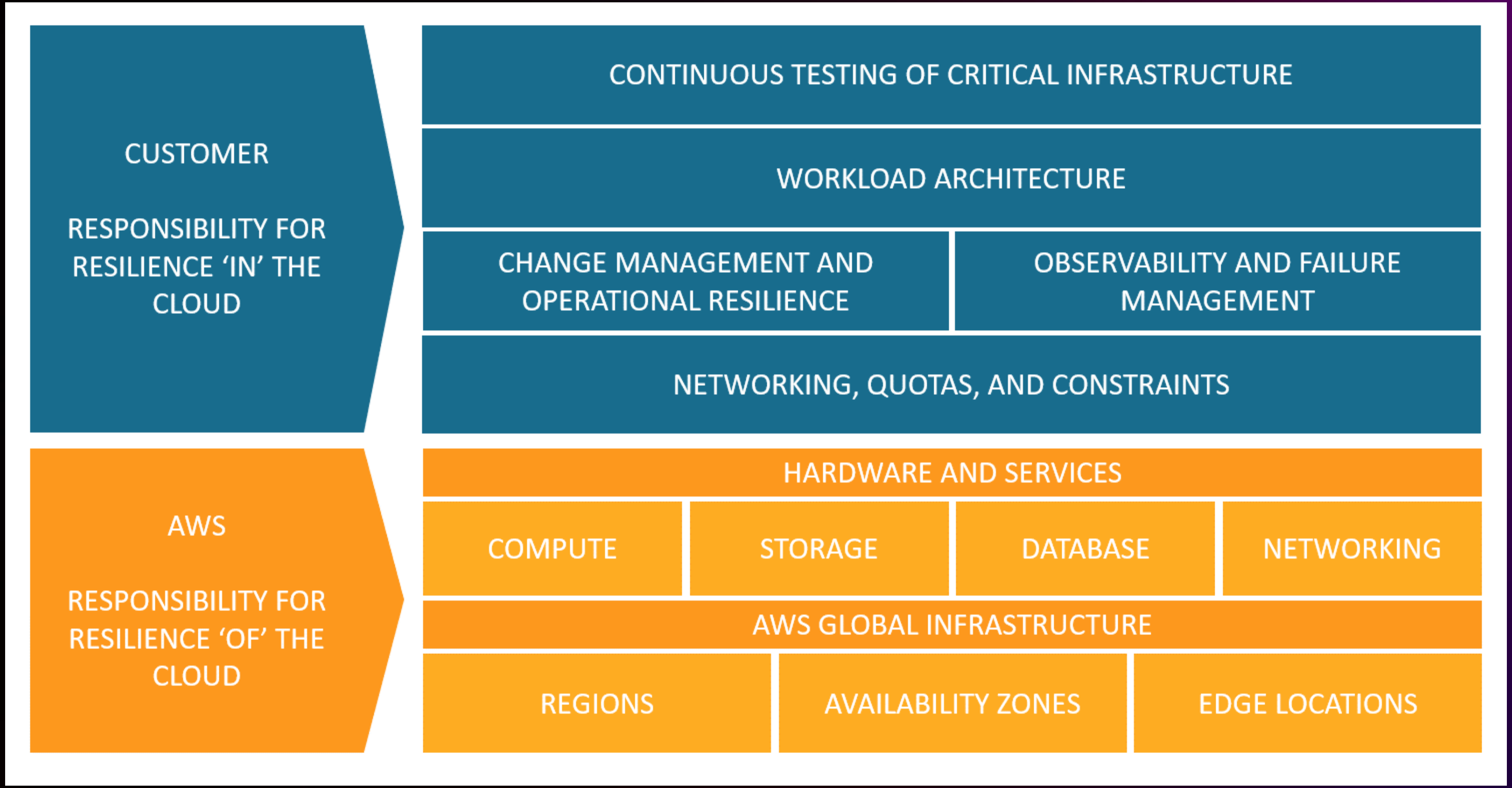


Chaos engineering is not about blowing up production

# Chaos engineering a different perspective

What you have control over

What you have no control over



CUSTOMER

RESPONSIBILITY FOR RESILIENCE 'IN' THE CLOUD

CONTINUOUS TESTING OF CRITICAL INFRASTRUCTURE

WORKLOAD ARCHITECTURE

CHANGE MANAGEMENT AND OPERATIONAL RESILIENCE

OBSERVABILITY AND FAILURE MANAGEMENT

NETWORKING, QUOTAS, AND CONSTRAINTS

AWS

RESPONSIBILITY FOR RESILIENCE 'OF' THE CLOUD

HARDWARE AND SERVICES

COMPUTE

STORAGE

DATABASE

NETWORKING

AWS GLOBAL INFRASTRUCTURE

REGIONS

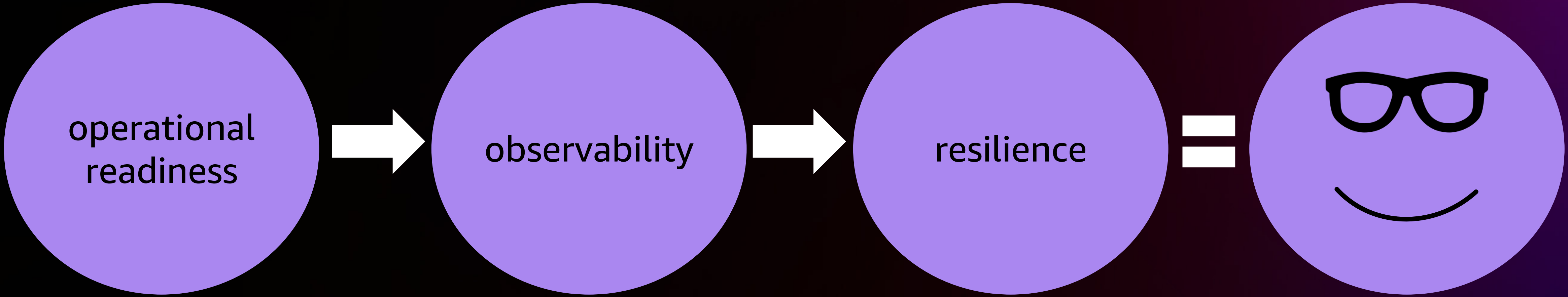
AVAILABILITY ZONES

EDGE LOCATIONS

SHARED RESPONSIBILITY MODEL FOR RESILIENCE

# Chaos engineering a different perspective

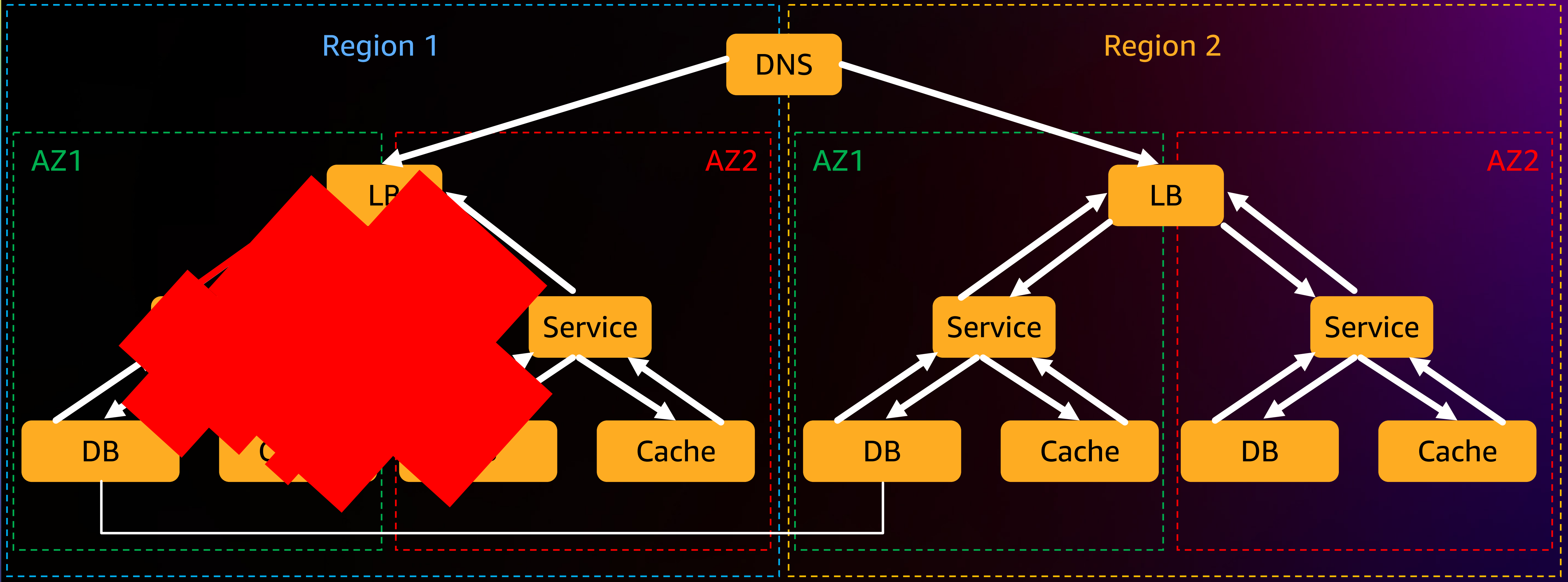
Chaos engineering leads to improved



# Fail-safe chaos experiments

Perform controlled experiments in which the assumption is that the load or faults that you inject will be tolerated by the system and are fail-safe.

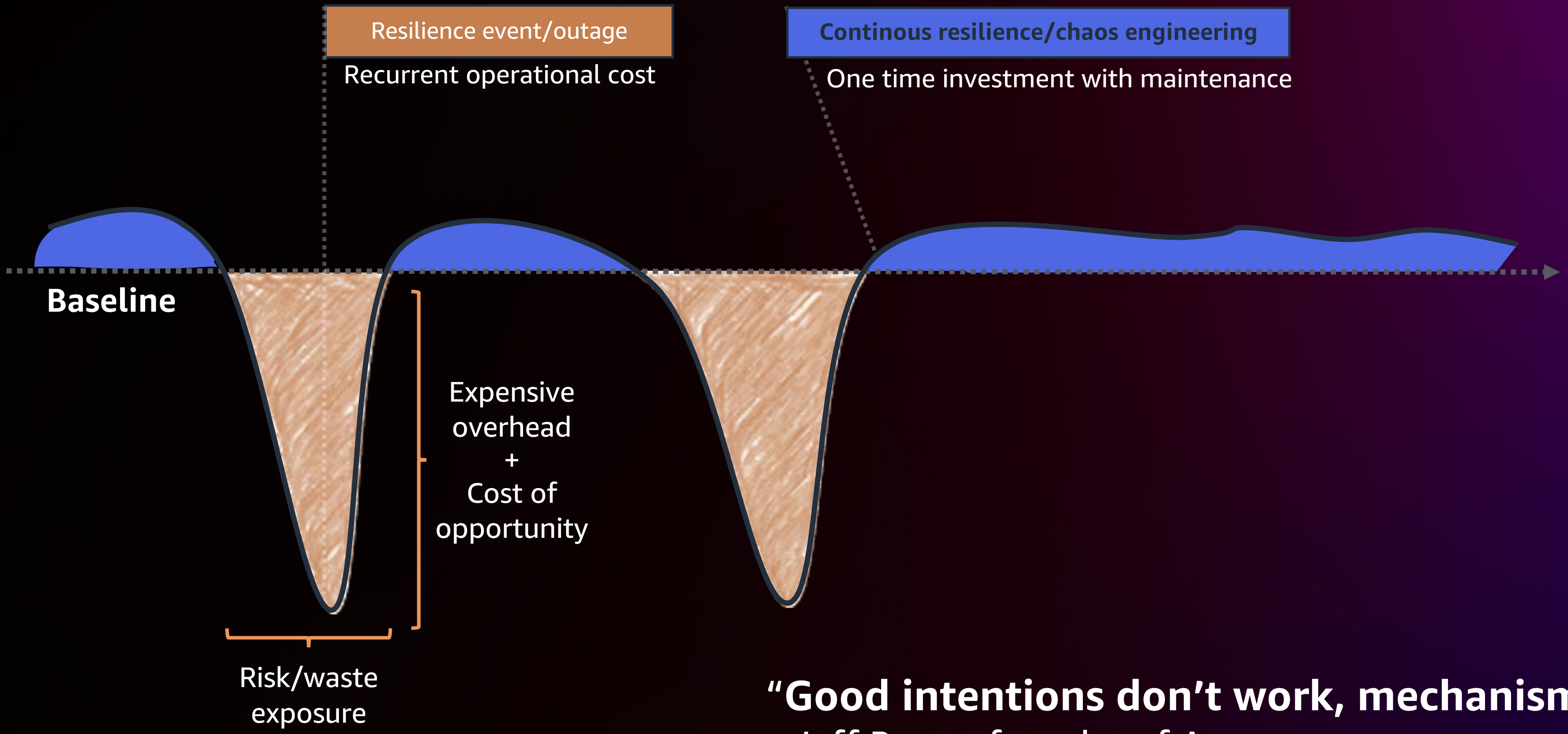
# How much confidence do you have in your system?



# Why chaos engineering?

- Resilience
- Technical debt
- Security posture
- Cost efficiency

Trade offs



Resilience event/outage

Recurrent operational cost

Continous resilience/chaos engineering

One time investment with maintenance

Baseline

Expensive overhead + Cost of opportunity

Risk/waste exposure

**“Good intentions don’t work, mechanisms do.”**
– Jeff Bezos, founder of Amazon

# Chaos engineering stories

- Financial Services
- Health Care
- Insurance
- Media & Entertainment
- Telco
- Retail
- Transport/Airlines
- Hospitality
- Food/Delivery

