CHAOS ENGINEERING BOOTCAMP

TAMMY BUTOW, GREMLIN
SRECON AMERICAS 2018

TAMMY BUTOW

SRE, GREMLIN

CAUSING CHAOS IN PROD
SINCE 2009

@TAMMYBUTOW
@GREMLININC
GREMLIN.COM

THANK YOU FRIENDS!

ANA MEDINA

UBER,
UDESTROY

KIM BANNERMAN

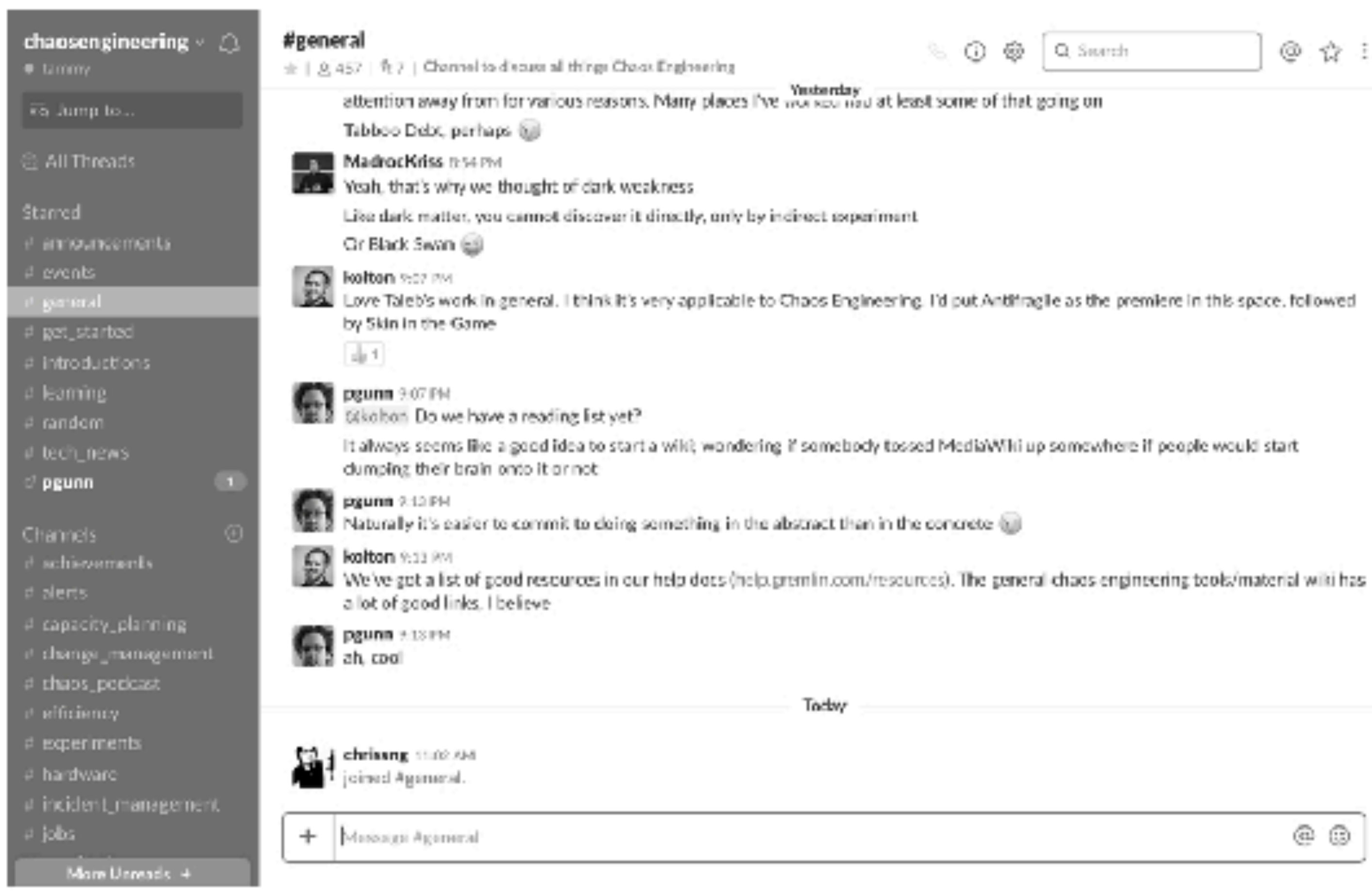
GOOGLE,
KUBERNETES

MATT WILLIAMS

DATADOG,
ALL THE THINGS

CHAOS ENGINEERING SLACK

JOIN THE #SRECON18 SLACK CHANNEL



https://slofile.com/slack/chaosengineering

THE CHAOS BOOTCAMP

1. DISCOVER AND EXPLORE THE PRACTICE OF CHAOS ENGINEERING
2. IMMERSE YOURSELF IN A DISCUSSION ON CHAOS ENGINEERING
3. DELVE INTO CHAOS ENGINEERING ON DISTRIBUTED SYSTEMS
4. EXPLORE THE APPLICATION OF CHAOS ENGINEERING IN YOUR COMPANY
5. LEARN HOW TO CRAFT YOUR OWN CHAOS ENGINEERING EXPERIMENTS
6. LEARN TECHNIQUES TO EVALUATE YOUR CHAOS ENGINEERING PRACTICE

THE CHAOS BOOTCAMP

+ **LAYING THE FOUNDATIONS (2:00 - 3:00)**
+ CHAOS ENGINEERING DISCUSSION (3:00 - 3:30)
+ AFTERNOON BREAK (3:30 - 4:00)
+ DISTRIBUTED SYSTEMS CHAOS (4:00 - 4:30)
+ CHAOS ENGINEERING IN YOUR COMPANY (4:30 - 4:45)
+ CRAFT YOUR OWN EXPERIMENTS (4:45- 5:00)
+ FEEDBACK AND EVALUATION TECHNIQUES (5:00-5:15)
+ ADVANCED TOPICS & Q + A (5:15 - 5:30)

PART I: LAYING THE FOUNDATION

WHAT IS CHAOS ENGINEERING

THOUGHTFUL PLANNED EXPERIMENTS DESIGNED TO REVEAL THE WEAKNESSES
IN OUR SYSTEMS.

WHY DO DISTRIBUTED SYSTEMS NEED CHAOS?

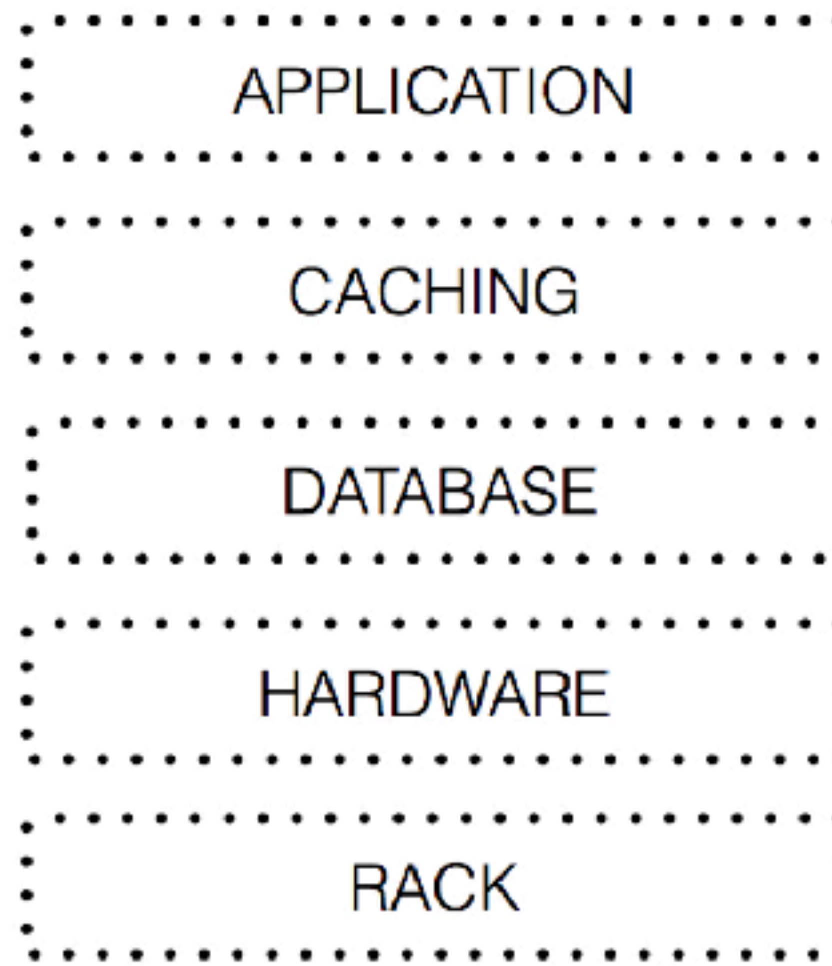
DISTRIBUTED SYSTEMS HAVE NUMEROUS SYSTEM PARTS.

