Chaos Engineering: Automating Chaos

Dimitra Zuccarelli, 20072495

February 2019

Contents

1 Introduction

The aim of this report is to explore the principles and application of a fairly new discipline, Chaos Engineering, pioneered by giants of the industry, Netflix, Google and AWS. In the practical section of this report, an Chaos tool called Chaos Toolkit will be used to conduct small Chaos experiments within a Kubernetes Cluster hosted on Google Cloud Platform.

2 Principles of Chaos Engineering

Chaos Engineering is defined in Principles of Chaos as: `"The discipline of experimenting on a distributed system in order to build confidence in the system's capability to withstand turbulent conditions in production"` [4]. Another, more simplified definition by Adrian Cockcroft defines it as: `"An experiment to ensure that the impact of failure is mitigated"` [1].

As the literature on Chaos Engineering commonly tends to point out, entropy is an inherent characteristic of any complex system. Each service may be optimised for the *local context* that the engineering team may possess, together with the help of extensive code coverage and testing. This however does not prevent failure from occurring *globally*, across multiple different services and multiple different factors [1]. Additionally, in most cases the tests in place (unit, functional, integration, etc) are mainly testing for known parameters and known outcomes. Chaos Engineering attempts to look at *stochastic* parameters to unmask unknown properties of the system.

Chaos Engineering is based on the principal of attempting to find the weaknesses in a system *that already exist*, in order to mitigate against potential future disasters. It is impossible to come across every factor that might go wrong in a complex system; in these cases, the idea is not necessarily to predict the disaster, but to detect the failure and act upon it in the smallest amount of time, minimising outage duration in order to remain under the company's downtime ”budget”. As Adrian Cockcroft, VP of Engineering at AWS, explains, Chaos Engineering is a tool to help restrict the impact of disaster, instead of attempting to mitigate against it entirely [1]. Chaos Engineering is a useful tool for Engineering teams to practice disaster response and recovery, become more confident in their system's reliability and the impact that any future failure may have [3].

2.1 How does Chaos Engineering differ from testing?

Chaos Engineering differs from traditional testing (unit, functional, integration, etc) in the fact that traditional testing attempts to prove a given assertion, based on outcomes and possibilities that a developer expects to happen. In that sense, traditional testing does not provide much extra context or information, apart from whether a unit is working as expected or not. Conversely, Chaos Engineering tests are often referred to as `chaos experiments`, since experimentation implies that *new* knowledge is being gained from conducting the tests [2]. Examples of chaos tests might include injecting failure into a system (for example, simulating a server malfunction in the data centre, or deleting partitions from a Kafka topic), but also simulating unpredictable traffic flow, network latency, etc in order to see how the system reacts under these circumstances [1].

2.2 The principles of Chaos Engineering

There are 5 principles published by Netflix in `principlesofchaos.org` and the book `Chaos Engineering` which encapsulate the concepts behind Chaos Engineering. These will be briefly summarised below.

2.2.1 Build a Hypothesis around Steady State Behaviour

Establishing a steady state behaviour means establishing what the system looks like at its normal, stable working state. Various system, health and business metrics can be used to find and troubleshoot performance issues:

- **System Metrics**: This includes metrics about resource usage such as the CPU load, memory usage, overall time taken to respond to web requests or to query the database, if the response is successful, etc.

- **Business Metrics**: This might include metrics about how the user interacts with the application and customer satisfaction, if the business is losing customers, etc.

Once the definition of "normal" is defined, deviations from what is normal must also be defined, e.g. what amount of deviation is acceptable/tolerable in order to still be considered "normal". After this, a hypothesis can be formulated about what will happen to the steady state of the system once a chaos experiment is performed. Netflix often performs chaos experiments on their production environment that involve the use of their `Simian Army`. With `Chaos Gorilla` for example, an entire region failure can be simulated. The hypothesis in this experiment is that the system continues to function according to its steady state, despite the entire AWS region not being available [3].

2.2.2 Vary Real-world Events

Systems are vulnerable to a large number of unpredictable events, regardless of their complexity. "Injecting" chaos into a system through Chaos Engineering is meant to mimic these real-world events, in order to understand how to restrict their impact (if this impact is negative) [3].

A common strategy for deciding which chaos experiments to run would be to prioritise them by the frequency and impact of the events occurring, and by the cost to the company of the incidents occurring. In his talk at AWS Re:Invent, Adrian Cockcroft notes that many companies and organisations have backup data centres in the occurrence of a malfunction in their primary data centre. However, very few companies test failover of an application to the backup data centre more than once a year. Almost none of these companies test failover of the entire datacentre to the backup data centre. The result is an untested, unreliable failover mechanism, which makes response time slow and does not provide much value to the company overall in the occurrence of disaster [1].

Many other events of this nature can looked at, from hardware failures, software failures like malformed responses, partial loss of messages in a data queue, spikes in traffic, network latency, and many more. Any of these events have potential to completely disrupt the steady state of the system and are therefore good candidates for chaos experiments [3, 1]. As stated previously, Netflix uses tools like `Chaos Monkey` and `Chaos Gorilla` to test failure of single EC2 instances and of entire regions or even availability zones `Chaos Kong` . While simulating failure of single instances is cheap and simple to do, simulating failures of entire regions is much more complex and much more costly. The upfront costs of running these experiments, however, in many cases seem to be largely beneficial. In fact, ensuring the resiliency of a system may potentially save a company millions when an outage occurs that the company is in a position to manage.

In summary, varying events means inducing different types of real world events and determining whether or not the system is capable of dealing with them.

2.2.3 Run Experiments in Production

Chaos Engineering has a different approach to traditional software testing processes. The aim of chaos experiments is to reveal problems in the *production* environment. Running experiments in production gives an overall snapshot of the behaviour of the system as well as user behaviour, which is often unpredictable. Real user behaviour, state and input is hard to simulate authentically, so the best way to guarantee authenticity is to use the actual information: `"The only way to truly build confidence in the system at hand is to experiment with the actual input received by the production environment"` [3].

This principle is one of the more self explanatory, but also one of the more controversial. This is due to the potential impact that performing these tests might have on the system, and the repercussions (cost, customer distrust) on the business. Chaos Engineers argue that systems are vulnerable to external factors regardless, and running these experiments in production is the best way to guarantee resiliency of the system. Uncovering potential weaknesses means being able to prevent them from occurring again [3]. Additionally, there are ways to manage unwanted side effects, such as minimising the impact that an experiment might cause [1]. This will be fleshed out in more detail in the final principle ("Minimise blast radius").

2.2.4 Automate Experiments to Run Continuously

Most talks on Chaos Engineering recommend a slow start, ease in approach when first adopting chaos experiments. Experiments can be started off initially in dev, staging and QA environments, notifying the whole team of when an experiment is going to occur, timing experiments, and using an 'opt-in' approach (e.g. only starting with teams and services who are not apprehensive, but excited by the prospect) [2].

By slowly adopting the approach, a growing level of confidence in the system will be gained and slowly more teams and services will opt-in to the idea of Chaos Engineering. Additionally, once experiments are successfully passing in production, they should be automated to run periodically, without prior knowledge to the duration or time at which they are running. This ensures an even greater level of reliability [3].

2.2.5 Minimise Blast Radius

Running these experiments in production could lead to unwanted side effects and bad customer experience. This is to some extent necessary in the short term, but in general the impact of these experiments should be contained as much as possible [3].

In the next section, a tool called Chaos Toolkit will be explored to automate some basic chaos experiments.

3 Practical: Automating Chaos Engineering with Chaos Toolkit

3.1 Chaos Toolkit

Chaos Toolkit is an extensible framework for implementing and automating Chaos experiments using a declarative syntax in JSON. It provides extensions for Cloud Providers such as Kubernetes, AWS, Google Cloud and Spring Boot for easily integrating tests within an existing infrastructure and conducting experiments with very little barrier to entry. The Kubernetes extension along with a 3 node GKE cluster will be explored in the practical section of this report, as it ties in to my Final Year Project architecture.

3.1.1 Installation and structure

Installing Chaos Toolkit is straightforward, and can be done using the `pip` package manager:

```
# Create a python virtual environment
python3 -m venv ~/.venvs/chaostk

# Activate the virtual environment
source ~/.venvs/chaostk/bin/activate

# Install Chaos Toolkit
(chaostk) $ pip install chaostoolkit
```

As stated previously, Chaos Toolkit can be extended by installing a variety of different plugins. The `chaosk8s` plugin needs be installed to allow simpler definitions for conducting Chaos experiments in Kubernetes:

```
# Install the chaosk8s extension
pip install chaostoolkit-kubernetes
```

A generic Chaos Toolkit experiment is defined declaratively using an open, language agnostic JSON format and contains four main sections: a description, a steady state, a method and rollbacks. To illustrate these sections, an example JSON file from `ChaosIQ` is used, which checks whether or not a file still exists after a re-deployment of the file has occurred.

