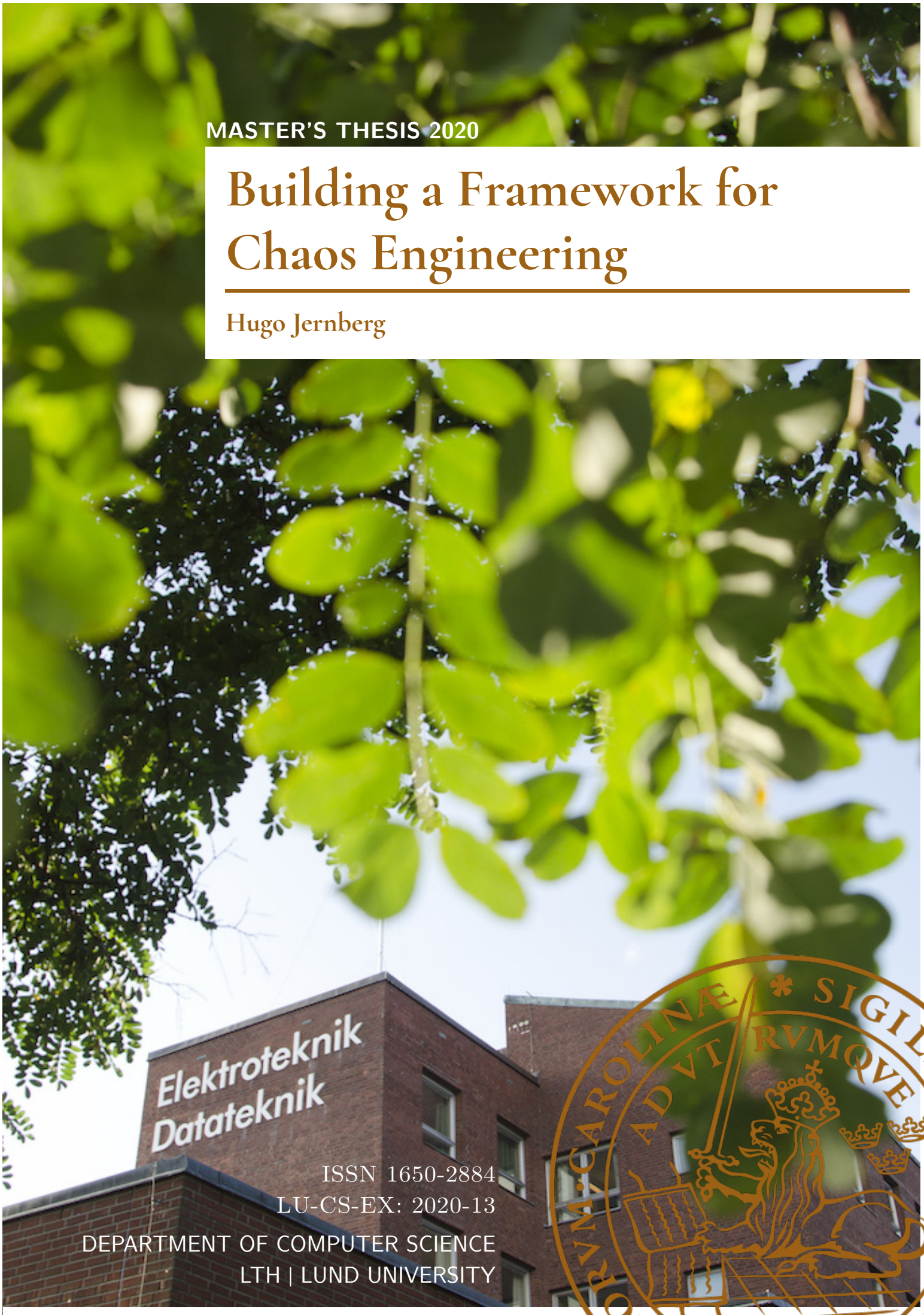


MASTER'S THESIS 2020

Building a Framework for Chaos Engineering

Hugo Jernberg

ISSN 1650-2884
LU-CS-EX: 2020-13

DEPARTMENT OF COMPUTER SCIENCE
LTH | LUND UNIVERSITY

EXAMENSARBETE
Datavetenskap

LU-CS-EX: 2020-13

Building a Framework for Chaos Engineering

Hugo Jernberg

Building a Framework for Chaos Engineering

Hugo Jernberg
ine15hje@student.lu.se

April 9, 2020

Master's thesis work carried out at ICA Gruppen AB.

Supervisors: Michael Adis, michael.adis@ica.se
Per Runeson, per.runeson@cs.lth.se

Examiner: Emelie Engström, emelie.engstrom@cs.lth.se

Abstract

Verifying a software system's ability to withstand turbulent conditions in its production environment can be a complex task, but it is nonetheless of uttermost importance to modern companies, who can suffer from huge losses if their systems are not resilient enough to endure such inevitable turbulence. This thesis proposes a framework for how to work with Chaos Engineering, a discipline aiming to evaluate and subsequently improve the resilience of a software system by using an experimental approach where the system under test is intentionally exposed to various types of problematic conditions to investigate how they are handled by the system. The proposed framework consists of four activities building on eight support documents, which collectively, with the help of a total of twelve open source Chaos Engineering tools, aim to define and implement a continuous and extensible Chaos Engineering practice for a software system. The framework was designed to suit the applications developed at ICA Gruppen AB and it was evaluated by applying parts of it to the software system which constitutes the website `ica.se`, including the e-shop which is accessible from the website. The applied parts were concluded to be feasible and they successfully discovered a set of initial improvement opportunities for the system's resilience, as well as a suitable Chaos Engineering practice for future resilience testing of the system.

Keywords: Chaos Engineering, software resilience testing, experimentation, design science research

Acknowledgements

I would like to thank my supervisors Per Runeson, LTH, and Michael Adis, ICA Gruppen AB, for providing me with invaluable feedback and continuous support throughout the project. I would furthermore like to thank my examiner, Emelie Engström, for giving me useful comments and suggestions. Finally, I also have to say thank you to every single person I have come in contact with at ICA during the twenty weeks of my project — thank you for being welcoming, generous with your time, and, last but not least, interested in what I have had to say.

Contents

Chapter 1
Introduction

Since the dawn of the modern computer era, revolutionary advances in software engineering have made the modern software development both flexible and fast. It is now possible for companies to deliver large-scale, complex software systems composed of thousands of computers working together in order to perform advanced tasks accurately and quickly [1]. However, with these progressions in mind, new matters of importance arise, for instance how the resilience of these complex and distributed systems can be verified. Software engineers should ask themselves how confident they can be that their complex software systems will work as intended, or, put differently, how extensive their *trust* in their systems can be [2].

This report constitutes the final product of a master's thesis project carried out at ICA Gruppen AB (ICA), a group of companies whose core business is grocery retail. Apart from grocery retail, the group also includes segments for real estate, banking and pharmacy operations. The purpose of the project was to examine how the resilience of the software systems developed at ICA could be improved by using tools and techniques for Chaos Engineering, a discipline based on the idea of experimenting on a software system to expose it to turbulent conditions on purpose [2]. The software system under test can be exposed to various types of turbulent conditions when Chaos Engineering is implemented, and examples include terminating a process which is critical for the system or introducing delays in its network communications. By implementing the principles of Chaos Engineering at ICA, the idea was to work towards a long-term goal of completely eliminating critical incidents in ICA's applications while at the same time maintaining a fast software development. The project aimed to gather how these principles for Chaos Engineering are best implemented at ICA in what will hereinafter be referred to as a framework. This framework was intended to be of support and guidance to development teams at ICA, enabling them to benefit from Chaos Engineering without previous expertise in the discipline. In this way, they themselves can still test, and thus own, all parts of the applications they develop.

This chapter starts by presenting ICA as an enterprise in section 1.1, with a focus on its software development. The research questions this thesis aims to answer are stated in section 1.2, and an outline of the remaining chapters of this report can be found in section 1.3.

1.1 ICA Gruppen AB

The vision of ICA is to make every day a little easier [3]. The company values simplicity, entrepreneurship and commitment, and over the last years ICA as a company has put an increasing focus on digitalization. The operations of the group are divided into five segments:

- **ICA Sweden**, a grocery retailer in Sweden. ICA Sweden has a market share of approximately 36 % of the Swedish grocery market, making it the country's leading grocery retailer [3]. ICA Sweden is also by far the largest segment of the ICA Group, contributing with more than 70 % of the group's total revenue.

- **Rimi Baltic**, a grocery retailer in the Baltic countries (Estonia, Latvia and Lithuania). Almost half of the Rimi Baltic stores are located in Latvia. Of the remaining stores, approximately 60 % are located in Estonia and 40 % in Lithuania [3].

- **Apoteket Hjärtat**, Sweden's largest chain of private pharmacies.

- **ICA Real Estate**, a commercial real estate firm operating in retail.

- **ICA Bank**, a business offering services in private and business banking.

The business model of ICA Sweden is to combine the benefits of private store ownership with the benefits of central scaling opportunities. Fundamentally, ICA Sweden owns the brand and the rights to the locations of the stores, while the stores themselves are owned and operated by local retailers. ICA Sweden provides the technical platforms that are necessary for the individual retailers, who are then free to put their efforts into adopting their stores according to the local markets, for instance by tailoring offers and assortments. Around 1,300 ICA Sweden stores are currently open, along with an additional 270 Rimi Baltic stores and 390 Apoteket Hjärtat pharmacies.

ICA Sweden's main website, and one of the software applications that are developed at ICA, is `ica.se`. There, visitors can for instance find information, search for stores and browse through current special offers. The most popular feature on the website, however, is to search amongst and browse through the available set of recipes. There are more than 22,000 recipes available at `ica.se`, ranging from appetizers and snacks to full meals and desserts. There is usually an especially intensive spike in traffic at this site just before Swedish holidays; Christmas is the number one reason for increased traffic, followed by Easter and Swedish Midsummer. The Christmas period is intense for a considerable part of December and does not end until after New Year's Eve. The other holidays are intense for shorter periods of time, Midsummer in particular.

At `handla.ica.se`, which is accessible from the top bar on `ica.se`, it is possible to shop online from some of the ICA Sweden stores. When visiting the website, users are asked to enter a postal code or select a city, which generates lists of stores which can deliver orders to the customers' homes or make them available for pickup at the physical stores. As of 2019, around 300 stores have connected to the e-commerce service. When a store has been selected, the e-shop is updated with that store's assortment. Users can navigate through categories, such as "Meat, poultry and fish" and "Fruit and vegetables", and add specific groceries to their shopping baskets, much like at a traditional store. The shop also shows current special offers, as

well as groceries that are produced locally and delivered from the selected store's local suppliers. Another feature is the so-called Receptshoppen, which enables users to browse through the available recipes and add all, or some, ingredients needed for a specific recipe. Additionally, by creating an account and logging in, users can see their most commonly purchased groceries and a list of their previous orders. Overall, ICA aspires for their combined online applications to be the customers' obvious number one choice for anything that concerns food.

Downtime is costly for ICA, perhaps when it comes to the company's e-shop in particular, and the cost of downtime includes, but is not limited to, a possible immediate loss in revenue if the customer decides to visit a competitor's e-shop instead of ICA's. The e-shop previously had a monolithic architecture, but since this architecture is currently being divided into numerous microservices, the number of failure points for the e-shop is growing. Some of these failure points are more critical than others; for instance, if the service which is responsible for showing product information goes down, the entire business of the e-shop becomes unavailable since users cannot open the products' information pages to add them to their shopping baskets. However, if the service allowing users to search among the groceries goes down it inarguably causes an inconvenience for the users, but they would still be able to shop by selecting categories and scrolling. Resilience and overall reliability can thus be said to be more important in some parts of ICA's e-shop than in others.

1.2 Research Questions

In this report, a framework is defined as a structure of suggested activities, along with necessary document support, which can support or guide how Chaos Engineering can be implemented. What such a framework should contain was not defined when the project started and answering that question was part of the research goal of this thesis. The framework should, however, contain sufficient building blocks to be useful even for applications which Chaos Engineering never has been used for previously.

The research goal of this thesis did not include finding new pieces of Chaos Engineering functionality or investigating how to improve the Chaos Engineering practices which are presented in related work. Instead, the contribution of the thesis can be seen as an investigation of how the existing functionalities and practices can be combined into a more long-term way to work with the Chaos Engineering discipline, answering questions such as which pieces of functionality to include for a specific application and how to improve the way the tools which provide these pieces of functionality are used.

The research questions this thesis aims to answer are:

RQ1: How can Chaos Engineering improve the resilience of software systems at ICA?

RQ2: What are the necessary building blocks of a Chaos Engineering framework, and how can they be implemented?

RQ3: Where is Chaos Engineering suitable to use at ICA, and can it be provided as a centralized service?

1.3 Outline

The rest of the report is structured as follows. Chapter 2 presents some previous work on Chaos Engineering along with topics related to Chaos Engineering. Chapter 3 contains a description of the project's research approach. In Chapter 4, the findings from the tool evaluation which was performed as part of this project's solution design are presented, and the rest of the framework is described in Chapter 5. Chapter 6 gives information on how the framework was evaluated before Chapter 7 discusses and concludes the results.

Chapter 2
Background

A common way to deliver a software service to its users today is over the Internet. Such a service is always implemented in the form of a *distributed system* [4], or, in other words, in the form of a collection of autonomous components that collaborate to appear to the user as a single system [1]. As the scale of a distributed system increases, so does the number of ways in which it can fail; not only can each component of a distributed system fail individually, but they can also fail when trying to interact with the other components of the system. This makes the task of ensuring that a distributed system works as intended a complex and challenging task [4]. At the same time, the importance of ensuring that the distributed system of a company works can be of uttermost importance to the business of that company. For instance, an estimate has been made that if `www.amazon.com` went down for a single minute, it would cost Amazon more than $220,000 [5]. Other organizations are not immune to similar downtime losses; the cost for an hour of downtime has been estimated to exceed $100,000 for 95 % of the companies with at least 1,000 employees [6]. This number is furthermore based on the organizations' average systems, not solely their most critical ones. Despite it being difficult to evaluate how well a distributed system behaves operationally, there exists a practice of conducting such an evaluation empirically. The name of that practice is Chaos Engineering.

Chaos Engineering can be defined as "the discipline of experimenting on a distributed system in order to build confidence in the system's capability to withstand turbulent conditions in production" [7]. In other words, it is an empirical approach for system verification. The overall goal with Chaos Engineering is to receive insight into how resilient the system under test is and thus learn about potential systemic weaknesses. By bringing such weaknesses to light before they cause problems which appear to the user as, for instance, outages or decreases in system performance, it is possible to prevent such problems from ever taking place.

This chapter will present relevant information on Chaos Engineering and related subjects, and it will present findings from the literature study which was performed as an introductory part of the project. The chapter starts with two general sections, in which a few

useful definitions in the area of software quality are given in section 2.1 and the discipline of fault injection testing is introduced in section 2.2. Then, findings from the Chaos Engineering literature study is presented in sections 2.3 through 2.7, where sections 2.3 through 2.5 first present Chaos Engineering in terms of its history, practical implementation and principles, respectively, before section 2.6 presents a model which can be used to map the extensiveness of a specific Chaos Engineering implementation and section 2.7 suggests some advantages and challenges with Chaos Engineering. Then, section 2.8 describes the to Chaos Engineering related topic of Continuous Experimentation. Finally, the chapter is concluded with sections 2.9 and 2.10 on application containerization and cloud service providers, topics of interest when examining Chaos Engineering tools.

2.1 Software Quality

The quality of software is not a trivial concept. Several efforts have been made over the past forty years to define software quality in terms of measurable quantities and one of the resulting software quality models is the ISO 9126, developed under the control of the International Organization for Standardization (ISO). ISO 9126 defines six categories of quality characteristics that are independent of each other, namely functionality, reliability, usability, efficiency, maintainability and portability [8]. These broad categories can be further decomposed into sets of subcharacteristics. There is no universal decomposition of the six categories, but ISO 9126 includes an example of one, where each category is decomposed into a number of subcharacteristics ranging from two to four.

Resilience, which Chaos Engineering aims to evaluate and subsequently improve in a software system, is also known as recoverability [9]. In turn, recoverability is one of the three subcharacteristics which reliability is decomposed into in the ISO 9126 sample quality model [8]. Below are some definitions of these software quality concepts:

- *Reliability* is a "set of attributes that bear on the capability of software to maintain its performance level under stated conditions for a stated period of time" [8]

- *Resilience*, or *recoverability*, is the "capability of the software to reestablish its level of performance and recover data directly affected in the case of a failure" [8]

A common and related term is *robustness*. This can be described as how sensitive a system is, or, in other words, how well it can handle erroneous input or conditions in the environment it operates in [8]. To distinguish between the terms, the following is a common understanding: robustness centers on how much the system can handle before it fails, whereas resilience centers on how well the system can recover after it has already failed. Reliability is a wider concept, containing everything which affects how capable the system is to maintain a failure-free operation. This is the understanding of the concepts which will be used in the rest of this report. Note that since Chaos Engineering mainly aims to improve a system's resilience, and resilience is part of a system's reliability, Chaos Engineering will by extension also aim to improve the system's reliability.

2.2 Fault Injection Testing

Chaos Engineering is a simulation-based fault injection technique [10]. Fault injection testing is commonly considered a white-box testing technique, meaning that the system's code is allowed as input when designing the tests. This is opposed to black-box testing techniques, in which the system is viewed as a black box, or, equivalently, a single functional unit, when testing it. To examine a system with black-box testing techniques, pairs of input-output are considered instead of the system's actual code. However, there exists a third testing strategy called grey-box testing, where both white-box and black-box testing strategies are combined. As opposed to white-box testing, where the internal structure of the system is known, and black-box testing, where the internal structure of the system is unknown, the internal structure is only partially known in grey-box testing. This means that test cases can be designed using internal data structures and algorithms, but the actual tests are performed at the black-box, or user, level [11]. In some pieces of literature, fault injection testing is considered a grey-box testing technique as opposed to a white-box one [10].

The idea of fault injection testing techniques is to insert faults or errors into the system under test on purpose. Note the difference between faults and errors: an *error* is commonly defined as a state of the system, whereas a *fault* often is considered to be the cause of an error [8]. A fault can notably remain unexposed in the system for an arbitrary amount of time, and not until some event activates the fault does it put the system in an error state. Note also that an error is not equivalent to a *failure*, which, unlike the previous terms, is an actual event. This event is what the user of the system can experience and occurs whenever the external behavior of the system differs from the system's intended behavior. To summarize, a fault can put the system in an erroneous state, and if the system is left in that state, a failure can occur. When conducting fault injection testing, the goal is to examine how the system reacts to being brought into these unwanted states [10]. Important questions to ask are: does the system fail, and if so, can it recover from the failure? In relation to the terms defined in section 2.1, the questions can be rephrased as: how robust is the system, and how resilient is it?

There are two different types of fault injection testing: compile-time fault injection testing and run-time fault injection testing [10]. *Compile-time fault injection testing* is based on introducing faults directly in syntax or semantics, that is, directly in the source code. *Run-time fault injection testing*, on the other hand, operates on a running system, and what it does is to trigger failure scenarios in the system by executing specific commands or in some other way introducing simulated real-world events representing one problem or another. That is to say, instead of altering the system's source code, run-time fault injection testing uses additional code to simulate errors.

Out of these two types, Chaos Engineering is a run-time fault injection testing technique. It experiments with a running distributed system and is not dependent on changing the code of the system under test itself. The basic explanation of Chaos Engineering is that it monitors the system's behavior in the form of some predetermined metrics and conducts experiments, in which faults are designed and executed in some part of the system, affecting a subset of its current users. In other words, the environment of Chaos Engineering experiments is the system's real production environment. The metrics that are monitored are supposed to check the system's overall functionality, often referred to as the system's *health*, and by monitoring them during the experiments, conclusions can be drawn about how well the system handles

the injected faults [10]. By learning about the system's behavior and becoming aware of potential weaknesses before they cause unexpected harm, the ultimate goal is to improve the system's resilience [7].