HARDWARE AND FIRMWARE FAILURES ARE COMMON. OUR SYSTEMS AND COMPANIES SCALE RAPIDLY

HOW DO YOU BUILD A RESILIENT SYSTEM WHILE YOU SCALE?
WE USE CHAOS!

FULL-STACK CHAOS INJECTION

YOU CAN INJECT CHAOS AT ANY LAYER TO INCREASE SYSTEM RESILIENCE AND SYSTEM KNOWLEDGE.



WHO USES CHAOS ENGINEERING?

1. NETFLIX
2. DROPBOX
3. GOOGLE
4. NATIONAL AUSTRALIA BANK
5. JET

WHAT ARE COMMON EXCUSES TO NOT USE CHAOS ENGINEERING?

HANDS-ON TUTORIAL (LET'S JUMP IN!)

NOW IT IS TIME TO CREATE CHAOS. WE WILL ALL BE DOING A HANDS-ON ACTIVITY WHERE WE INJECT FAILURE.

TIME TO USE YOUR SERVERS

IN GROUPS OF 3,
SSH INTO YOUR
KUBERNETES CLUSTER USING THE
CHAOS USER

VISIT YOUR PRIMARY IN
YOUR BROWSER (PORT 30001)

YOU MUST BE MEASURING METRICS AND REPORTING ON THEM TO IMPROVE YOUR SYSTEM RESILIENCE.

CHAOS WITHOUT MONITORING IS FUTILE

THE LACK OF PROPER MONITORING IS NOT USUALLY THE SOLE CAUSE OF A PROBLEM, BUT IT IS OFTEN A SERIOUS CONTRIBUTING FACTOR. AN EXAMPLE IS THE NORTHEAST BLACKOUT OF 2003.

COMMON ISSUES INCLUDE:

+ HAVING THE WRONG TEAM DEBUG
+ NOT ESCALATING
+ NOT HAVING A BACKUP ON-CALL

Northeast blackout of 2003

From Wikipedia, the free encyclopedia

The **Northeast blackout of 2003** was a widespread power outage that occurred throughout parts of the Northeastern and Midwestern United States and the Canadian province of Ontario on Thursday, August 14, 2003, just after 4:10 p.m. EDT.[1]

Some power was restored by 11 p.m. Many others did not get their power back until two days later. In more remote areas it took nearly a week to restore power.[2] At the time, it was the world's second most widespread blackout in history, after the 1999 Southern Brazil blackout.[3][4] The outage, which was much more widespread than the Northeast Blackout of 1965, affected an estimated 10 million people in Ontario and 45 million people in eight U.S. states.

The blackout's primary cause was a software bug in the alarm system at a control room of the FirstEnergy Corporation, located in Ohio. A lack of alarm left operators unaware of the need to re-distribute power after overloaded transmission lines hit unpruned foliage, which triggered a race condition in the control software. What would have been a manageable local blackout cascaded into massive widespread distress on the electric grid.

This image shows states and provinces that experienced power outages. Not all areas within these political boundaries were affected.

Contents [show]

Immediate impact [edit]

According to the New York Independent System Operator (NYISO) – the ISO responsible for managing the New York state power grid – a 3,500 megawatt power surge (towards Ontario) affected the transmission grid at 4:10:39 p.m. EDT.[5]

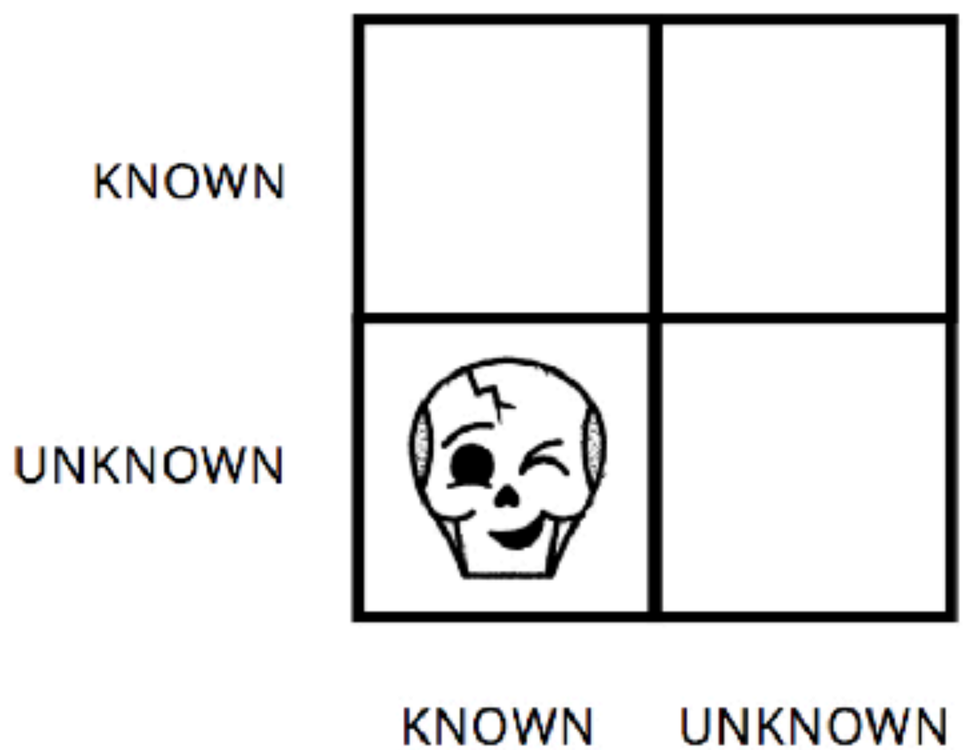
A LACK OF ALARMS LEFT OPERATORS UNAWARE OF THE NEED TO RE-DISTRIBUTE POWER AFTER OVERLOADED TRANSMISSION LINES HIT UNPRUNED FOLIAGE.

THIS TRIGGERED A RACE CONDITION IN THE CONTROL SOFTWARE.