1. **A description of the experiment**. This initial section aims to provide context to the overall experiment being performed. It will typically contain a title for the experiment (commonly expressed with a question), a definition, and a set of optional tags which can help categorise experiments. The question expressed in the title will be answered based on the outcome of the chaos test. An example of the description section is given below.

```
{
    "version": "1.0.0",
    "title": "Does the configuration file remain after deploying an update?",
    "description": "We expect the configuration file of our application to keep
        existing when updating it",
    "tags": [
        "io"
    ],
```

2. **A steady state hypothesis**. The steady state hypothesis section defines what the normal state of the system should look like. This might include probes that poll endpoints to make sure they are alive and healthy, for example. The steady state hypothesis is used by Chaos Toolkit to establish what normal is *before* conducting the experiment, and checking if that state is still in tact *after* the experiment is complete. An example can be found below.

```
    "steady-state-hypothesis": {
        "title": "File does exist",
        "probes": [
            {
                "type": "probe",
                "name": "file-found-where-it-should-be",
                "tolerance": true,
                "provider": {
                    "type": "python",
                    "module": "os.path",
                    "func": "exists",
                    "arguments": {
                        "path": "/tmp/somewhere.txt"
                    }
                }
            }
        ]
    },
```

3. **Begin the experiment**. The method section contains the actual chaos experiment that will be conducted, according to a list of instructions, or actions, that will be executed against the system. In this case, the experiment rolls out a new update of the application. The expected outcome of the experiment is that an application deployment does not affect the existence of the configuration file from the steady state hypothesis.

```
    "method": [
      {
            "type": "action",
            "name": "rollout-application-update",
            "provider": {
                "type": "process",
                "path": "echo",
                "arguments": "'updated'"
            }
        }
    ],
```

4. **Rollback the system**. Rollbacks are not exactly perfect rollbacks in the context of Chaos Engineering, since it is impossible to know the exact impact of the test. In this case, however, they can be used to revert the state of the system to what it was before the initial test. In this case no rollbacks are specified.

```
    "rollbacks": []
```

3.2 Experiment 1: Unavailable Service

3.2.1 Introduction

The experiments will aim to look at potential failure scenarios within a microservices codebase. The microservices used in this experiment were written in Node.js using the Express framework, and contain intentionally badly written code for illustration purposes. There are two microservices: a provider, which outputs an inventory of items in the catalogue `/list` endpoint, and a consumer microservice, which takes the list of items from the provider endpoint and outputs a message to the user. The services are deployed to a local Kubernetes cluster with the Dockerfile and deployment files from Appendix A, B and C. The consumer service is calling the provider service through an HTTP GET request to the provider's Kubernetes internal service URL (`http://provider-service:8080/weather`). The code for both services is provided below.

Listing 1: Provider code

```
const PORT = 8080;
const HOST = '0.0.0.0';

app.get('/', (req, res) => {
    res.send("Hello from provider");
});

app.get('/list', (req, res) => {
    let catalogue = [{
        "name": "stickers",
        "quantity": 15,
        "code": 223
    }, {
        "name": "keep cup",
        "quantity": 3,
        "code": 224
    }]

    res.send(catalogue);
});

app.listen(PORT, HOST);
console.log(`Running on http://${HOST}:${PORT}`);
```

Listing 2: Consumer code

```
const PORT = 8081;
const HOST = '0.0.0.0';

app.get('/', (req, res) => {
    res.send("Hello from consumer");
});

app.get('/invoke', (req, res) => {
    const url = "http://provider-service:8080/list"
    const result = request('GET', url).getBody('utf8');
    res.json("Catalogue contains: " + result)
});
```

3.2.2 Creating the experiment

The experiment will aim to ensure the availability of the microservices codebase. The steady state *hypothesises* that the system can withhold an unexpected failure in one of the services, and the chaos experiment will aim to *prove* that this is in fact the case.

Firstly, in the steady state hypothesis, probes are added to ensure both pods are in the `Running` state, and that the exposed service URL for the consumer is available.

Listing 3: Steady state hypothesis

```
"steady-state-hypothesis": {
        "title": "Services are all available and healthy",
        "probes": [{
                "type": "probe",
                "name": "provider-should-be-alive-and-healthy",
                "tolerance": true,
                "provider": {
                    "type": "python",
                    "module": "chaosk8s.pod.probes",
                    "func": "pods_in_phase",
                    "arguments": {
                        "label_selector": "app=provider-pod",
                        "phase": "Running",
                        "ns": "default"
                    }
                }
            },
            {
                "type": "probe",
                "name": "consumer-should-be-alive-and-healthy",
                "tolerance": true,
                "provider": {
                    "type": "python",
                    "module": "chaosk8s.pod.probes",
                    "func": "pods_in_phase",
                    "arguments": {
                        "label_selector": "app=consumer-pod",
                        "phase": "Running",
                        "ns": "default"
                    }
                }
            },
            {
                "type": "probe",
                "name": "application-must-respond-normally",
                "tolerance": 200,
                "provider": {
                    "type": "http",
                    "url": "${consumer_app_url}",
                    "timeout": 3
                }
            }
        ]
    },
```

- A probe is added to check that pods with the label `app=provider-pod` in the default namespace are in the `Running` state. This uses the `pods_in_phase` helper function from the Kubernetes Chaos Toolkit extension.
- Similarly, a probe is added to check that pods with the label `app=consumer-pod` in the default namespace are also in the `Running` state.

- Finally, a probe is added to check that the consumer service is responding with a 200 OK status. This does not require any additional extensions.

Next, the actual experiment is performed. The action involves using the `chaosk8s.pod.actions` module, specifically the `terminate_pods` function, to terminate pods in a given namespace with a given label. The provider pod has a label of `app=provider-pod` (see Appendix C), and is deployed in the default namespace, so like before these parameters will be passed in to the arguments object. Additionally, although less common, the `method` object can also take in probes. To illustrate this, a probe which attempts to read the pod logs after terminating the pod is added as well:

Listing 4: Chaos experiment

```
"method": [{
            "type": "action",
            "name": "terminate-provider-service",
            "provider": {
                "type": "python",
                "module": "chaosk8s.pod.actions",
                "func": "terminate_pods",
                "arguments": {
                    "label_selector": "app=provider-pod",
                    "ns": "default"
                }
            }
        },
        {
            "type": "probe",
            "name": "fetch-application-logs",
            "provider": {
                "type": "python",
                "module": "chaosk8s.pod.probes",
                "func": "read_pod_logs",
                "arguments": {
                    "label_selector": "app=provider-pod",
                    "last": "20s",
                    "ns": "default"
                }
            }
        }
    ],
```

To summarise:

- An action is added to the method array to terminate the provider pod. This will determine whether or not the consumer pod is able to respond normally, in the event that the provider service becomes unavailable for some reason. As stated previously, this is done using the `terminate_pods` function from the `chaosk8s.pod.probes` module.
- Finally, a probe is added to the method array to fetch the application logs for the provider pod after the first action has been executed. This is done using the `read_pod_logs` function from the `chaosk8s.pod.probes` module.

3.2.3 Running the experiment

Before running the experiment, the URL of the consumer services needs to be passed in to the JSON file. This can be done through an environment variable by adding the following block to the `experiment.json` file. The entire experiment can be found in Appendix D.

```
    "configuration": {
        "consumer_app_url": {
            "type": "env",
            "key": "CONSUMER_URL"
        }
    },
```

Running the experiment now is extremely straightforward. After activating the python virtual environment, the `chaos run` command can be used to run the test:

```
export CONSUMER_URL=http://35.222.64.12:8081/invoke

(chaostk) chaos run experiment.json
[INFO] Validating the experiment's syntax
[INFO] Experiment looks valid

[INFO] Running experiment: Does terminating the provider service impact user experience?
[INFO] Steady state hypothesis: Services are all available and healthy
[INFO] Probe: provider-should-be-alive-and-healthy
[INFO] Probe: consumer-should-be-alive-and-healthy
[INFO] Probe: application-must-respond-normally
[INFO] Steady state hypothesis is met!

[INFO] Action: terminate-provider-service
[INFO] Probe: fetch-application-logs
[ERROR]  => failed: kubernetes.client.rest.ApiException: (400)
   Reason: Bad Request
   HTTP response headers: HTTPHeaderDict({'Content-Type': 'application/json', 'Date': 'Fri, 22
       Mar 2019 09:06:20 GMT', 'Content-Length': '205'})
   HTTP response body:
       b'{"kind":"Status","apiVersion":"v1","metadata":{},"status":"Failure","message":"container
       \\"provider\\" in pod \\"provider-rs-spp97\\" is waiting to start:
       ContainerCreating","reason":"BadRequest","code":400}\n'

[INFO] Steady state hypothesis: Services are all available and healthy
[INFO] Probe: provider-should-be-alive-and-healthy
[ERROR]  => failed: chaoslib.exceptions.ActivityFailed: pod 'app=provider-pod' is in phase
    'Pending' but should be 'Running'
[WARNING] Probe terminated unexpectedly, so its tolerance could not be validated
[CRITICAL] Steady state probe 'provider-should-be-alive-and-healthy' is not in the given
    tolerance so failing this experiment

[INFO] Let's rollback...
[INFO] No declared rollbacks, let's move on.
[INFO] Experiment ended with status: deviated
[INFO] The steady-state has deviated, a weakness may have been discovered
```

This does the following:

- Checks the syntax of the `experiment.json` file.
- Checks the steady state hypothesis to ensure the system is currently in a stable state, before executing the chaos test. The steady state (probing the provider and consumer services) is met.
- Executes the chaos experiment through the defined actions (terminating the provider pod).
- A weakness is found that is affecting the availability of the service, and therefore the steady state of the system. This is due to the fact that the provider pod was not in the correct running state, and the consumer returned an unexpected status code, different to what was specified in the tolerances of the steady state. The hypothesis is therefore disproven, because the system cannot tolerate an

impact on the availability of one of its services. This is important information, that will be used to improve the reliability of the codebase and therefore of the user's experience.