https://github.com/ldomb/ChaosEngineeringPublicStories

# Chaos engineering adoption 2023

40% Will adopt chaos engineering as part  of their DevOps initiatives in 2023

Reducing unplanned downtime by 20%

Source: The I&O Leader's Guide to Chaos Engineering by Gartner

# Chaos engineering prerequisites

**Prerequisite 1:**

Basic monitoring/observability

**Prerequisite 2:**

Organizational awareness

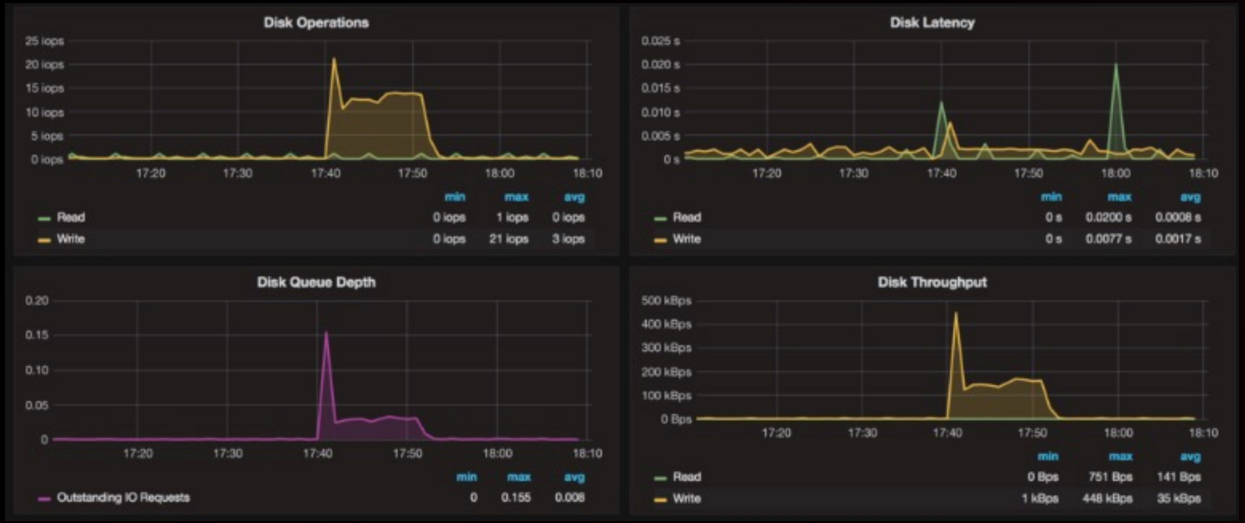
**Prerequisite 3:**

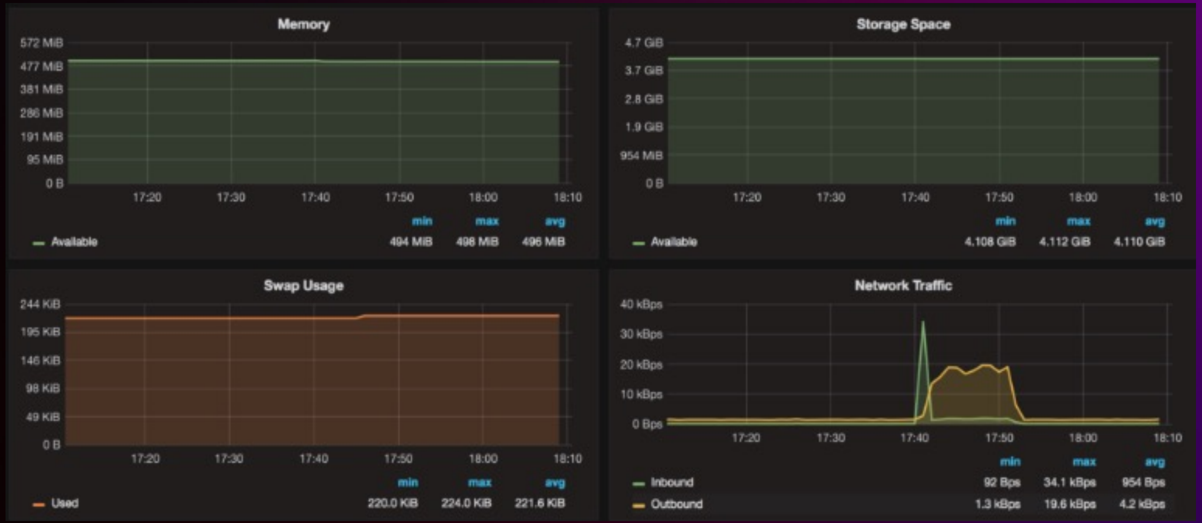
Real world events/faults

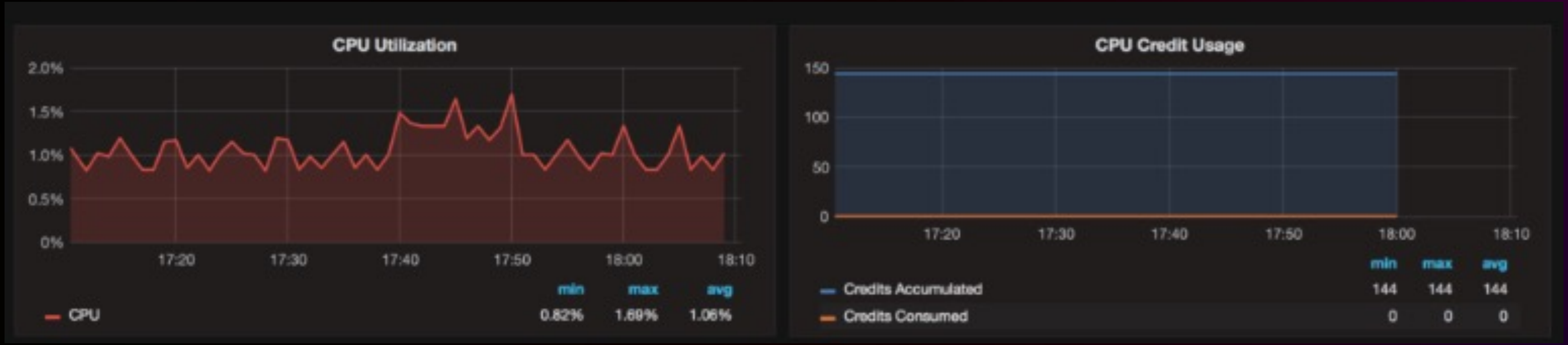
**Prerequisite 4:**

Remediate findings

# Prerequisite 1: Basic monitoring/observability



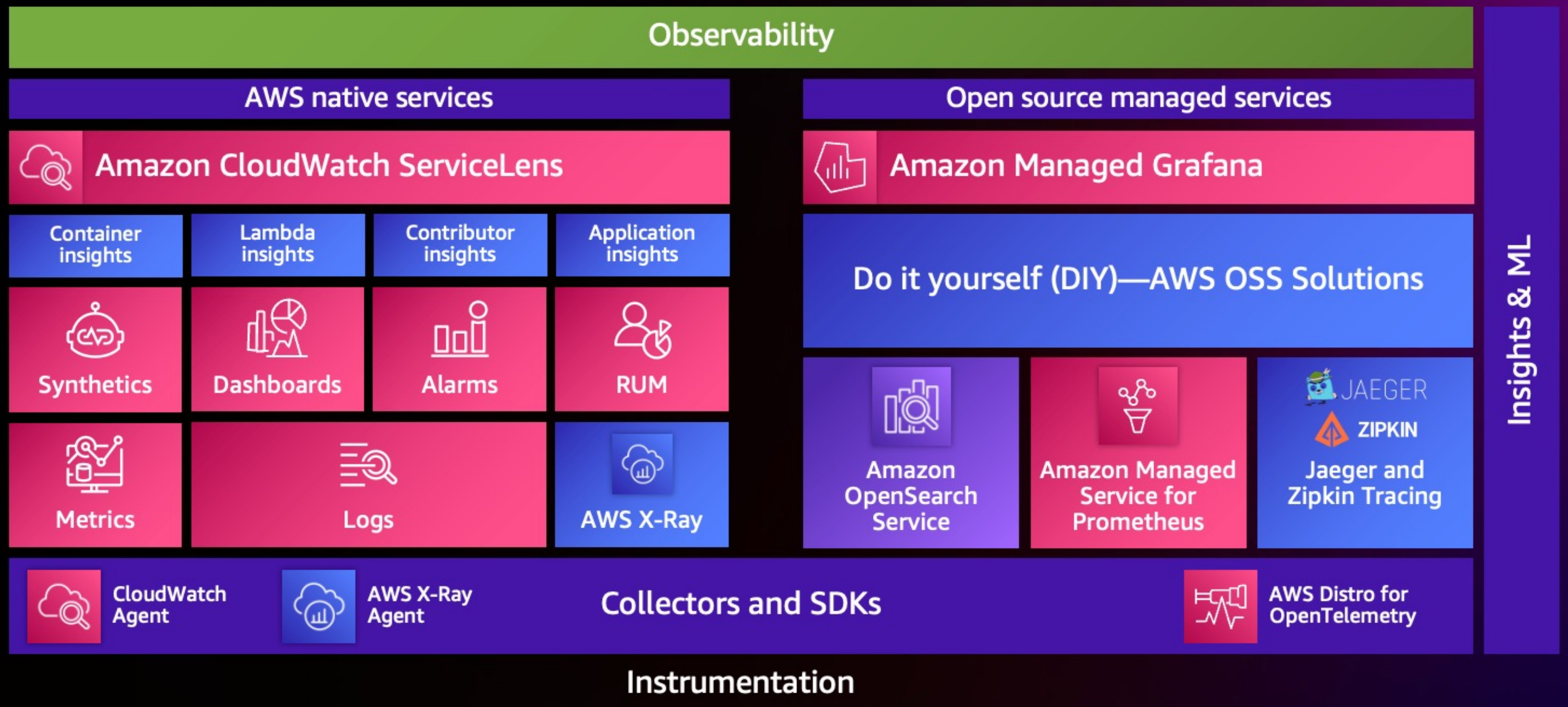




# Prerequisite 1: Basic monitoring/observability

## Observability

Find the needle in the haystack



Observability

AWS native services

Amazon CloudWatch ServiceLens

Container insights | Lambda insights | Contributor insights | Application insights

Synthetics | Dashboards | Alarms | RUM

Metrics | Logs | AWS X-Ray

Open source managed services

Amazon Managed Grafana

Do it yourself (DIY)—AWS OSS Solutions

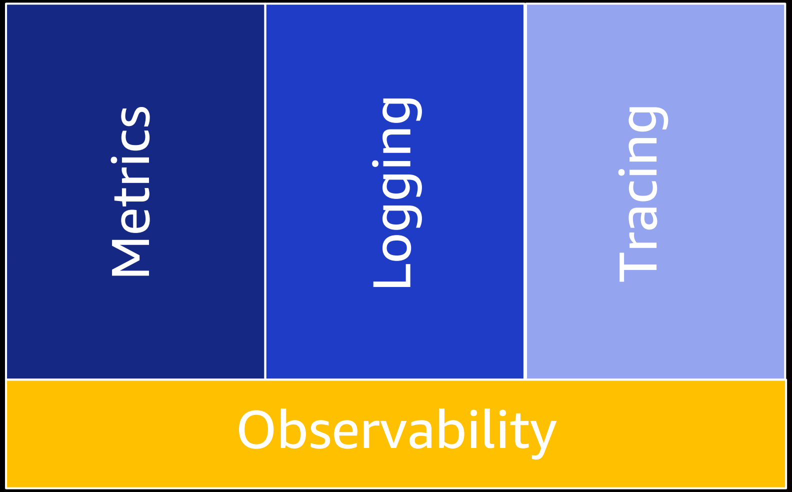
Amazon OpenSearch Service | Amazon Managed Service for Prometheus | Jaeger and Zipkin Tracing

Collectors and SDKs: CloudWatch Agent | AWS X-Ray Agent | AWS Distro for OpenTelemetry

Instrumentation

Insights & ML

# The 3 pillars of observability



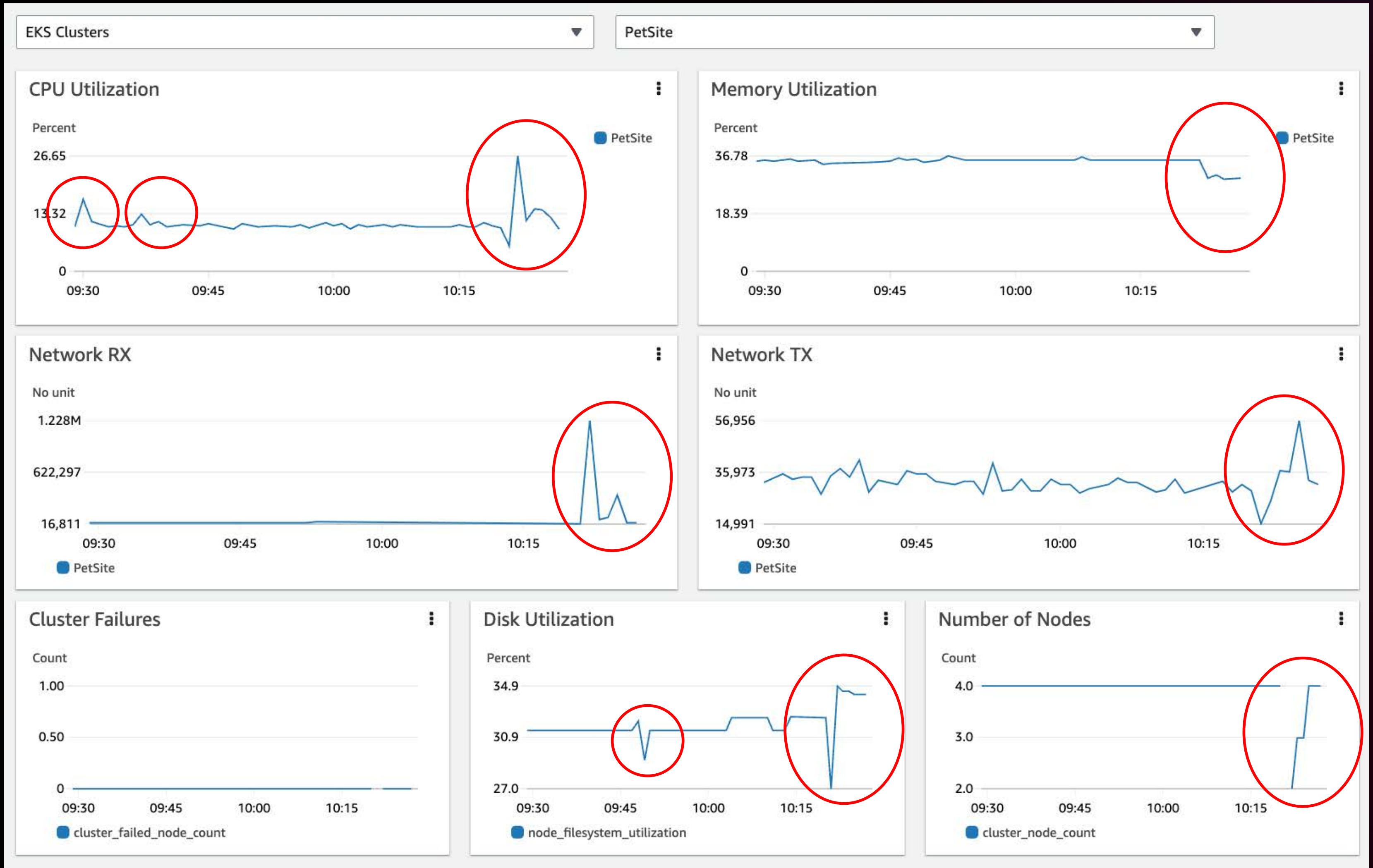
Do you understand the inner workings of your application?

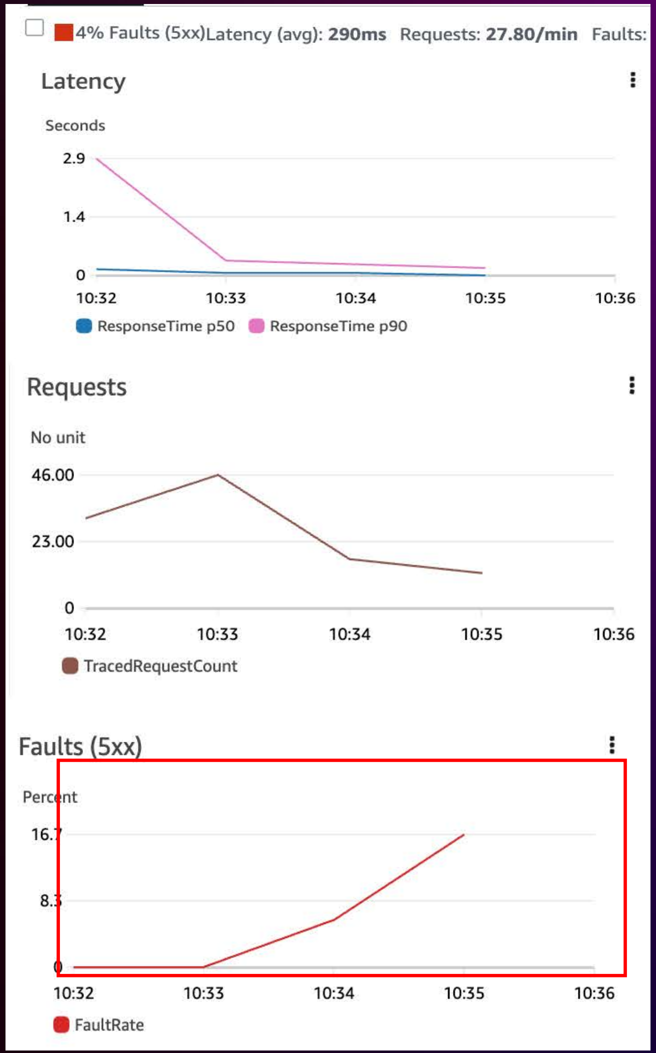
Do you understand any system state your application may have fallen into?

Can you understand the above, just by observing your tools?

Can you understand the state, no matter how unusual the situation is?