CASE STUDY:
KUBERNETES SOCK SHOP

1. UNDERSTAND SYSTEM
2. DETERMINE SLAs/SLOs/KPIs
3. SETUP MONITORING
4. INJECT CHAOS
5. MEASURE RESULTS
6. LEARN
7. INCREASE SYSTEM RESILIENCE

CHAOS TYPES



LET'S INJECT KNOWN CHAOS

1. GO TO YOUR CHAOS REPO

```
$ su - experiments
$ cd chaos_engineering_bootcamp
```

LET'S INJECT KNOWN CHAOS

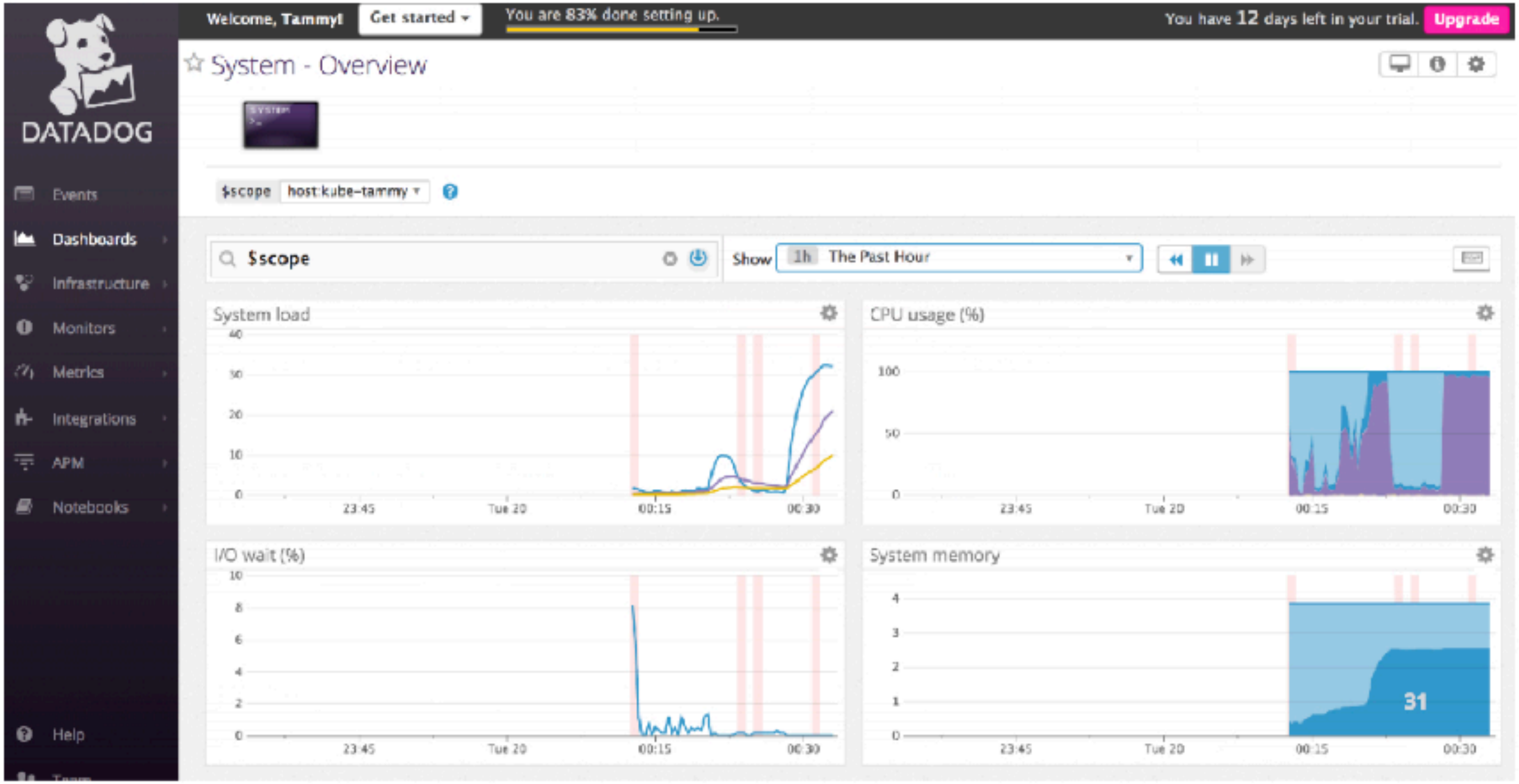
```
$ ls chaos_engineering_bootcamp
$ chmod +x chaos_cpu.sh
$ ./chaos_cpu.sh
$ top
```

CHAOS IN TOP

```
top - 07:30:27 up 18 min,  1 user,  load average: 28.42, 12.45, 5.54
Tasks: 266 total,  33 running, 233 sleeping,   0 stopped,   0 zombie
%Cpu(s): 96.9 us,  2.8 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.3 st
KiB Mem :  4046532 total,   316792 free,  2351388 used,  1378360 buff/cache
KiB Swap:        0 total,        0 free,        0 used.  1386896 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
18087 chaos     20   0   13936   3516   3080 R   6.2  0.1   0:07.10 openssl
18107 chaos     20   0   13936   3472   3036 R   6.2  0.1   0:07.13 openssl
18125 chaos     20   0   13936   3592   3160 R   6.2  0.1   0:07.10 openssl
18093 chaos     20   0   13936   3336   2900 R   5.9  0.1   0:07.23 openssl
18094 chaos     20   0   13936   3464   3032 R   5.9  0.1   0:07.05 openssl
18102 chaos     20   0   13936   3500   3064 R   5.9  0.1   0:07.10 openssl
18103 chaos     20   0   13936   3520   3084 R   5.9  0.1   0:07.30 openssl
18108 chaos     20   0   13936   3424   2992 R   5.9  0.1   0:07.08 openssl
18109 chaos     20   0   13936   3464   3028 R   5.9  0.1   0:07.14 openssl
18110 chaos     20   0   13936   3492   3056 R   5.9  0.1   0:07.21 openssl
18111 chaos     20   0   13936   3460   3012 R   5.9  0.1   0:06.99 openssl
18112 chaos     20   0   13936   3520   3084 R   5.9  0.1   0:07.09 openssl
18117 chaos     20   0   13936   3596   3160 R   5.9  0.1   0:06.97 openssl
18118 chaos     20   0   13936   3592   3156 R   5.9  0.1   0:07.01 openssl
18121 chaos     20   0   13936   3588   3156 R   5.9  0.1   0:07.09 openssl
18122 chaos     20   0   13936   3512   3076 R   5.9  0.1   0:07.08 openssl
18124 chaos     20   0   13936   3444   3000 R   5.9  0.1   0:07.05 openssl
18126 chaos     20   0   13936   3512   3076 R   5.9  0.1   0:07.13 openssl
18127 chaos     20   0   13936   3532   3096 R   5.9  0.1   0:07.05 openssl
18088 chaos     20   0   13936   3308   2884 R   5.6  0.1   0:07.17 openssl
18089 chaos     20   0   13936   3240   2820 R   5.6  0.1   0:07.21 openssl
18091 chaos     20   0   13936   3420   2984 R   5.6  0.1   0:07.19 openssl
18099 chaos     20   0   13936   3464   3028 R   5.6  0.1   0:07.06 openssl
18106 chaos     20   0   13936   3524   3088 R   5.6  0.1   0:07.11 openssl
18113 chaos     20   0   13936   3492   3056 R   5.6  0.1   0:06.96 openssl
18114 chaos     20   0   13936   3592   3160 R   5.6  0.1   0:07.18 openssl
18115 chaos     20   0   13936   3516   3080 R   5.6  0.1   0:07.18 openssl
18116 chaos     20   0   13936   3564   3132 R   5.6  0.1   0:07.20 openssl
18119 chaos     20   0   13936   3516   3080 R   5.6  0.1   0:07.09 openssl
18120 chaos     20   0   13936   3524   3084 R   5.6  0.1   0:07.12 openssl
18123 chaos     20   0   13936   3484   3048 R   5.6  0.1   0:07.04 openssl
18090 chaos     20   0   13936   3348   2916 R   5.2  0.1   0:07.18 openssl
 8503 root      20   0  462516  89856  42868 S   4.9  2.2   0:56.51 kubelet
 8781 root      20   0  104668  70156  41708 S   2.0  1.7   0:25.40 kube-controller
 1479 root      20   0 1436024  80344  29960 S   1.0  2.0   3:09.70 dockerd
 8975 root      20   0  178744 138264  46524 S   1.0  3.4   0:22.28 kube-apiserver
 8835 root      20   0 10.045g  40892  16656 S   0.7  1.0   0:11.76 etcd
13340 10001     20   0 1649936 274304   9816 S   0.7  6.8   1:02.37 java
13370 999       20   0  950384  58624  27752 S   0.7  1.4   0:03.93 mongod
```

