Master in Informatics Engineering

Internship

Final Report

downCloud - Fault Injection in Cloud Platforms

Autor:
Tiago Filipe Domingues Simões
tfsimoes@student.dei.uc.pt

Orientador:
Mário Zenha-Rela

June 28, 2016

Master in Informatics Engineering

Internship

Final Report

downCloud - Fault Injection in Cloud Platforms

Autor:
Tiago Filipe Domingues Simões
tfsimoes@student.dei.uc.pt

Orientador:
Mário Zenha-Rela

Juri Arguente:
Marilia Curado

Juro Vogal:
Alexandre Miguel Pinto

June 28, 2016

Resumo

Durante vários anos o núcleo de investigação SSE (*Software and Systems Engineering*) do CISUC (Centro de Informática e Sistemas da Universidade de Coimbra) tem vindo a desenvolver diversos injectores de falhas para efeitos de investigação. Contudo, tem-se observado que após o final dos trabalhos se perde o *know-how* das ferramentas e do ambiente de injecção, nomeadamente com a partida dos investigadores envolvidos. Isto torna em muitos casos impossível reutilizar ou mesmo replicar os resultados, devido à enorme complexidade envolvida. Para além deste problema, os testes e recolha de resultados são normalmente realizados de forma manual, não automatizada, utilizando *scripts* para recolher os dados, que posteriormente são analisados utilizando ferramentas de análise de dados externas.
Com este problema em mente o grupo de investigação SSE decidiu lançar o estágio *downCloud*, visando desenvolver uma aplicação web que permita doravante aos investigadores enviar cada injector desenvolvido para esta plataforma para o testar e para outros investigadores externos à equipa o usarem, para validar resultados obtidos ou mesmo para testarem a presença de erros nos seus sistemas. A intenção foi desenvolver uma solução totalmente automática, isto é, capaz de realizar as campanhas de injecção sem qualquer interacção humana ou *scripts*, e no final apresentar um relatório de síntese dos principais resultados obtidos. Posteriormente os investigadores podem descarregar os resultados de injecção em bruto para uma análise mais detalhada, se assim o desejarem.
Este foi o objectivo, plenamente atingido, deste estágio. De assinalar que não era nosso objectivo desenvolver qualquer injector de falhas, mas sim todo o ambiente de gestão de campanhas de injecção usando injectores desenvolvidos por terceiros.
A solução desenvolvida ao longo deste estágio já está disponível como um serviço (*SaaS*) que, através de uma aplicação web, permite que qualquer utilizador-investigador envie o seu injector de falhas para teste ou para que utilizadores 'finais' testem o impacto de erros de hardware nos seus sistemas utilizando injectores disponibilizados por terceiros.

Palavras-Chave

Injecção de falhas, servidores cloud, benchmark, aplicações web, engenharia de software

Abstract

During the last decades the SSE (Software and Systems Engineering) research group of of CISUC (Center for Informatics and Systems of the University of Coimbra) have developed several fault injectors for research purposes. However, after the end of the projects and the involved researchers leave, it is almost always impossible to reuse the research setup platform to replicate the experiments, due to its complexity. Moreover, most of the analysis is performed manually using external data processing and analysis tools.
With this problems in mind the SSE research group launched the *downCloud* master thesis, aiming to develop a web-based platform that allows researchers to upload their injectors to the platform to test them and also allows other interested parties to use them, to validate results or to test their own systems against hardware errors. It must be stressed that it was out of scope of this project to create a new fault injector, but rather to develop the environment to support researchers' fault injection campaigns.
These goals have been fully achieved, there is now an operational web platform (*SaaS*) functioning autonomously, so that it is now possible for any researcher to upload its fault injector and launch a fault injection campaign without human intervention, so that, after the campaign is over, a synthesis of the most relevant results is delivered. This solution also supports assessment by 'final' interested users of their system's resilience to hardware errors by using third party fault injectors.

Keywords

Fault Injection, cloud servers, benchmark, web applications, software engineer.

Contents

List of Figures

List of Tables

Acronyms

CISUC	Centre for Informatics and Systems of the University of Coimbra
IaaS	Infrastructure as a Service
MVC	Model View Control
ORM	Object-Relational Mapping
PaaS	Plataform as a Service
REST	Representational State Transfer
SaaS	Software as a Service
SEE	Software and Systems Engineering researach group of CISUC
SSH	Secure Shell
SPEC	Standard Performance Evaluation Corporation
TPC	Transaction Processing Performance Council

Chapter 1

Introduction

This document presents the work developed in partial fulfillment of the Master degree in Informatics Engineering at the University of Coimbra, in the academic year 2015/2016, by the student Tiago Filipe Domingues Simões.
In this chapter we present the *downCloud - Fault injection in clouds* project, its motivation, objectives and the document structure.

1.1 Motivation

During the past decades the Software and Systems Engineering research group of CISUC (Center for Informatics and Systems of the University of Coimbra) has developed several fault injectors for research purposes. However, after after the end of the projects and the involved researchers leave, it is almost always impossible to reuse the research platforms to replicate the experiments due to its complexity and insufficient documentation; researchers' focus is more on the experiments themselves than in the tools they use along the way, seen as a mean not as an end unto themselves. Moreover, most of the analysis is performed manually using external data processing and analysis tools, so not much information is left behind.
With this problems in mind the SSE research group launched the *downCloud* master thesis, aiming to develop a web-based platform that allows researchers to upload their injectors to the platform as well as all the scripts involved in setting up the research environment, in order to test their tools and also allowing other interested parties to use them, to validate results or to test third party systems against hardware errors. It must be stressed that it was out of scope of this project to create a new fault injector, but rather to develop the environment to support researchers' fault injection campaigns.
There is an interesting circularity in this project, as the idea is to develop a

cloud-based infrastructure that supports cloud-targeted fault-injection.

1.2 Cloud-as-a-target

Cloud computing is today a common-place reality: *Google*, *Netflix*, *Facebook*, *LinkedIn*, are some of the services available and used by millions of users around the globe. Following such titans of the internet, lots of big, small and medium enterprises start to use platforms in the cloud (e.g. *AmazonEC2* [2], *WindowsAzure* [3], etc), not only to store their data, but also keep available their services to their customers. These infrastructures provide different computational resources, what permit that several enterprises build their applications online without the costs of having their own physical infrastructure.
However, this change of infrastructure paradigm keeps raising many concerns, specially the resilience of this platforms to hardware errors. The continuous size reduction of the transistors in the processors and the progressive reduction in their work voltages has led to an increase of transient hardware errors. In most of the applications it is assumed that this errors are treated by the several layers of software, including the application layer.
The SSE research group of *CISUC* have decades of experience in the development of fault injectors in several platforms, from embedded systems to distributed big data warehouses. With the emergence of the dissemination of systems based on cloud computing, naturally arises the interesting to evaluate the resilience of such systems. It's specificity raise the need to create new tools to perform such experiments that, due to its scale and complexity, are not amenable to the 'handcrafted' approaches used in less complex systems. Thus, targeting the cloud for fault injection was the driver to build a SaaS infrastructure to support a cloud-based fault injection platform.

1.3 Objectives

This thesis has two major goals: on one hand the objective is to design and develop the first functional version of a fault-injection platform for systems hosted in the cloud, based on previous investigation of fault injection in hypervisors [4].
On the other hand, the solution itself should be designed as service (*SaaS*). The solution to be developed should also include a semi-automated process of fault injection and also a framework to support a preliminary analysis of

the results from the fault injection campaigns.
One of the main features of the platform to develop in this internship was its usability, in a perspective of a 'product': the platform was designed to be itself hosted in a cloud infrastructure and be available in two different modes, expert and auto. In the first mode is expected for the user to have enough knowledge to define the several parameters of the fault injection campaign. In the second scenario the user does not need to have the same expertise knowledge, and consequently the system performs a (semi) automatic definition of several parameters of the injection.
Other important aspect of the solution implemented was its intended scope: since there is a wide variety of services that are hosted in cloud platforms, this thesis will be focused solely on web applications. Given the innumerable platforms and solutions in the cloud, and the time available for this thesis work, this scope intends to lead to a first fully usable solution, e.g. the full functionality of the service, even if just on a subset of all the possible targets.
To deliver a complete usable solution, the thesis work also provides the a preliminary analysis of the results of the injections performed. For more advanced analysis the researcher/user can access the full fault injection data in an external database, that can be browsed and managed by external tools (e.g. *Oracle Business Intelligence* [6], *Pentaho* [7]).
For the thesis purpose we adopted a workload that has been extensively used by the SSE research team (extracted from a web-TPC suite), but the platform allows the researcher to select its own workload.
In summary, the main objectives for this thesis are:

- Independent application of methodologies and practices of a software engineer;
- Design and development of a software architecture capable of handling complex fault -injection campaigns (several different campaigns, in different cloud-based targets, with different types of injectors, and be able to make a preliminary analysis of results);
- Development of a semi-autonomous solution, capable to handling simultaneously different campaigns in different targets with different parameters;
- Development of the preliminary analysis of the results and deliver a meaningful report;
- Development of a front-end that allows user to setup a campaign with

their specific parameters and see the report and results of the experimental campaigns performed.

- Assessment of the correct operation of the system developed.
- Publication of a paper with the description of this injection framework.

Below we present a concise initial schedule of the work that was performed:

1º Semester work plan

- Analysis of state of the art in fault injection;
- Study of the technologies to be adopted, namely those cloud-based.
- Specification of architecture to implement;
- Selection of the workload;
- Specification of the data model for the platform;
- Write the intermediate report.

2º Semester work plan

- Development of the fault injection framework;
- Development of the preliminary analysis of injection results;
- Development of other system components;
- Testing of the platform.
- Delivery for third party testing in their own platforms (external assessment).
- Writing the final report (this document).
- Writing of a scientific paper describing this 'open' fault-injection framework.

1.4 Document structure

In this first chapter we contextualized the reader about the problem, the motivations to start this project and the thesis goals.
In the second chapter we present some background information about the methodologies, architectures and technologies used in this project.
In the third chapter we present the work plan, methodologies and processes of software engineering used during the work.
In the fourth chapter we describe how the requirements gathering was performed and validated, as well the specification of all functional and non functional requirements for the solution developed.
In the fifth chapter we describe of architecture designed and implemented.
In sixth chapter we describe all the actual work products (artifacts) that were developed during the two semesters.
In the seventh chapter we describe all the functional and quality tests that were performed to ensure the correctness of the platform.
And finally, this thesis concludes in the eighth and last chapter with a final reflection and the conclusions.

Chapter 2

Background concepts

In this chapter we present an overview of the core concepts involved in the *DownCloud* project. While it involved a rather extensive literature survey, this is not a state-of-art, but instead an overview of the core concepts of the many and varied fields covered by this work.

2.1 Cloud computing

We intend that the *downCloud* platform has the ability to scale up if needed to serve the worldwide fault-injection research community needs, and that this service can be available to everyone with an internet connection. Beyond this connected scalability our platform must be able to test the resilience of cloud-based servers. In view of these complementary needs we performed a 'state-of-practice' survey about cloud computing concepts and services that is synthesized below:
According to *The NIST definition of cloud computing* [8], *Cloud Computing* is "a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction.". In the same reference five essential characteristics for *Cloud Computing* are listed:

- **On-demand self-service:** A consumer can request and obtain computing capabilities, such as server time and network storage, without requiring human interaction;

- **Broad network access:** Capabilities are accessible over the network through standard mechanisms;

- **Resource pooling:** The provider's computing resources are pooled to serve multiple consumers, where the resources are dynamically assigned and reassigned according to consumer demand. Usually the consumer has no control or knowledge over the physical location of the resources;
- **Rapid elasticity:** Capabilities can be elastically provisioned and released, in order to adapt the demand;
- **Measured service:** Resource usage can be monitored and controlled providing transparency for both the provider and consumer of the utilized service.

2.1.1 Service models

There are three different types of service models that can be used to provide *Cloud Computing* services:

- **Software as a Service (SaaS):** The capability provided to the consumer with access through various client interfaces to software services running on the cloud. Examples of this service all *Google Apps* [10], *Dropbox* [11], *Office365* [12] from *Microsoft*;
- **Platform as a Service (PaaS):** The capability provided to the consumer to deploy his applications into a cloud infrastructure, by using programming languages, libraries,service and tools supported by the provider. The consumer can not manage or control the underlying cloud infrastructure, keeping only control over the deployed applications and possibility configuration settings for the application-hosting environment. Examples are *Heroku* [13], *OpenShift* [14] and *Windows Azure* [3];
- **Infrastructure as a Service (IaaS):** In this model the consumer is provided with the capability of manage the processing, storage, networks and other fundamental computing resources where the his applications run. Examples are *Amazon EC2* [2], *Google Compute Engine* [15] and *Rackspace* [16].

In Figure Figure 2.1 a visual comparison between IaaS, SaaS, PaaS and 'on premises' traditional servers is presented.

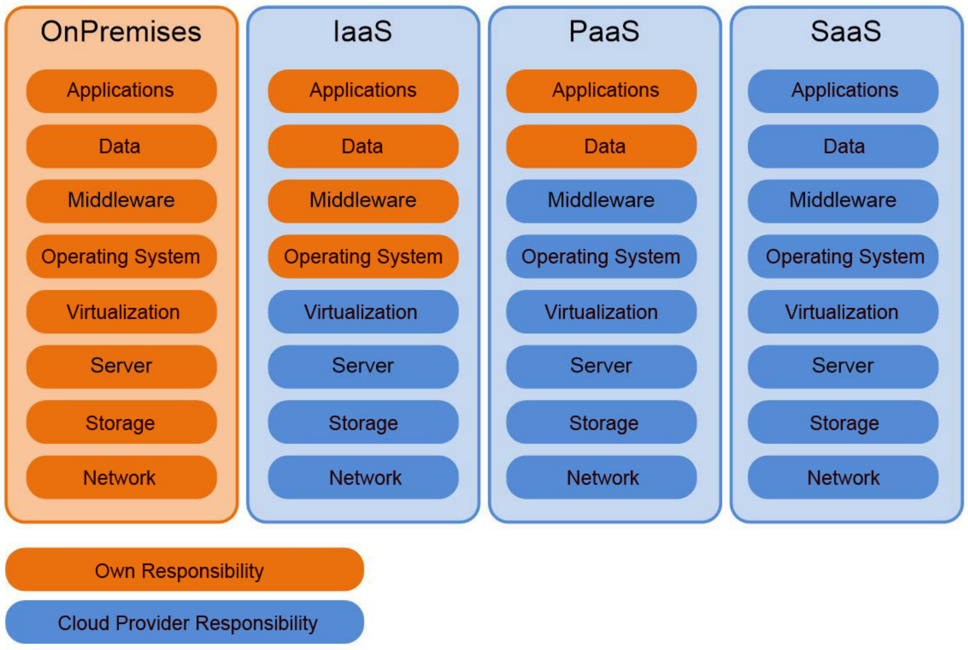


Figure 2.1: Comparison between own server, Iaas, Paas and SaaS [17]

As can be seen, the infrastructure responsibility varies depending on the service level, being totally 'in the cloud' for SaaS.

2.1.2 Deployment models

Not all clouds are public. Actually a cloud computing infrastructure can be deployed in several settings:

- **Public:** The cloud is provisioned for open use by the general public. (E.G. *Heroku* [13])
- **Private:** The cloud infrastructure is provisioned for exclusive use by a single organization.
- **Community:** The cloud infrastructure is provisioned for exclusive use by a group of organizations that share the same concerns.
- **Hybrid:** The cloud infrastructure is composed by two or more distinct cloud infrastructures (private, community, or public) that remain unique entities, are linked together by standardized or proprietary technology that enables data and application portability.

From the above survey of concepts we can now decide the type of target platforms aimed by our platform. Since we want to emulate hardware faults, thus we need direct access to the operating system kernel, the only possible target (by now) will be **IaaS** services. In what concerns deployment models, all are possible targets as long as the end-user has root access to their allocated servers. No other constraint to the applicability of our platform has been identified so far.

We shall now present our survey on benchmarking concepts in order to assess the resilience of the target systems.

2.2 Benchmarking

One of the core objectives of *downCloud* project is to provide a platform that allows any end user (most likely a researcher on fault-injection) to be able to test the resilience of his/her own servers in a cloud environment. Since we pretend to develop a service that allow such end user to test and evaluate his environment with several parameters, we surveyed the main concepts related with benchmarking.
Benchmarks are standard tools that allow the evaluation and comparison of different systems or components in different areas such as performance, dependability, security or resilience.
In order to produce valid and useful benchmarks there is certain criteria which must be respected. These criteria is discussed in the core reference on benchmarking, *The Benchmark Handbook* [18]:

- **Significance:** The benchmark has to measure the performance of the typical operation of the problem domain.
- **Portability:** The benchmark should be capable of running in several different systems and architectures.
- **Scalability:** The benchmark should be scalable to cover small and large systems.
- **Simplicity:** The benchmark should be understandable to avoid lack of credibility.

Further studies about benchmarking has yielded further criteria [19]:

- **Relevance:** The benchmark has to reflect something important;
- **Repeatable:** The benchmark result must be reproducible by running the benchmark under similar conditions;

- **Fair & Portable:** The benchmark should provide a fair comparison between systems and not favor or penalize a system or architecture over another;
- **Verifiable:** The results provided by the benchmark need to inspire confidence as of their validity and representativeness;
- **Economical:** Running the benchmark should be affordable.

Performance benchmarks are the most common. These benchmarks focus on performance of the systems for posterior comparison. The two most influential standardization organizations in this area are *TPC* (Transaction Processing Council) [20] and *SPEC* (Standard Performance Evaluation Corporation) [21].

In the context of this work we took a pragmatic approach, and adopted a benchmark used by the SSE group in previous studies which is based on a TPC benchmark to assess web-services.

2.3 Fault injection

For this first version of *downCloud*, we will focus on how cloud servers react to a fault injection that emulates transient hardware errors. This is due to the availability of an 'injection driver' that emulates such faults. So, we also did a survey about fault injection to understand better what other kind of fault injections exist, so that we could position the fault-injector made available to us compared to others.
Fault injection is an experimental technique for assessing the behavior of systems under the presence of faults. Actually it is the only way to intentionally trigger its fault tolerance mechanisms in a controlled way. Fault injection can be defined as "the process of deliberately introducing faults or errors in computer systems, allowing researchers and system designers to study how computer systems behave in the presence of faults" [22]. This experimental technique can be used in several different contexts and can serve different purposes, namely:

- Assess the effectiveness of fault-tolerance mechanisms.
- Study of error propagation and error latency.
- Tests the correctness of fault-handling mechanisms.
- Measure the time it takes for a system to detect and recover from errors.

