Test your Architectural Resilience with Chaos Engineering

Marcel Neidinger

Solutions Architect
Amazon Web Services

"Everything fails, all the time"

Werner Vogels
CTO Amazon Web Services

A few definitions

Resiliency

The ability for a system to recover from a failure induced by load, attacks, and failures.

Availability

Percentage of time a workload is available for use.

Fault Tolerance

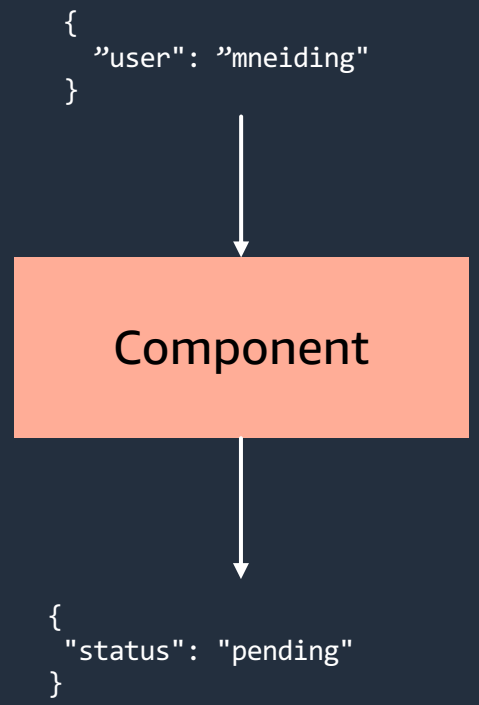
Handle interruptions without degrading the system performance

https://wa.aws.amazon.com/wat.concepts.wa-concepts.en.html

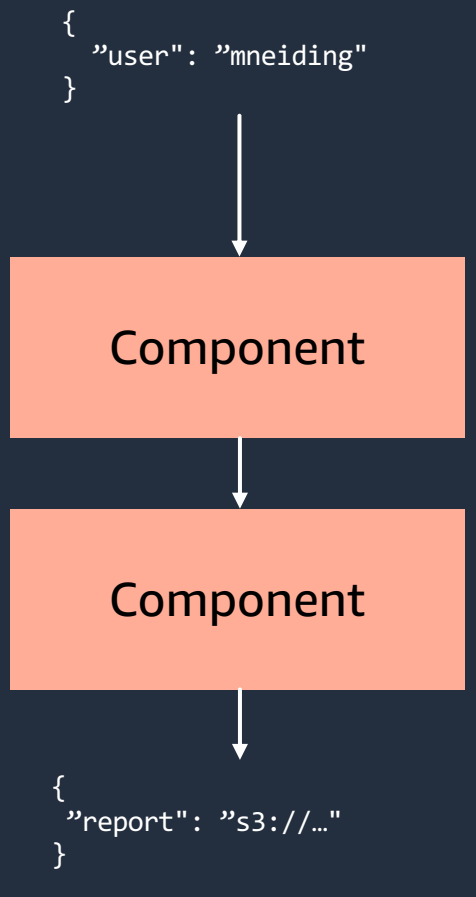
An Introduction to chaos engineering

Different ways of testing

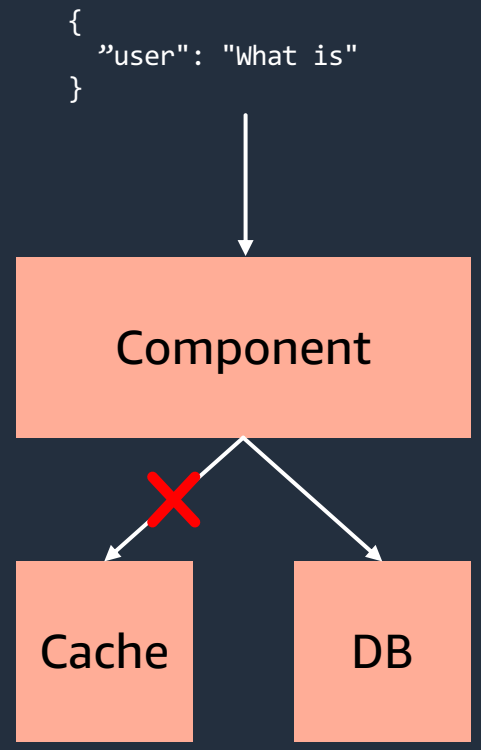
Unit Testing



Scenario Testing*



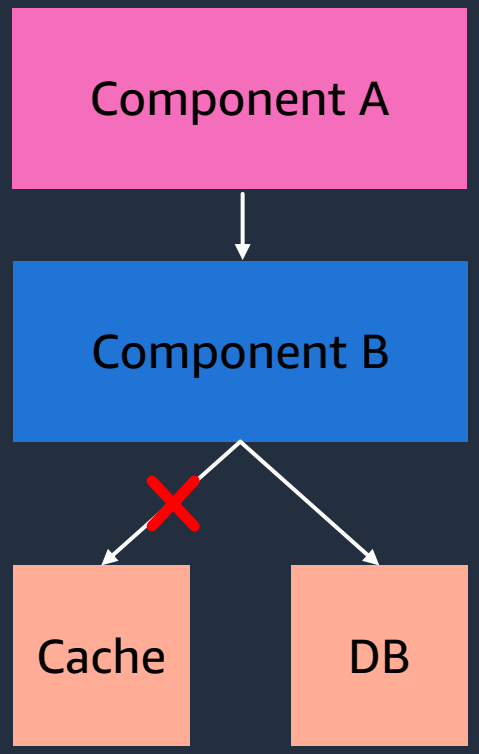
(Chaos) Experiments



*Sometimes also called Integration Testing

Failures are usually chain reactions of small incidents

A TIMELINE



An error in our automation changes a security rule to drop all traffic to the cache node

Component B redirects all traffic to the DB. Response time increases due to read-heavy operations.

Component A has a wrongly configured `wait` time leading to aborted requests.