2.3 History of Chaos Engineering

The coming five sections on Chaos Engineering and its history, practical implementation, principles, maturity, and advantages and challenges, respectively, are findings from what is referred to as a literature study in subsection 3.1.1. Subsection 3.1.1 describes how this literature study was carried out in more detail, but generally it can be said that the five sections attempt to highlight some core principles of the discipline as they have been perceived when reading previous work on Chaos Engineering. These pieces of related work have a varying degree of maturity; community websites with numerous contributors are referenced along with published e-books and journal articles. How these pieces of related work were found is also described in subsection 3.1.1. A few references not directly related to Chaos Engineering have been included as well, solely to add some pieces of trivia to the text. The sections are not to be seen as an all-covering summary of Chaos Engineering literature, but rather as an introduction to the subject and how it can be perceived when reading about it in different types of publications.

Chaos Engineering originated at Netflix, a provider of a subscription-based streaming service. In August 2008, Netflix suffered from an outage lasting no less than three days, during which its services were unavailable to its users. This outage was caused by a database corruption in one of their major databases [12]. This was the second big outage Netflix suffered that year; in March 2008, Netflix went down for eleven hours due to a problem in its maintenance system [13]. Up until then, Netflix's services were running on in-house racked servers, meaning that the number of failure points was limited to one [12]. However, starting 2008, Netflix decided to move one of their datacenters into the cloud [7]. In 2011, Netflix's whole architecture had been changed; the aforementioned racked servers were migrated and the system became a distributed one, running on Amazon Web Services (AWS). The new architecture was based on hundreds of microservices, drastically increasing the number of failure points. On one hand, this eliminated the huge dependency Netflix's services had on their previous in-house racked servers, but, on the other hand, it introduced a whole new spectrum of possible failures. This new complexity in Netflix's system thus required it to become significantly more reliable and fault-tolerant [12].

Since that first datacenter move in 2008, Netflix has conducted resilience testing in production in one way or another. This led to the introduction of a tool named Chaos Monkey in 2010, which, to date, still delivers the same essential functionality as it did when it launched [7]. The purpose of Chaos Monkey is to randomly select and terminate a running instance in production. It operates during the day, which minimizes the potential impact on users in two ways. Firstly, Netflix streams peak later at night, and the number of users online is thus lower when Chaos Monkey is enabled than it otherwise could have been. Globally, the traffic reaches its peak around 9 pm on weekdays, and no country has an earlier peak time than India, where the streams are at their highest at 5 pm [14]. Secondly, Netflix software engineers are at work during the day, making them able to quickly resolve any issues the experimental testing might cause. The effect of the Chaos Monkey tool was prominent, since it

made the failure scenario of an instance disappearing close to irrelevant to the operation of Netflix's services. Put differently, it provided a strong incentive for Netflix's software engineers to build resilient systems, which, as a result, has become part of the engineering culture at Netflix.

Since Chaos Monkey, a variety of Chaos Engineering tools have emerged (see Chapter 4). Netflix themselves released a whole suit of testing tools called the Simian Army in 2011, including not only Chaos Monkey for terminating running instances but also, for instance, Latency Monkey for injecting communication delays in client-server links and Chaos Kong for terminating entire AWS regions [10].

Today, there are numerous large companies who practice Chaos Engineering. Some of them even have their own Chaos Engineering tools offered as a service on the market. Examples of companies who practice Chaos Engineering include LinkedIn, Facebook, Amazon, Microsoft and Google, to mention a few. One probable reason for the spread of Chaos Engineering since it originated more than a decade ago, is the variety of benefits it provides. Such benefits are not necessarily limited to technical ones, but may include customer benefits and business benefits as well [15]. See subsection 2.7.1 for a more thorough portrayal of what these benefits can consist of.

2.4 Chaos Engineering in Practice

As Chaos Engineering revolves around conducting experiments in order to reveal weaknesses in distributed systems, there is a need to describe the way in which such experiments can be carried out. The experiments can be referred to as *Chaos Experiments* and one description of them can be found on the community website `principlesofchaos.org`. This website has its HTML code in a GitHub repository, which in turn has contributors from for instance employees at Netflix and research engineers at the Royal Institute of Technology in Stockholm, Sweden. The website lists four steps which a Chaos Experiment can consist of [2]:

1. Defining a steady state which characterizes the normal behavior of the system under test

2. Hypothesizing around how this steady state will be affected when running the experiment

3. Running the experiment by simulating real-world events

4. Proving or disproving the hypothesis

One reason why Chaos Engineering revolves around empirical experiments can be that the behaviors of large-scale software systems tend to be too complex to build all-covering predictive models which represent their responses to all possible events [7]. It can in other words be difficult to test how a large-scale software system will react to any given event, such as a latency increase or a component failure, without testing it directly on the system itself, and an empirical approach can thus be favored. Similarly, it is not always feasible to write a complete functional specification of a distributed system [16]. The number of possible inputs for such a large-scale system, depending not only on an underlying infrastructure but also on unpredictable user behavior, can simply be too large.

2.4.1 Defining Steady State

If a Chaos Experiment is started by defining a steady state, this represents defining which systemic behaviors are acceptable and which are not. If this is not defined in the beginning of an experiment, it is probable that it is more difficult to determine the outcome of the experiment when it has been carried out. The *steady state* of the system refers to its normal behavior, that is, the acceptable behavior it exhibits when no intentional fault has been injected into it. In other words, a steady state definition is normally based on the system's measurable output rather than attributes of the system which are strictly internal [2].

According to Sathiya Shunmugasundaram, Site Reliability Engineer at Apple, both technical metrics and business metrics should be used when defining the steady state of a system [17]. *Technical metrics* can include metrics such as latency, number of errors thrown over a period and the average CPU usage, but they can vary greatly between different cases. They describe the health of the system, but they are not always directly connected to revenue or the key business objectives of the company. This means that technical metrics such as the ones just mentioned sometimes are unable to answer questions regarding for instance customer satisfaction. *Business metrics*, on the other hand, should better capture such aspects. Examples of business metrics include number of logins per minute, number of failed logins per minute, or, in the case of Netflix, number of stream starts per second [7].

To define the steady state of a system, there are thus two questions that need to be answered: what are the relevant technical and business metrics for the system under test, and what are the acceptable values of those metrics? Naturally, tools for capturing these values need to be in place as well.

2.4.2 Hypothesizing About Steady State

To further facilitate the process of determining the outcome of a Chaos Experiment, a formal hypothesis describing what the experiment should result in can be formulated. Otherwise, it can sometimes be difficult to draw conclusions from the experiment. Experiences at Netflix have resulted in a recommendation to phrase Chaos Experiment hypotheses on the following form: "The events we are injecting into the system will *not* cause the system's behavior to change from steady state" [7]. If the hypothesis is phrased in the opposite way, that is, that the injected events *will* cause the system's behavior to change from its steady state, the logic is that there is no purpose in even conducting the experiment. The purpose of running Chaos Experiments is to uncover unknown weaknesses in the system, not proving that known weaknesses definitely will cause the problems that Chaos Engineering intends to avoid. Known weaknesses should always be addressed first. When they have been fixed, the Chaos Experiment can proceed to either prove that the system now is more resilient or that one or more weaknesses still remain.

2.4.3 Simulating Real-World Events

When a steady state has been defined and the hypothesis has been phrased, the next step is to inject the fault into the system. This is also referred to as performing a Chaos Engineering attack. The fault is normally based on real-world events, meaning that it simulates a problem which might occur in the production environment naturally, without injecting it on purpose.

There are of course too many possible events which may happen to be able to list or simulate all of them, but examples naturally include the attacks provided by the Chaos Monkey, Latency Monkey and Chaos Kong tools, mentioned in section 2.3. Events can also be combined and simulated simultaneously to truly force the system into a harmful state [7]. The tools described in Chapter 4 can all be used to inject a variety of faults into the system under test, or, equivalently, perform a variety of Chaos Engineering attacks on the system under test.

2.4.4 Proving or Disproving the Hypothesis

By continuing to monitor the steady-state metrics during the experiment, they can afterwards be used to determine the outcome of the experiment. If the metrics remain in their acceptable ranges, the hypothesis can be proved. If they do not, the hypothesis can instead be disproved, and a weakness in the system has been found.

2.5 Principles of Chaos Engineering

Several pieces of previous work on Chaos Engineering mention best practices for how to implement Chaos Engineering, and some of them have been gathered in this section. Firstly, Chaos Experiments should according to the Principles of Chaos community website be carried out directly in the production environment, that is, directly on production traffic [2]. The point of this is to ensure the authenticity and relevance of the experiments. The alternative, to run Chaos Experiments in testing environments, can be inferior since a testing environment does not always capture all aspects of a complete system running in production [10]. For instance, there can often be differences between a test system and the production system when it comes to how real users of the system behave compared to synthetic users [16]. However, it can also be considered to be of uttermost importance to start small when implementing Chaos Engineering. According to Tammy Bütow at Gremlin, one way of doing this is incrementally, by starting in a demonstration environment, then moving on to a staging environment, and finally, when sufficient confidence in the discipline of Chaos Engineering has been obtained, moving to the real production environment [18].

Another principle that is mentioned in Chaos Engineering literature is to minimize the blast radius of Chaos Experiments. The book titled Chaos Engineering, written by Casey Rosenthal et al., sees that this includes several aspects, for instance running experiments on only a subset of the system's users to ensure that not too many users are affected if an experiment results in a failure [7]. Another aspect that is mentioned regarding minimizing the blast radius of an experiment is the ability to abort a running experiment when it is noticed that it causes a failure. The termination can be automatic and the need for it can be detected in the process of monitoring the metrics that define the system's steady state. Note that minimizing the blast radius is related to the *business* of the enterprise conducting the Chaos Experiment; minimizing the blast radius seeks to minimize the effects on the *business*, which is not necessarily equivalent to minimizing the effects on the *technology* under test [12]. Apart from running the Chaos Experiment on only a subset of users and being able to abort the experiment, another way of minimizing the blast radius can be to conduct the experiments at times when software engineers are most likely to be able to quickly mend any unintentional damage [4].

When conducting Chaos Experiments manually, they tend to require quite some resources in the form of time and can therefore become expensive. Therefore, another Chaos Experiment principle can be to automate Chaos Experiments to run continuously [2]. It is, for instance, possible to integrate the Chaos Experiments into the pipeline for continuous integration and continuous deployment, which enables the automation of the production testing as a post-deploy step [19].

Also, it has been said to be important that Chaos Engineers view the system as a single unit when conducting the experiments, regardless of it being distributed and consisting of a great variety of running services [16]. Since the steady-state metrics are defined based on the system's output and not its internal attributes, the system boundary is where the system is observed during the experiment. This can have the consequence of putting the focus during an experiment on whether the system works or not, not the internal workings of *how* it works [2].

When it comes to the real-world events which are simulated in the system, the number of possible events can grow too large to be able to consider all of them. Therefore, it can be necessary to prioritize between the events to decide which ones to simulate. This prioritization can be based on different aspects, for instance the possible impact an event could have on the system or some estimated frequency of how often it occurs naturally [2].

To summarize, these are some best practices for how to implement Chaos Engineering which have been mentioned in previous work on the discipline:

- Run experiments in production
- Start small and increase incrementally
- Minimize the blast radius of the experiments
- Automate experiments to run continuously
- Observe the system at its boundary
- Prioritize experiments based on impact or frequency

2.6 Chaos Maturity Model

The *Chaos Maturity Model* (ChMM) provides a way of mapping how extensively Chaos Engineering is implemented in an organization. The model consists of two metrics: sophistication and adoption. These two form a map where sophistication commonly is located on the y-axis and adoption on the x-axis. By drawing the map and locating where the Chaos Engineering practice of an organization lies, it is possible not only to decide whether or not an improvement of the Chaos Engineering practice is needed, but also in which direction, or along which axis, one's efforts for such an improvement best are focused [7].

Sophistication of Chaos Experiments refers to their validity and safety, meaning that unsophisticated Chaos Experiments are dangerous to the organization and thus potentially invalid. Four levels of sophistication can be distinguished, defined by their level of commitment in a number of areas: the environment of the experiment, the degree of automation, the metrics that describe the steady state of the system and the complexity of the events that are simulated. The four levels are:

1. **Elementary**. Experiments are not run in production and they are conducted manually. The metrics describing the steady state of the system are technical. Simple events are simulated.

2. **Simple**. Production-like traffic is used, for instance a replay of the real traffic. The experiments are executed automatically, but monitoring and potential terminations are manual. Aggregated business metrics are included. Events are expanded, including events such as network latency.

3. **Sophisticated**. Experiments are run in production. Monitoring and termination are automated. Business metrics are disaggregated and used together with technical metrics. Events are combined to create complex failures.

4. **Advanced**. Experiments are run in every step of development, in every environment. The design of the experiments is automated. Business metrics are disaggregated and used together with technical metrics. Events are expanded, including events such as changing user patterns.

Adoption of Chaos Experiments refers to how extensively they reach in an organization. Four levels of adoption can be distinguished, defined by their level of commitment in a number of areas: the sanctioning of the experiments, the number of systems that are covered by the experiments and the level of organizational awareness of Chaos Engineering. The four levels are:

1. **In the Shadows**. Small and unstructured groups perform unsanctioned experiments. Few systems are covered. There is no or low organizational awareness.

2. **Investment**. Experiments are sanctioned in the form of part-time resources. A few critical systems are covered. There is an interest for Chaos Engineering in multiple teams.

3. **Adoption**. Experiments are sanctioned in the form of a full-time team. Most critical systems are covered. Stakeholders are invited to and participate in the experiments when they run.

4. **Cultural Expectation**. Chaos Engineering is part of the overall engineering culture. All critical systems and most noncritical systems are covered. Participation in the experiments is required.

Figure 2.1 shows an example of what a ChMM map can look like, based on Chaos Engineering at Netflix and which Chaos Engineering tools the organization relies on. The map in the figure includes three tools: Chaos Monkey, Chaos Kong and Chaos Automation Platform (ChAP). Their placement in the diagram was made by Netflix engineers based on the status of using that specific tool in the Chaos Engineering practice in their organization in 2017, aiming to give an example of how the model can be applied [7]. As can be observed in the figure, Chaos Monkey was at the time highly sophisticated and adopted, whereas Chaos Kong was slightly less advanced in both areas but still in the top-right quadrant of the map. ChAP had reached a reasonably high level of sophistication but was still “In the Shadows” when it came to adoption. The arrow in the figure shows the desired outcome of Netflix’s next planned efforts in extending the usage of the tool. This type of insight is what the ChMM can assist in obtaining.

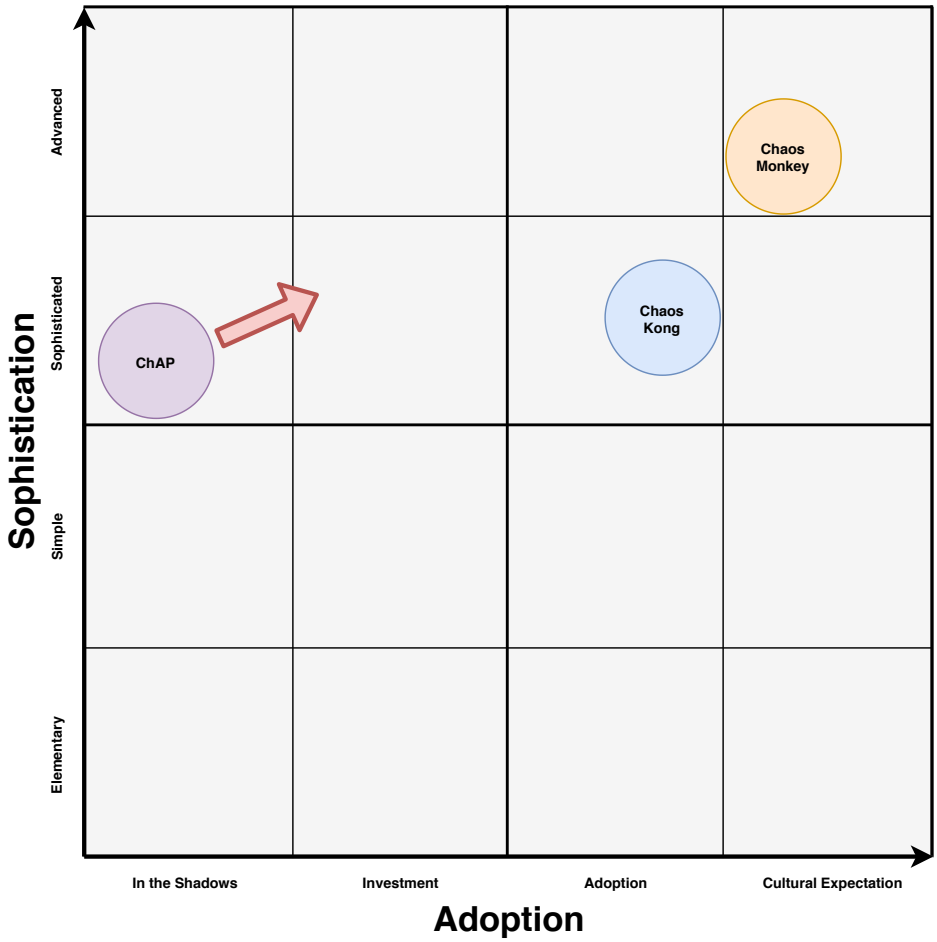


Figure 2.1: The Chaos Maturity Model applied to Netflix in 2007.

2.7 Advantages and Challenges with Chaos Engineering

This section presents some possible advantages which can be observed by practicing Chaos Engineering, as well as some drawbacks and challenges with the practice.

2.7.1 Advantages with Chaos Engineering

The fundamental advantage of carrying out Chaos Engineering can be said to be that it has the ability to identify weaknesses before they appear as real incidents and thus reduce the number of incidents which actually occur [20]. In turn, identifying incidents early can have several benefits on its own, and it can prevent not only user frustration but, by extension, economic losses as well. Also, it is generally easier for developers to fix a problem in a piece of code they just worked on as opposed to resolving problems in older code [19].

Chaos Experiments can further help engineers to better understand the systems they develop. Resolving real incidents can, naturally, provide this knowledge as well, but Chaos Engineering is a way of receiving the same types of insights without causing the company any

real pain [12]. Engineers may also learn which parts of the system that are the most critical ones, and which are less critical [20].

Since Chaos Experiments ideally are performed in the production environment, they also provide a sense of realism in the testing according to the previously referenced Chaos Engineering book written by Rosenthal et al. [7]. This gives a type of external validation, and questions such as whether or not the results of the experiment fully generalize to the phenomenon that is really of interest are completely eliminated.

Furthermore, Chaos Engineering can hopefully lead to a shift in the engineering culture of an enterprise, causing engineers to develop more resilient systems from the start. Design conversations tend to begin to center on what happens *when* a component of a system fails, as opposed to what happens *if* the component fails [20].

Finally, Chaos Engineering is applicable even for large-scale and complex systems, due to its empirical nature. It is possible to experiment on such systems even when other types of testing might be difficult to carry out [7].

2.7.2 Challenges and Drawbacks With Chaos Engineering

As Chaos Experiments observes a system at its boundary, Chaos Engineering has been criticized for neglecting the majority of the system, which represents the way it operates internally, when testing it [10]. Also, it has been pointed out that fault injection testing never is enough on its own to improve system resilience [21]. It can take a lot of effort to analyze the results of Chaos Experiments, and although the experiments can sometimes be excellent in identifying systemic weaknesses, they provide nothing in terms of fixing them.