- Test the correctness of fault-handling protocols.
- Verify failure mode assumptions of components or subsystems.
- Study the impact of faults and fault-tolerance mechanisms

Given the repeated use of fault injection related concepts in this document, the definition of the core concepts is given below, so that the reader can follow this document without ambiguity. **Fault** is the phenomenological cause of an error (e.g. a gamma ray hitting a transistor); **Error** is a wrong state of the system, i.e. a discrepancy between the correct and the actual internal state; **Failure** is an external visible manifestation of an error, i.e. when the system is unable to perform its desired function. [23]. **Target system** is the system that will endure the fault-injection testing. **Fault-load** is a (set of) operation(s) that will interfere with the system (actually that generates internal errors). **Workload** is as operation that will be used to exercise the system (e.g. an HTTP request). **Fault injection experiment** is the execution of a workload and fault-load in a controlled experimental environment. **Fault injection campaign** can be defined as the execution of several fault injection experiments.
Different fault injection techniques can be compared and characterized on the basis of several different properties:

- **Controllability**: the ability of control the injection of faults, both in time and space (location).
- **Observability**: the ability to observe the manifestations of the injected faults. This is normally an (un-met) technical challenge.
- **Repeatability**: the ability of being capable of repeating an experiment and obtain the same results. This is extremely difficult, as no system is exactly in the same state at the exact point in time due to concurrency and non-determinism of CPU execution.
- **Reproducibility**: the ability to reproduce the results of a fault injection campaign. This is easier to achieve than repeatability as we are dealing with statistical values.
- **Representativeness**: how accurate can be the emulated system, the workload and the fault-load represent the real system.

The most common fault injection techniques are hardware-implemented fault injection and software-implemented fault injection. These techniques vary in the method used to inject the faults. While software-implemented fault injection uses software for that purpose (e.g. a kernel driver that flips register or memory bits), the hardware-implemented fault injection make uses of specialized hardware components (external or internal, e.g. a boundary scan infrastructure).

2.3.1 Fault types

A fault can be classified as a software fault, a malfunction due to a software defect, or an hardware fault, a malfunction of an hardware component. Typical software faults, injected by fault injection tools are incorrectly assigned variables, wrong loop counter initialization, off-by-one overflows and incorrect comparison rules. Since the injection driver provided by the SSE for this work emulates hardware errors, software faults are out of scope.
Hardware faults can be of various kinds, such as stuck-to-on, stuck-to-zero and bit flip, and can affect a wide range of computer components, from CPU registers to memory transistors or communication buses. Our injection driver flips bits on the CPU registers.
Faults can be also classified, as permanent, intermittent and transient. A transient fault will disappear after some time, can also be causes by environmental effects. An intermittent fault will appear and reappear in an apparently random fashion. A permanent fault will remain active until the affected component is repaired.
The emulation of a permanent hardware fault is considerably more elaborate than emulating a transient fault, since a permanent fault requires a manipulation every time the target hardware component is read, while a transient fault needs only one manipulation. That is the case with our injection driver (bit flip of CPU registers)
Faults can also manifest themselves in a number of different ways. Some faults will cause the system or program to completely crash while others will cause it to enter an hang state. Some can have a less destructive effect, such as slowing down the performance of the system , making the system produce wrong or invalid output, or have no effect at all. Understanding the system behavior under faults is the main goal of fault injection. Our platform will deliver a synthesis of the major behavior outcomes after a fault-injection campaign.

2.4 System properties

We close this chapter by defining some global concepts so that the reader can understand which system properties can be assessed using fault injection.

Performance is the amount of work accomplished by a computer system. [30]

Dependability is the ability to provide services that can defensibly be trusted within a time-period. This may also encompass mechanisms designed to increase and maintain the dependability of a system or software. [31]

Security is the protection of information systems from theft or damage to the hardware, the software, and to the information on them, as well as from disruption or misdirection of the services they provide. [32]

Resilience is the ability to provide and maintain an acceptable level of service in the face of faults and challenges to normal operation. [33]

Availability refers to the ability of the user community to obtain a service or good, access the system, whether to submit new work, update or alter existing work, or collect the results of previous work. If a user cannot access the system, it is - from the users point of view - unavailable. Generally, the term downtime is used to refer to periods when a system is unavailable. [34]

2.5 Conclusion

At the end of this chapter the reader should have the background concepts involved in the downCloud project.
In the next chapter we shall present the work methodology and the techniques and processes of software engineering used during the project.

Chapter 3

Work Methodologies

In this chapter we present the work methodologies adopted in this project. We cover the software lifecycles adopted, the work plan and the major processes of software engineering used during development.

3.1 Software processes and lifecycle

The development process mostly used in this internship was based in the *Scrum Agile* methodology. However, during the first half of the first semester, we adopted the straightforward *Waterfall* methodology to do the study of background concepts, analysis of the problem and requirements gathering. Having consolidated this knowledge, in the second half of the first semester, during the experimentation with the software architecture, we adopted two week *sprints* for the work [35]. This approach was so effective, as the supervisor requests kept evolving as the software was built, that this methodology was adopted until the end of the internship. The artifacts produced during the this phase (agile sprints) can be consult in Annex 9.1.

3.1.1 Agile Development

The *Scrum Agile* methodology is considered very effective in projects, where the requirements of the solution are not stable. In our case, while the overall goals were clearly defined from the beginning (which allowed a solid architecture to be defined), *Scrum* intended to be the methodology of choice for projects with unstable requirements, so that incremental releases are delivered maximizing value to the 'client'. This approach requires high proximity with the client to define the next release. In our case, with weekly or bi-weekly meetings with the supervisor, this proximity was ensured.

Using this methodology it was possible to better manage the requirements, by a good communication with the supervisor. During the initial stages of the project, we used a *Waterfall* methodology, so that we could take advantage from the sequential more 'formal' steps (requirement gathering, technology survey, architecture design) .
In the last phases of the project the progression was done by a set of biweekly interactions (*sprints*). In our case there were only two roles of the several that Scrum suggests:

Main Roles

Product Owner Is who represent the client and have a vision over what he wants, having responsibility to communicate the vision to all the team of the *Scrum*. The product owner also prioritizes the task that exist in the *Product Backlog*. In this project this role was assumed by professor Raul Barbosa.

Development Team Is responsible for the delivery of increments of the project that could by used in the final product at end of each *sprint*. The development team for the downCloud project had only one person, the author.

Scrum Master Is responsible to ensure that the development team fulfills the good practices in *Scrum*, is also responsible to help all the elements from development team to ensure the work is done in best possible way. This role was performed mostly by the supervisor, professor Mário Rela.

Cycle of Agile Methodology

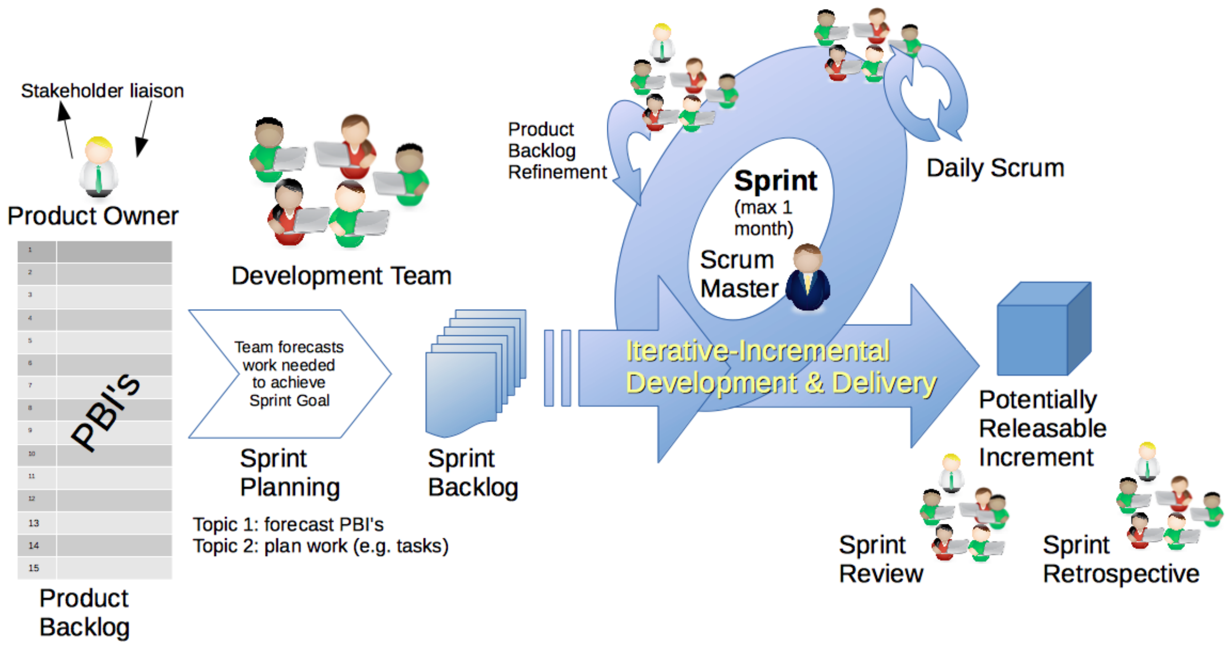


Figure 3.1: *Scrum* agile methodology cycle[36]

Product Backlog Is a list of tasks set and prioritized by the *Product Owner*.

Daily Scrum Meeting It is a brief reunion that happens every day at same hour, with maximum duration of 15 minutes, between all the elements of the development team, with the objective to have a situation report about the tasks complete and to be complete. Being a aone person project, the author updated his project log with a brief reflection.

Sprint Backlog It is the list of task identified by the *Scrum* team to be finished during the *sprint*. In this project the tasks are planned from *user stories*, and are estimated by the author according to the tasks that are present in *Product Backlog*, most of the time was used the technique of *Planning Poker* [37], to estimated the points. The duration of the development *sprints* varied between two and four weeks.

So, despite being a one member team, the adoption of different software lifecycles in different phases of the project has proven to be a good decision. We had the rare opportunity of realizing that a single project can have more than one lifecycle. Having the capability to change the approach according to the project needs was a great insight from this experience.

3.2 Work Plan

We shall now present the work plan for the internship, represented in two *Gantt* diagram, and also the artifacts produced in each phase.
The work plan for most of the first semester, was referred in the previous section, follows a classic perspective (waterfall), it was more focused in the requirements, survey and architectre design artifacts.
The work plan for the second semester was elaborated with the definitions of *milestones*, since the development proceeded according to the agile methodology also described in the previous section.
We also performed a risk analysis to identify and monitor the major risks for the project, as well as its impact. The risk analysis is present in Annex 9.2.

Work plan for 1º Semester

	2015											2016			
	October		November					December				January			
	19	26	2	9	16	23	30	7	14	21	28	4	11	18	22
Start of internship	*														
Environment and objectives	▨														
Definition and analysis of business requirements		▨													
Study of state of the art and background knowledg			▨	▨											
Requirements gathering					▨	▨									
Study of technologies							▨	▨	▨	▨	▨	▨			
Analysis and draw of architecture									▨	▨	▨	▨	▨		
Development of back-end (api)											▨	▨			
Development of back-end (campaign manager)												▨	▨		
Experimentation											▨	▨	▨		
Risk analysis														▨	
Write of the interim report											▨	▨	▨	▨	
Delivery of interim report															*

Description of work plan first semester

Below we describe in more detail each task from the work plan, that is present in previous diagram.

Environment and objectives Environment is the project vision and definition of the objectives to be completed in the internship with the supervisor.

Definition and analysis of business requirements Elaboration of a study about the needs of this product in the markets. While this was a research focused project, the needs of the fault-injection reserach community had to be understood.

Study of state of the art and background knowledge Study of state of the art as a way to support the development of this internship. Realization of a study about the products that exists, methodologies, architectures and technologies explored and conceptualized by the scientific community.

Requirements gathering Involves the process of requirements gathering, both functional and non-functional, in a first step by talking with some researchers of the SSE group and then some *brainstorming*, taking in account all the information collected. Development of low and high fidelity prototypes of solutions to obtain feedback from the researchers.

Study of technologies We studied the technologies used in the development frameworks (front-end and back-end).

Analysis and design of the software architecture This task was realized at same time as the study of technologies, by experimentation of alternatives, and also later during the development of some components of the platform. Same minor changes had to be introduced to accommodate later requirements.

Experimentation As the development of architecture went forward, at same time tests were being done to prove that f the design of the architecture was the most correct for the problem.

Risk analysis It was performed to identify and monitor risks that could negatively affect the project. The intermediate presentation was a strong driver for this activity.

Work plan for 2º Semester

	2016																					
	February				March				April				May					June				July
	8	15	22	29	7	14	21	28	4	11	18	25	2	9	16	23	30	6	13	20	27	1
Milestone 1				*																		
Milestone 2																*						
Development of back-end (campaign manager)	▨																					
Development of front-end (basic interface)	▨	▨	▨	▨																		
Development of front-end (user)					▨																	
Development of front-end (targets\machines)						▨	▨															
Development of front-end (campaigns)								▨	▨													
Development of back-end (automatization campaign										▨	▨	▨										
Development of back-end (analyzer)												▨	▨									
Development of front-end (reports)														▨								
Validation			▨	▨	▨										▨	▨	▨					
Write of the final report															▨	▨	▨	▨	▨	▨	▨	
End of internship																						*

Milestone 1: Deliver of the prototype full stack application, front-end and back-end, fully operation.
Milestone 2: Deliver of the prototype full autonomous capable doing analyzes of the data recovered from campaigns.

Excution work plan of 2º Semester

	2016																					
	February				March				April				May					June				July
	8	15	22	29	7	14	21	28	4	11	18	25	2	9	16	23	30	6	13	20	27	1
Milestone 1				*			*															
Milestone 2																*		*				
Development of back-end (campaign manager)	▨																					
Development of front-end (basic interface)	▨	▨	▨	▨	🟧	🟧																
Development of front-end (user)					▨	🟧	🟧															
Development of front-end (targets\machines)						▨	▨	🟧	🟧													
Development of front-end (campaigns)								▨	▨	🟧	🟧	🟧										
Development of back-end (automatization campaign										▨	▨	▨	🟧	🟧								
Development of back-end (analyzer)												▨	▨	🟧	🟧							
Development of front-end (reports)														▨	🟧	🟧						
Validation			▨	▨	▨										▨	▨	▨	🟧	🟧			
Write of the final report															▨	▨	▨	▨	▨	▨	▨	
End of internship																						*

The major delay that occurred in the project was by lack of experience from the technologies used in the project, taking several tasks related to the front-end to get major delays. Besides during the project there was a small change in the objectives. Taking some time of the intern to adapt the product to the feedback that was given.

In the tables above we present the work plan (initial expectation and actually executed) for the second semester. Since we used an agile approach, there was no initial 'plan' to follow, but rather a series of user stories that the supervisor kept prioritizing and updating as the work evolved.

3.3 Processes of Software Engineering

In this section we present the major processes and tools adopted to perform the project and develop mature versions in every release. There is also a brief discussion about the different development environments that the project used.

3.3.1 Version Control Software

In way to grant a better quality in the process of software development, there is a process of quality well set in order to guarantee a good management of the code repository and version control of the software produced.
We used the version control system Git, by the framework Bitbucket [1]. This systems is not centralized, but allows each programmer to keep their version of the code in their development area (commits). Using Git also allowed us to speed up the process of development of several components of software in a nonlinear way (branching/merging), since it allows the creation of several branches that represents different software features [38].
In the context of this project we used two remote branches one, master, that has the code tested and other one, development, that has all the features that are not yet fully implemented and tested. When the developer (the author) starts implementing a new software feature, he creates a new branch which contain the implementation of the necessary code for the development of this feature. A merge of a branch with the master, occurs only after the code is fully implemented and tested. This process is graphically depicted in the figure below.

[1] `https://bitbucket.org/`

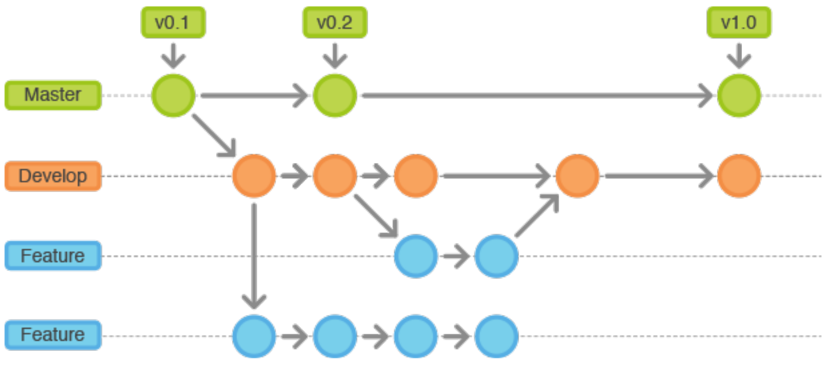


Figure 3.2: Structure of Git Branches[39]

3.3.2 Deployment Environments

During the development of the project it was necessary to use several deployment environments, in order to have different machines with different configurations to different purposes. The use of this process allowed us to have different technologies with different configurations and was also a way to get different data records for different reasons.
During the realization of this internship we adopted three different deployment environments:

Development environment

In this environment the author installed and configured the technologies to develop the software components needed in the context of the internship. This environment is more simple to allow the development to be more agile. In this environment are also performed all the unit tests as well as the first functional tests.

Staging environment

This environment presents the same configurations, but not the same resources of the production environment and has the objective to serve private tests and demonstrations. In here are done more functional tests and also usability tests.
Since it has the same configurations of the production environment, this allows that every feature will be stable when it goes into the production environment. This environment is updated in each sprint from the branch

of staging, with the purpose to showing the Product Owner and/or to the supervisor the new increment in the platform.

Production environment

This environment receives the version of the product that is publicly available. The application components and their databases are hosted in a private cloud (Linode [40]).
The production environment is updated in the beginning of each sprint, with the version of the application that was developed in the previous sprint, so that the production version can be first tested in the private area (the staging environment).

3.4 Conclusion

In this chapter we presented some of the main work methodologies used during the project, the work plan (planned and executed), and some software engineering processes that supported the work performed.

In the next chapter we will present the functional and non functional requirements gathered.

Chapter 4

Requirements Analysis

This chapter will describe how how we gathered the functional and non functional requirements for the project. We shall present the vision of the problem, the actors involved in the system, low and high fidelity prototypes and the specification of functional and non functional requirements.
To prepare the entire set of requirements, we started by gathering set of business requirements in order to obtain several key indicators of the adequacy of our future solution. First, we made an analysis of existing solutions, in order to know their key features. We realized that there is no product, commercial or open source, equivalent or similar to what we intend to develop. After that, we had several meetings with researchers of the SEE group, researchers that had already developed injectors, with the intention to see what were the real needs they want to see answered.
After this process, the requirements collected were discussed between the author, the product owner and the supervisor. This lead to a list of functional and non functional requirements to be developed as a the solution for the problem.