- No rollbacks are specified, so the experiment is complete.

3.2.4 Learning from the experiment

The experiment reveals a few weaknesses in the system:

- The services are directly communicating to each other through REST endpoints. This is known as synchronous inter-process communication. There are however some flaws in this case in the implementation of this communication method.
- The consumer service is calling the provider service and is not catering for any failure or latency that might exist within the network, and is also not accounting for potential failure of the provider service itself. There are a few potential solutions to this scenario. Firstly, the *Circuit Breaker* pattern, popularised by Michael T. Nygard in the book Release It!, could be used. The Circuit Breaker is a wrapper around the call to the provider service which attempts to reconnect if a disruption in the connection is observed, but will ultimately failover after a given number of retries/timeouts. Another, possibly more simple solution, would be to supply a fallback response to the user with partial functionality. Although this might not be the greatest user experience, it is preferable to simply erroring out and not providing a response at all.
- Another potential solution would involve asynchronous inter-process communication. This would allow the microservices to communicate asynchronously through a message bus and allows higher decoupling of the services, but does introduce additional complexity.

For simplicity, a simple fallback response will be supplied, and the chaos experiment will be re-run. The updated code for the consumer now returns a `"No items available"` message when it cannot reach the catalogue:

```
app.get('/invoke', (req, res) => {
    const url = "http://provider-service:8080/list"

    return fetch(url)
        .then(res => res.json())
        .then(json => {
            console.log(json)

            let catalogue = []
            json.forEach(function (obj) {
                catalogue.push(`${obj.name} (quantity: ${obj.quantity})`)
            });

            res.send(`Catalogue contains: ${catalogue.join(", ")}`)
        })
        .catch(err => {
            console.log(err)
            res.send("No items available")
        });
});
```

Re-running the chaos experiment shows that the experiment is successfully completed:

```
(chaostk) chaos run experiment.json
[INFO] Validating the experiment's syntax
[INFO] Experiment looks valid
[INFO] Running experiment: Does terminating the provider service impact user
    experience?
```

```
[INFO] Steady state hypothesis: Services are all available and healthy
[INFO] Probe: application-should-be-alive-and-healthy
[INFO] Probe: application-must-respond-normally
[INFO] Steady state hypothesis is met!

[INFO] Action: terminate-provider-service
[INFO] Pausing after activity for 4s...
[INFO] Steady state hypothesis: Services are all available and healthy
[INFO] Probe: application-should-be-alive-and-healthy
[INFO] Probe: application-must-respond-normally
[INFO] Steady state hypothesis is met!

[INFO] Let's rollback...
[INFO] No declared rollbacks, let's move on.
[INFO] Experiment ended with status: completed
```

This means that there is substantial evidence to conclude that the system can withhold failure of the provider service within given tolerances!

3.3 Experiment 2: Node Drainage

3.3.1 Introduction

The next experiment is based on the ChaosIQ blog post *Exploring Multi-level Weaknesses using Automated Chaos Experiments* and will use the same services and deployments as the first, but will attempt to look at the resilience of the cluster to unplanned maintenance or node drainage. This scenario could arise from a miscommunication between application teams and operations teams, and could potentially result in downtime and unavailability of the application if not managed correctly.

3.3.2 Creating the experiment

Firstly, to ensure that all nodes in the cluster are up and running, the following command can be executed:

```
kubectl get nodes

NAME                                             STATUS   ROLES    AGE   VERSION
gke-standard-cluster-1-default-pool-a35eaadf-2c3j Ready <none> 2h    v1.11.7-gke.12
gke-standard-cluster-1-default-pool-a35eaadf-6vjw Ready <none> 2h    v1.11.7-gke.12
gke-standard-cluster-1-default-pool-a35eaadf-xfz1 Ready <none> 2h    v1.11.7-gke.12
```

A new `experiment.json` file can be created with the same steady state probes as the first experiment. The `method` array will be updated to reflect the actions for the new experiment. The main action uses the `drain_nodes` function from the `chaosk8s.node.actions` module, and takes the name of the node to be drained/cordoned. One of the node names from the command above could be then chosen for the node drainage action, but in this case no `name` argument is supplied, so *all* nodes will be cordoned.

```
"method": [{
    "type": "action",
    "name": "drain_node",
    "provider": {
        "type": "python",
        "module": "chaosk8s.node.actions",
        "func": "drain_nodes",
        "arguments": {
            "delete_pods_with_local_storage": true
        }
    }
}],
"rollbacks": [{
    "type": "action",
    "name": "uncordon_node",
    "provider": {
        "type": "python",
        "module": "chaosk8s.node.actions",
        "func": "uncordon_node",
        "arguments": {}
    }
}]
```

To summarise:

- All of the nodes in the GKE cluster are drained, along with their persistent volumes. Any existing pods will be evicted from the node, and any new pods scheduled to the node will be stuck in `pending` state.
- Once the experiment has been executed, a rollback is specified to uncordon the node (or make the node available for scheduling once again).

3.3.3 Running the experiment

Once again, executing the experiment can be done by running the `chaos run experiment.json` command:

```
[INFO] Validating the experiment's syntax
[INFO] Experiment looks valid
[INFO] Running experiment: My application is resilient to admin-instigated node
    drainage
[INFO] Steady state hypothesis: Services are all available and healthy
[INFO] Probe: pods_in_phase
[INFO] Probe: application-must-respond-normally
[INFO] Steady state hypothesis is met!

[INFO] Action: drain_node
[INFO] Action: terminate-provider-service
[INFO] Steady state hypothesis: Services are all available and healthy
[INFO] Probe: pods_in_phase
[ERROR]  => failed: chaoslib.exceptions.ActivityFailed: pod 'app=provider-pod' is in
    phase 'Pending' but should be 'Running'
[WARNING] Probe terminated unexpectedly, so its tolerance could not be validated
[CRITICAL] Steady state probe 'pods_in_phase' is not in the given tolerance so
    failing this experiment

[INFO] Let's rollback...
[INFO] Rollback: uncordon_node
[INFO] Action: uncordon_node
[INFO] Experiment ended with status: deviated
[INFO]  The steady-state has deviated, a weakness may have been discovered
```

3.3.4 Learning from the experiment

Reading the results of the test, it can be inferred that once the chaos experiment has drained all the nodes, the deployment controller sees that no pods are currently up and running, despite the desired number of replicas specified as `3`. The replica set controller attempts to reschedule the pods, but they all fall into pending state, because all of the nodes have been cordoned. This causes an entire blackout of the system, and 100% downtime for the end user. This is of course not a desirable situation and should be avoided at all costs.