Component A sends retries increasing the load on component B

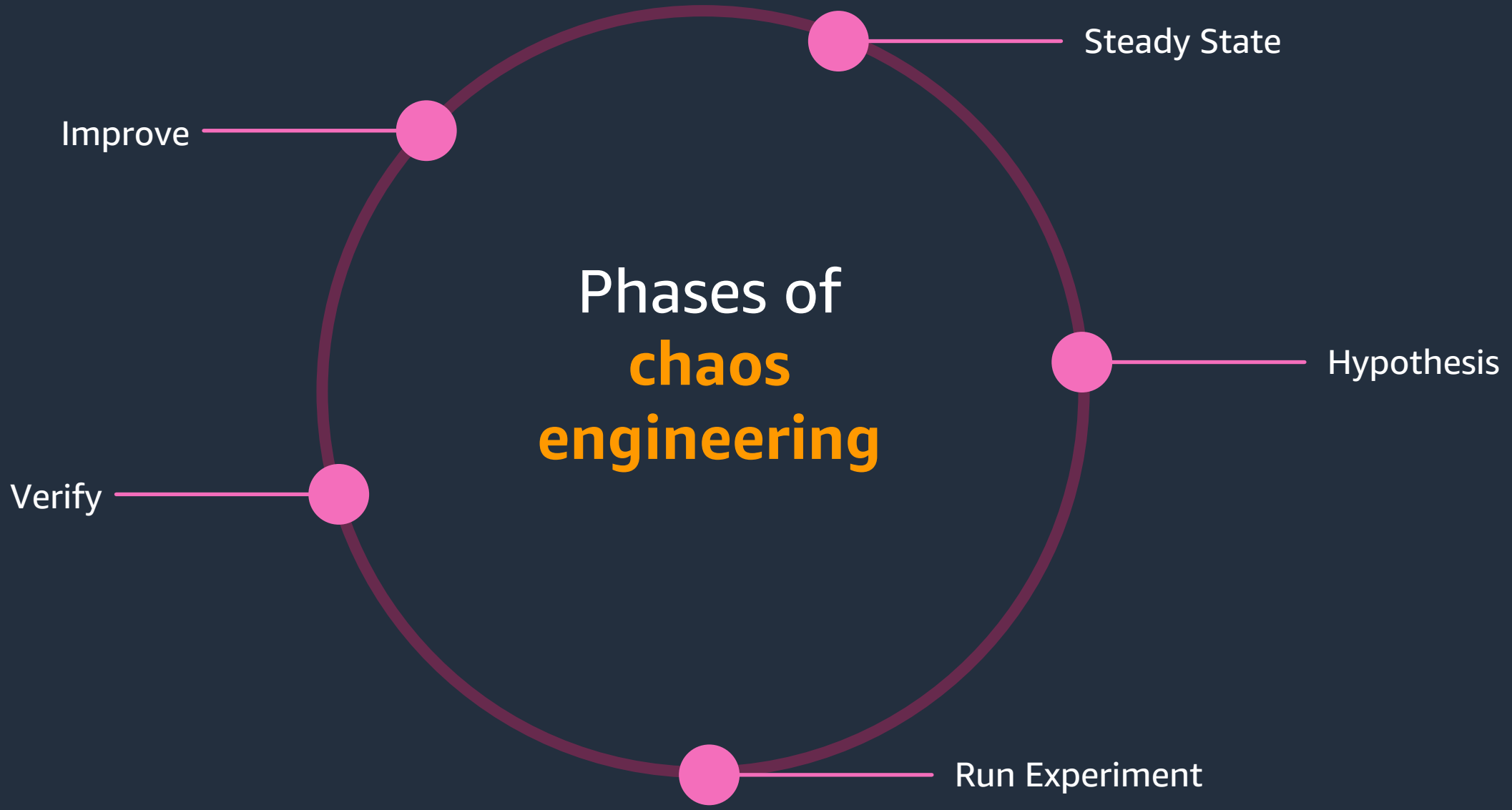
Component B can't scale and errors out.

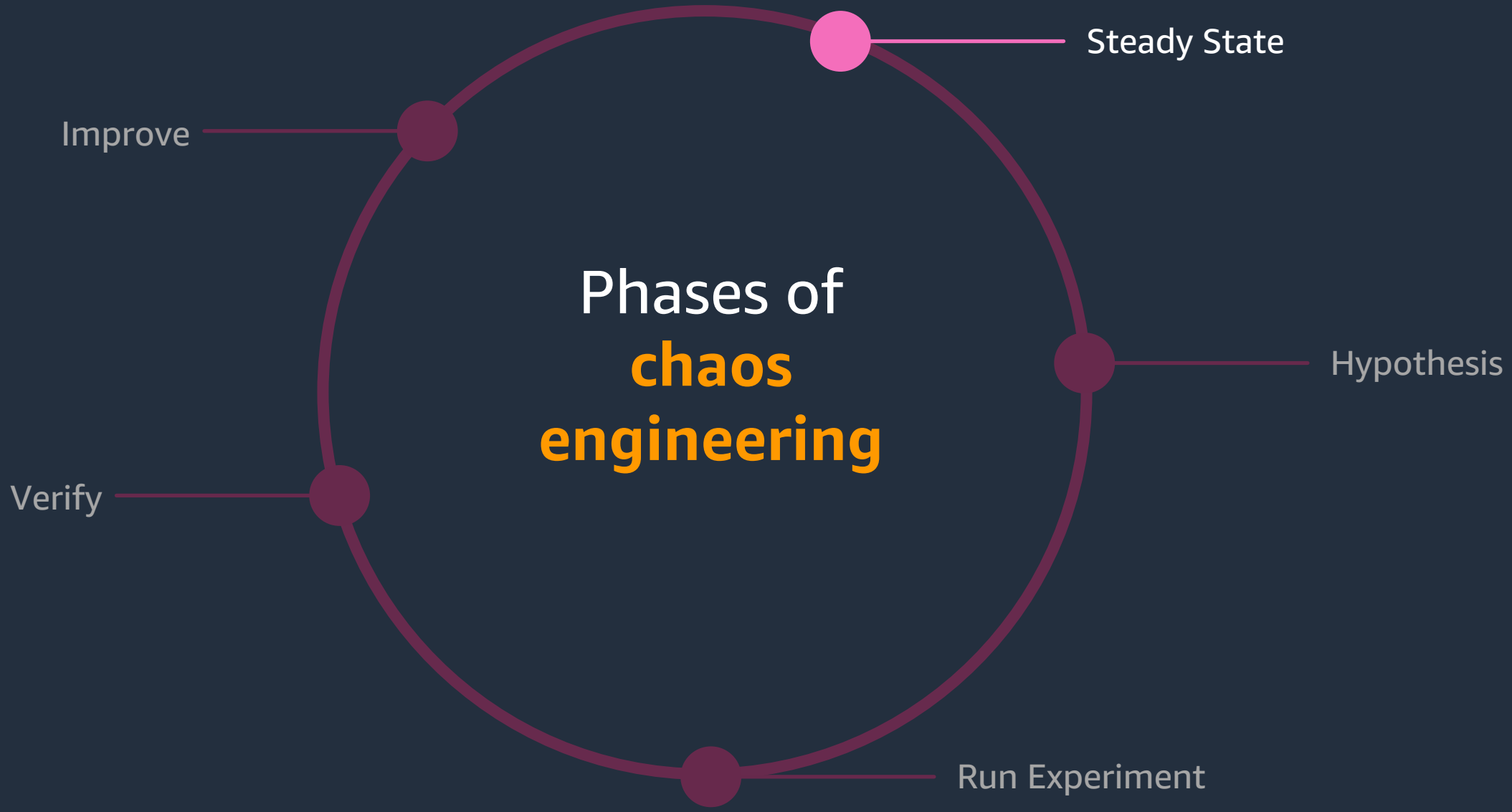
Circuit Breakers in Component A trigger. The system state is degraded.