An obvious risk when conducting Chaos Experiments is the possibility of causing negative impact that proves to be difficult to undo. If an experiment leads to a failure because the resilience of the system was overrated in advance, this can in the worst-case scenario cause severe harm [20].

Challenges with Chaos Engineering include the engineering effort of implementing it [20] as well as making it sufficiently systematic to cover enough scenarios [7]. An unsystematic approach can cause the amount of information that can be extracted from the experiments to decrease and is thus suboptimal.

2.8 Continuous Experimentation

Conducting experiments in production is not unique to Chaos Engineering. A similar discipline of experimentation is that of Continuous Experimentation, which is a general term referring to conducting experiments iteratively [22]. The disciplines of Chaos Engineering and Continuous Experimentation are similar, but their aims are different; where Chaos Engineering aims to evaluate and improve the resilience of a software system, Continuous Experimentation aims to evaluate different implementations and thus help select the superior one.

Continuously experimenting with a software system is one way of becoming more data-driven when developing software, as is clear when comparing it to the alternative of performing data analysis in an ad-hoc manner [23]. It is a discipline with a growing popularity,

and the number of organizations using not only customer data but also product data to support product decisions is increasing. The challenge of the discipline is to not just obtain the relevant data, but to also utilize it efficiently and thus be truly data-driven. The number of organizations who overcome this challenge remains to be low, regardless of the popularity of the discipline.

An elemental practice which can be conducted to receive continuous user feedback is to run two variants of a software system to different subsets of the total set of current users, which is known as A/B testing [22]. By measuring the outcome of such experiments in a systematic manner, it is possible to conclude which of the variants is the superior one and then select it for implementation. In other words, the software development can, by following this practice, become guided by evidence provided by real, authentic users. Such experiments are especially useful for software systems providing a service to a large amount of users, and examples of famous companies conducting A/B testing include Facebook, Google and Microsoft.

2.9 Application Containerization

When the number of users of a software system increases, it is vital for the system to be scalable. *Scalability* can be defined as the ability of a system to handle increases not only in number of users, but also in workload, system size and geographic area the system needs to cover [8]. There are two main approaches of improving the scalability of a software system, while at the same time ensuring that overhead costs are minimized and that the application can run on a variety of different platforms [24]. Those two approaches are virtualization and application containerization.

The fundamental idea of *virtualization* is to split computer hardware into multiple logical units called virtual machines [25]. Each virtual machine has its own amount of memory, storage and processing power and behaves just like an independent computer, despite the fact that it actually is only part of one computer. Using this technique it is possible to increase and decrease the resources of a specific virtual machine based on current needs. For instance, it is possible to increase the processing power of a virtual machine on which a suddenly popular software system is running, thus making the system able to handle the workload of an increased amount of users. One drawback of using virtual machines, however, is that every virtual machine needs a copy of an entire operating system to function, meaning that they in total require significant amounts of computer resources [24]. By extension, this also makes virtual machines complicated to move.

When using *containerization* to deploy applications, there is no need to launch a virtual machine for each application. Instead, everything the application needs to run, including its files and libraries, is packaged in what is called a container. Multiple containers can then be deployed on a single virtual machine and share its operating system with each other, making them lightweight and quicker to scale up [24]. Containers are thus more suitable than virtual machines when the number of total servers used for deploying needs to be minimized.

The dominating tool for application containerization is called Docker, which is used for packaging and provisioning containers as well as running them. However, it is common to combine the use of Docker, which can be described as a containerization tool, with a containerization management tool. While the containerization tool provides the fundamental technology of containerization, a containerization management tool can be used to automate

the processes of deploying the containers, scaling them up and down and balancing the loads between them. The dominating management tool for these processes is called Kubernetes. This tool can automatically scale applications, which is useful for instance when the workload of an application is different on weekdays and weekends. It would then be unnecessary to maintain the same amount of resources during the entire week, but to manually increase and decrease them would be too labor-intensive. Kubernetes also has a variety of other features, such as automatically orchestrating storage, rolling out updates in a progressive manner and restarting containers upon failures [26].

The architecture of Kubernetes is illustrated in Figure 2.2. As can be seen in the rightmost part of the figure, the containers live inside what is called pods. A Kubernetes pod can be described as a logical collection of all containers needed for an application to work, meaning all containers which need to interact with each other in order for the application to be able to perform its intended overall function. Pods are hosted inside of nodes, which are the actual worker machines, either physical machines or virtual machines.

The desired configuration of a specific deployment can be described in a file, called the application deployment file, which can then be fed to the Kubernetes Master as illustrated in Figure 2.2. In this file, it is possible to configure the pods and specify the desired number of replicas of each pod. It is then the role of the Master to decide how to divide these pods between nodes and schedule them correctly, and to subsequently ensure that the correct number of replicas of each pod is maintained at all times.

In relation to Chaos Engineering, there are many tools which simulate chaos in Docker and Kubernetes environments, which is clear when studying Table 4.1 in section 4.1. The tools commonly define their attacks in Docker and Kubernetes terminology, for instance in terms of which containers, pods and nodes to target during a Chaos Experiment.

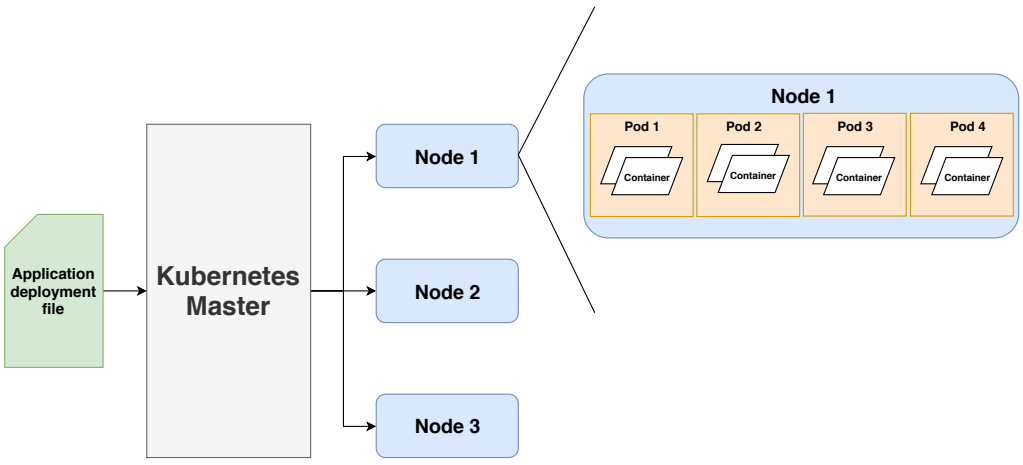


Figure 2.2: A simplified illustration of the Kubernetes architecture.

2.10 Cloud Service Providers

Apart from Chaos Engineering tools for Kubernetes environments, some tools are specified to be of special suitability when using certain cloud service providers. A *cloud service provider* is a company which provides other organizations with some cloud computing component [27].

The term *cloud computing* refers to using the Internet to store data in and access data from a remote server, as opposed to storing the data on the hard disk of the computer in use [28]. This can be reformulated as renting computer power and storage from a remote data center, paying as you go. There are three typical types of cloud computing components which can be offered by a cloud service provider: infrastructure as a service (IaaS), software as a service (SaaS) and platform as a service (PaaS) [27]. In the IaaS case, what is delivered is an infrastructure component such as a server or a storage unit. For SaaS providers, the offer instead includes some business technology, in the form of a software. Finally, PaaS providers deliver tools usually aimed for application development. Cloud service providers often deliver their services in the form of a subscription, commonly paid for twelve or four times per year.

Three cloud service providers dominate the market today, and they are also the ones which are most frequently mentioned as suitable providers in the documentation of Chaos Engineering tools. They are Amazon, Microsoft and Google, offering Amazon Web Services (AWS), Microsoft Azure and Google Cloud Platform (GCP). Which one to choose depends heavily on the workload and other characteristics of a specific project, and it is therefore common for companies to benefit from using multiple providers in different parts of their organizations [29].

AWS was released in 2006 and provides services which can be likened to building blocks, designed to be able to cooperate with each other. The services are categorized into several broad categories, including computing, security and database services [30]. The computing services include EC2, which sets up remote virtual machines that can be grouped into what is called ECS clusters. The EC2 instances can also be given block-level storage with the service EBS. The computing services also include Lambda, which is a serverless service for running functions in the cloud, and EKS, which allows for using Kubernetes in AWS environments. Other types of services include IAM for managing user identities and accesses, CloudWatch for monitoring AWS environments and triggering alarms when necessary, RDS for providing relational databases in the cloud and ELBV2 for balancing loads. EC2 instances and ECS clusters are common targets of Chaos Experiments in AWS environments, as are instances and clusters of the other mentioned services above.

Microsoft Azure was released in 2010, four years after AWS. Plausibly related to this is the fact that AWS offers over 200 services while Azure offers approximately half of that [31]. However, Microsoft Azure is easy to integrate with other Microsoft tools, making it suitable for organizations who already use numerous Microsoft components. As with AWS, Azure's services are categorized into broad categories, including computing, storage and databases. Two of the computing services are Azure Virtual Machines (VMs), hosting Windows and Linux virtual machines, and Azure Virtual Machine Scale Sets (VMSS), used for scaling the hosted Windows and Linux virtual machines, which lets the user create and manage multiple virtual machines that are identical to each other and at the same time balance the loads between the virtual machines. Similar to AWS EKS, another computing service that Azure provides is Azure Kubernetes Servcice (AKS), which allows users to manage clusters of virtual machines on which containerized services run. VMs, VMSS and AKS can all be experimented with using Chaos Engineering. When comparing Azure to AWS, it can be concluded that AWS is more commonly used than Microsoft Azure, having a market share of around 30 % compared to Microsoft Azure's 16 %. However, when it comes to growth rate, the numbers are reversed. Microsoft Azure has shown an annual growth rate of 75 % as of 2019, whereas the same number for AWS is 41 %.

GCP was made publicly available in 2011 and is thus the youngest, and also smallest in terms of number of services and market share, provider of the three. It offers around 60 services and has a market share of approximately one tenth of the total market [31]. It is, however, the provider with the highest annual growth rate, namely 83 % as of 2019.

The three providers are similar to each other in various ways, not excluding functionality and suitable use areas. They all come with their own sets of benefits and drawbacks, making it impossible to distinguish a universal provider to use in all cases. This contributes to the fact that which provider to choose is highly dependent on the characteristics of a specific project and can therefore explain why organizations often choose to use several providers in their operations.

Chapter 3

Research Approach

This chapter contains a description of the project's research approach. The approach has a problem-solving nature, meaning that it aims to propose a solution for a problem which has been experienced, which, furthermore, is the aim of most software engineering research [32]. The research paradigm followed during the project was the design science paradigm, which consists of three types of research activities: problem conceptualization, solution design and empirical validation [33].

Problem conceptualization is key in design science research since design science research is focused around real-world problems. The methods which are performed during the problem conceptualization aim to generate an understanding of a general problem in terms of a number of so-called problem instances, or, in other words, concrete real-world examples of how the general problem has materialized. A problem conceptualization is not equivalent to a problem description, since a conceptualization is based on a specific solution envisioning and contains aspects such as characterizing the domains of both the problem and the solution. In this case, the envisioned solution is the framework for Chaos Engineering.

During the *solution design*, the problem is mapped to a solution. This is a creative activity which should not be separated from the problem conceptualization, which aligns well with the fact that the problem conceptualization is carried out based on an envisioned solution.

Finally, the last research activity, *empirical validation*, consists of evaluating whether or not the proposed solution works in a real-world context, or, at least, a context similar to the real-world one. The activity thus aims to determine whether or not the proposed solution addresses the problem successfully.

This chapter is structured as follows. Section 3.1 describes the methods which were all used to work towards the problem-solving goal of the design science research paradigm, and how these methods relate to the three research activities described above is illustrated in Figure 3.1. Section 3.2 then lists the delimitations which limited the scope of this project.

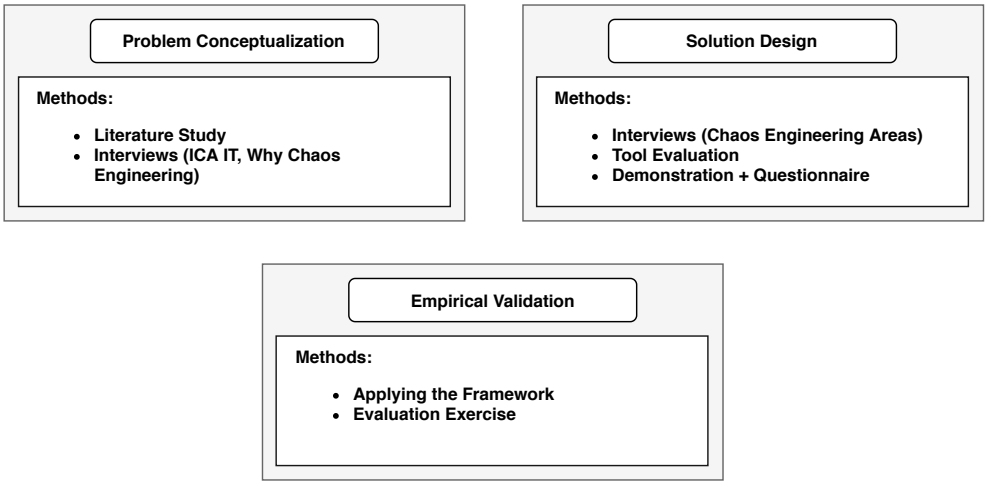


Figure 3.1: The methods which supported the problem conceptualization, solution design and empirical validation research activities.

3.1 Methods

This section describes how the methods in Figure 3.1 were carried out. Subsections 3.1.1 through 3.1.6 describe one of the six methods each.

3.1.1 Literature Study

A literature study was conducted in the area of Chaos Engineering as part of the problem conceptualization, in order to examine the current knowledge of the practice. The purpose was to gain an understanding of Chaos Engineering and find information on how it can and should be implemented, and at the same time generate a general sense of context in terms of how the discipline relates to other aspects of software testing. The goal was in other words to discern the core principles and definitions needed to give a relevant introduction to Chaos Engineering, as defined by the author of this report, while keeping the aim of the project as a whole in mind. This means that the aim was not to cover and summarize all existing pieces of literature on Chaos Engineering, partly because of the project's time limitation and partly because the literature study was simply the introductory part of the project, not its main focus.

The literature used in this literature study was initially found by searching the databases included in the Lund University Libraries, using the terms "Chaos Engineering" and "Software Chaos Engineering" and limiting the subjects of the results to subjects related to software engineering. When a piece of literature describing Chaos Engineering in some way was found, its abstract was read in order to determine whether or not its contents were relevant to use in the project. Here, pieces of literature were deemed irrelevant if they described implementations of interdisciplinary Chaos theories in general engineering fields or if they described specific Chaos Engineering tools or systems. These tool or system descriptions were instead saved to facilitate the start of the project's tool evaluation, described in subsection 3.1.3. If the decision regarding the relevance of a piece of literature could not be made by reading only the abstract, the conclusion was screened as well. The pieces of literature which were

considered to be useful after the initial screening were read and summarized, before they were given a unique ID and their general information, including title, authors and dates of publication and retrieval, was logged using the reference management tool RefWorks. The unique ID's were used to maintain a traceability throughout the project and to more easily be able to refer to the pieces of literature, without confusing them with each other. After each summarization, backward snowballing was performed, meaning that the references of the found piece of literature were screened in hopes of discovering additional publications and authors involved in subjects related to Chaos Engineering. Finally, the set of summaries was compiled into the information found in sections 2.3 through 2.7. For a list of the literature used in this report, see References.

3.1.2 Interviews

Interviews were held with various stakeholders at ICA. The interviews were organized in terms of the following themes: ICA IT, Why Chaos Engineering and Chaos Engineering Areas. For each theme, a number of people were interviewed to gather information relevant to that theme. Table 3.1 lists the people who were interviewed along with a description of their role at ICA and the date and theme of the interview. The structure of the interviews differed for the different themes, which is described below. However, common for all interviews was that the interviewees' answers were written down on a computer while the interviewees spoke, and that they at the end of the sessions were given a chance to validate how their answers had been perceived. More information on this and the overall structure of the interviews can be found in Appendix A.

The theme *ICA IT* aimed to gather information on how the IT department at ICA operates and which services it delivers. The interviews were also held to receive a first insight into the problems the IT department has suffered and they were, in other words, part of the problem conceptualization activity. Since the interview theme had a main purpose of giving an orientation to the ICA IT department, no effort in making the interviewee selection as representative as possible was made and only two people were interviewed. These interviewees were selected based on recommendations given by interviewees of other interview themes or other employees who came in contact with the project for one reason or another. The interviews were openly structured with question areas according to the interview guide found in Appendix A. The majority of the findings from the ICA IT interviews is not included in this report, since both interviews focused solely on ICA's operations and IT department in a way which was unrelated to a possible inclusion of Chaos Engineering. They were, however, useful when it came to getting familiar with the workplace, and they generated several useful contacts in the organization. The parts of the interview results which were included in the report are presented in section 1.1.

The theme *Why Chaos Engineering* aimed to gather motivation for why Chaos Engineering should be implemented at ICA by qualitatively examining the need for an improved software resilience. The selection of interviewees was based on stratification in an attempt to cover some of the variation in the population, and the categories used were "general employee"—"incident employee". An "incident employee" interviewee was defined as a person who is directly involved in solving IT incidents at ICA, whereas a "general employee" has other work tasks in the company's IT department. The interviews were semi-structured with questions according to Appendix A. In relation to the design science research paradigm, the Why

Table 3.1: A list of the people who were interviewed during the project along with their role at ICA, the date of the interview and the theme of the interview. Names have been omitted to establish anonymity and the interviewees have instead been given unique IDs.

Interviewee	Role Description	Date	Theme
P1	General Employee	2019-12-05	Why Chaos Engineering
P2	Incident Employee	2019-12-10	Why Chaos Engineering
P3	Developer	2019-12-16	Chaos Engineering Areas
P4	Incident Employee	2019-12-12	Why Chaos Engineering
P5	Scrum Master	2019-11-08	ICA IT
P6	IT Architect	2019-11-18	ICA IT
P7	Change Manager	2019-12-12	Chaos Engineering Areas
P8	Test Engineer	2019-12-11	Chaos Engineering Areas
P9	Developer	2019-12-16	Chaos Engineering Areas
P10	General Employee	2019-11-11	Why Chaos Engineering
P11	Solution Architect	2019-02-17	Chaos Engineering Areas
P12	Developer	2019-02-17	Chaos Engineering Areas

Chaos Engineering theme was part of the problem conceptualization. Since the aim of the interviews was to qualitatively examine the need for Chaos Engineering, the results were only processed by categorizing specific problem incidents mentioned by the interviewees into general categories of reasons why an improved resilience was considered to be of value. No effort in quantifying these results were made, due to the nature of the interviews. The findings from the Why Chaos Engineering interviews are presented in section 6.4 and discussed in section 7.1.

The theme *Chaos Engineering Areas* aimed to gather information on where at ICA Chaos Engineering is suitable. It was therefore part of the solution design activity. The selection of interviewees was based on the results of the questionnaire described in subsection 3.1.4. The interviews were semi-structured with questions according to Appendix A. The findings from the interviews were used to identify individuals who would be interested in participating in the research methods "Applying the Framework" and "Evaluation Exercise", described in subsections 3.1.5 and 3.1.6. No other findings from these interviews are presented or discussed in this report.

3.1.3 Tool Evaluation

There is currently a wide range of tools for Chaos Engineering and they differ when it comes to a range of factors, for instance functionality, prerequisites and age. Therefore, some of the available tools were evaluated as part of the project's solution design, with the aim of gaining insight into which tools were appropriate components of the Chaos Engineering framework. The tool evaluation was performed in a process containing four phases: tool characterization, tool selection, tool experimentation and tool comparison. Figure 3.2 illustrates how the

phases were related to each other, the actual activities they contained and the outputs they generated.