4.1 System Actors

In this section we will list and describe the role of the actors in the solution developed. Only two core actors were identified: Researcher (uploads a FI) and third party. A researcher assumes the role of 'third party' when wants to run a fault-injection campaign, as there is no conceptual difference between testing its own injector or use an injector created by someone else.

Actor	**Represents**	**Role**
Researcher	Person that uploads a fault injector.	Uploads a fault injection driver, to test it or to make it available to be used by a third party. This person can validate research data or test a cloud service.
Third party	Person that use the front-end	Use the tools present in the front-end, use a fault injector to test his system and see the report.

Table 4.1: Actors present in the solution

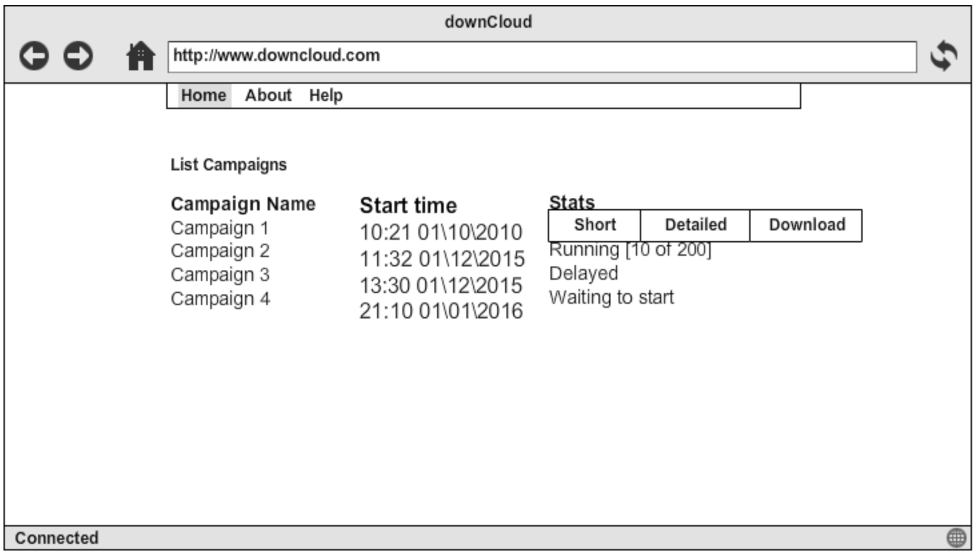
4.2 Prototyping

The use of prototyping is a technique useful during the process of requirements gathering, because allows the stakeholders[1] to clarify what they really want in a visual form.
The development of this work, in a first iteration using a low-fidelity prototype, allowed us to elaborate a first version of the system with few requirements defined, allowing us to specify an additional set of requirements desired. In a second interaction, it was possible to create prototypes that followed the requirements set in the first iteration, allowing this way a negotiation, with the stakeholders, to define additional requirements for the system.
In the early stages of development of prototypes, we addressed mostly decisions related to design and usability of the application since it should have a consistent interface in all the versions.
The prototypes elaborated are described below.

4.2.1 Low Fidelity Prototyping

In this phase we took some decisions related to features and related to the design of the solution. Below we show some examples of the prototypes designed.
In Figure 4.1 we depict the first version of the functionality stats of campaigns. This intend to demonstrate how the user can set and monitor the status of the campaigns that have been launched.

[1]Stakeholders: any person or organization that have interest or is affected by the project (e.i., client, project manager, analyst, developer.



List Campaigns

Campaign Name	Start time	Stats
Campaign 1	10:21 01\10\2010	Running [10 of 200]
Campaign 2	11:32 01\12\2015	Delayed
Campaign 3	13:30 01\12\2015	Waiting to start
Campaign 4	21:10 01\01\2016	

Figure 4.1: Low fidelity prototype of use case 'list campaigns'

In Figure 4.2 we depict the low-fidelity prototype of how the 'report campaign' use case would be presented.

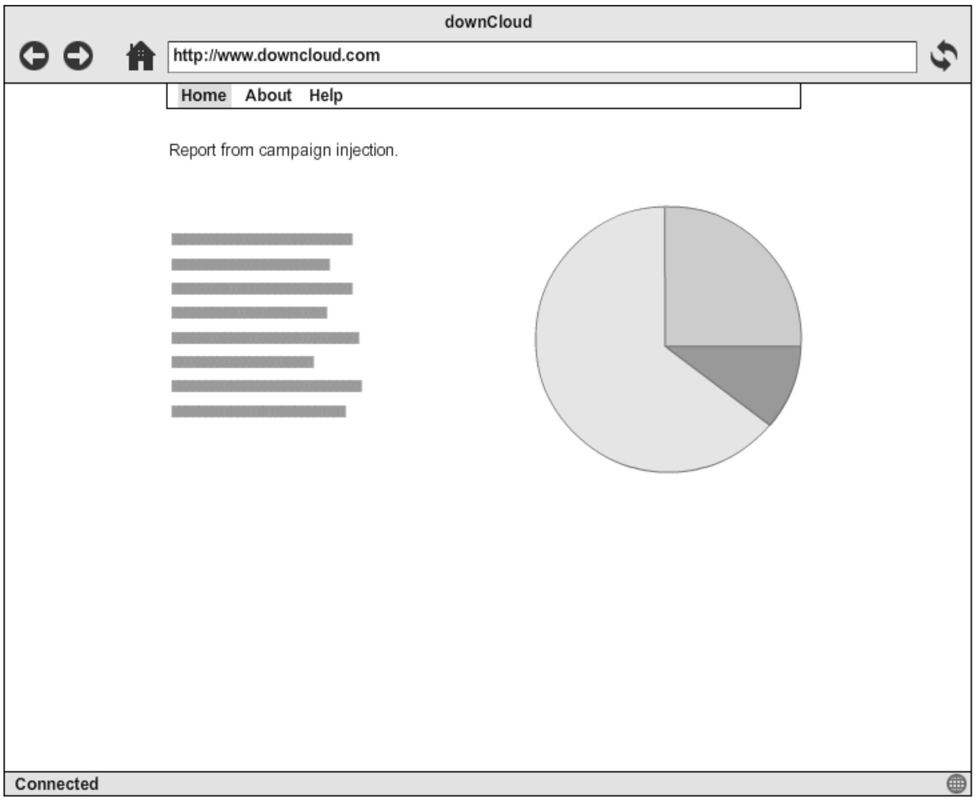


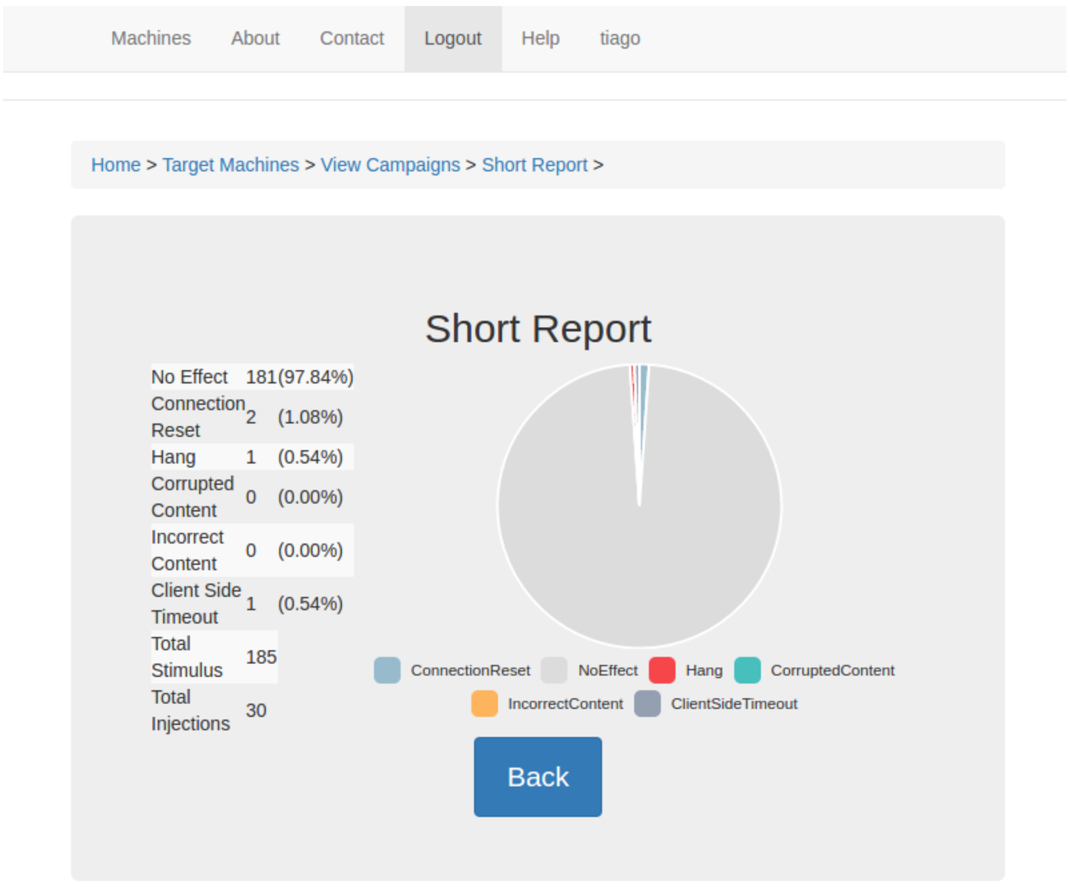
Figure 4.2: Low fidelity prototype of short report campaign

While simple, these first drafts were very effective to clarify the logic of operations and increase the author's awareness on how a fault-injection campaign is performed.

4.2.2 High Fidelity Prototyping

After the conclusion of the low fidelity prototype phase and after it being presented, discussed and validated with the stakeholders, a set of new requirements better adapted to the solution to be developed was elaborated. After the requirements were improved, it was possible to create a version of the system closer to the final version, using high fidelity prototypes. Some examples are shown below.
In Figure 4.3 and Figure 4.4 two high fidelity prototypes are presented. Both were validated with the stakeholders and its use case will be described in following sections along with its underlying functional and non functional

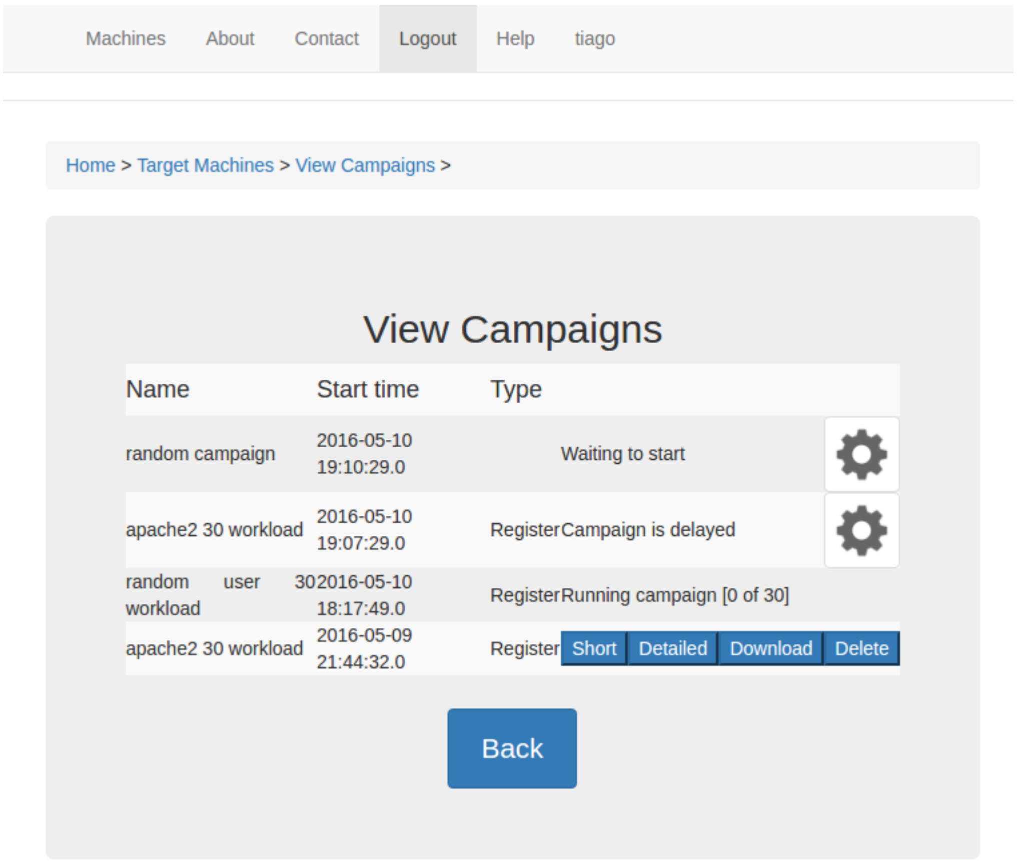
requirements.
In Figure 4.3 it can be seen the features and design of the use case 'short report'. It is possible to see the impact of faults injected, a measure of how well the server handled the injection campaign.



No Effect	181	(97.84%)
Connection Reset	2	(1.08%)
Hang	1	(0.54%)
Corrupted Content	0	(0.00%)
Incorrect Content	0	(0.00%)
Client Side Timeout	1	(0.54%)
Total Stimulus	185	
Total Injections	30	

Figure 4.3: High fidelity prototype of short report campaign

In Figure 4.4 is depicted the features of the use case 'managing campaigns' in a machine. Here the user can see the status of all the campaign and the type of injection performed. There are four possible states for campaigns: i. waiting for start (was schedule and is waiting for the correct time), ii. campaign delayed (the campaign was launched to run but there is already another running so it must waiting for its finish to start), iii. running campaign (the fault injection campaign is being performed) and iv. concluded. When a campaign is concluded it is possible to observe either a preliminary report Figure 4.3, or a detailed report with the moment of every injection with parameters and the tests that went wrong; This report can be downloaded for analysis using external data analysis tools.



View Campaigns

Name	Start time	Type		
random campaign	2016-05-10 19:10:29.0		Waiting to start	⚙
apache2 30 workload	2016-05-10 19:07:29.0	Register	Campaign is delayed	⚙
random user 30 workload	2016-05-10 18:17:49.0	Register	Running campaign [0 of 30]	
apache2 30 workload	2016-05-09 21:44:32.0	Register		Short Detailed Download Delete

Figure 4.4: High fidelity prototype of short list campaigns

4.3 Functional Requirements

The specification of functional requirements to be developed was realized using user stories, which are typical of agile development. A user story is a short and simple description of an interaction that an actor does in the

system [42]. The biggest advantage of using user stories in the specification of functional requirements relates to the fact that these force us think about what to develop in the view point of the final user; moreover, they are simple and fast to write, easy to any client without technical knowledge to understand and easily mapped to acceptance tests to validate the implementation of each feature.
For the definition of user stories one of the most used formats is textual, based in three expressions, which answer three questions related with each feature [43]:

- **As a/an** <actor in the system >(Who?)
- **I want to** <action >(What?)
- **So that** <benefit >(Why?)

The definition of priority in the requirements is very important to decide the implementation order according to the 'business' value seen by the client. We adopted the key words defined in RFC 2119, "Must", "Should", "Could" and "Optional" to prioritize the features to be developed.
We shall now present a brief synthesis of each requirement. The complete description of all requirements can be found in Annex 9.3.

ID	**Resume**	**Priority**
DC-01	Add new target system	Must
DC-02	Test if target system is accessible	Must
DC-03	Setup a campaign to start in the future	Must
DC-04	Lunch a campaign to start now	Must
DC-05	See a short report of the campaign results	Must
DC-06	Download a detailed report of a campaign	Must
DC-07	See the status of campaign	Must
DC-08	Receive an email notification of the end of campaign	Optional
DC-09	Upload an injection driver	Should

Table 4.2: Functional requirements

4.4 Non Functional Requirements

The goal to realize the gathering of non functional requirements is to describe the qualities the system must have, such as usability or performance. Since the user stories are not suitable for the elaboration of this type of requirements, there was no specific restriction to make them.

Scalability

The system must have the capability to increase his performance, under an increase payload in application level, without the need of changing the architecture of the system or loss of service due to the time need to resize resources.

Robustness

The system must ensure all the information integrity, either in stress situations or in application failure situations. It is not allowed to loose any information related to campaign injections that were performed. The system should have the capacity of recover information that was not processed in that same day.

Security

All accesses to the application and to the data must only be allowed for authenticated users. Also we must ensure the anonymization of all data. Finally, all communication external between the clients and the application must be encrypted.

Costs

The infrastructure most have the capacity to be able to adapt to the increasing or decreasing needs of the business (pay-as-you-go).

The platform architecture, described in the next chapter, was designed to support these non-functional requirements.

4.5 Conclusion

After this chapter the reader should know which are the actors of this system and the major functional and non functional requirements. In the next chapter we will present the software architecture of the platform.

Chapter 5

Architecture Analysis and Design

In this chapter we recall the initial problem and a brief view of the solution, detailing the architecture that will support the non.functional requirements. We present a short small description of each component and of how the the system works.

5.1 Architecture of the Solution

In this section we present the software architecture, i.e. the software components of the project *downCloud*, how they communicate with each other and how the client can use this application.
The *downCloud* platform is divided into three major components: front-end, web services and campaign controller, as can be seen in the image below. Figure 5.1.

