INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO
MESTRADO EM ENGENHARIA INFORMÁTICA




Engenharia de Resiliência

PAULO JORGE RICARDO ANDRADE
Outubro de 2018

POLITÉCNICO
DO PORTO

Resilience Engineering

Paulo Andrade

A dissertation submitted in partial fulfillment of
the requirements for the degree of Master's in Informatics Engineering,
Specialization Area of Software Engineering

Advisor: Isabel de Fátima Silva Azevedo

Enterprise advisor: Joaquim Vasco Oliveira dos Santos

Porto, October 2018

Dedicatory

To my family that support in every situation in my daily life, especially my two brothers and my parents. The hard work they put every day is an inspiration to me and without them none of the work done in my thesis was possible.

To my friends that are always present to hear my thoughts and to free my mind from work and problems.

To my two advisors, Isabel Azevedo and Vasco Santos, that helped me a lot through this entire thesis with innumerous revisions and suggestions. A big thank you for your attention.

Abstract

This thesis presents a study of a new discipline called Chaos Engineering and its approaches, that help to verify the correct behavior of a system and to discover new information about it, through chaos experiments like the shutdown of a machine or the simulation of latency in the network connections between applications. The case study was carried out at the company Mindera, to verify and improve the resilience to failures of a client's project.

Initially the chaos maturity of the project within the Chaos Maturity Model was in the first levels and it was necessary to increase its sophistication and adoption by conducting experiments to test and improve the resilience.

The cloud environment that the project uses, and the architecture is explained to contextualize the components that the experiments will use and test. Different alternatives to test disaster recovery plans are compared as well as the differences between the use of a test environment and the production environment. The value of carrying out experiments for the client project is described, as well as the identification of their value proposal. In the end, the analysis of the different chaos tools is performed using the TOPSIS method.

The four performed experiments test the system's resilience to failure of a database's primary node, the impact of latency in the network connections between different components, the system's reaction to the exhaustion of physical resources of a machine and finally the global test of a system's resiliency in the face of a server failure. After the execution, the experiences were evaluated by company experts.

In the end, the conclusions about the work developed are presented. The experiments carried out were classified as important for the project. A problem was found after in the latency introduction experiment and after changing the application's code, the system reaction was positive, and the number of responses was increased.

Keywords: Resilience Engineering, Chaos Experiments, Chaos Engineering.

Resumo

Esta tese apresenta um estudo de uma nova disciplina chamada *Chaos Engineering* e as suas abordagens, que ajudam a verificar o correto funcionamento e a descoberta de novas informações acerca de um sistema através de realização de experiências como o desligar de uma máquina ou a simulação de latência nas ligações de rede entre aplicações. O caso de estudo foi realizado na empresa Mindera, dentro de um projeto cliente, para verificar e melhorar a sua resiliência a falhas.

Inicialmente a maturidade de caos do projeto dentro do *Chaos Maturity Model* encontra-se nos primeiros níveis e tornou-se necessário aumentar a sua sofisticação e adoção através da realização de experiências para testar e melhorar a resiliência.

O ambiente de *cloud* que o projeto usa e a sua arquitetura é explicada para contextualizar os componentes que as experiências vão usar e testar. As diferentes alternativas de testar planos de recuperação a desastres são comparadas, assim como, as diferenças entre a utilização do ambiente de testes e de produção. O valor da realização de experiências para o projeto cliente é descrito, assim como a identificação da sua proposta de valor. No final, a análise das diferentes ferramentas de caos é realizada recorrendo ao método TOPSIS.

As quatro experiências executadas testam a resiliência do sistema perante a falha de um nó primário de uma base de dados, o impacto da latência nas ligações de rede entre diferentes componentes, a reação do sistema perante a exaustão de recursos físicos de uma máquina e por último o teste global da resiliência de um sistema perante a falha de um servidor. As experiências são posteriormente avaliadas por *experts* da empresa.

No final, as conclusões acerca do trabalho desenvolvido são apresentadas. As experiências realizadas foram classificadas como importantes para o projeto. Um problema foi encontrado na experiência de introdução de latência e após a alteração do seu código, a reação do sistema foi positiva e o número de respostas aumentou.

Palavras-Chave: Engenharia de Resiliência, Experiências de Caos, Engenharia de Caos.

Contents

List of Figures

List of Tables

List of Source Code

List of Acronyms and Initialisms

ARN	AWS Resource Name
ASG	Auto Scaling Group
AWS	Amazon Web Services
CE	Chaos Engineering
CMM	Chaos Maturity Model
CNF	Conjunctive Normal Form
CPU	Central Processing Unit
EC2	Elastic Cloud Compute
ELB	Elastic Load Balancer
FIT	Failure Injection Testing
HTTP	Hypertext Transfer Protocol
IAM	Identity and Access Management
ID	Identifier
Oplog	Operations Log
QA	Quality Assurance
QFD	Quality Functional Deployment
TCP	Transmission Control Protocol
VPC	Virtual Private Cloud
YAML	YAML Ain't Markup Language

1 Introduction

In this chapter, the context is presented and the project under analysis is described. Then, the problem that will be addressed is explained and the target project is classified within the chaos maturity model. The main objectives are described as well as the methodological approach to perform the experiments. This chapter ends with an overview of the structure of this document.

1.1 Context

Currently, most software applications are built as distributed systems and, increasingly, with a microservice architecture. Thus, monolithic applications are divided into loosely coupled components with different responsibilities and deployment processes, that work together to fulfil a common end goal. As each component to have its own development cycle, adding new functionalities, changing runtime configurations and creating deployments to production are performed independently from other services.

However, these continuous changes increase the probability of failure in some parts of the system. There are different approaches to design an application to prevent failures and still, running applications struggle to be available after the deployment. Unit tests and integration tests are important and can catch problems related with business logic, but many failures can occur in deployment environments that are not addressed by those tests.

The formal verification of a large-scale system is not reasonable, there are too many paths to search for failure. Model check methods (Alvaro & Tymon, 2017) need a formal specification which may not exist for some services. Moreover, services communicate with messages and the failures in communication usually results in the absence of message, detected with a timeout, which causes correct services not to hold as a correct system under composition.

The impossibility to completely test these new architectures brought some new approaches, such as fault injection and chaos engineering. Experiments against the system with introduction of failures in a controlled environment are used to test and verify that fault tolerance mechanisms work, and the system is prepared to hold against similar situations in production.

Chaos engineering allows to identify problems and provide ways to uncover system weaknesses in the system by constantly test the system against failures and build confidence that those not happen in the real environment. A problem that arises during these tests can immediately be addressed and does not have the chance to cause a real problem. The team is aware of the problem and is ready to mitigate it.

Different failures (Wasson, Bennage, & Buck, 2017) can be introduced such as shut down instances, crash processes, expire certificates, exhaust physical resources, redeploy an application, test unusual combinations of messages, latency introduction between the services and other real problems that usually happen in a real environment. It is important to complement these tests with a load testing tool, since there are some problems that only occur under load such as a throttling of the service or a database being saturated with requests.

This work was developed in Mindera, where the complexity of many systems is very high. A constant need of new features in products, many runtime configuration modifications and improvements in performance, results in many deployments within a day. Every deployment to production is previously tested with unit and integration tests, tested in different environments similar to production. However, there is a need to reach another level of resilience.

Mindera is a Portuguese software company that was founded in 2014 and since then has developed many software systems, with focus in web and mobile applications. The offices are situated in Porto, the head office, but also in Leicester, San Diego and Chennai. The company has many partnerships with clients in order to understand their needs and deliver high level software that create an impact in the users and business all over the world. Clients such as TVG, a horse betting platform, and Net-a-Porter, a luxury online store, have been working with Mindera for a long time.

There is also own product development. There are products developed in Mindera such as Statful – a telemetry system to monitor complex and high-performance environments – and Player Index – an application for betting in football around performance of the individual players.

The company works mostly with agile methodologies in a collaborative environment and performance testing and delivering represent an important aspect for the company: “With systems that are constantly evolving and changing, it is important to consider performance testing as part of the continuous delivery pipelines. From our perspective, performance testing is just another aspect of Quality Assurance and needs to be fully automated as part of the integration testing and production deployments” (Mindera, 2018).

Mindera focus on building high performance software systems that can, by their complex nature, be affected by many distinct fault combinations affect their behavior. These systems

need to be resilient and scalable, and other failure-testing solutions beyond the traditional ones are near mandatory.

This thesis will be developed in a project from a client and ethical issues related to professional obligations impose the project data to be treated with confidentiality. In order to address every component of the system, the name used to address the project is C project, from client project.

The C Project is a **Software as a Service** platform that provides to the users a way to distribute and manage their applications across multiple stores.

In order to provide value information to the users, the platform gathers information from many stores and transforms it so that users can find what is happening with their application and make informed decisions. This work is of special interest of the Data Intelligence Team.

1.2 Problem

A business supported by a digital system where people end up relying their lives on should gracefully respond to failures. One way is to introduce failures and realize how people deal with them, measuring time responses, and gathering other data as well to prepare the whole team to real failures in production environment, which, in fact, can be used in the process.

Disaster and recovery plans should be tested and changed, if necessary, which is the purpose of running Game Days. At Etsy (Allspaw, 2012) the steps in a Game Day evolve from imagining a failure scenario and implement what is needed to prevent it from affecting the system, then, cause this scenario to happen.

Whether to introduce failures in test environments, the safest option, or in the live production application, is a matter to be considered carefully. At a first sight, the obvious choice is a testing environment - it is safer and does not bring problems to the business (e.g. Loss of revenue, bad reputation, risk of the scenario to drift apart and cause a real problem). Meanwhile, when differences from the production environment exist, the tests are not completely valid and the confidence in the system is not increased, one of the main purposes.

This thesis practical area will be focused in the improvement of the chaos engineering approaches in a Mindera's client project using fault injection tools and approaches to discover more information about the system behavior and verify that fault tolerant mechanisms work properly.

In this project the results for the two metrics (sophistication and adoption) in the evaluation of the project in the chaos maturity model were the following:

- In **sophistication** the project is in the first level "Elementary". The experiments do not use production environment. The chaos is injected manually to perform simple actions such as, shut down an instance. Only system metrics are considered

- In **adoption,** the chaos tools and methodologies used are also in the first level, “In the shadows”. The chaos experiments are not sanctioned and not frequently adopted. Only few systems are covered and there is a low organization awareness about the experimentation.

Currently, the tools used to introduce chaos into the system are classified into the first levels - both sophistication and adoption of the Chaos Maturity Model. To improve it, a plan to introduce chaos must be developed; this involves running disaster scenarios such as Game Days and the introduction of new programs into the system such as Chaos Monkey.

There are two dimensions in the project that are going to be tested and use these new approaches to verify its resilience.

In the infrastructure, one of the main difficulties is that the underlying machines or virtual machines where services are running may eventually fail. This problem can be overcome by incorporating fault tolerance mechanisms into the system. Nevertheless, they should be proved to work correctly by introducing a failure that causes the machine to fail. In the current architecture, three databases are used to store data related to applications and are crucial to the business. The first is MongoDB, the second ElasticSearch and the third is RiakTS. The main identified issues regarding this area are the following:

- What happens when an application instance is shutdown, does the teams are correctly informed? Is the team notified about it by the system or are the user complains who notify the team?

- What happens when a machine where of the database is running goes down? Will it cause data loss? Does it support data redundancy?

- What if the node is the primary?

In the application level, the interaction between different applications should be tested and the behavior of a system measured when an event such as introduction of an unusual flow of users and latency increase in the network connections between services. The clients (libraries within an application) configuration to establish network connections should be tested and verified that are working properly in the presence of latency and a high number of users. The main issues that need to answered and fixed if needed are:

- What happens when a service is down? Do the calling services respond appropriately?

- How does the applications react to latency introduction in the connections to databases and other services?

- If the connections are all turned off after a service is down, when it returns to active, does the system recover correctly and has a graceful degradation? On the other hand, is it necessary to manually restart the machine?

- Test if the alerts are correctly configured, i.e. if the service is malfunctioning is it identified correctly as unavailable or it goes unnoticed?

These chaos engineering approaches would not be valid without the monitoring metrics of each machine and application with an appropriate monitoring tool. It allows to visualize the status of the system and configure important alerts for some metrics. After the system surpasses the defined thresholds, there is a problem and the operations team will be alerted.

1.3 Objectives

In this thesis, approaches based on Chaos Engineering (CE), will be used to improve a system resilience to failures. The thesis has the following objectives:

- Study and compare different alternatives of testing recovery plans, and measuring failure/repair behavior of components and services;
- Ponder on the implications of using a staging/testing or a production environment to test the plans;
- Study different approaches for chaos engineering and compare available tools;
- Considering different maturity levels, compare alternatives for using chaos engineering methods and tools in testing and/or production environments and delineate a plan to increase the CE sophistication and adoption by introducing chaos tools and events such as Chaos Monkey and Game Days, respectively;
- Simulate different network and system conditions and monitor how the system behaves in order to discover new information. Latency, unavailability and increase of input/output bandwidth usage are some of the conditions that are going to be simulated.

1.4 Methodological Approach

The development process and approach will be under the principles of chaos engineering. Using the chaos maturity model to evaluate the current project's level of chaos engineering as a start point to select improvements in sophistication and adoption of this discipline.

The following approach will be used in the development process:

- Evaluation of a system state and how does in fit in the chaos maturity model;
- Analysis of available tools and choose the ones that fit to the project;

- Definition of experiments to test the system resilience and validate fault tolerance mechanisms;
- Define the normal or steady state of the project using system metrics;
- Run an experiment;
- Compare the test results with the control group;
- Gather solutions and improvements to the resilience and adaptability of the system;
- Implement them.

After some research of the main components of the system, the experiments to carry out were chosen:

- The first, an experiment where infrastructure and the recovery of a database from failure is tested.
- The second experiment is to simulate problems in the connections between services. In the network connections latency and unavailability are going to be simulated to replicate real world events.
- The third experiment aims to measure the impact of physical resources exhaustion in a service.
- At last, the experiments are going to be automated with the introduction of Chaos Lambda. A lambda implementation of Chaos Monkey, that automatically and continuously test the system against instance failures.

1.5 Structure

This document has the following structure:

- Introduction: in this chapter there is an overview of the context and problem, as well as the objectives and the methodological approach that will be used;
- Enterprise Context: here, the project where this thesis will be developed is presented;
- State of the Art: in this chapter, the different topics used in this thesis are going to be described and analyzed. First, the different approaches to test a disaster recovery plan are enumerated and compared. Then, the environment choice is discussed and compared, staging or testing vs production. After, the two main existing approaches are

fully address and is presented a way to design experiments in both. At the end an analysis of existing chaos tools and comparison is done.

- Value Analysis: the innovation process used in this project is explained. After, the value of this project is defined, and the value proposal is described. The quality function deployment is explained as well as the customer requirements and the functional requirements. In the end, the decision about the tools to use in the project is analyzed using the multi-criteria decision-making method TOPSIS.
- Design: the experiments are described. The scope and type of attack, the affected components and how does the faults are injected are explained as well as the expected outcomes.
- Implementation and Evaluation: in this chapter, the experiments are executed. The results are gathered, and, in each experiment, the steady state of the system is compared with the metrics when the experiment is running. The hypotheses are tested and disproved using the difference between the two groups, the control group and the experiment group. If the difference is high, the system is not prepared to handle the failure and needs to be changed.

In addition, there are the annexes:

- Attachment A – QFD: the house of quality of the QFD is presented and the relation between the customer requirements and the functional requirements is identified and evaluated.
- Attachment B – Experiments: additional details about the implementation, such as executed commands and listings used to create the setups and to run the experiments.

2 Enterprise Context

In this chapter, the context of the company is given. First the cloud environment, where the project is deployed is described and some information about different cloud services is presented. After, the architecture of the system and the different components integration are explained. At the end, more information about the monitoring and log system's used is described.

2.1 Target Project

Through the thesis, failures were introduced in a project, the C project, in order to analyze and test its resilience and find existing problems and weaknesses. In this section the architecture of the project is explained to give some context about the components affected, how and where the failures can be introduced.

Resilience can be understood as the "ability of a system to recover from failures and continue to function. It's not about avoiding failures, but responding to failures in a way that avoids downtime or data loss. The goal of resiliency is to return the application to a fully functioning state following a failure" (Wasson et al., 2017).

The client project was created in 2010 and evolved in the last years. The project is a Software as a Service platform that provides the user app publishing to different stores, app management and visualization of data such as download and revenue, rankings and reviews visualization from different stores aggregated in one place, to identify trends and compare the owned apps with other applications to create business decisions to improve the app ranking and popularity.

The project runs in Amazon Web Services (AWS) cloud platform that provides many services and computing machines in order to run the applications and provide the final service to the user. The main concepts that are going to be used in the project are the following:

- **EC2**: elastic cloud compute are the machines provided by AWS and are the basic component of the cloud provider. The type of the machine is customizable and varies with the computational needs. The EC2 instances are the core of the AWS services and where the services are running.

- **ELB**: elastic load balancer is a component that allows redundancy in an application by providing an interface that distributes the traffic to the underlying EC2 instances. The ELB has another responsibility that is performing a health check request to the applications in a given period of time. This request returns a success response and the instances that do not respond to a configured number of health checks are considered unhealthy, removed from the ELB and does not receive traffic.

- **ASG**: auto scaling group is a component that is responsible to manage and scale a collection of EC2 instances with the same configuration. In the configuration, the user provides the minimum, desired and the maximum number of instances. This component also removes instances that are considered unhealthy and add instances to replace them when required.

- **VPC**: virtual private cloud is a logical division of the cloud environment. In the project, there are three different environments and each one has its VPC in order to create a division and limit the access between them and the internet.

In Figure 1 is presented an overview of the components and an example of the request path when a request is performed to the backend area of data intelligence in the project. This area of the project is responsible to provide information about rankings, apps and publishers.

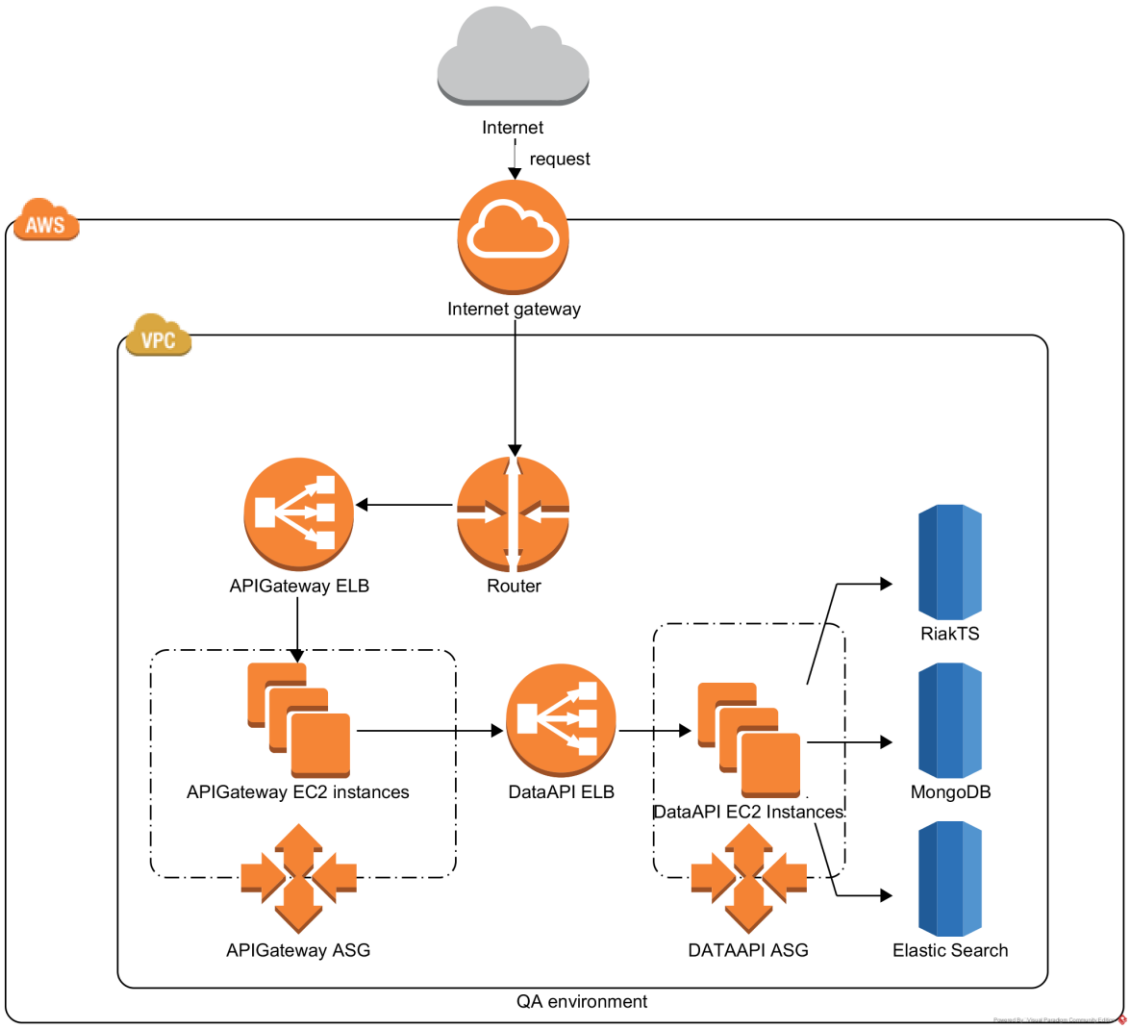

Figure 1 – Deployment Diagram

In a component level, there are three different databases in the project:

- RiakTS, a **time series database** that gather information based on time. It is used in the project to save rankings of mobile applications. A ranking has a position, application identification, location and a timestamp. The aggregation of the information is based on time. A primary key is a composed key of an ID and a timestamp. The structure of the database is a NoSQL optimized to searches in time.

- MongoDB, a **NoSQL database** used to store large amounts of data. The advantages of a document store database increase the performance to search large amounts of data. Indexes are a way to group information based on a characteristic that turns the response to be very fast when an indexed field is queried.

- ElasticSearch, a **distributed, scalable and fast search engine** that gives fast searches to support the needs of low response times when there is a large amount of data to search.

The service that communicates with the front-end application is called APIGateway and it is the entrance to the backend layer. Another important part of the project is the Data API service that provides a HTTP interface to the user experience services that provide information to the user.

UserManagement is a component responsible to manage the user session and authentication. It is important to give context to the APIGateway that does the validation of the authentication in every request using the user management service.

The component responsible for writing data is BatchWriter. It consumes messages from different message queues (where data from different sources is inserted) and writes it into the three different databases.

In Figure 2 is presented an overview of the main components in the project.