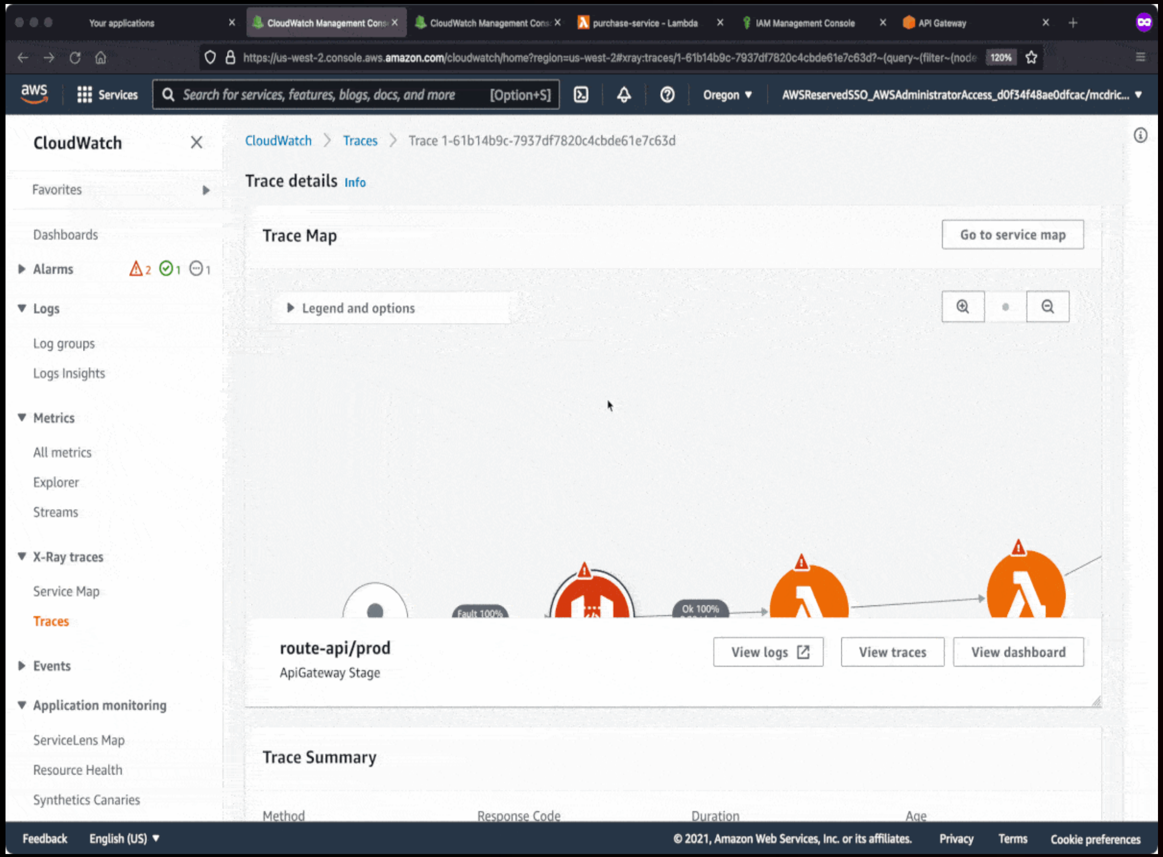
# Prerequisite 1: Basic monitoring/observability



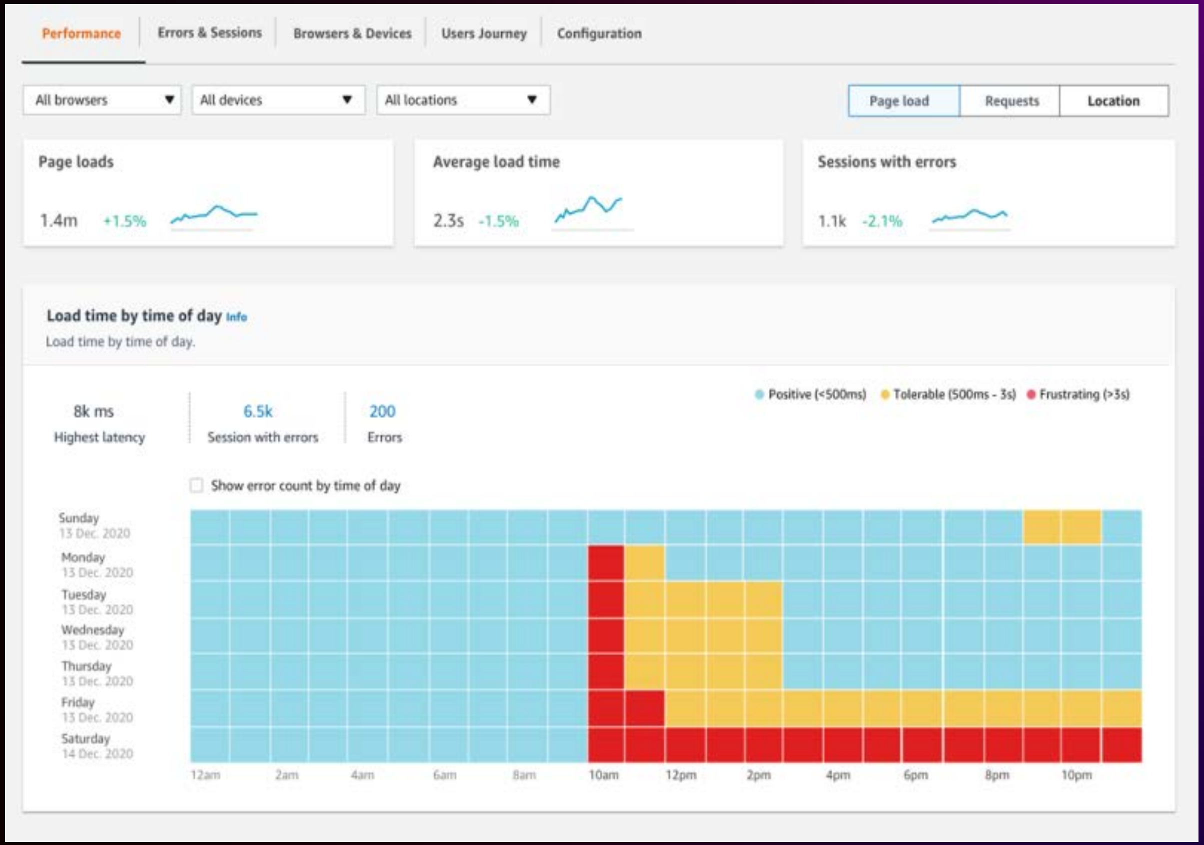


Top level observability

# Prerequisite 1: Basic monitoring/observability



Application monitoring

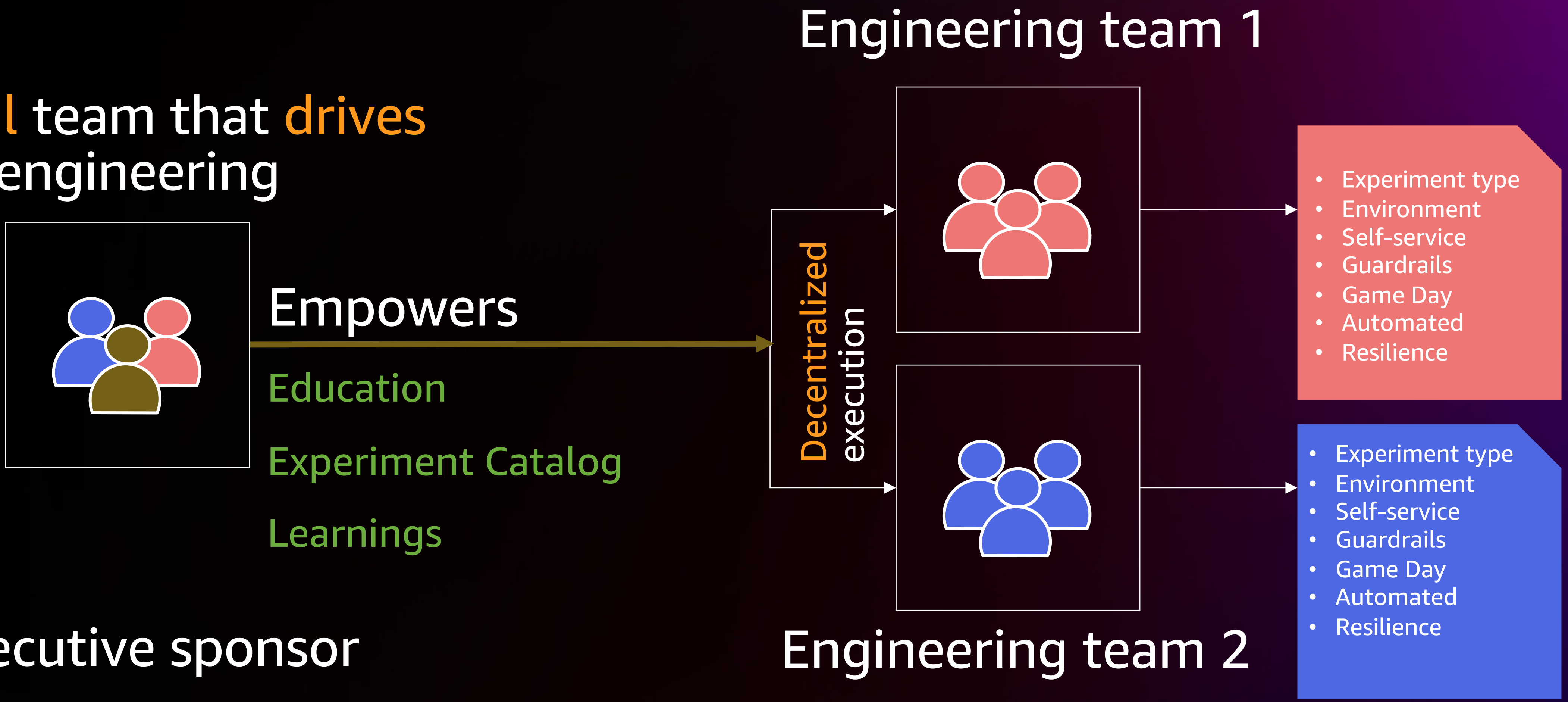


User monitoring

https://aws.amazon.com/blogs/mt/an-observability-journey-with-amazon-cloudwatch-rum-evidently-and-servicelens/

# Prerequisite 2: Organizational awareness

Central team that drives chaos engineering



Empowers

Education

Experiment Catalog

Learnings

Decentralized execution

Engineering team 1

- Experiment type
- Environment
- Self-service
- Guardrails
- Game Day
- Automated
- Resilience

Engineering team 2

- Experiment type
- Environment
- Self-service
- Guardrails
- Game Day
- Automated
- Resilience

Executive sponsor

# Prerequisite 3: Real-world experiments

## Code & configuration

e.g., bad deployment, cred expiration, host shutdown

## Infrastructure

e.g., datacenter failure, hardware failure

## Data and state

e.g., data corruption, overload

## Dependencies

e.g., third-party integrations, AWS services

## Highly unlikely, but technically feasible

e.g., physical loss of an entire Region, the internet is down

# Prerequisite 4: Remediate the findings

- Findings through chaos engineering experiments should be prioritized based on the level of impact they may cause
- Findings that involve the resilience or security of your workload should have priority over new features, as if not addressed timely, can impact your customers
- Find an executive sponsor that can help you address the priority if needed

# Continuous resilience

# Continuous resilience

Code & configuration

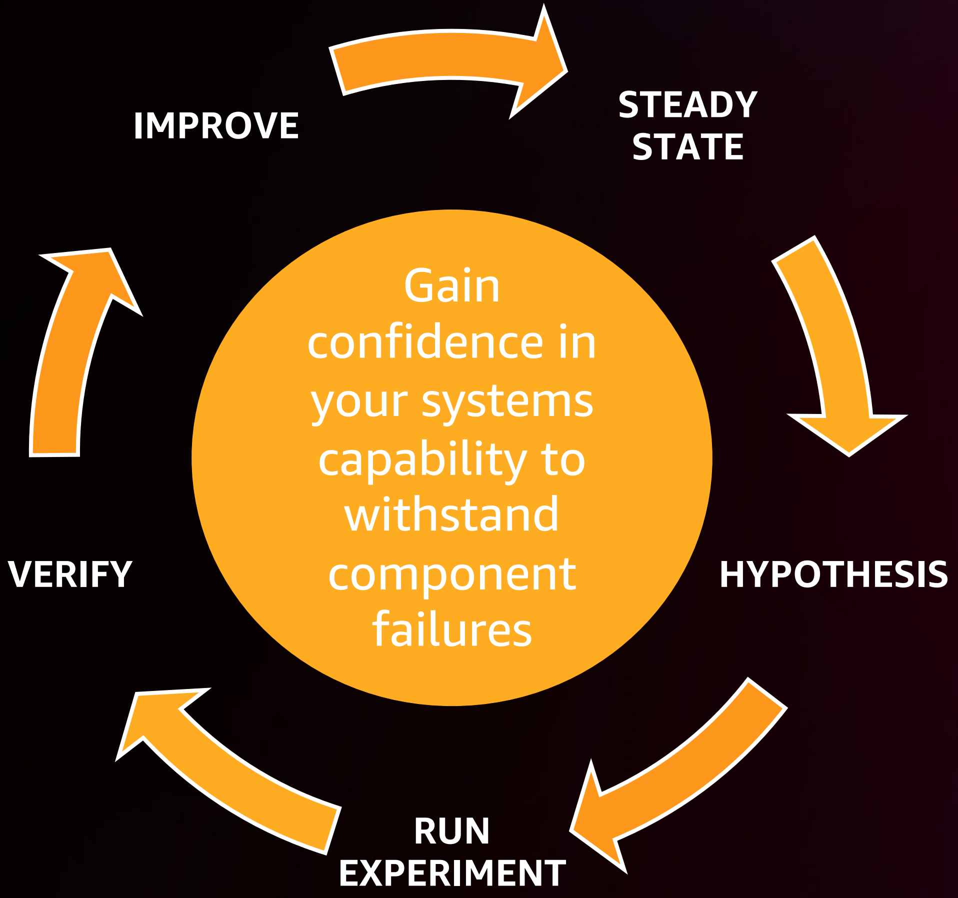
e.g. bad deployment, cred expiration

Infrastructure

e.g. datacenter failure, hardware failure

Data and state

e.g. data corruption


IMPROVE
STEADY STATE
HYPOTHESIS
RUN EXPERIMENT
VERIFY
Gain confidence in your systems capability to withstand component failures

Continuous resilience flywheel by Adrian Hornsby

Dependencies

e.g. third-party integrations, AWS services

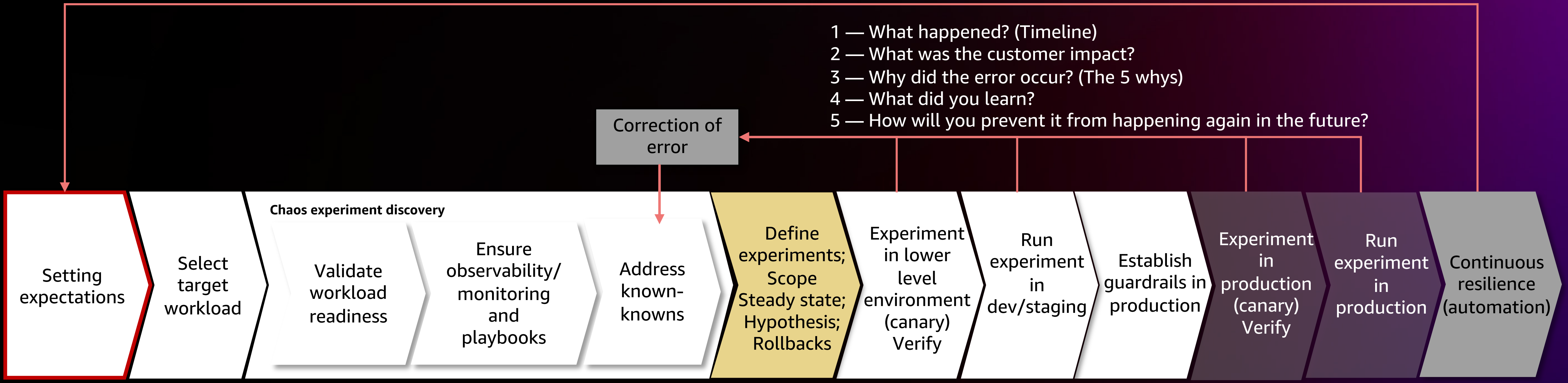
Highly unlikely, but technically feasible

e.g. physical loss of an entire Region, the internet is down

# Building a chaos engineering / continuous resilience program

# Setting expectations

PRIORITIZE FINDINGS, FIX THEM, RE-ITERATE. CONTINUOUSLY RUN EXPERIMENTS AS OFTEN AS WORKLOAD NEEDS; SCALE MECHANISM TO OTHER WORKLOADS.