Chaos Engineering is **NOT** about breaking things randomly without a purpose.

Chaos Engineering is about breaking things in a controlled environment and through well-planned experiments.

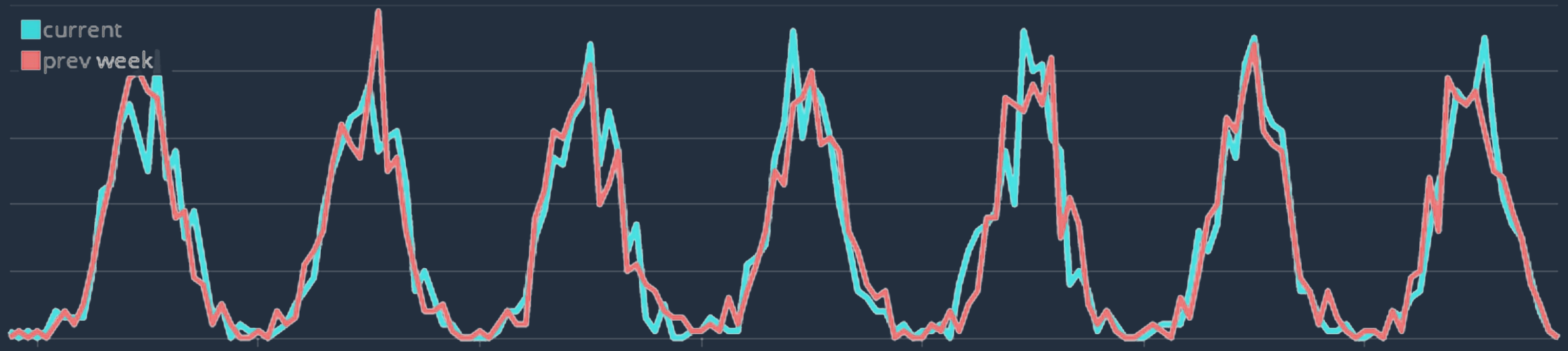
While you **CAN** do this in production you **DON'T** have to.





Steady State

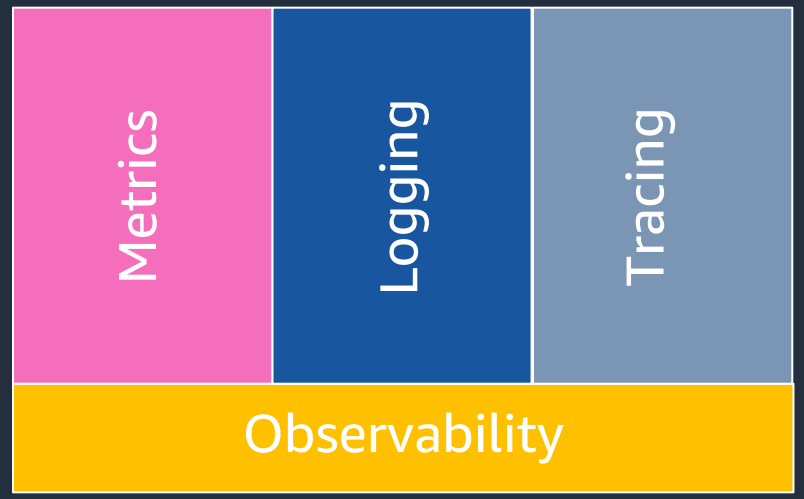
THE “NORMAL” BEHAVIOR OF YOUR SYSTEM



Business + Ops

Observability as a Pre-Requirement

HOW DO YOU MEASURE "NORMAL"?