CHAOS IN DATADOG



LET'S STOP THE KNOWN CHAOS

1. KILL WHAT I RAN

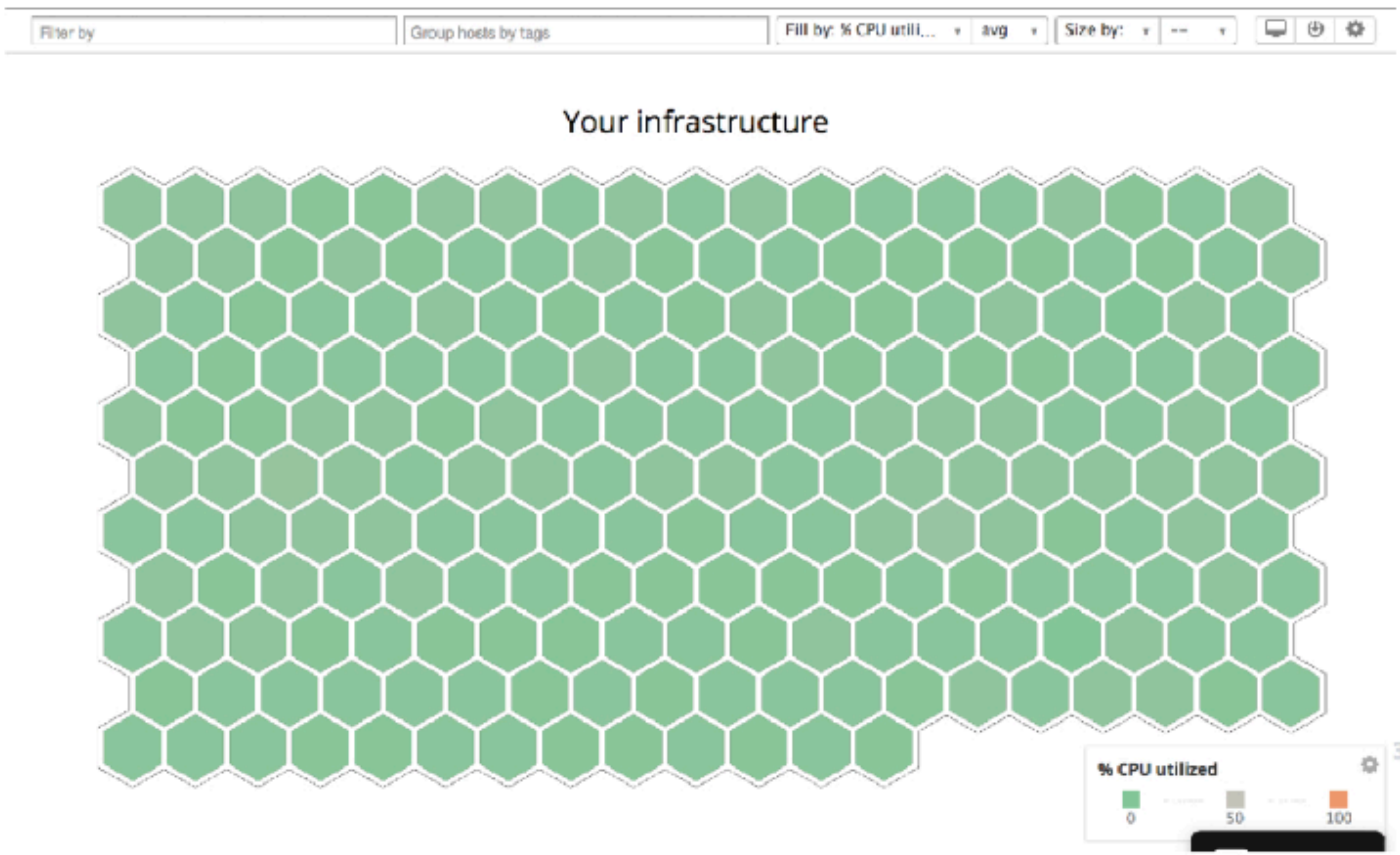
```
$ pkill -u experiments
```

NO MORE CHAOS IN TOP

```
Tasks: 200 total,   2 running, 198 sleeping,   0 stopped,   0 zombie
%Cpu(s):  4.9 us,  2.6 sy,  0.0 ni, 91.8 id,  0.2 wa,  0.0 hi,  0.2 si,  0.3 st
KiB Mem :  4046532 total,   344328 free,  2322212 used,  1379992 buff/cache
KiB Swap:        0 total,        0 free,        0 used.  1416592 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
 8503 root      20   0  463572  90832  42928 S   6.0  2.2   1:10.34 kubelet
 8781 root      20   0  104668  70144  41708 R   3.0  1.7   0:31.37 kube-controller
 8975 root      20   0  178744 138524  46524 S   2.0  3.4   0:26.85 kube-apiserver
 1479 root      20   0 1436024  80408  29960 S   1.3  2.0   3:12.67 dockerd
13924 999       20   0  952880  59340  27772 S   1.3  1.5   0:05.45 mongod
 8835 root      20   0 10.045g  42148  16836 S   1.0  1.0   0:14.81 etcd
14590 999       20   0  297828  62520  29340 S   1.0  1.5   0:05.99 mongod
 9064 root      20   0   49088  33684  23516 S   0.7  0.8   0:04.02 kube-proxy
    7 root      20   0       0      0      0 S   0.3  0.0   0:01.85 rcu_sched
 1601 root      20   0  885796  19192   8968 S   0.3  0.5   0:00.83 containerd
 1931 dd-agent  20   0  210460  14376   7456 S   0.3  0.4   0:01.02 trace-agent
 1938 dd-agent  20   0  195604  27456   7008 S   0.3  0.7   0:04.10 python
13370 999       20   0  950384  58624  27752 S   0.3  1.4   0:05.36 mongod
13604 10001     20   0 1650072 269164   9456 S   0.3  6.7   1:03.40 java
14680 1001      20   0  616976  58028  14108 S   0.3  1.4   0:04.10 node
20766 chaos     20   0   40540   3880   3192 R   0.3  0.1   0:00.05 top
    1 root      20   0   37948   6164   4128 S   0.0  0.2   0:05.35 systemd
    2 root      20   0       0      0      0 S   0.0  0.0   0:00.00 kthreadd
    3 root      20   0       0      0      0 S   0.0  0.0   0:00.32 ksoftirqd/0
    5 root       0 -20       0      0      0 S   0.0  0.0   0:00.00 kworker/0:0H
    8 root      20   0       0      0      0 S   0.0  0.0   0:00.00 rcu_bh
    9 root      rt   0       0      0      0 S   0.0  0.0   0:00.04 migration/0
   10 root      rt   0       0      0      0 S   0.0  0.0   0:00.00 watchdog/0
   11 root      rt   0       0      0      0 S   0.0  0.0   0:00.00 watchdog/1
   12 root      rt   0       0      0      0 S   0.0  0.0   0:00.05 migration/1
   13 root      20   0       0      0      0 S   0.0  0.0   0:00.37 ksoftirqd/1
   14 root      20   0       0      0      0 S   0.0  0.0   0:00.01 kworker/1:0
   15 root       0 -20       0      0      0 S   0.0  0.0   0:00.00 kworker/1:0H
   16 root      20   0       0      0      0 S   0.0  0.0   0:00.00 kdevtmpfs
   17 root       0 -20       0      0      0 S   0.0  0.0   0:00.00 netns
   18 root       0 -20       0      0      0 S   0.0  0.0   0:00.00 perf
```

DATADOG MONITORING



WHAT KIND OF CHAOS CAN YOU INJECT?

1. KILL MYSQL PRIMARY
2. KILL MYSQL REPLICA
3. KILL THE MYSQL PROXY

HOW DO WE MAKE MYSQL RESILIENT TO KILLS?

WE USE SEMI SYNC, GROUP REPLICATION AND WE CREATED A TOOL CALLED AUTO REPLACE TO DO CLONES AND PROMOTIONS.

CHAOS CREATES RESILIENCE

INJECT CHAOS IN YOUR SYSTEM

WHAT TYPES OF CHAOS DID YOU INJECT?
WHAT WAS YOUR HYPOTHESIS?

?