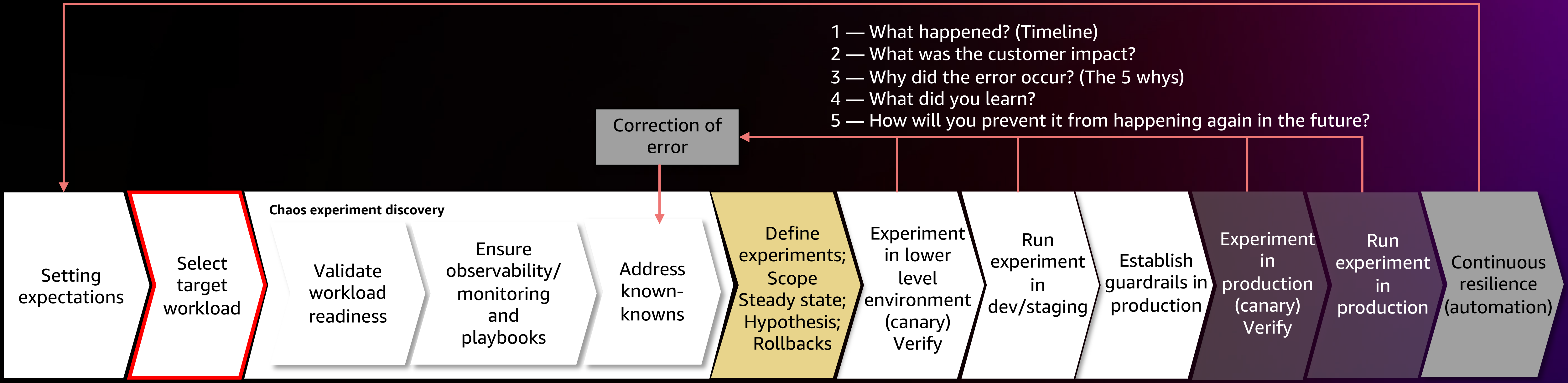

1 — What happened? (Timeline)
2 — What was the customer impact?
3 — Why did the error occur? (The 5 whys)
4 — What did you learn?
5 — How will you prevent it from happening again in the future?
Correction of error
Setting expectations
Select target workload
Chaos experiment discovery
Validate workload readiness
Ensure observability/ monitoring and playbooks
Address known-knowns
Define experiments; Scope Steady state; Hypothesis; Rollbacks
Experiment in lower level environment (canary) Verify
Run experiment in dev/staging
Establish guardrails in production
Experiment in production (canary) Verify
Run experiment in production
Continuous resilience (automation)

# Setting expectations

- Project plan
- Roles and responsibilities
- Contribution
- Outcome

# Selecting the target workload

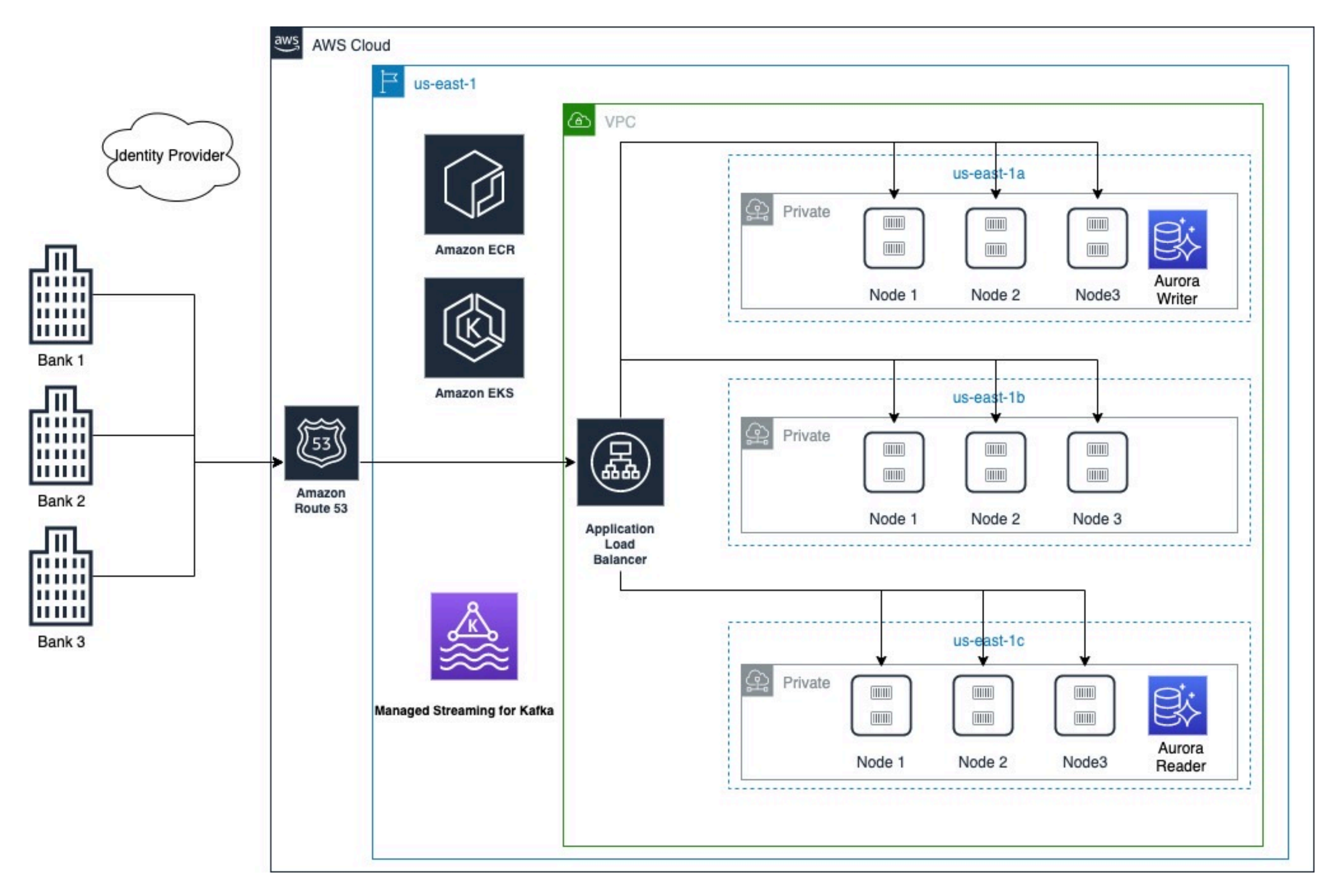
PRIORITIZE FINDINGS, FIX THEM, RE-ITERATE. CONTINUOUSLY RUN EXPERIMENTS AS OFTEN AS WORKLOAD NEEDS; SCALE MECHANISM TO OTHER WORKLOADS.



1 — What happened? (Timeline)
2 — What was the customer impact?
3 — Why did the error occur? (The 5 whys)
4 — What did you learn?
5 — How will you prevent it from happening again in the future?
Correction of error
Chaos experiment discovery
Setting expectations
Select target workload
Validate workload readiness
Ensure observability/ monitoring and playbooks
Address known-knowns
Define experiments; Scope Steady state; Hypothesis; Rollbacks
Experiment in lower level environment (canary) Verify
Run experiment in dev/staging
Establish guardrails in production
Experiment in production (canary) Verify
Run experiment in production
Continuous resilience (automation)

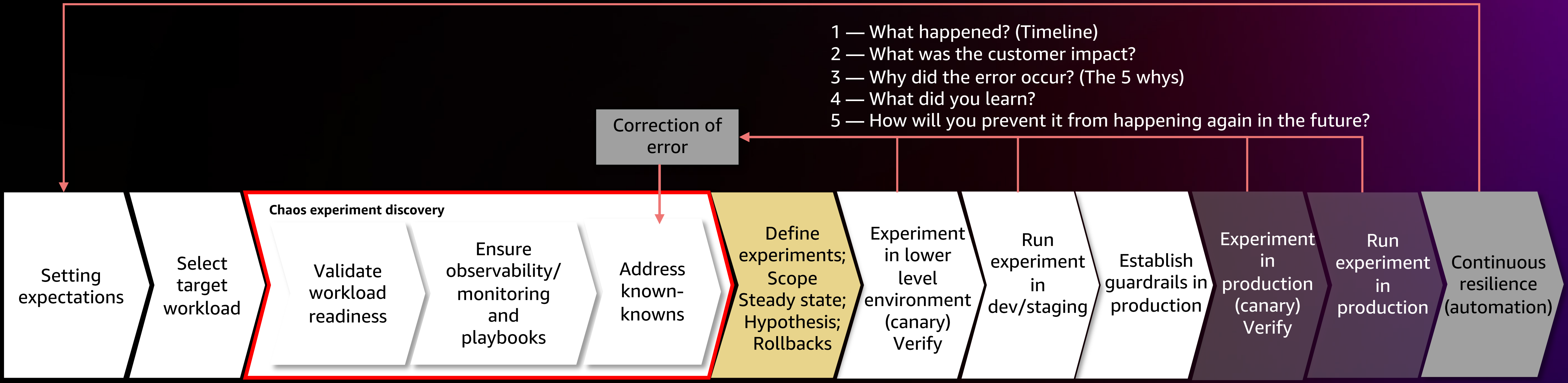
# Target workload selections

For your first experiment, choose a workload that if degraded has only minimal to no impact to your internal or external customers.



# Chaos experiment discovery

PRIORITIZE FINDINGS, FIX THEM, RE-ITERATE. CONTINUOUSLY RUN EXPERIMENTS AS OFTEN AS WORKLOAD NEEDS; SCALE MECHANISM TO OTHER WORKLOADS.



1 — What happened? (Timeline)
2 — What was the customer impact?
3 — Why did the error occur? (The 5 whys)
4 — What did you learn?
5 — How will you prevent it from happening again in the future?

Correction of error

Setting expectations

Select target workload

**Chaos experiment discovery**

Validate workload readiness

Ensure observability/ monitoring and playbooks

Address known-knowns

Define experiments; Scope Steady state; Hypothesis; Rollbacks

Experiment in lower level environment (canary) Verify

Run experiment in dev/staging

Establish guardrails in production

Experiment in production (canary) Verify

Run experiment in production

Continuous resilience (automation)

# Chaos experiment discovery

Operational readiness review

Failover

Observability

Well-Architected Review

Runbooks

Retries

Health checks

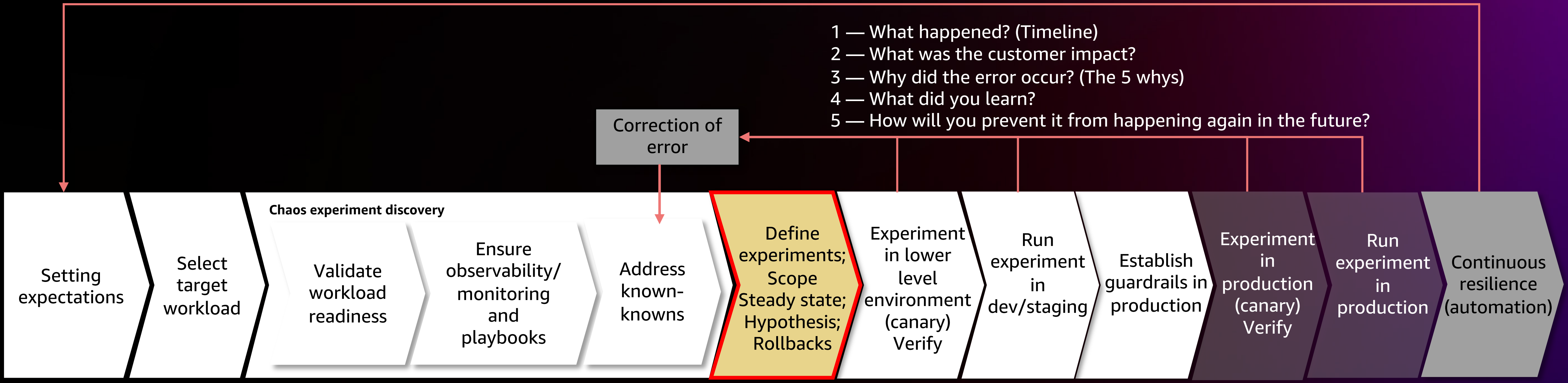
Circuit breakers

Incident response

Fix known issues before moving forward with the experiment!

# Define experiment

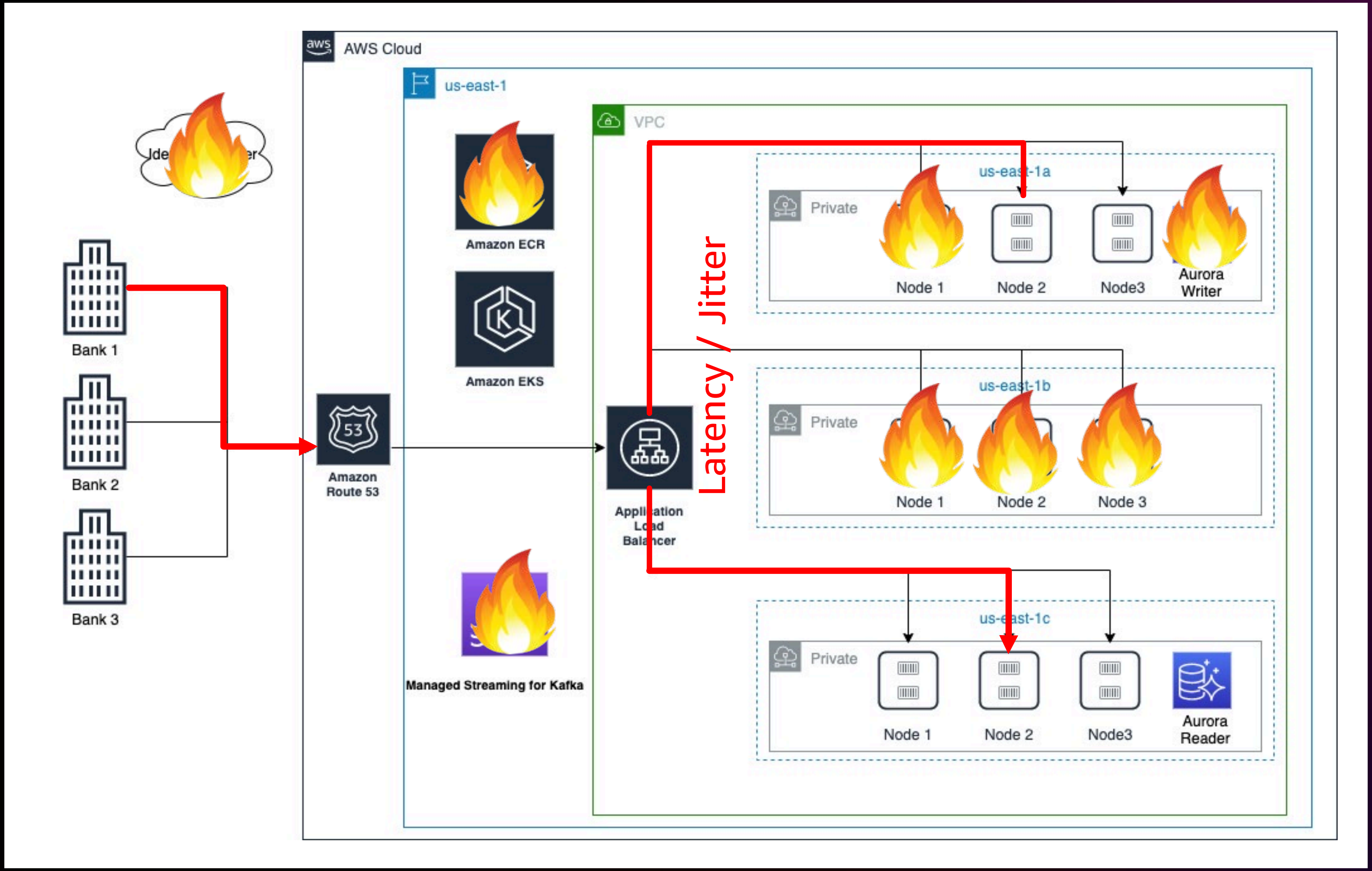
PRIORITIZE FINDINGS, FIX THEM, RE-ITERATE. CONTINUOUSLY RUN EXPERIMENTS AS OFTEN AS WORKLOAD NEEDS; SCALE MECHANISM TO OTHER WORKLOADS.



1 — What happened? (Timeline)
2 — What was the customer impact?
3 — Why did the error occur? (The 5 whys)
4 — What did you learn?
5 — How will you prevent it from happening again in the future?