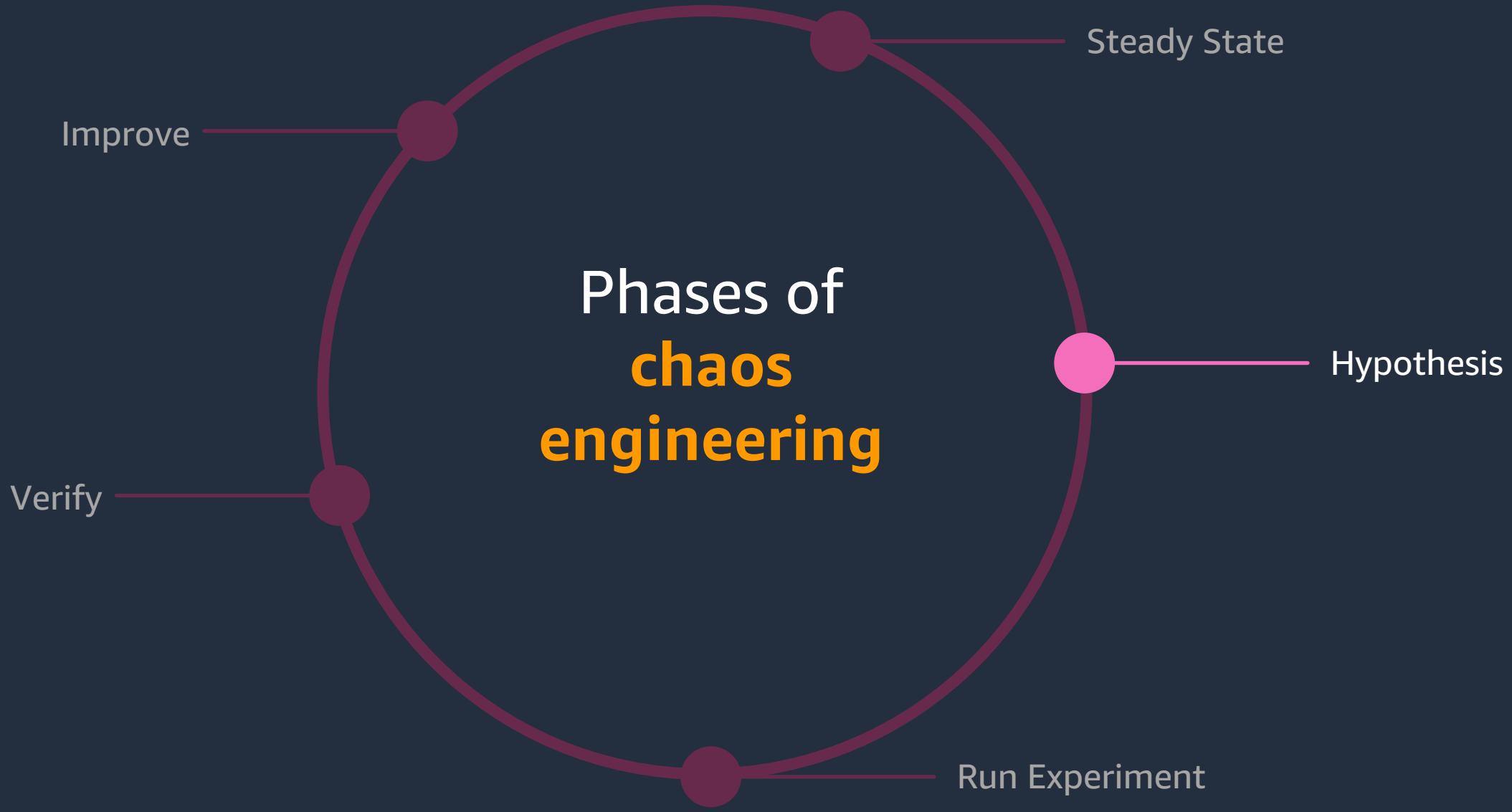


Do you understand the inner workings of your application?

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?



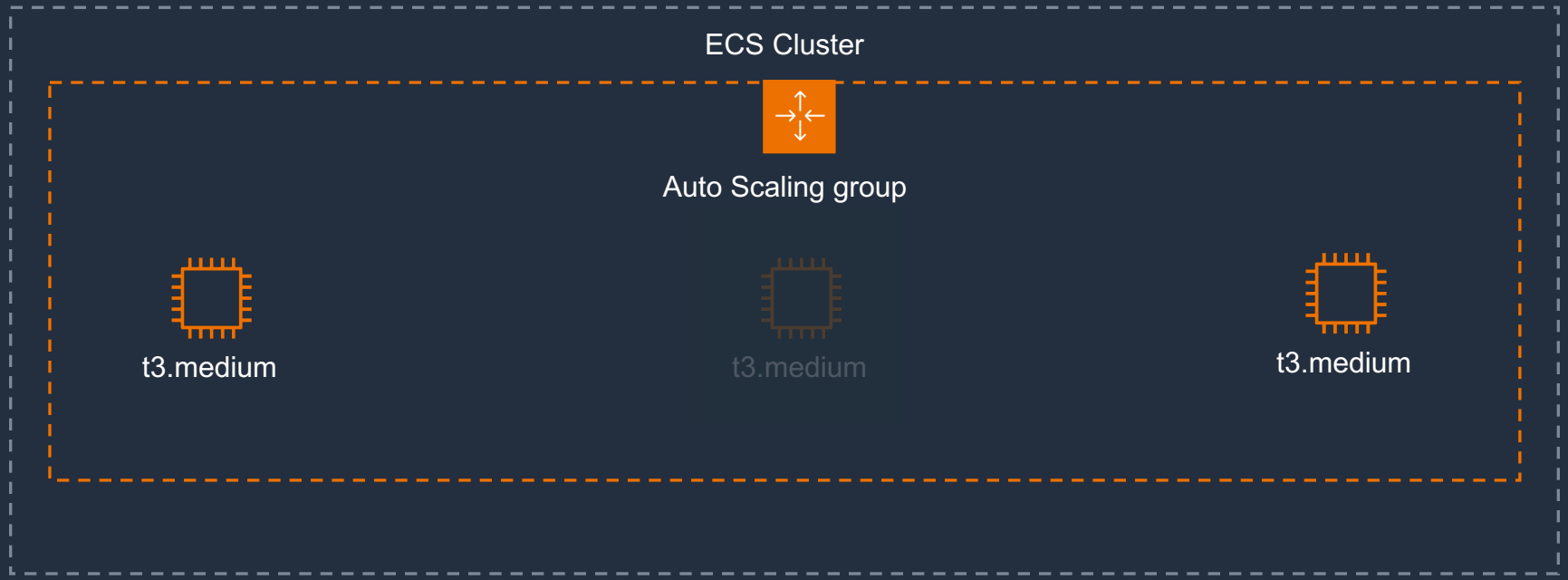
Hypothesize that this **steady state** will continue in both the control group and the experimental group.

unaffected parts of our system

affected parts of our system

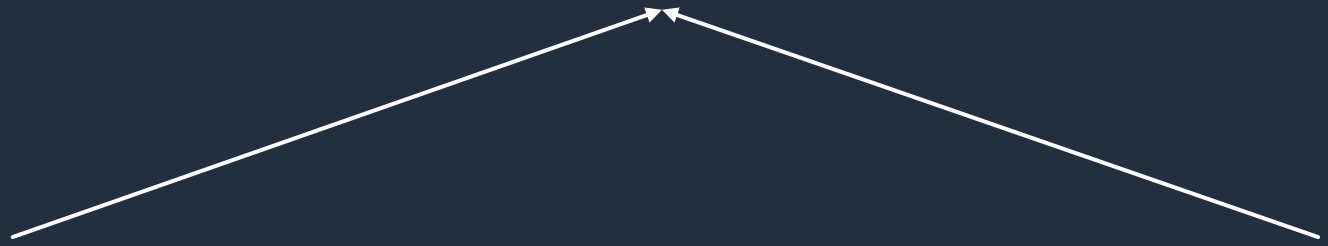
Hypothesis

What if a host of our ECS cluster shuts down?