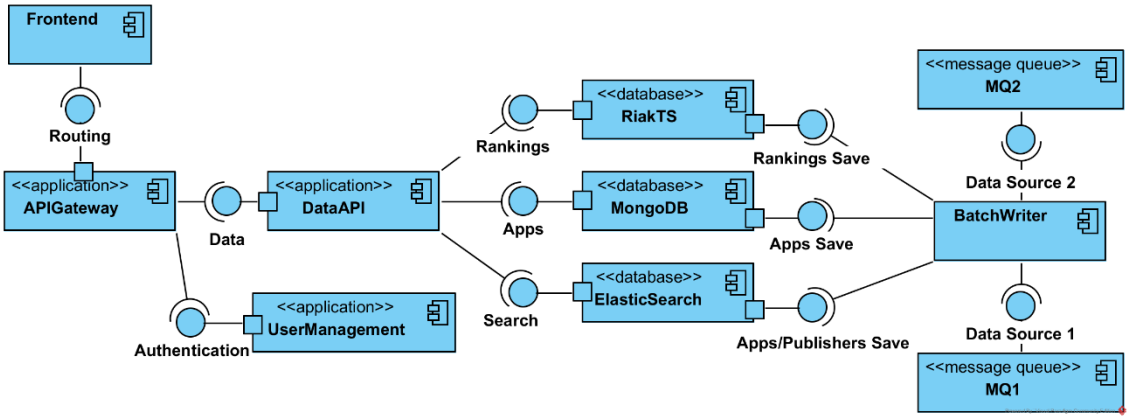

Figure 2 – Component Diagram

In the Figure 2, the data sources play an important role in the system. Other components of the project search for data available in the internet and insert it in two different queues. One related with apps and other with resources.

In terms of fault recovery there is an important aspect in the queue's configuration. To prevent the loss of data and to prevent a bottleneck of messages that cannot be processed, for each existent queue, two more are created. One is the backup queue and the other is the dead letter queue.

After a message has been processed, it is sent to the backup queue. In a given time, if there is a loss of data or some problem in the database that data backups do not cover completely, the BatchWriter can be configured to consume from the backup queue and populate again the database.

The other queue is the dead letter. In order to prevent messages that are not possible to process in the queue and decrease the number of messages that need processing and prevent a bottle neck. After a message is consumed in the SQS, it goes to the "in-flight" mode. After a configured time and if the message is not deleted from the queue, it returns to the original queue and

increases the receive count. In the original queue there is a redrive policy to configure a message's maximum receive count and after the value is hit, the message is sent to the dead letter queue.

2.2 Monitoring

Monitoring (McCaffrey, 2016) is one of the solutions used that provides information about the system behavior at software level (response times, status codes or requests' count) or in a hardware level (CPU usage, free disk space, memory usage and read/write disk operations).

The monitoring has evolved and it is now classified into two different types (Rogers, 2018): **white box monitoring** that is the monitoring of an application on its metrics. Instrumenting the code and knowing what is happening inside an application, metrics such as HTTP Status Codes, throughput, error rates and response times of methods. The logs and tracing also play an important part in the white box monitoring and are very useful for debugging purposes. On the other side, **black box monitoring** that is more related with server's metrics, providing information such as CPU, memory and disk operations.

As the complexity of the system has increased, more difficult and harder it was to monitor the system and to debug problems. Observability is a superset of monitoring that better classifies the current approaches to monitor the behavior of a system and its "about being about to understand how does a system behaves in production" (Sridharan, 2017). The observability has four different components: **monitoring**, **alerting**, **distributed tracing systems** and **log aggregation/analytics**. Monitoring and alerting are a proactive way of identifying failures and wrong behaviors. Tracing systems and logs provide context after the incorrect failure is found and more related to identifying its root cause.

In the project, monitoring is both provided by the Statful and CloudWatch, log aggregation and search is only provided by CloudWatch, alerting is a combination of Statful, CloudWatch and PagerDuty. The last tool is responsible for the management of the alerts.

In this section, two tools are analyzed because they are largely used in Mindera, and one of them (Statful) is an internal product that is used for both black and white monitoring, receiving infrastructure metrics from the servers where the application run and application's metrics from code instrumentation. The other (CloudWatch) is mostly used for white monitoring providing log aggregation and search, but also for black box monitoring with information about every service that belongs to the cloud environment used in the project (e.g. ASG, ELB and SQS information).

2.2.1 Statful

Statful (Statful, 2018) is a telemetry system that gather system metrics from software applications, physical devices and present it to the user so that the current system status and information can easily be accessed and interpreted.

In the architectural level, a time-series database receives data sent by the applications or machine monitoring applications. The system receives data points and each one has a timestamp, tags - that can be used to: identify the application, metric's type, layer of the application (controller, service, repository and others) and every relevant association. This allows the data to be grouped and presented to the user.

After this, graphics can be created and visualized with the relevant information in order to diagnosis problems and verify the correct behavior of the system. Figure 3 presents an example of a dashboard in Statful. The metrics are from June and the IP addresses are omitted but each one is mapped to an instance in the AWS. In the dashboard are four different widgets and each one refers to a different metric in the RiakTS instances. The first one is the average of free memory available in the disk used to store data in all the machines. The second is the CPU idle percentage used to find indirectly how much the database is using. The last two are load and the free space in the disk that holds the data in the different machines available. Other metrics are configured such as disk write/read operations and network bandwidth.

The Figure 3 presents an example of a Statful dashboard showing the RiakTS system metrics.

Figure 3 – Riak TS System Metrics

The example presented in Figure 3, provides information about infrastructure metrics from the servers where RiakTS is running. These metrics provide much information about the current state of the machines that are hold a node of the RiakTS database cluster. The metrics are indicators of what resources the database is using and can be used to find bottlenecks in the infrastructure and configure alerts. E.g. limit the disk free space, so an alert is emitted after that limit is reached and the operation to increase the disk capacity or add a new database node is performed in time, without any data losses and performance decrease.

Applicational metrics are obtained by instrumenting the applications' code to retrieve metrics such as response times, successful/unsuccessful requests and memory usage, and send them to Statful. The metrics are completely programable and every important indicator can be saved in the monitoring system. Statful does not provide any automatic fault-recovery mechanism, just provide information about the overall state of the system and alerting.

2.2.2 CloudWatch

CloudWatch is a monitoring service provided in AWS. This service collects data from both service and infrastructure and provide a way to visualize all the data together in a single platform. The data goes from monitoring metrics to logs and every amazon service provides it as well as the instances deployed within AWS.

The custom instances deployed in AWS have a cloud watch agent configured that sends infrastructure metrics such as CPU, memory and network bandwidth usage. These metrics can be used to visualize the current state of the system and for example: see how many messages are stored in an AWS queue, see how many instances running are in an ASG, see how many requests an ELB received in the last hours. The metrics can be used to reduce the time that takes to identify a problem in the platform.

Logs are another way to find the services' errors that caused a problem to a user. CloudWatch also provides a way to see the application logs from a deployed service. These logs reach the CloudWatch by configuring a cloud watch agent to check the logs that a running application send to a local file and redirect them to the monitoring platform.

Another important aspect of CloudWatch are the metrics can be used to configure auto scaling policies in an ASG to scale up or down instances from a service. The ASG has three different parameters that bound the limits of scaling, minimum, desired and maximum. The scaling policies usually are increase capacity after an average CPU percentage was exceeded or decrease capacity if the percentage is below a decrease limit.

At last, within CloudWatch alerts can be configured in order to alert the development team about failures in the system. The logs are only store in the CloudWatch and this service is used as a complementary monitoring system of Statful.

3 State of the Art

In this chapter, the different approaches to testing a disaster recovery plan and the main differences between the preparation and production environment are described. Then, the approaches of Game Days and Chaos are detailed. At the end, an analysis and evaluation of existing chaos tools is done.

3.1 Testing a Disaster Recovery Plan

There are several reasons (Stechyson, 2015) for testing a disaster recovery plan and data backups. First it increases trust in the current system, after a design, is more likely the system to fail. The plans should be assured that work properly and the team can handle them correctly. The business cannot be damaged - the system should be covered with every important part of the infrastructure having a backup plan. At the end one of the important reasons to have a plan tested is that tests cost less than outages in the real environment.

The resilience (Wasson et al., 2017) in a disaster recover differ from high availability when the problem is wider and usually affects an entire region. It is not possible for the system to recover from failure and manual intervention is needed. Availability is the ability from an application to response and remain healthy, even in presence of problems.

In 2014, Facebook (Sverdlik, 2014) shutdown an entire data center to test his disaster recovery plan. This test was in an entirely new scale of infrastructure testing failures, shutdown an entire region. The service remained online, and the recovery plans proved to be correctly working. The plans had only some failures that engineers could after review and improve.

This is one of the key aspects of this kind of testing. Have the recovery plan tested before a real problem happen in reality and, when the team is not prepared so solve it. It can cause a system

outage when it happens and that will be a real problem. However, in a controlled situation, a failure in the plan can be tackled and solved with more time than in a real scenario. Another aspect of these tests is to embrace failure, Facebook encourages his employees to take risks and do not have problems if a failure occurs. This improves the quality and velocity of the released software.

There are two different approaches (Atchison, 2016) to test disaster recovery plans. One approach of testing into the production environment is Game Days:

- This is the best way to start testing a recovery plan and since it involves a team being ready to solve any issue, there is some guarantee that a problem can be solved.
- It helps with the adoption of failure and increase of the team response time to a future similar failure in production.
- This testing is manual and can address any recovery plan, sometimes is hard to create a tool that runs on a daily basis and automate it to create failures. Sometimes the problem is not something that can be automated. Such as testing the resilience of a database. This cannot be automated because the risks are too high.

Another approach is Chaos Engineering where the system is resilient to a consistent kind of failures such as machine shutdowns. Chaos Monkey is a good example of this:

- The monkey will shut down a random instance replicating a problem that might occur, ensuring that all services built are resilient to machine failures and the system does not have a single point of failure.
- Running it on business hours ensures any problem that rises is tackled by and engineer during its work hours.
- This increases the organization's adoption of failure to the highest levels. Every service that is built must have a fault tolerance mechanism to overcome the failure of a machine.

3.2 Staging vs Production Environment

Test a recovery plan in staging or in a production environment is a question that must be carefully approached.

At a first sight, the main difference is that staging environment is a safer option. Disruptive tests that would raise problems to users in production can be performed without any fear in staging. The environments must be totally decoupled so there is not any shared resource between them. No production service can be using a staging database/service or the other way around.

The problem with using just staging is that usually the test environments are not scaled at the production level. The main reason is the cost saving and those environments do not have the same quantity of users. If there is a way to mimic the production environment and seed the databases with the same quantity of information, it is a better approach of testing and should cover the same failures that could happen in production.

Another problem with using a testing environment is “the existence of any differences in those environments brings uncertainty to the exercise” and “the risk of not recovering has no consequences during testing, which can bring hidden assumptions” (Allspaw, 2012). Since the goal is to increase the confidence in the system, the choice of a test environment might bring some doubts.

Test in production environment (Atchison, 2016) could be foolish to think but, on the other side, there are many advantages that could result from this approach:

- First, the team gets involved. It modifies the behavior of a team to embrace failures and to increase the awareness of the team. When a similar event happens in a normal day, the team already knows how to proceed.

- It increases confidence in the system and it assures that a recovery plan works as expected in production.

- The problems discovered during the testing will be improved before a real and unexpected similar event would happen. An event could occur outside the business hours and the team might not be ready. That would cause serious problems into the live production.

Testing in production has other problems. The first one is that users can be affected by the testing. Data could also be lost and not possible to recover from it without spending much time or it can be lost forever. In addition, a chain reaction of failures in the system - that was not programmed - can result in a long downtime of the services and a loss of money for a company.

In conclusion:

- For testing purposes, environments of testing are a good and safer option. At the end, staging should always be the first approach and then production, only if it is needed.

- If there is not a similar environment to production, using it is the only way to be sure that the recovery plans work as expected. Meanwhile the live environment must be used only if the testing is carefully planned and the upsides of testing are more advantageous than the downsides with losses to the business.

3.3 Game Days

Game Days (Chang & Talwai, 2017) are an approach of testing a recovery plan with the creation of a potentially harmful scenario in a controlled and safer environment. They can be used to discover weaknesses in the system, create useful alerts and fix services in order to respond in failure scenarios.

In Gremlin, Game Days (Kolton, 2017) are used to verify the system as a whole. First in test environments and at last into production. It gives a way to proactively test the system and the team can decide its own terms of engagement.

The first step is to identify what to test. A database shutdown, an entire region failure through the simulation in the cloud provider or the failure of a critical service. These tests need to be carefully planned.

The most important points in a Game Day testing are the following (Kolton, 2017):

- First, the best way is to communicate across the teams. Communication is key. Any problem that surges can be solve soon as possible.

- Starting small and using a test environment is the safer way to start. Test the systems at a functional level. The systems handle errors correctly, what about latency. Then increase the scope and the blast radius.

- Intensify the testing; create new scenarios that test your entire system. What happens if the traffic increases? What if the database is slow, there are timeouts to prevent it from killing other services?

- At last, run the Game Day into production. The live environment is what important to test, is where the customers are, and the revenues comes from.

- One of the most important outcomes from Game Days other than test recovery plans are increase the team ability to solve incidents in production. This is a proactively way to train the team, so it can act more quickly to find solution and with more confidence when a problem surge.

In Etsy, these are the steps to run Game Days (Allspaw, 2012) :

- Imagine an event that can happen and affect your system;

- Study and implement what is needed to prevent it from affecting your users and business;

- Introduce the failures or events simulation into production and verify that your system is resilient to it. Increase the confidence to withstand against these events.

At Stripe, there is a similar approach (Hedlund, 2014):

- Gather the team and imagine scenarios of failure that could occur in the system;
- Have the documentation about the architecture of the system ready and find the existing dependencies between the services and databases of the testing component;
- Find one or more failures that could be injected into the system;
- Record one or more outcomes scenarios that could come from those failures;
- Have data backup and plans to be ready if something goes wrong;
- Inject the failures and watch the results. See if the plans work as expected and save the findings – metrics, bugs, failures and improvements.

At the end of the Game Day, it is important to report what happened. The following answers must be present into the summary (Chang & Talwai, 2017):

- The scenario tested: what needs to be tested and how it will be verified.
- Expected outcomes: the expected behavior of the system or service to the failure.
- Actual outcomes: what actually happened, work as expected and what went wrong.
- Follow-up actions: List of improvements and new plans to solve the existing problems or to address the new ones discovered. It is also good to write down a description of Game Days to do in the future.

Conclusion: To plan a Game Day, the two different approaches of Etsy and Stripe can be combined ending with writing down the conclusions about the experiments. As outcomes of the event, the increase of confidence in the system and the verification of the recovery plans are the most important. The follow up actions are important to change the system in order to increase its resilience.

3.4 Chaos Engineering

Chaos Engineering (Netflix, 2017b) is the “the discipline of experimenting on a distributed system in order to build confidence in the system’s capability to withstand turbulent conditions in production”.

This discipline (Rosenthal et al., 2017) differs from usual testing because the goal is to generate new information about the system behavior instead of testing conditions. This information is related to events in production such as a sudden increase into traffic, race condition between

applications, uncommon combination of events, delays between services and identify how the system reacts in these conditions.

The idea behind Chaos Engineering is to find how the system behaves in those situations and improve its resilience. Experimentation will reveal weaknesses in the system that could cause outages and create problems to the users. These issues can be address and solved in a proactively way without letting it cause chaos in your live system.

There is another important property of the software development that Chaos Engineering helps to improve. Usually the performance, availability and fault tolerance are the optimized properties in the applications. There is a fourth property in Netflix, that is velocity of feature development and its related to how fast a new functionality is implemented and released to the user. Chaos Engineering helps "by supporting high velocity, experimentation, and confidence in teams and systems thought resilience verification" (Rosenthal, Hochstein, Blohowiak, Jones, & Basiri, 2017).

In Google (Beyer, Jones, Petoff, & Murphy, 2016) there is also a sense of finding a balance between maxing out availability and how fast new features are developed. Site reliability engineers find the balance between the benefits of innovation and efficient functionalities, and the possible risk of unavailability in order to increase user's satisfaction.

3.4.1 Necessary Conditions to Perform Chaos

There are two different conditions in the system that are needed in order to perform chaos (Rosenthal et al., 2017):

- The system must be resilience to service failures and network latency. There is no need to introduce chaos engineering if the project already has unsolved issues. After the problems are solved with timeouts, circuit breakers, redundancy or other fault tolerance mechanisms, this discipline can be applied.

- The system must have a monitoring system. The behavior of the services must be observed in order to compare and identify what the chaos cause to the system.

If these two points are checked into the system, chaos engineering is a good way to start the identification of weaknesses and improvement of the resilience.

3.4.2 Principles of Chaos

The principles of chaos are guidelines to create an experiment in order to discover system issues. There are four fundamental steps should be considered (Netflix, 2017b):

- First, one is to start by defining what the normal state of the system is through outputs and metrics analysis. This state is referred as its "steady-state".

- Hypothesize about the status of the experimental group “steady-state” compared with the control group. The assumption is, the experimental group’s state is the same after the chaos introduction.

- Expose the experimental group with the introduction of failures in the system such as machine failures, latency introduction, malformed responses, traffic spikes or others.

- Test the hypothesis and try to disprove it. Compare the results of the experimental group with the control and see the differences.

At last, the harder it gets to mess with the “steady-state” of your system the higher is the confidence in it. If the hypothesis is disproved and weaknesses are uncovered, improvements can be done without any problems caused to the user. These issues could have been at a larger scale and cause serious harm to the business.

There are some advanced principles that must be considered after the chaos introduction experiments is done at a small scale and there is a necessity to increase the scope of the tests.

The advanced principles are the following (Netflix, 2017b):

- **“Hypothesize about a steady state”**: here the usual state of the system must be measured in order to analyze its behavior. Here the system is considered as a whole instead of analyzing metrics such at an application level, metrics such as system errors, response time and downtime are considered.

- **“Vary real-world events”:** Consider variables that reflects an event into the system. Here hardware failures must be considered such as unexpected machine termination, CPU usage in the maximum levels, no memory RAM available (this could be caused by a memory leak) and other real-world events such a spike of users, a mal intentioned user in the system or even an exploit attack.

- **“Run experiments in production”**: In order to increase the confidence into the system and make sure that there is resilience to fault in the real environment, it is preferable to use production.

- **“Automate experiments to run continually**”: this is one important principle that is create automation. Manual testing is an intense and time costly task that is not scalable when there is a need to be applied to several applications or many times.

- **“Minimize the blast radius”**: At last, decrease the affected area of a chaos experiment. In order to use production, customer problems must be avoided. It is not possible to eliminate them. For a short period – if there is a problem discovered in production – users can be affected. The goal of this principle is to affect only a small quantity of users by routing a small percentage of traffic to the experimental group and another to the

control group. This way the hypothesis can be tested with a small amount of real problems.

3.4.3 Design an Experiment

Designing an experiment is similar to the one of a Game Day, since it is also a way of introducing chaos. Here are the steps in order to do it (Rosenthal et al., 2017):

- **First, start from choosing a hypothesis**: Identify an event or problem that can occur into your system. Gather your team and brainstorm scenarios of failure or events that in the past cause many problems to the system.

- **Choose the area of experiment**: it is important to minimize the impact on the users since the ultimate goal is to run the experiment into production, so the risk must be carefully address. First test environments must be used since they are safer and then the experiments start to increase the scope until they end in production. In production start small and increase the scope if needed.

- **Identify the behavior and metrics that are going to be observed:** This step is very important. These metrics are your guarantee that the experiment is being monitored and it is possible to shut down the experiment if there is an unusual spike of errors.

- **Advice your team and organization that there is chaos incoming**: This is important to create awareness in the team, so a situation can be immediately solved. In the initial experiments is better to increase the confidence of the people doing the experiment and reduce the tension.

- **Execute the experiment**: after all the above steps is time to release the chaos. First, the monitoring is very important to see what is happening and if the availability is the same as before. Alerts should be configured in order to reduce the possible problems to the users because a cascading failure can be caused, and it might be necessary to terminate the experiment. The goal is not to cause any problem to the user but to verify the resilience of the system.

- **Interpret and extract information from the results:** This is important to disapprove or verify the hypothesis and see if the system can withstand to the introduced failure or real-world event. This can be considered the most important point of the experiment. Follow up actions and improvements come from the result analysis.

- **Expand the scope of the experiment:** here the area covered by the chaos experiments is increased. First, the experiment starts in a small scale, then it is expanded so that weakness that are only uncovered at a larger scale are discovered.

- **Evolve and automate the experiment**: the experiments are easier to reproduce, and they must run continuously to uncover new fragilities all over the system. This way your system can be considered resilience to the events inserted in the experiments.

3.4.4 Chaos Maturity Model

To formalize the chaos engineering, Netflix has developed a model to classify the state of a chaos program in each organization. This model is called Chaos Maturity Model (CMM) and it gives a methodology that helps to delineate a plan to adopt and improve chaos.

This model is an important component to identify the status of the tools used in the experiments and create a way to improve it. There are two dimensions in this model, sophistication and adoption.

Sophistication is related to the safety and validity of the experiments. There are four different levels in this dimension (Rosenthal et al., 2017):

- **"Elementary":** The production environment is not used to perform the experiments. The process is not automated, and it needs to be run manually. The outcomes of the experience are application metrics and not related to the business. There are only simple events simulated (e.g. manually turn down of a machine).

- **"Simple":** Here the environment used is very similar to the production. The process of setting up the experiment is automated and only the execution and termination needs to be manually performed. Events that are more complex are simulated/introduced, such as, network latency into the tests. The outcomes are manually aggregated in order to compare. The tool provides a historic of data to compare the test and the control groups.

- **"Sophisticated":** In this level, the sophistication of the tool is now very high. The production environment is used. The process of setup, termination and analysis is all automated. Here the metrics are upgraded to a system level: business metrics are used, instead of application level. There is an increase in the complexity of events and now a combination of failures is introduced to measure the impact in the services. The tool provides a way to visualize the information and compare the results between the two different groups of testing.

- **"Advanced":** experiments run in the entire environment and in each step of the service development. Everything is fully automated, from the setup to the termination. The blast radius of the experiment is reduced, using techniques as A/B testing in order to route small percentages of traffic to the experimental group. The events simulated are now more sophisticated. The events are now more complex and simulate real world events such as a change in a normal combination of messages, an unusual number of

retries in a functionality and state or corruption of messages. The business metrics are now interpreted to create business indicators.

Adoption is associated with how much is the awareness that a tool gives to an organization in terms of chaos and how extent and wide are the experiments. Increases the scope of the experiment, exposing more issues and giving more confidence in the system. There are four different levels (Rosenthal et al., 2017):

- **"In the shadows":** there is a low or inexistent awareness about the existing experiments into the organization. Research and new projects around chaos are not incentivized and support. Only a small quantity of systems is addressed. The tests are not frequent.

- **"Investment":** The projects are now supported. There are resources assigned in order to perform chaos. Several teams are engaged and involved in the experiments. Critical services are now tested. Chaos is still not performed actively.

- **"Adoption":** In this level, the adoption is now very high. Teams are fully dedicated to performing experiments in the system. Alerts and operations team responses are integrated with the tests. Critical services are often and actively tested. Game Days are used as a common practice in order to create awareness.

- **"Cultural Expectation":** Chaos engineering is now fully adopted, and the services are built resilient in order to "survive" to the tests. Every critical service is frequently tested, and the other are tested more regularly. Experimentation is part of the development.

3.5 Chaos Tools

In this section, an overview of the existing chaos tools is described. The tools play an important role in automating the introduction of failures into a system, removing the manual steps required to conduct an experiment and increasing the sophistication of the experiments.

3.5.1 Simian Army

Chaos Monkey was the first tool released that randomly shutdown instances in production. The focus of this tool was to prevent single point of failure in the system. The tool cause failures to happen more regularly and to uncover systemic failures in services that cannot handle correctly instance failure. Also, the engineering teams across an organization are aligned to build resilient systems that withstand against machine failures.