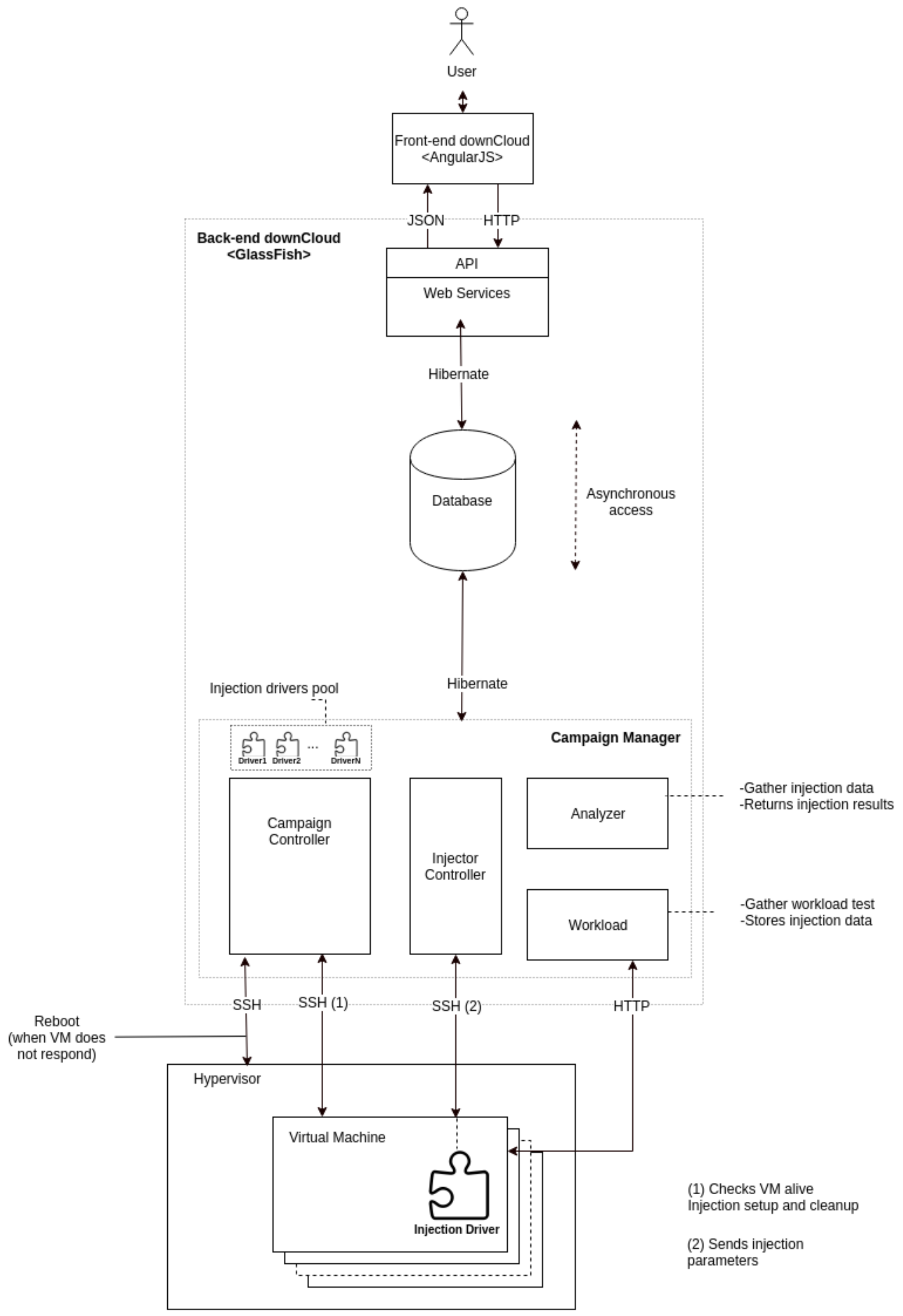


Figure 5.1: Global perspective of the architecture

Front-end downCloud The front-end is where the client interacts with application to manage the machines, the targets for the campaign injections, manage the campaigns injection and also see the result of them. For the development of the front-end we chose to use the design pattern Model-View-Control (MVC), for an easy maintenance and evolution of the tool.

Back-end downCloud The back-end is responsible for launching new campaigns, at the scheduled times, initialize the database if necessary, and make connection to it. For the development of the back-end we also adopted the design pattern Model-View-Control (MVC).

Web Services This module will provide a **REST API** to be used by the front-end or directly by other applications to access the functions of *downCloud*.

Campaign Controller The Campaign controller the component that makes the campaigns to be autonomous. It controls the Injector Controller and Workload operations to ensure autonomous operation, e.g. by checking that the virtual machine, target of the campaign, is operational. If it detects any problem in the target it will force a reboot and resumes injection.

Injector Controller This module uses an 'Injection driver', a kernel module inside the target system, to inject faults with the parameters that the user requested.

Workload This is the source of the HTTP requests that are made to the target system to exercise the system and to assess how the system is dealing with the fault injection by evaluating the answers to the HTTP requests. We remind the reader that we have scoped this first version of *downCloud* to focus only on web based applications.

Analyzer This component runs during and after the campaign to analyze all data collected during the campaign injection to deliver a synthesis report to the user about his target system behavior.

Injection driver This is the module, resident in the target system, that actually performs the injection. There is a pool of many possible 'injection drivers' provided by researchers, with different capabilities. For the present thesis we used a 'CPU register bit-flip injector' to emulate transient hardware errors.

5.2 Technologies Used

In this section we present the technologies that were used in the project *downCloud* and a brief explanation why they were choose over other possible technologies.

REST

Representation state transfer (REST) is a web standards based architecture that uses HTTP protocol for data communication. Unlike some more complex protocols, like SOAP, REST allows the system to take care of data and to be showed in XML, JSON or HTML format, making easier the integration and communication of information. [24]
Since the development of *downCloud* is to be full-stack, the client/supervisor decided that the back-end, should have an API allowing the communication with other components besides the front-end.

Java

Java is a programming language and computing platform first released by Sun Microsystems in 1995. *Java* is a very well-design and mature platform from the beginning without ever ceasing to innovate, have existed so far without introducing any major backwards-compatibility regressions, works in many hardware environments and it's open source. [25]
Since *Java* is a programming language well mature that is being improved over the time and since the author has a good working knowledge of it, it was decided to be the back-end development language adopted.

Hibernate ORM

Hibernate ORM is an object-relation mapping framework for *Java*. It provides a framework for mapping an object-oriented domain model to a relational database. *Hibernate* provides all the benefits of other object/relational

mapping tools, is free and open source, the learning curve is short since *Hibernate* ORM is totally object orientated and has lot of support from an active community. [26]
So it was decided to use *Hibernate* ORM in the project to facilitate the management of the SQL database.

GlassFish

GlassFish is an open-source application server project stated by *Sun Microsystems*, in 2005, for the *Java EE*. *GlassFish* have a good administration and management interface, good support by development tools, lost of information and is fast to start up and deploy an application. [27]
It was decided to use *GlassFish* in the project since is easy to maintain and reduce the effort of deployment.

AngularJS

AngularJS is a *JavaScript* framework designed to make front-end development as easier as possible. *AngularJS* is a framework built and maintained by *Google*, is a complete solution for rapid front-end development, extends in HTML and incorporate unit enterprise-level testing. [28] [29]
Since the author did not have any knowledge of any front-end framework and *AngularJS* has a big community and also have skilled engineers of *Google* to help, *AngularJS* was the technology choice to the front-end framework of the project.
As can be seen, our choice of technologies was quite conservative: we intentionally decided do not experiment with more recent (but also more immature) technologies, as our goal was to focus on the problem using the best known technologies that 'fit' well the problem.

5.3 Architectural design strategy

During the design of the architecture described previously in this chapter, the scalability aspect was a key driver. We took the Design-Implement-Design (D-I-D) strategy [44] to keep refining the architecture.
The architecture was designed to ensure perfect independence of components, from application to the physical (server) level, thus allowing different ways of achieving scalability for each software component and also ensuring the robustness of the whole system. It must also be pointed out that the architecture designed by the author has also the capacity to be scalable horizontally

for all the components, without the need of changing the application, physical hardware or technologies adopted.

5.4 Conclusion

In this chapter we presented the architecture designed to solve the problem and how it addresses the major quality attributes required (scalability, availability).
In next chapter we address the development work involved during the project.

Chapter 6

Development work

In this chapter we describe the development work performed during the project. This concerns more the coding phase (low level design and implementation).

6.1 Injection Campaigns

In order to fulfill the functional requirements related to the creation and execution of campaigns, it was developed code that manages several asynchronous processes to run and test a target system.
The main goal of this code is to inject faults while the system is being stressed, services being accessed, and keep the target system working, while at same time compare the data that is being collected from the target system with the one that is expected to receive (gold run). This is how we know the target system resilience to errors.
In this section we show the approach used for the implementation of this code and how it has been validated.

6.1.1 Problem Definition

The campaign injection must be fully autonomous, from the beginning to the end, with no need of human or script intervention. Since there is injection fault occurring, the system can have several types of errors; the worst behavior is an 'hang', where the system freezes without providing any feedback. Other behaviors involves the injector driver (resident in the target system) being blocked by the operating system (if it activates security mechanism of the OS), or the communication channel, SSH (Secure Shell), suffer a fault injection and being forced to close. Every possible weird behavior can occur under fault-injection and that is one of the major challenges of designing an unattended fault-injection tool. Having this in mind we elaborated a sets of scenarios and recovery mechanisms that our tool had to support.

Objectives

It should be possible to have a campaign running without stopping, thus needing human intervention to restart. The tool should inject faults in the target system, restart when needed, exercise it using a given workload and be able to evaluate the correctness of the responses provided by the target system to the workload.

Requirements of the algorithm

The algorithm developed should be able to deal with the following scenarios:

- **The target system is hung:** the target system can have different reactions to the injections, one of them, the more critic, the system getting hang providing no answer to any request. When this behavior is detect, the algorithm should stop the injections and the workload and restart the target system. Then, when the target system is up again, continue the campaign;

- **The injection driver has been blocked by the kernel:** the operating system kernel has several mechanism that protect processes from loaded components misbehavior, and when using the injector fault it's behavior can trigger one of this safety mechanism. When this happens, it causes the injector to get block and unable to perform any more injections;

- **Be able to analyze the workload results:** during the process of exercising the target system by using a workload, we need to compare the replies before the campaign injection starts with those during fault-injection to detect erroneous behavior of the target system (this implies that the fault injected had no fatal impact in the target).

6.1.2 Our approach to achieve these goals

After the definition and analysis of each specific scenario, we developed possible solutions for the resolution of the problems raised.

Execution Campaign Injection

We shall now describe how a campaign is executed and how it deals with problems, such as hang or the injection driver blocked by the operating system.
The user sets a campaign in the desired target system, and *downCloud* will wait until the scheduled time.
The first thing to be done is to send the source code of the injection driver (which the user previously selected) to the target system and compile it locally. This is needed due to the many diverse configurations possible, thus ensuring portability across different platforms. After the compilation is complete, the system will perform a gold run with the services (workload) the user provided. The workload had also been previously uploaded to the *downCloud* platform. The tool will run every service once before any injection to be performed and save the result for comparision during the campaign, to analyze if the the answer is the expected or not and, if not, identify what kind of failure occurred (well formed reply but wrong content, malformed reply, hang, etc...).
After the gold run is completed, the system will start the campaign injection in the target system. The campaign controller will first start a new process with the workload; the workload will choose randomly a service to exercise the target, executes it and compares the result obtained with the correct result provided by the gold run. This result can be classified in six distinct categories: i.no effect, ii. corrupted content, iii. incorrect content, iv. connection reset, v. client side timeout and vi. hang. The characterization of each one will be presented below.
With the workload running the campaign controller will also start the injection controller; this component is responsible for selecting the injection parameters among the several parameters that the user has chosen and send it to the injection driver. This selection is necessary because the parameters that the user can chose do not need to be specific, but can be several alternative values (e.g. register R0 or R1 or R15) or a range of values (e.g. register R0 to R5). The injection controller then waits for the injection driver to inject a fault in the target system.
After each injection the campaign controller performs some tests (e.g. ping), runs some linux commands (e.g. opens a ssh connection and performs a remote directory listing), to check if the target machine is operational and confirm whether the injection driver was blocked or not by the remote kernel. If during the tests it detects that the injection driver is blocked it stops the workload and the injector controller and forces a reboot of the target system through the hypervisor (remeber we are targeting a cloud platform, i.e. a

virtualized environment) . After the reboot is completed the code of the injection driver is resent and compiled again, after which the workload and the injector driver are started again and the campaign continues.
If it is detected that the target system got hang, the procedure is the same: the campaign controller stops the workload and the injection controller, declares the target system as hang and starts the reboot process. After the reboot is done, the campaign controller will start the workload and the injection controller and continue the fault injection campaign.

After the conclusion of the campaign the system cleans the target system, removes the injection driver, and reboots the target system to clear any possible errors that could have remained latent.

Classification of Injection Outcomes

During our campaigns we classify the target system behavior according to a client oriented view, i.e. we classify the results as the client perceives them. These behaviors can be classified in the following categories:

- **No Effect:** The injection error has no visible effect on the provided service, both in terms of performance and correctness;
- **Incorrect Content:** Occurs when the answer sent by the target system is syntactically correct HTML, but with an incorrect content, for example words missing or a wrong hash;
- **Corrupted Content:** In this case the the answer provided is corrupted data (syntactically incorrect HTTP packet, invalid HTML code or just pure garbage). For example, a browser would handle a similar occurrence by displaying an error message;
- **Connection Reset:** The TCP connection between the server and the client is reset by the server's network stack;
- **Client Side Timeout:** Occurs when the answer from the server exceeds the time limit predefined (we used five seconds), and issues a client side timeout;
- **Hang:** The target system stops producing any output and answering to any subsequent requests. Eventually all requests made to the target system will issue a client side timeout.

Data Model

In this section we present the data model developed, that supports the campaign injection. The data model was developed in MySQL, a relational database system.
The full description about the data model can be consulted in Annex 9.4.

Campaigns Represents all the campaigns in general; here is stored all the general information about the campaign information not related to the attack, but information related to when to inject (the time), and how to inject (the process).

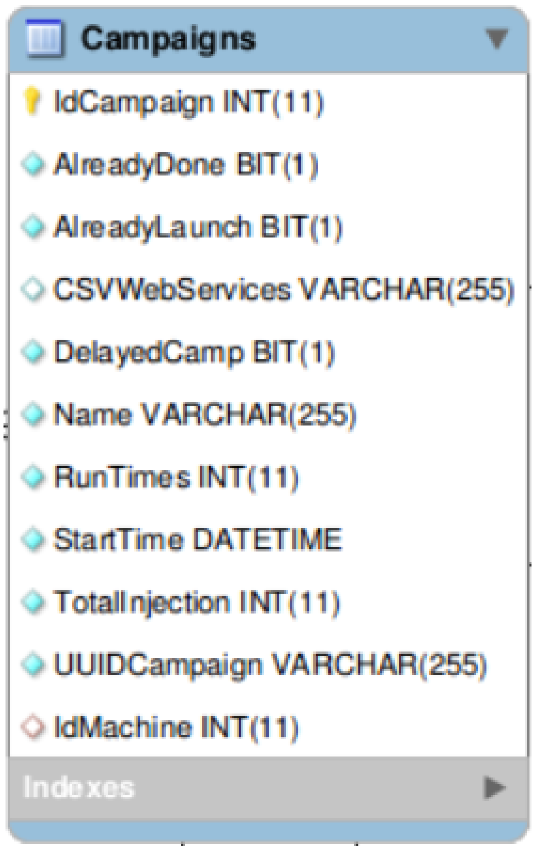


Figure 6.1: Data model of Campaigns

CampaignsRegister It represents the injection driver that attacks the cpu registers with a flip bit. It stores all the information related to the parameters to the CPU registers' injection.

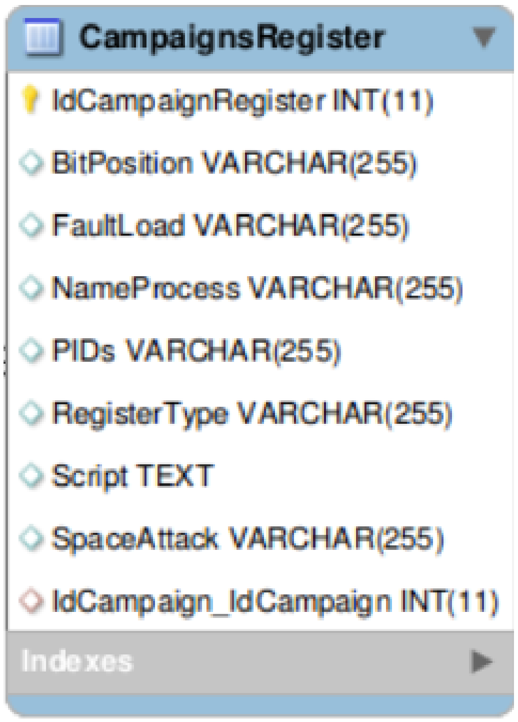

CampaignsRegister
IdCampaignRegister INT(11)
BitPosition VARCHAR(255)
FaultLoad VARCHAR(255)
NameProcess VARCHAR(255)
PIDs VARCHAR(255)
RegisterType VARCHAR(255)
Script TEXT
SpaceAttack VARCHAR(255)
IdCampaign_IdCampaign INT(11)
Indexes

Figure 6.2: Data model of CampaignsRegister

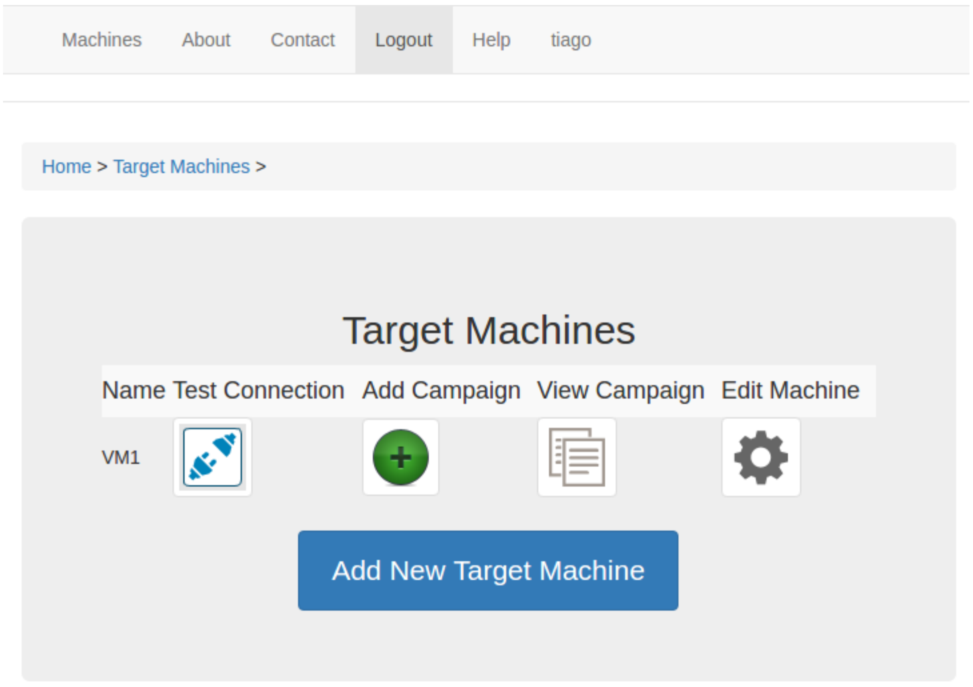
Implementation

As refereed in the architecture (section 5.1) the campaign injection is implemented in the back-end, in the component 'campaign controller'. The campaign manager controls when is the correct time to start the campaign controller that leads to the start of the other processes, injection controller and workload.

6.1.3 Interface and Features

In this section we present the implemented features that allow the visualization the evolution of a campaign and the results of it.