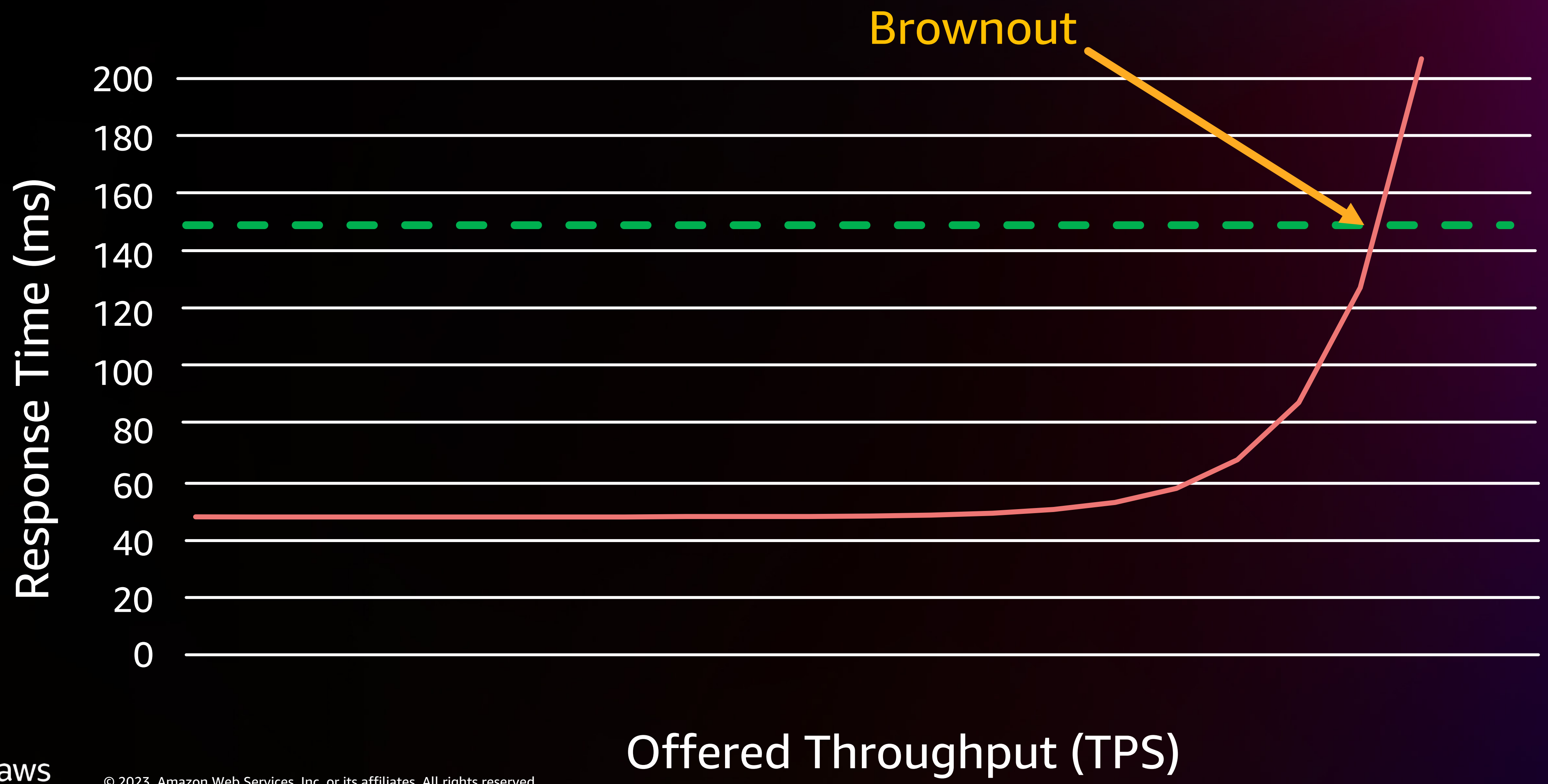
Correction of error
Setting expectations
Select target workload
Chaos experiment discovery
Validate workload readiness
Ensure observability/ monitoring and playbooks
Address known-knowns
Define experiments; Scope Steady state; Hypothesis; Rollbacks
Experiment in lower level environment (canary) Verify
Run experiment in dev/staging
Establish guardrails in production
Experiment in production (canary) Verify
Run experiment in production
Continuous resilience (automation)

# Chaos experiment definition – Failure modes

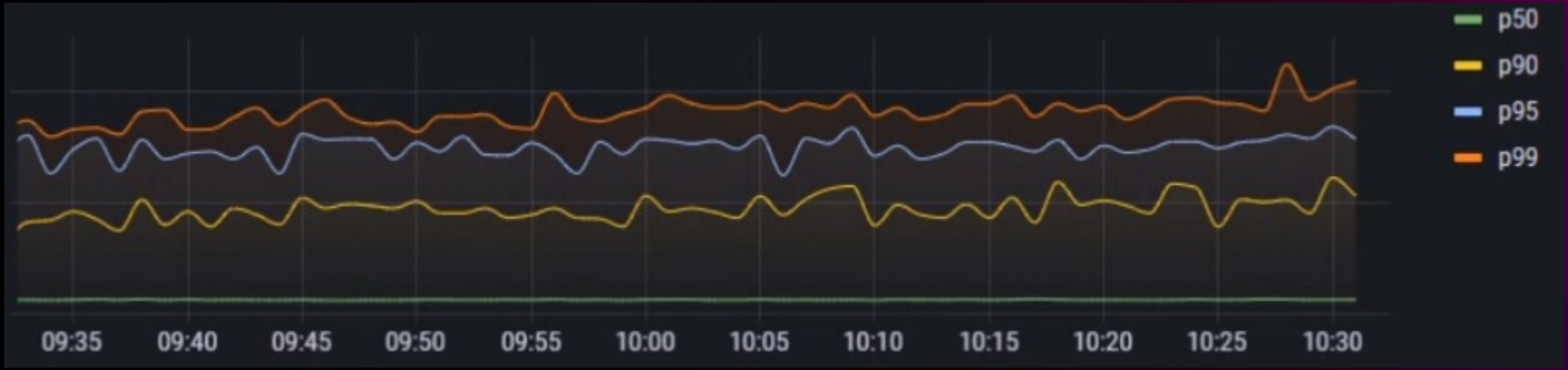


# Chaos experiment definition - Experiment #1

BROWNOUT



# Chaos experiment definition - Steady state



Payments – Transactions per second

Retail – Order per second

Streaming – Stream starts per second

Media/Audio – Playback event started

# Chaos experiment definition - Hypothesis

At a rate of 300 TPS, if 40% of the nodes in the EKS node-group are terminated, the Transaction Create API continues to serve the 99th percentile of requests in under 100 ms (steady state)

The EKS nodes will recover within 5 minutes, and pods will get scheduled and process traffic within 8 minutes after the initiation of the experiment

Alerts will fire within 3 minutes

# Chaos experiment definition – Bring all together

TEMPLATE

## Chaos Experiment

Realtime Payments

Workload Name

Resilience

Contribution

Brownout - Terminate 40% nodes

Action

Staging

Environment

30 minutes

Duration

300 TPS

Load

Amazon EKS Payments Cluster

Targets

Some clients calls will time out

Fault isolation boundary

CW Alarm when node count < 60%

Stop Condition

CFN template to built nodes

Rollback

Chaos-ready

Resource Tag/ID/Filter

OpenSearch/CWL/X-ray

Observability / Logging

At a rate of 300 TPS, If 40% …

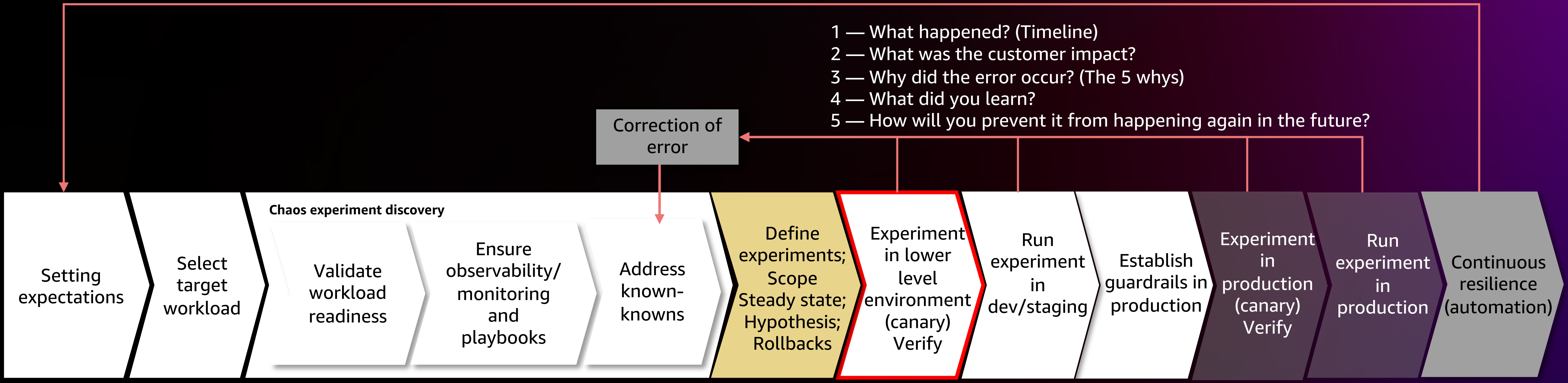
Hypothesis

Findings

COE – Actions to mitigate fault

# Prime the environment for the experiment

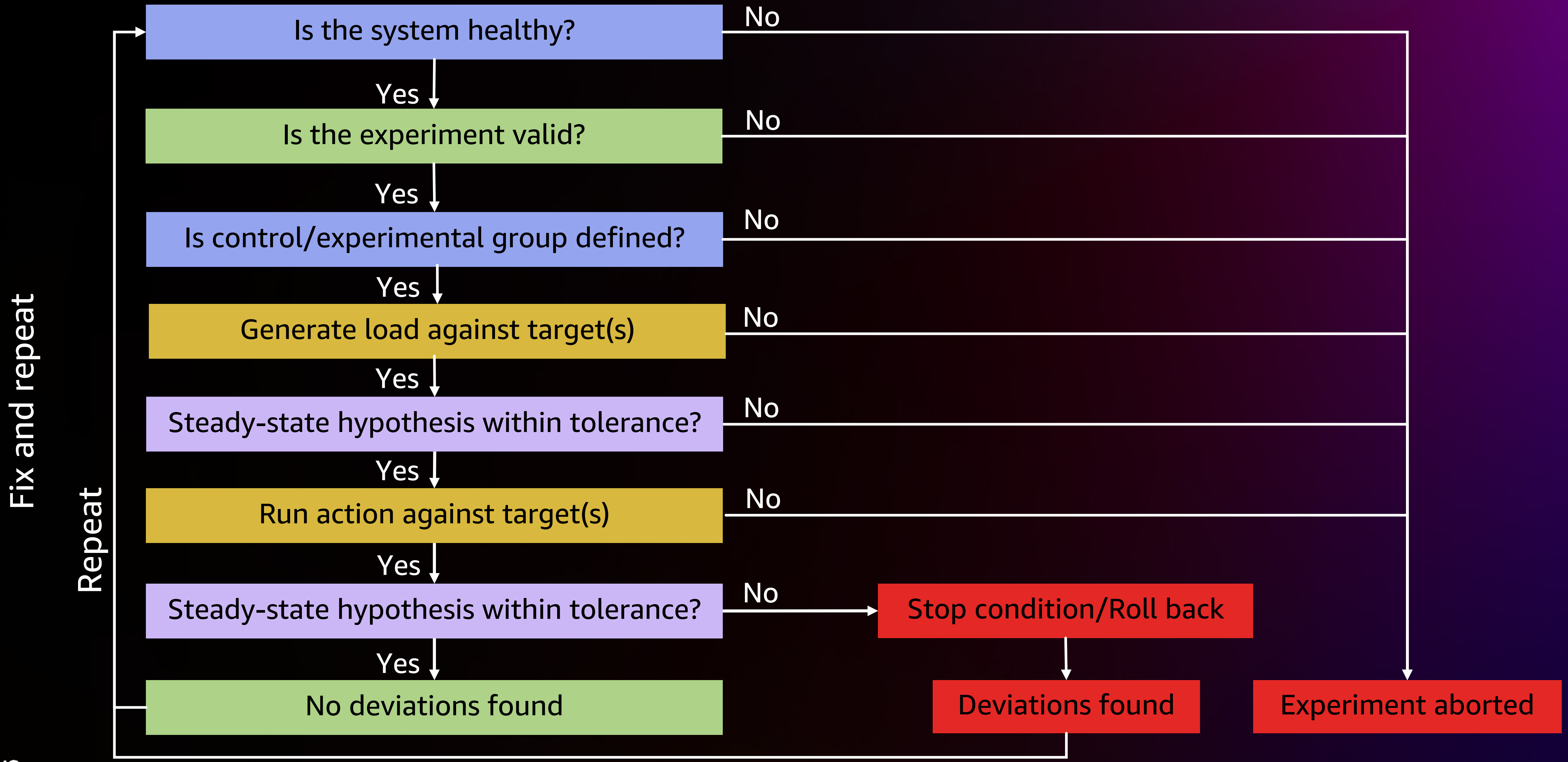
PRIORITIZE FINDINGS, FIX THEM, RE-ITERATE; CONTINUOUSLY RUN EXPERIMENTS AS OFTEN AS WORKLOAD NEEDS; SCALE MECHANISM TO OTHER WORKLOADS



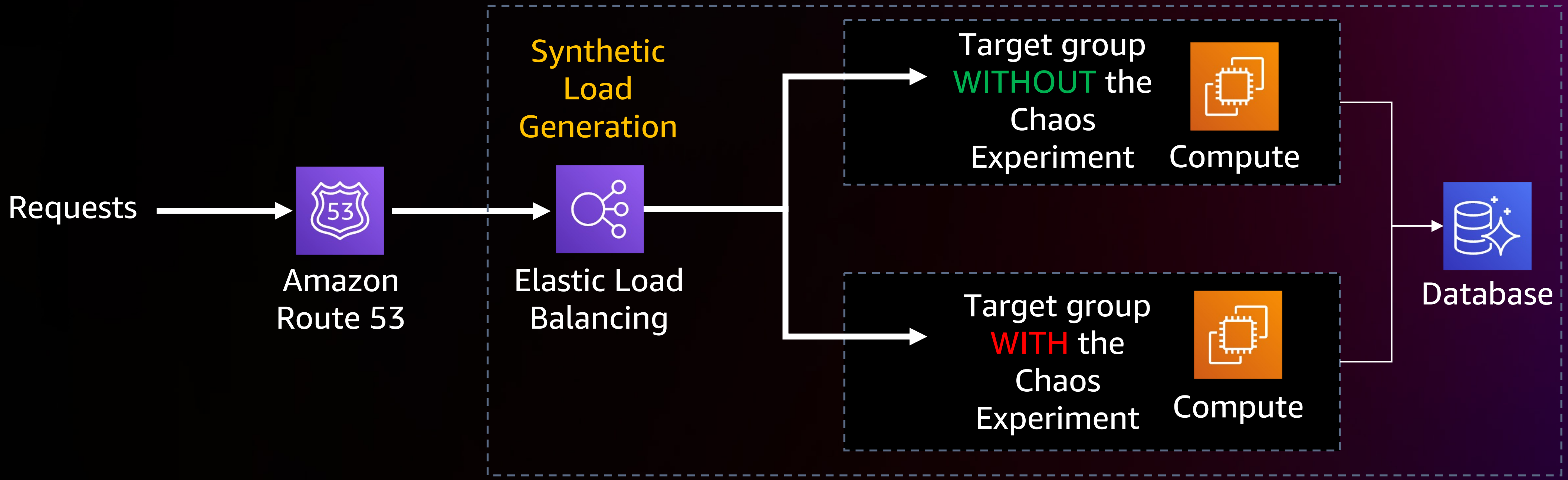
1 — What happened? (Timeline)
2 — What was the customer impact?
3 — Why did the error occur? (The 5 whys)
4 — What did you learn?
5 — How will you prevent it from happening again in the future?

