APPLYING PRINCIPLES OF CHAOS ENGINEERING to SERVERLESS...
history of Smallpox

earliest evidence of disease in 3rd Century BC Egyptian Mummy
est. 400K deaths per year in 18th Century Europe.
history of Smallpox

earliest evidence of disease in 3rd Century BC Egyptian Mummy
est. **400K deaths per year** in 18th Century Europe.

1798

first vaccine developed

Edward Jenner
history of Smallpox

earliest evidence of disease in 3rd Century BC Egyptian Mummy

est. **400K deaths per year** in 18th Century Europe.

WHO certified global eradication

1798 first vaccine developed

Edward Jenner

1980
Vaccination is the most effective method of preventing infectious diseases
stimulates the immune system to recognize and destroy the disease before contracting the disease for real
Chaos Engineering

controlled experiments to help us learn about our system’s behaviour and build confidence in its ability to withstand turbulent conditions
Yan Cui
http://theburningmonk.com
@theburningmonk

Principal Engineer @ DAZN
“Netflix for sports”
offices in London, Leeds, Katowice and Amsterdam
DAZN to screen Italian football’s top league

Streaming service’s deal for Serie A games is latest challenge to broadcasters
available in Austria, Switzerland, Germany, Japan, Canada and Italy

US coming soon ;(-)
UK-based sport streaming service adopts Netflix model after $1bn deal

DAZN aims to revolutionise US pay-per-view boxing after link-up with promoter Eddie Hearn

The Netflix and Amazon model that has revolutionised entertainment TV viewing is being brought to the world of sport, as a London-based streaming
available on 30+ platforms
~500,000 concurrent viewers
“Netflix for sports”
offices in London, Leeds, Katowice and Amsterdam
Off the top of your head, do you know the answer to these questions: what will happen to your application if Amazon S3—one of the most widely used Amazon Web Services—suddenly
Past Meetup

**The joys of destruction**

Hosted by Steve Ganly
From London SRE Meetup
Public group

You shared feedback on August 29. As more people give feedback, you'll see the highlights here.

You went 28 people went

Thursday, March 29, 2018
6:30 PM to 8:30 PM

Skills Matter | CodeNode
10 South Place, London, EC2M 2RB · London

How to find us
The lovely Skills Matter people will tell you which room we’re in on the night.
Why did you break production?
Because I can!
Kolton Andrus, CEO of Gremlin

Russ Miles, CEO of ChaosIQ

Nora Jones, Chaos Engineer at Netflix
Kolton Andrus, CEO of Gremlin

Russ Miles, CEO of ChaosIQ

Nora Jones, Chaos Engineer at Netflix
THE FLY

Cartoon-Box 20
it’s about building confidence, NOT breaking things
PRINCIPLES OF CHAOS ENGINEERING
Last Update: 2017 April

Chaos Engineering is the discipline of experimenting on a distributed system in order to build confidence in the system’s capability to withstand turbulent conditions in production.

Advances in large-scale, distributed software systems are changing the game for software engineering. As an industry, we are quick to adopt practices that increase flexibility of development and velocity of deployment. An urgent question follows on the heels of these benefits: How much confidence we can have in the complex systems that we put into production?

Even when all of the individual services in a distributed system are functioning properly, the interactions between those services can cause unpredictable outcomes. Unpredictable outcomes, compounded by rare but disruptive real-world events that affect production environments, make these distributed systems inherently chaotic.

We need to identify weaknesses before they manifest in system-wide, aberrant behaviors. Systemic weaknesses could take the form of: improper fallback settings when a service is unavailable; retry storms from improperly tuned timeouts; outages when a downstream dependency receives too much traffic; cascading failures when a single point of failure crashes; etc. We must address the most significant weaknesses proactively, before they affect our customers in production. We need a way to manage the chaos inherent in these systems, take advantage of increasing flexibility and velocity, and have confidence in our production deployments despite the complexity that they represent.

An empirical, systems-based approach addresses the chaos in distributed systems at scale and builds confidence in the ability of those systems to withstand realistic conditions. We learn about the behavior of a distributed system by observing it during a controlled experiment. We call this Chaos Engineering.

CHAOS IN PRACTICE

To specifically address the uncertainty of distributed systems at scale, Chaos Engineering can be thought of as the facilitation of experiments to uncover systemic weaknesses. These experiments follow four steps:

1. Start by defining ‘steady state’ as some measurable output of a system that indicates normal behavior.
2. Hypothesize that this steady state will continue in both the control group and the experimental group.
3. Introduce variables that reflect real world events like servers that crash, hard drives that malfunction, network connections that are severed, etc.
4. Try to disprove the hypothesis by looking for a difference in steady state between the control group and the experimental group.