The solution to this unfortunate string of events involves the `Pod Disruption Budget`. This dictates that a minimum number of pods of a service must be up and running at all times:

Listing 5: Pod Disruption Budget for the provider service

```
{
    "apiVersion": "policy/v1beta1",
    "kind": "PodDisruptionBudget",
    "metadata": {
        "name": "provider-pdb"
    },
    "spec": {
        "minAvailable": 3,
        "selector": {
            "matchLabels": {
                "app": "provider-pod"
            }
        }
    }
}
```

To prove that the system will ensure that a minimum of 3 pods for the provider service are up and running at all times, the chaos experiment can be run again:

```
[INFO] Validating the experiment's syntax
[INFO] Experiment looks valid
[INFO] Running experiment: My application is resilient to admin-instigated node drainage
[INFO] Steady state hypothesis: Services are all available and healthy
[INFO] Probe: pods_in_phase
[INFO] Probe: application-must-respond-normally
[INFO] Steady state hypothesis is met!

[INFO] Action: drain_node
[ERROR]  => failed: chaoslib.exceptions.ActivityFailed: Failed to evict pod
    provider-f65b5ccf8-8vm5v:
    {"kind":"Status","apiVersion":"v1","metadata":{},"status":"Failure","message":"Cannot evict
    pod as it would violate the pod's disruption budget.",
"reason":"TooManyRequests","details":{"causes":[{"reason":"DisruptionBudget","message":"The
    disruption budget provider-pdb needs 3 healthy pods and has 3 currently"}]},"code":429}
[INFO] Steady state hypothesis: Services are all available and healthy
[INFO] Probe: pods_in_phase
[INFO] Probe: application-must-respond-normally
[INFO] Steady state hypothesis is met!

[INFO] Let's rollback...
[INFO] Rollback: uncordon_node
[INFO] Action: uncordon_node
[INFO] Experiment ended with status: completed
```

The result is that the pods cannot be evicted, since the budget enforces 3 pods running at all times, and no downtime for the user!

3.4 Experiment 3: Gremlin Attack

3.4.1 Introduction

Gremlin provides *Chaos as a Service* and can be set up within a number of different environments to unleash chaos with relative ease. In this experiment, Gremlin will be set up and used to perform a resource attack on the GKE cluster.

3.4.2 Installation

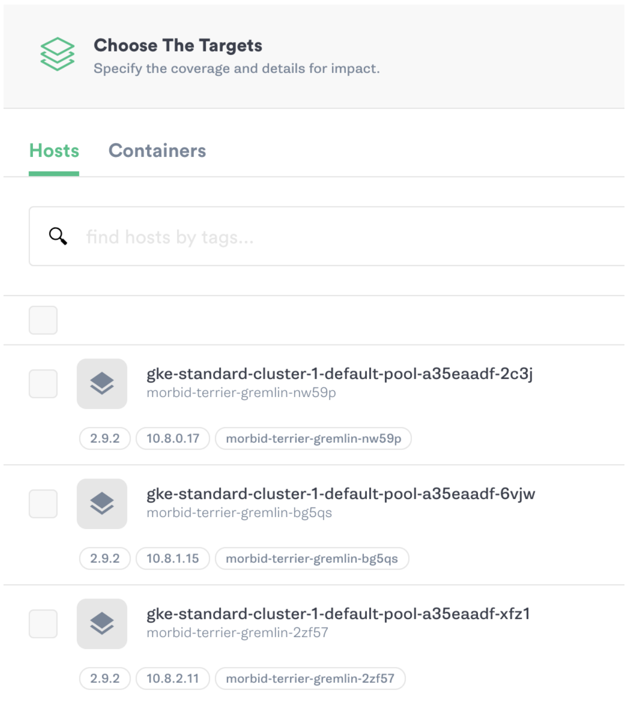
Gremlin requires the Gremlin Client to be installed within the cluster as a pod on each node, in order to perform its attacks. This can be done on Kubernetes using a series of Helm charts. Helm is a popular package manager for Kubernetes.

```
# Install helm
brew install kubernetes-helm

# The next four commands are a fix for the error
# "helm error: no available release name found"
kubectl create serviceaccount --namespace kube-system tillerkubectl create clusterrolebinding
    tiller --clusterrole=cluster-admin --serviceaccount=kube-system:tiller

kubectl create serviceaccount --namespace kube-system tiller

kubectl create clusterrolebinding tiller-cluster-rule --clusterrole=cluster-admin
    --serviceaccount=kube-system:tiller

kubectl patch deploy --namespace kube-system tiller-deploy -p
    '{"spec":{"template":{"spec":{"serviceAccount":"tiller"}}}}'

# Add the gremlin repo
helm repo add gremlin https://helm.gremlin.com

# Install gremlin on kubernetes
helm install gremlin/gremlin --set gremlin.teamID=<team-id>
```

The Gremlin pods need to be in the `Running` state before a new attack can be created. This takes a few minutes.

```
kubectl get pods

NAME                          READY  STATUS   RESTARTS  AGE
morbid-terrier-gremlin-2zf57 1/1    Running  0         15h
morbid-terrier-gremlin-bg5qs 1/1    Running  0         15h
morbid-terrier-gremlin-nw59p 1/1    Running  0         15h
```

Navigating to `Attacks > New Attack`, the targets list should now contain a list of the GKE Nodes within the cluster. These nodes will be used to execute chaos attacks with Gremlin:


Choose The Targets
Specify the coverage and details for impact.
Hosts Containers
find hosts by tags...

gke-standard-cluster-1-default-pool-a35eaadf-2c3j
morbid-terrier-gremlin-nw59p
2.9.2 10.8.0.17 morbid-terrier-gremlin-nw59p

gke-standard-cluster-1-default-pool-a35eaadf-6vjw
morbid-terrier-gremlin-bg5qs
2.9.2 10.8.1.15 morbid-terrier-gremlin-bg5qs

gke-standard-cluster-1-default-pool-a35eaadf-xfz1
morbid-terrier-gremlin-2zf57
2.9.2 10.8.2.11 morbid-terrier-gremlin-2zf57

Figure 1: GKE Nodes detected in Gremlin Dashboard

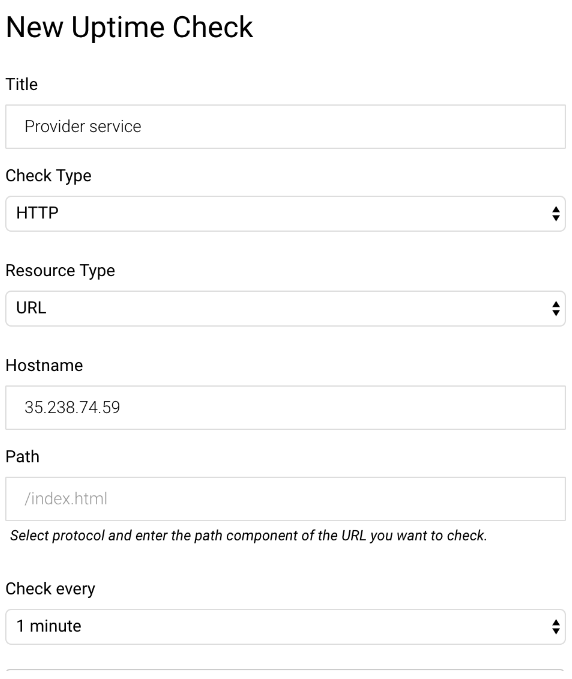
3.4.3 Monitoring with Google Stack Driver

Next, to ensure that the services are in fact available when executing new attacks with Gremlin, some different monitoring tools can be used. In this case, after exploring Prometheus & Grafana, DataDog and Google Stackdriver, I decided to settle with Stackdriver as I found it the most intuitive tool to work with. Using Stackdriver together with the built-in system monitoring tools for GKE, allows a comprehensive overview of the entire system.

After creating a new Stackdriver project, two new uptime checks can be added for the provider service and the consumer service respectively.

```
NAME             TYPE         CLUSTER-IP    EXTERNAL-IP   PORT(S)         AGE
consumer-service LoadBalancer 10.11.245.10  35.222.64.122 8081:31519/TCP  1h
provider-service LoadBalancer 10.11.246.174 35.238.74.59  8080:30729/TCP  18h
```

In the Uptime Check dashboard, under `New Uptime Check`, the details for the provider's service endpoint can be added. Under advanced options, port `8080` is specified.



New Uptime Check

Title: Provider service

Check Type: HTTP

Resource Type: URL

Hostname: 35.238.74.59

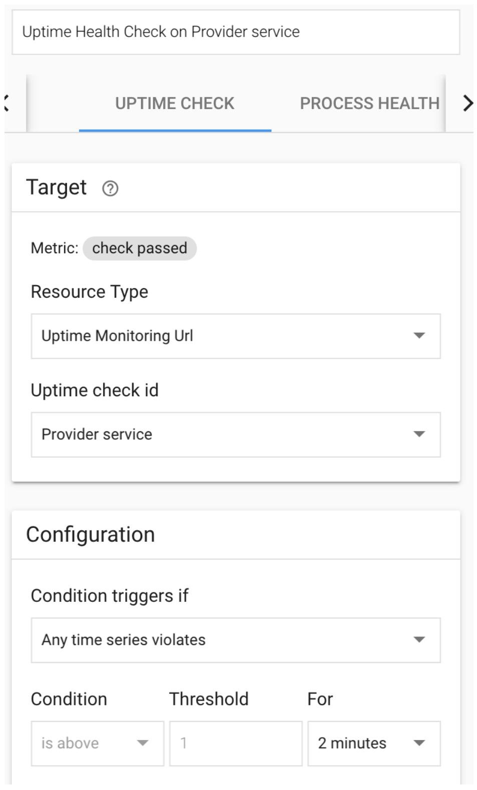
Path: /index.html

Select protocol and enter the path component of the URL you want to check.