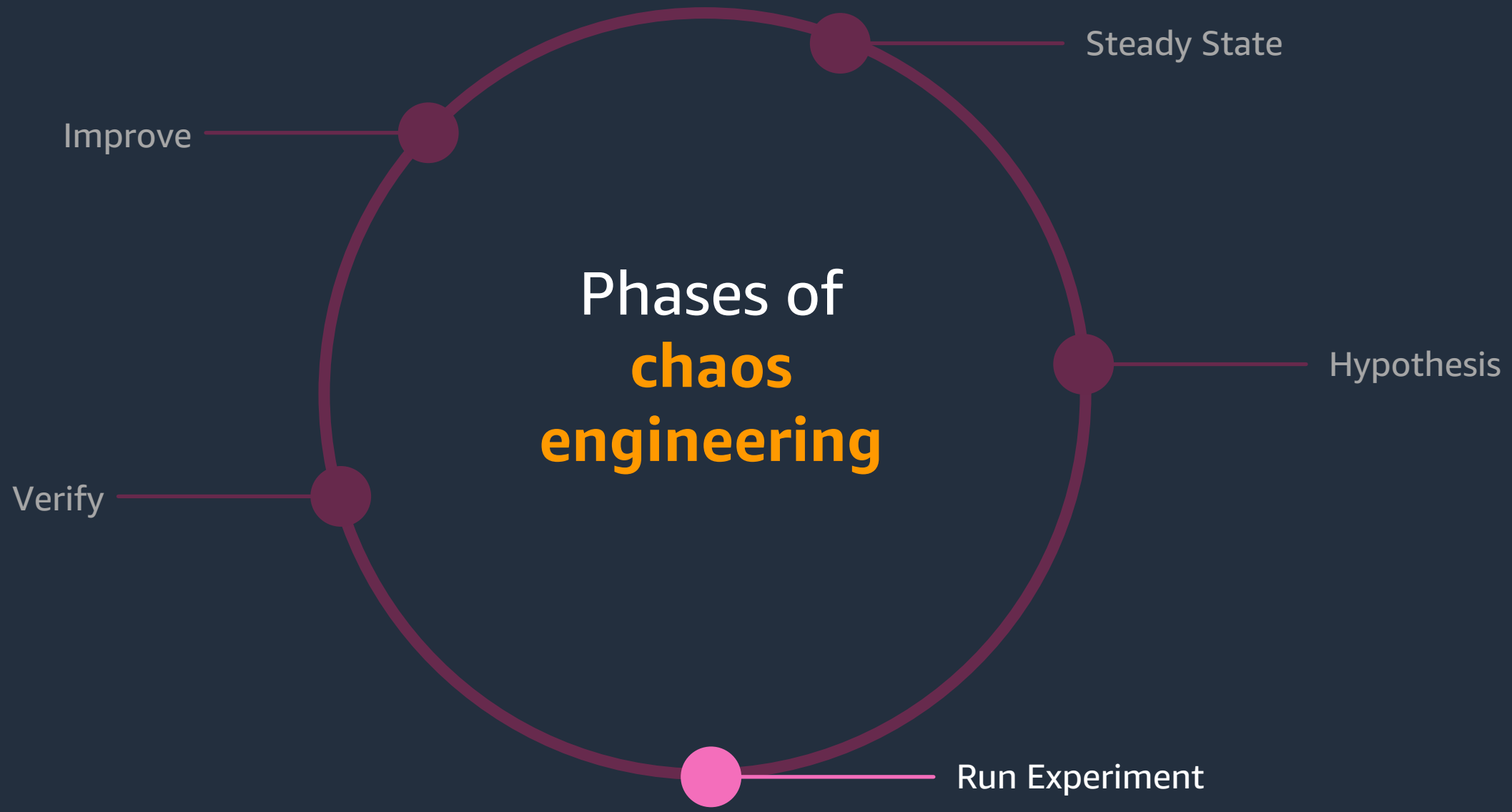
Hypothesis Considerations

What if a host of our ECS cluster shuts down?



Have we architected against this?

Do we want to guard against this?