The harder it is to disrupt the steady state, the more confidence we have in the behavior of the system. If a weakness is uncovered, we now have a target for improvement before that behavior manifests in the system at large.

http://principlesofchaos.org
define “Steady State”
aka. what does normal, working condition looks like?
this is not a steady state
STEP 2.

hypothesize steady state will continue in both control group & the experiment group

ie. you should have a reasonable degree of confidence the system would handle the failure before you proceed with the experiment
explore unknown unknowns
away from production
treat production with the care it deserves
the goal is **NOT**, to actually hurt production
If you know the system would break, and you did it anyway…
then it’s NOT a chaos experiment.

It’s called being IRRESPONSIBLE.
STEP 3.

**inject realistic failures**

e.g. server crash, network error, HD malfunction, etc.
Turn failure into resilience.

Gremlin provides you the framework to safely, securely, and easily simulate real outages with an ever-growing library of attacks.

REQUEST A DEMO
STEP 4.

disprove hypothesis

i.e. look for difference with steady state
if a **WEAKNESS** is uncovered, **IMPROVE** it before the behaviour manifests in the system at large
Chaos Engineering: controlled experiments to help us learn about our system’s behaviour and build confidence in its ability to withstand turbulent conditions.
Chaos Engineering

controlled experiments to help us **learn** about our system’s behaviour and **build confidence** in its ability to withstand turbulent conditions.
ensure everyone knows what you’re doing
ensure everyone knows what you’re doing

NO surprises!
Communication

Timing
run experiments during office hours
AVOID important dates
COMMUNICATION

TIMING

CONTAIN BLAST RADIUS
smallest change that allows you to detect a signal that steady state is disrupted
rollback at the first sign of TROUBLE!
COMMUNICATION
TIMING
CONTAIN BLAST RADIUS
don’t try to run before you know how to walk.
**chaos monkey** kills an EC2 instance

**chaos gorilla** kills an AWS Availability Zone

**chaos kong** kills an entire AWS region
Serverless chaos monkey for AWS (runs on AWS Lambda)

reliability-engineering  aws  fault-tolerance  chaos-monkey

58 commits  1 branch  6 releases  3 contributors

Branch: master

hassy Merge pull request #19 from davinerd/tags-support

- bin
  - Rename files for the new name
  - 11 months ago

- lambda
  - added support for tags
  - 4 months ago

- lib/commands
  - fix: Resolve module path correctly
  - 11 months ago

- .gitignore
  - Updating .gitignore
  - 2 years ago

- CONTRIBUTING.md
  - Rename Chaos Lambda to Chaos Llama
  - 11 months ago

- Chaosfile.json
  - added support for tags
  - 4 months ago

- LICENSE.txt
  - All basics in place
  - 2 years ago

- README.md
  - doc: Update link to website in README
  - 8 months ago

- package.json
  - 2.0.2
  - 11 months ago
there is no server...
there is no server...

that you can kill
“We need to identify weaknesses before they manifest in system-wide, aberrant behaviors. Systemic weaknesses could take the form of: improper fallback settings when a service is unavailable; retry storms from improperly tuned timeouts; outages when a downstream dependency receives too much traffic; cascading failures when a single point of failure crashes; etc. We must address the most significant weaknesses proactively, before they affect our customers in production. **We need a way to manage the chaos inherent in these systems**, take advantage of increasing flexibility and velocity, and have confidence in our production deployments despite the complexity that they represent.”

— Principles of Chaos Engineering
there are more inherent chaos and complexity in a Serverless architecture
smaller units of deployment
but **A LOT** more of them!
more difficult to harden around boundaries
more intermediary services, and greater variety too
more intermediary services, and greater variety too

each with its own set of failure modes
more configurations,
more opportunities for misconfiguration
more unknown failure modes in infrastructure that we don’t control
often there's little we can do when an outage occurs in the platform
improperly tuned timeouts
missing error handling
missing fallback when downstream is unavailable
define “Steady State”
aka. what does normal, working condition looks like?
what metrics do you monitor?
9X-percentile latency ✔
error count ✔
yield (% of requests completed) ✔
harvest (completeness of results) ✔
STEP 2.

hypothesize steady state will continue in both control group & the experiment group

ie. you should have a reasonable degree of confidence the system would handle the failure before you proceed with the experiment
## API Gateway