Correction of error
Setting expectations
Select target workload
Chaos experiment discovery
Validate workload readiness
Ensure observability/ monitoring and playbooks
Address known-knowns
Define experiments; Scope Steady state; Hypothesis; Rollbacks
Experiment in lower level environment (canary) Verify
Run experiment in dev/staging
Establish guardrails in production
Experiment in production (canary) Verify
Run experiment in production
Continuous resilience (automation)

# Chaos experiment execution flow


Is the system healthy?
No
Yes
Is the experiment valid?
No
Yes
Is control/experimental group defined?
No
Yes
Generate load against target(s)
No
Yes
Steady-state hypothesis within tolerance?
No
Yes
Run action against target(s)
No
Yes
Steady-state hypothesis within tolerance?
No
Stop condition/Roll back
Yes
No deviations found
Deviations found
Experiment aborted
Fix and repeat
Repeat

# Controlled experiments through canary deployments in lower-level environment

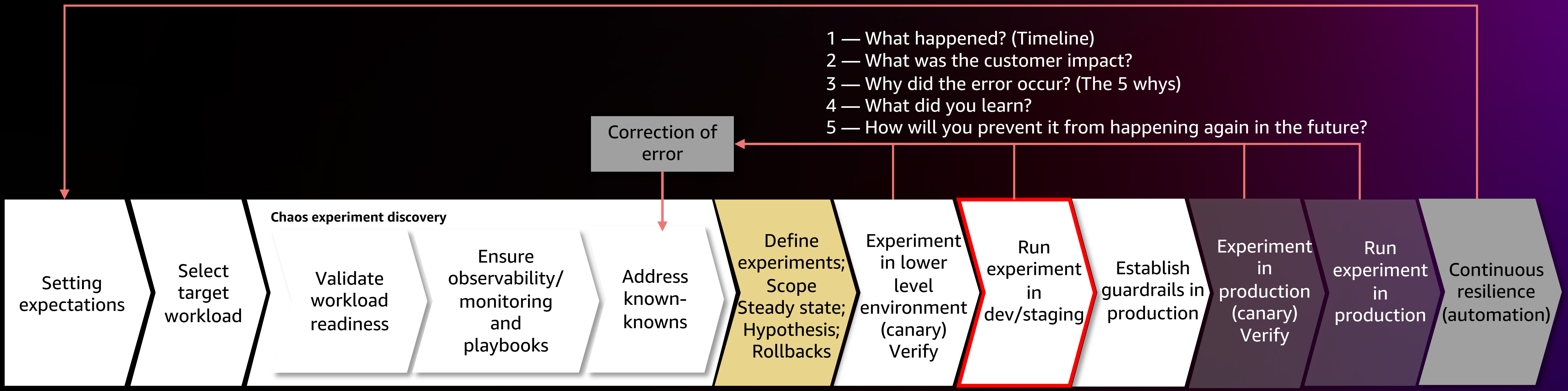

Requests
Amazon Route 53
Synthetic Load Generation
Elastic Load Balancing
Target group WITHOUT the Chaos Experiment
Compute
Target group WITH the Chaos Experiment
Compute
Database

Verify that both groups are healthy before moving forward

Source: https://medium.com/@adhorn/chaos-engineering-q-a-how-to-safely-inject-failure-ced26e11b3db

# Run the experiment in development/staging

PRIORITIZE FINDINGS, FIX THEM, RE-ITERATE; CONTINUOUSLY RUN EXPERIMENTS AS OFTEN AS WORKLOAD NEEDS; SCALE MECHANISM TO OTHER WORKLOADS



1 — What happened? (Timeline)
2 — What was the customer impact?
3 — Why did the error occur? (The 5 whys)
4 — What did you learn?
5 — How will you prevent it from happening again in the future?

Correction of error

Setting expectations

Select target workload

Chaos experiment discovery

Validate workload readiness

Ensure observability/ monitoring and playbooks

Address known-knowns

Define experiments; Scope Steady state; Hypothesis; Rollbacks

Experiment in lower level environment (canary) Verify

Run experiment in dev/staging

Establish guardrails in production

Experiment in production (canary) Verify

Run experiment in production

Continuous resilience (automation)

# AWS Fault Injection Actions

- Stop, reboot, and terminate instance(s) (Amazon EC2)
- API throttling/internal error/unavailable error (Amazon EC2)
- Increased memory or CPU load (Amazon EC2)
- Kill process (Amazon EC2)
- Latency injection (Amazon EC2)
- Drain container instances (Amazon ECS)
- Terminate task (Amazon ECS)
- Increase memory or CPU consumption per task (Amazon ECS)
- Terminate node group instances (Amazon EKS)
- Litmus Chaos/Chaos Mesh integration (Amazon EKS)
- Network Connectivity Disruption (Amazon EC2)
- Database stop, reboot, and failover (Amazon RDS)

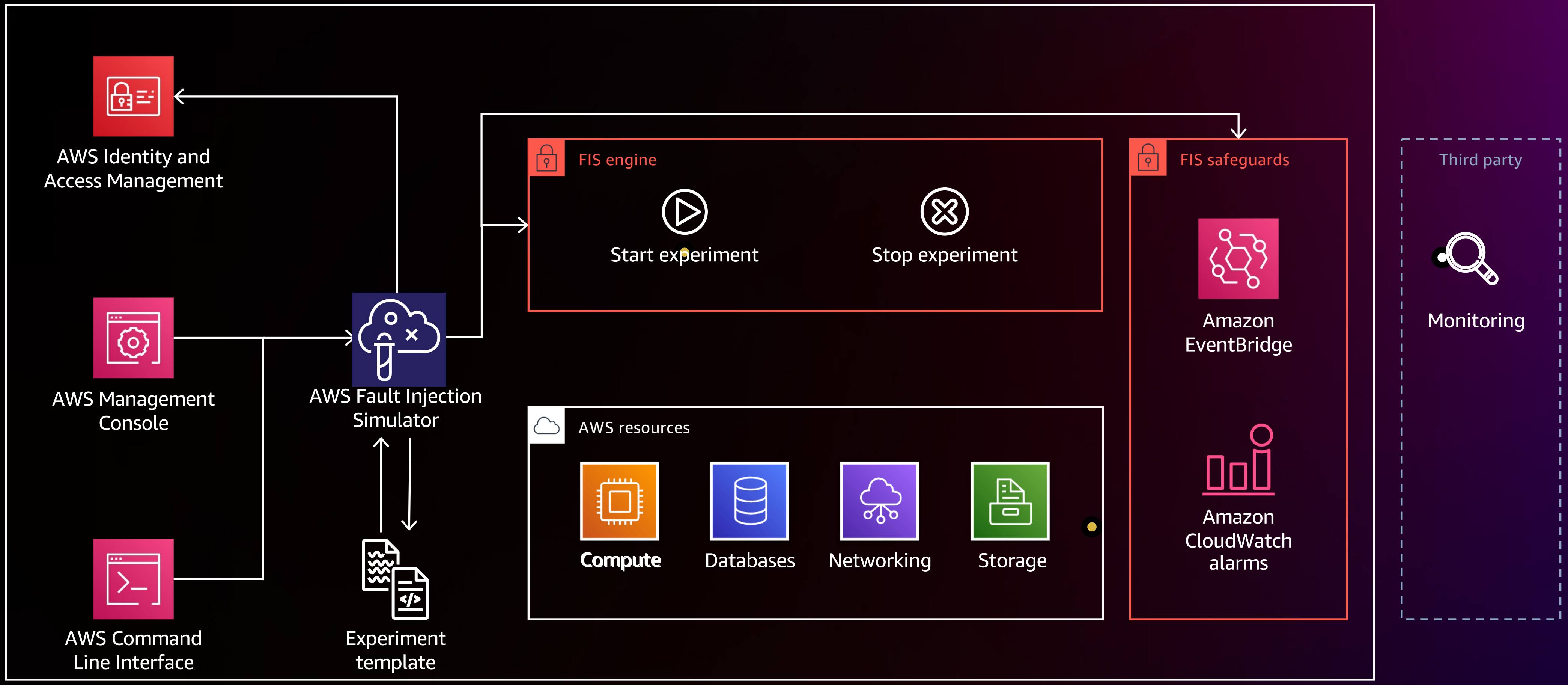
# AWS Fault Injection Actions

- Systems manager send command (SSM Agent)
- Increased memory or CPU load (Amazon EC2)
- Kill process (Amazon EC2)
- Latency injection (Amazon EC2)
- Increase memory or CPU consumption per task (Amazon EC2)
- Network port blackhole (Amazon EC2)
- Network latency
- Network latency at target source
- Network packet loss
- Network packet loss at target source

# Execute the controlled experiment

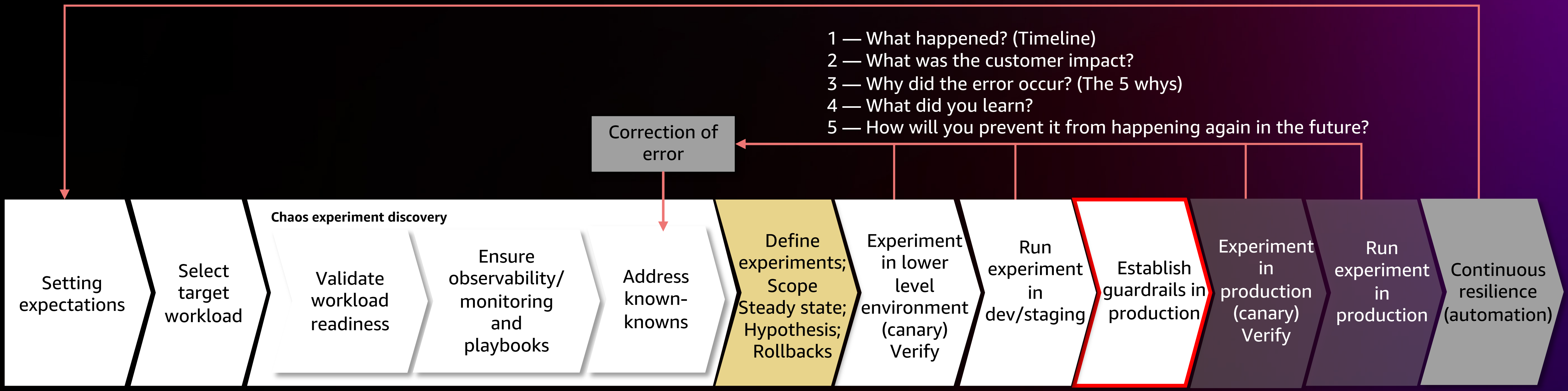
EXECUTE THE CONTROLLED EXPERIMENT

## AWS Fault Injection Simulator



# Establish guardrails in production

PRIORITIZE FINDINGS, FIX THEM, RE-ITERATE; CONTINUOUSLY RUN EXPERIMENTS AS OFTEN AS WORKLOAD NEEDS; SCALE MECHANISM TO OTHER WORKLOADS

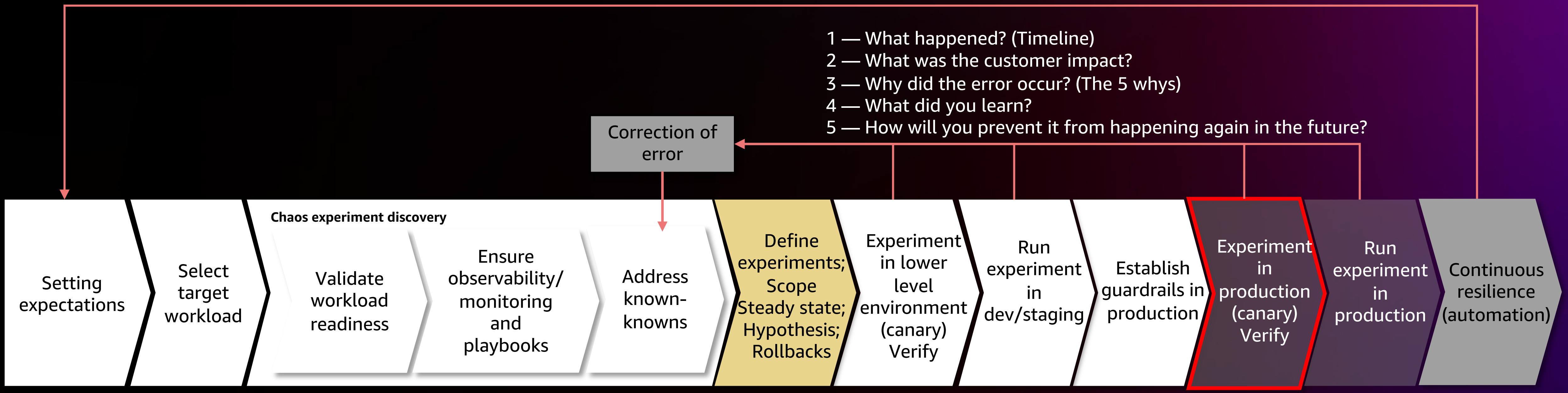

1 — What happened? (Timeline)
2 — What was the customer impact?
3 — Why did the error occur? (The 5 whys)
4 — What did you learn?
5 — How will you prevent it from happening again in the future?
Correction of error
Chaos experiment discovery
Setting expectations
Select target workload
Validate workload readiness
Ensure observability/ monitoring and playbooks
Address known-knowns
Define experiments; Scope Steady state; Hypothesis; Rollbacks
Experiment in lower level environment (canary) Verify
Run experiment in dev/staging
Establish guardrails in production
Experiment in production (canary) Verify
Run experiment in production
Continuous resilience (automation)

# Establish guardrails in production

- Run experiments off peak hours
- Validate that your IAM Permissions are sufficient for your experiment in production
- Validate if your fault isolation boundary is the same as in the lower-level environment or changes in the production environment
- Decide if synthetic traffic will be generated, or if you are planning to run the experiment against a subset of your customers
- Validate that observability is in place
- Validate that your runbooks/playbook are up to date in production

# Prime the prod environment for the experiment

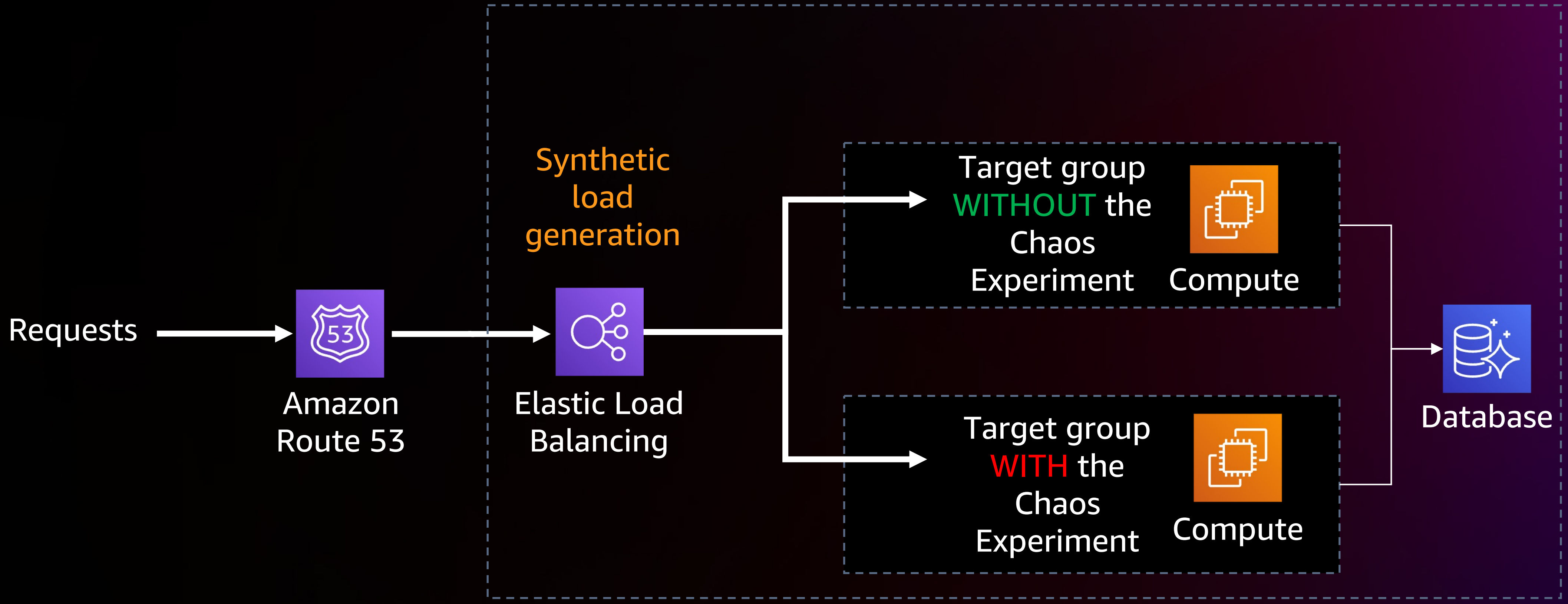
PRIORITIZE FINDINGS, FIX THEM, RE-ITERATE; CONTINUOUSLY RUN EXPERIMENTS AS OFTEN AS WORKLOAD NEEDS; SCALE MECHANISM TO OTHER WORKLOADS