After this first success, Netflix (Netflix, 2012) created the Simian Army, a collection of tools that with the purpose of introducing chaos in a cloud environment to test and improve the system's resilience. The Chaos Monkey belongs to the army and his companions are the following:

- Latency monkey was created to introduce latency and increasing the latency to a very high level to simulate instance shutdown. This tool was important to measure how a component react to other dependencies failures without compromising the whole system since the latency were introduced at the upstream of the service. This monkey is now removed from the army since one of the advanced principles in the chaos principles is reduce the blast radius and this tool introduced failures too extensive and could damage the user experience.

- Conformity monkey was built to verify that an instance was following with predefined rules of good practices and send a notification the owner of the instance. This monkey is helpful to keep the standards and prevent misconfigurations that could become harmful.

- Security monkey (Netflix, 2018) is a monkey that monitor the Amazon web services and google cloud platform accounts to prevent insecure configurations and changes in the policies. This tool helps to record every change previously done to present the user what was changed and when it occurred. This is important when managing a micro service architecture system where is hard to be aware of the whole system current state.

- Janitor monkey is a monkey that identify unused resources in the cloud and clean them up. This tool is helpful since the cloud environment provides unlimited resources and is very easy to lose track of them. This tool must be defined with a set of rules and when it identifies an unused resource marks it and schedule a cleanup.

- 10–18 Monkey is a monkey to test application resilience to different languages and charsets that were needed to serve users that vary in language, culture and region.

- Chaos Gorilla is a monkey that simulates the failure of an entire amazon availability zone. This tool was used by Netflix to test if the region failover mechanism was working and could manage to redirect all the requests to another available region without manual intervention and impact to the users.

The only remaining monkeys of the simian army are Chaos Monkey, Conformity Monkey and Janitor Monkey. The others were removed after an update. Latency, Chaos Gorilla, 10-18 monkey are not present in the Simian Army project and not available outside of it. Security monkey is available but as a standalone project.

Simian Army is a project and only are considered the existing Chaos Monkey since it is the only tool that introduces chaos.

About the tool, Table 1 presents some additional details.

Table 1 – Simian Army Details

Characteristic	Value
Cloud Provider	AWS.
Layer	Machine Instance Termination.
Popularity	6185 Stars in GitHub.
Sophistication level	3rd.
Adoption level	3rd.
Open Source (Y/N)	Yes.

3.5.2 Chaos Monkey and Similar Tools

One of the most important monkeys that inspired many other tools was Chaos Monkey. It automatically and continuously test the system against instance failures This tool follows one of the advanced principles of chaos engineering that is "automate the experiments to run continuously". Automated tools are useful since manual testing is a time-consuming task and it is not scalable.

Automation (Lafeldt, 2016) is useful to discover systemic weaknesses and should be used instead of manually shutdown instances. Meanwhile, Game days are also necessary since they gather a team together to discuss failure modes and setup chaos experiments that are good to share different ideas and approaches that are impossible to automate.

3.5.2.1 Chaos Monkey 2.0

In 2016, Chaos Monkey (Netflix, 2017c) was upgraded to the version 2.0 and it was integrated with Spinnaker. This integration made possible the use of this tool in different cloud environments other than AWS. The only downside is that continuous delivery platform Spinnaker must be used in order to run this new version. Also, there were some improvements in the interface and now instance terminations can be tracked by sending metrics that indicate an instance termination to another external system.

The tool is detailed in Table 2.

Table 2 – Chaos Monkey 2.0 Details

Characteristic	Value
Cloud Provider	AWS, Google Compute Engine, Azure, Cloud Foundry or a Kubernetes Environment.
Layer	Machine Instance Termination.
Sophistication level	4th.
Adoption level	4th.
Popularity	3682 Stars in GitHub.
Open Source	Yes.

Requirement	Spinnaker.

3.5.2.2 Chaos Lambda

Chaos Lambda (Shoreditch Ops, 2018) is a server less implementation of the chaos monkey that runs as an AWS lambda to randomly terminate instances within the cloud environment. This implementation is not so complete, less functionalities than the Chaos Monkey and only runs in a small set of regions. On the other side is very easy to deploy, has a small code base and does not need maintenance. Since Chaos Monkey is now integrated with Spinnaker, this implementation can be very useful if this deployment platform dependency is not available in the project.

About the tool, Table 3 shows more details.

Table 3 – Chaos Lambda Details

Characteristic	**Value**
Environment	AWS Lambda
Cloud Provider	AWS
Layer	Machine Instance Termination
Sophistication level	3rd.
Adoption level	3rd.
Popularity	165 Stars in GitHub.
Open Source	Yes.

3.5.2.3 Pumba

Pumba (Gaia Dev Analytics, 2018) is a tool developed to test and introduce random failures in a Docker environment. It can be configured to randomly remove or stop running containers – lightweight virtual instances in Docker – or to simulate internet adverse conditions. This tool is used to continually introduce chaos into the Docker system and see how the system is prepared for these chaotic conditions.

Pumba supports the injection of different failures in the running containers and for a given period of time. The tool has different commands such as kill, pause, stop and remove a container and the possibility to add latency to every request that it receives and to add packet losses. This tool can be easily used to introduce entropy in a docker instance and verify if it is prepared for a production environment.

About the tool, Table 4 presents a more detailed view.

Table 4 – Pumba Details

Characteristic	**Value**

Environment	Docker hosts. Orchestrators that use Docker as the container engine such as Kubernetes or Docker Swarm
Cloud Provider	Generic
Layer	Machine instance termination and network
Sophistication level	3rd.
Adoption level	3rd.
Popularity	781 Stars in GitHub.
Open Source	Yes.

3.5.2.4 Kube Monkey

Kube Monkey (Sobti, 2018) is a chaos monkey implementation for a system running in Kubernetes clusters. It randomly deletes pods - instances in a Kubernetes system – to verify that services are resilient to system failures.

About the tool, Table 5 presents more details.

Table 5 – Kube Monkey Details

Characteristic	**Value**
Environment	Kubernetes
Cloud Provider	Generic
Layer	Machine Instance Termination
Sophistication level	2nd.
Adoption level	2nd.
Popularity	681 Stars in GitHub.
Open Source	Yes.

3.5.2.5 Chaos Lemur

Introduction: This monkey (Hale, 2018) is an adaptation from Chaos Monkey that randomly terminates virtual machines in an environment managed using BOSH.

About the tool, Table 6 show some additional details.

Table 6 – Chaos Lemur Details

Characteristic	**Value**
Environment	CF BOSH
Cloud Provider	Generic
Layer	Machine Instance Termination
Sophistication level	2nd.
Adoption level	2nd.
Popularity	48 Stars in GitHub.

Open Source	Yes.

3.5.2.6 Monkey Ops

Introduction: Monkey Ops (Produban, 2018) is a tool similar to Chaos Monkey that run chaos inside an OpenShift environment.

More details are present in the Table 7.

Table 7 – Monkey Ops Details

Characteristic	Value
Environment	OpenShift
Cloud Provider	Generic
Layer	Machine Instance Termination
Sophistication level	2nd.
Adoption level	2nd.
Popularity	23 Stars in GitHub.
Open Source	Yes.

3.5.2.7 Chaos Dingo

Chaos Dingo (Spring, 2018) is a tool that can be used to bring chaos in Azure services.

About the tool, Table 8 presents additional information.

Table 8 – Chaos Dingo Details

Characteristic	Value
Environment	Generic
Cloud Provider	Azure
Layer	Machine Instance Termination
Sophistication level	2nd.
Adoption level	2nd.
Popularity	8 Stars in GitHub.
Open Source	Yes.

3.5.2.8 PowerfulSeal

Powerful seal (Bloomberg, 2018) is a tool to introduce chaos in a Kubernetes environment. It is similar to Chaos Monkey and can shut down pods in Kubernetes and virtual machines.

The tool has 3 different running modes:

The interactive, that allows the user to discover information about the cluster, manually introduce chaos in order to see what happens to the system. This mode can be used to work on nodes, pods, deployments and namespaces. This mode can be used to acquire knowledge about the system and, after some time, build testing policies.

- The autonomous where the seal reads scenarios introduced in a policy file and starts to introduce chaos in the environment based on it. The policy file has scenarios and each one is composed from matches, filters and actions. The matches are the criteria used to select the chaos targets. The filters are used to filter the selection of matches and add some different criteria such as, work only during business hours. The actions are the fault injections that are going to be introduced.

- The label mode is an alternative more controlled than the autonomous mode. Only the pods and allows the user to specify what pods can be killed, the schedule and the probability to happen.

- The setup is achieved in four different steps: pointing the tool to the Kubernetes cluster using the Kubernetes configuration file, specify the cloud driver and the credentials to create the connection to the cloud, guarantee that the tool has SSH connection to the environment and specify a set of policies.

The Figure 4 shows the setup and the interactions between the different components using the PowerfulSeal.

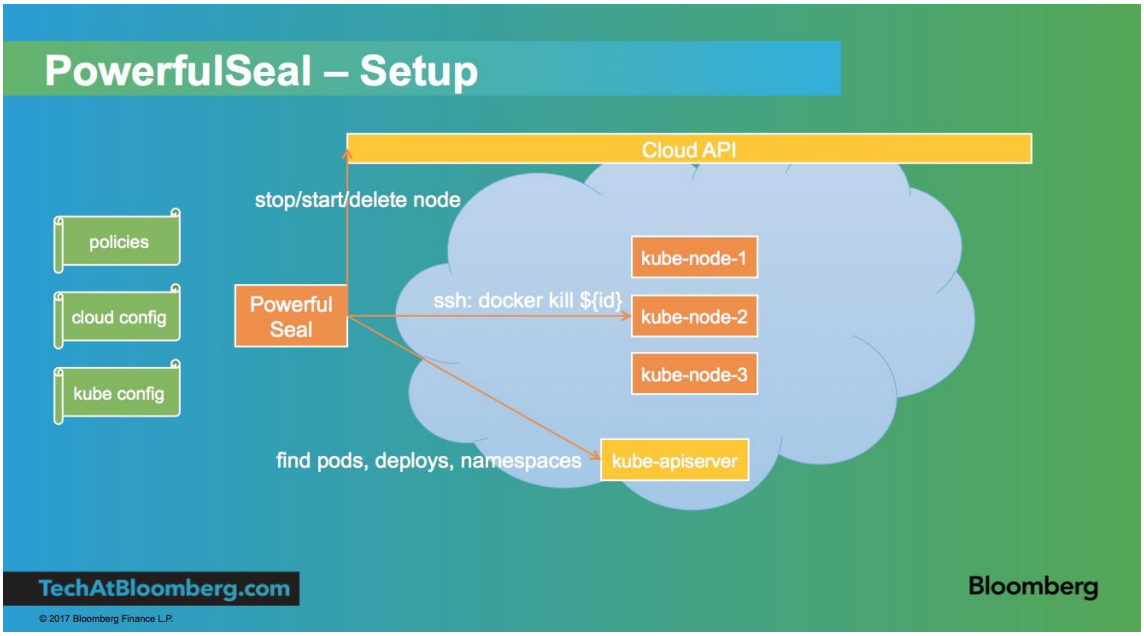

Figure 4 – PowerfulSeal Setup (Bloomberg, 2018)

As described in the Figure 4, the tool can shut down, start and delete nodes. Uses an API to find pods, deploys and namespaces available in the Kubernetes cluster and can SSH to the Kubernetes nodes in order introduce chaos.

About the tool, Table 9 shows some additional information.

Table 9 – PowefulSeal Details

Characteristic	Value
Environment	Kubernetes
Cloud Provider	Generic
Layer	Machine Instance Termination
Sophistication level	3rd.
Adoption level	3rd.
Popularity	519 Stars in GitHub.
Open Source	Yes.

3.5.3 Simoorg

Simoorg (LinkedIn, 2018) is a framework developed by LinkedIn that can be used to introduce failures in a service and log the observations. It is important to find issues and fix them before they happen in the production environment.

This is a framework highly customizable that can be used to schedule failure injection against an application cluster with a mechanism to revert the cluster to a healthy state. This framework has a modular architecture that provides a way to add new plugins to modify the current behavior and add more functionalities needed to test specific requirements of a project.

This tool was built to work in different operating systems and to introduce more failures than simulating instance or hardware failures.

About the tool, more details are presented in Table 10.

Table 10 – Simoorg Details

Characteristic	Value
Environment	Generic
Cloud Provider	Generic
Layer	Customizable Framework to Introduce Chaos
Sophistication level	3rd.
Adoption level	3rd.
Popularity	155 Stars in GitHub.
Open Source	Yes.

3.5.4 Chaos Kong

After the AWS dynamo DB failure in US-EAST-1 region (AWS, 2011) in the Elastic Cloud Compute instances that caused an outage and many customers to lose their services. Netflix (Netflix, 2011) had prepared to similar failure situations and the customers did not notice any problems,

only a higher error rate and latency during the incident. The problem was solved with a manual intervention that involved changing the deployment and configuration, to move set of services to another region. As lessons learned, there was the need automation of region failover and recovery process and create a tool that simulate a total region failure.

The tool to simulate was created and named, first Chaos Gorilla but then renamed to Chaos Kong (Netflix, 2015). It was developed after the success of the Chaos Monkey and to increase the scope of the experiments. Instead of shutting down a single instance, it simulates the failure of an entire AWS region. Netflix uses this tool once a month to simulate exercises of a region outage and identifies systemic weaknesses and proves that region failover mechanisms works correctly and transfers all the traffic to another available region.

Netflix had been prepared for region failure situations using frequently Chaos Kong experiments and after another AWS outage in US-East region occurred (AWS, 2015), the traffic was correctly transferred to another available region and the users did not experienced any problems with the service, as mentioned in the Netflix report (Netflix, 2015).

A more detailed view is presented in Table 11.

Table 11 – Chaos Kong Details

Characteristic	**Value**
Environment	Generic.
Cloud Provider	AWS.
Layer	Region Failure.
Open Source	No.

3.5.5 Blockade

Blockade (Freeman & LaBissoniere, 2018) is a framework to test network failures and cluster partitions in applications running in a Docker environment. This is a typical use case that happens in distributed databases that have a master slave architecture. When the master node has a failure and becomes unhealthy a slave node must be elected master and the cluster should work as usual. This framework can be used to test these situations.

About the tool, Table 12 shows more details.

Table 12 – Blockade Details

Characteristic	**Value**
Environment	Docker
Cloud Provider	Generic
Layer	Network and Cluster Partitions
Sophistication level	2nd.
Adoption level	2nd.
Popularity	522 Stars in GitHub.

Open Source	Yes.

3.5.6 Chaos Proxies

Chaos proxies are tools that allow the user to simulate unexpected network failures in different network protocols. In order to create failure scenarios in a machine, there is the need of root access to it and a good knowledge about operating system. Proxies are a good way to avoid it and easily simulate failures between applications and databases.

3.5.6.1 Chaos HTTP Proxy

Chaos HTTP proxy (Bounce Storage, 2017) is a proxy to introduce failures at HTTP protocol level. This tool supports many failures that can happen when using HTTP such as error responses or server errors, request or response Content-MD5 header's corruption, simulate timeouts, redirects and other failures. It is a simple tool that can be used to provide information about how a service handles network and server failures.

About the tool, Table 13 presents some additional details.

Table 13 – Chaos HTTP Proxy Details

Characteristic	**Value**
Environment	Generic
Cloud Provider	Generic
Layer	Network - HTTP Protocol Chaos
Sophistication level	1st.
Adoption level	1st.
Popularity	114 Stars in GitHub.
Open Source	Yes.

3.5.6.2 Toxy

Toxy (Tomás, 2018) is a more complete HTTP proxy that can be used to simulate unexpected network chaos and failures scenarios to test system resilience. The tool is designed to test the system fault tolerance mechanism in disruption-tolerant networking and service-oriented architectures, where the network connectivity failures happen regularly.

The proxy allows the introduction of poison that intercept the flow of an HTTP request and introduce some failures such as delaying network packets, add some latency jitter, replying with custom error or status code or limit the bandwidth.

The proxy has rules are validation filters that inspect each request and match some request properties such as headers, method, query parameters. The requests with a positive match are injected with the poisons defined to the request path.

Toxy is built on top of rocky API and its provided methods, features and middleware layer can be used to configure the proxy programmatically. The proxy also provides an HTTP interface, to extend and change the proxy in runtime.

The ability to extend completely the proxy, create new poisons and rules provide a simple and powerful way to create failure scenarios and completely test the system fault tolerance mechanisms.

About the tool, some details are presented in Table 14.

Table 14 – Toxy Details

Characteristic	**Value**
Environment	Generic.
Cloud Provider	Generic.
Layer	Network - HTTP protocol chaos.
Sophistication level	2nd.
Adoption level	2nd.
Popularity	2394 Stars in GitHub.
Open Source	Yes.

3.5.6.3 Toxiproxy

Toxiproxy (Shopify, 2014) is a TCP proxy that can be used to simulate different network conditions, such as latency, service unexpected shutdown and can be used to proxy different calls to any service using the TCP protocol.

The Toxiproxy can listen in different ports and send the requests to different upstream and has this advantage in comparison to other proxies available. It is comparable to a server that offers a HTTP interface to add proxies and change its behavior adding toxics, unexpected network conditions that can be introduced in a given proxy, to each request. The toxic can be applied to the upstream or downstream. The upstream is the connection between the proxy and the service that is the next destination. The downstream is applied before the response is returned by the proxy.

The proxy has client libraries that can be used to dynamically add or change the ports that listen and send traffic to the upstream or to add "toxics" to each route. The clients communicate with the proxy using HTTP. Also, there is a command-line application called “toxiproxy-cli” that is installed as a companion and can be used to interact with the tool to check the available proxies, create another one and add new toxics.

Shopify (Eskildsen, 2015) has been using it as part of their development to increase resilience in the system and verify that the system will continue to work when some components are down.

Shopify uses a resilience matrix to both document the relation between a service and a business capability and how the failures in the services affect its functionalities. After the matrix is completed, an integration test is created for each interaction between a service and a business area.

The Toxiproxy connects with the application under test to create unexpected network conditions to verify that fault tolerant mechanisms are correctly configured to handle the failures. The failures simulate a service unavailability, latency in the responses or request timeouts. Another use of the proxy is to have it deployed in a test environment and simulate failures in an environment closer to production.

Timeouts bound the time that a request must take in order to get a response from the service called. After the calls to a service start to timeout, the following requests probability also will not return in time. The successive calls do not return a successful response and add more work to the called service instead of giving some time for it to recover.

Circuit breakers are a way to prevent this situation and configure an error threshold, that after it is reached the circuit opens and does not allow more requests to pass. Usually the call has a configured fallback behavior that returns a message or have another way to obtain a response.

Toxiproxy creates a very simple way to simulate these failures scenarios and to test, for example, a circuit breaker's configuration within an application.

About the tool, Table 15 presents a more detailed view.

Table 15 – Toxiproxy Details

Characteristic	**Value**
Environment	Generic.
Cloud Provider	Generic.
Layer	Network - TCP Protocol Chaos.
Sophistication level	2nd.
Adoption level	2nd.
Popularity	2464 Stars in GitHub.
Open Source	Yes.

3.5.6.4 Vaurien

Vaurien (Mozilla, 2018) is a TCP proxy that allows the user to introduce chaos in the network connections. At the start it is just a proxy that receives data and follow it to a backend service. The tool has different protocols such as TCP, HTTP, Redis and Memcache. The TCP is default

one and it should be enough to delay network connections and simulate the unavailability of a service.

Vaurien work with behaviors that are classes invoked each time that a request goes through the proxy. The available behaviors are used to delay the request, simulate errors and hang without returning any response. The chaos is introduced in the proxy with the manipulation behaviors in a given execution.

The tool brings some built-in protocols and behaviors, meanwhile it is completely extensible and custom implementations of the can be implemented. The tool offers a command line interface that allows to setup the proxy defining different variables. The address where the Vaurien is going to listen, the backend where the traffic is sent and the different behaviors.

In order to control the Vaurien live, the "http" option can be specified, and it will provide a server that runs in localhost on the default port 8080. The interface provided by the server is used to put and get behaviors to the proxy.

More details are presented in Table 16.

Table 16 – Vaurien Details

Characteristic	**Value**
Environment	Generic.
Cloud Provider	Generic.
Layer	Network - TCP protocol chaos.
Sophistication level	1st.
Adoption level	1st.
Popularity	325 Stars in GitHub.
Open Source	Yes.

3.5.7 Failure Injection Testing

Failure Injection Testing (FIT) is "a platform that simplifies creation of failure within our ecosystem with a greater degree of precision for what we fail and who we will impact. FIT also allows us to propagate our failures across the entirety of Netflix in a consistent and controlled manner" (Andrus, Gopalani, & Schmaus, 2014).

Latency monkey was used to introduce latency into the system, meanwhile the blast radius was so high that it could be harm to the system instead of helpful. FIT was built in order to reduce the impact of failures and increase the safety of the experiments.

The FIT service updates the Zuul proxy with failure metadata. The metadata has the failure scope that only apply to targeted requests. After a request match to the failure scope, the request's context is decorated with failure details metadata that travel with it. The Figure 5 presents a FIT experiment.

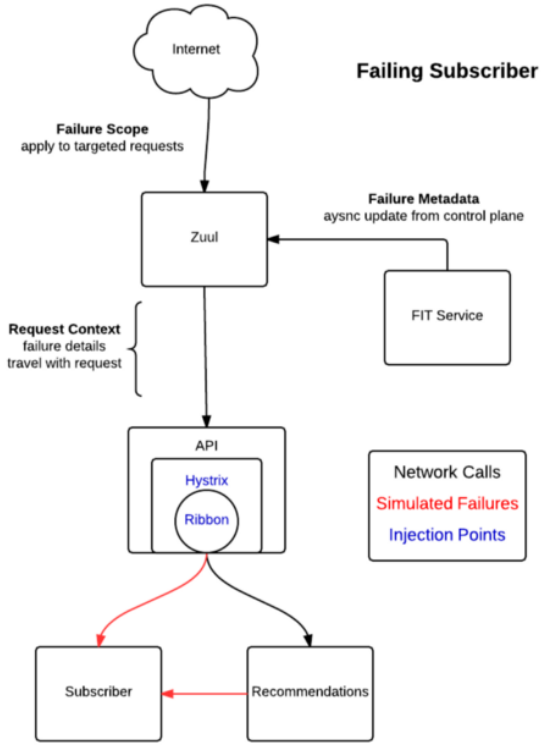

Figure 5 – FIT Experiment (Andrus et al., 2014)

The different steps involved in a FIT experiment are described in Figure 5 as well as where the failures are injected. Netflix uses Hystrix to isolate failures and define callbacks, Ribbon as communication layer to a remote service, EVCache client to access Memcached cached data and Astyanax client to communicate with Cassandra.

In the different libraries enumerated above, are different injection points where failures can be simulated. The services are prepared to receive the details metadata and create different failures. It can be a delay in the service's response, a timeout in the request to a remote service or raise an exception to simulate a server error.

This tool is not open source but describes a way to start exploring the introduction of failures in a controlled way into the system in order to minimize the blast radius of the experiments.

3.5.8 Chaos Automation Platform

ChAP (Netflix, 2017a) is a chaos automation platform built to reduce the blast radius of the test and increase the safety and frequency of the experiments. FIT was used to introduce chaos at a service level but the experiment metrics and the global were mixed. ChAP provides a way to overcome this issue.

Canary analysis is a process that is used to prevent potential harmful releases to be deployed into production environments. It uses a "deployment pattern in which new code is gradually

introduced into production clusters" (McCaffrey, 2016). In the deployment framework (Spinnaker, 2018) this technique can be used where the new code is release and a percentage of the current traffic is redirected to it. The results of the new version are compared against the old version to decide if the deployment should proceed or must be cancelled.