Check every: 1 minute

Figure 2: Uptime check for the provider service

Next, a new alert is created which is triggered when the service is down for over 2 minutes. The alert can be attached to a new notification channel by selecting the type of channel. In this case I chose email and specified my personal email address. If the alert is triggered, a new email will be sent to my email address with details of the incident.



Uptime Health Check on Provider service

Target

Metric: check passed

Resource Type: Uptime Monitoring Url

Uptime check id: Provider service

Configuration

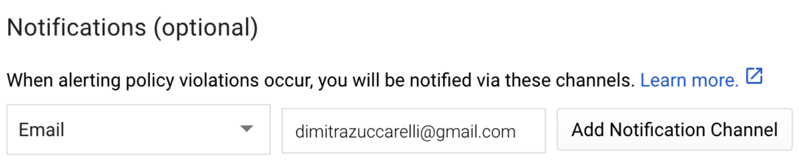
Condition triggers if: Any time series violates

Condition: is above

Threshold: 1

For: 2 minutes

Figure 3: Adding alerts


Notifications (optional)

When alerting policy violations occur, you will be notified via these channels. Learn more.

Email: dimitrazuccarelli@gmail.com

Figure 4: Adding email alerts

The same process can be repeated for the consumer service. After a few minutes, the the check is up and running and all locations are passing:

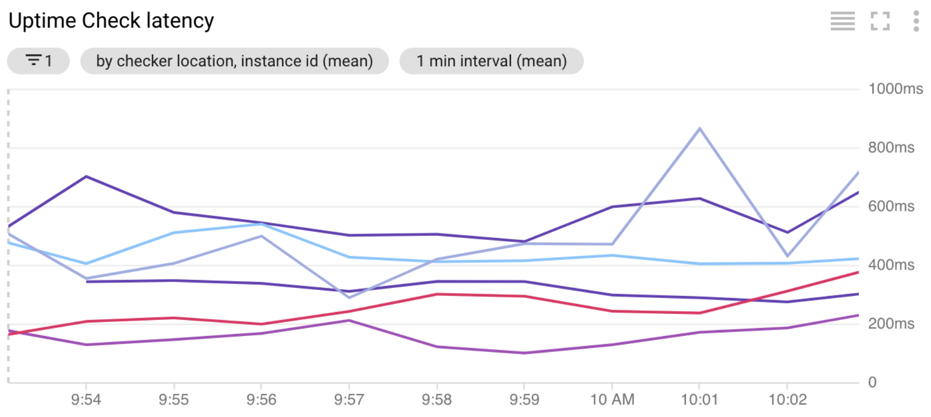
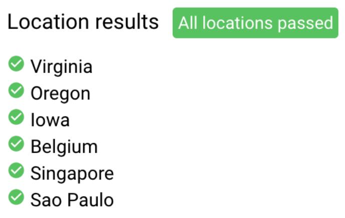


Figure 5: Uptime Check Graph

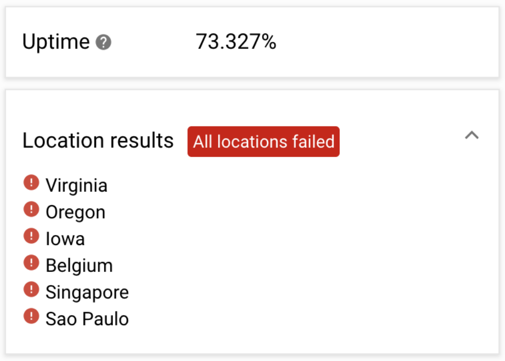

Location results All locations passed

- ✓ Virginia
- ✓ Oregon
- ✓ Iowa
- ✓ Belgium
- ✓ Singapore
- ✓ Sao Paulo

Figure 6: Location checks

To verify that the alerts are working as expected, the pods from the provider service can be killed by scaling the replicas down to `0` in the deployment. This should cause the probe to fail and an alert email notification to be sent.

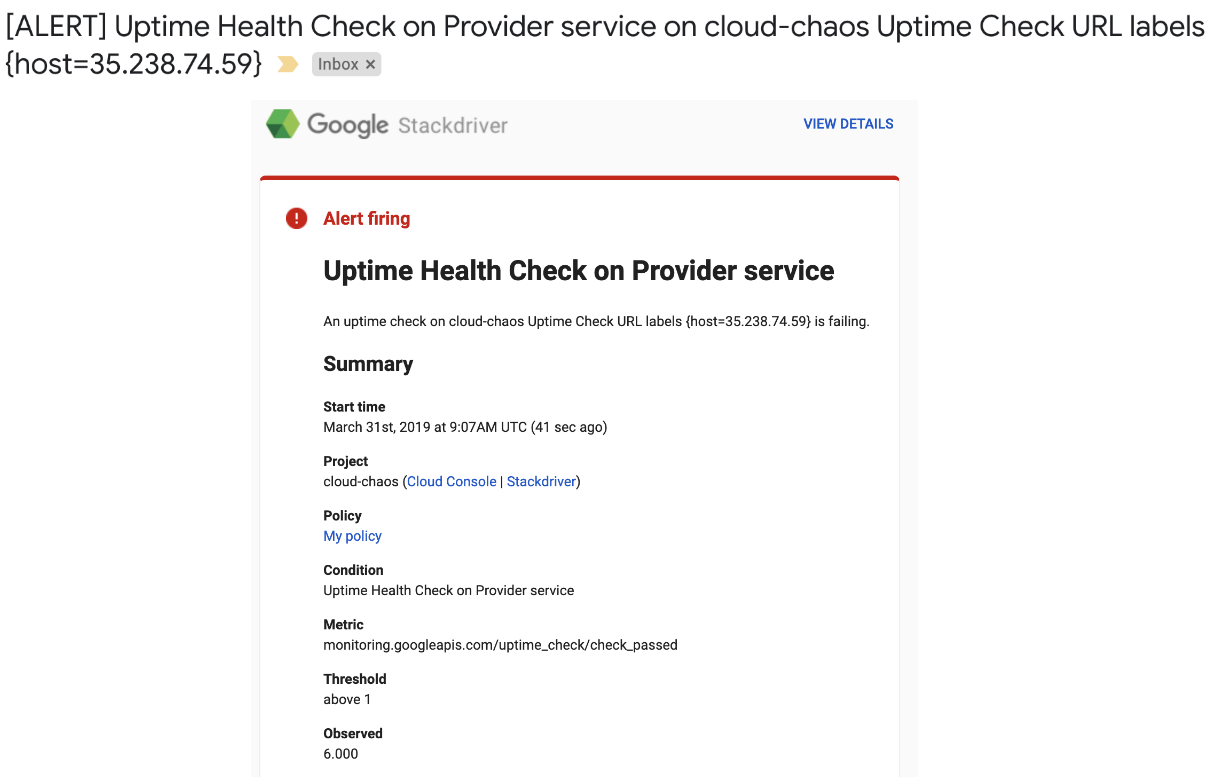
As expected, the uptime percentage slowly starts to drop as all the locations start to fail. An alert email is sent after a few minutes.


Uptime 73.327%

Location results All locations failed

- ! Virginia
- ! Oregon
- ! Iowa
- ! Belgium
- ! Singapore
- ! Sao Paulo

Figure 7: Uptime Check percentage


[ALERT] Uptime Health Check on Provider service on cloud-chaos Uptime Check URL labels {host=35.238.74.59} Inbox ×

Google Stackdriver VIEW DETAILS

Alert firing

Uptime Health Check on Provider service

An uptime check on cloud-chaos Uptime Check URL labels {host=35.238.74.59} is failing.

Summary

Start time
March 31st, 2019 at 9:07AM UTC (41 sec ago)

Project
cloud-chaos (Cloud Console | Stackdriver)

Policy
My policy

Condition
Uptime Health Check on Provider service

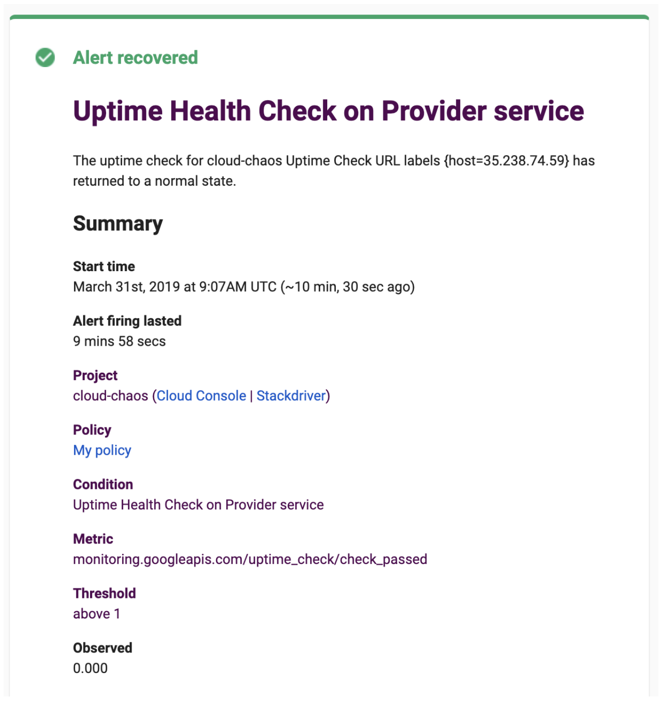
Metric
monitoring.googleapis.com/uptime_check/check_passed

Threshold
above 1

Observed
6.000

Figure 8: Email notification

Scaling the pods back up causes everything to return back to normal:


Alert recovered

Uptime Health Check on Provider service

The uptime check for cloud-chaos Uptime Check URL labels {host=35.238.74.59} has returned to a normal state.