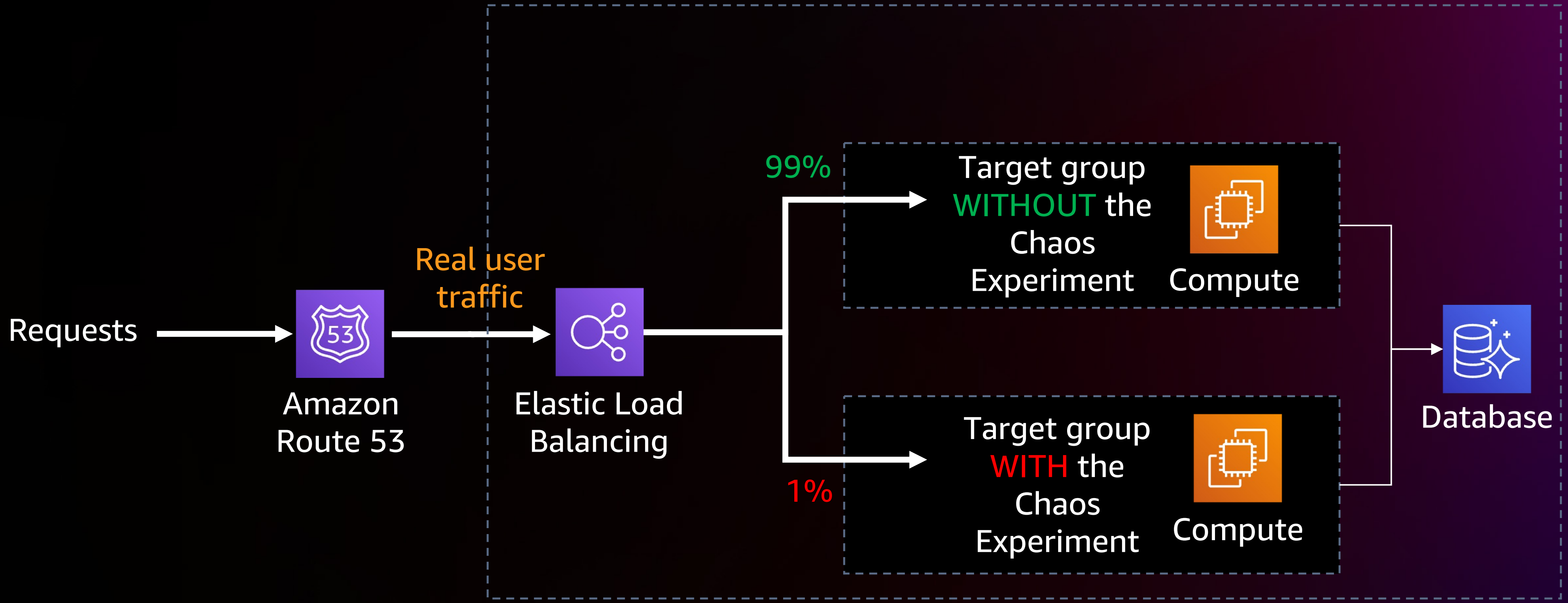
1 — What happened? (Timeline)
2 — What was the customer impact?
3 — Why did the error occur? (The 5 whys)
4 — What did you learn?
5 — How will you prevent it from happening again in the future?

Correction of error
Setting expectations
Select target workload
Chaos experiment discovery
Validate workload readiness
Ensure observability/ monitoring and playbooks
Address known-knowns
Define experiments; Scope Steady state; Hypothesis; Rollbacks
Experiment in lower level environment (canary) Verify
Run experiment in dev/staging
Establish guardrails in production
Experiment in production (canary) Verify
Run experiment in production
Continuous resilience (automation)

# Controlled experiments through canary deployments

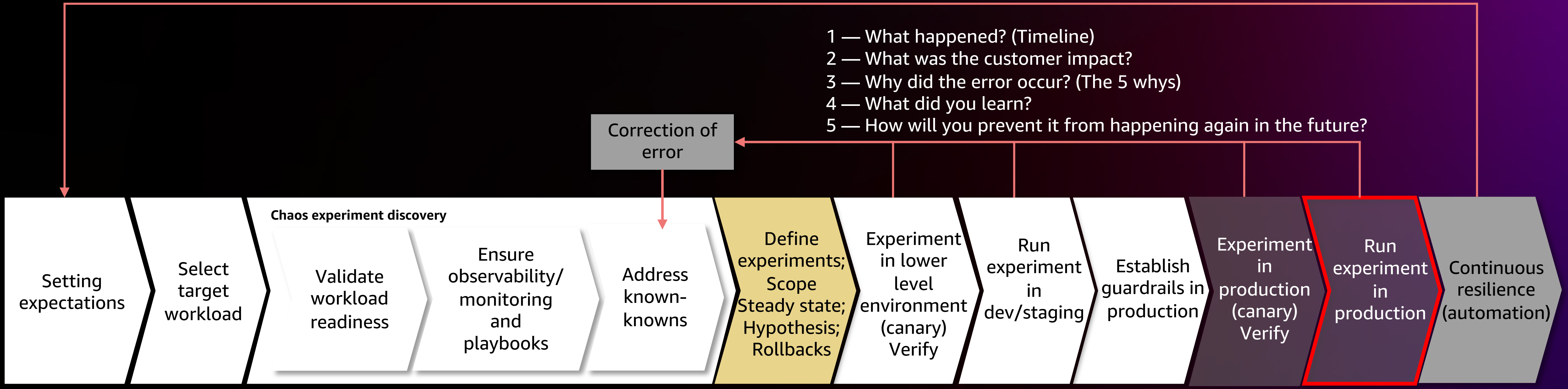

Synthetic load generation
Requests
Amazon Route 53
Elastic Load Balancing
Target group WITHOUT the Chaos Experiment
Compute
Target group WITH the Chaos Experiment
Compute
Database

# Controlled experiments through canary deployments



# Run the experiment in production

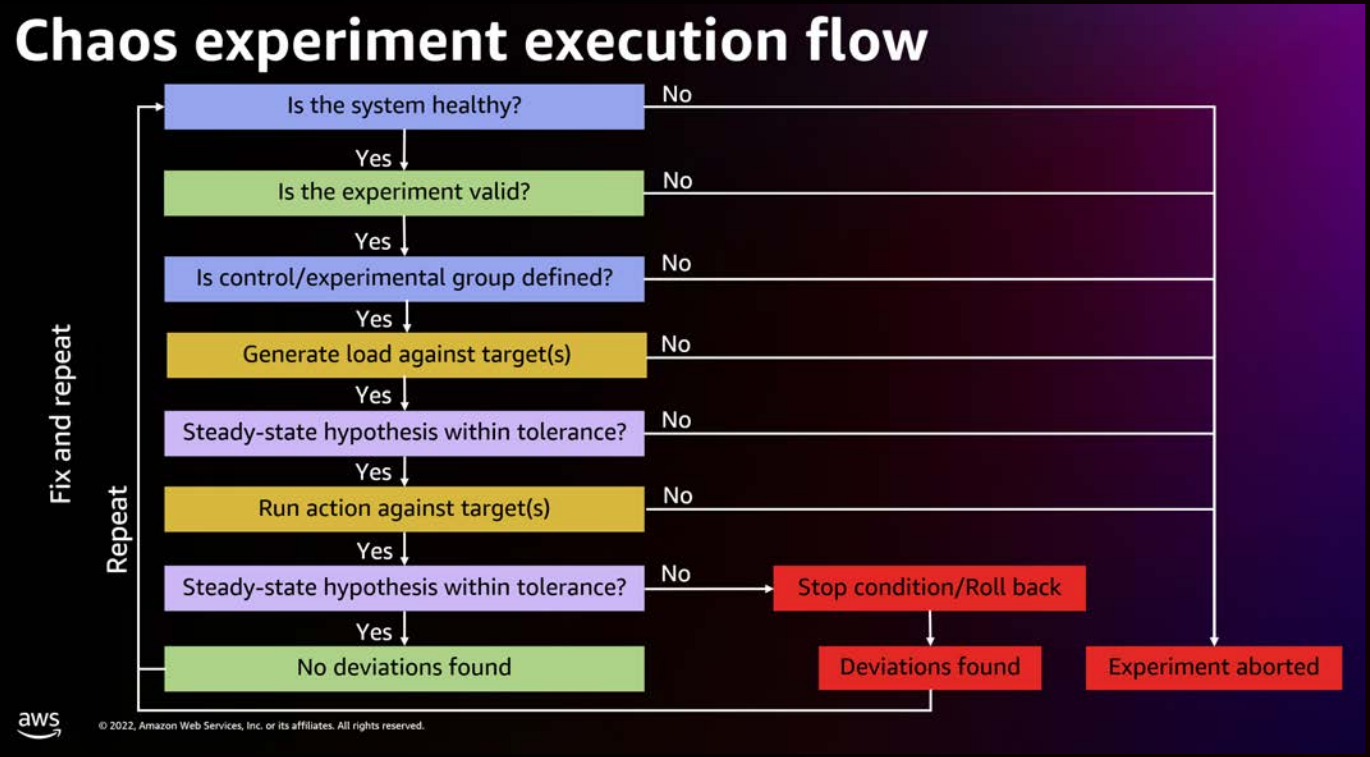
PRIORITIZE FINDINGS, FIX THEM, RE-ITERATE; CONTINUOUSLY RUN EXPERIMENTS AS OFTEN AS WORKLOAD NEEDS; SCALE MECHANISM TO OTHER WORKLOADS



1 — What happened? (Timeline)
2 — What was the customer impact?
3 — Why did the error occur? (The 5 whys)
4 — What did you learn?
5 — How will you prevent it from happening again in the future?

Correction of error
Setting expectations
Select target workload
Chaos experiment discovery
Validate workload readiness
Ensure observability/ monitoring and playbooks
Address known-knowns
Define experiments; Scope Steady state; Hypothesis; Rollbacks
Experiment in lower level environment (canary) Verify
Run experiment in dev/staging
Establish guardrails in production
Experiment in production (canary) Verify
Run experiment in production
Continuous resilience (automation)

# Execute the controlled experiment in production



## Chaos experiment execution flow

Is the system healthy?

Is the experiment valid?

Is control/experimental group defined?

Generate load against target(s)

Steady-state hypothesis within tolerance?

Run action against target(s)

Steady-state hypothesis within tolerance?

No deviations found

Stop condition/Roll back

Deviations found

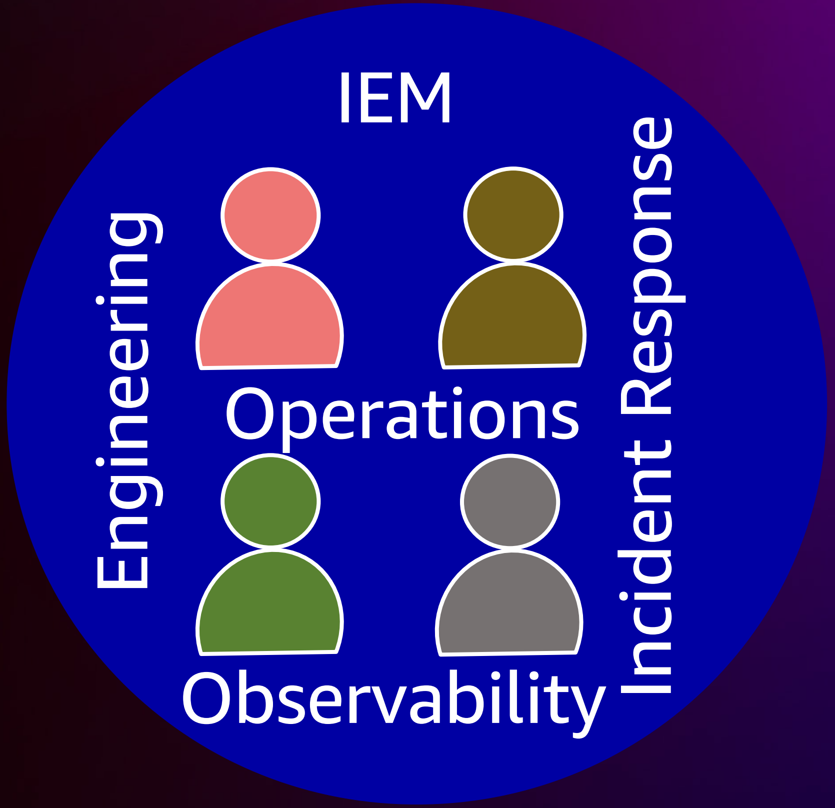
Experiment aborted

Yes

No

Fix and repeat

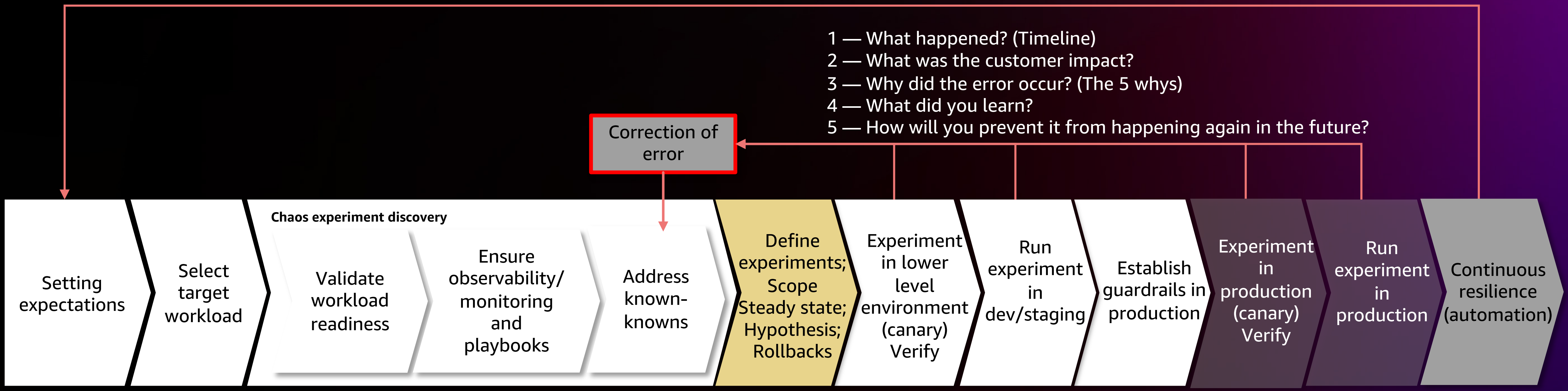
Repeat



Workload Experts

# Run the experiment in production

PRIORITIZE FINDINGS, FIX THEM, RE-ITERATE; CONTINUOUSLY RUN EXPERIMENTS AS OFTEN AS WORKLOAD NEEDS; SCALE MECHANISM TO OTHER WORKLOADS



1 — What happened? (Timeline)
2 — What was the customer impact?
3 — Why did the error occur? (The 5 whys)
4 — What did you learn?
5 — How will you prevent it from happening again in the future?
Correction of error
Setting expectations
Select target workload
Chaos experiment discovery
Validate workload readiness
Ensure observability/ monitoring and playbooks
Address known-knowns
Define experiments; Scope Steady state; Hypothesis; Rollbacks
Experiment in lower level environment (canary) Verify
Run experiment in dev/staging
Establish guardrails in production
Experiment in production (canary) Verify
Run experiment in production
Continuous resilience (automation)

# Post mortem - Correction of error

How did we communicate

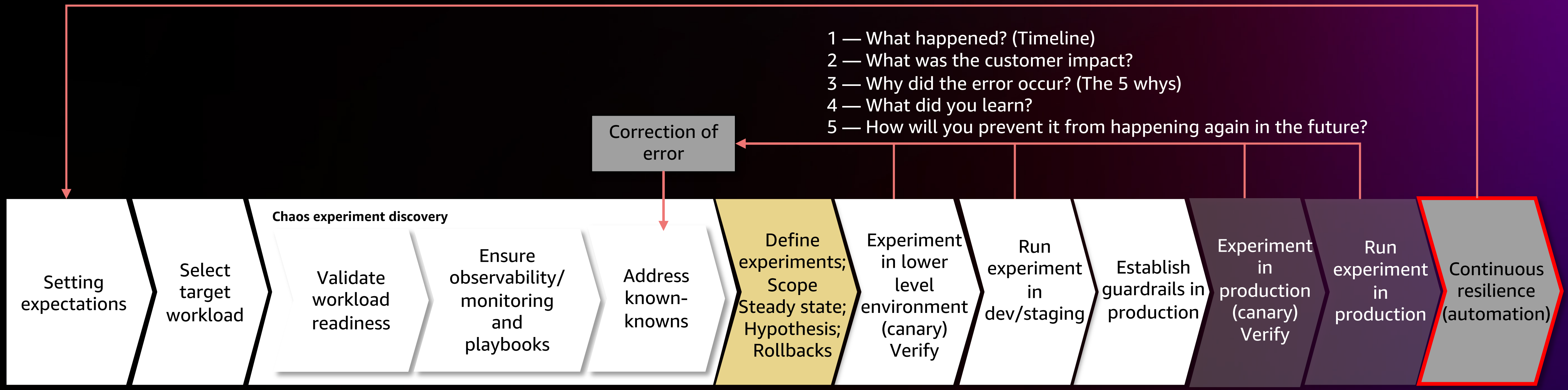
What did we learn?