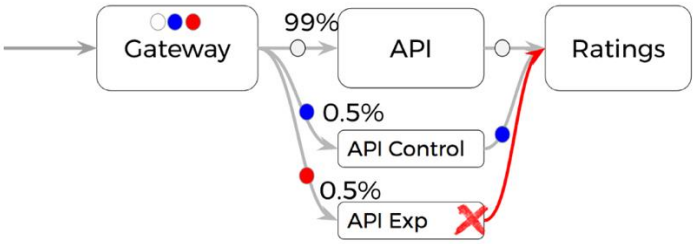

Figure 6 – Example of a ChAP Experiment (Netflix, 2017a)

In the Figure 6 is presented an experiment using ChAP. It uses the technique of canary analysis to perform the experiments. A small percentage of the traffic is routed to the experimental group and an equal one to the control one. After a failure is introduced, the results between the groups are compared. If there is a significant deviation between them, an improvement needs to be performed.

This split in the traffic reduces the impact in the users without affecting the validity of the experiment. It follows the advance principle of chaos engineering that is "Reduce the blast radius". This service was integrated with the continuous deployment platform in order to continually test services and discover weaknesses. Only the non-critical services are currently tested in this way.

3.5.9 Lineage-driven fault injection

The current approaches in chaos engineering are performed using an engineer guided search and using random fault injection. The first one, using the expertise of a team member, deep paths and combinations of events can be found. Meanwhile, it is a time expensive task and is not scalable as the system complexity increases. The random failures can be automated but some combination of events that could happen in production are difficult to discover using this technique.

Lineage-driven fault injection (Alvaro, Rosen, & Hellerstein, 2015) is an approach that considers the data lineage of an application that resulted in a "good" outcome and go backwards to find what failures in the execution could prevent it. The lineage is the computational steps that lead to a given outcome. It is a step to create an automated process that can find the critical paths in an application and test combination of failure events that are missed from the random failure injection.

The LDFI considers that fault tolerance is all about redundancy. The components involved in a given good outcome can fail at any time and have two different states that are running or crashed. An action in the system, for example user login, has a defined path and different steps that go through different services.

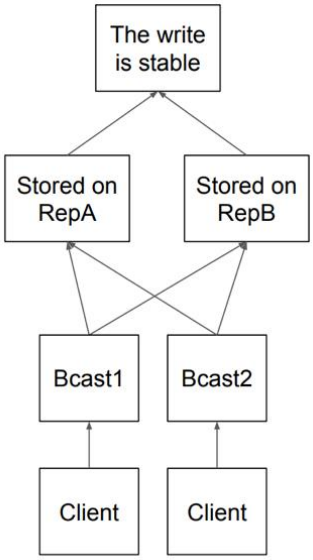

Figure 7 – Example of a System Execution (Alvaro et al., 2016)

A lineage can be converted formula in the conjunctive normal form with the different combinations of failures that could cause the system to fail. Using the example in the Figure 7, the four different components between the client and the stable write outcome could cause the system to fail.

The resulting CNF formula with the failures combinations is $(\mathrm{Bcast1} \wedge RepA) \vee (Bcast2 \wedge RepA) \vee (Bcast1 \wedge RepB) \vee (Bcast2 \wedge RepB)$. There are four services that can fail between the client and the stable write, resulting in a total search space of 16 (2^4) combinations.

Meanwhile, there are some failures combinations that would never affect the system. For example, failing the Bcast1 and the RepA at the same time would not cause any problem, the path Bcast2 to RepB gives another way to perform a stable write. Using a SAT (satisfiability) solver and using the formula in the CNF as input, the combinations of failures is reduced only to the ones that could really impact the system. The search space is dramatically reduced from sixteen to only two failures in the example. The failure of two broadcasts (Bcast1 and Bcast2) or two replicas (RepA and RepB) at the same time could prevent a stable write.

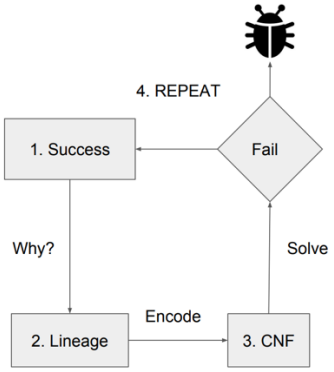

Figure 8 – LDFI Simple Architecture Overview (Alvaro et al., 2016)

The Figure 8 show the LDFI architecture. It starts with a successful pass through the system and tries to figure out what the steps are that make the request successful, the system's lineage. The lineage is encoded into a CNF formula that is solved in order to discover the combinations of failures that would break the flow. The different combinations of failures are injected and if there is a counter example that would prevent the "good" outcome, the program ends and show the lineage to help in the debug. After all the combinations are tested and there are no failures in the outcome, the system is proven to be correct under a specific configuration.

The problem in this approach is the requirement of the program specification in the Dedalus language and that is not reasonable to create it for a large-scale system with a large number of services.

Netflix (Alvaro et al., 2016) has implemented the research prototype LFDI in their systems in order to automate the failure testing in the company. The LDFI was integrated with de FIT tool that was already developed, in order to test the generated experiments from running the LDFI.

The first step that needed to be solved was that measuring a system is hard and sometimes the HTTP status codes are not enough to measure the success or failure in a user functionality. The problem was solved using real user metrics reported by the different devices used to run the Netflix's application. The absence of a message manifested as a timeout is considered an error. Also, when a user does not report a metric, it is considered an error. There is a high probability that the introduced chaos caused the application to crash.

The next step was to find the lineage in a flow in order to discover the different paths and combinations of faults that could be injected. Using the tracing system available annotated with the different injection points, a graph similar to the lineage could be generated.

Another problem was the message replay. It is not possible to replay a method within the system. The state of the data changes and the outcome could not be the same. All the failure testing in the company used the production environment with real traffic. In order to solve it, each request that could cause the same behavior in the back end were treated as if they were the replay of a single request. Creating equivalence classes that represent different interactions with the system and, for each request, predict the class where it belongs solved the problem.

The Netflix test the implementation in the App Boot, a part of the system critical to the business and responsible for the startup of the application. The case study has 100 different services. The resultant search space was 2^{100} that is a number of failures combinations impossible to test. The experiments were reduced to only 200 that resulted in the discovery of 6 critical bugs that could affect and prevent system startup.

3.5.10 Some remarks

There are different tools that can be used in order to introduce chaos into the system to verify and improve the resilience of a system. The layers that each tool operates are different from machine failures to simulate network conditions and application errors.

At a machine failure level, the most relevant and sophisticated tools are Simian Army and Chaos Monkey 2.0. These tools offer a trustful relationship and provide many configurations to the user.

In order to simulate network conditions, proxies of TCP and HTTP can be used. The most relevant proxies in each protocol are Toxiproxy and Toxy, respectively. They can be integrated with other tools and integrated in the development pipeline in order to simulate different error scenarios and network chaotic conditions.

The most sophisticated tools now have a reduced blast radius and run continuously. In order to reach that level, chaos should be introduced at a small scale first and with introduce simple failures. After the chaos automation, the scope should be increased and there is when the more sophisticated tools are used.

In the analysis of the tools, only the open source tools are considered and compared. The project runs in AWS and the tools compatible with the project are the following: Simian Army, Chaos Lambda, Simoorg, Chaos HTTP proxy, Toxy, Toxiproxy and Vaurien.

4 Value Analysis

In this chapter will be presented the value analysis of the thesis. First it will be described the innovation process that resulted in the opportunity identification and analysis, idea generation and selection, and finally the concept definition. Then, the value of the project is analyzed, and it is described the value for the customer. As a result, the value proposal is defined. In the end, the decision about the tools to use in the project is analyzed using the multi-criteria decision-making method TOPSIS.

4.1 Business and Innovation Process

There are three different areas of the innovation process:

- The first one is the **Fuzzy Front End (FEE)** – where the opportunity is identified, some ideas are generated, and a new concept of product is created.
- The second one is the **New Product Development (NPD)** – in this area, a concept is developed, and a new product is built and produced.
- Last one is the **Commercialization** – where the product is promoted and distributed.

There are many formal ways to managing projects that are situated in the NPD but there is a lack of research in the formalization of the Fuzzy Front End.

The New Concept Development (Koen et al., 2002) is a model that helps to formalize it, offering a common language and the key definition of his main components “in order to increase the

value, amount and success probability of “high-profit” concepts entering product development and commercialization”.

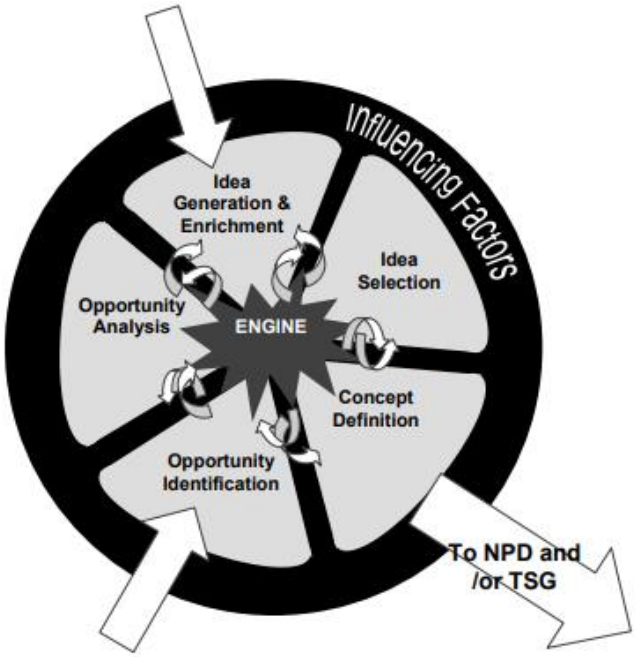

Figure 9 – The NCD model (Koen et al., 2002)

In Figure 9 is presented the NCD model. This model is divided in three different parts:

- The model engine;
- The five elements of the NCD;
- The influencing factors;

4.1.1 Model Engine

The model engine is the power of the five elements of the model. It represents the senior and executive managers’ level of support to keep the process working and it is related to everything that an organization provides leadership, culture and business strategies.

In Mindera, an important aspect of organization culture is to consider change as a good thing. The embrace of change and open mind of the workers helped to gather information about the

existent and failures that can occur in the project as well as find new solutions to improve the resilience.

4.1.2 Five Elements of the NCD Model

There are five different elements in the NCD model and each one will be explored according to the project where the thesis will be developed.

Opportunity Identification

The main opportunity identified in this project is need of increase the resilience of the system and the verification of existing fault tolerant mechanisms and recovery plans.

In the project, there is no defined approach to do experimentation in the system thought failure injection and there are no chaos tools integrated in the development process. In addition, there is no approach to continuously test the system against a type of failures such as machine failures.

The project already has a monitoring system and it is a need to learn how the system reacts to some real events. The new information comes from analyzing the previous state of the system and how it reacted after the entropy creation.

In the identification of those situations and opportunities the following methods, tools and techniques were used:

- Identification of existing problems in the project. With the development of some functionalities during the time in the company, e.g. development of a service that used a client to connect to a database.
- Companies trends to use new approaches and methodologies to introduce chaos in a system;
- Research of existing chaos tools and analysis in order to study what value they could bring to the project;
- Scenario planning – imagine a situation in the future that could affect the project and could cause problems to the users.

Opportunity Analysis

In the opportunity, analysis there were identified several testing approaches and tools used by other companies and what value does this testing can bring to the company.

This opportunity to introduce new ways of testing is valuable because of the improvement of the resilience of the system and there is the development of a culture to embrace failure. Failure

carefully introduced increases the acquisition of new information about the system and improves the preparation of the team to troubleshoot and solve a future problem.

Testing recovery plans bring two improvements to the project. First, one is the resilience improvement. The system is going to handle more failure scenarios. The second one is the increase of confidence in the system.

Idea Generation and Enrichment

There were found two different approaches that can be used in order to perform chaos into the project. Game Days that are an event where a team is gathered and manually run experiments to test the system.

Chaos engineering that is a discipline that contemplates more complex ways of introducing failures into to the system. Many tools can be used to create chaos conditions into the system.

Idea Selection

To address the problem in the project a combination of both approaches was chose. First, the experiments will start with Game Days with some manually introduced chaos. Then those experiments are going to be automated and chaos tools are going to be integrated in the project.

Concept Definition

To improve the resilience of the system at first small tests will be done using the Game Days approach. Failures will be introduced at an infrastructure level to test the resilience of the databases. Then another experimentation using the Game Days will be done but in an application level. The databases will be simulated to be down and the client services that use them will be monitor and test if there is a graceful degradation of them. At last, chaos tools will be integrated in the project and the experiments will be automated.

4.1.3 Influencing factors

The influencing factors “are the corporation’s organizational capabilities, customer and competitor influences, the outside world’s influences, and the depth and strength of enabling sciences and technology” (Koen et al., 2002).

In this thesis the main influencing factor is that “modern distributed systems are simply too large, too heterogeneous, and too dynamic for these classic approaches to software quality to take root” (Alvaro & Tymon, 2017) . So new approaches of testing must be used and one of them is through experimentation.

4.2 Value for the customer

To define the value of a product or service, first the customers must be identified. Mindera is the company where the thesis will be developed and the main interested part in the project. The objective is to increase the resilience in the system of a client by finding new ways of testing recovery plans and fault tolerant mechanisms. This way it is possible to increase the confidence into the system and increase the project's availability.

First the main concepts of value, value for customer and perceived value are going to be explained.

Value: "has been defined in different theoretical contexts as need, desire, interest, standard /criteria, beliefs, attitudes, and preferences" (Nicola, Ferreira, & Ferreira, 2012).

Value for customer: "is any demand-side, personal perception of advantage arising out of a customer's association with an organization's offering, and can occur as reduction in sacrifice; presence of benefit (perceived as either attributes or outcomes); the resultant of any weighed combination of sacrifice and benefit (determined and expressed either rationally or intuitively); or an aggregation, over time, of any or all of these" (Woodall, 2003).

Perceived Value: "different customers perceive different value for the same products /services. In addition, organizations involved in the purchasing process can have different perceptions of customers' value delivery" (Wolfgang Ulaga & Eggert, 2006).

Different values-based drivers have a direct influence in the perception of the value of a product or service to the customer. In the Table 17 presented some of the existing ones.

Table 17 – Value Based Drivers (Lapierre, 2000)

Scope	Benefits	Sacrifices
Product	Alternative solutions Product Quality Product Customization	Price
Service	Responsiveness Flexibility Reliability Technical Competence	Price
Relationship	Supplier's Image Trust Supplier's Solidary with Customers	Time/Effort/Energy Conflict

In Table 18 includes sacrifices such as time and effort associated with the services use and research, and benefits that increase the value of the proposal, according to the project.

Table 18 – Benefits and Sacrifices of the Value Proposal

Domain / Scope	Service	Relationship
Benefits	Responsiveness Flexibility Reliability Technical competence	Image Trust
Sacrifices		Time/effort/energy Conflict

Analyzing the Table 18 the following benefits and sacrifices have a direct influence into the value of this thesis in relation with the improvement of the system resilience:

- Responsiveness: provide fast answers and solutions to problems. Learn from the problem and improve the efficiency of the response.
- Flexibility: increase in the ability to handle change and the velocity of feature development.
- Reliability: improve the resilience of the system and verify that recovery plans work as expected. This increase the availability of the system and turns it more reliable.
- Technical competence: the users in the future will have less problems and increased availability of the system.
- Image: increase of the credibility and reputation with a high available system.
- Trust: the confidence in the system is improved after
- Time/effort/energy: time is needed to implement and discuss the approaches. The approach of Game Days requires a lot of team effort and energy.
- Conflict: the approaches of testing through experimentation could be unpredictable sometimes and carry some temporary issues to the project.

Longitudinal perspective of value

Value for customer has four different temporal positions that are presented in the Figure 10.

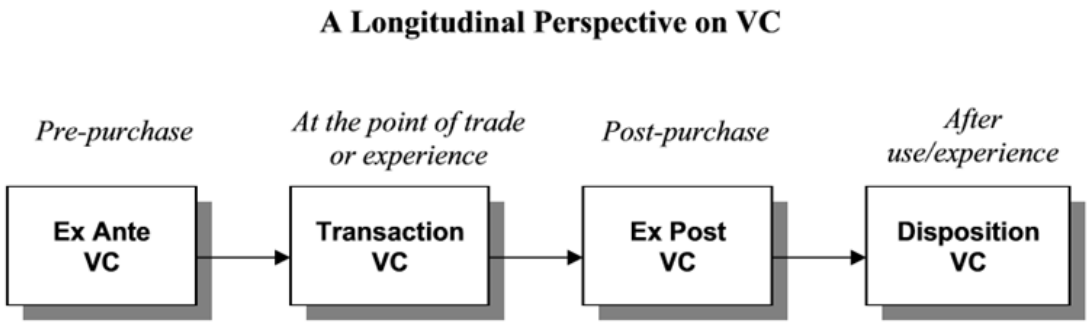

Figure 10 – Longitudinal Perspective of Value for the Customer (Woodall, 2003)

Explaining the Figure 10, the different temporal positions can be used to situate the benefits and sacrifices of the value for customer in the different points (Woodall, 2003):

- **Ex Ante VC:** the pre-purchase position where there is a need to identify where the customer is going to find the value in the product/service. The value that the client desire to receive from it.
- **Transaction VC:** the value experienced at the point of trade or when using the product.
- **Ex Post VC:** post-purchase and acquisition of product and service. Here is where the performance, received and delivered value have impact on the customer.
- **Disposition VC:** after the use or experience.

In this project, in the **Ex Ante VC** the value that the project will give to the customer is a more reliable software with an increase of flexibility of the process development.

In the **Transaction VC** the service will be more technical competent, flexible and responsive. The identification of problems is done in a proactively way that protects the service from having outages and downtime in the future.

After the experience of the service in the **Ex Post VC**, as the confidence of the system and the resilience is improved, the trust and image of the service also improves. The users will experience fewer errors and have a more reliable system.

In the **Disposition VC**. The failure introduction in the system improved the responsiveness and now the teams will provide a better answer to possible problems. The confidence in the system overall is increased.

4.3 Value proposal

The value proposal indicates the set of (tangible) products or services (intangible) that create value for a specific segment of customers (Osterwalder, Pigneur, Bernarda, Smith, & Papadakos, 2014).

Value proposal of this project: Increase the confidence in the system capability to resist in unexpected and turbulent conditions with the introduction of testing through experimentation approaches and resilience improvement.

The main goal of the project is the identification of problems in the system and implement solutions to minimize or solve them. This way, the overall resilience is increased as well as the confidence of the system resistance against failures.

4.3.1 Quality functional deployment

The quality functional deployment ensures that the focus through the development is focused in the customer requirements. In the QFD model, the house of quality, identifies the relation between the user requirements and the functional requirements. The functional requirements are the experiments that are going to be performed to test the system, the analysis of the results and identification of problems, and the solutions that are going to be implemented.

The current project situation and project after the experiments are compared in order to compare what the project should look like after. The identification of problems and integration of new chaos tools to increase the verification of the system. The solutions to minimize the impact and solve the problems increase the system resilience to failure.

Figure 31 presents the QFD house of quality. The customer requirements are the following:

- Improve the flexibility of the service: the service handles well change and provides new features more frequently.
- Improve the team responsiveness: the team is more responsive to problems and more easily find solutions.
- Improve the confidence in the system: the confidence in the project is increased.
- Discover problems and weaknesses in the system: one of the two most important points. The main goal of the experiments is to find problems and weaknesses in the system and provide a way to improve the resilience.
- Improve the system resilience: After the problems and weaknesses are discovered, new solutions to minimize or improve the problems are implemented.

The functional requirements are the following:

- Test the resilience to a database primary node failure: it aims to test the resilience of the system to database failures.

- Measure the impact of latency in a service: test the application behavior when the latency in the network connections increases.

- Measure the impact of resource exhaustion in a service: test the behavior of the system when a server goes out of resources such as memory and CPU.

- Test the global resilience in the project using the Chaos Lambda tool: continuously and randomly, test the system against machine failures.

- Analyze the results and identify possible problems or weaknesses: Important to discover problems and weaknesses. The outcomes of each experiment are analyzed and interpreted, and follow-up actions are created.

- Implement solutions to improve the system's resilience: as a final goal, the implementation of solutions to improve the resilience against failures and unexpected conditions.

The aim is to improve the project in all the customer requirements. The relation between results' analysis and the improvement of the resilience is very high. There is a high correlation between them. The Game Days and the Chaos Monkey are experiments and the results created need to be analyzed.

4.4 Multi-Criteria Decision Making

In order to decide the most valuable chaos tool the **Technique of the Order Preference by Similarity to Ideal Solution** method was applied. TOPSIS (Nicola, 2018) is a multi-criteria decision making method that considers three different aspects in order to measure what is the ideal alternative:

- Attributes or criteria of quality that are benefits to the ideal solution;

- Attributes or criteria of quantity that are benefits to the ideal solution (more is better);

- Attributes or criteria that are costs and disadvantages (more is worst);

There are two different hypotheses that are used in order to find the ideal solution:

- The ideal solution – the one with the best values for the different attributes in the decision;

- The negative ideal solution – the one with the worst values.

The different attributes and criteria considered to analyze the tools were the following:

- Reliability – measured by analyzing the popularity of each GitHub repository. This helps to find the most trustful to use.
- Sophistication – the level of the tool in the CMM. This attribute indicates which experiments are safer and better.
- Adoption – the level in the CMM that indicates the adoption of the tool within organization. It is related with how many systems are covered and what is the use and awareness of the experiments.
- Compatibility – a binary value that indicates if the tool is compatible with the project.

All those attributes mentioned above are a benefit to the ideal solution. The weights of the attributes/criteria are presented in Table 19.

Table 19 – Weights of the Attributes/Criteria

Attribute/Criteria	**Reliability**	**Sophistication**	**Adoption**	**Compatibility**
Weight	0.25	0.15	0.1	0.5

In the Table 19 are described the different weights used to measure the ideal alternative. The preference was reliability > sophistication > adoption and the compatibly with the highest value and an influence of 50 % in the result.

In the measurement of the weights the following criteria was used:

- Reliability is preferred to sophistication and adoption, so the tool does not cause any harm to the project.
- Sophistication has more value than adoption. A tool that has more valid experiments and safer is better than a widely adopted.
- Compatibility is a necessary condition to apply the tool in the project. The value of 50 % is to influence the result to bring more value to a compatible program.

Another component of the TOPSIS are the different alternatives to study. The final goal is to find the closest alternative to the ideal solution. The evaluation of the Chaos tools are presented in the Table 20.

Table 20 – Chaos Tools' Evaluation

Weights	**0.25**	**0.15**	**0.10**	**0.5**
	Reliability	Sophistication	Adoption	Compatibility
Simian Army	6185	3	3	1

Chaos Monkey 2.0	3682	4	4	0
Chaos Lambda	165	3	3	1
Pumba	781	3	3	0
Kube Monkey	681	2	2	0
Chaos Lemur	48	2	2	0
Monkey Ops	23	2	2	0
Chaos Dingo	8	2	2	0
Powerful Seal	519	3	3	0
Simoorg	155	3	3	1
Blockade	522	2	2	0
Chaos HTTP Proxy	114	1	1	1
Toxy	2394	2	2	1
Toxiproxy	2464	2	2	1
Vaurien	325	1	1	1

In the Table 20 are presented the different evaluation of the tools (alternatives) within the different attributes and criteria. The values were obtained after a careful analysis of each tool and the following criteria was applied:

- In the reliability were considered the stars in the GitHub repository. The stars mean the people that follows and are interested in the project.
- Every tool was evaluated within the CMM in terms of sophistication and adoption. There are four different levels in the model from the 1st to the 4th level.
- The compatibility is 1 if the chaos tool can be used in the project and 0 if it is not possible.