<table>
<thead>
<tr>
<th>Feature</th>
<th>Description</th>
<th>Limitation</th>
</tr>
</thead>
<tbody>
<tr>
<td>API caching TTL</td>
<td>300 seconds by default and configurable between 0 and 3600 by an API owner.</td>
<td>Not for the upper bound (3600)</td>
</tr>
<tr>
<td>Integration timeout</td>
<td>29 seconds for all integration types, including Lambda, Lambda proxy, HTTP, HTTP proxy, and AWS integrations.</td>
<td>No</td>
</tr>
<tr>
<td>Header value size</td>
<td>10240 Bytes</td>
<td>No</td>
</tr>
<tr>
<td>Payload size</td>
<td>10 MB</td>
<td>No</td>
</tr>
</tbody>
</table>
consider the effect of cold-starts & API Gateway overhead
use short timeout for API calls
the goal of a timeout strategy is to give HTTP requests the **best chance to succeed**, provided that doing so does not cause the calling function itself to err.
fixed timeout are tricky to get right...
fixed timeout are tricky to get right...

too short and you don’t give requests the best chance to succeed
fixed timeout are tricky to get right...

too long and you run the risk of letting the request timeout the calling function
and it gets worse when you make multiple API calls in one function...
**approach 1: split invocation time equally**

(e.g. 3 requests, 6s function timeout = 2s timeout per request)

request 1

2s

request 2

2s

request 3

2s

function invocation

6s

request is times out even though there are sufficient time in the invocation to perform all 3 requests
approach 2: every request is given nearly ALL the invocation time (free for all)
(e.g. 6s function timeout = 5s timeout per request)

---

function invocation

function times out, wasn’t given a chance to perform recovery steps
set the request timeout based on the amount of invocation time left
The Context Object Methods (Node.js)

The context object provides the following methods.

`context.getRemainingTimeInMillis()`

Returns the approximate remaining execution time (before timeout occurs) of the Lambda function that is currently executing. The timeout is one of the Lambda function configuration. When the timeout reaches, AWS Lambda terminates your Lambda function.

You can use this method to check the remaining time during your function execution and take appropriate corrective action at run time.

The general syntax is:

```
context.getRemainingTimeInMillis();
```
set timeout based on remaining invocation time

request 1

request 2

request 3

reserved time for performing recovery actions

timeout is set based on the remaining invocation time, minus recovery time

function invocation

6s
set timeout based on remaining invocation time

request 1

request 2

request 3

reserved time for performing recovery actions

request is timed out, and recovery action is carried out

timeout is set based on the remaining invocation time, minus recovery time

function invocation

6s
recovery
log the timeout incident with as much context as possible
e.g. timeout value, correlation IDs, request object, ...
report custom metrics
NETFLIX ORIGINAL

RICKY GERVAIS
HUMANITY

Play
My List

Continue Watching for Yan

Benji
NEW EPISODE WEEKLY

BLACK LIGHTNING

DEADPOOL

Dirk Gently's
OUTSIDER
TOTAL CHECKING (...6795)

Available balance
$0.00

see activity

Transfer Money
be mindful when you sacrifice precision for availability, **user experience is the king**
STEP 3.

inject realistic failures

e.g. server crash, network error, HD malfunction, etc.
where to inject latency?
hypothesis:

function has appropriate timeout on its HTTP communications and can degrade gracefully when these requests time out
should also be applied to 3rd parties services we depend on, e.g. DynamoDB
what’s the blast radius?
hypothesis:

all functions have appropriate timeout on their HTTP communications to this internal API, and can degrade gracefully when requests are timed out.
large blast radius, risky..
could be effective when used away from production environment, to weed out weaknesses quickly
not priming developers to build more resilient systems
development
Priming (psychology):

Priming is a technique whereby exposure to one stimulus influences a response to a subsequent stimulus, without conscious guidance or intention.

It is a technique in psychology used to train a person's memory both in positive and negative ways.
Facebook's '2G Tuesdays' simulate super slow internet in the developing world

By Rich McCormick | Oct 28, 2015, 5:58am EDT

In an attempt to understand the millions of people in emerging markets who only have access to slow internet connections, Facebook is adopting a new opt-in initiative called 2G.
make dev environments better resemble the turbulent conditions you should realistically expect your system to survive in production
hypothesis:

the client app has appropriate timeout on their HTTP communication with the server, and can degrade gracefully when requests are timed out
STEP 4.