Summary

Start time
March 31st, 2019 at 9:07AM UTC (~10 min, 30 sec ago)

Alert firing lasted
9 mins 58 secs

Project
cloud-chaos (Cloud Console | Stackdriver)

Policy
My policy

Condition
Uptime Health Check on Provider service

Metric
monitoring.googleapis.com/uptime_check/check_passed

Threshold
above 1

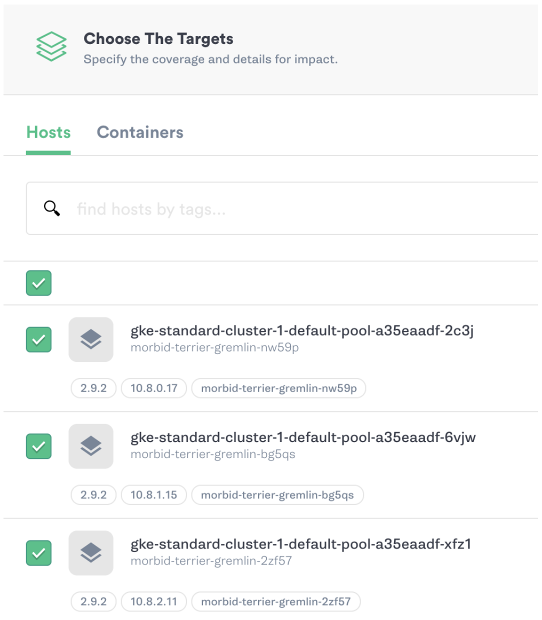
Observed
0.000

Figure 9: Email notification

3.4.4 Preparing the attack

The attack can now be prepared using the Gremlin dashboard.

For the targets, all of the GKE nodes should be selected:


Choose The Targets
Specify the coverage and details for impact.

Hosts | Containers

find hosts by tags...

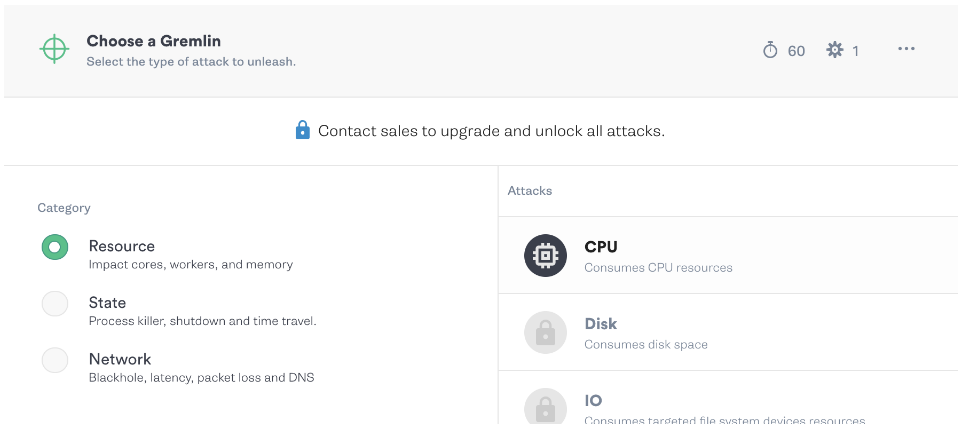
✓

✓ gke-standard-cluster-1-default-pool-a35eaadf-2c3j
morbid-terrier-gremlin-nw59p
2.9.2 | 10.8.0.17 | morbid-terrier-gremlin-nw59p

✓ gke-standard-cluster-1-default-pool-a35eaadf-6vjw
morbid-terrier-gremlin-bg5qs
2.9.2 | 10.8.1.15 | morbid-terrier-gremlin-bg5qs

✓ gke-standard-cluster-1-default-pool-a35eaadf-xfz1
morbid-terrier-gremlin-2zf57
2.9.2 | 10.8.2.11 | morbid-terrier-gremlin-2zf57

Next, the "`Gremlin`", or attack type, is selected. For this experiment a CPU resource attack will be executed:


Choose a Gremlin
Select the type of attack to unleash.
60 1

Contact sales to upgrade and unlock all attacks.

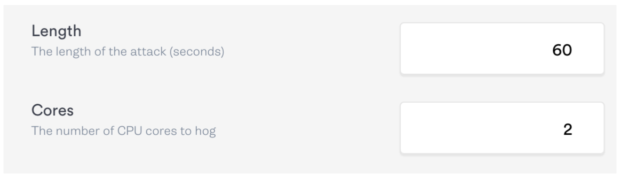
Category

[● Resource] Impact cores, workers, and memory
State — Process killer, shutdown and time travel.
Network — Blackhole, latency, packet loss and DNS

Attacks

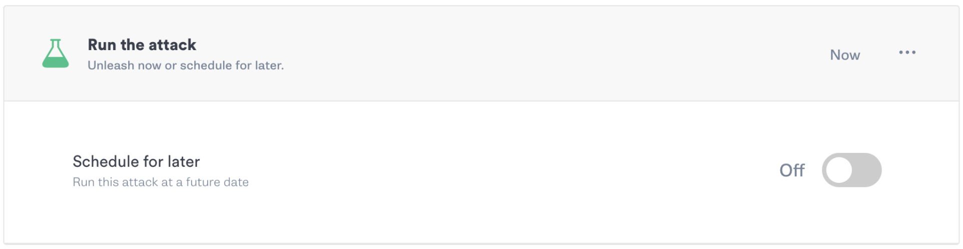
CPU — Consumes CPU resources
Disk — Consumes disk space
IO — Consumes targeted file system devices resources

The attack will target all available CPUs for 1 minute:

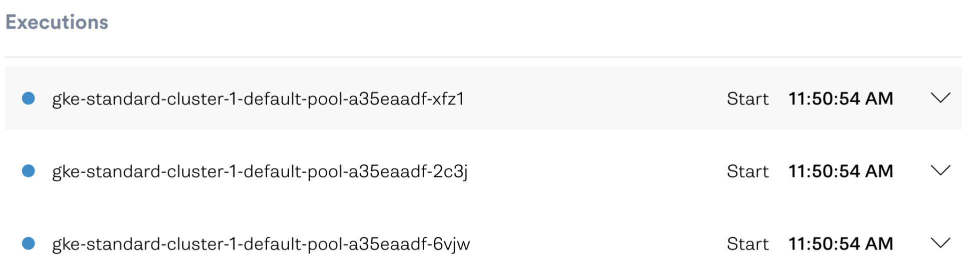


Length The length of the attack (seconds)	60
Cores The number of CPU cores to hog	2

Finally, the Gremlin can be unleashed to cause chaos within the cluster!