The Table 21, presents the relativeness closeness to the ideal solution and the ranking of the chaos tools.

Table 21 – Relativeness Closeness to the Ideal Solution

Chaos Tool	Score	Rank
Simian Army	0.93	1.0
Toxiproxy	0.63	2.0
Toxy	0.62	3.0
Chaos Lambda	0.51	4.0
Simoorg	0.51	5.0
Vaurien	0.50	6.0
Chaos HTTP Proxy	0.49	7.0
Chaos Monkey 2.0	0.38	8.0
Pumba	0.15	9.0
Powerful Seal	0.14	10.0

Kube Monkey	0.09	11.0
Blockade	0.08	12.0
Chaos Lemur	0.07	13.0
Monkey Ops	0.07	14.5
Chaos Dingo	0.07	14.5

In the Table 21 are the relativeness closeness to the ideal solution of the tools. These are the results of the evaluation using the TOPSIS method. Analyzing the results, there are 5 different tools that are going to be considered in first place to introduce chaos in the project. Simian Army, Toxiproxy, Toxy, Chaos Lambda and Simoorg. These are the best tools in terms of reliability, sophistication, adoption and compatibility.

5 Design

In this chapter, the experiments conducted to test the resilience of the system are explained, as well as the points where the failures were injected. These experimentations were designed considering the Maturity evaluated in the section 1.2, both sophistication and adoption were in the first level. To increase adoption and sophistication, experiments were performed and two chaos tools were introduced: Toxiproxy to simulate unstable network conditions and Chaos Lambda to globally test the resilience of the system against instance failure.

As point of start in the project to test the system resilience and as mentioned in the 3.4.3 section, the designed experiments are performed in a test environment and with a small scope. After the confidence in the test increases, the production environment should be used, and the scope increased.

5.1 Local Resilience Experiments

The experiments were designed to introduce failures in the different components of the project to discover weaknesses and improvement possibilities. Thus, the experimentations were as follows:

- The first, described in section 5.1.1, is an experiment where the recovery of the system from the failure of the primary node of MongoDB is tested.

- The second experiment (see section 5.1.2) is to simulate problems in the connections between services. In the network connections latency and unavailability are going to be simulated to replicate real world events, that is an advanced principle of chaos as described in section 3.4.2.

- The third experiment, which is detailed in section 5.1.3, aims to measure the impact of physical resources exhaustion in a service.

- At last, the experiments were automated with the introduction of Chaos Monkey (see section 5.2).

5.1.1 Database Systems Resilience

A machine eventually is going to fail. This chaos experiment aims to prepare the system to a database instance failure. This test is going to be performed in the testing environment and the results are going to increase the confidence regarding this problem and to discard some misconfiguration in the database. The Table 22 presents additional details about the experiment.

Table 22 – Database Systems Resilience Test

Hypothesis	MongoBD and RiakTS are resilient to machine failure, the fault tolerance mechanisms configurations provide the guarantee that at least one machine can go down. Alerts need to be correctly configured and after a machine failure an alert should be triggered to notify the team.
Attack	Instance failure.
Scope	Single instance.
Important metrics	Available instances and number of successful responses.
Expected results	There are no data losses in the database. The steady state of the system changes, with some instances from the database receiving more traffic. An alert is triggered after the database node is turned down. After the recovery, the system is expected to be back to normal. If the primary node of the database is down, a replica should be promoted to primary.
Setup	This experience will be performed in the QA environment. This environment already has a MongoDB replica set deployed and running. The fault tolerance mechanism of this database is going to be tested.

In this environment MongoDB architecture is a replica set that consists in a group of Mongo processes that hold the same data set. Replication (MongoDB, 2018c) provides, in addition to load balancing, redundancy and availability of the data. Since every process that belongs to a replica set holds the same data, if one instance where the mongo process is running is shutdown, the database continues running without data losses and downtime.

The architecture of the database is the same in production and QA environment. Meanwhile in QA there is only one replica set. In Figure 11 is presented the MongoDB replica set architecture.

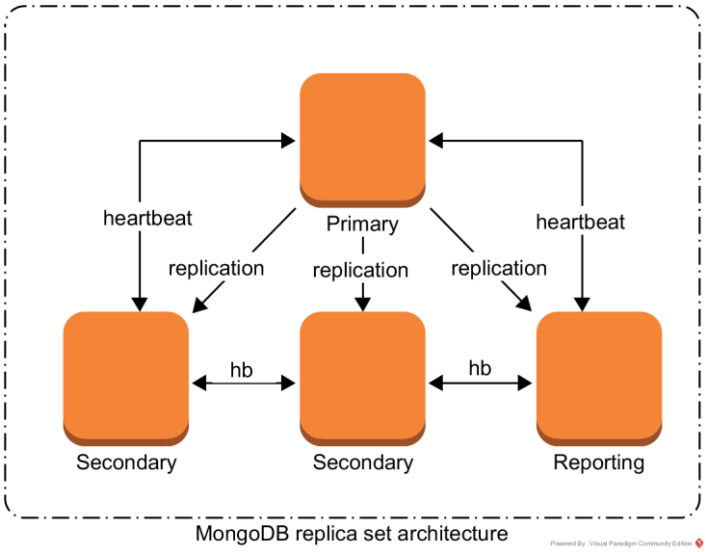

Figure 11 – MongoDB Replica Set Architecture

In the Figure 11 deployment diagram, there are 2 nodes that are secondary, 1 node responsible for reporting metrics and a primary node. The latter receives read and write operations and replicates the write ones to all the other nodes. Without a primary node, write operations cannot be performed. This node is similar to those that are secondary but was elected as the principal.

A secondary node holds the same data set as the primary but can only receive read operations and has vote and priority values equal to 1. The votes are necessary for the primary node elections and the priority is the eligibility of a node. In the elections, the highest priority node becomes the primary.

The reporting node holds the same dataset as the other secondary nodes. It has 0 priority and cannot be elected as primary or start an election. Meanwhile, this node participates in the elections with also 1 vote. The responsibility of this node is to report statistics.

According to the MongoDB documentation (MongoDB, 2018b), the number of instances that can be shut down in a four-membered replica set is one at a time. With 4 members, the majority needed for an eligible node to win an election is three, which means that if more than one instance is shutdown, this majority will never be achieved.

In Figure 12 is presented where the fault is introduced and the fault tolerance mechanism that exists in MongoDB.

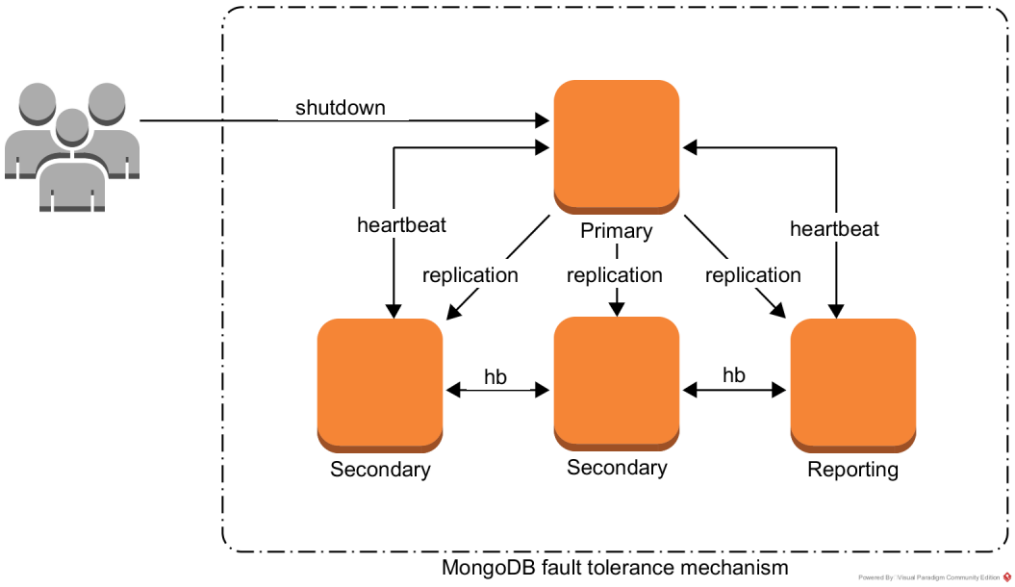

Figure 12 – MongoDB Fault Tolerance Mechanism

In this experiment, the primary node is going to be manually shutdown. In the Figure 12, the fault tolerance mechanism main concepts.

The first concept is the replication. Operations that modify the data present in the database are recorded in a collection called Oplog – operations log – that keeps a record of the latest operations in the database. Only the primary node can receive write operations so the primary's Oplog have the current state of the database. The other nodes copy the primary Oplog and apply those operations maintaining the same dataset.

The other concept is a heartbeat. Replica set members send heartbeats to each other every 2 seconds and if the ping does not have a response within 10 seconds, it is marked as unreachable. After the primary node is marked as unreachable a process called automatic failover is triggered and a node that can be elected starts the election to become the primary. During the elections the database does not provide write operations since there is no primary. The mean time to complete an election should not exceed 12 seconds.

The Figure 13 present the state of the replica set after the primary node is shut down and after the elections with a secondary node winning the election. This is the scenario that is going to be tested in order to make sure that the fault tolerance mechanism work as expected.

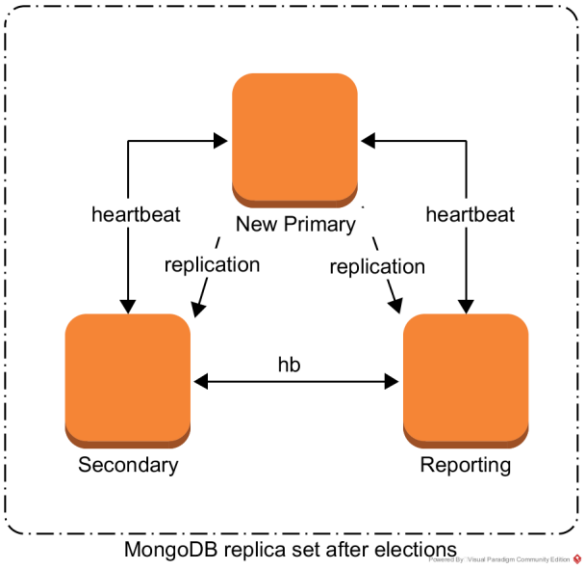

Figure 13 – MongoDB Replica Set After Elections

After the shutdown of a node, number of members of the replica set is three and from that moment, the database is no longer tolerant of instance failures. A node needs the majority of the votes (also known as quorum) to be elected as primary node and, in a replica set with 4 members, 3 votes are needed. In the Figure 13 there are only 3 members left, after another member is shutdown down, it is not possible to elect a new primary node. Moreover, without a primary, the replica set cannot perform write operations so, it is not fault tolerant to instance failure at this point. Meanwhile, read operations are still possible and the dataset is saved. Even if more nodes are disabled, the data is safe.

Finally, the old primary will be turned on and it should return to the replica set. The node will copy the last missing operations from the primary's Oplog, apply them and everything will return to normal.

5.1.2 Unreliable Network Connection

In the modern software development, with increase of distributed systems and more common micro-service architectures, network connections are highly need as they ensure communications between services. This chaos experiment aims to simulate poor network conditions between different services in order to see how the system responds when latency is increased and when the unavailability is simulated. The main idea is to test clients' connection to services and databases in order to, for example, adjust the timeouts in the applications if there is a window to improvements, but also identify misconfigurations or application's resilience problems.

The problem with using just timeouts is that it can put too much backpressure in other components. Too much calls to another service do not give it time to recover and eventually will break it.

There are different alternatives to solve this problem. The first one needs a circuit breaker. Instead of keep calling a service that returns timeouts or reject connections, there is an error threshold that after hit, the circuit break will open and do not allow more requests to a given service. This way, the other service can recover and after some configured time, the circuit should close.

In order to simulate a huge consumption of input/output bandwidth, an unusual flow of users can be introduced. This test is aim also to test if the queries to the database are optimized and if there are indices to support them. In this experiment the DataAPI connection to other databases are going to be tested. Every other service that are connected using network could also be tested. The Table 23 has some additional information about the experiment.

Table 23 – Unreliable Network Connection Description

Hypothesis	The application is prepared to handle latency, unavailability and database Input/O bandwidth.
Attack	Unavailability, latency and I/O.
Scope	Single instance.
Important Metrics to Watch	Response time and number of successful responses.
Expected Results	After the database unavailability is simulated and application business logic in test flows through them, server errors are expected. After the latency is increased, traffic is reduced and slower. The timeout limits are tested and may be tuned if necessary. As the latency increases the number of users that receive successful responses decreases. At some point the service reject more requests.
Setup	In this experiment, the TCP proxy Toxiproxy was introduced in order to simulate chaos in the different databases. The proxy is going to be deployed into a local development environment and different conditions such as unavailability and latency are going to be introduced. There is a proxy for each database and in each one, chaotic network conditions are going to be simulated. This test aims to test the resilience of DataAPI, a service that consumes data from three different databases.

The service Data API is connected to the three different databases and consume from each one of them. The Figure 14 shows what is the setup of the Data API without the integration of the Toxiproxy.

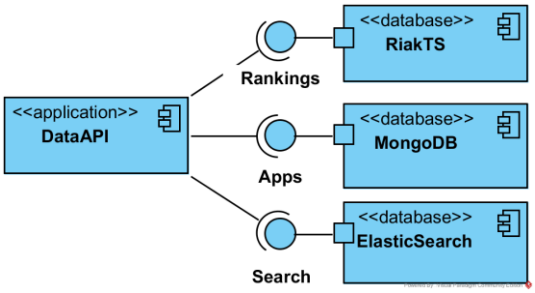

Figure 14 – Data API Setup

The databases are responsible for different use cases in the project. ElasticSearch provides a fast way to search data, RiakTs a fast way to retrieve rankings data and MongoDB has all the data related with Apps. To show the differences between the setups, the Figure 15 shows where the chaos is introduced and how the Toxiproxy does connects to the databases and DataAPI. The setup is simple, and the service has three different clients that connect to each existing database.

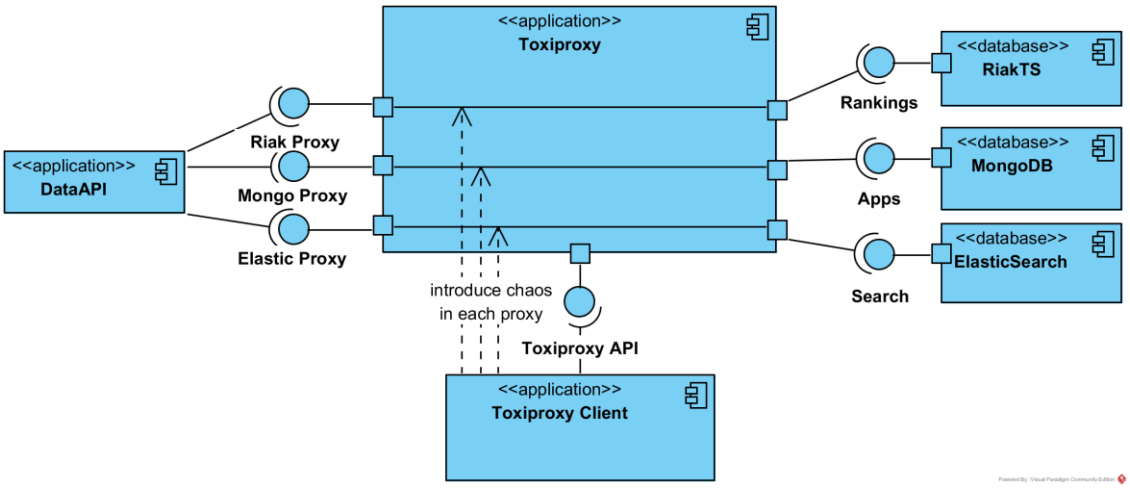

Figure 15 – Toxiproxy Integration

Toxiproxy has an HTTP interface to insert proxies, which are represented by the following properties: the name of the proxy, the host and port where the proxy is going to listen and the upstream where the traffic should flow. The interface has another resource to introduce chaos in the proxies called toxics. It is used to introduce latency, simulate unavailability and other unreliable network conditions.

This experiment requires three different databases, one application, Toxiproxy and one load testing tool. The load testing tool is important to simulate users and to discover how the chaos affects them.

At first, the steady state of the system is defined using the system without any chaos introduced. This is the control group of this experiment and is going to be compared with the results of the experiments performed.

In order to create the setup of this environment the chosen tool was Docker Compose. Compose is a tool to running multiple containers. It consists in the YAML file to define the configuration for the different applications. After the configuration creation and with the command “docker compose up” the setup is created.

More important, Compose simulates an isolated environment similar to the ones that exists in the QA and production environment. The differences are in the network connection that in the local environment does not have any latency and components that exist only in the AWS cloud that provide redundancy. The simulated environment is presented in the Figure 16.

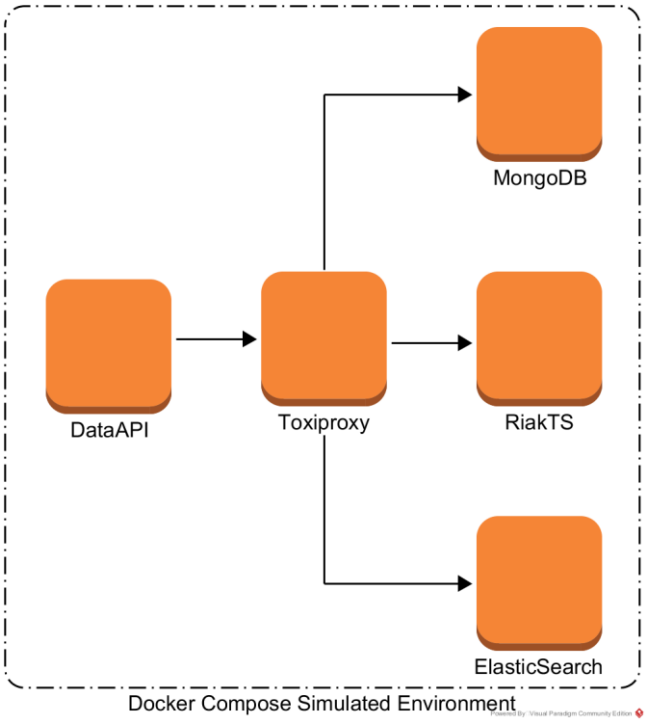

Figure 16 – Docker Compose Simulated Enviroment

Analyzing the Figure 16, in comparison with QA and production, the simulated environment in this experience does not have an ELB to distribute load for each available DataAPI instance and ASG responsible for scale up and down the number of instances. However, the development environment is sufficient to begin to identify the impact of latency and unavailability on a running application.

5.1.3 Resource Exhaustion

Resources eventually hit their limit and an application is forced to run under this situation. As the users' flow increases, more CPU, memory and disk are needed to fulfil the requirements. It is possible to scale horizontally or vertically in order to prevent resource exhaustion. This is a simulation of the resource exhaustion in a chaos experiment to see how the system behaves. About the experiment, the Table 24 shows a more detailed description.

Table 24 – Resource Exhaustion Description

Hypothesis	The system considers the instance where this test is applied as unhealthy. The traffic is routed to another instance by the elastic load balancer and the system is protected against single instance resource exhaustion
Attack	CPU, memory and disk
Scope	Single instance
Important Metrics to Watch	CPU, memory and disk usage. Response times and number of successful responses.
Expected Results	The response times become higher, there is an error increase and a reduction of successful responses. After the instance is recovered, the system state should return to normal
Setup	The QA environment is going to be used in the experiment. The Store Service is going to be the service under test. In the production environment the minimum number of instances running a critical service is three. This provides redundancy and the system becomes fault tolerant to two instances shutdown

The number of instances is three, as described in Figure 17, so after two instances are down, the elastic load balancer redirects all traffic to one instance and the system does not become unavailable. Watchlist Service has three different instances in production to allow redundancy. Before the experiment, the service in the QA environment is going to be scaled to three instances, so the tests become closer to production.

A fork bomb is a denial-of-service attack where a process creates continually new processes to completely exhaust the resources in a machine. In order to perform this test, a fork bomb is going to be used to exhaust the machine resources and turn it unreachable. At first the system is slowed down until the resources to run an application are unavailable and then it becomes unresponsive.

The elastic load balancer performs a health check to every service instance. The current health check configuration in the watchlist service ELB is presented in the Table 25.

Table 25 – Watchlist Service ELB Configuration

Variable	Value
Timeout	15 seconds
Interval	30 seconds
Unhealthy Threshold	5
Healthy Threshold	2

Analyzing the Table 25, the configuration means that after 5 consecutive times that a health check fails, the instance is considered unhealthy, removed from the ELB and does not receive more traffic. Then the auto scaling group removes the instance and launch a new one to replace it. The health check is an HTTP request to the URI /health and a response with a status code of 200 is considered a success.

The Figure 17 shows the current setup of the watchlist service and how does the chaos is going to be introduced.

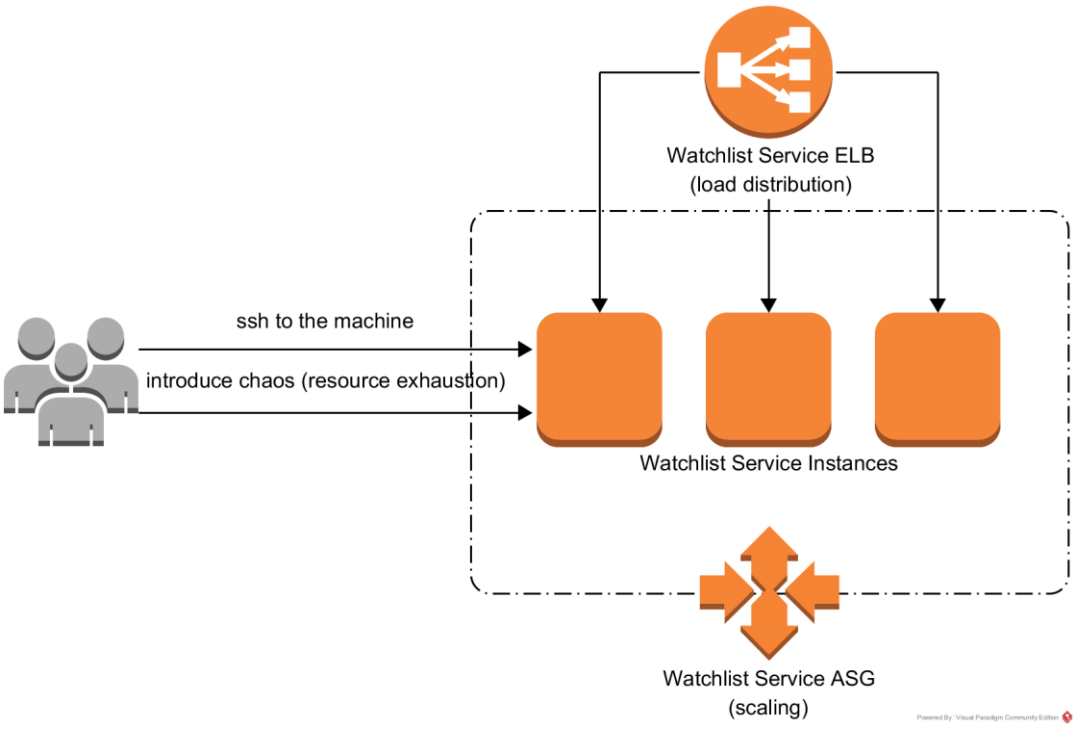

Figure 17 – Watchlist Service Setup

Using the load test tool Vegeta (Senart, 2018) to simulate users, this test aims to find out what the impact of resource depletion on the system is. The number of successful responses and the response time will be monitored. The time an instance takes to become unhealthy and to be replaced is going to be measured.