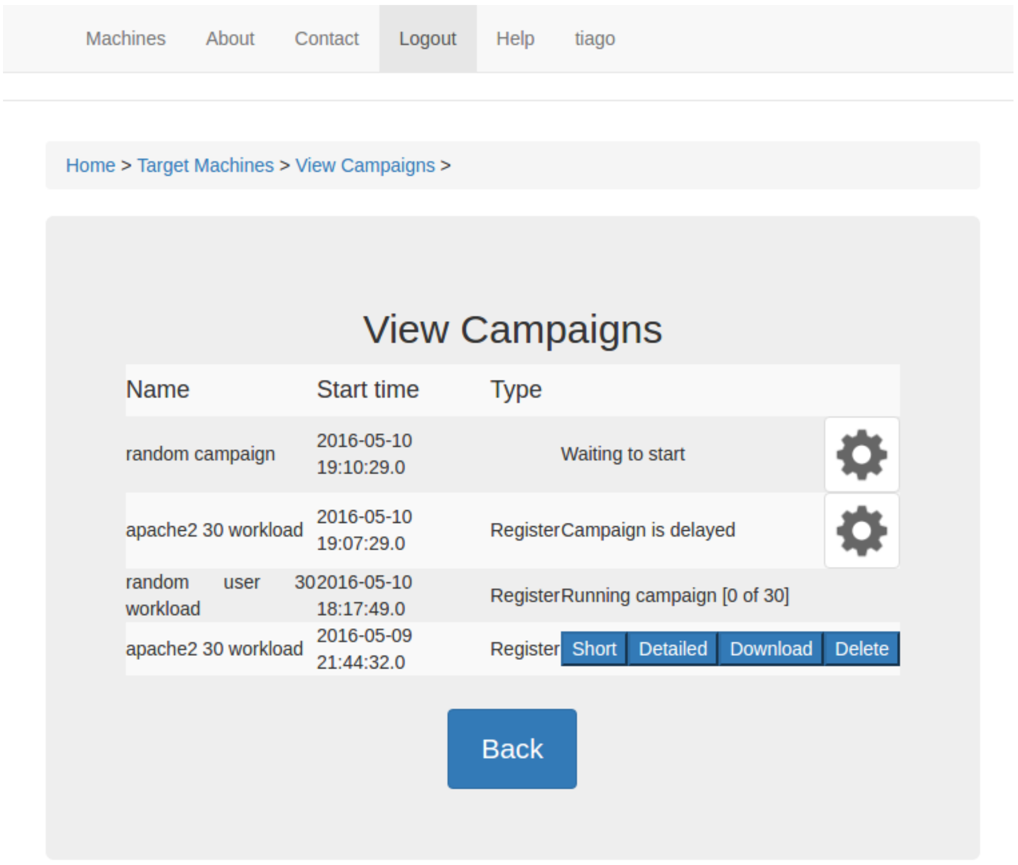
In the image 6.3 we show the interface implemented, that shows to the user the available target systems. From this menu he can test the connection to the target system (reachable?), add a new campaign, view campaigns related to the target system and edit the campaign paramenters.



Name	Test Connection	Add Campaign	View Campaign	Edit Machine
VM1				

Figure 6.3: Interface developed to visualize all the system target the user have

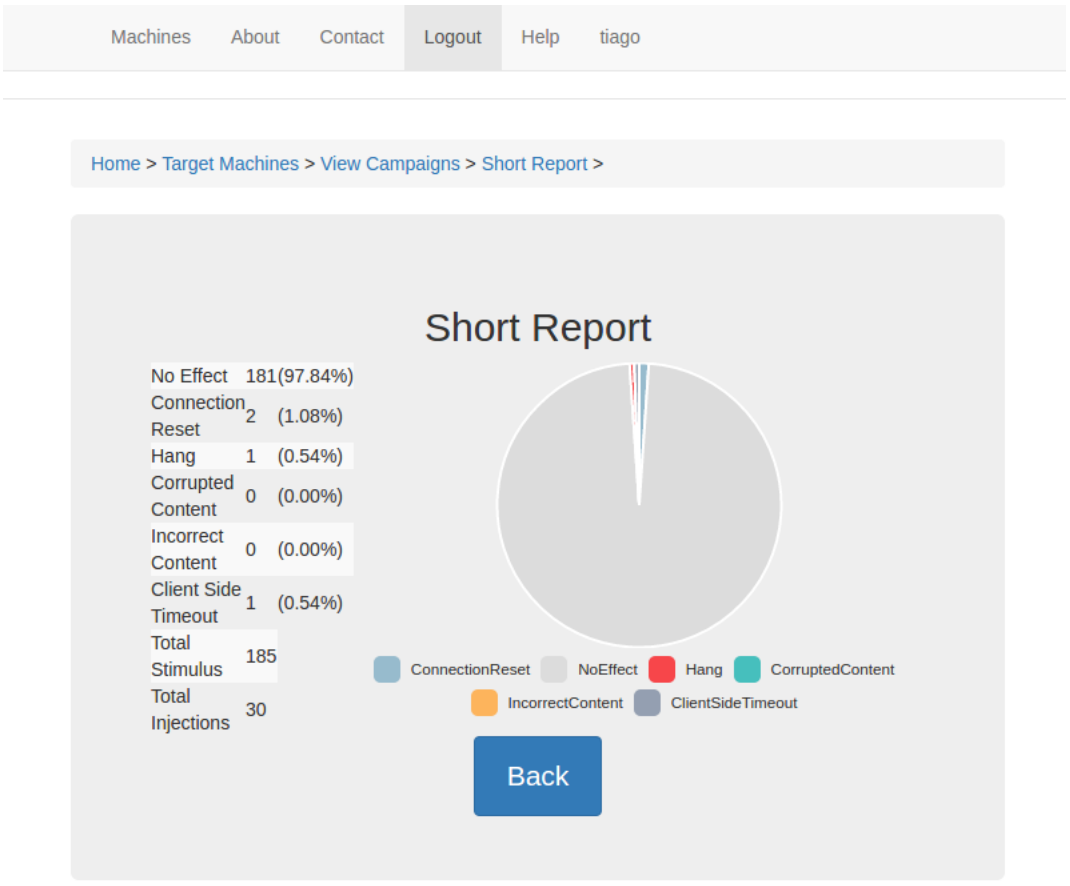
The image 6.4 depicts the interface implemented, to show to the user the campaigns that are set or finished. This screen provides the user with a short report or a detailed report from the campaigns concluded or download a full report of it. If a campaign is not running or is being delayed, because other campaign is running (in the same target) , the user has the possibility to edit that campaign (i.e. campaigns already started cannot be edited).



Name	Start time	Type	
random campaign	2016-05-10 19:10:29.0	Waiting to start	⚙
apache2 30 workload	2016-05-10 19:07:29.0	RegisterCampaign is delayed	⚙
random user 30 workload	2016-05-10 18:17:49.0	RegisterRunning campaign [0 of 30]	
apache2 30 workload	2016-05-09 21:44:32.0	Register	Short Detailed Download Delete

Figure 6.4: Interface developed to visualize all the campaigns and the evolution of their execution

In image 6.5 we present the interface implemented, that displays to the user the synthesis results about a campaign injection. This interface shows, on the right side, the results in a pie chart; when the user pass the mouse over the pie chart an hovering legend is displayed. On the left side the numerical results of the injection as well as the total of stimulus made (http requests), and the total number of injections performed during the whole campaign.



No Effect	181	(97.84%)
Connection Reset	2	(1.08%)
Hang	1	(0.54%)
Corrupted Content	0	(0.00%)
Incorrect Content	0	(0.00%)
Client Side Timeout	1	(0.54%)
Total Stimulus	185	
Total Injections	30	

Figure 6.5: Interface developed to visualize the results obtained during the tests

6.1.4 Features Developed

In the architecture chapter (chapter 5 we referred that the functionalities concerning the campaign injection running, are implemented in the back-end.

The front-end component uses the framework *AngularJS*, thus allows the application logic to be implemented in the client side (JavaScript) and develop the interface and templates styles (HTML, CSS) in a dynamic way. The features developed allows to support the following operations related to the functional requirements:

- Add a new target system;
- Test if everything is ok with target system (reachability; meaningful responses)
- Schedule a campaign to start in the future;
- Setup a campaign to start now;
- See a brief report of a finished campaign;
- Download a full report of a finished campaign;
- Check the status of a campaign (scheduled, waiting, active, finished);

6.1.5 Validation

To test if everything was correctly implemented we performed several campaigns, some to attack the Apache web server and others to attack random processes of the kernel.
The campaigns occurred at same time in different virtual machines in the same physical machine with the following configuration, table 6.1.

CPU	Intel Core i5-6500T Processor 4 physical cores running up to 3.10 GHz
Cache	6 Mb
RAM	8Gb DDR3
HHD	2 of disk 400Gb

Table 6.1: Hardware Specification of hypervisor

For this experiences we had three virtual machines, each one with one core of CPU and 1 gigabyte of RAM. The virtual machines were running the same system image and same workload.
The workload consist in a scenario where multiple external HTTP clients simulated accesses to several pages in an Apache. This webserver serves three different web pages, where one of its content, in every page, is the result from a SHA1 hash, using as input a stream obtained from /dev/zero. This workload stresses the system's CPU and its memory. In our campaigns the stream size was 1024 megabytes. The results of selected campaigns can

be seen in Annex 9.5.1.
The results collected in the campaign targeting the kernel are all very similar (no erroneous output was generated), because almost always after each injection done to the system the kernel blocked the injection driver. This required a target restart almost every time after each injection, in order to bring the target system back to its initial state.
On the other hand the campaigns targeting the Apache server produced interesting and more varied results and by the law of large numbers, the average of the results obtained from a large number of trials is expected be close to a realistic value. It was observed that the results of the campaigns converged to the same values after each experiment. These values also agree with one obtained by the researchers in their 'handcrafted' experiments not using the platform.

6.2 Conclusion

After this chapter the reader must have a detailed understanding of how the injection campaigns have been designed to support autonomous operation even in the presence of hangs and blocking of the injection driver by the target kernel.
In the next chapter we shall focus on the acceptance and stress tests to validate the platform.

Chapter 7

Validation

In this section we show how we validated the features implemented in the solution. Beyond unit testing at the component level, we performed acceptance tests with the product owner, in each *sprint* of development and at the end of each milestone.
We also performed workload tests with the intention of validating the quality attributes of the system. These tests were focused on validating the robustness, availability and reliability of the application in extreme conditions (stress) and to identify the components that can create a performance bottleneck in the system.

7.1 Acceptance Tests

This kind of tests have as main objective the verification of the functional requirements implemented, and ensure that the solution works the way expected by the final user. The elaboration of these tests in parallel with the development allowed the product owner to contribute in a constructive way to the internship progress and planning of the sprints during the development process.
The acceptation tests realized can be consulted in Annex 9.5.

7.2 Stress Tests

During the internship we had no opportunity to perform tests in a commercial cloud environment due to the costs involved; however these tests were realized in the staging environment (a private cloud) with the same configurations of the production configurations.
Since the capacities of the machine in the staging environment is not the

same of what can be found in commercial cloud environment, we did not achieve the same values. However, taking in account the numbers obtained in this private environment, it is legitimate to assume that with the resources available in a commercial cloud environment we will achieve an even better performance, since the need of scale depends in the number of the clients and not on the projected architecture.

7.2.1 Stress Test Scenario

Display results of campaign After authentication in the system, access one of campaigns in the list of campaigns in one of target system and select the short report to view and analyze the information available.
We selected this test case as the one that is most computationally demanding to the system. Injecting faults is very light, as most activity is performed by the injection driver in the targets. During injection our platform is actually idle, waiting for results.

Conditions of test

Workload	**Wait time**	**Running time**
100 concurrent users	3 to 4 seconds	30 minutes

Table 7.1: Tests conditions to view and analyze reports

Of 100 concurrent users, about 20% of them, have access the page of short report of campaign.

7.2.2 Observations and improvements

In a first interaction of this scenario, it was possible to obtain a mean throughput of thirty reports in one second. Since this number does not meet our self-imposed minimal requirements, we carried out a tuning of infrastructure.

1. **Replication of data base MySQL:** with this modification it was possible to obtain a mean of fifty answers for second. However we realized the bottleneck was not in the data base, but in the web application.

2. **Replication of web stack:** we implemented a load balancer and replicated the back-end, which led to an increase of the mean throughput to one hundred reports in each second. This already met our acceptable threshold.

7.3 Conclusion

In this chapter we described how we validated the platform, both in functional and qualitative terms, namely on what performance (response time) is concerned. We also presented the solution to circumvent the performance bottleneck identified.
In the next chapter we conclude this thesis with a brief reflection on the overall work performed.

Chapter 8

Conclusion and future work

In this last chapter we present the main obstacles found, the experience acquired and the future evolution of the project.

8.1 Competences achieved

The features developed during this internship, for the project *downCloud* involving the support to run campaigns and analyze of the data collected, are now stable and ready to be used in production. The first bet testers outside the SSE group have already started to use the tool.

The integration of the author in this project was, without doubt, a big contribution to his education as a professional in software engineering, since the intern needed an adaptation to the project, the processes and the methodologies used along the development. This adaptation process occur in the best way possible, since the author had an easy contact with the product owner and the supervisor. This increased our capabilities to deal with real world projects, as we had already some previous work experience.
Besides the competences acquired connected to software engineering, methodologies of agile development, gathering requirements and management of tasks, this internship allowed the intern to acquire technical knowledge related to front-end development, relational data bases, architecture principles of scalability and frameworks like *AngularJS* and *Hibernate*.
In brief, the experience and knowledge gathered during the internship will allow the intern to be a professional with better capacities and better prepared to the job market.

8.2 Main difficulties overcome

At the beginning of this internship there were some periods of indefinition, regarding the internship and the requirements of the project, which took some weeks to become more concrete.
Since the intern did not have any experience in use agile methodology in development projects, there was some initial difficulties to adapt to this process, having overcome this difficult as the project progressed.
Other difficulty felt was the lack of knowledge and experience using the framework *AngularJS*, it took some weeks to the intern could use the framework without (many) problem.
Another difficult felt during the project was the process of software testing and quality. This is an area the intern did not have almost any knowledge or experience. It was not possible, due to logistics and planning issues, to realize a better stress test plan, in order to define a better sizing of the resources for the infrastructure in a cloud server.

8.3 Future Work

In the beginning of the internship the requirements for the project were not very clearly defined and the development took place with this problem. Thus we started using an agile approach with the intention of creating a minimal viable product in every interaction, without the need to go back to revise and upgrade the whole code. However, at the middle of the second semester, during a meeting with the client/supervisor, there was a small change in the requirements that had a big impact in the project. This change lead to a significant increment in the complexity of the solution. Now the core requirements are fixed in the way the client wants the final product. However, after all the experience gained along the project, the author would consider, in *downCloud* 2.0 version, a refactoring of all the code from scratch, even knowing that in hindsight, we could have always made better. Moreover, the next versions of *downCloud* should include support for different targets (the current version targets only web services) as well as non cloud-based platforms, such as embedded devices, i.e. target the full internet-of-things stack.

Chapter 9

Annex

9.1 Annex A - Artifacts of agile methodology

In this section like said in section 3.1 were will be showed the artifacts related to the agile methodology, produced during the internship. In here it will be presented the *backlogs* of which *sprint* and its *burndown chart.*

Sprint 1

Task	State	Story Points
Prototype of web injector	Done	12

Table 9.1: Sprint 1

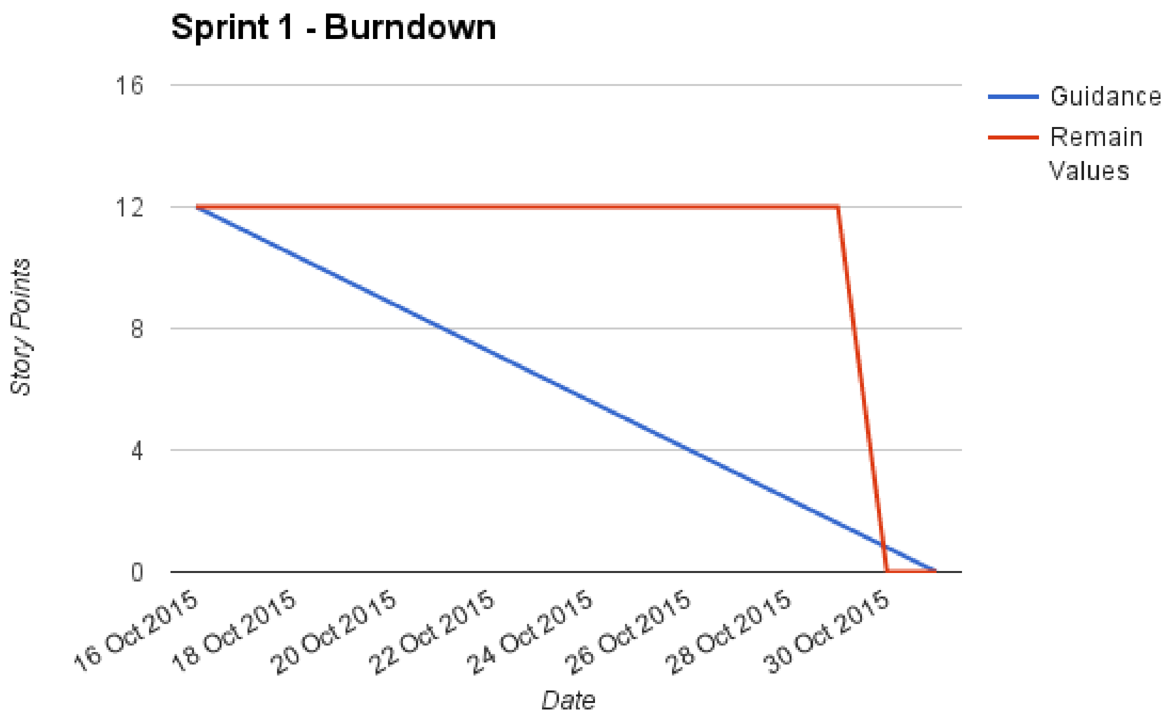


Figure 9.1: Sprint 1 Burndown

Sprint 2

Task	State	Story Points
Database and hibernate configuration	Done	5
Creation web services for User	Done	3
Creation web services for Machines	Done	5
Creation web services for Campaign	Done	5
Creation mechanism for launch Campaigns	Done	3
Creation mechanism for cleaning Tokens	Done	5