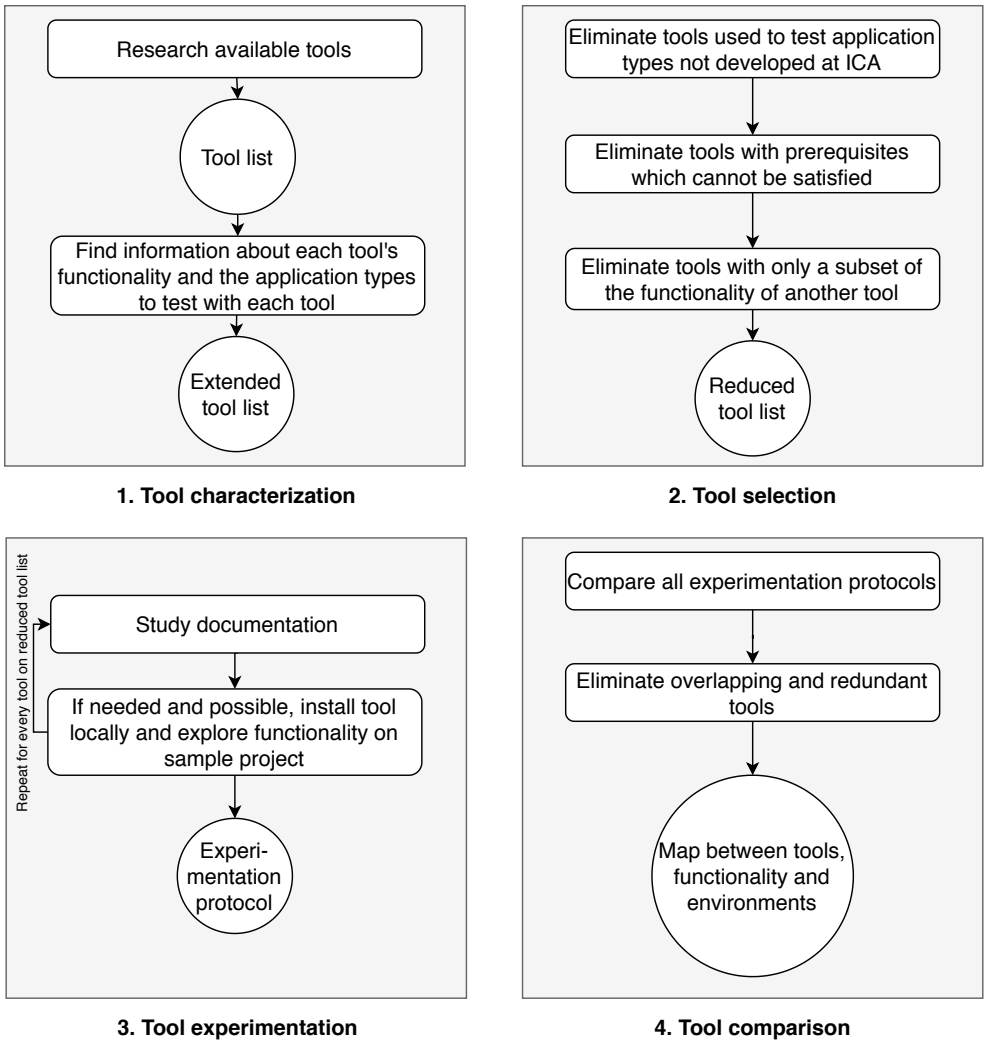


Figure 3.2: The tool evaluation process. Rectangles symbolize activities and circles symbolize outputs.

The *tool characterization* phase started by researching available Chaos Engineering tools, an activity which was partly initialized during the literature study since any mentioned Chaos Engineering tool in the studied literature was included in the reference summaries described in subsection 3.1.1 and references describing single tools were saved for this purpose exclusively. Also, additional tools were found by using the search string "chaos engineering tool OR chaos engineering open source tool OR oss chaos engineering tool OR <tool name> alternatives OR chaos engineering resources", where <tool name> was replaced with names of tools which had already been found. The purpose of this was to generate useful results of two types. The first type was the actual tools, in terms of links to specific Chaos Engineering tool repositories or, in some cases, websites describing a Chaos Engineering tool. The second type was repositories summarizing other Chaos Engineering resources, such as `https://github.com/dastergon/awesome-chaos-engineering`. When compared to traditional search engines, such as Google, testing to use the search string to search the Lund

University Libraries databases proved unsuccessful in terms of the number of results it generated, which is why the Google approach was favored in this case. All found resources were subsequently screened in hopes of finding references to other tools. The activity generated a list of tool names and short tool descriptions. Then, the posts on the list were extended by adding information about what the tools can do and where the tools are applicable, in terms of the applications they can be used to test. Such information was found by reading the introductory parts of the tools' documentation and browsing the information on the developers' websites. Finally, the tool list was homogenized to ensure that tools were described in similar and cohesive terms. This was done by creating categories of tool functionalities and application types to test with the tool, respectively, and categorizing each tool into one or more categories of both types. The extended tool list and these categories can be found in section 4.1.

Next, during the *tool selection*, the extended tool list was used to select tools for further evaluation. This selection started by removing tools used for testing application types which are not developed at ICA. Then, tools with prerequisites which for some reason would be impossible to satisfy given the current circumstances at ICA were also removed. For these two types of eliminations, an IT architect at ICA was consulted. Finally, the categories defined during the tool characterization phase were used to find overlapping tools, before every tool with only a subset of the functionality of another tool was eliminated. All eliminations are described and motivated in section 4.2, for the sake of transparency. The tool selection phase generated a reduced tool list as output.

For every tool on the reduced tool list, the *tool experimentation* phase was then iterated. The tool experimentation aimed to fill out the protocol in Appendix B for every tool on the reduced tool list. During this phase, a larger part of the tool's documentation, or, in most cases, the tool's entire documentation, was studied. If it was possible and needed, the tool was then installed locally and the types of attacks the tool could perform were explored on an arbitrary sample software. Here, "needed" refers to two things: either it was needed to install the tool because it was impossible to fill out every field of the protocol in Appendix B based on the documentation only, or it was needed because it was decided that the tool's entire functionality had not yet been uncovered or understood. "Possible" refers mainly to word "locally" in the install activity, since not all tools are to be used in a local environment. Due to the time limitation of the project, it was declared unfeasible to set up appropriate environments for all tools on the reduced tool list, which for instance would include cloud environments and environments for containerized applications. The results of the tool experimentation phase are presented in section 4.3.

Finally, during the *tool comparison* phase, the experimentation protocols were analyzed to find similarities and differences between the tools. Tools which were found redundant from an ICA perspective were eliminated, which is described and motivated in section 4.4. Here, the word "redundant" refers to a tool with a functionality that is a complete subset of the combined functionalities of some other tools. The priority was to eliminate as many tools as possible without removing any piece of functionality, in order to generate a framework that was as wide as possible in terms of functionality and as simple as possible in terms of usability. The output from the phase was a map, illustrating which tools to use for which purposes. This map was also the final output of the entire tool evaluation activity, and it can be found in section 4.4.

3.1.4 Demonstration and Questionnaire

On repeated occasions, a demonstration of Chaos Engineering was held for different stakeholders at ICA. The purpose of this was two-fold. Firstly, it aimed to give an introduction to the discipline to relevant ICA employees, thus creating an awareness of and interest in the subject. By holding the demonstrations for people who were not familiar with the concept of Chaos Engineering in advance, they also became more likely to be able to help with the project and more suitable as future interviewees. Secondly, during and after the demonstration, the participants were asked to fill out a questionnaire, thus giving their answers to questions of importance related to the project.

The demonstrations started by defining Chaos Engineering and Chaos Experiments, after which an introduction to the Chaos Monkey tool was given. To facilitate the demonstrations, an open source example application had been downloaded and slightly modified prior to holding the demonstrations. The application was a Spring Boot application built around the concept of microservices and is available at `https://examples.javacodegeeks.com/spring-boot-microservices-example/`. The modifications made to the original application consisted of adding a dependency to the Chaos Monkey for Spring Boot tool, adding a property to enable a required Chaos Monkey profile when running the application and making it possible to control the tool using HTTP requests. Finally, the main part of the demonstration consisted of applying the Chaos Monkey for Spring Boot tool to the microservices application to simulate three problems: latency, thrown exceptions and random termination of a service. The demonstration aimed to show what a tool for Chaos Engineering could do, not what a complete Chaos Experiment might look like. Therefore, the simple Spring Boot application which was used had no resilience mechanisms to be able to withstand the problems which were simulated during the demonstration. Instead, the problems were fully visible to the user, giving an illustration of what the tool was actually causing.

The accompanying questionnaire was handed out before the demonstration to the people who were physically present and sent out after the demonstration to the people who participated remotely over a video call. The questionnaire consisted of five questions, three regarding where at ICA Chaos Engineering might be applicable and two regarding how extensive the respondent's perceived benefits of Chaos Engineering were. The respondents were also free to give any other information they thought relevant or helpful. The five questions are given in their full form in Appendix C. The two questions regarding the perceived benefits of Chaos Engineering (questions 4 and 5 in Appendix C) were also asked during interviews belonging to the Why Chaos Engineering and Chaos Engineering Areas interview themes, if the interviewee had not already given their answer in connection to one of the demonstrations. The three questions regarding where at ICA Chaos Engineering might be applicable (questions 1 through 3 in Appendix C) were asked and more thoroughly discussed during interviews belonging to the Chaos Engineering Areas interview theme.

The results of the two questions regarding the perceived benefits of Chaos Engineering are presented in section 6.3 and simply aim to give an indication of the employees' attitudes towards Chaos Engineering. These results are then discussed in section 7.1. The results of the three questions regarding Chaos Engineering areas helped identify interviewees for the Chaos Engineering Areas interview theme and also helped formulate one of the project's delimitations, and they are therefore presented in section 3.2. Since these results had an impact on the contents of the framework, the "Demonstration and Questionnaire" was placed

in the solution design research activity.

3.1.5 Applying the Framework

To build a useful and relevant framework, and to attain an agile and incremental way of working, early versions of the framework were applied to sample ICA applications as it was being built. The purpose of this was to test how feasible the recommendations presented in the framework were, and also to further demonstrate Chaos Engineering to stakeholders at ICA. Applying the framework was furthermore a way to collect feedback from the people who might ultimately be using the framework to adopt the principles of Chaos Engineering to test their applications. In relation to the design science research paradigm, the method of applying the framework was placed in the empirical validation research activity.

The method of applying the framework was carried out by creating proposals for activities supporting the implementation of Chaos Engineering and then testing to apply the activities to software applications developed at ICA. This was done at meetings with ICA employees who work with the chosen application in some way. At the meetings, the participants were instructed to work through the activities while being observed, so that feedback could be collected regarding the time it took to finish the activity, the ease with which the activities were understood and how well they produced the results they are intended to produce. This observation was obviously subjective and qualitative, but the participants also got to give their own feedback regarding the framework's usability orally during and after the meeting. When it came to selecting the applications and participants, no effort was made in covering any specific variation in the total number of ICA applications or employees. Instead, interviewees or people who had been recommended by an interviewee during the Chaos Engineering Areas themed interviews were contacted. The applications considered were, however, limited to the ones which make up `ica.se` and ICA's e-commerce. This was one of the project's delimitations, which will be further described in section 3.2. The findings from applying the framework are presented in section 6.1 and discussed in section 7.2.

The initial intention was to apply all activities of the framework to some ICA application at least once, to validate the feasibility of all proposed activities. This required interest and time from the development teams of the ICA applications in question, and it was found that there was no lack of interest but unfortunately a great lack of time, due to a number of reasons. The activity took place in the weeks before Easter, which, as mentioned in section 1.1, are always stressful for ICA, but the year the project took place this was particularly true due to an unforeseeable event (the outbreak of the corona virus, SARS-CoV-2) which drastically increased the general demand for groceries. Also, a particularly severe incident caused `ica.se` to go down for several hours during the same time, which shifted all focus away from Chaos Engineering for the time being. This forced the activity of applying the framework to stop when the first activity of the framework, the discovery activity, had been applied three times and the first step of the second activity of the framework, the implementation activity, had been applied once. The remaining activities were thus left unevaluated by this method.

3.1.6 Evaluation Exercise

The final activity of the project was an evaluation exercise, which was part of the empirical validation research activity. The exercise was prepared by planning a simulation of Chaos

Engineering, in which participants would be able to try to apply the framework and hopefully both understand and criticize its contents, regardless of previous experience in software development. The simulation was prepared in the following way: real-world equivalents of a software application and Chaos Engineering tools were defined to be a tower of various building materials and envelopes filled with materials which aimed to bring the towers down. During the exercise, the idea was to liken the participants with development teams and let them build a tower (a software application) and use the framework to decide which of the items in the envelope (which of the Chaos Experiments) were suitable to test the towers with, based on the same logic as the logic behind the recommendations of the framework. Then, the participants were to use their findings when attempting to bring their towers down (and, in other words, test their towers' resilience) before finding improvement opportunities in how to make them more resilient. Prior to the exercise, the different parts of the framework had been converted into simplified versions which were suitable for towers rather than software applications. The real parts of the framework were to be presented during the exercise as well, and hopefully the simulation could give the participants a context in how to use them which could make it easier to understand and criticize them. After holding the Chaos Engineering Areas interviews, it was decided to invite participants with experience in developing ways of working and processes, and therefore, people working as process managers, or in direct contact with process managers, were invited to the exercise.

Unfortunately, the ICA office where the exercise was to take place was closed during the project's last weeks, due to the same unforeseeable event as the one referenced in the previous subsection (the outbreak of the corona virus, SARS-CoV-2). Therefore, the exercise could not take place in the form it was initially planned. Instead, the participants got to participate over a video call, and the task of applying the framework had to be explained to the participants rather than letting them apply the framework themselves. Their feedback was then collected digitally; the participants were encouraged to give feedback orally during the video call, or send it in via email if that was preferred.

After the exercise, the idea was to use the opinions of the exercise's participants as a foundation to make the final changes to the framework. The results of the method are presented in section 6.2 and discussed in section 7.2.

3.2 Delimitations

In order to limit the scope of the project, the following delimitations were made:

- Only open source tools were considered during the tool evaluation. The reason for this was the project's time span of 20 weeks.

- The applications at ICA which were considered when building the framework were the ones which make up `ica.se` and ICA's e-commerce. For instance, the framework was never applied to any other application during the project and the tools were during the tool evaluation evaluated based on the infrastructure of these applications only. The results leading to this delimitation came from asking IT professionals at ICA where the principles of Chaos Engineering are best suited in the organization, in terms of the company's software applications. There were clear trends in the given answers. 23 professionals were asked, out of which 18 chose to give an answer to the question. The

respondents were allowed to state any number of software applications, resulting in a total of 15 unique answers appearing. Out of these, six answers appeared more than once. The six reappearing answers had different degrees of popularity. Four of the answers were present in between 11 % and 22 % of the answers, which can be considered low in comparison to the remaining two answers, which were precisely `ica.se` and ICA's e-commerce. 50 % of the answers included the e-commerce whereas two-thirds of the answers included `ica.se`. Therefore, the project was limited to these applications.

- As will be described in section 5.6, the process of introducing new tools at ICA consists of seven conceptual steps. Recommendations regarding how to introduce the tools which were evaluated during the project only took the first step into account, in which a Proof of Concept is performed to explore the tool's value in one of ICA's development teams. This decision was made based on a recommendation from ICA's continuous delivery center.

Chapter 4
Tool Evaluation

This chapter presents the findings from the tool evaluation which was performed as part of the project's solution design. As illustrated in Figure 3.2, the tool evaluation consisted of four phases: tool characterization, tool selection, tool experimentation and tool comparison. The findings from these four phases are presented in sections 4.1 through 4.4, respectively.

4.1 Tool Characterization

After researching the available open source Chaos Engineering tools, a total of 27 tools were found. This is not claimed to be an all-covering number, but it was decided in collaboration with the project's supervisor at ICA that it was sufficient for the sake and scope of this project. Names and descriptions of the tools are listed in Table 4.1, along with suitable applications which the tools can be used to test according to their documentation and developer websites.

Table 4.1: The extended tool list which was produced during the tool characterization phase.

Tool ID	Tool Name	Description of Functionality	Suitable Applications to Test
T1	Chaos Monkey	Used for instance termination. Terminates virtual machines and containers.	Applications managed with Spinnaker (a continuous delivery platform)
T2	kube-monkey	Used for instance termination. Terminates Kubernetes pods.	Applications managed with Kubernetes

T3	Chaos Toolkit	Used for instance termination, simulating network problems and stressing machines. Exact functionality depends on installed drivers.	Applications running on AWS, Azure, Cloud Foundry (a cloud computing platform) or GCP. Applications managed with Kubernetes
T4	ChaoSlingr	Used for security Chaos Engineering. Performs changes to VPC security groups.	Applications running on AWS
T5	PowerfulSeal	Used for instance termination. Terminates Kubernetes pods and nodes.	Applications managed with Kubernetes
T6	drax	Used for instance termination. Terminates DC/OS tasks.	Applications running on DC/OS (a distributed operating system)
T7	WireMock	Used for simulating network problems. Mocks HTTP APIs which the application under test relies on.	Applications relying on other services
T8	Pod-Reaper	Used for instance termination. Terminates Kubernetes pods.	Applications managed with Kubernetes
T9	Muxy	Used for simulating network problems. Is a proxy for the transport, session and HTTP layers.	Applications relying on other services
T10	Toxiproxy	Used to simulate network problems. Is a proxy for the session layer.	Applications relying on other services
T11	Pumba	Used for instance termination and simulating network problems. Terminates Docker containers and injects network delays.	Applications containerized with Docker
T12	Blockade	Used for instance termination and simulating network problems. Terminates Docker containers, adds delays, simulates packet losses and duplicates and partitions the network.	Applications containerized with Docker
T13	Chaos Lambda	Used for instance termination. Terminates AWS EC2 and ECS instances.	Applications running on AWS

T14	Chaos Monkey for Spring Boot	Used for instance termination, simulating network problems and finding implementation faults in an application's source code. Terminates instances of, adds delays to and throws exceptions in Spring Boot Controllers, RestControllers, Services, Repositories and Components.	Spring Boot (Java) applications
T15	Byte-Monkey	Used for finding implementation faults in an application's source code. Throws exceptions, adds method call delays and nullifies arguments.	Java applications
T16	GomJabbar	Used for instance termination. Terminates running services in private cloud environments.	Applications running in private cloud environments
T17	Turbulence	Used for stressing machines and simulating network delays. Imposes CPU/RAM/IO loads, adds delays, simulates packet losses and partitions networks.	Applications running on Cloud Foundry
T18	Chaosblade	Used for instance termination, simulating network problems, finding implementation faults in an application's source code and stressing machines. Exact functionality depends on installed executors.	Applications containerized with Docker. Applications managed with Kubernetes. Java and Go applications
T19	Cthulhu	Used for instance termination. Terminates Kubernetes pods, AWS EC2 instances and GCP virtual machines.	Applications running on AWS or GCP. Applications managed with Kubernetes
T20	Byteman	Used for finding implementation faults in an application's source code. Throws exceptions and adds method call delays.	Java applications
T21	Litmus	Used for instance termination, simulating network problems and stressing machines. Terminates Docker containers and various Kubernetes components, adds delays, simulates network losses, corrupts network packets, imposes CPU loads and performs disk fills and disk losses.	Applications containerized with Docker. Applications managed with Kubernetes

T22	Perses	Used for finding implementation faults in an application's source code. Throws exceptions and adds method call delays.	Java applications
T23	ChaosKube	Used for instance termination. Terminates Kubernetes pods.	Applications managed with Kubernetes
T24	Chaos Dingo	Used for instance termination. Terminates Azure virtual machines and Azure virtual machine scale sets.	Applications running on Azure
T25	Monkey-Ops	Used for instance termination. Terminates OpenShift pods.	Applications managed with OpenShift (an alternative to Kubernetes)
T26	Chaos Lemur	Used for instance termination. Terminates virtual machines.	Applications running on Cloud Foundry
T27	Chaos HTTP Proxy	Used for simulating network problems. Is a proxy for the HTTP layer.	Applications relying on other services

Tools were characterized in two main dimensions: the applications they can be used to test and their functionality, here used interchangeably with the Chaos Engineering attacks the tools are able to perform. For instance, as can be seen in Table 4.1, there are eight different tools which are explicitly suitable for applications that are managed with Kubernetes: T2, T3, T5, T8, T18, T19, T21 and T23. Similarly, if the sought functionality is instance termination, there are eighteen different tools which implement this functionality in some way: T1, T2, T3, T5, T6, T8, T11, T12, T13, T14, T16, T18, T19, T21, T23, T24, T25 and T26.