5.2 Global Resilience Experiment

In this integrative experiment, Chaos Monkey was implanted for automation of instance failure. Chaos Monkey randomly chooses a running instance and turns it off. Making the occurrence of failure more frequent is beneficial to the system. All services deployed after tool execution must be fault tolerant for instance's failure.

A new service that does not handle this failure will cause the system to fail and an alert to be triggered. The monkey only works during business hours, so the problem can be solved immediately by the working teams. The Table 26 presents a detailed description of the experiment.

Table 26 – Global Resilence Experiment Description

Hypothesis	The entire system is resilient to instances unexpected shutdown.
Attack	Instance failure.
Scope	Multiple instances.
Important metrics to Watch	Instances available and number of successful responses.
Expected Results	The steady state of the system does not change. Some instances start receiving more traffic. An alert is triggered after the machine is turned down. After the recovery, the system is expected to be back to normal.
Setup	In this experiment, the chaos monkey that is going to be used is the Chaos Lambda. It will target different services, but databases are out of the scope of this experiment.

Services are stateless, and an instance failure does not cause any harm to the system. There is no data to be lost and if there is an identified problem it can be solved without causing any relevant problem. Databases are stateful and are going to be target in manual experiments where the actions can be taken, and the experiments run knowing of what can possibly happen. Running the chaos monkey against databases can cause problems and that is not the purpose of chaos engineering. In this experiment, only the stateless services are going to be tested to avoid causing irreversible failures.

Chaos Lambda has a serverless architecture and it is very easy to implement in comparison to Simian Army's Chaos Monkey. This monkey runs in a lambda function and target different auto scaling groups and identify its instances. There is a probability that decides if an instance is shutdown in a given execution.

A CloudWatch Event is launched in a configured schedule that triggers the execution of the function. These two components, events and function are the only components necessary in order to deploy chaos lambda.

Chaos Lambda is a node package that already has the deployment methods created and the only necessary steps is to install it in a local machine, define a lambda role in AWS, setup the credentials, create a configuration file and then run it. The role is a set of permissions that are associated with the lambda function and allow it to terminate instances. The credentials are needed in order to run the methods that deploy the chaos lambda function to the AWS.

The creation of a configuration file is necessary in order to define the frequency that the chaos lambda runs and what are the targeted auto scaling groups. The configuration file consists in a JSON file with three fields: “interval”, “enableForASGs” and “disableForASGs”. The interval in minutes that is the frequency which the function is called, the enable field is the white list of auto scaling groups that are going to be targeted and the disable is the black list of ASG that should not be touched by the tool. The Figure 18 presents how does chaos monkey is going to be integrated in the project.

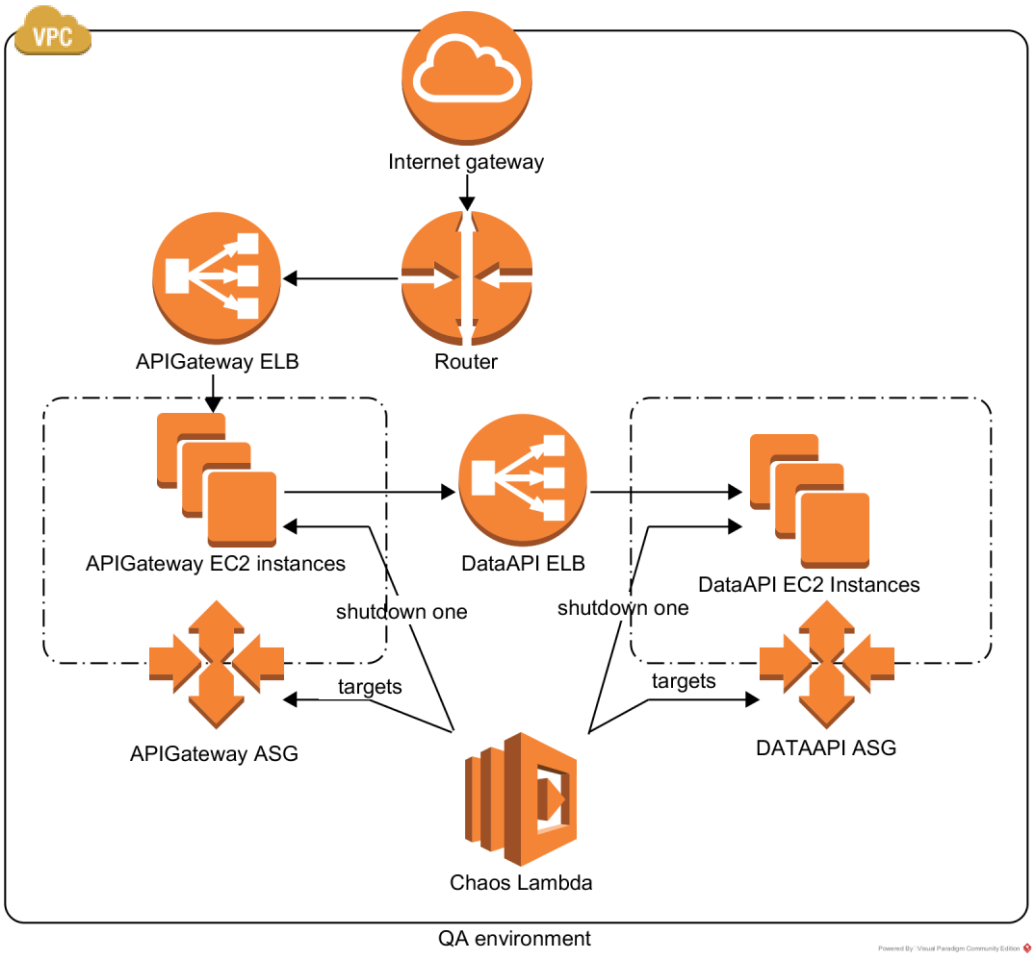

Figure 18 – Chaos Monkey Integration

The tool runs in the configured interval and targets the ASG defined in the configuration file and try to shut down one instance. In the beginning, the ASG of the DataAPI and APIGateway will be the target. After the first weeks of having the tool running, the scope is going to be increased.

This tool aims to test the global resilience of the system against a machine failure. Alerts should be configured in order to know what cause the application to be shutdown.

5.3 Future Experiments

A set of experiments, discussed in previous sections, were chosen to be carried out in a short time. Others were also designed and will be conducted in the next months. In fact, the proposed work represents just the start of a more and intense work to increase the system resilience against failures.

The principles of chaos mentioned in the section 3.4.2 were used to design the experiments but, the **run experiments in production** and **minimize of the blast radius** principles were not addressed. To perform future experiments in the production environment, a tool similar to Failure Injection Testing is going to be developed in order to reduce the blast radius and safety of the experiments. Failures and latency increase for specific requests should be easy to simulate without harming the project and creating problems to the clients, using a tool similar to FIT.

The people involved in the experiments and the awareness of this project in Mindera were low and as a future work, the results and follow up actions of the experiments should be documented and exchanged with other projects so that chaos tools should be used globally in Mindera. In the company, there are other projects that already performed chaos experiments testing the system against region failover and deployed chaos monkey into their infrastructure. The future experiments can be performed in other projects other than the target project of this thesis, to increase the adoption of chaos engineering in the company.

6 Implementation and Evaluation

The experiments followed the principles of chaos (Netflix, 2017b) that provide a guideline to develop experiments. The tests aimed to test the resilience of the system and build confidence that the system withstand against specific failures. The resultant weaknesses are identified and fixed, before they manifest in the real environment and cause unexpected problems to the business. The harder it is to disturb the system behavior, the greater is the confidence in the system's capability to handle failures.

In every experiment, the Vegeta (Senart, 2018) load testing tool together with Jaggr (Poitrey, 2018a) and Jplot (Poitrey, 2018b). The command used is presented in **Error! Reference source not found.** and explained in Table 35. The combination of the tools provides a real time analysis of the results and a graphical perspective of what is happening during the experiment.

In the realization of the experiments, the following steps were used:

1. The definition of the “steady state” of the system. This state is the normal behavior of the system and the measured output was used to define a control group for the experiment. In order to create some requests and help to define a baseline to compare the effects of the failures in the system, the load testing tool Vegeta was used. In the experiments, the first step was running a load test with some requests and save the results. The results have a measured output of the system behavior defined by the metrics: requests per second, HTTP status codes, latency percentiles and, bytes in and out.

2. The definition of the hypothesis that the experiment would hold the same “steady state” in the control group and in the experiment group. The introduced failures do not cause problems to the state of the system proving that system is resilient against a specific failure. The hypotheses are defined in the section 5.

3. The third step is the introduction of failures to simulate real-world problems and events in the system. The tests reveal weaknesses in the system and validate the existing fault tolerance mechanisms. At the start of the experiment, the load testing was used to replicate the same traffic simulated in the first step. After the applications start to receive requests, the chaos is introduced, and the results are saved.

4. In the final step, the results of the control and the experiment group are compared and based on the difference between the observations, the hypothesis is tested and disproved "steady state" is not maintained.

6.1 Database Systems Resilience

The experiment aimed to test the MongoDB resilience against node failure and particularly, the primary node. The shutdown of the primary node should not provoke any data losses and the steady state of the system should be the same during and after the failure injection. The test was performed in the QA testing environment and the failures were introduced manually as described in the section 5.1.1.

Before the introduction of the primary node's shutdown, the results of the load testing, that represent the normal behavior of the system, are present in the Table 27. The table shows the load testing using Vegeta and targeting DataAPI, a consumer of MongoDB.

Table 27 – Results of Database System's Resilience Steady State

Requests	Total=18000	Rate=30			
Duration	Total=10 minutes				
Latencies	Mean≈71.6ms	P50≈65.0ms	P95≈83.4ms	P99≈234.6ms	Max≈1.3s
Bytes In	Total=390897000	Mean=21716.5			
Bytes Out	Total=0	Mean=0			
Success	100%				
Status Codes	200=18000				

The system uses the MongoDB Cloud Manager (MongoDB, 2018a) to manage the infrastructure and monitor, automate and back up the data. The state of the QA MongoDB replica set is presented in the Figure 19.

Figure 19 – MongoDB QA Replica Set

In the Figure 19, the current primary node of the replica set is **mongo-public-qa-nodes-2**. This node is going to be the target of the failure. The commands used are simple and described in the Table 36. After the failures were introduced, the automatic failover was triggered, and a new primary node was elected.

The Figure 20 shows the replica set after the mongo process in the primary instance was shut down and a secondary node was promoted to primary.

Figure 20 – MongoDB QA Replica Set After Failure

Analyzing the Figure 20, the new primary node is **mongo-public-qa-nodes-1**. The fault tolerance of the MongoDB was tested and works as expected.

Another load testing was performed during the introduction of the failure to measure the system behavior to the database primary node's shut down. The load testing targeted the DataAPI and the results are present in the Table 28.

Table 28 – Results of Database System's Resilience During Failure

Requests	Total=18000	Rate=30			
Duration	Total=10 minutes				
Latencies	Mean≈ 89.6ms	P50≈64.4ms	P95≈83.3ms	P99≈1.0s	Max≈5s
Bytes In	Total= 390897000	Mean=21716.5			
Bytes Out	Total=0	Mean=0.0			
Success	100%				
Status Codes	200=18000				

Analyzing and comparing the difference between the results of Table 27 and Table 28, the steady state of the system was not changed. The success rate is 100% and the latencies does not change looking at the P95. The maximum of 5 seconds could be related to the leader election, during this time the system response time was higher. After the election, the system returned to the normal behavior. The system is resilient to the failure of a primary MongoDB node.

6.2 Unreliable Network Connection

The main goal of the experiment was to measure the impact of latency in a service and the target of the tests was the Data Api, a Spring Boot application. The setup used is described in the Figure 15 and has four components: Data Api connected to Toxiproxy and the proxy connect to three databases, ElasticSearch, MongoDB and RiakTS. In the tests, Vegeta was used and targeting the Data Api.

In the simulation, Docker Compose was used and described in the Listing 3. The configuration file has a **MongoDB** container, named mongo and serving in the port 27017, the **ElasticSearch** container, named elastic and listening to the port 9200, the **RiakTS** container, named riak and listening in the ports 8087 and 8098. The mongo-seed is a custom container with mongo, just to insert the schema and feed data in the MongoDB instance, using the mongos client. Then the **Toxiproxy** container, named toxiproxy, with the start command of the proxy to use the configuration file defined in "**/config/toxiproxy.json**" and listening in the port 8474, used by the client toxiproxy-cli, and other three ports that proxy the calls for each database.

The configuration file is defined in Listing 2, named "**toxiproxy.json**", and it is available inside the Toxiproxy container using a volume. The 33000 port is connected with Elasticsearch, the 33001 connected with MongoDB and 33002 connected with RiakTS. At last, the DataApi

container that is connected to the 3 different ports of the Toxiproxy. The connection between the service and the Toxiproxy is defined in a properties file in the DataApi service.

An important aspect of the setup is the use of container names directly as hostnames inside the Compose environment in order to connect containers. The hostname is resolved with the IP Address inside the simulated environment.

The Table 37 shows the commands used in this experiment. At first, the environment was created using the command “docker-compose up -d”. After the setup was completed, a load testing was performed to measure the “steady state” output of the system. The Figure 21 shows the normal behavior of the system.

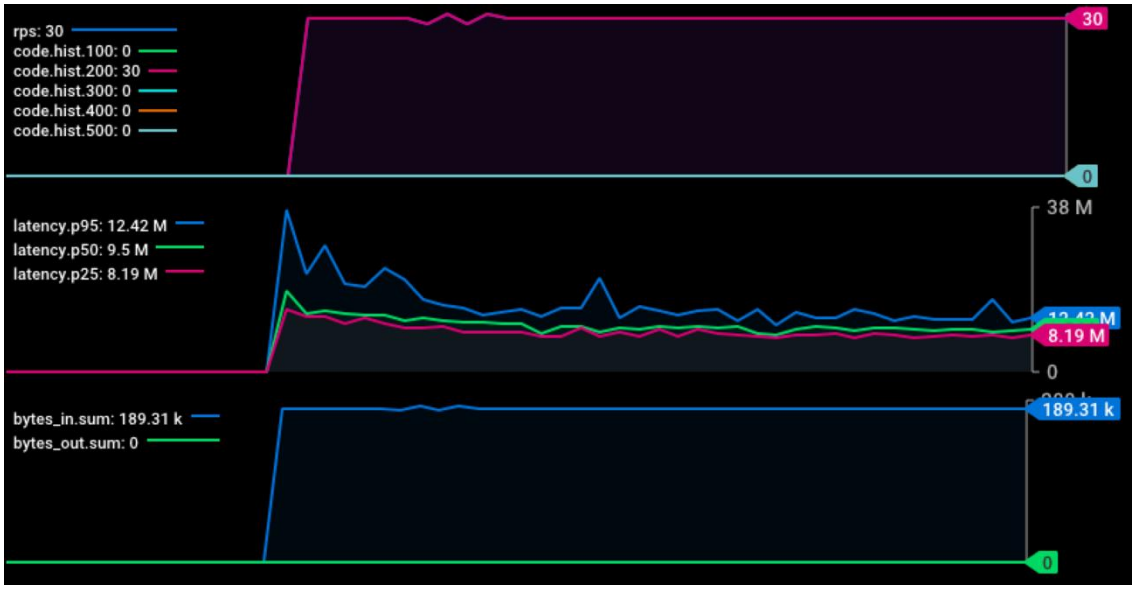

Figure 21 – Real-time Metrics of the Experiment Steady State

Analyzing the Figure 21, the behavior of the system without any problems have a request per second rate of 30 and only 200 status codes, and, a latency around 9 milliseconds. About the “steady state”, Table 29 presents some additional details.

Table 29 – Results of Unreliable Network Connection Steady State

Requests	Total=18000	Rate=30			
Duration	Total=10minutes				
Latencies	Mean≈8.37ms	P50≈8.18ms	P95≈11.7ms	P99≈16ms	Max≈53.8ms
Bytes In	Total=113589000	Mean=6310.5			
Bytes Out	Total=0	Mean=0			
Success	100%				
Status Codes	200=18000				

After the normal behavior was measured, a latency toxic was introduced in the connection between the Toxiproxy and MongoDB using the Toxiproxy Client. The command used is described in the Table 37, that introduced a latency of 100 milliseconds with a variation (jitter) – positive and negative – of 25 milliseconds. Then, another load test was performed to measure the impact of latency in the service. The metrics during the experiment with latency are presented in the Figure 22.

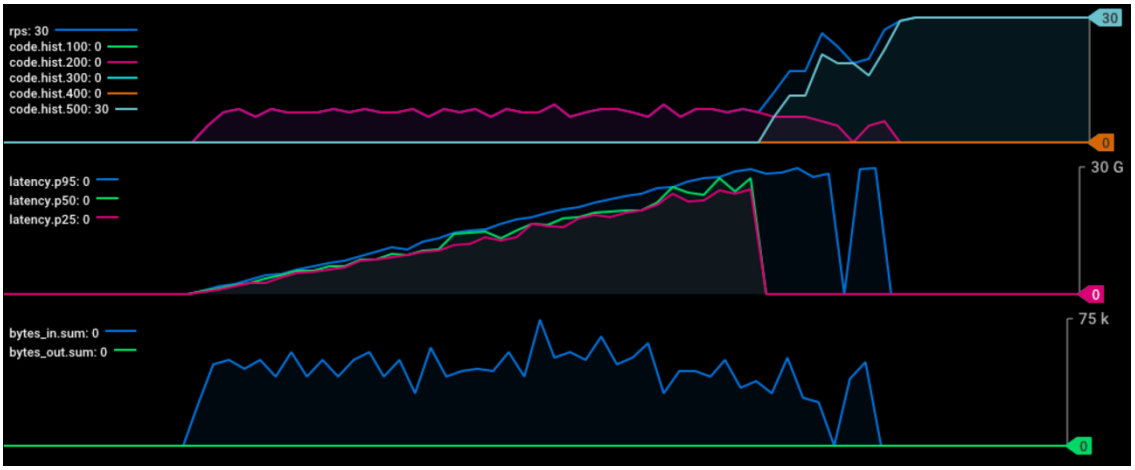

Figure 22 – Real-time Metrics of Experiment with Latency

Analyzing the Figure 22, the latency increases to 30 seconds sometime after the start of the experiment and the service only respond with 8 RPS, since the beginning of the experiment. The successful responses, represented with a status code of 200 are the same as the responses per second and after a while, the service only start to respond with 500 HTTP status code and the RPS returned to 30 – all the requests received a timeout. The Table 30 presents some additional details about the results.

Table 30 – Results of Unreliable Network Connection with Latency

Requests	Total=18000	Rate: 30			
Duration	Total=10minutes				
Latencies	Mean≈248.3ms	P50=0s	P95=0s	P99≈11.8s	Max≈29.8s
Bytes In	Total=1823235	Mean=101.3			
Bytes Out	Total=0	Mean=0.0			
Success	1.64%				
Status Codes	200=295 500=17705	Error: timeout awaiting response headers			

Analyzing both Table 29 and Table 30, the differences between the results were very high and only for 30 request per second. After the latency introduction, the service could only handle some requests (8 or 9 per second) and after some time, the requests stopped to receive a

response. The ratio was reduced from 100% of successful responses to 1,64%. The maximum latency was 30 seconds, meanwhile, the latency introduced was only of 125 milliseconds maximum.

A strange behavior was that after the completion of the load test, the service remained unresponsive for a while. In the container logs, there was an exception "**org.springframework.web.context.request.async.AsyncRequestTimeoutException:null**" repeated several times. Another problem was, during the service being unresponsive, the health check always returned a successful response, responding that the service was up and healthy.

Digging further in the problem, the use of "**CompletableFuture.supplyAsync()**" in the controller methods was causing the problems. All the requests would use the same pool "**ForkJoinPool.commonPool()**" and that caused a bottleneck in the service.

The Common Pool is configured by default to have as many threads as the number of cores available in the machine, using the method "**Runtime.getRuntime().availableProcessors()**". In the container, the number of available processors was 4. After adding the latency of 100 milliseconds, a request took a mean of 400 milliseconds to be performed (because there is more than one call to the MongoDB Database) and the requests are synchronous.

So, the number of threads in the pool is 4 and the latency is 400 milliseconds, causing the service to process 8 requests per second. The other 22 requests were queued after the request timeout configured was surpassed, the requests start to receive a timeout, but the task associated with the request, was not **removed from each threads task's queue**, causing a really long queues (that was why the service remained unresponsive after the test, it was still processing the requests).

The health check **does not use the thread pool**, so it returned a successful response, even with the service being unresponsive. The results turn to be very expressive and the use of the common pool a problem when performing **blocking requests**. The common pool should never be used to perform blocking operations since it is used also in another operations of Java such as parallel streams, and if not used carefully, can cause bottlenecks and problems in a Java application.

As a way to solve the problem two different approaches were considered:

- Turn the requests to be synchronous and avoid using a thread pool. The difference between this approach, is the increase of the load to the request acceptor threads, in case of this service, the Tomcat thread pool. When all the threads are busy, the service start to reject requests.

- Instead of using the common pool, create a custom thread pool with more threads and use it when processing the requests. This solution is good and would prevent the acceptor threads from being occupied.

Comparing the two approaches, the first one was chosen. The reason was that there was no problem identified using the first option and the default. The code has been changed to transform the requests to synchronous and to use the acceptor threads to process them. The real-time metrics are presented in the Figure 23.