Table 9.2: Sprint 2

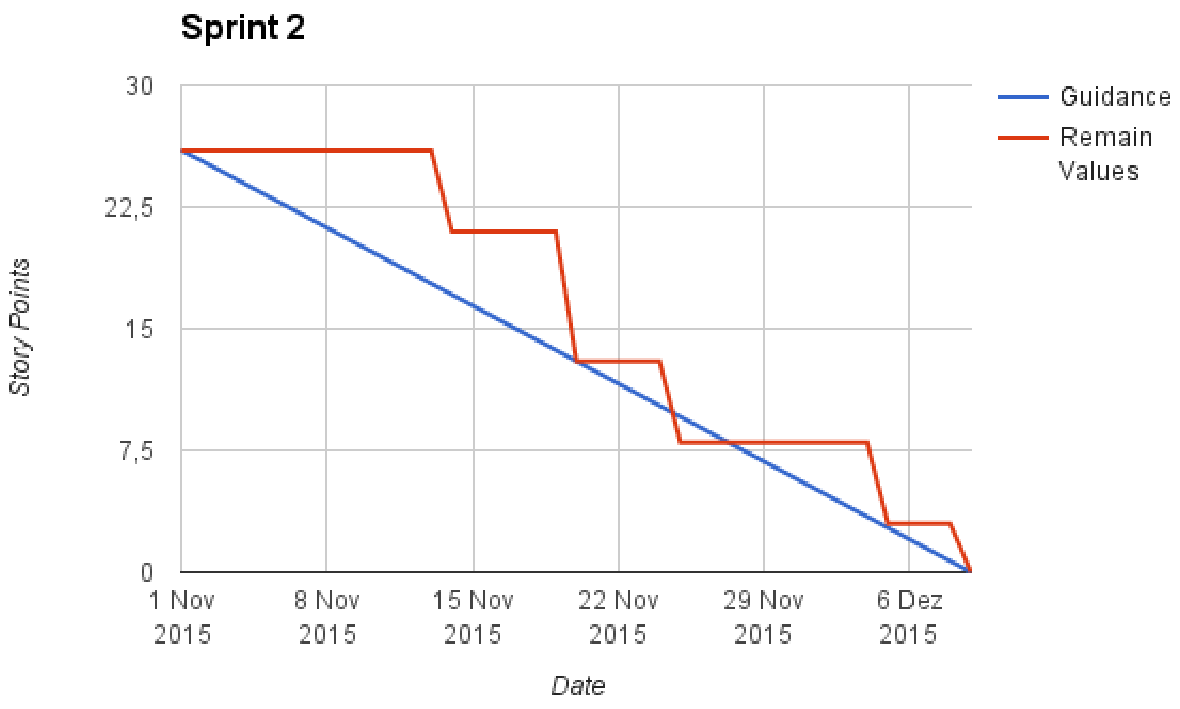


Figure 9.2: Sprint 2 Burndown

Sprint 3

Task	State	Story Points
Learning AngularJS	Done	15
Start development of front-end	Done	10
Basic login and register of User	Done	10

Table 9.3: Sprint 3

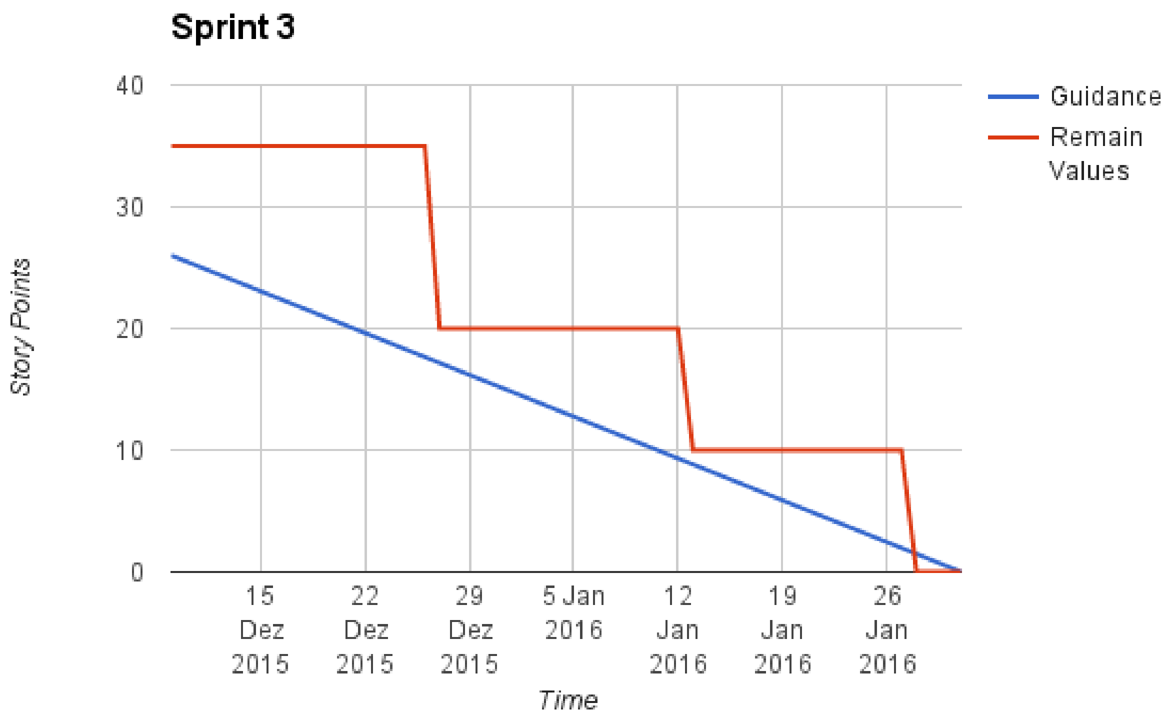


Figure 9.3: Sprint 3 Burndown

Sprint 4

Task	State	Story Points
Creation front-end for Machines	Not done	10
Creation front-end for Campaigns	Not done	10

Table 9.4: Sprint 4

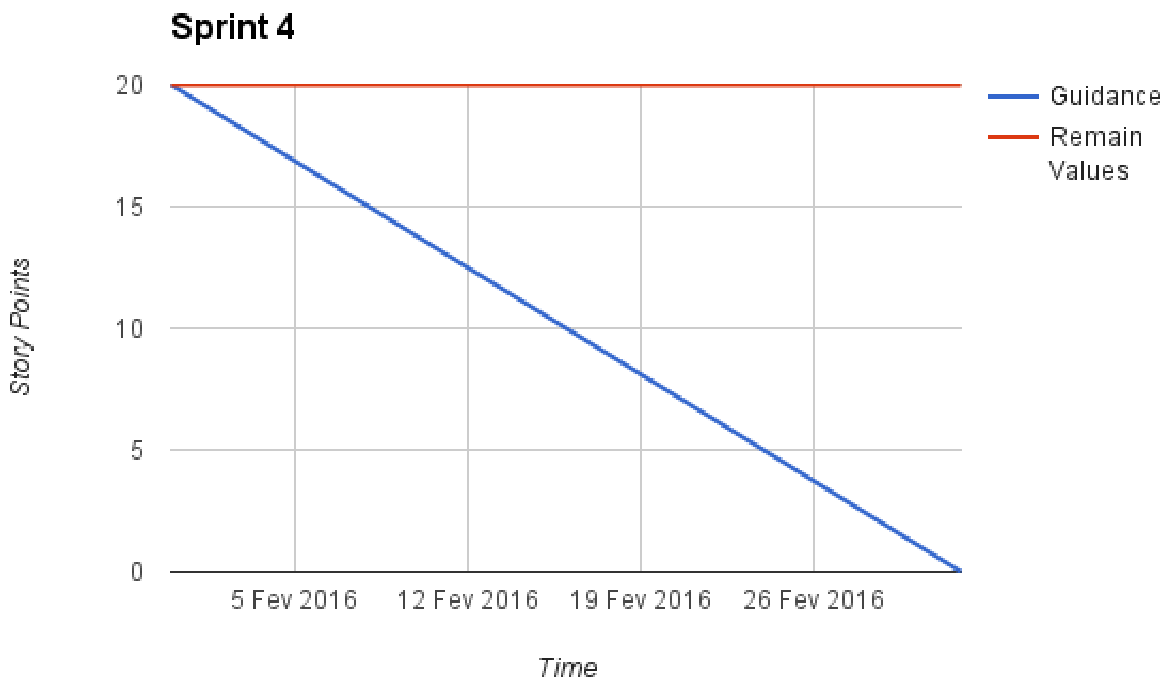


Figure 9.4: Sprint 4 Burndown

Sprint 5

Task	State	Story Points
Creation front-end for Machines	Done	3
Creation front-end for Campaigns	Done	3
Test of firsts campaigns	Done	3
Correction of critical bug in back-end	Done	3

Table 9.5: Sprint 5

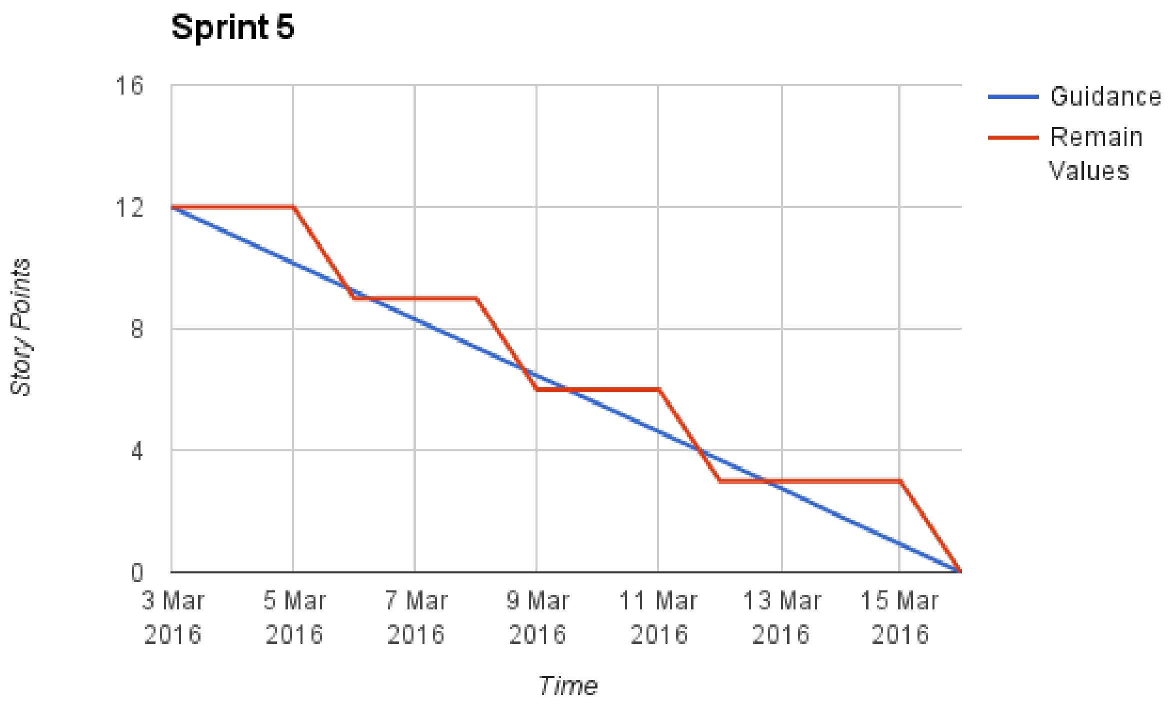


Figure 9.5: Sprint 5 Burndown

Sprint 6

Task	State	Story Points
Correction of bugs in front-end	Done	5
Adaptation of front-end with feed back of client	Done	5
Add new options parameters for Campaigns	Done	15
Construction of Analyzer	Done	15

Table 9.6: Sprint 6

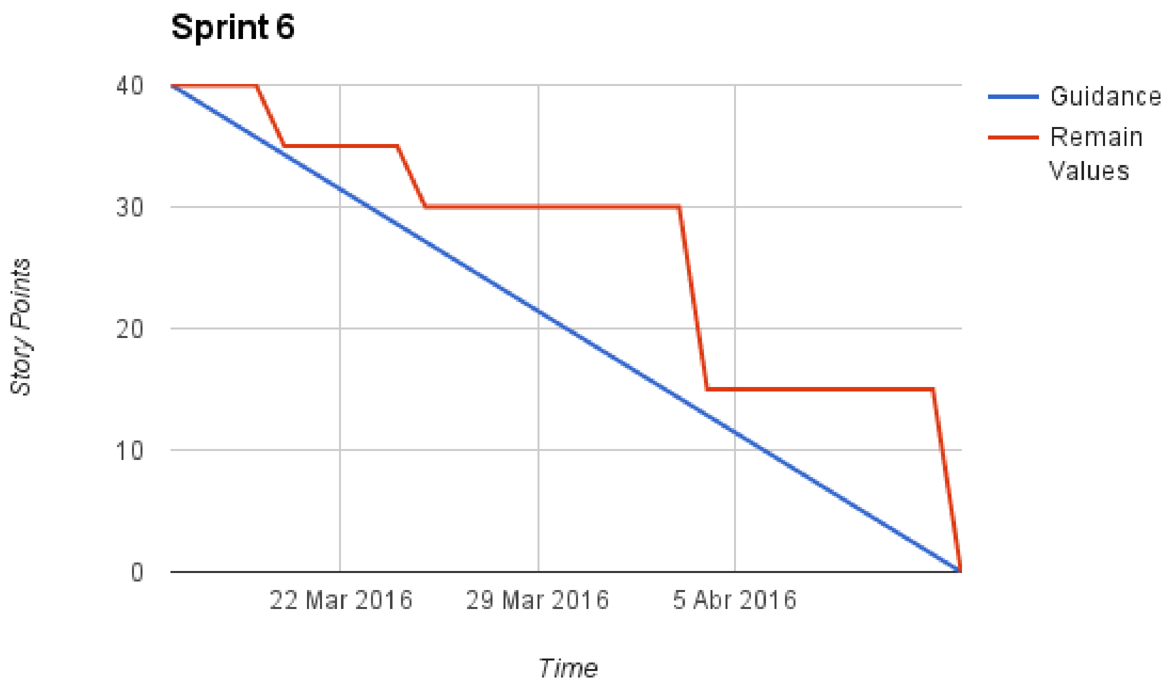


Figure 9.6: Sprint 6 Burndown

Sprint 7

Task	State	Story Points
Correction of bugs in back-end	Done	5
Change layout of front-end by client feedback	Done	5

Table 9.7: Sprint 7

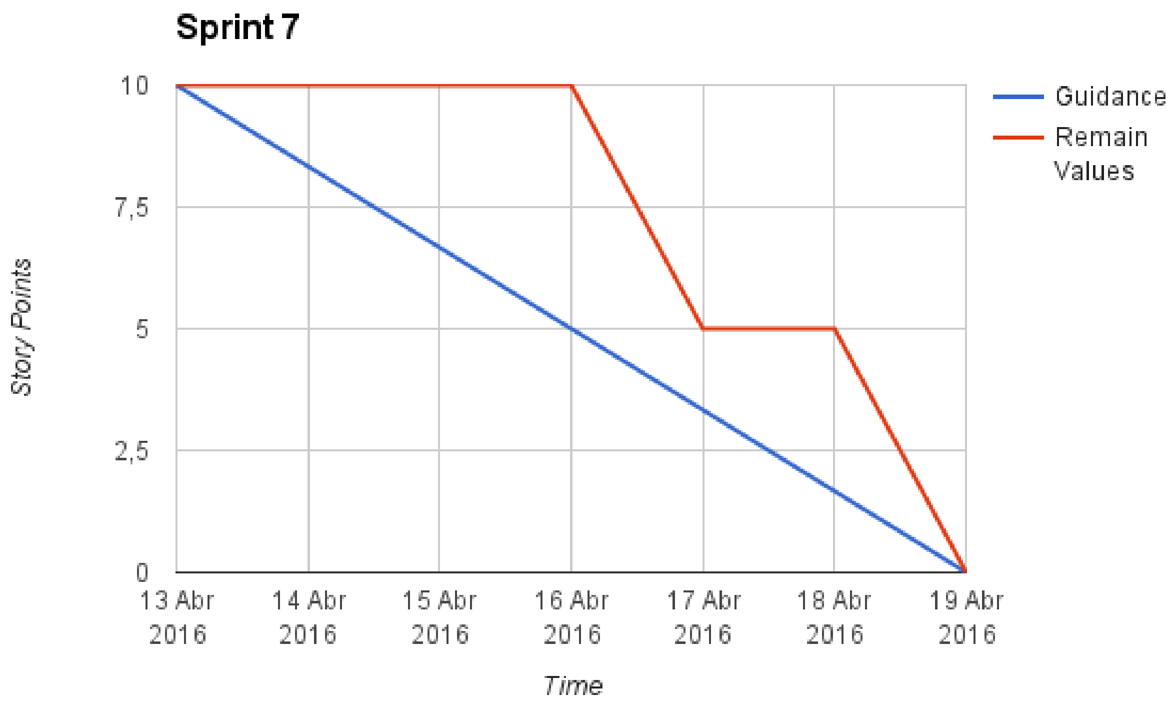


Figure 9.7: Sprint 7 Burndown

Sprint 8

Task	State	Story Points
Automation of injection and test mechanism	Done	20

Table 9.8: Sprint 8

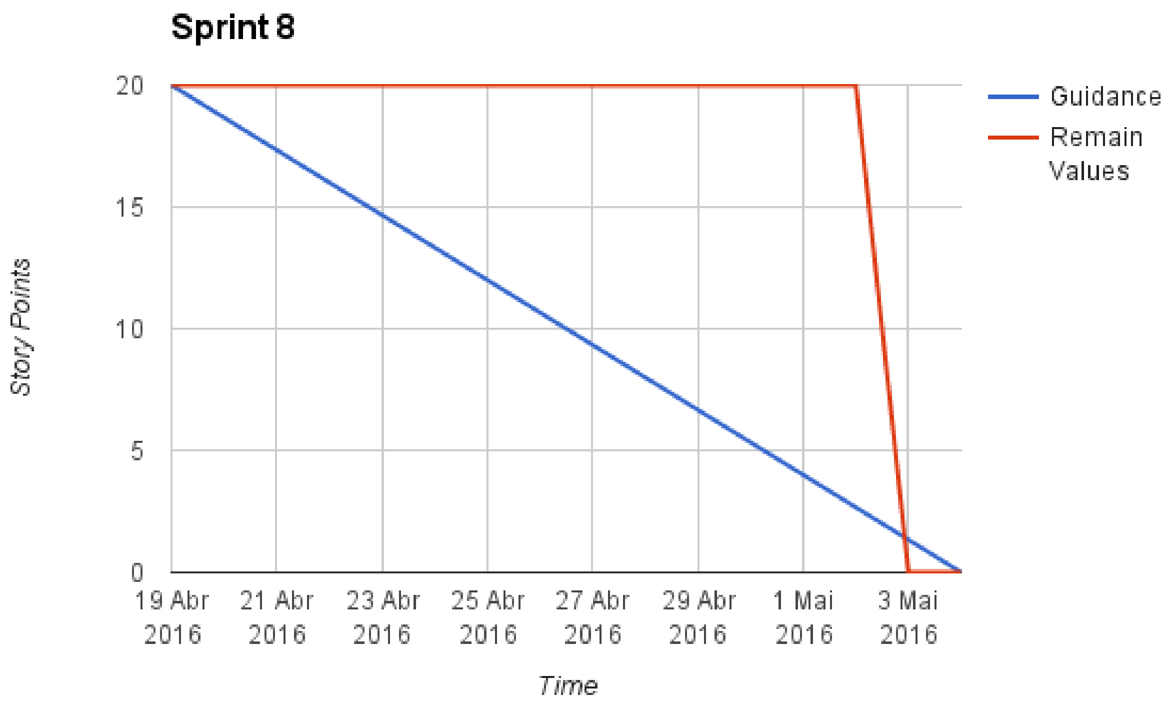


Figure 9.8: Sprint 8 Burndown

Sprint 9

Task	State	Story Points
Prepare and deploy the staging area	Not done	15
Correction of injector (driver dead)	Not done	15