As a preparation for the following phases, the tools were organized into categories to more easily be able to find overlaps. A total of five categories of Chaos Engineering attacks were identified, based on the functionalities described in Table 4.1. All 27 tools are able to perform attacks belonging to at least one of these five categories. Also, 13 categories of applications to test were identified, based on the suitability information in the rightmost column in Table 4.1. The identified application categories either represent applications needing a certain architectural component to run, such as a tool or a cloud service provider, or applications of certain types, meaning applications written in certain programming languages or relying on certain types of APIs. As with the functionality categories, each tool belongs to at least one of the application categories. The five functionality categories and 13 application categories are listed in Table 4.2, along with the number of tools which belong to the different categories. Descriptions of the functionality categories are given here:

- Instance termination attacks. This category contains tools which can stop, delete or in some other way terminate a component or process needed for the application to run.
- Network attacks. Tools in the network problem category impair network conditions in some way, for instance by adding delays or simulating packet losses.
- Machine-stressing attacks. Here, stressing machines refers to injecting problems close to hardware, such as imposing CPU loads or filling disks.

- Security attacks. Only one tool (T4) explicitly mentions that it can be used for security Chaos Engineering, which here refers to Chaos Engineering aiming to find weaknesses in the system's ability to withstand malicious attacks.

- Implementation attacks. These tools perform attacks to unveil issues in the application's own source code, in terms of incorrect calls to methods, argument declarations, exception handling or similar.

Noteworthy is that the application categories are not mutually exclusive when seen from an application's perspective, and they capture different aspects of a software system, including not only an application's own code but also the infrastructures and platforms it relies on. An application which for instance is managed with Kubernetes is by extension also containerized with Docker, since Kubernetes is an orchestration tool for Docker containers, and, furthermore, it is possible that the same application happens to be written in Java. However, the tools have been placed in application categories based solely on what is explicitly mentioned in their documentation, and a tool which incidentally is able to test Java applications but is specifically aimed towards Kubernetes environments is thus not automatically placed in the Docker and Java categories, only the Kubernetes category. The categories thus represent which perspective a tool takes when used as a Chaos Engineering tool; tools placed only in the Kubernetes category inject chaos only in Kubernetes-native components, regardless of the implications it has on an application that is is managed with Kubernetes or how the application under test is structured otherwise. To find all useful tools for a specific application, every category therefore needs to be evaluated as either relevant or irrelevant for the distinct structure of that application. It is then possible that the found tools will be able to perform Chaos Engineering attacks from various points of view.

An illustration of how the tools are categorized into one or several application categories can be found in Figure 4.1.

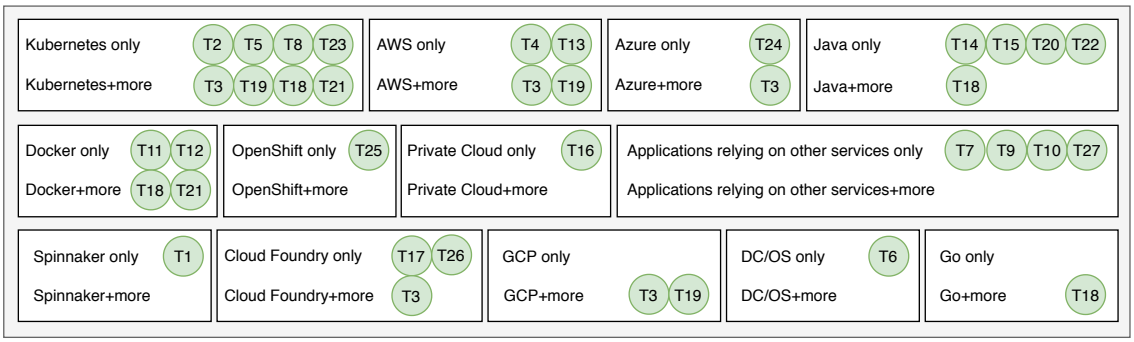


Figure 4.1: The 13 application categories, with shortened names, and the 27 tools placed in their category or categories. Tools belonging to one category only are placed on the top rows, whereas tools belonging to several categories are placed on the bottom rows in corresponding categories.

Table 4.2: The functionality and application categories, and the number of tools belonging to each category.

Functionality category	Nbr of tools	Application category	Nbr of tools
Instance termination attacks	18	Applications containerized with Docker	4
Network attacks	11	Applications managed with Kubernetes	8
Machine-stressing attacks	4	Applications managed with Spinnaker	1
Security attacks	1	Applications managed with OpenShift	1
Implementation attacks	5	Applications running on Cloud Foundry	3
		Applications running on AWS	4
		Applications running on Azure	2
		Applications running on GCP	2
		Applications running in private cloud environments	1
		Applications running on DC/OS	1
		Java applications	5
		Go applications	1
		Applications relying on other services	4

4.2 Tool Selection

To eliminate tools used to test application types not developed at ICA, which was the first activity of the tool selection phase, the 13 application categories shown in Figure 4.1 were used as a starting point. The rule for elimination was: if a tool belongs only to irrelevant application categories, eliminate the tool. Here, the term "irrelevant application categories" refers to the application categories which represent application types that are not currently developed at ICA. The irrelevant categories were defined in collaboration with an IT architect at ICA, based on his knowledge and opinions. The irrelevant categories were defined to be:

- Applications managed with Spinnaker. This tool is not used at ICA.
- Applications running on Cloud Foundry. The consulted IT architect suspected that some limited number of teams might use Cloud Foundry in some limited way, but it is not a widespread cloud computing platform at the company.
- Applications running on GCP. Google does not provide any cloud computing services to ICA.
- Applications running on DC/OS. This distributed operating system is not used at ICA.
- Go applications. No applications are currently developed in the Go language. There exists, however, an interest in the programming language at the company, and it has been planned to investigate its uses at a future point in time.

This resulted in eliminating four tools: T1, T6, T17 and T26. As shown in Figure 4.2, these tools are used only for applications in the irrelevant categories that are listed above. Another category, "Applications managed with OpenShift", was also considered for elimination due to the same reasons as the ones described above for eliminating "Applications running on CloudFoundry". However, the consulted IT architect still thought Chaos Engineering tools for OpenShift environments could be of use in the organization. It was decided to trust his judgement and thus keep "Applications managed with OpenShift" as one of the categories to investigate further.

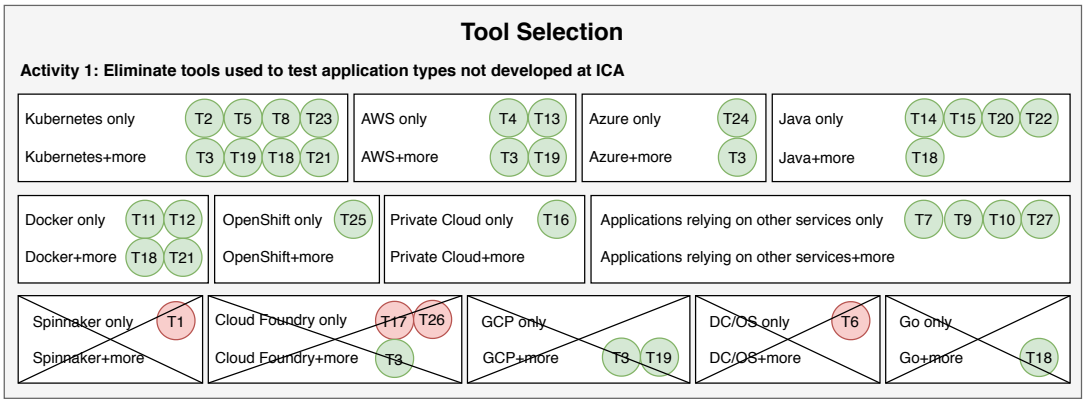


Figure 4.2: The first activity of the tool selection phase: eliminating tools used to test application types that are not developed at ICA.

Next, the prerequisites of the remaining 23 tools were examined to find subjects of further elimination. During this activity it was found that most tools lacked specific limitations apart from the types of applications they can be used to test. Some tools for Kubernetes environments only required the use of a certain version or later of Kubernetes, and tools for other environments had similar requirements. Likewise, tools written in a certain programming language, such as Python, sometimes required a certain version of the language, such as Python 3.5 or later. Such requirements were not considered to be problematic for ICA. However, it was discovered that the main part of the documentation of T18 was in Chinese, thus making it impractical to use at ICA. It was therefore eliminated. The activity thus resulted in eliminating only one of the 23 tools on the tool list, as depicted in Figure 4.3.

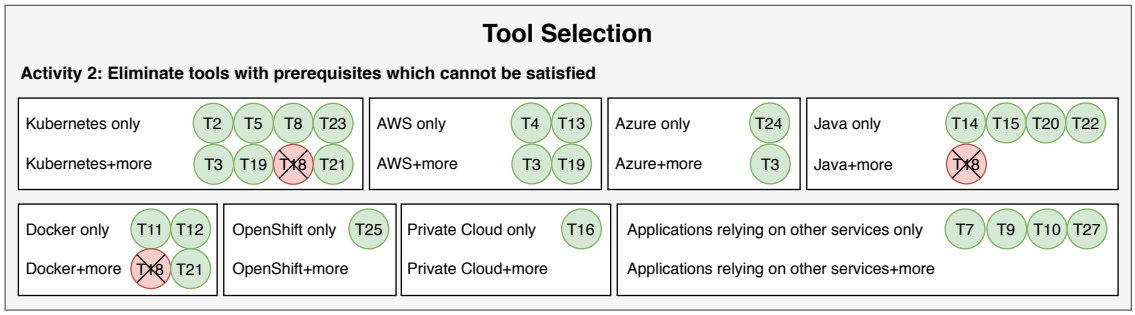


Figure 4.3: The second activity of the tool selection phase: eliminating tools with prerequisites which cannot be satisfied at ICA.

The remaining 22 tools proceeded to the last activity of the tool selection phase, which was to use the functionality and application categories to identify overlapping tools. The aim of this was to examine whether or not some of the tools only had a subset of the functionality of another tool. If that was the case, the smaller tool in terms of functionality was eliminated. This was based on the assumption that a framework consisting of as few tools as possible, while not making any compromises regarding the total number of attacks the tools can perform, would be both easier to use for someone who is responsible for configuring Chaos Experiments as well as easier to introduce in the organisation for someone who is responsible for managing the discipline. The activity was performed in iterations, which are described in Table 4.3. An iteration had an application category as a starting point, and the logic was to start with comparing tools which can be used to test applications in that application category only. Then, in the following iteration, the tools which were not eliminated in the previous iteration were compared with tools which can be used to test applications not only in the category under consideration but at least one other category as well. If the exact functionality of one of the remaining tools from the first iteration was found in a tool included in the second iteration, the tool remaining from the first iteration was eliminated. An exception from this logic was made for the category “Applications running on Azure” for which only two tools were suitable, making only one iteration necessary. Exceptions from the logic were also made for the categories “Applications written in Java” and “Applications relying on other services”, since all remaining tools in those categories belonged to no other categories. Only one iteration was therefore needed in those cases as well. Also, the last iteration had the purpose of comparing appropriate tools with T3 specifically, since T3 was found to be the widest tool in terms of application types it can be used to test.

In total, the activity eliminated ten tools, as shown in Figure 4.4.

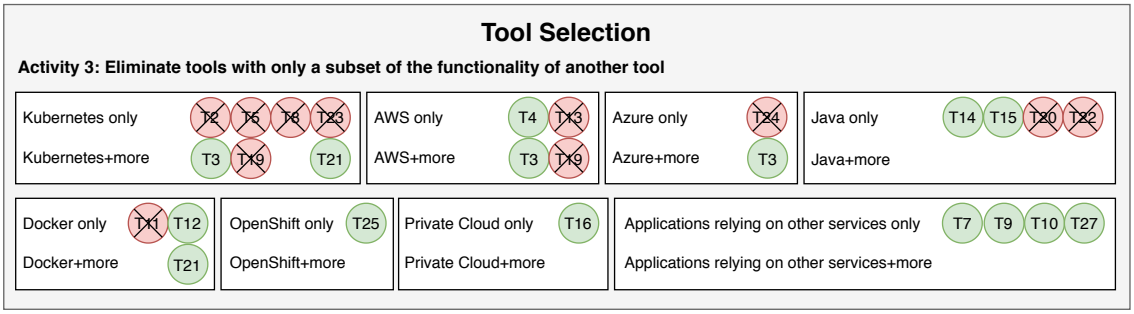


Figure 4.4: The third activity of the tool selection phase: eliminating tools with only the subset of the functionality of another tool.

Table 4.3: A description of the iterations of the third and last activity of the tool selection phase. The categories in the second column were chosen from the lists of functionality and application categories in Table 4.2. The fourth and last column of the table lists the tools eliminated in each iteration.

Iteration	Application Category	Description	Eliminated Tool(s)
1	Applications managed with Kubernetes	Tools used solely in Kubernetes environments were identified to be T2, T5, T8 and T23. All four tools can terminate pods. T5 can also terminate nodes. T2, T8 and T23 are thus subsets of T5 and were therefore eliminated.	T2, T8, T23
2	Applications managed with Kubernetes	T5 (the remaining tool from iteration 1) was compared with tools which can be used in both Kubernetes environments and other environments, namely T3, T19 and T21. The functionality of T5 was found to be a subset of the functionality of T3 since T3 can terminate pods and nodes and also perform other types of attacks.	T5
3	Applications running on AWS	Tools used solely in AWS environments were identified to be T4 and T13. It was concluded that their functionalities do not overlap.	None
4	Applications running on AWS	T4 and T13 (the remaining tools from iteration 3) were compared with tools which can be used in both AWS environments and other environments, namely T3 and T19. The functionality of T13 was found to be a subset of the functionality of T3 and it was therefore eliminated.	T13

5	Applications running on Azure	The only tool used solely in Azure environments, T24, was compared with the only tool which can be used in both Azure environments and other environments, namely T3. T24 can stop and restart Azure VMs and VMSS, which T3 also can, in addition to performing other types of attacks. T24 was therefore eliminated as a subset of T3.	T24
6	Java applications	Tools used solely to test Java applications were identified to be T14, T15, T20 and T22. T14 is used for Spring Boot applications specifically and is alone in that, and so it was decided not to eliminate it. The other three tools can all throw exceptions and introduce latency on method calls. T15 can also nullify arguments, making T20 and T22 subsets of T15. It was also found that T20 has a main purpose separate from Chaos Engineering, which also contributed to its elimination.	T20, T22
7	Applications relying on other services	Tools used solely to test applications relying on other services were identified to be T7, T9, T10 and T27. T7 is a tool for mocking, whereas the other three are tools for proxying. The difference between mocking and proxying is that a mock server simulates responses from the requested service itself, whereas a proxy server only acts as an intermediary between a client and the server which provides the client with the services it requests. T9 is a proxy for HTTP, transport and TCP. T10 is a proxy for TCP. T27 is a proxy for HTTP. The functionalities of the tool were found to overlap, but no complete subsets were found.	None
8	Applications containerized with Docker	Tools used solely to test applications in Docker environments were identified to be T11 and T12. Both tools can terminate Docker containers and inject network delays, and T12 can also simulate other types of network problems. T11 was thus concluded to be a subset of T12.	T11
9	Applications containerized with Docker	T12 (the remaining tool from iteration 8) was compared with the only tool which can be used in both Docker environments and other environments, namely T21. T12 was not concluded to be a subset of T21.	None

10	One or more of the following: Applications running on AWS / Applications running on Azure / Applications managed with Kubernetes	It was deduced that the tool which can test the most number of application types is T3, and iteration 10 therefore compared T3 with tools which can test at least one of the application types that T3 can test. The tools to compare with T3 were identified to be T4, T19 and T21. The three tools were compared with T3 individually. T4 and T21 were not considered to be subsets of T3, but T19 was and it was therefore eliminated.	T19

At the end of the tool selection phase, twelve tools remained on the so-called reduced tool list. In other words, the tool selection phase managed to eliminate 15 of the 27 initial tools in total, thus more than halving the number of tools. Figure 4.4, which shows the remaining tools after the third and last activity of the tool selection phase, also shows the reduced tool list which was the output of the entire tool selection phase. The twelve tools which have not been crossed out in the figure symbolize the ones on the reduced tool list and the ones which proceeded to the tool experimentation phase.

4.3 Tool Experimentation

Since the tool experimentation aimed to fill out the protocol in Appendix B for each of the remaining twelve tools, the phase was carried out in precisely twelve iterations. The findings from the iterations are described in subsections 4.3.1 through 4.3.12, which act as textual descriptions of parts of the experiment protocols. The protocols gathered information of five types: general information (such as name and age), availability (links to code and documentation), functionality (such as types of attacks and how to install and control the tool), environment (such as prerequisites and suitable infrastructure components for applications to test with the tool), and finally community (answers to whether or not the project has been deprecated and if additional community material, such as finished Chaos Experiments that are ready to use, exists). The textual descriptions in the subsections below give information related to at least one of these types for every tool. Since the majority of ICA's process for introducing new tools lies outside of the scope of this thesis, as described in section 3.2, no recommendations will be given regarding whether or not the information stated below makes a tool especially appropriate, or inappropriate, at ICA.

4.3.1 T3: Chaos Toolkit

One of the first findings when experimenting with T3 was that it was designed to be extensible for any system, since it defines an open API. As a result, there now exists a range of

publicly available drivers for the tool, making it possible to use for testing several types of applications. This is the reason why T3 was the widest tool in terms of application types in Table 4.1. When installing the tool (which is done using pip, the Python installer), it is up to the user to select which drivers to install along with the tool's core library.

Apart from making the tool widely useful, the fact that it defines an open API has also resulted in drivers for several other Chaos Engineering tools. What this means is that the functionality of other open source tools can be accessed using T3 as the main tool. In fact, drivers have been made publicly available for three of the other tools on the reduced tool list outputted from the tool selection phase: T7, T10 and T14.

The Chaos Experiments of T3 are configured in so-called experiment files, which are JSON-encoded. An experiment file can define some general information about the experiment, such as title, description, tags and version, but its main parts are the steady state hypothesis and the method. In turn, the *steady state hypothesis* must declare probes, which are activities to measure and observe the system's health. The probes contain a property on their own called tolerance, which defines what the results of the probing should be compared to in order to conclude whether or not the system currently is in its steady state. The *method* can also contain probes, but most importantly it contains actions, which interact with the system to simulate the actual problems that are to be injected into the system.

T3 also provides a command line interface (CLI). The typical workflow of using T3 is to use the CLI to do the following:

1. Execute "chaos discover" and specify any of the tool's publicly available drivers as input. This generates a file listing probes and actions in the specified driver which can be used for the system under test.