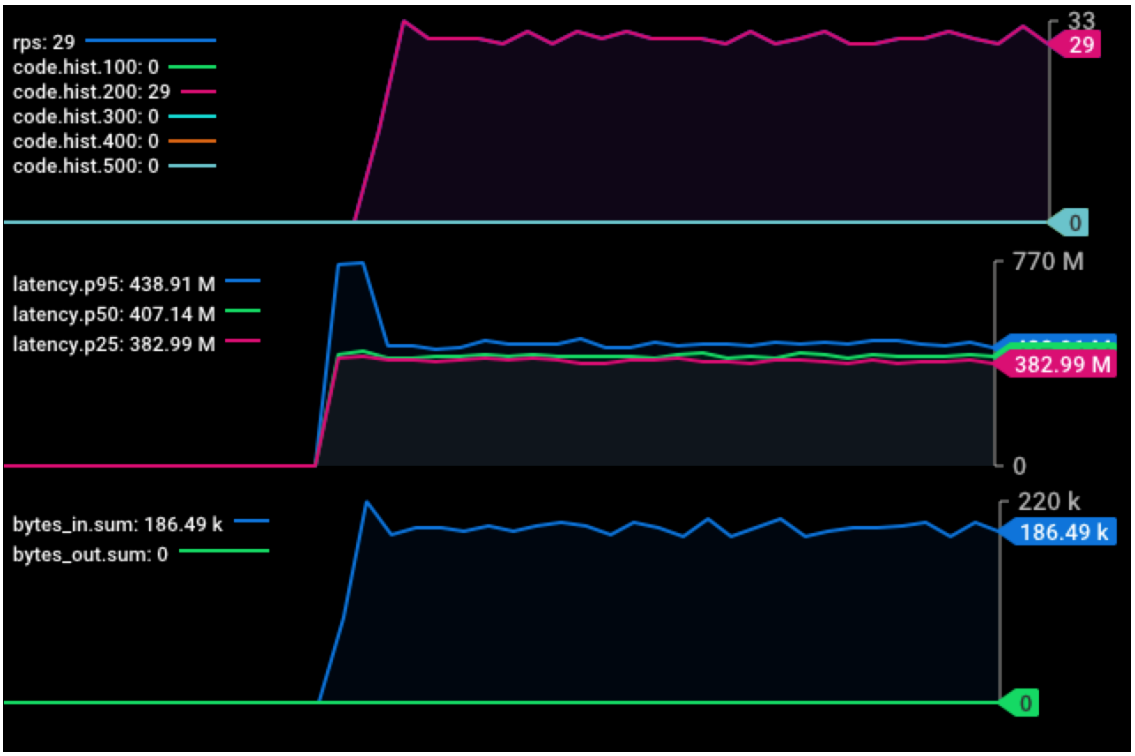

Figure 23 – Real-time Metrics of Experiment with Latency after Code Modification

Analyzing the Figure 22 and Figure 23, the changes are significant. The requests after the code modification show a constant latency and the request successes and rate returned to 30 per second. The “steady state” is now maintained in comparison to the control group. About the experiment after the modification, Table 31 presents some additional information.

Table 31 – Results of Unreliable Network Connection with Latency after Code Modification

Requests	Total=18000	Rate: 30			
Duration	Total=10minutes				
Latencies	Mean≈410ms	P50=409ms	P95=458ms	P99≈475ms	Max≈781ms
Bytes In	Total= 113589000	Mean=6310.50			
Bytes Out	Total=0	Mean=0.0			
Success	100.00%				
Status Codes	200=18000				

Analyzing the Table 31, the success ratio is now again in 100% and the latency increase to 400 milliseconds, time that a request takes with a latency of 100 milliseconds. Comparing to the Table 29, the system can withstand the introduction of latency.

The load test rate has been increased to 100 to prove that the system can now handle correctly slow database requests. The Table 32 shows the results after the code modification and with an increase of the request per second.

Table 32 – Results after Code Modification with an Increased Rate

Requests	Total=60000	Rate: 100			
Duration	Total=10minutes				
Latencies	Mean≈408ms	P50=408ms	P95=456ms	P99≈474ms	Max≈804ms
Bytes In	Total=378630000	Mean=6310.50			
Bytes Out	Total=0	Mean=0.0			
Success	100.00%				
Status Codes	200=60000				

Analyzing the Table 32, the service responded to 100 request per second and the success ratio is 100%. The latency was kept within 400 milliseconds. In conclusion, the code modification proved to solve the problem and turned the service to be available under the introduction of latency.

6.3 Resource Exhaustion

The experiment of resource exhaustion aimed to measure the impact in the system of a machine with unavailable resources to run an application. Every service needs memory and CPU resources for a normal operation, and without it, an application starts to slow down and eventually, is terminated.

The design of the experiment is described in the section 5.1.3. The service under test is the Watchlist Service but the results are equal to every other application in the system. The use of a fork bomb attack continuously creates processes that need memory and CPU to run, until the machine goes out of resources and becomes unresponsive.

Before the experiment, the Watchlist auto scaling group was scaled to three instances as described in Table 38. The Table 33, presents the steady state of the Watchlist Service using the Vegeta to perform the load testing and targeting the Watchlist Elastic Load Balancer. The ELB distributes the traffic for all the three instances available.

Table 33 – Results of Resource Exhaustion Steady State

Requests	Total=60000	Rate: 100

Duration	Total=10minutes				
Latencies	Mean≈86.2ms	P50=71.6ms	P95=97.9ms	P99≈406.4ms	Max≈4.4s
Bytes In	Total=8640000	Mean=144.00			
Bytes Out	Total=0	Mean=0.0			
Success	100.00%				
Status Codes	200=60000				

Analyzing the Table 33, the usual latency of a request is 98 milliseconds and all the requests are succeeded. In order to perform the experiment, the commands were used as described in the Table 38. After the ASG was scaled and the normal behavior of the system measurement, the load testing was performed again, and the failure was introduced. The commands consisted in the manual connection to the instance with secure shell (SSH) and the introduction of the fork bomb. The Table 34, presents the results of the Watchlist Service with chaos.

Table 34 – Results of Resource Exhaustion Using a Fork Bomb

Requests	Total=60000	Rate: 100			
Duration	Total=10minutes				
Latencies	Mean≈147.9ms	P50=70.1ms	P95=113.8ms	P99≈2.8s	Max≈9.2s
Bytes In	Total=8639588	Mean=144.00			
Bytes Out	Total=0	Mean=0.0			
Success	≈100.00%				
Status Codes	200=59998 504=2	Error: Gateway Timeout			

Analyzing and comparing both Table 33 and Table 34, the results are very similar and the differences between the latency are not significant, a change from 98 to 114 milliseconds in the P95. The two response with a 504 HTTP status code, occur when the Watchlist Service running is not capable to answer to the request. After the service becomes unresponsive, the health check requests that ELB performs against the Watchlist Service start to fail and the instance stops receiving traffic.

The Statful receives metrics from the all the services running in the system. The Figure 24 are presented the monitoring of the memory of the machines running the Watchlist application.

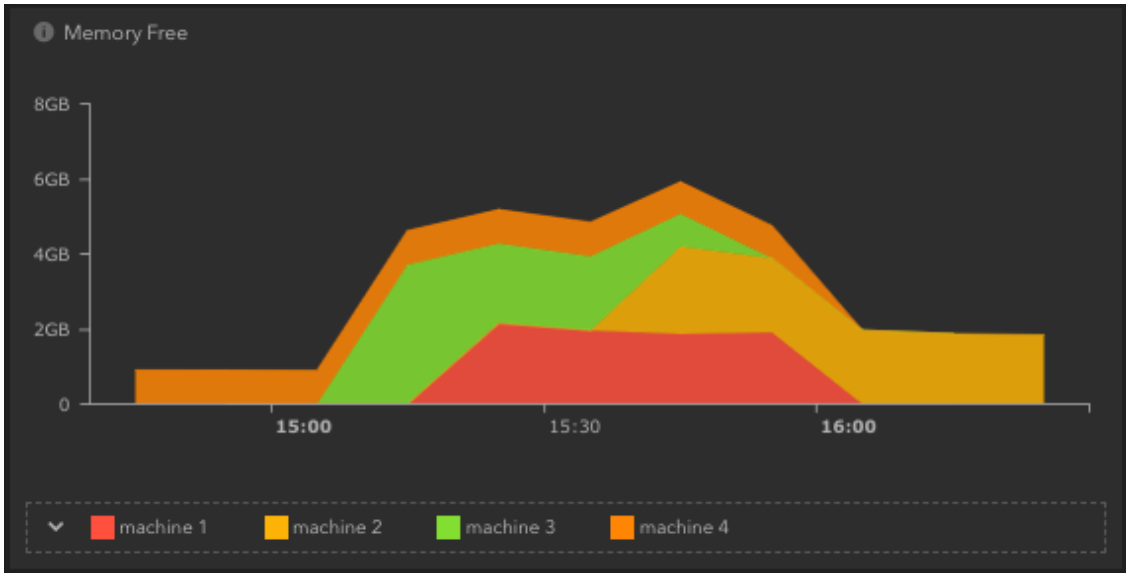

Figure 24 – Statful Metrics with Watchlist Service Machines' Memory Free

Analyzing the Figure 24, at first, there was only the machine 4 with 1GB of free memory. After the service was scaled, two more machines are started with 2GB free memory each. The experiment started around **15:30** and targeting the **machine 3** and after some time, its free memory was reduced, and machine became unavailable. The **machine 2** was launched to replace the **machine 3** by the Watchlist ASG and everything returned to the normal. At **16:05** the experiment was completed and the ASG scaled down to one instance.

As a conclusion, the use of three machines and the cloud components ELB and ASG, described in Figure 17, provide redundancy and a self-healing mechanism – health check that indicates if a machine is healthy or not, and the replacement of an unhealthy instance by the ASG – turn the system resilient to resource exhaustion.

6.4 Global Resilience

The experiment Global Resilience objective was to continuously test the overall system against a single point of failure. A machine will eventually fail and this experiment, bring the failure of a machine more often, to validate the capability to withstand against those failures and build more confidence in the system. The Chaos Lambda (Veldstra, 2018) is a Chaos Monkey (Netflix, 2017b) serverless implementation that runs in a configured schedule and in each execution, identifies a EC2 instance that matches the ASG present in a configuration file and terminates based on a probability.

The implementation uses the AWS Lambda (AWS, 2018) service that allows to deploy a function that only runs when triggered by an event and is executed by the AWS. This approach uses a serverless architecture that removes the need of configuration of a server (EC2 instance) to run the application, allowing to save more resources. The codebase is small and easy to understand, and the deployment is very easy to perform.

At first a fork from the source code was created, so the code was changed and the changed code is available at Chaos Lambda, a fork from the original Chaos Lambda (Veldstra, 2018). The schedule of events that trigger the Chaos Lambda was changed to only run hourly during **business hours** (from 9 a.m. to 3 p.m.) so the developers could solve any problem that would rise in the development environment – as a way of preparation for a real failure in a machine in the production environment.

The steps necessary to deploy the Chaos Lambda into de AWS are the following:

1. Clone the source code available in GitHub and run the command "**npm install -g**" in the project root, to install the project and make it available globally in the local machine.

2. Create a new profile to deploy the lambda, using the AWS Client. The command needed is "**aws configure --profile Chaos"** and the insertion of an **Access Key ID**, a **Secret Access Key**, the **Region Name** and the optional **Output Format**. This configures a new profile to connect from the AWS Client to the user account, to deploy the Chaos Lambda.

3. Set the region and the profile environment variables, using "**export AWS_REGION=eu-west-1**" and "**export AWS_PROFILE=Chaos**". The variables are used in the deployment.

4. Create a custom policy with **EC2.describeInstances** and **EC2.terminateInstances** in the Identity and Access Management (IAM). Create a new **role** and attach the created custom policy and also the **CloudWatchLogs** policy. These are the needed policies for the Lambda function. The permission to describe existing EC2 instances and to terminate one of them. The CloudWatchLogs is necessary to send logs from the function execution.

5. The first step to deploy the Chaos Lambda is using the command "**chaos-lambda deploy -r RoleArn**". The command will create the Lambda in the AWS and associate it with the created **role** in the previous step. The Lambda deployed will not do anything because this step does not configure the CloudWatch Events Rule necessary to trigger the function execution. After the first deployment, a file named **chaos_lambda_config.json** created with the **FunctionArn** and **LambdaRoleArn**. AWS resource names (ARN) identify uniquely a resource.

6. The second step is running the command "**chaos-lambda deploy -c CustomChaosFile.json**" and the deployment is completed. The JSON file has the configuration of **interval** which the function runs, the termination **probability,** the definition of the **targeted ASGs** and an **optional slack hook**. The tool has two different modes of operation: one with a whitelist defined by the property **enableForASGs** that specifies the ASGs to be targeted; the second, targeting every ASG, except for those defined in the **disableForASGs** array, that are not touched. When both **enableForASGs** and **disableForASGs** are present, only the rules inside the **enableForASGs** are used. As a complement, the **enableForTags** allows to consider as targets the instances with one of the defined tags.

The second deployment, creates a new CloudWatch Events rule that create events in a defined schedule, configures the lambda to be triggered by a cloud watch event and then, associates the CloudWatch Event Rule with the lambda. The Chaos Lambda is now activated and ready.

7. It is possible to check the status of the function after the deployment, running the command "**chaos-lambda status**". The command "**chaos-lambda enable**" will enable and "**chaos-lambda disable**" will disable the event creation and consequently the lambda will be activated/deactivated, respectively.

After the completion of the steps, from 1 to 6, the Chaos Lambda was deployed successfully, and the setup is described in the Figure 18. Each time the function executes, it targets the defined ASGs and terminate one of its instances, based on a 20 % probability, as defined in Listing 4. The function was only deployed in a simulated environment using a different AWS account other than the client project account. It was not possible to gather results about this experiment.

6.5 Evaluation

In order to measure the impact of the work performed in this project, four experts from Mindera evaluated the experiments and results. The developers have some years of experience developing several high-performance, scalable and resilient software systems.

In order to obtain a better classification of the experiments, a questionnaire was developed. The first question classified the importance of using Chaos Engineering approaches in a project. The questions from 2 to 6 evaluated each experiment and the problem found in the Unreliable Network Conditions experiment. The questions were evaluated from a scale from 1 to 5 in the level of importance, from "**Not Important**" to "**Very Important**". The last question was an open question to gather more feedback about the experiments and about chaos engineering. The questions and the respective answers are the following:

1. Chaos engineering is the discipline of performing experiments on a system to build confidence and validate that it is prepared to withstand against turbulent conditions in production. How do you classify the use of this methodology in a project?

   The responses are presented in Figure 25.

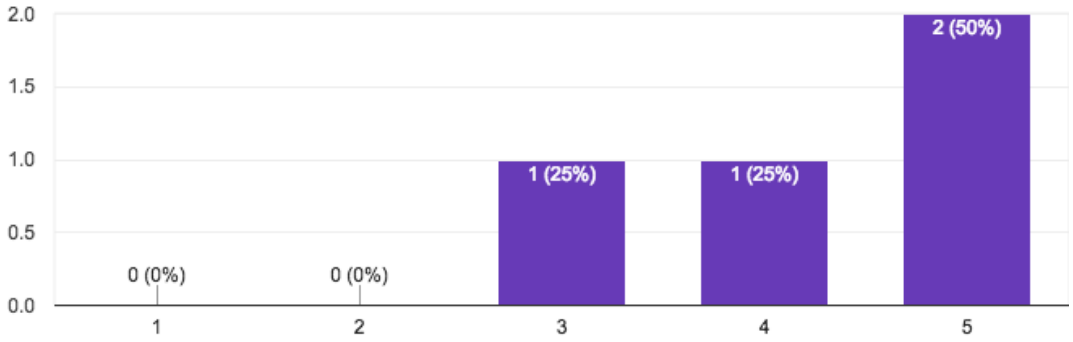

Figure 25 – Chaos Engineering Importance's Evaluation

2. **Database Systems Resilience**. In this experiment, the MongoDB fault tolerance mechanism against the primary node failure was tested. After the failure of the node, the automatic failover was triggered, and a new primary node was elected. The system proved to handle correctly the failure and the database correctly configured. How do you classify this experiment?

   The results are described in Figure 26.

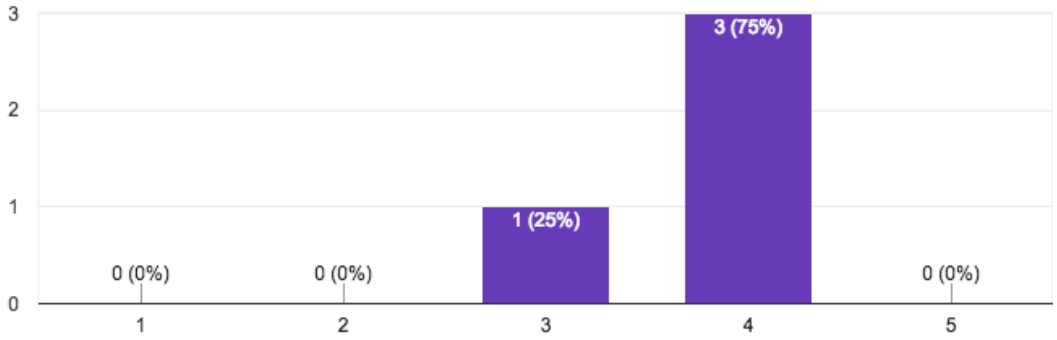

Figure 26 – Database Systems Resilience Experiment's Evaluation

3. **Unreliable Network Connection**. In this experiment, the impact of latency in an application was measured. The service under testing started to fail after some minutes and proved to be intolerant to low levels of latency. The problem was related to the exhaustion of the threads by using a thread pool to perform blocking operations. The problem was solved, and the application started to respond to a higher rate of requests per second (from 8 to 100). How do you classify this experiment?

   The classifications are presented in Figure 27.

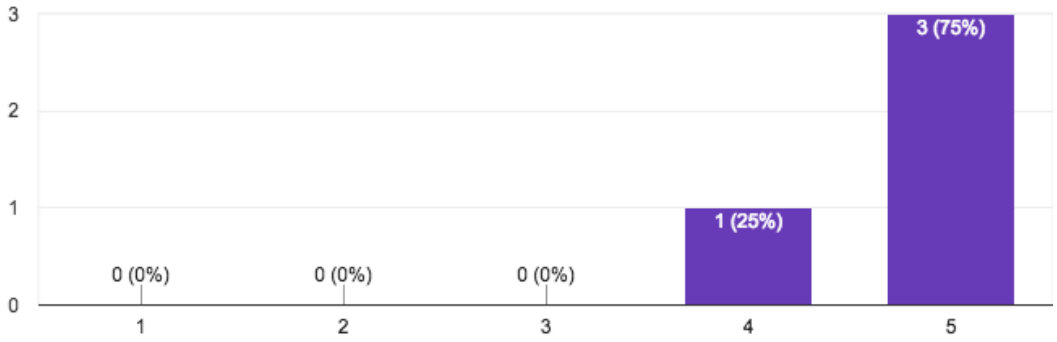

Figure 27 – Unreliable Network Connection Experiment's Evaluation

4. In the **Unreliable Network Connection**, a problem was found, and the tested application was not capable to withstand against the latency introduction in the connections to the database. How do you classify the importance of the problem found?

   The question's evaluations are described in Figure 28.

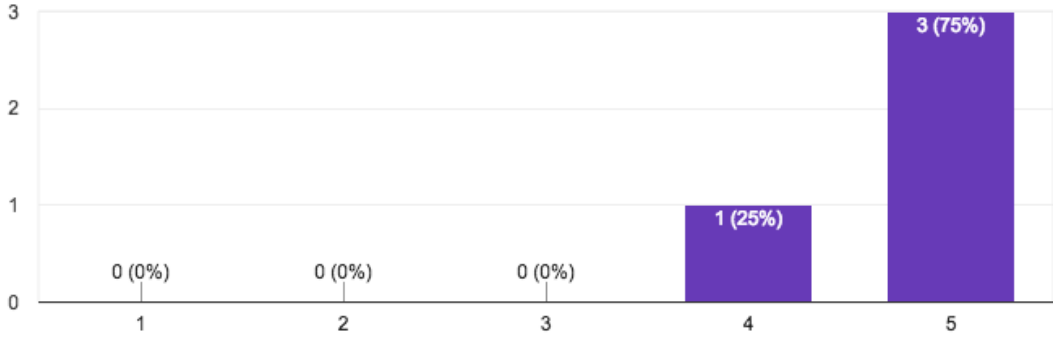

Figure 28 – Unreliable Network Problem's Evaluation

5. **Resource Exhaustion**. In this experiment, the impact of resource exhaustion in a given application was evaluated. Sometime after the start of the experiment, the instance where the chaos was introduced became unresponsive. The system proved to be resilient to this kind of failure. The health check of the ELB considered the application unhealthy and instance under test was terminated. A new instance was created by the ASG to replace it and the system returned to normal behavior. How do you classify this experiment?

   The Figure 29, presents the results of the question.

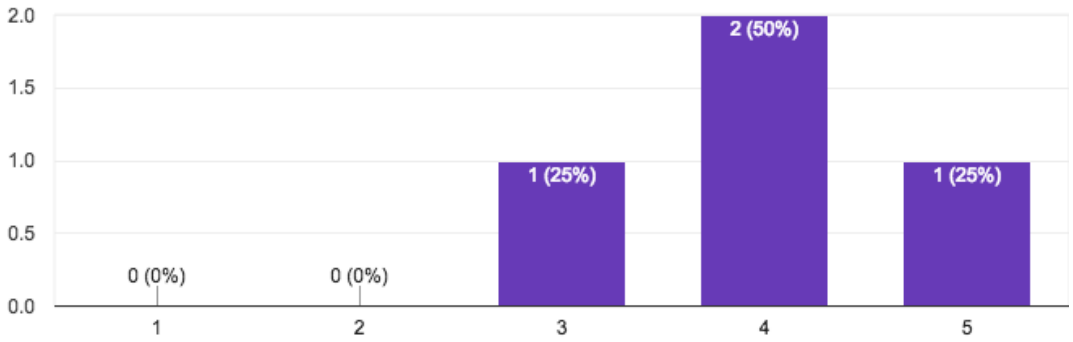

Figure 29 – Resource Exhaustion Experiment's Evaluation

6. **Global Resilience**. In this experiment, Chaos Lambda was deployed to AWS. The tool is a lambda implementation of Chaos Monkey that runs in business hours and randomly terminates an instance in the environment (it only targets the instances under the configured ASGs names). An instance will eventually fail, and the tool cause this common failure to happen more regularly in order to uncover systemic failures in services that cannot handle correctly instance failure. The tool also plays an important role in the adoption of chaos within an organization, every member developing a service after the deployment of the chaos tool must build and deploy the application to be resilient to instance failure, otherwise the service is going to fail. How do you classify this experiment?

   The evaluation of the final experiment is presented in Figure 30.

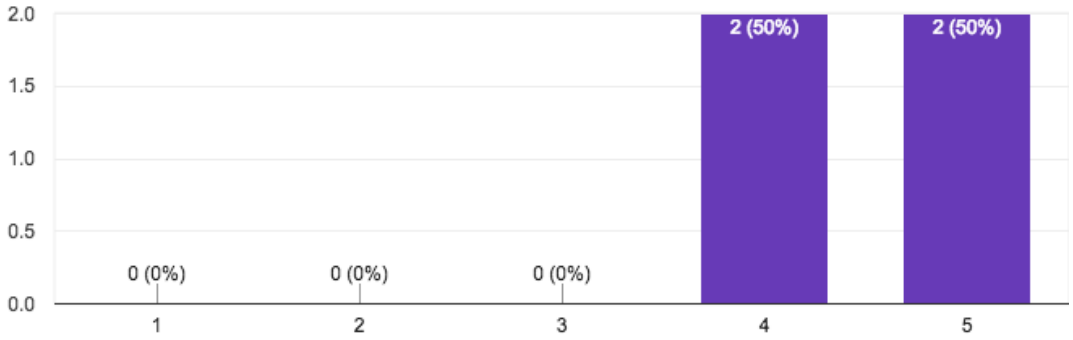

Figure 30 – Global Resilience Experiment's Evaluation

7. What is your opinion about the experiments performed and how does use of chaos engineering can help a project to achieve higher levels of resilience?

   "*Chaos engineering helps to enforce higher levels of resilience in a software system, by exposing the reaction of system when facing a failure. In perspective of reactive systems (see reactive manifesto), chaos engineering helps to achieve (or "measure") some characteristics of a reactive system: responsive, resilient and elastic" from Mehul Irá.*

"These experiments are of extreme importance. It's easy to assume things will work seamlessly and withstand failures, specially when working in cloud environments where many products/services are managed and withstand failures. Having the confidence that things will continue to operate and the system will self-heal is of the utmost importance so that customers don't get impacted and the image of the company/product is preserved" from Vasco Santos.