Table 9.9: Sprint 9

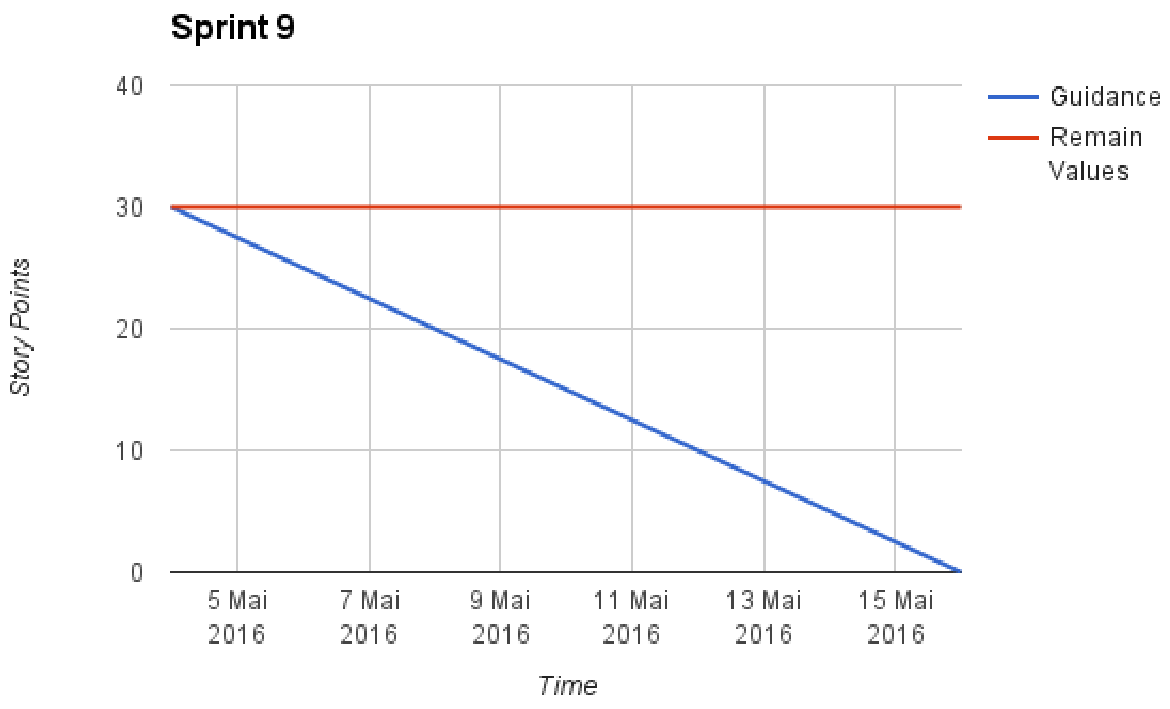


Figure 9.9: Sprint 9 Burndown

Sprint 10

Task	State	Story Points
Prepare and deploy the staging area	Done	5
Correction of injector (driver dead)	Done	5
Run tests to test staging area	Done	3

Table 9.10: Sprint 10

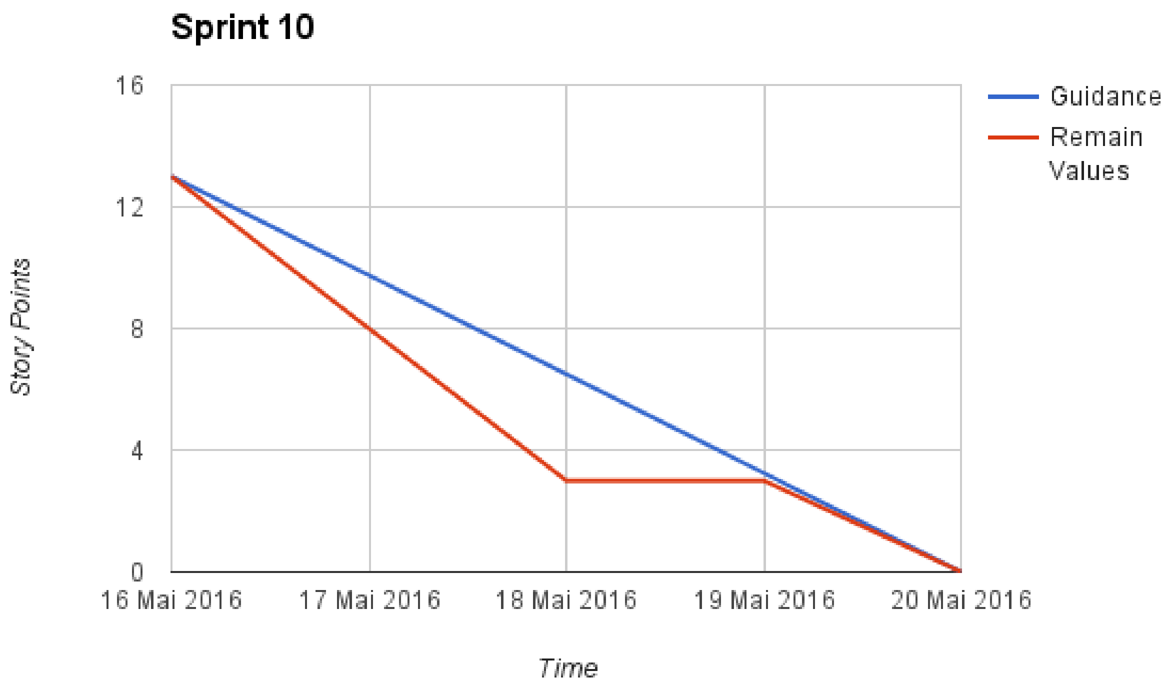


Figure 9.10: Sprint 10 Burndown

Sprint 11

Task	State	Story Points
Correction of hibernate and mysql erro	Done	5
Change mechanism of restart target	Done	5
Changes in front-end and back-end	Done	5

Table 9.11: Sprint 11

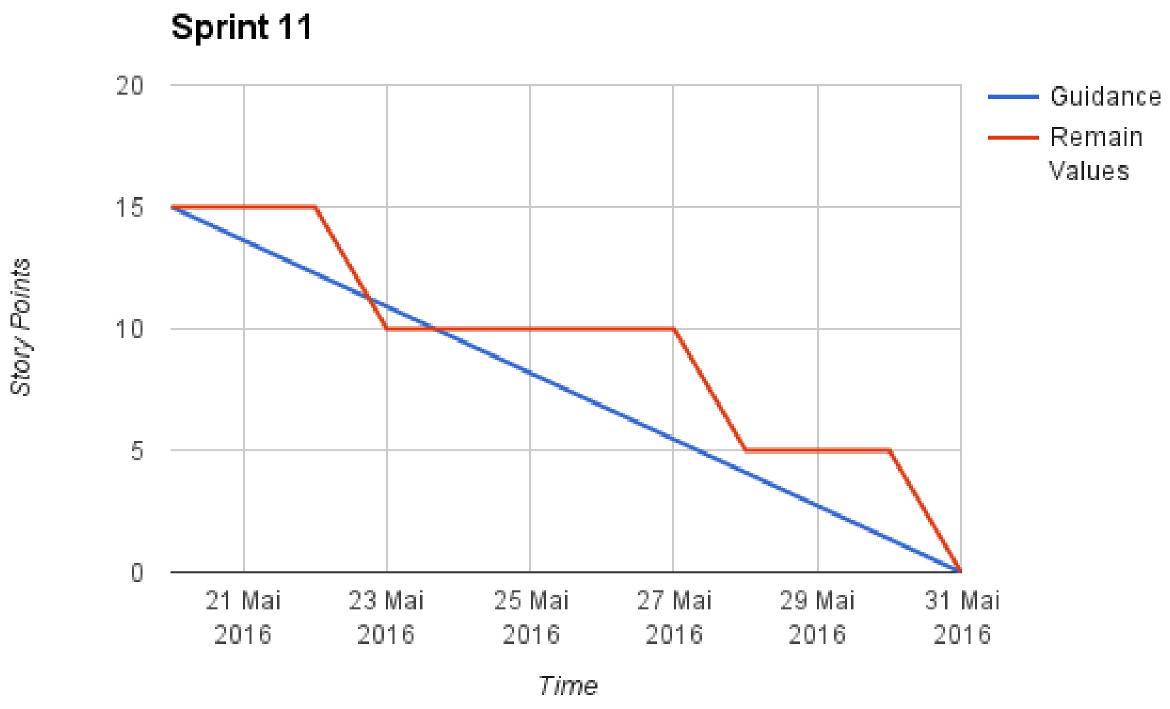


Figure 9.11: Sprint 11 Burndown

Sprint 12

Task	State	Story Points
Run several campaigns and analysis of the results	Done	10

Table 9.12: Sprint 12

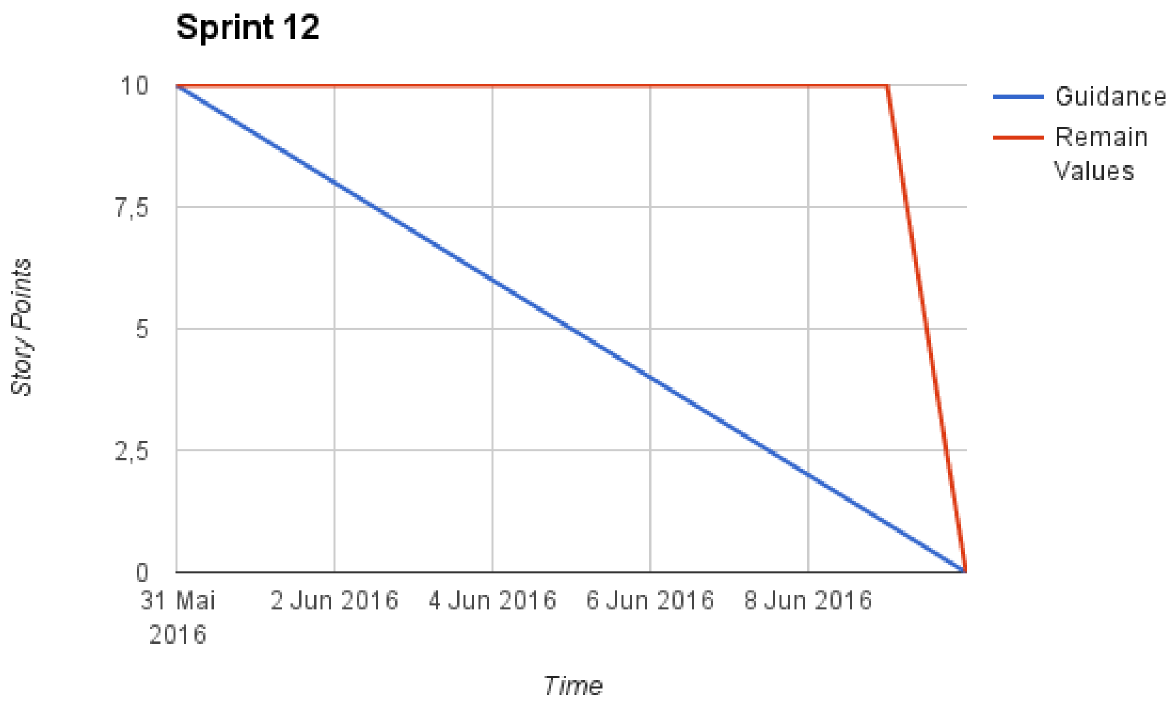


Figure 9.12: Sprint 12 Burndown

9.2 Annex B - Risk Analysis

Every project of Software Engineering there is a possibility of something bad happen. This unforeseen events might result in failure to meet the goals of the project. To prevent this there was make an analyze to detect in time this unforeseen events.
Following it will be identified the risks that could occur during the project, and if they occur or not, as well the impact, probability of happen and strategies to decrease the risk.

Risk 01	**Type** Resource	**Impact** Critical	**Probability** High
Description During the time of the internship the acquisition of a cloud server for realization of real stress tests could not happen, which will not allow the realization of stress tests in real deployment area.			
Decrease risk strategy Creation of a *staging* area and execute the stress tests in there, in order to perform some tests and assume that in cloud server the results will be better.			
Occurrence This event occur during the internship and it was used the decrease risk strategy as way to realize the stress tests and have a minimal perception of the system.			

Table 9.13: Risk 01

Risk 02	**Type** Project	**Impact** Critical	**Probability** High
DescriptionWith the abstract vision and a non stable requirements related to the project, it can take to a redefinition of the requirements for the project.			
Decrease risk strategy During the meetings of the project see if there is any changing related to the project, in order to adapt to the new requirements.			
Occurrence This event occur and was used the decrease risk strategy in order to minimize the impact of this risk.			

Table 9.14: Risk 02

Risk 03	**Type** Resource	**Impact** High	**Probability** High
Description During the internship do not be release a beta version, to be use by a real user for gathering real data and feedback about the product.			
Decrease risk strategy During several reunions with the product owner let him use the product, to obtain feedback from someone outside of the development team.			
Occurrence This event occur, but during the internship every time there was a reunion with the client we let him use the product getting feedback from someone outside of the team.			

Table 9.15: Risk 03

Risk 04	**Type** Project	**Impact** Critical	**Probability** High
Description During the internship the need to learn a new language, *AngularJS*, could take more time than expected, which will delay the development of the front-end.			
Decrease risk strategy Before starting the development of the front-end try to learn from online tutorials and use sample code in front-end.			
Occurrence This event occur during the internship and was used the decrease risk strategy during the development of the front-end to try not delay the project.			

Table 9.16: Risk 04

9.3 Annex C - Requirements Analyses

For the prioritization of requirements was used the key words, defined in RFC 2119, "Must", "Should", "Could" and "Optional".

9.3.1 D.1 - Functional Requirements

- **As a/an** <actor in the system >(Who?)
- **I want to** <action >(What?)
- **So that** <benefit >(Why?)

ID DC-01	**Priority** Must
As a/an web user	
I want to add a new target system	
So that could make an injection campaign to that system to test it	

Table 9.17: User story DC-01

ID DC-02	**Priority** Must
As a/an web user	
I want to test if the connection to the target system is ok	
So that proceed to test if the parameters given about the target system were the corrects	

Table 9.18: User story DC-02

ID DC-03	**Priority** Must
As a/an web user	
I want to set a injection campaign to a given date in the future	
So that the campaign will start to execute at the correct time to test the server	

Table 9.19: User story DC-03

ID DC-04	**Priority** Must
As a/an web user	
I want to launch a campaign in this moment	
So that the campaign start to execute right now to test the server	

Table 9.20: User story DC-04

ID DC-05	**Priority** Must
As a/an web user	
I want to see a short report how the campaign went	
So that I could analyze the resilience of my server during the campaign	

Table 9.21: User story DC-05

ID DC-06	**Priority** Must
As a/an web user	
I want to download a full report of a campaign	
So that to be able to analyze the report to see why and when the problems occur	

Table 9.22: User story DC-06

ID DC-07	**Priority** Must
As a/an web user	
I want to know the status of the campaign	
So that to know if the campaign is running, at each injection, if is delayed or concluded	

Table 9.23: User story DC-07

ID DC-08	**Priority** Optional
As a/an web user	
I want to receive an email with a notification when a campaign concluded	
So that be able to proceed with the analyses of the results	

Table 9.24: User story DC-08

ID DC-09	**Priority** Should
As a/an researcher	
I want to be able to upload my injector	
So that people can use it to validate my results or to test their server	

Table 9.25: User story DC-09

9.4 Annex D - Data Model

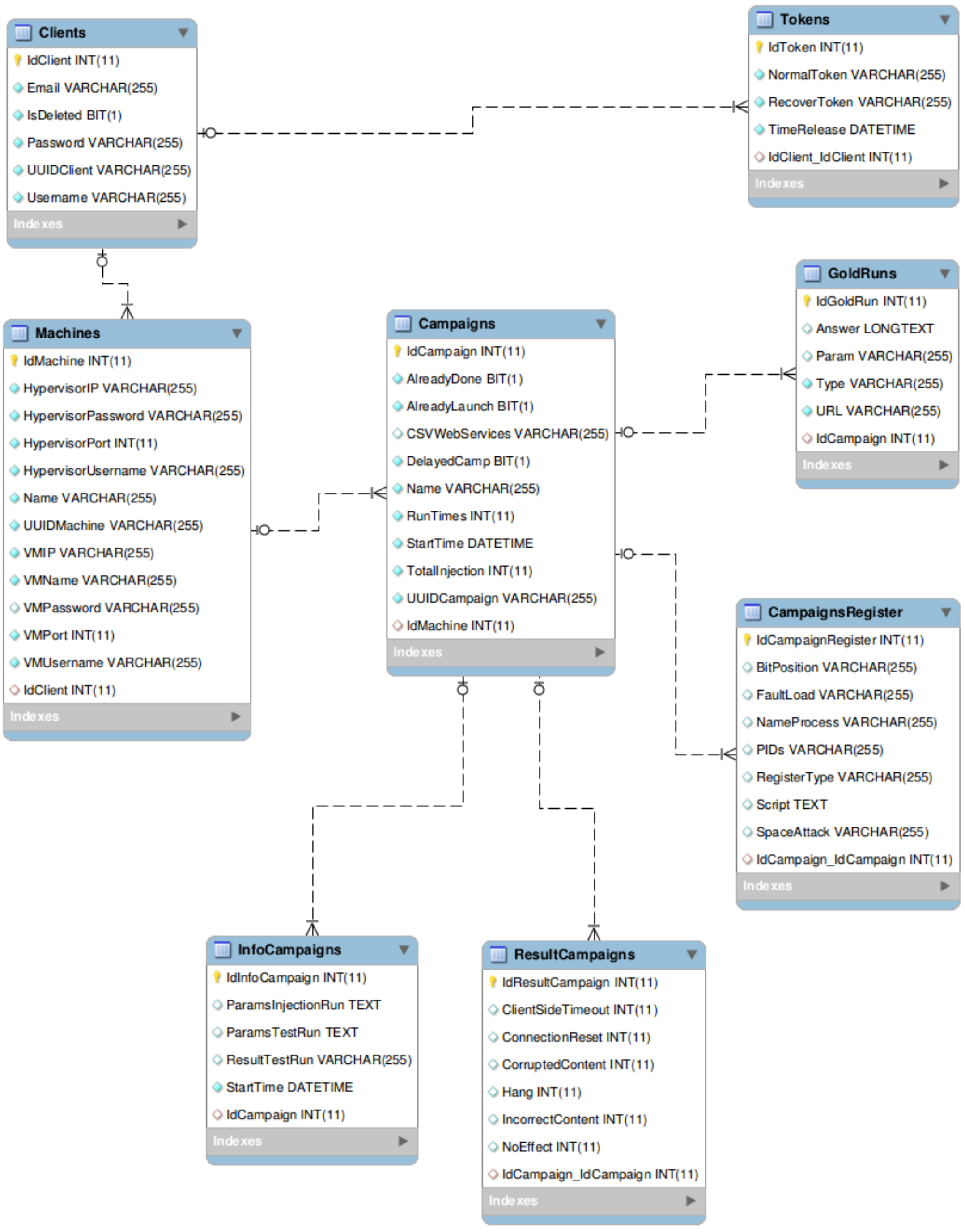


Figure 9.13: Data model diagram

Entity	Description
Clients	Represents a user of the application.
Tokens	Represents the control of accesses that a user have by the web application or by an external application that contacts the REST API.
Machines	Represents the target systems that a user have.
Campaigns	Represents the campaigns in general, with general data.
CampaignsRegister	Represents a specific campaign, in this case campaign to register of cpu, with the desired parameters.
GoldRuns	Represents the first run of each service in a workload to compare during tests.
InfoCampaigns	Represents detail information about the campaign, when everything happen and what happen.
ResultCampaigns	Represents a short report of the results of each campaign.