THE CHAOS BOOTCAMP

+ LAYING THE FOUNDATIONS (2:00 - 3:00)
+ **CHAOS ENGINEERING DISCUSSION (3:00 - 3:30)**
+ AFTERNOON BREAK (3:30 - 4:00)
+ DISTRIBUTED SYSTEMS CHAOS (4:00 - 4:30)
+ CHAOS ENGINEERING IN YOUR COMPANY (4:30 - 4:45)
+ CRAFT YOUR OWN EXPERIMENTS (4:45- 5:00)
+ FEEDBACK AND EVALUATION TECHNIQUES (5:00-5:15)
+ ADVANCED TOPICS & Q + A (5:15 - 5:30)

PART II: CHAOS DISCUSSION

THE CHAOS BOOTCAMP

CHAOS ENGINEERING DEBATE TIME

- FOUR VOLUNTEERS
- TWO TEAMS - 1 TEAM IS FOR, 1 TEAM IS AGAINST
- **TOPIC:** "EVERY COMPANY SHOULD BE DOING CHAOS ENGINEERING"
- EACH PERSON GETS A GO AND SPEAKS FOR 2 MINS MAX
- WE ALL VOTE ON A WINNER (APPLAUSE-O-METER)

THE CHAOS BOOTCAMP

+ LAYING THE FOUNDATIONS (2:00 - 3:00)
+ CHAOS ENGINEERING DISCUSSION (3:00 - 3:30)
+ **AFTERNOON BREAK (3:30 - 4:00)**
+ DISTRIBUTED SYSTEMS CHAOS (4:00 - 4:30)
+ CHAOS ENGINEERING IN YOUR COMPANY (4:30 - 4:45)
+ CRAFT YOUR OWN EXPERIMENTS (4:45- 5:00)
+ FEEDBACK AND EVALUATION TECHNIQUES (5:00-5:15)
+ ADVANCED TOPICS & Q + A (5:15 - 5:30)

30 MIN AFTERNOON BREAK

3:30 - 4:00

THE CHAOS BOOTCAMP

+ LAYING THE FOUNDATIONS (2:00 - 3:00)
+ CHAOS ENGINEERING DISCUSSION (3:00 - 3:30)
+ AFTERNOON BREAK (3:30 - 4:00)
+ **DISTRIBUTED SYSTEMS CHAOS (4:00 - 4:30)**
+ CHAOS ENGINEERING IN YOUR COMPANY (4:30 - 4:45)
+ CRAFT YOUR OWN EXPERIMENTS (4:45- 5:00)
+ FEEDBACK AND EVALUATION TECHNIQUES (5:00-5:15)
+ ADVANCED TOPICS & Q + A (5:15 - 5:30)

PART III: DISTRIBUTED SYSTEMS CHAOS

CHAOS MONKEY

YOU SET IT UP AS A CRON JOB THAT CALLS CHAOS MONKEY ONCE A WEEKDAY TO CREATE A SCHEDULE OF TERMINATIONS.

HAS BEEN AROUND FOR MANY YEARS! USED AT BANKS, E-COMMERCE STORES, TECH COMPANIES + MORE

Deb Bakshiyev Follow
May 9 · 6 min read

Chaos Monkey is a service that randomly terminates VM instances and containers–these frequent failures promote the creation of resilient services. Chaos Monkey 2.0 is tightly integrated with Spinnaker: it relies on the Spinnaker APIs to terminate instances, retrieves deployment information from Spinnaker, and is configured using the Spinnaker UI.

Here, I'll walk you through setting up and running Chaos Monkey on Google Compute Engine (GCE).

https://medium.com/continuous-delivery-scale/running-chaos-monkey-on-spinnaker-google-compute-engine-gce-155dc52f20ef

Chaos Monkey

Search docs

- Home
- Configuration file format
- Configuring behavior via Spinnaker
- Decryptor
- Error counter
- How to deploy
- Outage checker
- Running locally
- Running tests
- Termination behavior
- Tracker
- Vendoring dependencies

GitHub Next »

Docs » Home

Edit on GitHub

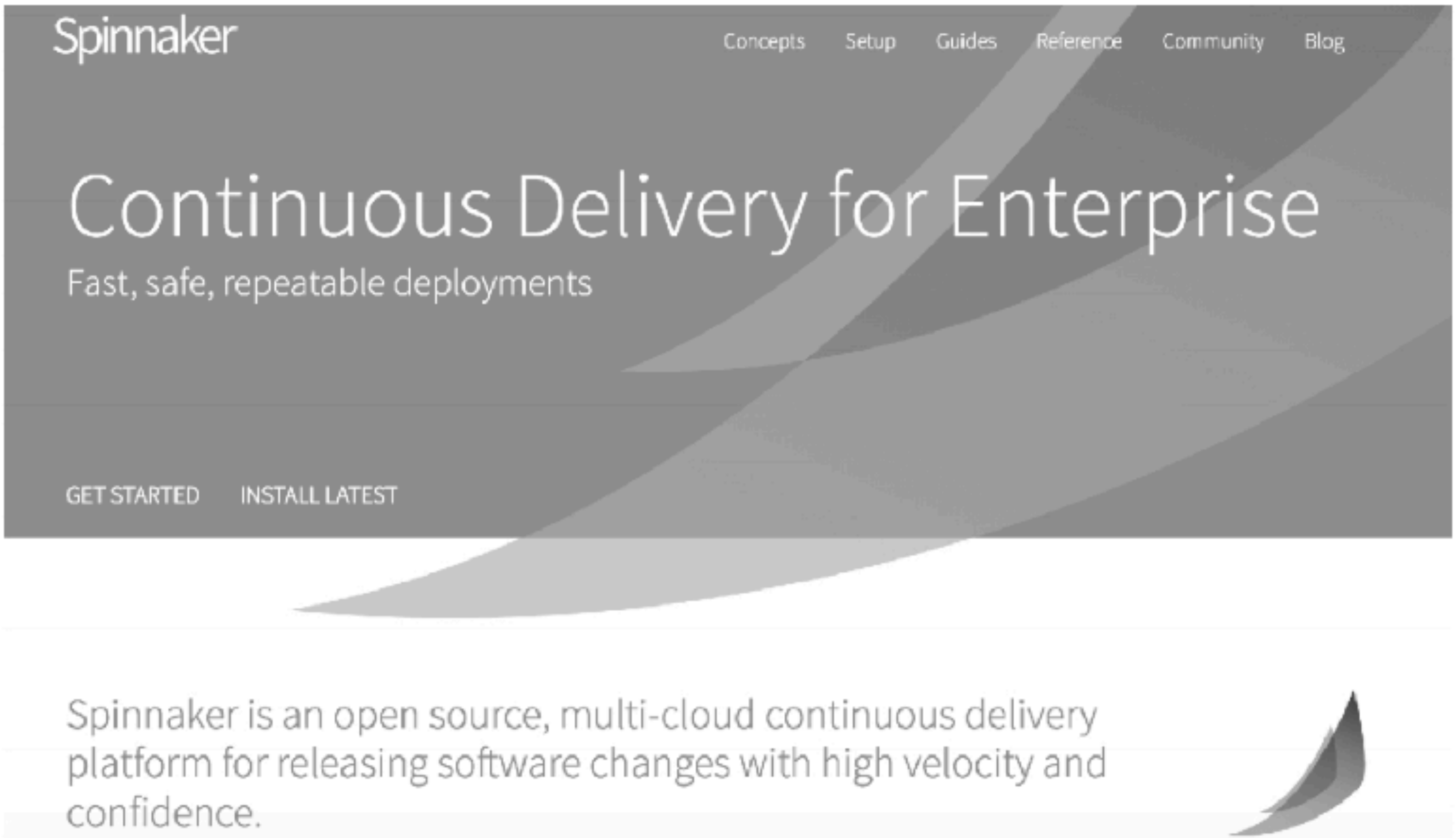
Chaos Monkey is responsible for randomly terminating instances in production to ensure that engineers implement their services to be resilient to instance failures.

See how to deploy for instructions on how to get up and running with Chaos Monkey.