"The application of chaos engineering principles and techniques to this project allowed us to obtain valuable insights into the underlying properties of the system. Since production loads are typically predictable and the overall development cycle prioritizes solving these production problems, it is well possible that an apparently resilient system still has undiscovered, serious faults, but only exhibited when certain rare conditions are met. One example of such issues that these experiments helped to flag was the thread exhaustion by using a common thread pool at the applicational level. Raising awareness to this issue helped solve it in other services and even other projects" from João Costa.

"Chaos engineering is essential to guaranteeing the resilience of a project. It gives you measurable data that you can act upon. The need for resilience testing in an automated manner also grows with a project's scale, like any form of process automation, and can give a much greater level of confidence when shipping with very little long term work" from Gabriel Pinto.

Analyzing the four experiments' evaluation, the evaluation was very positive, and all the experiments had a good average classification (3.75, 4.75, 4, 4.5), respectively. The overall average classification of the experiments was 4.25 and were considered important for the project. The experiment with a higher average classification (4.75) was the Unreliable Network Connection with the highest impact in the project. The problem identified was classified with 4.75 and considered important. The experiment of Database Systems Resilience obtained the lower classification and as a future work, more testing in the database's area should be performed.

7 Conclusion

This thesis was a case study in testing using experimentation, to identify problems and weaknesses in a system. The experiments were performed in the testing environment of QA and local development machine's environment. This is the starting point to introduce new methodologies of testing in a client project and Mindera.

As a start, when performing chaos engineering is recommended to start with a testing environment and in a small area. As the confidence in the testing increases, the scope should be increased, and the environment changed to production to really verify the system resilience.

The differences between the test environment and the production environment are relevant to the experiments; some problems in the QA environment may not be replicable in production. Therefore, there is an evaluation of each problem regarding the replication in the production in order to have measure the real relevance of the problems.

7.1 Results and Objectives Achieved

In the implementation, there were four different experiments executed to verify the resilience of the system and to measure the impact of latency in an application. The system proved to be resilient to the introduced failures: shutdown of the primary node of the MongoDB and resource exhaustion in a machine. The latency introduced made the system to be unavailable and a problem was identified related with the exhaustion of a thread pool. After the problem was solved, the system proved to be capable to withstand against latency introduction in the connections between a service and a database.

Regarding the evaluation, the results were very positive, and experiments proved to be important to the project. The problem was important to find an application's bottleneck and to gather more information about threads exhaustion and how possible problems can easily occur

when latency is introduced. Some company members helped in the problem resolution and several solutions were identified.

Answering the questions defined in the section 1.2:

- After a node of the database is shutdown the teams are informed, and the alerts are only configured in production. The was not the possibility to test this case in production.
- There is a redundancy in the data and after the shutdown of a node or a primary node, there are no losses in the data and the system remain responsive.
- When a service is down, the other systems respond with an error message. More redundancy could be added in this behavior.
- The latency impact in an application was very high but now the problem was identified, and the code application needs to be changed to overcome the problem.

The last two questions could not be tested and answered. The future work should be done to answer them by performing experiments that address the problems. The database being down and how does it affect a service (e.g the service should be identified as unhealthy and shut down) should be investigated. A more proactive way to test the alerts and to measure the response time to each one, needs be achieved by performing experiments and Game Days to engage the team and embrace failure.

The objectives were successfully achieved. Information about testing disaster plans and use of different environments to perform chaos are presented. The chaos tools are analyzed in the section and compared, resulting in some tools to have a higher classification and the only considered to the project. Toxiproxy and Chaos Lamda were the only used to improve the chaos maturity and to improve the system resilience. In the last objective, only latency was simulated, and other real-world events were not possible to test.

7.2 Contributions

As contributions to the project, the resilience test of some components of the system was important to discover problems and to build confidence in the system. At the start, the project chaos maturity level was in the first level both in adoption and sophistication. Performing the experiments and using the chaos proxy to achieve the latency introduction, the project maturity was improved to 2 in sophistication with the use of the chaos proxy but in the adoption remained in the first level.

Contributions to Mindera were very important. The research in the area and about chaos tools was important to discover more information about chaos engineering and to think more about testing through experimentation. The setup of chaos proxy and the Chaos Lambda (even with

the tool was not be used in the target project) was valuable to understand how simple experiments can be performed in a project.

7.3 Limitations and Recommendations for Future Research

There were some limitations through the thesis. The project under test was a client project and confidentially was a need in order to perform the experiments and use the project. This caused some information about the project and business to be treated carefully and the focus moved only to the implementation details and some context about the project was lost, even with the freedom to perform experiments in the project.

The experiments were performed without the engagement of a team and only the testing environment was used to perform the experiments. As a future work, experimentation in the production environment should be the way to get more accurate results about the system behavior, always without causing problems to the business.

The last experiment was not possible to be performed using the target project because the moment was not the ideal to perform chaos. This is a related problem with chaos engineering because the maturity of the project must be extremely high to adopt this discipline and to embrace failure. In the client project, the product was not developed enough, and the adoption chaos engineering was not a need to start having people dedicated to the area.

Future work also goes through the achievement of the higher levels of chaos maturity and more advanced principles of chaos. Simulate more real-world events, automate experiments to run continuously and reduce the blast radius by only introducing failures to a small number of users. Reducing the blast radius and scope of the experiment is the way to safely start to perform the experiments in production.

References

Allspaw, J. (2012). Fault Injection in Production. *Communications of the ACM*, *55*(10), 48. https://doi.org/10.1145/2347736.2347751

Alvaro, P., Andrus, K., Sanden, C., Rosenthal, C., Basiri, A., & Hochstein, L. (2016). Automating Failure Testing Research at Internet Scale. In *Proceedings of the Seventh ACM Symposium on Cloud Computing* (pp. 17–28). New York, NY, USA: ACM. https://doi.org/10.1145/2987550.2987555

Alvaro, P., Rosen, J., & Hellerstein, J. M. (2015). Lineage-driven Fault Injection. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data* (pp. 331–346). New York, NY, USA: ACM. https://doi.org/10.1145/2723372.2723711

Alvaro, P., & Tymon, S. (2017). Abstracting the Geniuses Away from Failure Testing. *Commun. ACM*, *61*(1), 54–61. https://doi.org/10.1145/3152483

Andrus, K., Gopalani, N., & Schmaus, B. (2014). *FIT: Failure Injection Testing*. Retrieved from https://medium.com/netflix-techblog/fit-failure-injection-testing-35d8e2a9bb2

Atchison, L. (2016). *Architecting for Scale: High Availability for Your Growing Applications* (1st ed.). O'Reilly Media, Inc.

AWS. (2011, April 29). Summary of the Amazon EC2 and Amazon RDS Service Disruption in the US East Region. Retrieved 5 October 2018, from https://aws.amazon.com/message/65648/

AWS. (2015, September 20). Summary of the Amazon DynamoDB Service Disruption and Related Impacts in the US-East Region. Retrieved 5 October 2018, from https://aws.amazon.com/message/5467D2/

AWS. (2018, August 10). Amazon Lambda - AWS. Retrieved 8 October 2018, from https://aws.amazon.com/pt/lambda/

Beyer, B., Jones, C., Petoff, J., & Murphy, N. R. (2016). *Site Reliability Engineering: How Google Runs Production Systems*. O'Reilly Media, Incorporated. Retrieved from https://books.google.pt/books?id=81UrjwEACAAJ

Bloomberg. (2018). *Powerful Seal*. Bloomberg. Retrieved from https://github.com/bloomberg/powerfulseal

Bounce Storage. (2017). *Chaos HTTP Proxy*. Bounce Storage. Retrieved from https://github.com/bouncestorage/chaos-http-proxy

Chang, E., & Talwai, A. (2017, August 31). 3 lessons learned from an Elasticsearch game day. Retrieved 13 February 2018, from https://www.datadoghq.com/blog/elasticsearch-game-day/

Eskildsen, S. (2015). *Building and Testing Resilient Ruby on Rails Applications*. Retrieved from https://shopifyengineering.myshopify.com/blogs/engineering/building-and-testing-resilient-ruby-on-rails-applications

Freeman, T., & LaBissoniere, D. (2018). *Blockade*. worstcase. Retrieved from https://github.com/worstcase/blockade

Gaia Dev Analytics. (2018). *Pumba*. Gaia Dev Analytics. Retrieved from https://github.com/gaia-adm/pumba

Hale, B. (2018). *Chaos Lemur*. Strepsirrhini Army. Retrieved from https://github.com/strepsirrhini-army/chaos-lemur

Hedlund, M. (2014, October 28). Game Day Exercises at Stripe: Learning from `kill -9`. Retrieved 27 January 2018, from https://stripe.com/blog/game-day-exercises-at-stripe

Koen, P. A., Ajamian, G. M., Boyce, S., Clamen, A., Fisher, E., Fountoulakis, S., … Seibert, R. M. (2002). Fuzzy Front End : Effective Methods, Tools, and Techniques.

Kolton, A. (2017, July 11). It's Gameday. Retrieved 13 February 2018, from https://www.gremlin.com/it-is-gameday/

Lafeldt, M. (2016, June 16). Chaos Monkey for Fun and Profit. Retrieved 30 January 2018, from https://mlafeldt.github.io/blog/chaos-monkey-for-fun-and-profit/

Lapierre, J. (2000). Customer-perceived value in industrial contexts. *Journal of Business & Industrial Marketing*, *15*(2/3), 122–145. https://doi.org/10.1108/08858620010316831

LinkedIn. (2018). *Simoorg*. LinkedIn. Retrieved from https://github.com/linkedin/simoorg

McCaffrey, C. (2016). The Verification of a Distributed System. *Commun. ACM*, *59*(2), 52–55. https://doi.org/10.1145/2844108

Mindera. (2018). *Mindera*. Retrieved from https://www.mindera.com/

MongoDB. (2018a, May 10). MongoDB Cloud Manager. Retrieved 5 October 2018, from https://www.mongodb.com/cloud/cloud-manager

MongoDB. (2018b, May 10). Replica Set Deployment Architectures. Retrieved 5 October 2018, from https://docs.mongodb.com/manual/core/replica-set-architectures

MongoDB. (2018c, August 22). Replication. Retrieved 22 August 2018, from https://docs.mongodb.com/manual/replication/

Mozilla. (2018). *Vaurien*. Community Libs. Retrieved from https://github.com/community-libs/vaurien

Netflix. (2011, April 29). Lessons Netflix Learned from the AWS Outage. Retrieved 5 October 2018, from https://medium.com/netflix-techblog/lessons-netflix-learned-from-the-aws-outage-deefe5fd0c04

Netflix. (2012). *Simian Army*. Retrieved from https://github.com/Netflix/SimianArmy

Netflix. (2015). *Chaos Engineering Upgraded*. Retrieved from https://medium.com/netflix-techblog/chaos-engineering-upgraded-878d341f15fa

Netflix. (2017a). *ChAP: Chaos Automation Platform*. Retrieved from https://medium.com/netflix-techblog/chap-chaos-automation-platform-53e6d528371f

Netflix. (2017b). *Principles of Chaos Engineering*. Retrieved from http://principlesofchaos.org/

Netflix. (2018). *Security Monkey*. Netflix, Inc. Retrieved from https://github.com/Netflix/security_monkey

Netflix, N. (2017c). *Chaos Monkey*. Retrieved from https://github.com/Netflix/chaosmonkey

Nicola, S. (2018, February). *Multi-Criteria Decision Making - TOPSIS method*. Apresentação.

Nicola, S., Ferreira, E., & Ferreira, J. J. P. (2012). A NOVEL FRAMEWORK FOR MODELING VALUE FOR THE CUSTOMER, AN ESSAY ON NEGOTIATION. *International Journal of Information Technology & Decision Making*, *11*(03), 661–703. https://doi.org/10.1142/S0219622012500162

Osterwalder, A., Pigneur, Y., Bernarda, G., Smith, A., & Papadakos, T. (2014). *Value Proposition Design: How to Create Products and Services Customers Want*. Wiley. Retrieved from https://books.google.pt/books?id=LCmtBAAAQBAJ

Poitrey, O. (2018a). *jaggr*. Go. Retrieved from https://github.com/rs/jaggr

Poitrey, O. (2018b). *jplot*. Go. Retrieved from https://github.com/rs/jplot

Produban. (2018). *Monkey Ops*. Produban. Retrieved from https://github.com/Produban/monkey-ops

Rogers, K. (2018, August 10). Black Box vs. White Box Monitoring: What You Need To Know. Retrieved 11 October 2018, from https://devops.com/black-box-vs-white-box-monitoring-what-you-need-to-know/

Rosenthal, C., Hochstein, L., Blohowiak, A., Jones, N., & Basiri, A. (2017). *Chaos Engineering* (Mike Loukides). 1005 Gravenstein Highway North, Sebastopol, CA95472, United States of America: O'Reilly Media, Inc.

Senart, T. (2018). *Vegeta*. Go. Retrieved from https://github.com/tsenart/vegeta

Shopify. (2014, October). Toxiproxy. Retrieved 30 January 2018, from https://github.com/Shopify/toxiproxy

Shoreditch Ops. (2018, January 30). Chaos Lambda. Retrieved 30 January 2018, from https://artillery.io/chaos-lambda/

Sobti, A. (2018). *Kube Monkey*. Retrieved from https://github.com/asobti/kube-monkey

Spinnaker. (2018, January 29). Spinnaker. Retrieved 29 January 2018, from https://www.spinnaker.io/

Spring, J. (2018). *Chaos Dingo*. Retrieved from https://github.com/jmspring/chaos-dingo

Sridharan, C. (2017, October 4). Monitoring in the time of cloud native. Retrieved from https://cdn.oreillystatic.com/en/assets/1/event/262/Monitoring%20in%20the%20time%20of%20cloud%20native%20Presentation.pdf

Statful. (2018, January 29). Statful. Retrieved 29 January 2018, from https://statful.com/index.html

Stechyson, J. (2015, May 5). 8 Reasons to Test your Data Backups and Disaster Recovery Plan. Retrieved 10 February 2018, from https://hostedbizz.com/eight-reasons-to-test-your-data-backups-and-disaster-recovery-plan/

Sverdlik, Y. (2014, September 15). Facebook Turned Off Entire Data Center to Test Resiliency. Retrieved 10 February 2018, from http://www.datacenterknowledge.com/archives/2014/09/15/facebook-turned-off-entire-data-center-to-test-resiliency

Tomás. (2018). *Toxy*. Retrieved from https://github.com/h2non/toxy

Veldstra, H. (2018). *Chaos Lambda*. JavaScript, Shoreditch Ops. Retrieved from https://github.com/shoreditch-ops/chaos-lambda

Wasson, M., Bennage, C., & Buck, A. (2017). *Designing resilient applications for Azure*. Retrieved from https://docs.microsoft.com/en-us/azure/architecture/resiliency/

Wolfgang Ulaga, & Eggert, A. (2006). Value-Based Differentiation in Business Relationships: Gaining and Sustaining Key Supplier Status. *Journal of Marketing*, *70*(1), 119–136. https://doi.org/10.1509/jmkg.2006.70.1.119

Woodall, T. (2003). Conceptualising 'Value for the Customer': An Attributional, Structural and Dispositional Analysis. *Academy of Marketing Science Review*, *12*.

Attachment A – QFD

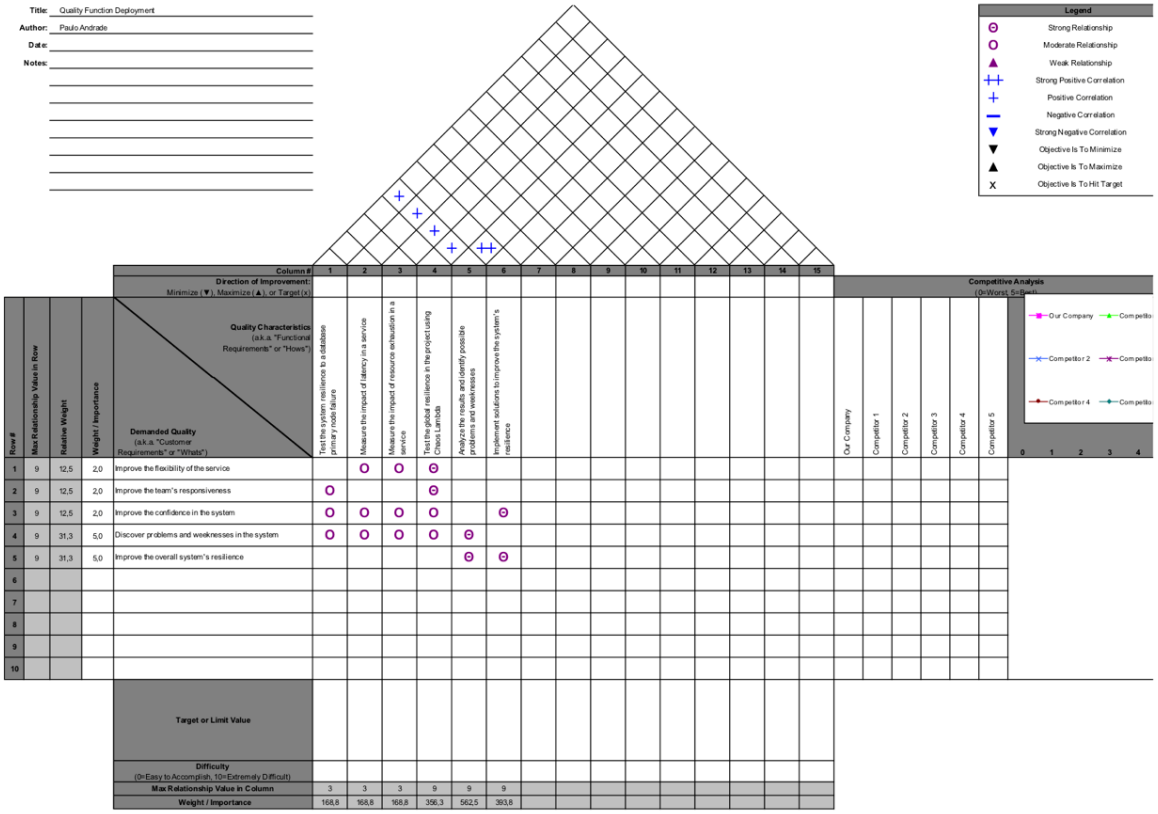

Figure 31 – Quality Function Deployment

Attachment B – Experiments

Vegeta

Table 35 – Load Test Commands

Description	Command
Create a load test with a constant rate of 30 request per second, targeting the requests defined in the file **targets.txt** with a duration of **10 minutes** and send the results to the standard output	Vegeta attack -rate 30 -duration 10m -targets targets.txt
Receive the results from the previous command and save them into a file. At the same time, send the results to the standard output. This command is good to save the results in a file and, after the experiment, create a report using the command "**vegeta report results.bin"**	Tee results.bin
Encode every result received from the previous command as JSON.	Vegeta encode
Aggregate in real time the JSON logs received in every second (configurable time) and send the aggregation with RPS, histogram with the status codes, latencies and bytes exchanged to the standard output	Jaggr @count=rps \ hist\[100,200,300,400,500\]:code \ p25,p50,p95:latency \ sum:bytes_in \ sum:bytes_out
Plot the aggregated results in the terminal to present real time statistics of the experiment	Jplot rps+code.hist.100+code.hist.200\ +code.hist.300+code.hist.400\ +code.hist.500 \ latency.p95+latency.p50+latency.p25 \ bytes_in.sum+bytes_out.sum

```
vegeta attack -rate 30 -duration 10m -targets targets.txt | tee results.bin |
vegeta encode | \
   jaggr @count=rps \
         hist\[100,200,300,400,500\]:code \
         p25,p50,p95:latency \
         sum:bytes_in \
         sum:bytes_out | \
    jplot rps+code.hist.100+code.hist.200+code.hist.300+\
code.hist.400+code.hist.500 \
         latency.p95+latency.p50+latency.p25 \
         bytes_in.sum+bytes_out.sum
```

Listing 1 – Vegeta Load Test

Database Systems Resilience Experiment

Table 36 – Database System Resilience Commands

Description	Command
Connect to the MongoDB Instance	ssh -i key.pem user_x@mongo_ip_x
Find the Process ID of the Mongo Process	ps aux | grep mongod
Kill the Process	kill mongod_pid

Unreliable Network Connection Experiment

Table 37 – Unreliable Network Connection Commands

Description	Command
Setup the Environment	docker-compose up -d
Inspect the Available Proxies	toxiproxy-cli list
Create a new latency toxic with 100 ms latency and 25 ms jitter (variation)	toxiproxy-cli toxic add dataapi_dev_mongoDB -t latency -n mongoLatencyToxic -a latency=100 -a jitter=25

```
[
  {
    "name": "dataapi_dev_elasticSearch_http",
    "listen": "toxiproxy:33000",
    "upstream": "elasticsearch:9200"
  },
  {
    "name": "dataapi_dev_mongoDB",
    "listen": "toxiproxy:33001",
    "upstream": "mongodb:27017"
  },
  {
    "name": "dataapi_dev_riakTS",
    "listen": "toxiproxy:33002",
    "upstream": "riak-ts:8087"
  }
]
```

Listing 2 – Toxiproxy Configuration File

```yaml
version: '3.3'
services:
  mongodb:
    image: mongo:3.4
    container_name: mongo
    ports:
      - 27017:27017
  elasticsearch:
    image: docker.elastic.co/elasticsearch/elasticsearch:5.5.2
    container_name: elastic
    ports:
      - 9200:9200
    environment:
      - http.host=0.0.0.0
      - transport.host=127.0.0.1
      - xpack.security.enabled=false
  riak-ts:
    image: basho/riak-ts
    container_name: riak
    ports:
      - 8087:8087
      - 8098:8098
  mongo-seed:
    build: ./mongo-seed
    container_name: mongo-seed
    depends_on:
      - mongodb
  toxiproxy:
    image: shopify/toxiproxy:2.1.3
    container_name: toxiproxy
    entrypoint: "/go/bin/toxiproxy"
    command: ["-host=0.0.0.0","-config=/config/toxiproxy.json"]
    ports:
      - 8474:8474
      - 33000:33000
      - 33001:33001
      - 33002:33002
    volumes:
      - ./toxiproxy/config:/config
    depends_on:
      - riak-ts
      - elasticsearch
      - mongodb
  dataapi:
    image: custom/dataapi:1.0.7
    container_name: dataapi
    ports:
      - 8080:8080
      - 9090:9090
    environment:
      - ENVIRONMENT=resilience-it,search-elastic-http
    depends_on:
      - toxiproxy
```

Listing 3 – Docker Compose Configuration YAML File

Resource Exhaustion

Table 38 – Resource Exhaustion Commands

Description	Command
Scale Up the Auto Scaling Group to a Production Level	aws autoscaling update-auto-scaling-group --auto-scaling-group-name watchlist-qa-asg --min-size 3 --max-size 3
Connect to a Watchlist Instance	ssh -i key.pem user_x@watchlist_ip_x
Introduce the Fork Bomb	bomb(){ bomb|bomb& };bomb
After the test is done. Revert the scaling.	aws autoscaling update-auto-scaling-group --auto-scaling-group-name watchlist-qa-asg --min-size 1 --max-size 1

Global Resilience

```
{
  "interval": "60",
  "probability": 20,
  "enableForASGs": ["apigateway-qa-asg", "dataapi-qa-asg"]
}
```

Listing 4 – Chaos Lambda Configuration File