2. Execute "chaos init", which takes the file generated in step 1 and asks the user to specify which parts of it to include in an experiment file.

3. Execute "chaos run" which executes the experiment file from step 2 while logging its steps in a journal file.

4. Execute "chaos report", which takes the journal from step 3 and produces a report in either a PDF or HTML format.

As mentioned in Table 4.1, the exact functionality of T3 depends on which drivers are installed. However, the attacks the tool can perform fall under the instance termination, network and machine-stressing categories. By installing appropriate drivers, attacks belonging to at least one of these categories can be executed in AWS, Azure and Kubernetes environments. Table 4.4 contains a matrix illustrating the exact functionality in these environments. This matrix includes only Chaos Engineering attacks, which represent the actions, as they are called in T3 terminology, used to simulate a problem in the system under test. Other types of actions have been omitted, as have probes.

When experimenting with T3, an appropriate format was decided upon for the map which was to be outputted from the tool evaluation. It was decided to construct it in the form of a matrix, with tools on the y-axis and functionality on the x-axis. The tools' functionalities on the x-axis would on the highest level consist of the five functionality categories defined in Table 4.2. Below these five categories, they would be divided into columns representing application types, based on the 13 application categories also defined in Table 4.2

but naturally disregarding the five categories defined as irrelevant. To give two examples, this would create matrix columns for instance termination in Kubernetes environments and machine-stressing in Azure environments. The columns would then be decomposed into subcolumns, for instance splitting Kubernetes into, at least, Kubernetes nodes and pods. The matrix cells, representing the intersection between a tool on the y-axis and an attack in a specific environment on the x-axis, would in the map be highlighted if the tool is able to perform an attack of the corresponding attack type in the corresponding environment. The cells could also contain a specification of the exact attack possible in this subcolumn. Table 4.4 gives an idea of what these types could be, such as termination, detachment and draining. This is technical and completely dependent on the environment. Figure 4.5 illustrates the intended appearance of the map.

	Instance termination																			
	Column 1				**Column 2**				**Column 3**				**...**				**Column n**			
	Subcolumns				*Subcolumns*				*Subcolumns*				*Subcolumns*				*Subcolumns*			
Tool 1	xxx	xxx		xxx													xxx		xxx	
Tool 2			xxx		xxx		xxx													
Tool 3									xxx											
...						xxx		xxx					xxx	xxx	xxx	xxx		xxx		xxx
Tool n										xxx	xxx									

Figure 4.5: An illustration of how the instance termination part of the tool map could look. The other types of attacks would complete the map in its horizontal direction. In the actual map, the column labels would be replaced with application categories. The subcolumns would have individual labels describing decomposed components of corresponding application category. The "xxx" cell texts would be replaced by attack specifications.

It was decided to build the map progressively during the tool experimentation phase, one tool at a time, starting with a blank map and letting each tool add one row to the map along with a varying number of columns and subcolumns, depending on the number of attacks it can perform, application types it is suitable for, components in those application types it can target, and columns and subcolumns already added by previous tools. The total number of possible columns can be calculated to be 5x9=45, for five functionality categories in nine relevant application categories. However, far from all types of attacks can be performed in all environments, which reduces the number of actual columns drastically. By building the map progressively, the intention was to not add more columns — or, by similar argumentation, subcolumns — than necessary.

In the case of T3, it added six columns: instance termination in Kubernetes, AWS and Azure, network attacks in Azure and machine-stressing attacks in AWS and Azure. The columns were decomposed into one or several subcolumns corresponding to the boldface words in Table 4.4. When it comes to instance termination in Kubernetes, the added subcolumns were for pods, nodes, StatefulSets and deployments of microservices. For instance

Table 4.4: The functionality of tool T3, Chaos Toolkit, for applications managed with Kubernetes, applications running on AWS and applications running on Azure.

	Instance termination	**Network**	**Machine-stressing**
Kubernetes	**Pods** can be terminated. **Nodes** can be deleted, cordoned and drained. **StatefulSets** (workload objects for deploying and scaling pods) can be deleted. **Microservices**' deployments can be deleted.	None	None
AWS	**EC2** instances can be terminated, restarted and stopped. The **ECS** service can have clusters and services deleted, tasks stopped and container instances deregistered. **EKS** clusters can be deleted. **IAM** roles can be detached from policies. **CloudWatch** rules can be deleted and disabled and can also have their targets removed. **ELBV2** loadbalancers can be deleted and have their targets deregistered. **RDS** database clusters, instances and endpoints can be deleted, clusters and instances can be stopped, cluster failovers can be forced and instances can be forced to reboot.	None	**EC2** instances can have their EBS volumes detached. **Lambdas** can have their concurrency limits removed.
Azure	**VM**s can be deleted, restarted and stopped. **VMSS** instances can be deleted, restarted, stopped and deallocated. Nodes in **AKS** can be deleted, restarted and stopped. **Web App**s can be deleted, restarted and stopped.	**VM**s can have latency added.	**VM**s can have their number of I/O operations per second increased, their disks filled with random data and their CPUs stressed up to 100 %

termination in AWS, they were for the EC2, ECS, EKS, IAM, CloudWatch, ELBV2 and RDS services. For instance termination in Azure, they were for VMs, VMSS, AKS nodes and Web Apps. For network attacks in Azure, it was for VMs. For machine-stressing attacks in AWS, they were for the EC2 and Lambda services. For machine-stressing attacks in Azure, it was for VMs.

4.3.2 T4: ChaoSlingr

T4 is the only tool out of the examined ones to explicitly have security-based Chaos Engineering as its main aim. T4 runs serverlessly in AWS and thus relies heavily on the cloud service provider's Lambda feature. It is completely AWS-native, meaning that it is scheduled and executed fully in AWS utilizing the Lambda feature.

The key part of T4 is an open and extensible framework, aimed to be a structure which enables anyone to implement their own security-based Chaos Engineering functionality. It was the goal of the developers of T4 to build a wide open-source community, where the extensible design of the tool would encourage new developers to contribute with additional tool functionality to the tool repository. The project has, however, since then been deprecated; the code behind the tool is still available, but it is not actively maintained or monitored. Therefore, the repository has been left with functionality for only one security-based Chaos Engineering attack, in spite of the ambitions of the original developers. This functionality is called PortSlingr, and what it does is to perform unauthorized firewall port changes to see if a firewall, or some other monitoring, alerting or correction mechanism in the AWS environment, is able to detect and preferably block the event. More specifically, the attack consists of opening or closing a port randomly. The target of the attack is an AWS security group, which is a feature of the Virtual Private Cloud (VPC) service. Such a security group acts as a firewall for the AWS EC2 instances it has been associated with. Users can limit the range of their Chaos Experiments by tagging security groups with opt-in and opt-out tags. The targeted security group will then be chosen from the pool of groups tagged as opt-in.

T4 thus added one column to the tool map, namely security attacks in AWS environments. Its only subcolumn was for VPC Security Groups.

4.3.3 T7: WireMock

T7 is a tool for mocking HTTP services, and it can be used in two ways: either as a Java library, to which it is possible to add a dependency, or as a stand-alone process, running independently of the system under test. In the first case, the dependency to add is:

```
<dependency>
   <groupId>com.github.tomakehurst</groupId>
   <artifactId>wiremock</artifactId>
   <version>2.25.1</version>
   <scope>test</scope>
</dependency>
```

The alternative of running it as a stand-alone process is accomplished by downloading a stand-alone JAR for the tool and then executing:

```
java -jar wiremock-standalone-2.25.1.jar
```

The use case for T7 in terms of Chaos Engineering is to simulate what happens when an API which the system under test is reliable on stops working. It is therefore probable that injecting such faulty behavior in the real API would be difficult. The problems WireMock can generate are to send back any HTTP response code that indicates an error, add a latency to the response and create different sorts of corrupted ("bad") responses. When adding latencies to responses, the latencies can be of several types. Firstly, they can be configured to a response-specific fixed value. Secondly, it is also possible to set all latencies to a fixed global value. Thirdly, the latencies can be generated randomly, following a lognormal or uniform distribution. Fourthly, consecutive responses can be grouped into chunks and returned all at once, which aims to simulate a slow network.

The corrupted, or bad, responses can also be of four different types. The first one is a fully empty response. The second one is a malformed response, corresponding to sending a header with an OK status but following it with random meaningless data and then closing the connection. The third one is like the second one, but without the initial OK status header. Finally, the fourth one is to simply close the connection. The typical consequence of this is that a "Connection reset by peer" error is thrown. Noteworthy is that the tool documentation warns for using this fourth type of response on Windows, where it is probable to fail in making the connection reset and instead cause it to hang.

To the tool map T7 added one column, representing network attacks in applications relying on other services. The subcolumn T7 added was named HTTP.

4.3.4 T9: Muxy

T9 is a proxy, and the documentation of T9 defines the tool's functionality in terms of the layers it can operate at in the OSI model. The OSI model, where OSI stands for Open Systems Interconnection, is a framework describing how a networking system works, or, in other words, how data can be transmitted from one point to another [34]. The model consists of seven layers, working in collaboration to transmit the data and ranging from the physical layer at Layer 1 (responsible for the actual connection between devices, containing information in the form of bits) to the application layer at Layer 7 (responsible for producing the data to be transmitted and displaying it to the user).

T9 operates at Layers 4, 5 and 7, named the transport, session and application layers, respectively. The responsibility of the *application* layer is described above. The *transport* layer is often called the heart of the OSI model since is responsible for the end-to-end delivery of the full message, by being in the middle of the bottom three layers (often called hardware layers) and the top three layers (often called software layers). The *session* layer establishes the connections and is also responsible for maintaining them, as well as ensuring authentication and security.

The desired configuration of T9 can be declared in a YAML file, where two components can be defined: proxies and middlewares. The *proxies* can be of HTTP or TCP type. The *middlewares*, which are able to intervene with the requests that are sent to actually inject the problems, can be of the following types: delay, logger, HTTP tamperer, network shaper and TCP tamperer. The *delay* middleware is the most basic one, simply adding latencies to different requests or responses. *Logger* simply logs messages and does not inject any problems.

The remaining three middlewares take different perspectives of the problems they inject. The *HTTP tamperer*, working at Layer 7, can modify the headers or bodies of responses as well as their status codes. The *network shaper* (working at Layer 4) takes the perspective of interfering with the network traffic. It does this by limiting bandwidth, causing packet losses and introducing jitters. Finally, the *TCP tamperer* (working at Layer 5) modifies the bodies of TCP packets, and can truncate final message characters and also randomize characters.

In the tool map, T9 thus added two subcolumns to the column representing network attacks in applications relying on other services, where there previously was only one subcolumn for HTTP added by T7. The two new subcolumns were named TCP and Transport. To align the tool map with the OSI model, the names of the three subcolumns were then altered to include their corresponding layer numbers: 7, 5 and 4, for HTTP, TCP and Transport, respectively.

4.3.5 T10: Toxiproxy

T10 is a TCP proxy, which can be expressed in OSI model terminology as Layer 5 to make it more comparable to T9. The tool consists of two parts: the TCP proxy itself and a client for communicating with it, which is done over HTTP. After downloading the tool, it needs to be populated, which the documentation refers to as letting the tool know which endpoints to proxy and where to proxy them. Such configurations can be written in a JSON file and mainly contains information about appropriate endpoint names, addresses to listen on and address of the upstreams.

After populating T10, it can be used by configuring the application under test to connect through the tool. Then, so-called toxics are what injects problems. The following toxics exist: latency, bandwidth, slow_close, timeout, slicer and limit_data. Their names suggest the problems they inject. *Latency* adds a delay to the requests. *Bandwidth* limits a connection's bandwidth, in the sense of the amount of data which can be transmitted per second. *Slow_close* delays a TCP socket (an endpoint instance) from closing for a specified amount of time. *Timeout* prevents data from passing through, and it can also be configured to close the connection altogether after a certain amount of time. *Slicer* slices TCP data into smaller packets and can add an optional delay between each packet. Finally, *limit_data* allows a certain number of bytes of data pass through before it closes the connection.

T10 did not add any new column or subcolumn to the tool map, since the network attacks in applications relying on other services column already existed, as did a subcolumn for TCP.

4.3.6 T12: Blockade

T12 is used to test applications containerized with Docker. This containerization tool was introduced in section 2.9 on precisely application containerization, where a container was described as a unit in which everything an application needs to run can be gathered. T12 targets such Docker containers. The tool is configured in a YAML file, in which all containers needed for a specific distributed system can be described, as can the links between them and the problems to inject. The configuration file consists of two main sections, one for specifying the system's containers and one for specifying the problems to inject. The *container* section specifies how each container should be launched, which can be described by using a number of parameters such as how many copies of a container to launch or how many seconds T12

should wait before launching it. However, only one parameter is mandatory to enter, namely the one where the Docker image to use for the container is specified. A Docker image can be thought of as a blueprint or a mold, since it acts as a template for how to create the actual Docker containers. A container is thus like a running instance of an image. The problem section of the configuration file, which is referred to as the *network* section, can specify how two types of network problems should be injected. One is called *slow*, and it adds a delay with a certain value and distribution (uniform, normal, pareto or paretonormal) to the network, and the other is called *flaky*, which simulates how often packets should be lost (described as a percentage).

T12 then provides a command line interface (CLI) for activating the problems. The Docker containers specified in the first section of the configuration file are started by running the "up" command. The commands "slow" and "flaky" can then be used to activate the problems specified in the second section of the configuration file. Apart from these two problems, the CLI can also be used to stop and kill some or all containers that are running. It can also introduce packet duplication and partition the network, which simulates a lost network connection. What the partition does is to split the running containers into groups, so that containers in one group cannot communicate with containers in another. The tool documentation mentions a typical usage for this problem, namely testing a leader election system, in which one of the containers acts as a leader and the others answer to this leader. The partition problem could then isolate the leader from the other containers, to ensure that a reliability mechanism such as forcing another container to take over as the leader works as intended.

T12 added two columns to the tool map, corresponding to instance termination in Docker and network attacks in Docker. Both columns contained the same subcolumn for Docker containers.

4.3.7 T14: Chaos Monkey for Spring Boot

T14 is a tool developed specifically for testing Spring Boot applications. Spring Boot is a Java framework, aiming to simplify the process of building an application. Spring Boot is part of the wider Spring framework, which has a main aim of enabling enterprises to easily create Java applications for their businesses. One of Spring Boot's core features is its auto-configuration feature, which makes configuring Spring more simple from a developer's point of view. This is due to the fact that the configuration is done automatically, based on a project's added dependencies. Spring Boot applications also rely on the so-called Spring Boot annotations. Such an annotation is a form of metadata added above methods or classes, indicating that the corresponding method or class has a special purpose in the Spring framework. Scanning a Spring Boot application for a specific part of it can thus be made based on these annotations.

The tool T14 was, as described in subsection 3.1.4, used to demonstrate the principles of Chaos Engineering to various stakeholders at ICA. It was decided that repeatedly holding this demonstration sufficed in terms of experimenting with T14, and so the iteration of the tool experimentation phase for T14 simply consisted of transferring the learnings from the demonstrations to the tool map.

The tool itself consists of a Java library. Installing the tool is therefore done by adding a dependency to the library. The dependency to add is:

```
<dependency>
   <groupId>de.codecentric</groupId>
   <artifactId>chaos-monkey-spring-boot</artifactId>
   <version>2.1.1</version>
</dependency>
```

where version can denote any of the tool's releases or snapshots. In order for the tool to work, the application to test also has to be started with the Spring profile called "chaos-monkey" set to active. This can be done in two ways: running the application with the line

```
java -jar your-app.jar --spring.profiles.active=chaos-monkey
```

or adding the line

```
spring.profiles.active=chaos-monkey
```

to the application's property file. In this file, it is also possible to configure which Spring Boot components to target during the Chaos Experiment. This is done by utilizing the Spring Boot annotation feature, by adding the line

```
chaos.monkey.watcher.<annotation>=[true/false]
```

to the file, where <annotation> can be replaced with any of the five Spring Boot annotations supported by the tool: @Controller, @RestController, @Service, @Repository or @Component. The Chaos Monkey Watcher mentioned in the line above is the tool component which scans the application for the annotations set to true, for the purpose of knowing which parts of the application to attack. Three types of attacks can be performed with T14: injecting latency, throwing exceptions and killing the application. These attacks can be configured during runtime by HTTP requests, assuming that one final change has been made to the application's properties file, namely adding the line

```
management.endpoint.chaosmonkey.enabled=true
```

and ensuring that the line of the file describing which endpoints to expose either includes the Chaos Monkey one or is set to exposing all endpoints, which is denoted by a star (*).

T14 added three columns to the tool map: instance termination in Java applications, network attacks in Java applications and implementation attacks in Java applications. All three columns had only one subcolumn, for Spring Boot applications.

4.3.8 T15: Byte-Monkey

T15 is used to test Java applications. It works on the bytecode level, on compiled code, in other words the machine code in a .class file which has been generated by the compiler from the source code in a .java file. T15 works by introducing small changes in this bytecode, which will simulate the tool's Chaos Engineering problems. The tool is started by downloading its latest release and executing the following line:

```
java -javaagent:byte-monkey.jar=mode:<mode>,rate:<rate>,
                              filter:<filter> -jar your-java-app.jar
```

As can be seen in the line above, the tool has three options: mode, rate and filter. *Mode* symbolizes the problems to inject and <mode> can thus be replaced with either fault, latency, nullify or scircuit (for short-circuit). These names correspond to the problems that are injected. *Fault* throws the first exception which is declared in a method signature, assuming that there exists an exposed public default constructor to use to instantiate it. If not, a generic ByteMonkeyException is thrown instead. *Latency* forces a method to do nothing for a number of milliseconds, before its actual instructions are executed. *Nullify* replaces a method's first argument which is not primitive with null, assuming that the method has arguments and that they are not all primitive. In those cases, nothing happens to the method. Finally, *scircuit* attacks try-catch blocks by throwing exceptions in the beginning of the try block to intentionally force execution of one of the catch blocks. If the mode is not specified when running the tool, the default is the fault mode.

The second option, *rate*, specifies a decimal number which represents the percentage of eligible methods to attack. For instance, if mode is set to latency and rate is set to 0.75, it means that 75 % of the method calls will include an initial pause before executing the method's instructions. If no rate is given the default is 1, which will result in attacking 100 % of the method calls. Finally, the third option, *filter*, enables users to specify a string representing which parts of the file package to attack. The tool documentation gives the example of setting the filter option to "uk/co/probablyfine", which will only activate the attacks in the package tree below precisely uk.co.probablyfine. It is possible to be as specific as filtering down to a single method name, by appropriately customizing the string "package/path/-ClassName/MethodName". Again, there exists a default if no filter is given, which is "*", representing all methods in the package.

If the mode is set to either latency or scircuit, the command line starting the tool can be extended by adding one more option. This option is called "latency" for the latency mode and "tcindex" for the scircuit mode, specifying the number of milliseconds the injected delay should last and the index of the catch block to force execution of, if a try-catch block consists of multiple catch blocks. Defaults are 100 ms and -1 (which represents the first catch block).

T15 did not add any column since a column for implementation attacks in Java applications was added by T14. It did add a new subcolumn however, named Bytecode.

4.3.9 T16: GomJabbar

T16 is explicitly meant for testing applications running in a private cloud environment, meaning a cloud environment dedicated for the use of one organization only. Such a cloud environment is typically also managed by the company in question itself. T16 is integrated with the tool Consul, which is a software for connecting separate services with each other that is usable in various environments, naturally including private cloud environments. Such services are in T16 terminology mapped to so-called modules, which are organized into clusters and can be given tags. These three levels (cluster, module and tag) are used to select the targets to attack with T16.