disprove hypothesis

i.e. look for difference with steady state
how to inject latency?
static weaver (e.g. AspectJ, PostSharp), or dynamic proxies
manually crafted wrapper library
const co = require('co');
const Promise = require('bluebird');

let injectLatency = co.wrap(function*(config) {
  if (config.isEnabled === true && Math.random() < config.probability) {
    let delayRange = config.maxDelay - config.minDelay;
    let delay = Math.floor(config.minDelay + Math.random() * delayRange);

    console.log(`injecting [${delay}ms] latency to HTTP request...`);
    yield Promise.delay(delay);
  }
});
const co = require('co');
const Promise = require('bluebird');

let injectLatency = co.wrap(function*(config) {
    if (config.isEnabled === true && Math.random() < config.probability) {
        let delayRange = config.maxDelay - config.minDelay;
        let delay = Math.floor(config.minDelay + Math.random() * delayRange);

        console.log(`injecting [{delay}ms] latency to HTTP request...`);
        yield Promise.delay(delay);
    }
});
const co = require('co');
const Promise = require('bluebird');

let injectLatency = co.wrap(function* (config) {
    if (config.isEnabled === true && Math.random() < config.probability) {
        let delayRange = config.maxDelay - config.minDelay;
        let delay = Math.floor(config.minDelay + Math.random() * delayRange);

        console.log(`injecting [${delay}ms] latency to HTTP request...`);
        yield Promise.delay(delay);
    }
});
let Req = function* (options) {

    let request = getRequest(options);
    let fullResponse = options.resolveWithFullResponse === true;

    let latencyInjectionConfig = options.latencyInjectionConfig;
    yield injectLatency(latencyInjectionConfig);  // <- this is the import bit

    try {
        let resp = yield exec(request);
        return fullResponse ? resp : resp.body;
    } catch (e) {
        if (e.response && e.response.error) {
            throw e.response.error;
        }

        throw e;
    }

};
let Req = function* (options) {
  let request = getRequest(options);
  let fullResponse = options.resolveWithFullResponse === true;

  let latencyInjectionConfig = options.latencyInjectionConfig;
  yield injectLatency(latencyInjectionConfig); // <- this is the import bit

  try {
    let resp = yield exec(request);
    return fullResponse ? resp : resp.body;
  } catch (e) {
    if (e.response && e.response.error) {
      throw e.response.error;
    }

    throw e;
  }
};
configured in SSM Parameter Store

```
{
    "internalApi": "https://xx.amazonaws.com/dev/internal",
    "chaosConfig": {
        "httpClientLatencyInjectionConfig": {
            "isEnabled": true,
            "probability": 0.5,
            "minDelay": 100,
            "maxDelay": 5000
        }
    }
}
```
{  
"internalApi": "https://xx.amazonaws.com/dev/i",

"chaosConfig": {  

"httpClientLatencyInjectionConfig": { 

"isDisabled": true, 

"probability": 0.5, 

"minDelay": 100, 

"maxDelay": 5000  

}  

}  

}
no injected latency
<table>
<thead>
<tr>
<th>Method</th>
<th>Response</th>
<th>Duration</th>
<th>Age</th>
<th>ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>--</td>
<td>200</td>
<td>81.0 ms</td>
<td>8.6 sec (2017-11-04 15:54:25 UTC)</td>
<td>1-59fde2b1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Name</th>
<th>Res.</th>
<th>Duration</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>latency-injection-demo-dev-public-api-a</td>
<td>200</td>
<td>81.0 ms</td>
<td>✓</td>
</tr>
<tr>
<td>latency-injection-demo-dev-public-api-a</td>
<td>200</td>
<td>74.0 ms</td>
<td>✓</td>
</tr>
<tr>
<td>HTTP client</td>
<td>-</td>
<td>59.0 ms</td>
<td>✓</td>
</tr>
<tr>
<td>## exec request</td>
<td>-</td>
<td>58.0 ms</td>
<td>✓</td>
</tr>
</tbody>
</table>
with injected latency