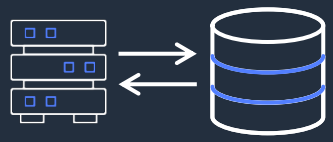
Designing real-world experiments

Code & configuration

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

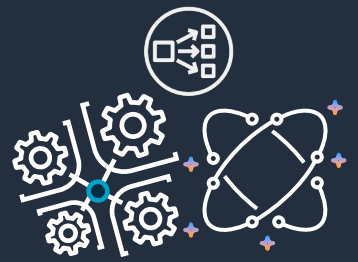
Infrastructure

e.g., datacenter failure, hardware failure



Data and state

e.g., data corruption, overload



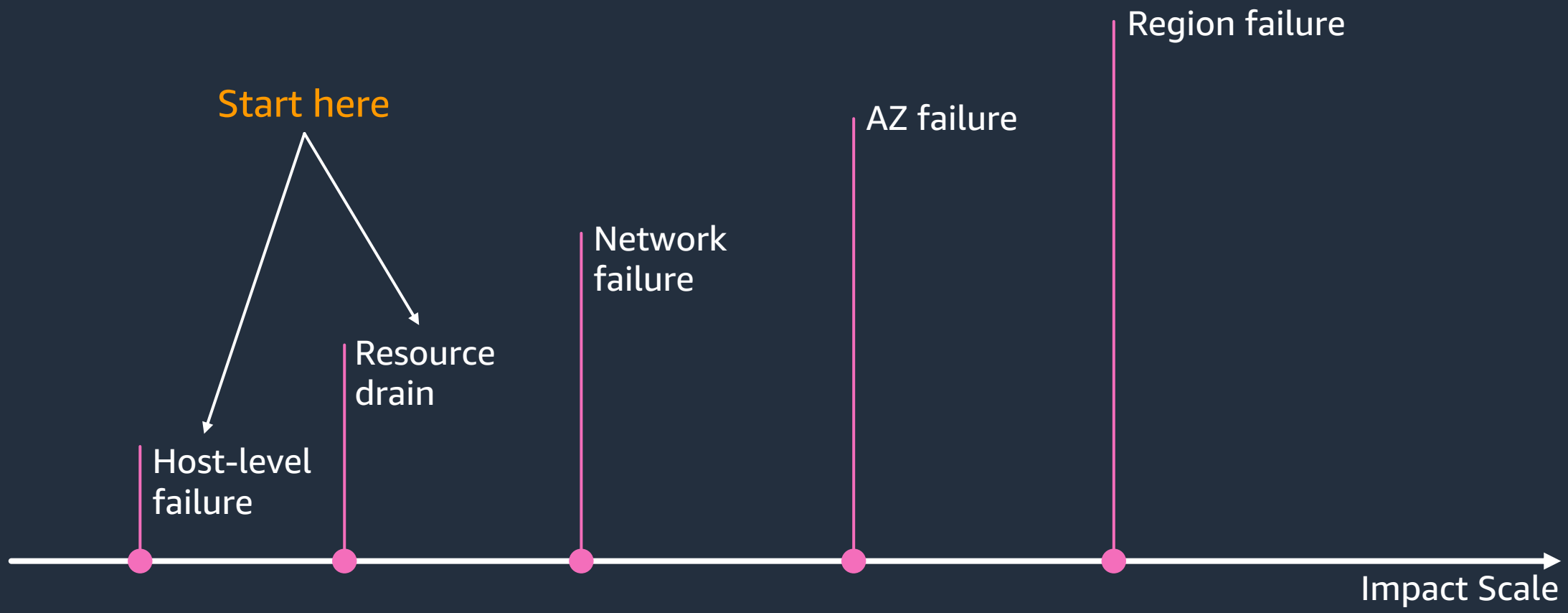
Dependencies

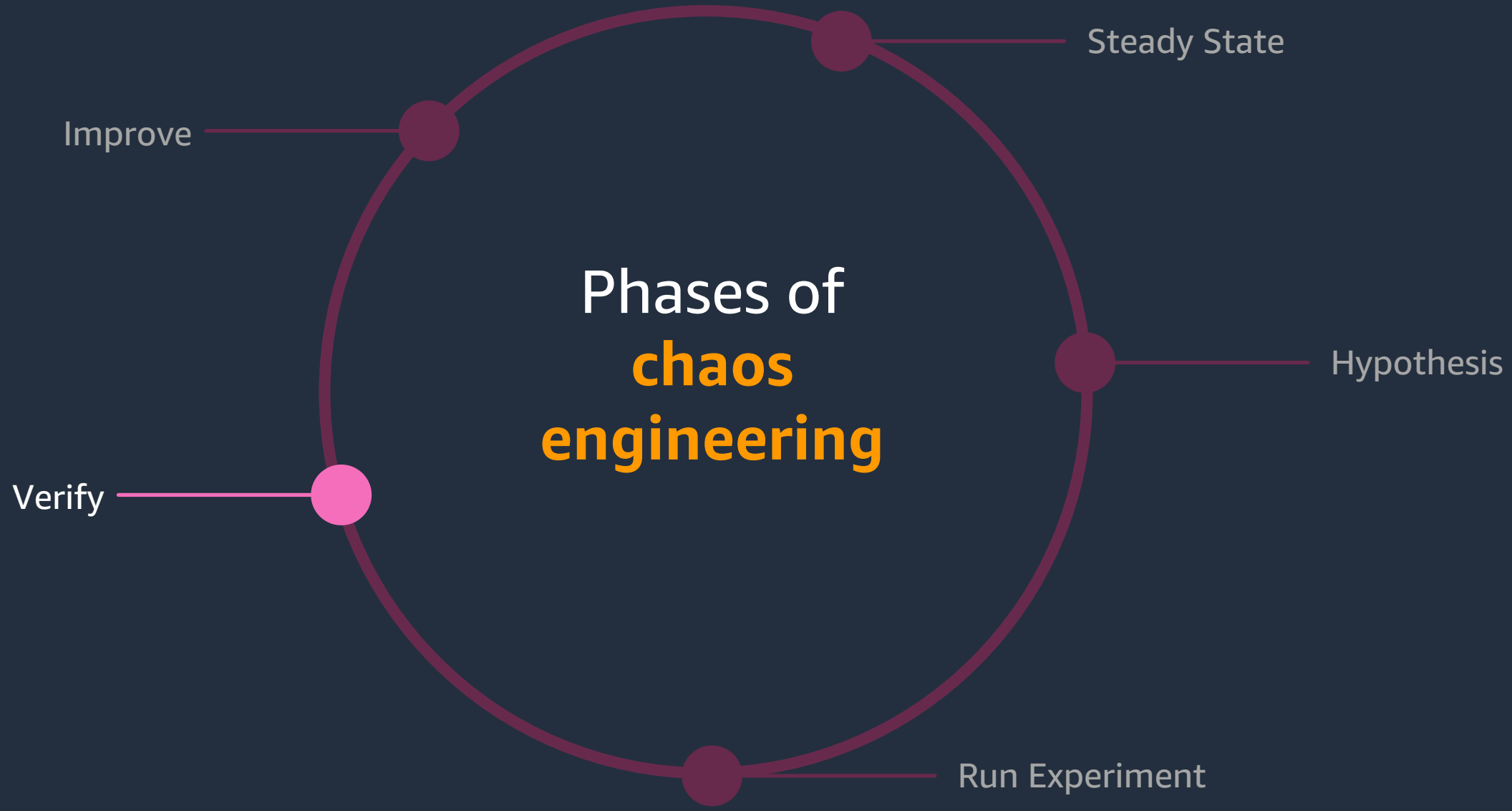
e.g., third-party integrations, AWS services