Installing T16 is a question of cloning its repository and then running the standard build command

```
mvn clean install
```

in it. The tool is then configured in a YAML file consisting of three main sections: executor

class, filters and commands. The *executor class* section simply allows users to choose between one of the two available command executor classes, or replace them both with an own implementation if that is preferred. The *filters* section allows users to specify clusters, modules and tags which should be included and excluded as targets during the Chaos Experiments. Finally, the *commands* section is where it is specified which attacks to perform. Here it is also possible to add own functionality, if it is desired to use the tool for other types of attacks than the two which exist out-of-the-box. These two existing attacks are to gracefully or gracelessly terminate a running service. Here, "gracefully" refers to shutting down the service with init.d, whereas "gracelessly" refers to terminating the service with pkill -9 instead.

T16 added one column to the tool map corresponding to instance termination in private cloud environments, divided into a single subcolumn for Consul modules.

4.3.10 T21: Litmus

T21 is a framework used for testing Kubernetes environments. Its functionality is available in the form of finished Chaos Experiments, and the framework is also extensible, allowing users to implement their own additional experiments. To install the core components of T21, the line

```
kubectl apply -f https://litmuschaos.github.io/pages/
                                        litmus-operator-v1.0.0.yaml
```

needs to be run. These core components are then complemented by installing the actual Chaos Experiments, which is done on a Kubernetes cluster as Kubernetes Custom Resources. Finally, a Service Account needs to be created for the tool, the application needs to be annotated with

```
litmuschaos.io/chaos="true"
```

and a YAML file called chaosengine.yml configuring the so-called ChaosEngine, which connects the application under test with the actual Chaos Experiments, needs to be created. Then, the experiment can be run with

```
kubectl create -f chaosengine.yaml
```

and running the command

```
kubectl describe chaosresult <chaos-engine-name>-
                                        <chaos-experiment-name>
```

afterwards will let the user know if the experiment passed or failed.

The finished Chaos Experiments which are available along with the tool provide the following functionality. A Docker container in a pod can be killed. A pod can be deleted, its disk storage can be filled, latency can be injected into its network, its network can be lost altogether and network packet corruption can be injected. A pod can also have its CPU stressed, which a node also can. A node can furthermore suffer from disk loss and be drained.

If the tool Kafka is used for the application under test and it is deployed as a Kubernetes StatefulSet, two additional Chaos Experiments are available for introducing failures in Kafka

Broker pods and disks. Also, if the tool CoreDNS is used as a DNS server, one additional Chaos Experiment exists for deleting a CoreDNS pod.

T21 can also attack OpenEBS components. OpenEBS is used for attaching storage to containers in Kubernetes environments. If OpenEBS is used for the application under test, T21 can inject failures in pods and containers in OpenEBS pools and targets. It can also introduce network delays and network losses in OpenEBS targets.

In the tool map, T21 contributed to the already existing column for instance termination in Kubernetes environments in the pod and node subcolumns, while adding subcolumns for CoreDNS pods, Kafka broker pods, OpenEBS pods and OpenEBS containers to the same column. It also contributed to the column and subcolumn for instance termination in Docker containers. It added a column for network attacks in Kubernetes environments, with two subcolumns for pods and OpenEBS targets. It also added a column for machine-stressing attacks in Kubernetes environments, with three subcolumns for pods, nodes and Kafka broker disks.

4.3.11 T25: Monkey-Ops

T25 is the only tool explicitly meant for testing applications that are managed with OpenShift, which can be described as an alternative to Kubernetes. At ICA it is Kubernetes which is the most widely used tool out of the two, but it was nonetheless the belief of the IT architect who was consulted as part of the tool selection phase that OpenShift is in use in some parts of the company's software development. T25 can target two OpenShift components, namely pods and DeploymentConfigs, which are templates used to describe the applications that are to be deployed, for instance specifying replica counts and deployment strategies. When targeting an OpenShift pod, T25 can terminate it. When targeting a DeploymentConfig, T25 can only scale the number of replicas, not attack it in another way.

T25 can be installed in two ways, either with the use of Docker, by downloading and running a Docker image, or as a separate OpenShift project, by following a set of steps starting with creating a service account and ending with creating a new application for the tool in an OpenShift project. Regardless of how the tool is installed, it can be configured to run in two different modes, called the background mode and the rest mode. In the *background* mode, the tool runs until it is stopped. In the *rest* mode, it runs only for a certain amount of time. T25 can also be configured with an interval time, defined as the number of seconds between attacks.

To the tool map, T25 added a column corresponding to instance termination in OpenShift with a subcolumn for pods.

4.3.12 T27: Chaos HTTP Proxy

T27 is an HTTP proxy, operating at OSI Layer 7. Installing T27 consists of downloading one of its releases from the tool repository and then building it. It runs as a stand-alone process and is configured in a configuration file with the extension .conf. The tool has the functionality of modifying headers in the form of swapping upper-case and lower-case letters and corrupting them, and in responses the tool can also reorder the headers. Furthermore, it can cause timeouts and redirects and simulate server errors. Finally, it can also simulate a server connection break.

T27 did not add any column or subcolumn to the tool map, since a column and subcolumn already existed for network attacks in applications relying on other services and HTTP, respectively.

4.4 Tool Comparison

This section presents the final tool map outputted from the tool evaluation activity. The tool map can be found in Figure 4.6, and explanations of the attacks specified in the map's cells are given in Table 4.5. In total, the tool map shows 87 different attacks, placed in a total of 17 columns divided into a total of 40 subcolumns. The tool map is also available as a PDF file here: `https://drive.google.com/file/d/1vM6V9ysGPsHi5RYQiYbQtM2ug1O5SoIc/view?usp=sharing`.

The first activity of the tool comparison phase, to compare the experimentation protocols with each other, mainly consisted of homogenizing the terms in the tool map. Since the map was built iteratively during the previous phase, several terms were accidentally used to describe the same functionality, which was fixed during this activity. The second activity, to eliminate redundant tools, resulted in no changes to the tool map, as it was found that no tool was a complete combination of the other tools in the map. The final tool map thus contained all twelve tools which were examined during the tool experimentation phase.

Table 4.5: Specification of Chaos Engineering attacks.

Instance termination attacks	
Termination	Terminates a component. Can also be referred to as Deletion.
Restart	As Termination, but also restarts the component.
Stopping	Stops a running process.
Detachment	Used for components which are connected to other components. Detaches the targeted component from the ones it is connected to.
Disabling	Used for rules. Sets the targeted rule as inactive.
Target removal	Used for rules. Makes components previously affected by the targeted rule unaffected by it.
Failover	Used for databases. Makes a database unavailable.
Blocking	Prevents further scheduling on components that other components or processes are scheduled on.
Draining	As Blocking, but also evicts currently scheduled components and processes.
Network attacks	
Delay	Adds a latency to a network connection.
Network loss	Temporarily detaches a network connection.
Network partition	Splits a network of components into groups.
Bandwidth limitation	Limits the number of transmitted bytes per second.
Data limitation	Limits the total number of transmitted bytes.
Closing delay	Prevents an endpoint from closing for a time.
Data stop	Prevents data from passing through.
Data slicing	Slices data into smaller separate packets.
Packet loss	Loses packets being transmitted.
Packet duplication	Duplicates packets being transmitted.
Jitter introduction	Introduces timing errors.
Error	Returns an error.
Content change	Modifies a header or a body.
Redirection	Temporarily or permanently redirects packets.
Connection break	Closes a network connection.
Machine-stressing attacks	
CPU stress	Stresses a CPU value.
I/O increase	Increases the number of I/O operations per second.
Disk storage fill	Fills a disk storage with random data.
Disk loss	Detaches a disk from the rest of the machine.
Security attacks	
Unauthorized port change	Opens or closes a firewall port.
Implementation attacks	
Exception	Forces an exception to be thrown.
Method call delay	Pauses execution in the beginning of a method call.
Argument nullification	Replaces an argument with null.

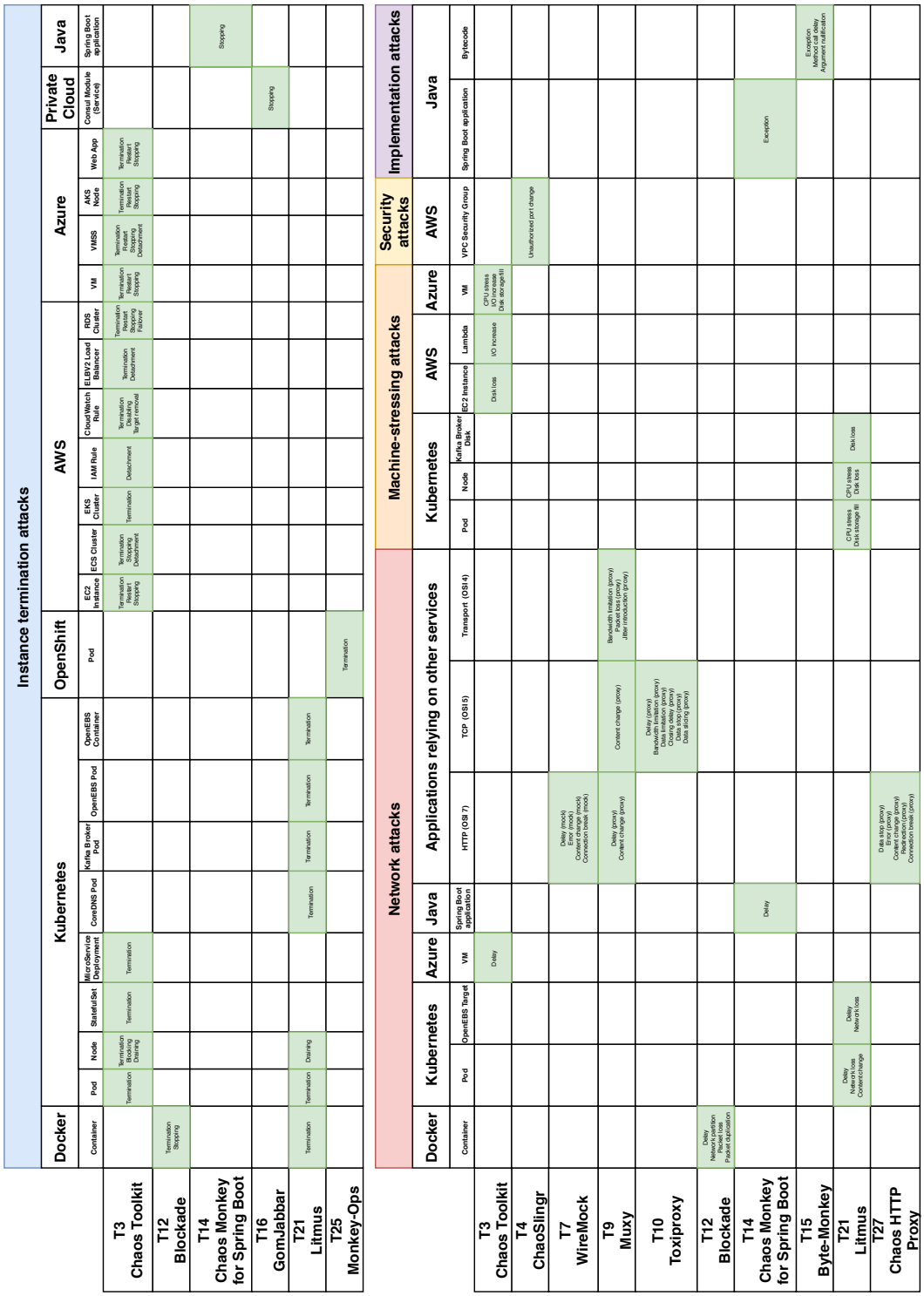


Figure 4.6: The tool map. The horizontal axis contains columns for specific functionalities in specific application categories, which are further decomposed into subcolumns describing the component or service which is attacked. The vertical axis contains tool IDs and tool names. Every filled cell contains a textual description of the attacks which corresponding tool can perform in the environment that is specified by the horizontal axis.

Chapter 5
Proposed Chaos Engineering Framework

This chapter presents the framework for Chaos Engineering in its full form, completing the project's solution design which started by producing the tool map as described in the previous chapter. The framework represents a suggested way to work continuously and iteratively with Chaos Engineering at ICA, from the perspective of the company's development teams. It aims to contain all the necessary building blocks in order for the teams to start using Chaos Engineering to test the applications they develop. The recommended way of working has been split into *activities* to perform, where each activity contains sets of steps, support documents and outputs. The *steps* are the actual actions to perform, the *support documents* are the resources needed to carry out the steps, and the *outputs* are the results of the steps.

As illustrated in Figure 5.1, there are four proposed activities in the framework: discovery, implementation, sophistication and expansion. They are described individually in sections 5.1 through 5.4 and are then combined into the full framework in section 5.5. This section also gives an explanation to the arrows in Figure 5.1, which aim to illustrate how the activities are suggested to be repeated. The last section of the chapter, section 5.6, presents how tools are introduced at ICA, a topic of interest when it comes to applying the framework to ICA's applications.

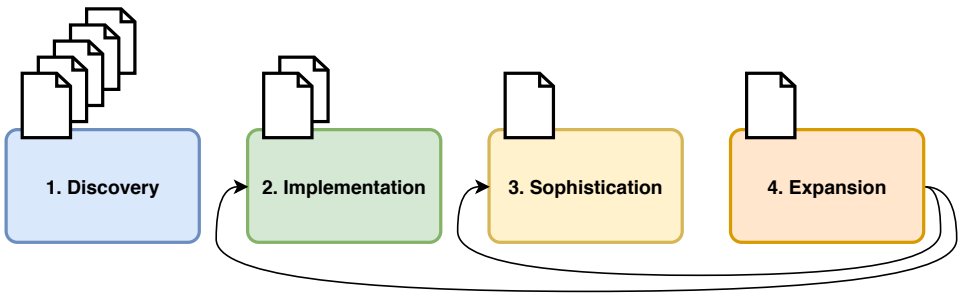


Figure 5.1: The four activities of the framework. The documents in the picture illustrate the number of support documents which are meant to be used when performing the different activities.

5.1 Activity 1: Discovery

The *discovery* activity builds a backlog of Chaos Experiments which are possible and suitable to run for the application under test. It also starts to build on a set of improvement opportunities for the application, which succeeding activities will continue to build on. Apart from the experiment backlog and improvement opportunities outputs, the discovery activity also consists of five steps and five support documents, as illustrated in Figure 5.2.

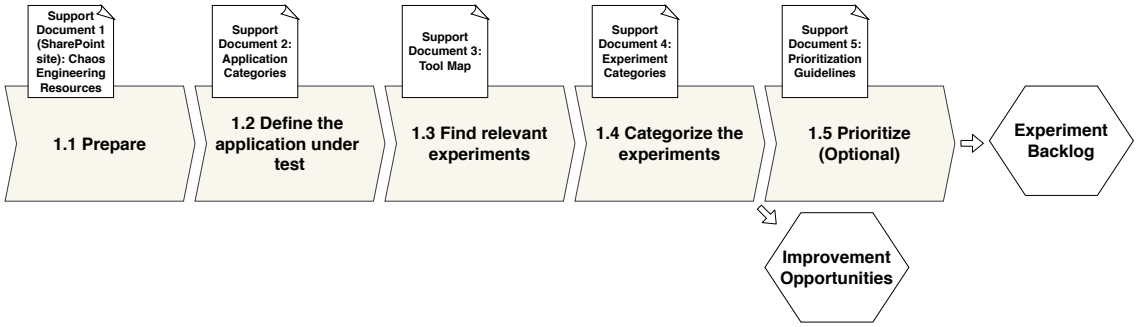


Figure 5.2: The first activity of the framework: discovery.

The steps that are recommended to follow during the discovery activity are:

1.1 Prepare. The first step consists of getting familiar with the framework and Chaos Engineering in general, since it is important that the development team who is to start using Chaos Engineering is aware of what the discipline consists of and what its benefits and challenges are before they get started. Resources to facilitate this have been gathered in the first support document, named Chaos Engineering Resources. These resources include this report, a description of the framework activities, all support documents and links to Chaos Engineering literature. Since Microsoft Office is widely used at ICA, it was decided to give the first support document the physical form of a SharePoint site. SharePoint is a web-based Microsoft Office application for connecting teams with content and information. It can also be integrated with Microsoft Teams, another Microsoft Office application which is heavily used at ICA.

1.2 Define the application under test. This step consists of defining the application under test in terms of the eight application categories which were defined in Chapter 4 and which are present in the tool map. The second support document, Application Categories, lists and describes these eight categories, and it states conditions which have to be fulfilled by the application in order for it to belong to each of the eight categories. For instance, the first condition is: "Some part of the application is containerized with Docker". An application which fulfils this condition belongs to the Docker application category.

1.3 Find relevant experiments. Based on the Tool Map support document and the application categories which the application belongs to, as determined in the previous step, this step consists of finding all the experiments that are relevant to the application under test. In the tool map, the relevant experiments are the ones which are specified in the cells that have columns corresponding to one of the application categories which the application belongs to.

1.4 **Categorize the experiments**. Every experiment found in the previous step is now to be categorized into one of the five categories that are listed and described in the fourth support document, Experiment Categories. The experiments in the first two categories should no longer be considered. The experiments in categories three and four should be added to the improvement opportunities output, in two separate columns: experiments for which the application lacks monitoring solutions and experiments for which the application lacks resilience mechanisms, respectively. Finally, experiments in the fifth category should be added to the experiment backlog. The five categories are:

- **Category 1:** The experiment does not apply technically to the application under test
- **Category 2:** The experiment is not of interest to the team and will probably not bring value to the application under test
- **Category 3:** The experiment will probably affect the application under test in ways it is not currently possible to monitor
- **Category 4:** The experiment should not be handled well by the application under test
- **Category 5:** The experiment should be handled well by the application under test

1.5 **Prioritize**. This step is an optional step which can be performed if the number of experiments in the backlog is considered to be too large after the previous step. The fifth support document, Prioritization Guidelines, presents two ways of prioritizing: one where the experiments are assigned estimations of likelihood and impact, and one where the most valuable experiments are distinguished based on intuition.

After the discovery activity, first versions of two outputs have been generated, namely the improvement opportunities and the experiment backlog. The improvement opportunities output can suggest weak areas in the application under test, in terms of the way it is monitored as well as the way it attains its level of resilience. The experiments listed in this output, in two separate columns, should not be run for the application under test as it currently is. The opposite can be said for the experiments in the experiment backlog; as the name suggests, all experiments listed in this document should, when performed, bring value to the application when it comes to testing its resilience.

To prepare for the following activities, a third column should be added to the improvement opportunities output, for experiments which have been run but resulted in a disproved hypothesis. Such a column will be needed already for the framework's second activity, implementation.

The framework has been designed with loops which intend to repeat the implementation, sophistication and expansion activities regularly. However, unlike these activities, the discovery activity should only have to be performed once. Only if the infrastructure of the application under test changes, so that the set of application categories the application belongs to changes, does the discovery activity need to be repeated.

5.2 Activity 2: Implementation

Adhering to the Chaos Engineering principle of starting small, it is not advisable to implement all experiments in the backlog simultaneously. Therefore, the *implementation* activity consists of setting up and running only one Chaos Experiment. The implementation activity should be performed every a time a new Chaos Engineering tool is about to be used for the application under test. The first time the implementation activity is performed, no tools or experiments should be in place for the application under test. When the implementation activity is revisited, any number of tools and experiments may be in place, depending on the number of previous iterations and the number of experiments which previously implemented tools are used for.

The implementation activity is illustrated in Figure 5.3. Users of the framework should know that their choices made in steps 2.2 and 2.3 of the implementation activity do not need to be final, because the third activity of the framework, sophistication, will give them a chance to adjust their choices in order to improve their Chaos Engineering practice gradually. The most important part of the implementation activity is to just get started with a new tool and to learn how to use it manually.