Once you're up and running, see configuring behavior via Spinnaker for how users can customize the behavior of Chaos Monkey for their apps.

Next

https://netflix.github.io/chaosmonkey/



https://www.spinnaker.io/

CHAOS KONG

TAKES DOWN AN ENTIRE AWS REGION. NETFLIX CREATED IT BECAUSE AWS HAD NOT YET BUILT THE ABILITY TO TEST THIS.

AWS REGION OUTAGES DO HAPPEN!

CHAOS FOR KUBERNETES

ASOBTI, AN ENGINEER @ BOX CREATED
https://github.com/asobti/kube-monkey

IT RANDOMLY DELETES KUBERNETES PODS IN THE CLUSTER ENCOURAGING AND VALIDATING THE DEPLOYMENT OF FAILURE-RESILIENT SYSTEMS.

SIMIAN ARMY

A SUITE OF TOOLS FOR KEEPING YOUR CLOUD OPERATING IN TOP FORM. CHAOS MONKEY IS THE FIRST MEMBER. OTHER SIMIANS INCLUDE JANITOR MONKEY & CONFORMITY MONKEY.

https://github.com/Netflix/SimianArmy

GREMLIN INC

GREMLIN IS BUILDING A
CHAOS ENGINEERING PLATFORM.
FIRST COMPANY FOUNDED TO DO THIS.

RUN GREMLIN AGENTS ON YOUR
HOSTS OR IN CONTAINERS.
11 PRE-BUILT ATTACKS.
SCHEDULE ATTACKS WITH THE UI, API OR CLI.

GREMLIN.COM
@GREMLININC

LET’S GO BACK IN TIME TO LOOK AT WORST OUTAGE STORIES WHICH THEN LED TO THE INTRODUCTION OF CHAOS ENGINEERING.

CHAOS @ DROPBOX

DROPBOX'S WORST OUTAGE EVER

SOME MASTER-REPLICA PAIRS WERE IMPACTED WHICH RESULTED IN THE SITE GOING DOWN.

https://blogs.dropbox.com/tech/2014/01/outage-post-mortem/

CHAOS @ DROPBOX

1. CHAOS DAYS
2. RACK SHUTDOWN
3. SERVICE DRTs

QUICK THOUGHTS.....

+ SO MANY WORST OUTAGE STORIES ARE THE DATABASE.
+ I LEAD DATABASES AT DROPBOX & WE DO CHAOS.
+ FEAR WILL NOT HELP YOU SURVIVE "**THE WORST OUTAGE**".
+ DO YOU TEST YOUR ALERTS & MONITORING? WE DO.
+ HOW VALUABLE IS A POSTMORTEM IF YOU DON'T HAVE ACTION ITEMS AND DO THEM? NOT VERY.

UBER'S **WORST OUTAGE EVER:**

1. MASTER LOG REPLICATION TO S3 FAILED
2. LOGS BACKED UP ON PRIMARY
3. ALERTS FIRE TO ENGINEER BUT THEY ARE IGNORED
4. DISK FILLS UP ON DATABASE PRIMARY
5. ENGINEER DELETES UNARCHIVED WAL FILES
6. ERROR IN CONFIG PREVENTS PROMOTION

— Matt Ranney, UBER, YOW 2015

Scaling Uber with Matt Ranney

by Pranay | December 4, 2015 | in Cloud Engineering, Greatest Hits, Podcast | 0 |

00:00 00:00

Podcast: Play in new window | Download

"If you can make a system that can survive this random failure testing, then you will more or likely survive whatever other chaotic conditions exist."

Uber is a transportation and logistics company that manages many aspects of its ride-sharing services through mobile apps and distributed technology. Uber faces unique challenges in rapidly scaling its services internationally, and at one point increased its developer headcount from 200 to over 1000 in less than a year.

Matt Ranney is the Chief Systems Architect at Uber and was previously a founder and CTO of Voxer. At QCon San Francisco, he gave a talk called Scaling Uber.

CHAOS @ UBER

+ UBER BUILT UDESTROY TO SIMULATE FAILURES.
+ DIDN'T USE NETFLIX SIMIAN ARMY AS IT WAS AWS-CENTRIC.
+ ENGINEERS AT UBER DON'T LIKE FAILURE TESTING (ESP. DATABASES) ......THIS IS DUE TO THEIR **WORST OUTAGE EVER**:

— **Matt Ranney, UBER, YOW 2015**

CHAOS @ NETFLIX

SIMIAN ARMY CONSISTS OF SERVICES (MONKEYS) IN THE CLOUD FOR GENERATING VARIOUS KINDS OF FAILURES, DETECTING ABNORMAL CONDITIONS, AND TESTING THE ABILITY TO SURVIVE THEM. THE GOAL IS THE KEEP THE CLOUD SAFE, SECURE AND HIGHLY AVAILABLE.

+ **CHAOS MONKEY**
+ **JANITOR MONKEY**
+ **CONFORMITY MONKEY**

CHAOS @ GOOGLE

GOOGLE RUN DRTs AND HAVE BEEN FOR MANY YEARS

CHAOS @ TYPESAFE

"RESILIENCE HAS TO BE DESIGNED. HAS TO BE TESTED. IT'S NOT SOMETHING THAT HAPPENS AROUND A TABLE AS A SLEW OF EXCEPTIONAL ENGINEERS ARCHITECT THE PERFECT SYSTEM. PERFECTION COMES THROUGH REPEATEDLY TRYING TO BREAK THE SYSTEM"

— VICTOR KLANG, TYPESAFE

INTRODUCING CHAOS IN A CONTROLLED WAY WILL RESULT IN ENGINEERS BUILDING INCREASINGLY RESILIENT SYSTEMS.

HAVE I CONVINCED YOU?

BUILDKITE

DECIDED TO REDUCE DATABASE CAPACITY IN AWS. RESULTED IN AN OUTAGE AT 3:21AM. PAGERDUTY WAS MISCONFIGURED AND PHONES WERE ON SILENT.

NOBODY WOKE UP DURING THE 4 HOUR OUTAGE.....

STRIPE

"A DATABASE INDEX OPERATION RESULTED IN 90 MINUTES OF INCREASINGLY DEGRADED AVAILABILITY FOR THE STRIPE API AND DASHBOARD. IN AGGREGATE, ABOUT TWO THIRDS OF ALL API OPERATIONS FAILED DURING THIS WINDOW."

https://support.stripe.com/questions/outage-postmortem-2015-10-08-utc

OUTAGES HAPPEN.

THERE ARE MANY MORE YOU CAN READ ABOUT HERE:

https://github.com/danluu/post-mortems

PART IV: CHAOS ENGINEERING IN YOUR OWN COMPANY

PART V: CRAFT YOUR OWN CHAOS ENGINEERING EXPERIMENTS

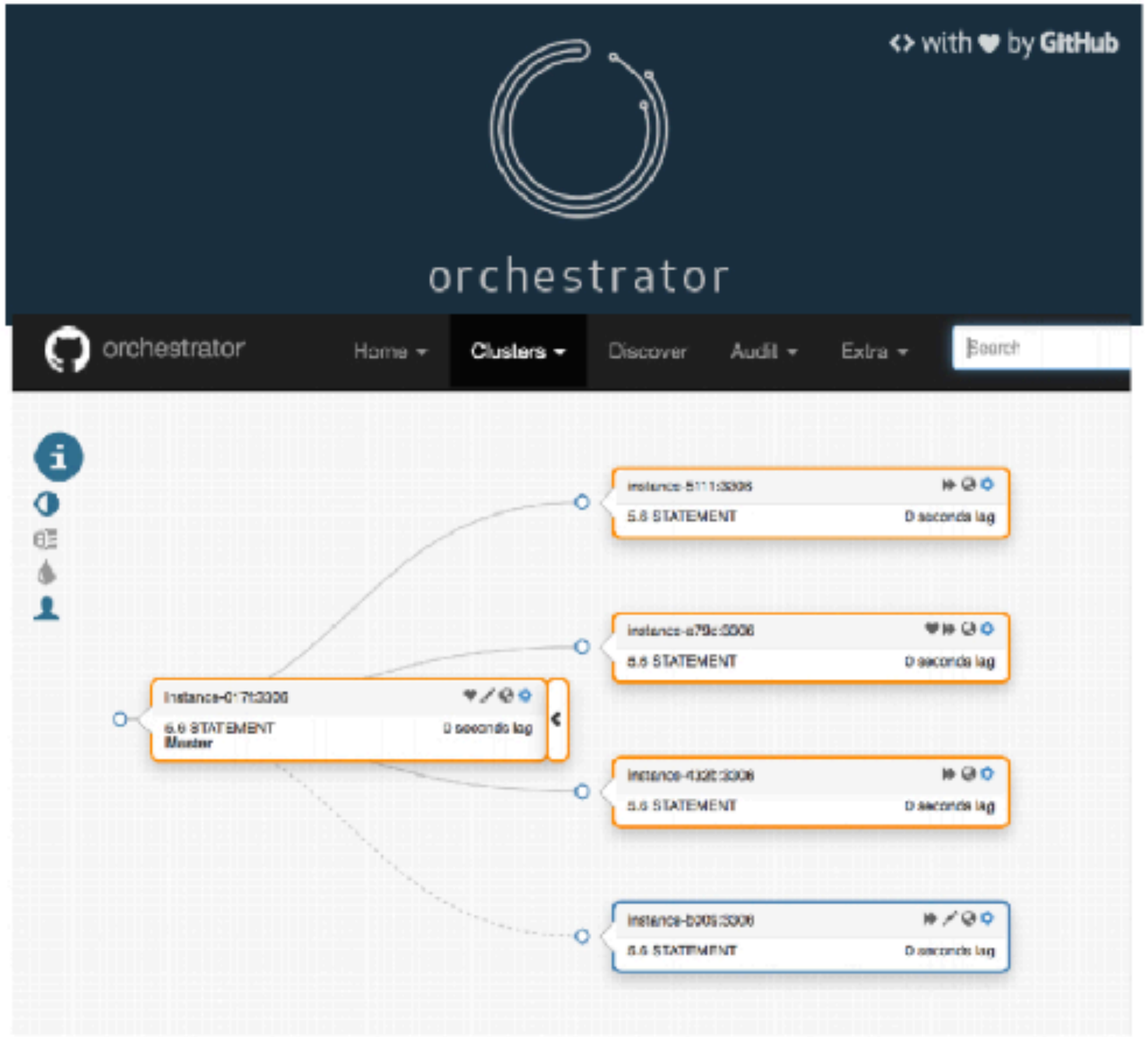
PART VI: FEEDBACK AND EVALUATION TECHNIQUES

PART VII: ADVANCED CHAOS + Q & A

CHAOS ENGINEERING FOR DATABASES

- GOOD TO USE:
  - MYSQL
  - ORCHESTRATOR
  - GROUP REPLICATION
  - SEMI SYNC

https://github.com/github/orchestrator



Authored by Shlomi Noach at GitHub. Previously at Booking.com and Outbrain

https://github.com/github/orchestrator

GO CLIENT TO THE CHAOS MONKEY REST API

THIS PROJECT WAS STARTED FOR THE PURPOSE OF CONTROLLED FAILURE INJECTION DURING GAME DAYS.

https://github.com/mlafeldt/chaosmonkey

```
go get -u github.com/mlafeldt/chaosmonkey/lib
```

Chaos Lemur

build passing

This project is a self-hostable application to randomly destroy virtual machines in a BOSH-managed environment, as an aid to resilience testing of high-availability systems. Its main features are:

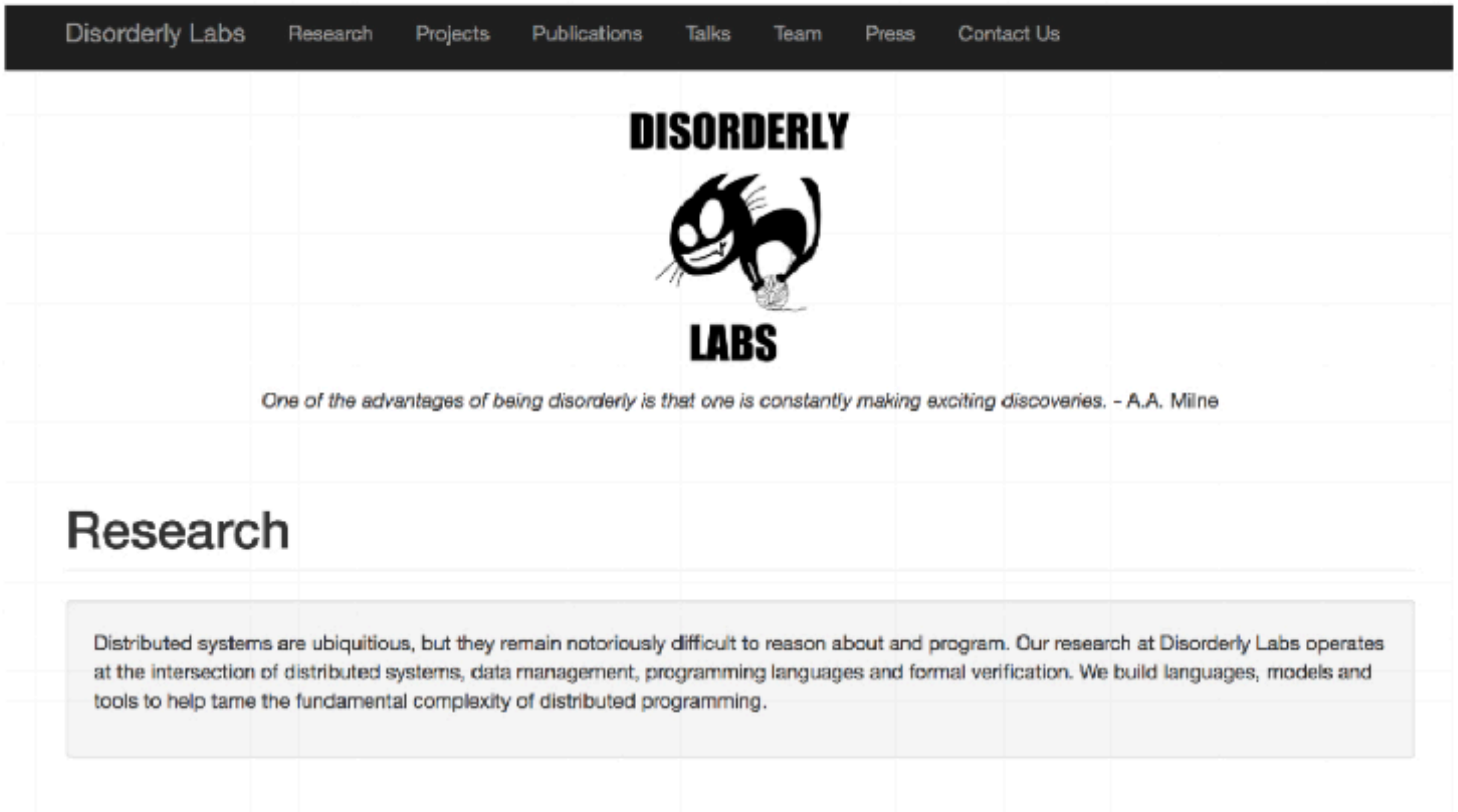
- Triggers on a user-defined schedule, selecting 0 or more VMs to destroy at random during each run.
- Manual triggering of unscheduled destroys.
- Per-deployment and per-job probabilities for destruction of member VMs.
- Optional blacklisting of deployments and jobs to protect their members from destruction.
- Runs against different types of IaaS (e.g. AWS, vSphere) using a small infrastructure API.
- Optionally records activities to DataDog.