Highly unlikely, but technically feasible

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

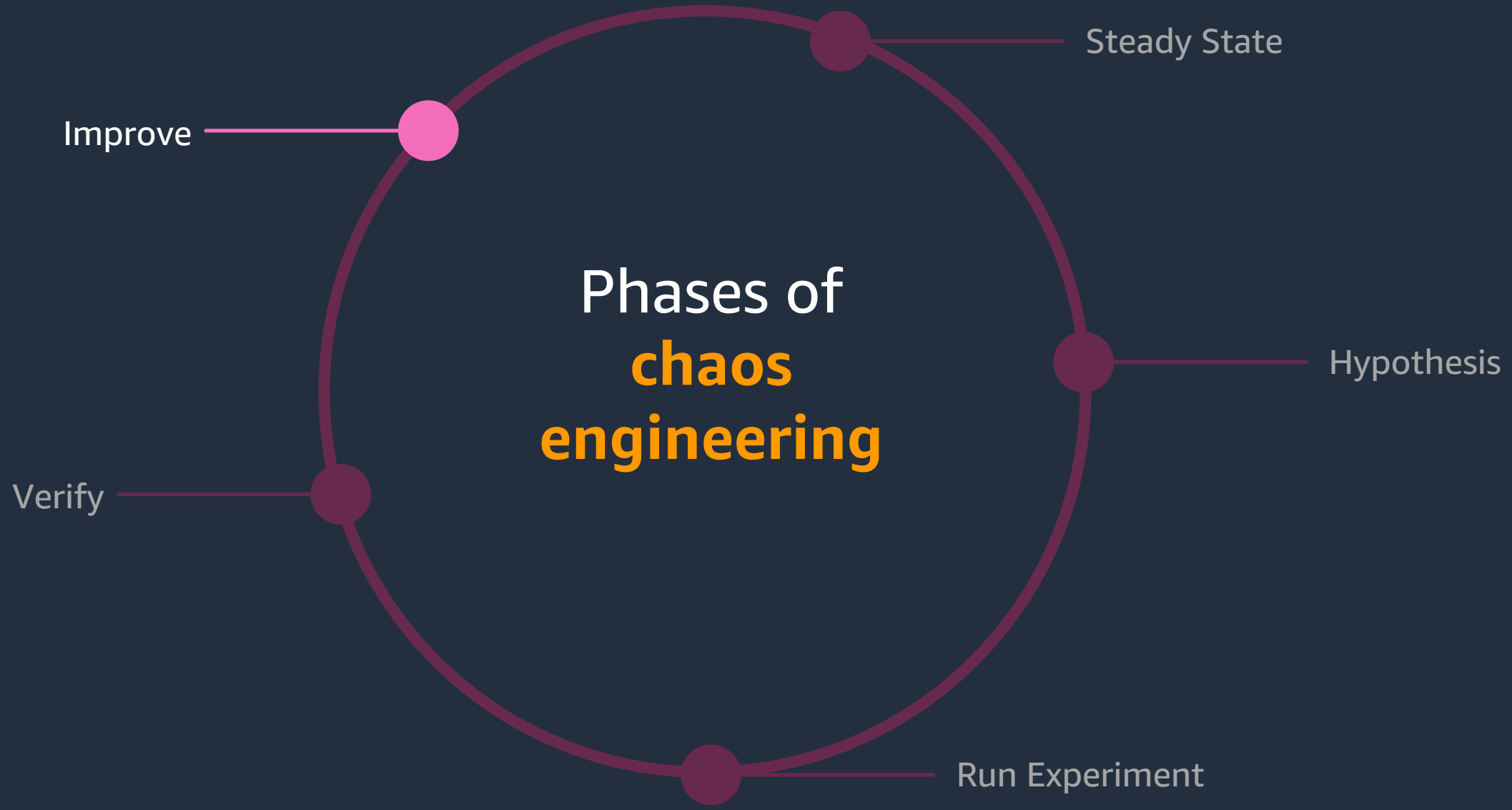
Start small





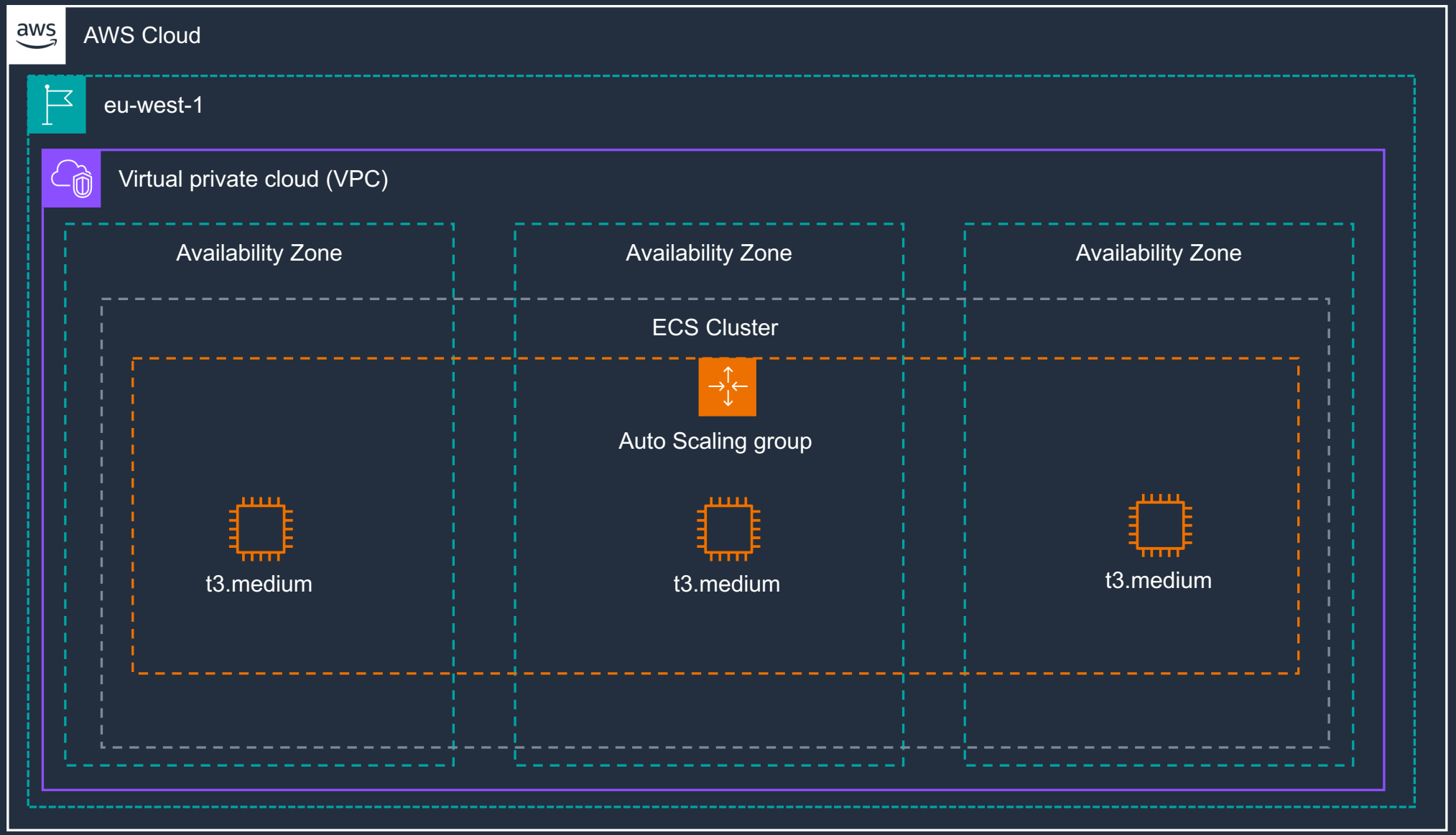
Quantify your experiment

- Time to detect?
- Time to notification?
- Time to escalation?
- Time for graceful degradation to start?
- Time for self-healing to happen?
- Time to (full|partial) recovery?
- Time to all-stable?



Fix it!

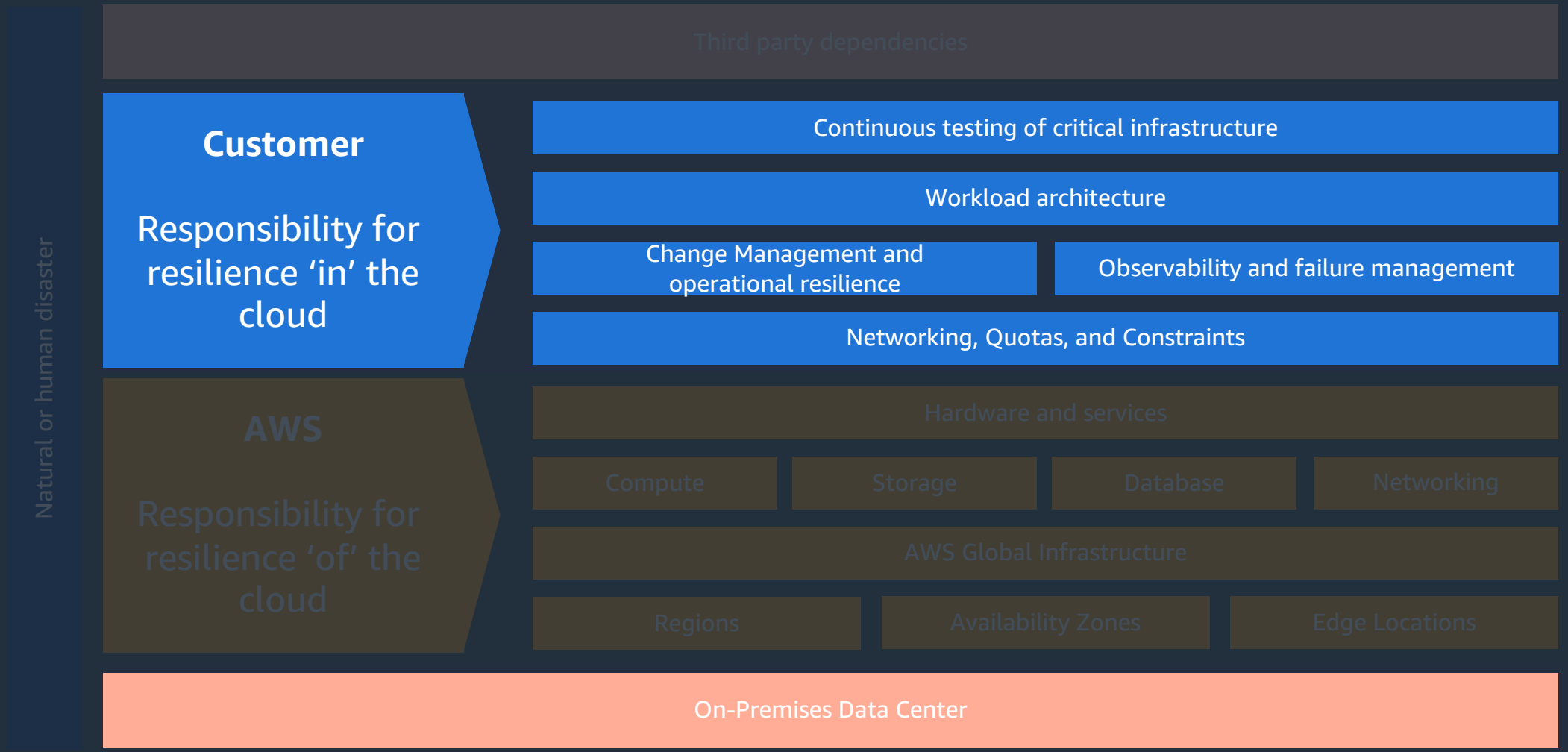
A simple example of chaos engineering



Let’s formulate an experiment

- **Steady State** Application is reachable. Response time is <500ms
- **Hypothesis** What if a ECS node gets terminated?
- **Experiment** Turn of a EC2 instance running the ECS cluster

Resilience on AWS – A shared responsibility



Thank you!

Marcel Neidinger
mneiding@amazon.com

A few things you can do

DDoS yourself*

$ wrk -t12 -c400 -d30s http://127.0.0.1/api/health

Run CPU Stress with stress-ng

$ stress-ng --cpu 0 --cpu-method matrixprod –t 60s

Add latency to your network

$ tc qdisc add dev eth0 root netem delay 300ms

* Note: **Do not run DDoS against your AWS environments**.
More information: https://aws.amazon.com/security/ddos-simulation-testing/