Table 9.26: Description data base entities

Following it will be describe the entities of the project.

Field	Data type	Index	Description
IdClient	Integer	Single Field	Identifier of the client in the system.
Email	String	No	Email of the client to be notified.
IsDeleted	Boolean	No	Identify if the user is deleted or not.
Password	String	No	Password of access to login.
UUIDClient	String	Single Field	Identifier to be used outside of the system.
Username	String	No	Username of access to login.

Table 9.27: Description table Clients

Field	**Data type**	**Index**	**Description**
IdToken	Integer	Single Field	Identifier of the token in the system.
NormalToken	String	Single Field	String to validate the access of a user in the system.
RecoverToken	String	Single Field	String to re validate the normal token.
TimeRelease	Date	No	Date when the token was created or re validate.
IdClient_IdClient	Integer	Single Field	Id of the client that have this token.

Table 9.28: Description table Tokens

Field	**Data type**	**Index**	**Description**
IdMachine	Integer	Single Field	Identifier of the machine in the system.
HypervisorIP	String	No	Ip to the hypervisor of target system.
HypervisorPassword	String	No	Password to access the hypervisor using ssh.
HypervisorPort	Integer	No	Port of access to use ssh.
HypervisorUsername	String	No	Username to access the hypervisor using ssh.
Name	String	No	Name to be showed when list the target systems.
UUIDMachine	String	Single Field	Identifier to be used outside of the system.
VMIP	String	No	Ip to target system.
VMName	String	No	Name of the target system in the virtual machine manager.
VMPassword	String	No	Password to access the target system using ssh.
VMPort	Integer	No	Port of access to use ssh.
VMUsername	String	No	Username to access the target system using ssh.
IdClient	Integer	Single Field	Id of the client that have this machine.

Table 9.29: Description table Machines

Field	**Data type**	**Index**	**Description**
IdCampaign	Integer	Single Field	Identifier of the campaign in the system.
AlreadyDone	Boolean	No	Flag that identify if campaign is concluded.
AlreadyLaunch	Boolean	No	Flag that identify if campaign is running.
CSVWebServices	String	No	Name of the file with the services to realize tests.
DealyedCamp	Boolean	No	Flag that identify if campaign is delayed.
Name	String	No	Name to be show during the list of the campaigns.
RunTimes	Integer	No	Number of injections the campaign will have.
StartTime	Date	Single Field	Date with the time the campaign will start.
TotalInjection	Integer	No	Total of injections already made.
UUIDCampaign	String	Single Field	Identifier to be used outside of the system.
IdMachine	Integer	Single Field	Id of machine that is associated with this campaign.

Table 9.30: Description table Campaigns

Field	Data type	Index	Description
IdGoldRun	Integer	Single Field	Identifier of the gold run in the system.
Answer	LongText	No	Answer received by the target system.
Params	String	No	Parameters to be used in the request to target system.
URL	String	No	URL to access the service of the target system.
IdCampaign	Integer	Single Field	Id of the campaign that is associated with this gold run.

Table 9.31: Description table GoldRuns

Field	Data type	Index	Description
IdCampaignRegister	Integer	Single Field	Identifier of the campaign register in the system.
BitPosition	String	No	Parameter to be used in the injection related to bit position.
FaultLoad	String	No	Name of the file with fault workload to be replicated.
NameProcess	String	No	Name of the process to be attack during injection.
PIDs	String	No	PIDs of process to attack.
RegisterType	String	No	Parameter to be used in the injection related to register type.
Script	String	No	Script to run to get the processes to inject faults.
IdCampaign_IdCampaign	Integer	Single Field	Id of the campaign that is associated with this campaign register.

Table 9.32: Description table CamapignsRegister

Field	**Data type**	**Index**	**Description**
IdResultCampaign	Integer	Single Field	Identifier of the result campaign in the system.
ClientSideTimeout	Integer	No	Number of times that occur a client side timeout event during campaign.
ConnectionReset	Integer	No	Number of times that occur a connection reset event during campaign.
CorruptedContent	Integer	No	Number of times that occur a corrupted content event during campaign.
Hang	Integer	No	Number of times that occur a hang event during campaign.
IncorrectContent	Integer	No	Number of times that occur a incorrect content event during campaign.
No effect	Integer	No	Number of times that occur a no effect event during campaign.
IdCampaign_IdCampaign	Integer	Single Field	Id of the campaign that is associated with this campaign register.

Table 9.33: Description table ResultsCampaigns

Field	**Data type**	**Index**	**Description**
IdInfoCampaign	Integer	Single Field	Identifier of the info campaign in the system.
ParamsInjectionRun	String	No	Parameters used during a injection.
ParamsTestRun	String	No	Parameters used during a test.
StartTime	Date	No	Time the test or injection started.
IdCampaign	Interget	Single Field	Id of the campaign that is associated with this info campaign.

Table 9.34: Description table InfoCampaigns

9.5 Annex E - Tests and Validation

9.5.1 Validation Campaigns

	Total Injections 10000		
	Total Stimulus 51177	**Total Stimulus** 48310	**Total Stimulus** 38724
No Effect	46143 (90.16%)	44216 (91.53%)	35370 (91.34%)
Incorrect Content	4549 (8.89%)	3587 (7.42%)	2884 (7.45%)
Corrupted Content	56 (0.11%)	41 (0.08%)	38 (0.10%)
Connection Reset	7 (0.01%)	12 (0.02%)	6 (0.02%)
Client Side Timeout	376 (0.73%)	399 (0.83%)	383 (0.99%)
Hang	46 (0.09%)	55 (0.11%)	43 (0.11%)

Table 9.35: Evaluation campaigns to Apache

	Total Injections 1000		
	Total Stimulus 23354	**Total Stimulus** 23855	**Total Stimulus** 23855
No Effect	23353 (100.00%)	23850 (99.98%)	23850 (99.98%)
Incorrect Content	0 (0.00%)	0 (0.00%)	0 (0.00%)
Corrupted Content	0 (0.00%)	0 (0.00%)	0 (0.00%)
Connection Reset	0 (0.00%)	0 (0.00%)	0 (0.00%)
Client Side Timeout	0 (0.00%)	5 (0.02%)	5 (0.02%)
Hang	1 (0.00%)	0 (0.00%)	0 (0.00%)

Table 9.36: Evaluation campaigns to Kernel

9.5.2 Acceptance Tests of Features

Requirement	**Description of test**	**Validation**
DC-01	Add new target system	✓
DC-02	Test if target system is ok	✓
DC-03	Set a campaign to start in a future	✓
DC-04	Lunch a campaign to start now	✓
DC-05	See a short report how the campaign results	✓
DC-06	Download a detailed report of campaign	✓
DC-07	See the status of campaign	✓
DC-08	Receive an email informing the end of campaign	✗[1]
DC-09	Be able to upload an injector	✗[2]

Table 9.37: Acceptation tests of features

[1]Requirement of low priority during the internship.

[2]Requirement not implemented due to change in planning.

Bibliography

[1] Frederico Cerveira, Raul Barbosa and Henrique Madeira *Benchmarking de Infraestruturas de Virtualização para a Cloud*, 6 of July 2015

[2] Elastic Compute Cloud (EC2) Cloud Server & Hosting – AWS. 2016. Elastic Compute Cloud (EC2) Cloud Server & Hosting – AWS. [ONLINE] Available at: `https://aws.amazon.com/ec2/`. [Accessed 20 January 2016].

[3] Microsoft Azure: Plataforma de Informática em Nuvem e Serviços. 2016. Microsoft Azure: Plataforma de Informática em Nuvem e Serviços. [ONLINE] Available at: `https://azure.microsoft.com/pt-pt/`. [Accessed 20 January 2016].

[4] Frederico Cerveira, Raul Barbosa, Henrique Madeira and Filipe Araujo, *Recovery for Virtualized Environments*, in 11th European Dependable Computing Conference, 2015.

[5] Chaos Monkey · Netflix/SimianArmy Wiki · GitHub. 2015. Chaos Monkey · Netflix/SimianArmy Wiki · GitHub. [ONLINE] Available at: `https://github.com/Netflix/SimianArmy/wiki/Chaos-Monkey`. [Accessed 14 November 2015].

[6] Business Intelligence - Overview — Oracle. 2016. Business Intelligence - Overview — Oracle. [ONLINE] Available at: `https://www.oracle.com/solutions/business-analytics/business-intelligence/index.html`. [Accessed 20 January 2016].

[7] Pentaho — Data Integration, Business Analytics and Big Data Leaders. 2016. Pentaho — Data Integration, Business Analytics and Big Data Leaders. [ONLINE] Available at: `http://www.pentaho.com/`. [Accessed 20 January 2016].

[8] Petter Mell and Timothy Grance, *The Nist Definition of Cloud Computing*, in NIST Special Publication 800-145, September 2011

[9] Marco Vieira, Henrique Madeira, Kai Sachs and Samuel Kounev, *Resilience Benchmarking* in *Resilience Assessment and Evaluation of Computing Systems*, 2012, pp. 284-301

[10] Google Apps for Work – Email, ferramentas de colaboração e muito mais . 2016. Google Apps for Work – Email, ferramentas de colaboração e muito mais . [ONLINE] Available at: `https://apps.google.com/intx/pt-PT/`. [Accessed 20 January 2016].

[11] Dropbox. 2016. Dropbox. [ONLINE] Available at: `https://www.dropbox.com/`. [Accessed 20 January 2016].

[12] Microsoft Office — Ferramentas de Produtividade para Casa e Escritório. 2016. Microsoft Office — Ferramentas de Produtividade para Casa e Escritório. [ONLINE] Available at: `https://products.office.com/pt-pt/home`. [Accessed 20 January 2016].

[13] Heroku — Cloud Application Platform. 2016. Heroku — Cloud Application Platform. [ONLINE] Available at: `https://www.heroku.com/`. [Accessed 20 January 2016].

[14] OpenShift by Red Hat. 2016. OpenShift by Red Hat. [ONLINE] Available at: `https://www.openshift.com/`. [Accessed 20 January 2016].

[15] Compute Engine - IaaS — Google Cloud Platform. 2016. Compute Engine - IaaS — Google Cloud Platform. [ONLINE] Available at: `https://cloud.google.com/compute/`. [Accessed 20 January 2016].

[16] Rackspace: Managed Dedicated & Cloud Computing Services. 2016. Rackspace: Managed Dedicated & Cloud Computing Services. [ONLINE] Available at: urlhttps://www.rackspace.com/pt. [Accessed 20 January 2016].

[17] Integration of Cloud Services like Office 365 in the enterprise network - FirstAttribute AG . 2016. Integration of Cloud Services like Office 365 in the enterprise network - FirstAttribute AG . [ONLINE] Available at: `http://www.firstattribute.com/en/news/integration-of-cloud-services-like-office-365/`. [Accessed 20 January 2016].

[18] Jim Gray, *Benchmark handbook: for database and transaction processing systems* Morgan Kaufmann Publishers Inc., 1992.

[19] Karl Huppler, The art of building a good benchmark, in First TPC Technology Conference (TPCTC 2009), Lecture Notes in Computer Science, 2009

[20] TPC-Homepage V5. 2016. TPC-Homepage V5. [ONLINE] Available at: http://www.tpc.org/. [Accessed 20 January 2016].

[21] SPEC - Standard Performance Evaluation Corporation. 2016. SPEC - Standard Performance Evaluation Corporation. [ONLINE] Available at: https://www.spec.org/. [Accessed 20 January 2016].

[22] Raul Barbosa, Johan Karlsson, Henrique Madeira and Marco Vieira, *Fault Injection* in *Resilience Assessment and Evaluation of Computing Systems*, 2012, pp. 263-281

[23] Algirdas Avizienis et al. "Basic concepts and taxonomy of dependable and secure computing". In: Dependable and Secure Computing, IEEE Transactions on 1.1 (2004), pp. 11–33.

[24] RESTful Web Services Introduction. 2016. RESTful Web Services Introduction. [ONLINE] Available at: http://www.tutorialspoint.com/restful/restful_introduction.htm [Accessed 14 January 2016].

[25] Java & JVM Conquer the World — zeroturnaround.com . 2016. Java & JVM Conquer the World — zeroturnaround.com . [ONLINE] Available at: http://zeroturnaround.com/rebellabs/java-jvm-conquer-the-world/. [Accessed 14 January 2016].

[26] Why i choose Hibernate for my project?. 2016. Why i choose Hibernate for my project?. [ONLINE] Available at: http://www.mkyong.com/hibernate/why-i-choose-hibernate-for-my-project/. [Accessed 14 January 2016].

[27] The Great Java Application Server Debate with Tomcat, JBoss, GlassFish, Jetty and Liberty Profile — zeroturnaround.com . 2016. The Great Java Application Server Debate with Tomcat, JBoss, GlassFish, Jetty and Liberty Profile — zeroturnaround.com . [ONLINE] Available at: http://zeroturnaround.com/rebellabs/the-great-java-application-server-debate-with-tomcat-jboss-glassfish-jetty-and-liberty-profile/. [Accessed 15 January 2016].

[28] Journey Through The JavaScript MVC Jungle – Smashing Magazine. 2016. Journey Through The JavaScript MVC Jungle – Smashing Magazine. [ONLINE] Available at: https://www.smashingmagazine.com/2012/07/journey-through-the-javascript-mvc-jungle/. [Accessed 15 January 2016].

[29] 10 Reasons Why You Should Use AngularJS. 2016. 10 Reasons Why You Should Use AngularJS. [ONLINE] Available at: `http://www.sitepoint.com/10-reasons-use-angularjs/`. [Accessed 15 January 2016].

[30] Wikipedia. 2016. Computer performance - Wikipedia, the free encyclopedia. [ONLINE] Available at: `https://en.wikipedia.org/wiki/Computer_performance`. [Accessed 25 May 2016].

[31] Wikipedia. 2016. Dependability - Wikipedia, the free encyclopedia. [ONLINE] Available at: `https://en.wikipedia.org/wiki/Dependability`. [Accessed 25 May 2016].

[32] Wikipedia. 2016. Computer security - Wikipedia, the free encyclopedia. [ONLINE] Available at: `https://en.wikipedia.org/wiki/Computer_security`. [Accessed 25 May 2016].

[33] Wikipedia. 2016. Resilience (network) - Wikipedia, the free encyclopedia. [ONLINE] Available at: `https://en.wikipedia.org/wiki/Resilience_(network)`. [Accessed 25 May 2016].

[34] Wikipedia. 2016. High availability - Wikipedia, the free encyclopedia. [ONLINE] Available at: `https://en.wikipedia.org/wiki/High_availability`. [Accessed 25 May 2016].

[35] Roger S. Pressman. Software Engineering - A Practitioner's Approach

[36] Wikipedia. 2016. Scrum (software development) - Wikipedia, the free encyclopedia. [ONLINE] Available at: `https://en.wikipedia.org/wiki/Scrum_(software_development)`. [Accessed 26 May 2016].

[37] Cohn, M. 2016. Planning poker: An agile estimating and planning technique. [ONLINE] Available at: `http://www.mountaingoatsoftware.com/agile/planning-poker` [Accessed: 27 May 2016].

[38] Git - Basic Branching and Merging. 2016. Git - Basic Branching and Merging. [ONLINE] Available at: `https://git-scm.com/book/en/v2/Git-Branching-Basic-Branching-and-Merging`. [Accessed 28 May 2016].

[39] Atlassian. 2016. Git Workflows and Tutorials — Atlassian. [ONLINE] Available at: `https://www.atlassian.com/pt/git/workflows#!workflow-gitflow`. [Accessed 28 May 2016].

[40] SSD Cloud Hosting & Linux Servers - Linode. 2016. SSD Cloud Hosting & Linux Servers - Linode. [ONLINE] Available at: `https://www.linode.com/`. [Accessed 28 May 2016].

[41] Amazon Web Services, Inc.. 2016. Amazon Simple Storage Service (S3) - Cloud Storage. [ONLINE] Available at: `https://aws.amazon.com/s3/`. [Accessed 28 May 2016].

[42] Mike Cohn. 2016. User Stories and User Story Examples by Mike Cohn. [ONLINE] Available at: `http://www.mountaingoatsoftware.com/agile/user-stories`. [Accessed 03 June 2016].

[43] Mike Cohn. 2016. User Story Template Advantages. [ONLINE] Available at: `http://www.mountaingoatsoftware.com/blog/advantages-of-the-as-a-user-i-want-user-story-template`. [Accessed 03 June 2016].

[44] Martin L. Abbott and Michael T. Fisher. Scalability Rules - 50 Principles for Scaling Web Sites.