Although Chaos Lemur recognizes deployments and jobs, it is not possible to select an entire deployment or job for destruction. Entire deployments and jobs will be destroyed over time by chance, given sufficient runs.

https://github.com/strepsirrhini-army/chaos-lemur

INDUSTRY + ACADEMIA COLLABORATION

DISORDERLY LABS



DISORDERLY LABS

Lineage-driven Fault Injection

Peter Alvaro
UC Berkeley
palvaro@cs.berkeley.edu

Joshua Rosen
UC Berkeley
rosenville@gmail.com

Joseph M. Hellerstein
UC Berkeley
hellerstein@cs.berkeley.edu

ABSTRACT

Failure is always an option; in large-scale data management systems, it is practically a certainty. Fault-tolerant protocols and components are notoriously difficult to implement and debug. Worse still, choosing existing fault-tolerance mechanisms and integrating them correctly into complex systems remains an art form, and programmers have few tools to assist them.

We propose a novel approach for discovering bugs in fault-tolerant data management systems: *lineage-driven fault injection*. A lineage-driven fault injector reasons *backwards* from correct system outcomes to determine whether failures in the execution could have prevented the outcome. We present MOLLY, a prototype of lineage-driven fault injection that exploits a novel combination of data lineage techniques from the database literature and state-of-the-art satisfiability testing. If fault-tolerance bugs exist for a particular configuration, MOLLY finds them rapidly, in many cases using an order of magnitude fewer executions than random fault injection. Otherwise, MOLLY certifies that the code is bug-free for that configuration.

enriching new system architectures with well-understood fault tolerance mechanisms and henceforth assuming that failures will not affect system outcomes. Unfortunately, fault-tolerance is a *global* property of entire systems, and guarantees about the behavior of individual components do not necessarily hold under composition. It is difficult to design and reason about the fault-tolerance of individual components, and often equally difficult to assemble a fault-tolerant system even when given fault-tolerant components, as witnessed by recent data management system failures [16, 57] and bugs [36, 49].

Top-down testing approaches—which perturb and observe the behavior of complex systems—are an attractive alternative to verification of individual components. Fault injection [1, 26, 35, 44, 59] is the dominant top-down approach in the software engineering and dependability communities. With minimal programmer investment, fault injection can quickly identify shallow bugs caused by a small number of independent faults. Unfortunately, fault injection is poorly suited to discovering rare counterexamples involving complex combinations of multiple instances and types of

https://people.ucsc.edu/~palvaro/molly.pdf

DISORDERLY LABS

Automating Failure Testing Research at Internet Scale

Peter Alvaro
UC Santa Cruz
palvaro@ucsc.edu

Kolton Andrus
Gremlin, Inc. (Formerly Netflix)
kolton@gremlininc.com

Chris Sanden Casey Rosenthal Ali Basiri Lorin Hochstein
Netflix, Inc.
csanden,crosenthal,abasiri,lhochstein @netflix.com

Abstract

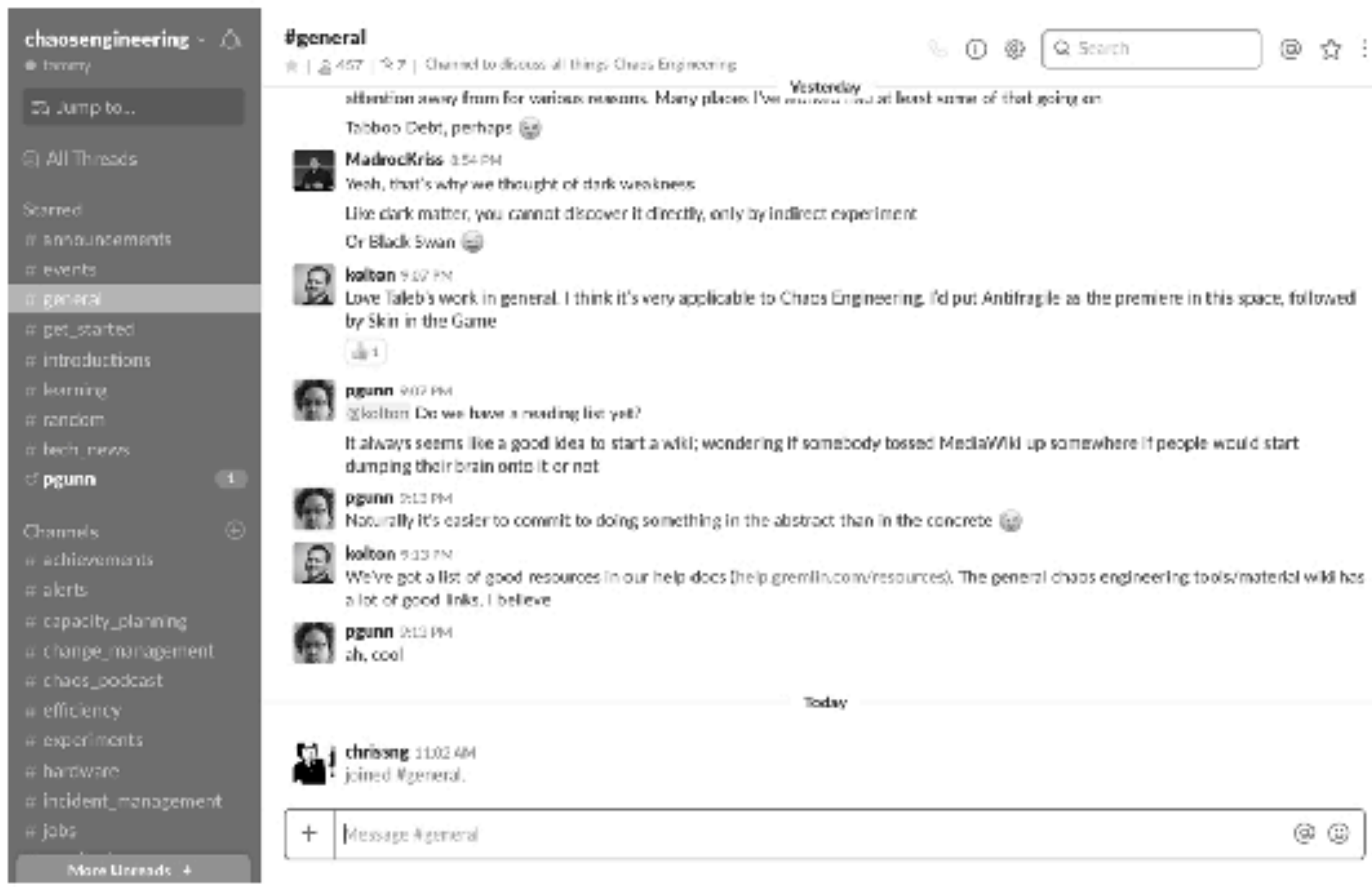
Large-scale distributed systems must be built to anticipate and mitigate a variety of hardware and software failures. In order to build confidence that fault-tolerant systems are correctly implemented, Netflix (and similar enterprises) regularly run *failure drills* in which faults are deliberately injected in their production system. The combinatorial space of failure scenarios is too large to explore exhaustively. Existing failure testing approaches either randomly explore the space of potential failures randomly or exploit the "hunches" of domain experts to guide the search. Random strategies waste resources testing "uninteresting" faults, while programmer-guided approaches are only as good as human

the rule. In order to provide an "always on" experience to customers, the software used by Internet companies must be he written to anticipate and work around a variety of error conditions, many of which are only present at large scale. It is difficult to ensure that such fault-tolerant code is adequately tested, because there are so many ways that a Internet-scale distributed system can fail.

Chaos Engineering [10], or "experimenting on a distributed system in order to build confidence in the system's capability to withstand turbulent conditions in production," has is emerging as a discipline to tackle resilience of these large-scale distributed systems [28, 35]. Engineers create frameworks that automate failure injection, usually on

https://people.ucsc.edu/~palvaro/socc16.pdf

HOW CAN YOU CONTINUE YOUR CHAOS ENGINEERING JOURNEY?



https://slofile.com/slack/chaosengineering

CONTINUE BREAKING THINGS
ON PURPOSE TOGETHER

THANKS FOR ATTENDING THE: CHAOS ENGINEERING BOOTCAMP

@TAMMYBUTOW