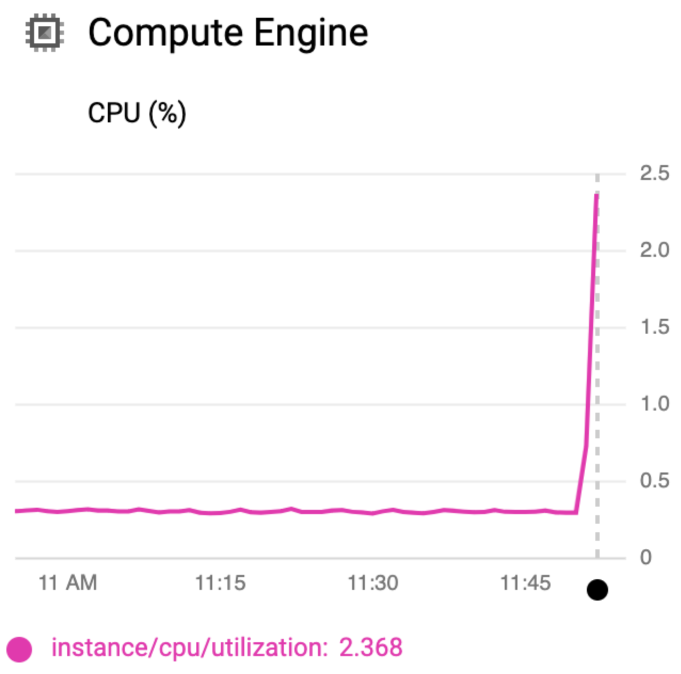

Run the attack
Unleash now or schedule for later.
Now

Schedule for later
Run this attack at a future date
Off

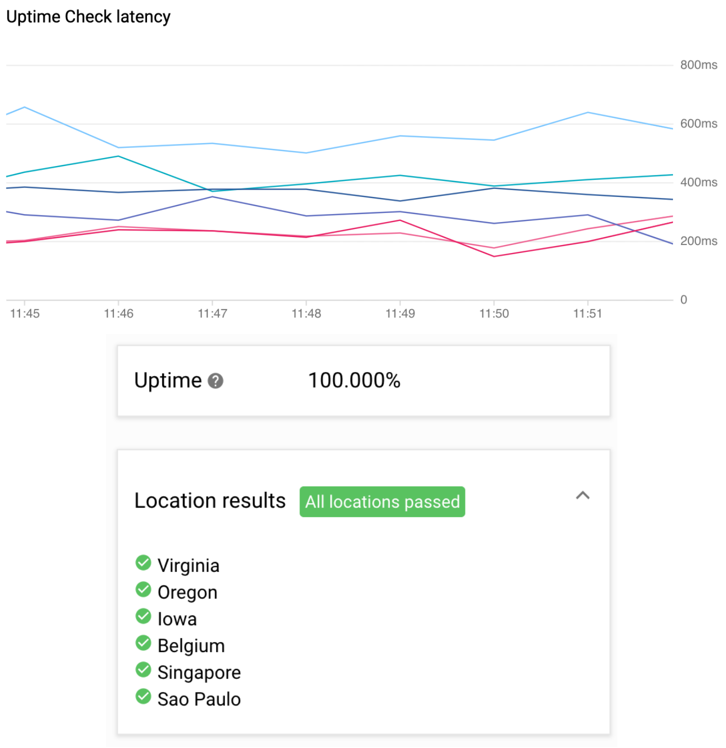

Executions

gke-standard-cluster-1-default-pool-a35eaadf-xfz1	Start	11:50:54 AM
gke-standard-cluster-1-default-pool-a35eaadf-2c3j	Start	11:50:54 AM
gke-standard-cluster-1-default-pool-a35eaadf-6vjw	Start	11:50:54 AM

The monitoring set up previously will be used to check if users can still connect to the service. As shown below, all CPU resources show a significant peak:



However, the services continue to be accessible and no downtime is experienced:

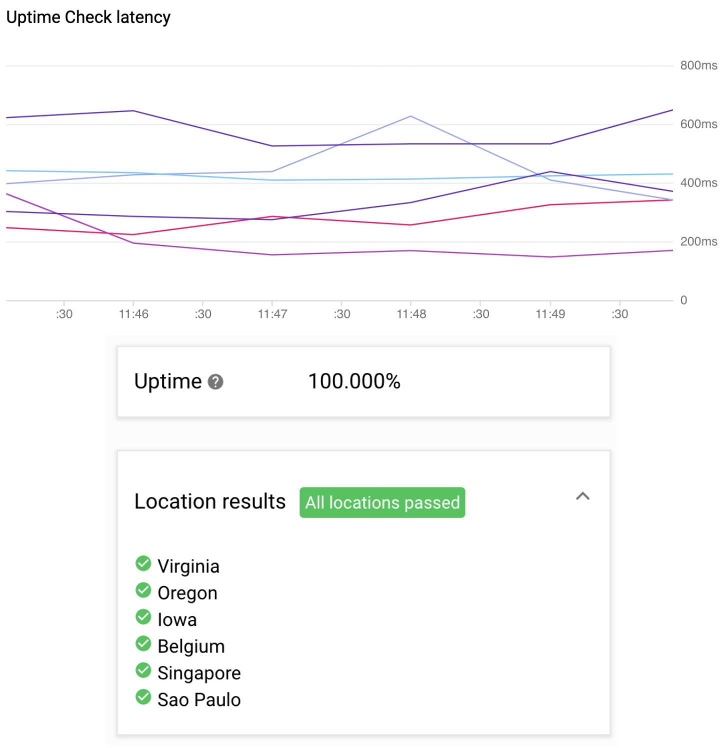


Uptime 100.000%

Location results All locations passed

- ✓ Virginia
- ✓ Oregon
- ✓ Iowa
- ✓ Belgium
- ✓ Singapore
- ✓ Sao Paulo

Figure 10: Uptime for Consumer



Uptime 100.000%

Location results All locations passed

- ✓ Virginia
- ✓ Oregon
- ✓ Iowa
- ✓ Belgium
- ✓ Singapore
- ✓ Sao Paulo

Figure 11: Uptime for Provider

3.4.5 Automating Gremlin attacks with Chaos Toolkit

Finally, Gremlin provides an extension for Chaos Toolkit to automate Gremlin chaos experiments. The experiment from above will be repeated, this time using a Chaos Toolkit JSON file:

```
{
    "title": "Can the system handle an attack on node CPU?",
    "description": "CPU usage may impact response time",
    "secrets": {
        "gremlin": {
            "email": {
                "type": "env",
                "key": "GREMLIN_EMAIL"
            },
            "password": {
                "type": "env",
                "key": "GREMLIN_PWD"
            },
            "org_name": {
                "type": "env",
                "key": "GREMLIN_ORG_NAME"
            }
        }
    },
    "steady-state-hypothesis": {
        "title": "Services are all available and healthy",
        "probes": [{
                "type": "probe",
                "name": "application-should-be-alive-and-healthy",
                "tolerance": true,
                "provider": {
                    "type": "python",
                    "module": "chaosk8s.probes",
                    "func": "all_microservices_healthy",
                    "arguments": {
                        "ns": "default"
                    }
                }
            },
            {
                "type": "probe",
                "name": "application-must-respond-normally",
                "tolerance": 200,
                "provider": {
                    "type": "http",
                    "url": "http://35.222.64.122:8081/invoke",
                    "timeout": 5
                }
            }
        ]
    },
    "method": [{
        "name": "attack-on-cpu",
        "type": "action",
        "background": true,
        "provider": {
            "type": "python",
            "module": "chaosgremlin.actions",
            "func": "attack",
            "secrets": [
                "gremlin"
            ],
            "arguments": {
                "command": {
                    "type": "cpu"
                },
                "target": {
                    "type": "Random"
                }
            }
        }
    }]
}
```

To run the experiment, the `chaos run experiment.json` command is executed as before:

```
[INFO] Validating the experiment's syntax
[INFO] Experiment looks valid
[INFO] Running experiment: Can the system handle an attack on node CPU?
[INFO] Steady state hypothesis: Services are all available and healthy
[INFO] Probe: application-should-be-alive-and-healthy
[INFO] Probe: application-must-respond-normally
[INFO] Steady state hypothesis is met!

[INFO] Action: attack-on-cpu [in background]
[INFO] Steady state hypothesis: Services are all available and healthy
[INFO] Probe: application-should-be-alive-and-healthy
[INFO] Probe: application-must-respond-normally
[INFO] Steady state hypothesis is met!

[INFO] Let's rollback...
[INFO] No declared rollbacks, let's move on.
[INFO] Experiment ended with status: completed
```

This proves the conclusions from the first Gremlin attack, that the system can in fact withstand spikes in CPU usage on the GKE nodes. This experiment should now be run at random times in production to confirm resiliency to this attack.

4 Conclusions

I thoroughly enjoyed working on this term paper, as it is very much aligned with my own interests and the work explored for my Final Year Project. Reading the Chaos Engineering book and watching all of the talks available online was extremely interesting and gave me new insight into the logic of testing enterprise applications.

There were a lot of learning experiences and hurdles along the way:

- The content available online is somewhat limited. Given that Netflix is the pioneer of this fairly new field, most of the resources available online are limited to publications and talks delivered by Netflix employees, which means that variation of experiences from other companies, especially smaller ones, are entirely lacking.
- Most of the chaos tools are extremely new, and there was a lot of struggle involved with setup and configuration, even though they are designed to make it as easy as possible to get started. It was almost impossible to find in depth documentation of features, and there were no similar issues online of problems that I faced while carrying out the practical element of the term paper. I faced a lot of issues trying to work with Gremlin in particular, from installation to experimentation. I found that there was no documentation available around tweaking configuration and parameters, especially for the Chaos Toolkit extension. In summary, some of the tools were quite buggy to work with and this was mostly due to the fact that these products are not fully matured yet.
- Trying to switch to a chaos mindset from a testing mindset is hard to do! Chaos Engineers have to be extremely creative in the experiments they attempt to conjure up, as they need to explore a realm of unpredictable factors that could arise in a production environment. In Chaos Engineering, failure is seen as a positive, since it provides an opportunity to build reliability and resiliency. This is a very different mindset to failure in traditional testing.