<table>
<thead>
<tr>
<th>Name</th>
<th>Res.</th>
<th>Duration</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>latency-injection-demo-dev-public-api-a</td>
<td>200</td>
<td>3.5 sec</td>
<td>✔</td>
</tr>
<tr>
<td>latency-injection-demo-dev-public-api-a</td>
<td>-</td>
<td>3.5 sec</td>
<td>✔</td>
</tr>
<tr>
<td>HTTP client</td>
<td>-</td>
<td>3.5 sec</td>
<td>✔</td>
</tr>
<tr>
<td>## latency injection</td>
<td>-</td>
<td>3.4 sec</td>
<td>✔</td>
</tr>
<tr>
<td>## exec request</td>
<td>-</td>
<td>63.0 ms</td>
<td>✔</td>
</tr>
<tr>
<td>Name</td>
<td>Res.</td>
<td>Duration</td>
<td>Status</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>------</td>
<td>----------</td>
<td>--------</td>
</tr>
<tr>
<td>latency-injection-demo-dev-public-api-a</td>
<td>200</td>
<td>3.5 sec</td>
<td></td>
</tr>
<tr>
<td>HTTP client</td>
<td>-</td>
<td>3.5 sec</td>
<td></td>
</tr>
<tr>
<td>### latency injection</td>
<td>-</td>
<td>3.4 sec</td>
<td></td>
</tr>
<tr>
<td>### exec request</td>
<td>-</td>
<td>63.0 ms</td>
<td></td>
</tr>
</tbody>
</table>
factory wrapper function
(think bluebird’s promisifyAll function)
const co = require('co');
const apiHandler = require('./lib/apiHandler');
const injectable = require('./lib/injectable');
const Promise = require('bluebird');
const AWS = require('aws-sdk');
const asyncDDB = Promise.promisifyAll(new AWS.DynamoDB.DocumentClient());
const dynamodb = injectable.injectableAll(asyncDDB);
let req = {
    TableName: 'latency-injection-demo-dev',
    Key: { id: 'foo' }
};

// the wrapped (and promisified) DocumentClient's async functions can
// take an optional argument (the last argument) which is expect to be
// the format of the latencyInjectionConfig, i.e. of the shape
// { isEnabled, probability, minDelay, maxDelay }
let item = yield dynamodb.getAsync(req, latencyInjectionConfig);
2017-11-04T22:00:56.715Z a1ca570a-c1ab-11e7-9c5a-7fad5899050f AWS_XRAY_DAEMON_ADDRESS is

2017-11-04T22:00:56.719Z a1ca570a-c1ab-11e7-9c5a-7fad5899050f AWS_XRAY_CONTEXT_MISSING is

2017-11-04T22:00:56.731Z a1ca570a-c1ab-11e7-9c5a-7fad5899050f Subsegment streaming threshold set

2017-11-04T22:00:56 START RequestId: a1ca570a-c1ab-11e7-9c5a-7fad5899050f Version: $LATEST

2017-11-04T22:00:56.768Z a1ca570a-c1ab-11e7-9c5a-7fad5899050f {"resource":"/public-b","path":"/public-b"}

2017-11-04T22:00:56.768Z a1ca570a-c1ab-11e7-9c5a-7fad5899050floading cache keys: public-api-b.com

2017-11-04T22:00:56.875Z a1ca570a-c1ab-11e7-9c5a-7fad5899050f successfully loaded cache keys: public-api-b.com

2017-11-04T22:00:56.920Z a1ca570a-c1ab-11e7-9c5a-7fad5899050f injecting [1366ms] latency to wrapped function...

2017-11-04T22:00:56.920Z a1ca570a-c1ab-11e7-9c5a-7fad5899050f injecting [1366ms] latency to wrapped function...

2017-11-04T22:00:58.294Z a1ca570a-c1ab-11e7-9c5a-7fad5899050f SUCCESS {"message":"everything"

2017-11-04T22:00:58 END RequestId: a1ca570a-c1ab-11e7-9c5a-7fad5899050f

2017-11-04T22:00:58 REPORT RequestId: a1ca570a-c1ab-11e7-9c5a-7fad5899050f Duration: 1534.60 ms Billed Duration: 1600
injected [1366ms] latency to wrapped function...
Demo for how to apply latency injection to Lambda functions

Add topics

6 commits 1 branch 0 releases 1 contributor MIT

Branch: master

Latest commit ea92e33 on 4 Nov 2017

- added injectable helper for AWSSDK, etc.
- it works!
- init commit
- initial commit
- init commit
ERROR INJECTION
failures are INEVITABLE
the only way to truly know your system’s resilience against failures is to test it through **controlled** experiments
WHAT DOESN’T KILL YOU, MAKES YOU STRONGER.
WHAT DOESN'T KILL YOU MAKES YOU STRONGER EXCEPT BEARS

BEARS WILL KILL YOU
vaccinate your serverless architecture against failures
Yan Cui
http://theburningmonkey.com
@theburningmonkey
We’re hiring! Visit engineering.dazn.com to learn more.

“Netflix for sports” offices in London, Leeds, Katowice and Amsterdam

follow @DAZN_ngnrs for updates about the engineering team.