Was everyone needed present?

How do we share what we've learned?

# Continuous resilience

PRIORITIZE FINDINGS, FIX THEM, REITERATE; CONTINUOUSLY RUN EXPERIMENTS AS OFTEN AS WORKLOAD NEEDS; SCALE MECHANISM TO OTHER WORKLOADS



1 — What happened? (Timeline)
2 — What was the customer impact?
3 — Why did the error occur? (The 5 whys)
4 — What did you learn?
5 — How will you prevent it from happening again in the future?

Correction of error
Setting expectations
Select target workload
Chaos experiment discovery
Validate workload readiness
Ensure observability/ monitoring and playbooks
Address known-knowns
Define experiments; Scope Steady state; Hypothesis; Rollbacks
Experiment in lower level environment (canary) Verify
Run experiment in dev/staging
Establish guardrails in production
Experiment in production (canary) Verify
Run experiment in production
Continuous resilience (automation)

# Automate the experiments

AUTOMATE

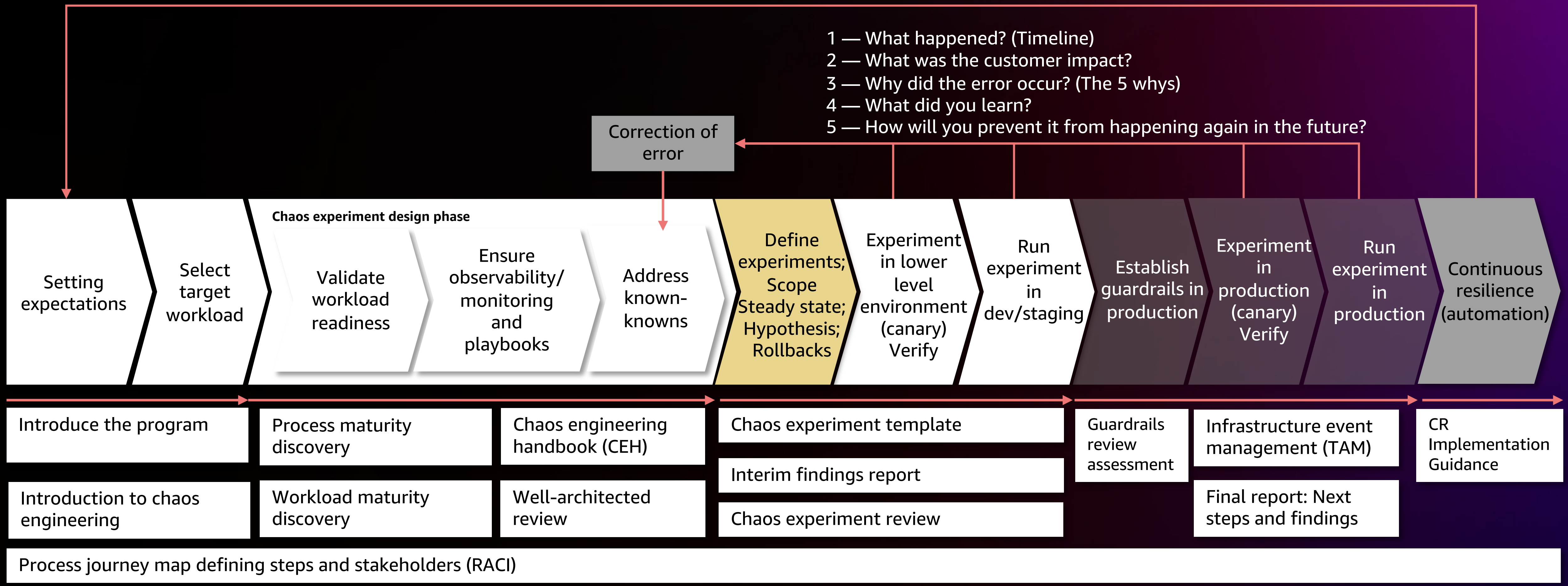
Individual experiments

AWS GameDays

Scheduled experiments

# Supporting documents

PRIORITIZE FINDINGS, FIX THEM, REITERATE; CONTINUOUSLY RUN EXPERIMENTS AS OFTEN AS WORKLOAD NEEDS; SCALE MECHANISM TO OTHER WORKLOADS



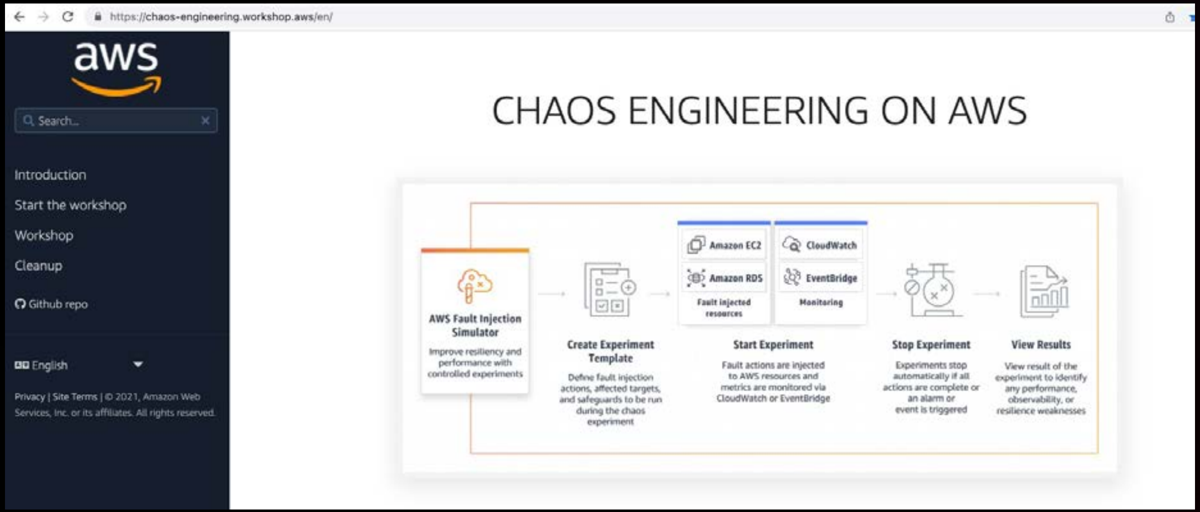
1 — What happened? (Timeline)
2 — What was the customer impact?
3 — Why did the error occur? (The 5 whys)
4 — What did you learn?
5 — How will you prevent it from happening again in the future?

Correction of error
Setting expectations
Select target workload
Chaos experiment design phase
Validate workload readiness
Ensure observability/ monitoring and playbooks
Address known-knowns
Define experiments; Scope Steady state; Hypothesis; Rollbacks
Experiment in lower level environment (canary) Verify
Run experiment in dev/staging
Establish guardrails in production
Experiment in production (canary) Verify
Run experiment in production
Continuous resilience (automation)

- Introduce the program
- Introduction to chaos engineering
- Process maturity discovery
- Workload maturity discovery
- Chaos engineering handbook (CEH)
- Well-architected review
- Chaos experiment template
- Interim findings report
- Chaos experiment review
- Guardrails review assessment
- Infrastructure event management (TAM)
- Final report: Next steps and findings
- CR Implementation Guidance
- Process journey map defining steps and stakeholders (RACI)

# Chaos engineering AWS resources

# Chaos engineering/observability workshops


https://chaos-engineering.workshop.aws/en/
aws
Search...
Introduction
Start the workshop
Workshop
Cleanup
Github repo
English
Privacy | Site Terms | © 2021, Amazon Web Services, Inc. or its affiliates. All rights reserved.

CHAOS ENGINEERING ON AWS

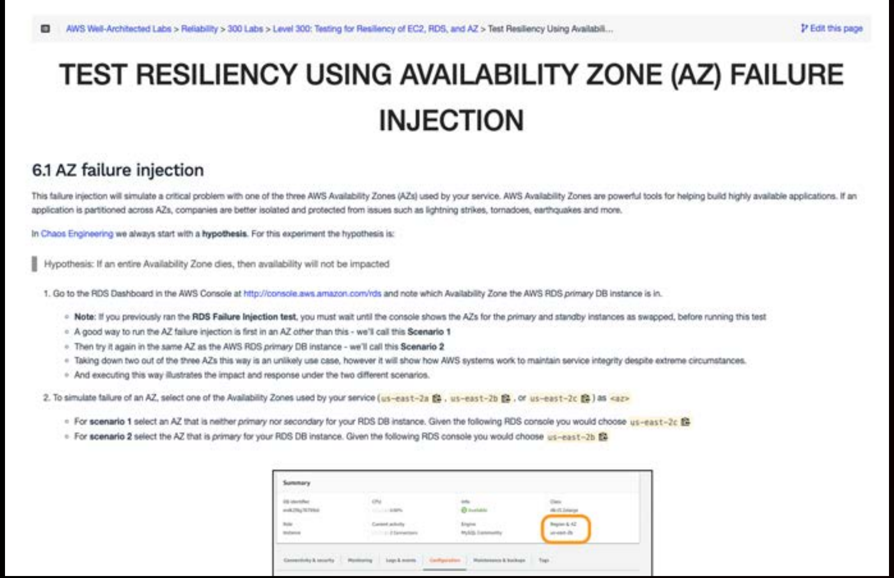
**AWS Fault Injection Simulator**
Improve resiliency and performance with controlled experiments

**Create Experiment Template**
Define fault injection actions, affected targets, and safeguards to be run during the chaos experiment

**Start Experiment**
Fault actions are injected to AWS resources and metrics are monitored via CloudWatch or EventBridge

**Stop Experiment**
Experiments stop automatically if all actions are complete or an alarm or event is triggered

**View Results**
View result of the experiment to identify any performance, observability, or resilience weaknesses


AWS Well-Architected Labs > Reliability > 300 Labs > Level 300: Testing for Resiliency of EC2, RDS, and AZ > Test Resiliency Using Availabili...
Edit this page

## TEST RESILIENCY USING AVAILABILITY ZONE (AZ) FAILURE INJECTION

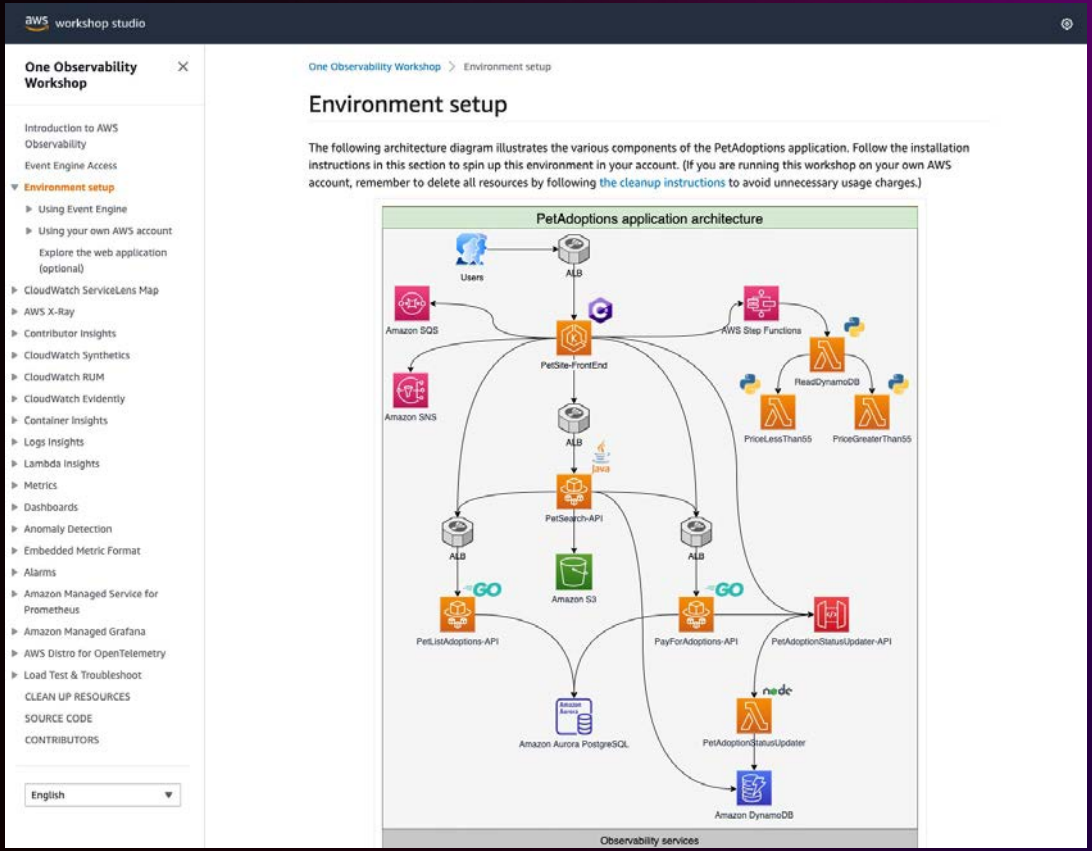
### 6.1 AZ failure injection

This failure injection will simulate a critical problem with one of the three AWS Availability Zones (AZs) used by your service. AWS Availability Zones are powerful tools for helping build highly available applications. If an application is partitioned across AZs, companies are better isolated and protected from issues such as lightning strikes, tornadoes, earthquakes and more.

In Chaos Engineering we always start with a **hypothesis**. For this experiment the hypothesis is:

> Hypothesis: If an entire Availability Zone dies, then availability will not be impacted

1. Go to the RDS Dashboard in the AWS Console at http://console.aws.amazon.com/rds and note which Availability Zone the AWS RDS *primary* DB instance is in.
   - **Note**: If you previously ran the **RDS Failure Injection test**, you must wait until the console shows the AZs for the *primary* and *standby* instances as swapped, before running this test
   - A good way to run the AZ failure injection is first in an AZ *other* than this - we'll call this **Scenario 1**
   - Then try it again in the same AZ as the AWS RDS *primary* DB instance - we'll call this **Scenario 2**
   - Taking down two out of the three AZs this way is an unlikely use case, however it will show how AWS systems work to maintain service integrity despite extreme circumstances.
   - And executing this way illustrates the impact and response under the two different scenarios.
2. To simulate failure of an AZ, select one of the Availability Zones used by your service (`us-east-2a`, `us-east-2b`, or `us-east-2c`) as `<az>`
   - For **scenario 1** select an AZ that is *neither primary nor secondary* for your RDS DB instance. Given the following RDS console you would choose `us-east-2c`
   - For **scenario 2** select the AZ that is *primary* for your RDS DB instance. Given the following RDS console you would choose `us-east-2b`



## Environment setup

The following architecture diagram illustrates the various components of the PetAdoptions application. Follow the installation instructions in this section to spin up this environment in your account. (If you are running this workshop on your own AWS account, remember to delete all resources by following the cleanup instructions to avoid unnecessary usage charges.)

# Chaos engineering/observability workshops

Chaos engineering on AWS

Validating security guardrails with chaos engineering

Resilience engineering

Serverless chaos workshop

Observability workshop

AWS fault isolation boundaries

Multi-AZ resilience patterns

Chaos engineering stories

# Thank You!

Naren Gakka
twitter: @narengka