In conclusion, while it is easy to see why companies, especially smaller companies and start-ups, would be hesitant to start injecting failure into their production environments, it is also easy to see how the adoption of Chaos Engineering could have so much potential to improve user experience in modern day applications. Switching to a mindset of accepting failure and attempting to work with it, rather than fearing it, can be hugely beneficial to companies when disaster eventually does arise: entropy is an inherent characteristic of any system, as stated previously, and should therefore be embraced.

Appendices

A Dockerfile Example

```
FROM node:8

WORKDIR /app

COPY package.json /app

RUN npm install

COPY . /app

EXPOSE 8080

CMD [ "node", "index.js" ]
```

B Consumer Deployment

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: consumer
spec:
  replicas: 3
  template:
    metadata:
      labels:
        app: consumer-pod
    spec:
      containers:
        - name: consumer
          image: dimitraz/chaos-node-consumer:latest
          ports:
            - containerPort: 8080
---
apiVersion: v1
kind: Service
metadata:
  name: consumer-service
spec:
  selector:
    app: consumer-pod
  ports:
    - port: 8081
  type: LoadBalancer
```

C Provider Deployment

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: provider
spec:
  replicas: 3
  template:
    metadata:
      labels:
        app: provider-pod
    spec:
      containers:
        - name: provider
          image: dimitraz/chaos-node-provider:latest
          ports:
            - containerPort: 8080
---
apiVersion: v1
kind: Service
metadata:
  name: provider-service
spec:
  selector:
    app: provider-pod
  ports:
    - port: 8080
  type: LoadBalancer
```

D Experiment 1: experiment.json

```
{
    "version": "1.0.0",
    "title": "Does terminating the provider service impact user experience?",
    "description": "How does terminating the provider affect the rest of the web app?",
    "tags": [
        "kubernetes",
        "availability"
    ],
    "configuration": {
        "consumer_app_url": {
            "type": "env",
            "key": "CONSUMER_URL"
        }
    },
    "steady-state-hypothesis": {
        "title": "Services are all available and healthy",
        "probes": [{
                "type": "probe",
                "name": "provider-should-be-alive-and-healthy",
                "tolerance": true,
                "provider": {
                    "type": "python",
                    "module": "chaosk8s.pod.probes",
                    "func": "pods_in_phase",
                    "arguments": {
                        "label_selector": "app=provider-pod",
                        "phase": "Running",
                        "ns": "default"
                    }
                }
            },
            {
                "type": "probe",
                "name": "consumer-should-be-alive-and-healthy",
                "tolerance": true,
                "provider": {
```

```
                "type": "python",
                "module": "chaosk8s.pod.probes",
                "func": "pods_in_phase",
                "arguments": {
                    "label_selector": "app=consumer-pod",
                    "phase": "Running",
                    "ns": "default"
                }
            }
        },
        {
            "type": "probe",
            "name": "application-must-respond-normally",
            "tolerance": 200,
            "provider": {
                "type": "http",
                "url": "${consumer_app_url}",
                "timeout": 3
            }
        }
    ]
},
"method": [{
        "type": "action",
        "name": "terminate-provider-service",
        "provider": {
            "type": "python",
            "module": "chaosk8s.pod.actions",
            "func": "terminate_pods",
            "arguments": {
                "label_selector": "app=provider-pod",
                "ns": "default"
            }
        }
    },
    {
        "type": "probe",
        "name": "fetch-application-logs",
        "provider": {
            "type": "python",
            "module": "chaosk8s.pod.probes",
            "func": "read_pod_logs",
            "arguments": {
                "label_selector": "app=provider-pod",
                "last": "20s",
                "ns": "default"
            }
        }
    }
],
"rollbacks": []
}
```

E Experiment 1: Updated experiment.json

```
{
    "version": "1.0.0",
    "title": "Does terminating the provider service impact user experience?",
    "description": "How does terminating the provider affect the rest of the web app?",
    "tags": [
        "kubernetes",
        "availability"
    ],
    "configuration": {
        "consumer_app_url": {
            "type": "env",
            "key": "CONSUMER_URL"
        }
    },
    "steady-state-hypothesis": {
        "title": "Services are all available and healthy",
        "probes": [{
                "type": "probe",
                "name": "application-should-be-alive-and-healthy",
                "tolerance": true,
                "provider": {
                    "type": "python",
                    "module": "chaosk8s.probes",
                    "func": "all_microservices_healthy",
                    "arguments": {
                        "ns": "default"
                    }
                }
            },
            {
                "type": "probe",
                "name": "application-must-respond-normally",
                "tolerance": 200,
                "provider": {
                    "type": "http",
                    "url": "${consumer_app_url}",
                    "timeout": 5
                }
            }
        ]
    },
    "method": [{
        "type": "action",
        "name": "terminate-provider-service",
        "provider": {
            "type": "python",
            "module": "chaosk8s.pod.actions",
            "func": "terminate_pods",
            "arguments": {
                "label_selector": "app=provider-pod",
                "ns": "default"
            }
        },
        "pauses": {
            "after": 4
        }
    }],
    "rollbacks": []
}
```

F Experiment 1: Updated consumer code

```
const express = require('express');
const fetch = require('node-fetch')
const app = express();

const PORT = 8081;
const HOST = '0.0.0.0';

app.get('/', (req, res) => {
    res.send("Hello from consumer");
});

app.get('/invoke', (req, res) => {
    const url = "http://provider-service:8080/list"

    return fetch(url)
        .then(res => res.json())
        .then(json => {
            console.log(json)

            let catalogue = []
            json.forEach(function (obj) {
                catalogue.push(`${obj.name} (quantity: ${obj.quantity})`)
            });

            res.send(`Catalogue contains: ${catalogue.join(", ")}`)
        })

        .catch(err => {
            console.log(err)
            res.send("No items available")
        });


});

app.listen(PORT, HOST);
console.log(`Running on http://${HOST}:${PORT}`);
```

References

[1] Adrian Cockroft. Adrian cockroft: "chaos engineering - what is it, and where it's going" - chaos conf 2018, Oct 2018.

[2] Nora Jones. Aws re:invent 2017 - nora jones describes why we need more chaos - chaos engineering, that is, Dec 2017.

[3] Nora Jones, Ali Basiri, Lorin Hochstein, Aaron Blohowiak, and Casey Rosenthal. *Chaos Engineering*. O'Reilly Media, Inc., 2017.

[4] principlesofchaos.org. Principles of chaos engineering.

Bibliography

- Site Reliability Engineering
  https://landing.google.com/sre/sre-book/toc/index.html
- Getting started with Chaos Engineering - Paul Stack
  https://www.youtube.com/watch?v=3Oc4-cMkGJY&t=2366s
- AWS re:Invent 2017: Performing Chaos at Netflix Scale (DEV334)
  https://www.youtube.com/watch?v=LaKGx0dAUlo&t=1255s
- Principles of Chaos Engineering - Netflix - SRECon2017
  https://www.youtube.com/watch?v=6ilMZqKdMMU&t=1849s
- GOTO 2016 - Chaos Intuition Engineering at Netflix - Casey Rosenthal
  https://www.youtube.com/watch?v=Q4nniyAarbs&t=181s
- Chaos Engineering: Why the World Needs More Resilient Systems
  https://www.youtube.com/watch?v=KhqfOXltR_M&t=961s
- Adrian Cockcroft on Chaos Architecture
  https://www.youtube.com/watch?v=ja6n5etN8hk
- GOTO 2018 - Developing a Chaos Architecture Mindset - Adrian Cockcroft
  https://www.youtube.com/watch?v=vHl7EZ5o0uY
- AWS re:Invent 2018: Breaking Containers: Chaos Engineering for Modern Applications on AWS (CON310)
  https://www.youtube.com/watch?v=B1nUzbuVEUs&t=1196s
- Chaos Engineering - Casey Rosenthal
  https://www.youtube.com/watch?v=OJL9qvP1qzQ
- Using Chaos to Build Resilient Systems
  https://www.youtube.com/watch?v=Mz3cXPV42Ks&t=205s
- Make Applications more Resilient on Kubernetes by adding Chaos Engineering
  https://medium.com/chaos-toolkit/make-applications-more-resilient-on-kubernetes-by-adding-chaos
- Interactive Chaos Engineering Tutorials
  https://medium.com/chaos-toolkit/interactive-chaos-engineering-tutorials-c29d3a863d26
- Run Chaos Experiments Without Risking Your Job
  https://blog.loadmill.com/run-chaos-experiments-without-risking-your-job-2c8a5f4b0bfc
- Gremlin
  https://www.gremlin.com/
- Chaos Toolkit
  https://chaostoolkit.org/