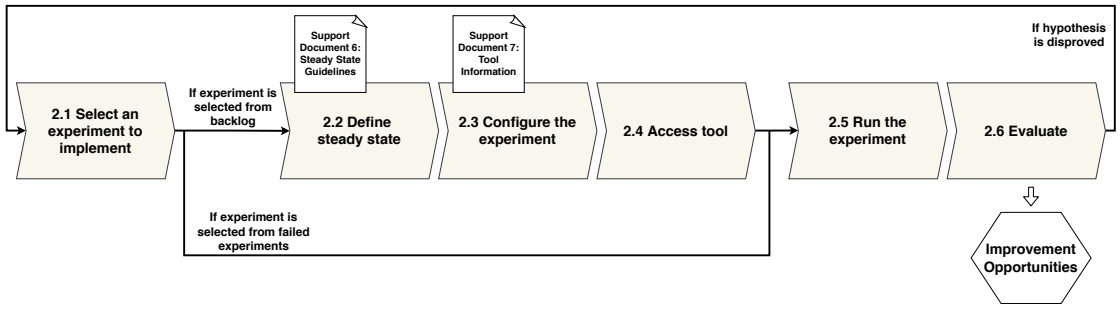


Figure 5.3: The second activity of the framework: implementation.

As illustrated in Figure 5.3, it is not recommended to move on from the implementation activity unless it ends with a successful experiment where the hypothesis is proved to be correct. If the hypothesis is instead disproved, the implementation activity should be restarted, either with a new experiment or with the same experiment. This is described more thoroughly below.

The steps that are recommended to follow during the implementation activity are:

2.1 Select an experiment to implement. If the implementation activity is performed for the first time, it is only possible to select experiments from the experiment backlog. Otherwise, there is a chance that experiments can be selected from the third column of the improvement opportunities output as well, which contains previously executed experiments where the hypothesis was disproved, since such experiments would benefit from being run again. If the experiment is selected from the backlog, the following should be considered:

- The backlog can contain experiments for which tools that have not yet been introduced at ICA need to be used. Already introduced tools should always be considered first. If this is impossible, it might also be impossible to continue with the next steps and activities until more tools have been introduced.

- If a prioritization was made during the discovery activity, one of the highest prioritized experiments can be selected.
- If a prioritization was not made during the discovery activity, the experiment which seems most simple is a good first choice.

If the experiment is selected from the improvement opportunities instead of the backlog, the following should be considered:

- An improvement in the application's resilience mechanisms must have been made since the experiment was executed the last time. Otherwise, it is pointless to repeat the experiment.
- Since all necessary experiment configurations should have been made the last time the experiment was executed, steps 2.2, 2.3 and 2.4 of the implementation activity can be skipped.

2.2 Define steady state. Defining steady state consists of characterizing the application's normal behavior in a measurable way. To define steady state, the sixth support document, Steady State Guidelines, instructs the development team to answer the following questions:

- Which part(s) of the application do we need to monitor to know how the application under test behaves during the experiment?
- Which metrics characterize the application's behavior in this part/these parts of the application?
- For these metrics, what are the acceptable values? "Acceptable values" here refers to the values of the metrics which represent the application's normal behavior.
- How can these metrics be monitored and logged during the experiment — with a tool or manually?

The Steady State Guidelines support document also states the Chaos Engineering principle that it is preferable to define the steady state in terms of both technical metrics and business metrics. When this step has been performed, the experiment's hypothesis can be phrased as: "Running the experiment <Selected experiment> will not cause the application's behavior to change from <Steady state definition>".

2.3 Configure the experiment. In this step, there are three choices which need to be made, regarding the tool to use for the experiment, the environment of the experiment, and the experiment's blast radius. When it comes to the *tool* to use, there will be either one, two or three tools which can be used for the selected experiment. One of them has to be selected, obviously giving precedence to tools which have been introduced at ICA. The seventh support document, Tool Information, presents information on all tools in the tool map and how to configure them, taken from their documentation or website. Links to the each tool's documentation and website are also given, along with links to their repositories. When it comes to the experiment's *environment*, it is advised to set up some sort of safe testing environment and not experiment in production from the start. Finally, the *blast radius* should according to the principles of Chaos Engineering be minimized when it comes to ICA's business. Deciding how to not end up with devastating consequences is therefore part of configuring the experiment.

2.4 **Access tool**. How to get access to the tool to use for the selected experiment will depend on how it is decided to introduce the tool at ICA. This is not part of this thesis's scope. See section 5.6 for more information on this.

2.5 **Run the experiment**. When access to the tool has been granted, the next step is to execute the experiment according to the configurations made in step 2.3 and the hypothesis defined in step 2.2. The experiment should be run manually to learn how the tool works.

2.6 **Evaluate**. When the experiment has been run, the next step consists of either proving or disproving the hypothesis based on the experiment's results. If the hypothesis is disproved, the experiment should be added to the third column of the improvement opportunities output, and an improvement of the application's resilience mechanisms should be planned. Then, the implementation activity should be restarted. If the hypothesis is proved, this should be marked in the backlog. Then it is possible to move on to the sophistication activity. Note that if it is concluded that the hypothesis can be proved based on the experiment's results, performing the implementation activity has verified that at least one of the application's resilience mechanisms works as intended.

5.3 Activity 3: Sophistication

The name of the *sophistication* activity is inspired by the Chaos Maturity Model, or the ChMM, which was introduced in section 2.6. To recapitulate, the ChMM is a model for mapping how extensively Chaos Engineering is implemented in an organization and it aims to provide insights into how to improve such an implementation. The model's two metrics are sophistication and adoption, where sophistication refers to the validity and safety of the Chaos Experiments and adoption refers to their sanctioning and reach in the organization. From the perspective of a single development team, the most relevant aspect to consider for improvement is therefore the experiments' sophistication. Sophisticating the team's current Chaos Engineering practice is also the aim of the third activity of the framework, illustrated in Figure 5.4.

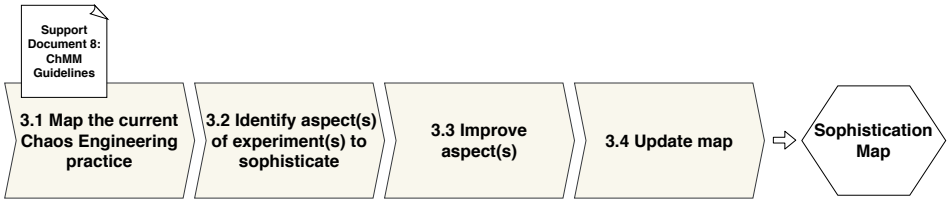


Figure 5.4: The third activity of the framework: sophistication.

The steps that are recommended to follow during the sophistication activity are:

3.1 **Map the current Chaos Engineering practice**. Inspired by the sophistication metric of the ChMM, described in the eight support document which is named ChMM Guidelines, the following three aspects should be considered for every experiment which is currently in use for the application under test:

- Experiment environment. Is the experiment (1) not run in production, (2) run with production-like traffic or (3) run in production?
- Automation. Is the experiment (1) run and monitored manually, (2) either run or monitored automatically, or (3) both run and monitored automatically?
- Metrics. Is the steady state defined by (1) technical metrics only, (2) technical metrics and some aggregated business metrics, or (3) technical metrics and disaggregated business metrics?

Each experiment should, when this step has been performed, be mapped to three values between 1 and 3 in a sophistication map output.

3.2 Identify aspect(s) of experiment(s) to sophisticate. Aspects answered with (1) in the previous step should be prioritized for sophistication over aspects answered with (2), which, in turn, should be prioritized over aspects answered with (3). By similar logic, if several experiments are implemented for the application under test, experiments with many 1's should be prioritized to sophisticate over experiments with many 2's and 3's.

3.3 Improve aspect(s). How to sophisticate the experiments' environment, degree of automation and metrics will be highly dependent on the application under test and thus on the development team's knowledge of the application.

3.4 Update map. Before moving on from the activity, the 1's and 2's which were sophisticated in the previous step should be updated to 2's or 3's in the sophistication map, to ensure that it is updated the next time the activity is performed.

5.4 Activity 4: Expansion

The fourth activity, *expansion*, adds the principle of increasing a Chaos Engineering implementation incrementally to the framework. The activity is based on the assumption that it is always easier to expand the current usage of a Chaos Engineering tool to include more of its functionality than it is to start using a completely new tool. Only if the already implemented tools have been totally exhausted in terms of the experiments they can be used for, or if it for some reason is prioritized to implement a Chaos Experiment which the already implemented tools cannot provide, should a new tool be considered for the application under test. The expansion activity is illustrated in Figure 5.5.

The steps that are recommended to follow during the expansion activity are:

4.1 Find experiments possible to run with tools that are already in use. Experiments can be found in the experiment backlog or in the third column of the improvement opportunities output, for experiments which resulted in a disproved hypothesis. If no experiments are found, or if priority is given to other experiments which the tools that are in use cannot perform, the rest of the expansion activity is not relevant. Instead, the implementation activity should be revisited to find a new tool to use.

4.2 Select one of the experiments. If the experiment is selected from the improvement opportunities, step 4.3 can be skipped.

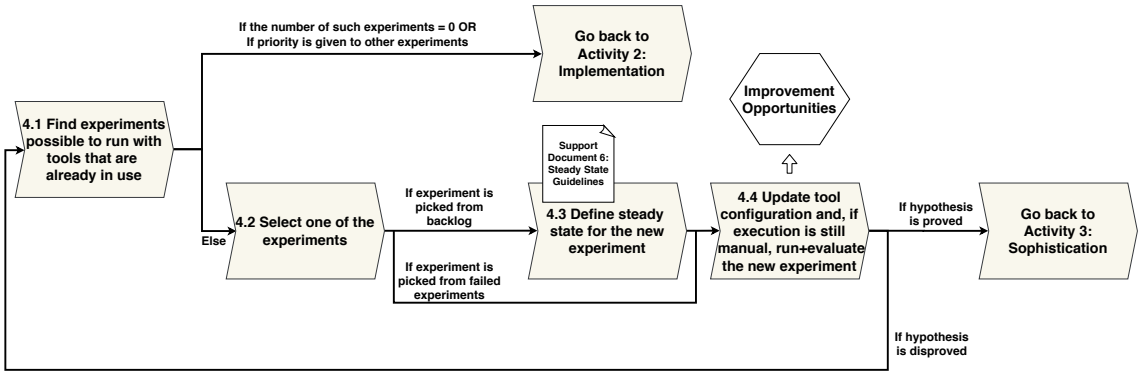


Figure 5.5: The fourth activity of the framework: expansion.

4.3 Define steady state for the new experiment. This step is the same as step 2.2 of the implementation activity and the same support document, Steady State Guidelines, can be used for it.

4.4 Update tool configuration and, if execution is still manual, run+evaluate the new experiment. If the usage of the tool in question has been automated previously and there is a wish to not add a non-automated usage of it, this step consists of updating the tool configuration to include the new experiment. If the usage is still manual since the tool was first used during the implementation activity, this step merely consists of repeating steps 2.3, 2.5 and 2.6 of the implementation activity for the new experiment. When the step has been performed, the next action depends on the outcome of the experiment. If the hypothesis is proved, the experiment should be marked as implemented in the backlog before the sophistication activity is revisited. If the hypothesis is disproved, the experiment should be added to the improvement opportunities output's third column for experiments which resulted in a disproved hypothesis, an improvement of the application's resilience mechanisms should be planned and the expansion activity should be restarted.

5.5 The Full Framework

When combining the four activities described in sections 5.1 through 5.4, the suggested way for a development team at ICA to work with Chaos Engineering to test their application can be illustrated as in Figure 5.6. Essentially, the framework becomes two loops: one outer loop and one inner loop. The outer loop contains the implementation, sophistication and expansion activities, and it introduces a new Chaos Engineering tool, initially for a single experiment. Then, the inner loop repeats the sophistication and expansion activities until the newly introduced tool has been exhausted in terms of the experiments it can be used for. Not until then should the next iteration of the outer loop continue to introduce yet another tool. Apart from these loops, there are times when single activities (more specifically, the implementation and expansion activities) should be repeated or restarted.

By applying the framework, three outputs are produced, namely the experiment backlog, the improvement opportunities and the sophistication map. None of them is meant to be static; instead, they should be updated for as long as Chaos Engineering is in use for the ap-

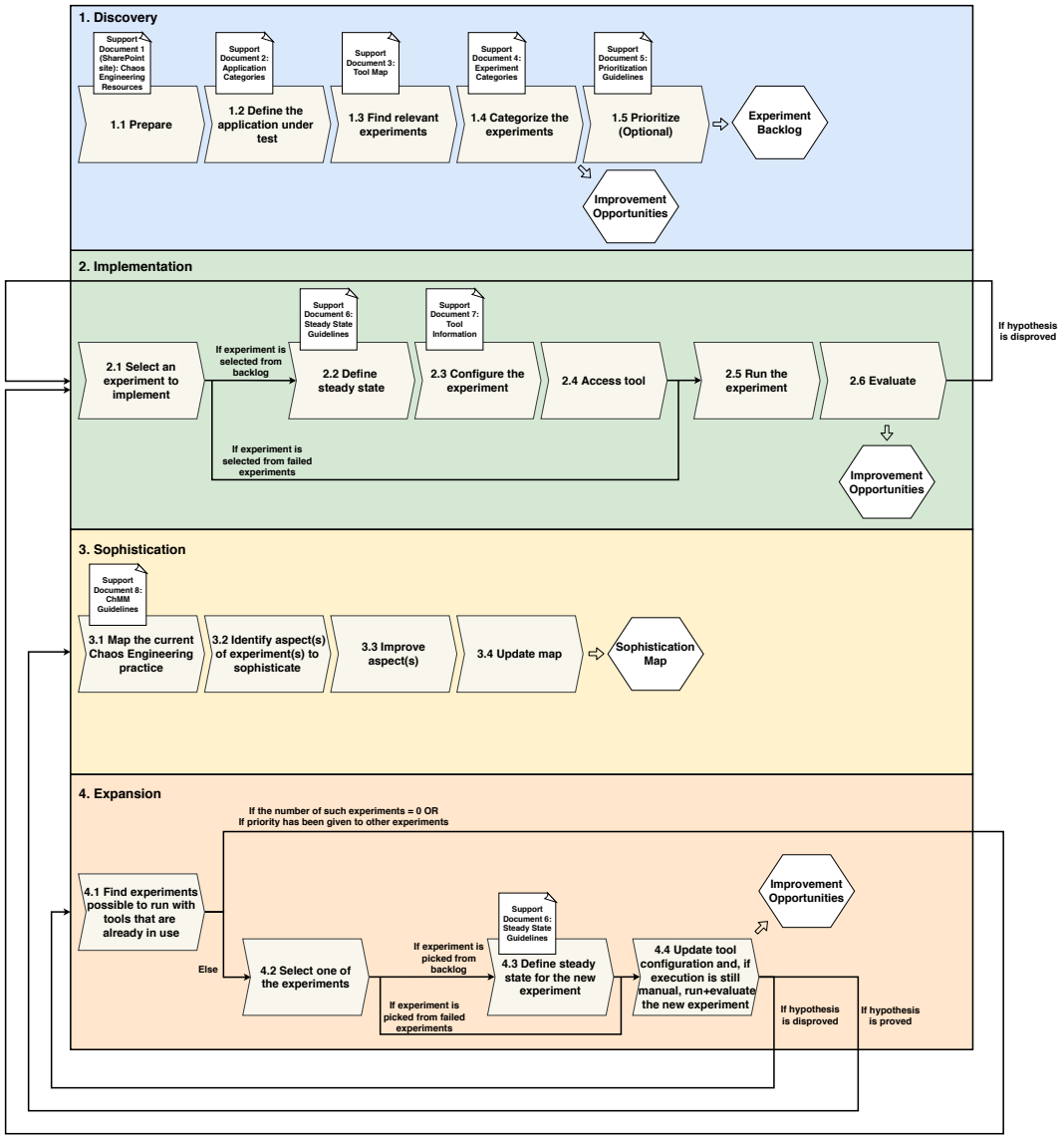


Figure 5.6: All four activities in the framework, combined into the full framework.

plication under test. The *experiment backlog* will be updated whenever new experiments are successfully introduced. It is a way to keep track of the Chaos Engineering implementation for the application under test. The *improvement opportunities* will contain experiments which cannot be executed since the application is not resilient enough and experiments which were executed but failed, also because the application was not resilient enough when the experiment was run. The improvement opportunities output will thus be updated whenever new efforts are made in improving the application's resilience and experiments are rerun. It will also contain parts of the application that are not currently being monitored in a satisfying way, which will be updated if new monitoring solutions are implemented. Finally, the *sophistication map* will be updated whenever the sophistication activity leads to an improved sophistication for one or several experiments. It can be seen as an overview of how mature the current Chaos Engineering practice is.

Exactly how the activities of the framework are performed is up to every user; they can

use whatever tool or solution they feel comfortable with to keep track of their progress. The framework's support documents and a description of the framework itself should be enough to accomplish this. However, a suggestion is to use the Excel file available at the SharePoint site which constitutes the Chaos Engineering Resources support document. In this file, Visual Basic for Applications (VBA) has been used to automate some of the steps, which should make the process of applying the framework faster and less labor-intensive. The framework's support documents have also been included in this Excel file.

5.6 Introducing Tools at ICA

The activities of the framework are based on the following fact: introducing new tools at large companies, such as ICA, is not always trivial. It is a process which can affect multiple teams in the organization, and exactly how the process is carried out at ICA depends on a variety of factors, some of which are described in this section. Because of this, it is not feasible to introduce all twelve tools in the framework's tool map at the exact same time, meaning that when a specific development team wants to apply the framework to their application, there is a high probability that only a few of the twelve tools are ready to use at ICA. Therefore, the framework is based on the recommendation to give precedence to already introduced tools before other tools are considered. This recommendation mainly affects the implementation and expansion activities of the framework, described in sections 5.2 and 5.4, respectively.

There exists a rather standardized process for introducing new tools at ICA. This process contains the following conceptual steps, here paraphrased rather than stated exactly:

1. Proof of Concept (PoC). Test the tool together with a development team, to explore the value of introducing it. If it is found to be of some value, continue; otherwise, there is no need to.

2. Find a team at ICA to be responsible for the tool. Determine which resources they would need to be able to manage the tool.

3. Calculate the tool's total costs. The total costs for a general tool include costs for purchasing, licensing, management and the necessary infrastructure.

4. Decide if the tool should be provided as a bought service or as a self-service. It was mentioned in section 5.2 that how to access the tool depends on how the tool is introduced, and this mainly referred to this step of the introduction process. Bought services are accessed by placing an order in HelpICA, ICA's system for ordering resources of different kinds. Self-service tools are administered by ICA's center for continuous delivery.

5. Package the service. This includes defining the tool, how to use it, how to access it and how to receive help if problems arise when using the tool.

6. Go through ICA's handover process to transfer the responsibility for the tool to the team found in step 2.

7. Find possible users of the tool and perform demonstrations to roll out the tool in the organization. This should start small and then increase incrementally.

New tools also need to be evaluated from a security and risk perspective before they are introduced. This process can sometimes be long and complicated, perhaps particularly if the tools in question are open source. The main reason for this is that ICA lacks specific security guidelines when it comes to open source tools, meaning that proposed tools need to be evaluated individually, on a case-to-case basis, before they can be approved and subsequently introduced.

ICA's process for introducing new tools mostly lies outside of the scope of this project, but it is, however, recommended to introduce tool T3, Chaos Toolkit, at ICA first. T3 is the widest tool in in terms of the number of application categories it belongs to, belonging to three categories while the other tools belong to only one or, in the case of T21, two. Because of this, it can be assumed that out of the twelve tools, T3 should be the one which can be used for the largest number of ICA applications. No recommendations are made here regarding which tools to consider after T3, but a reasonable approach could be to interview users of the framework at a later point in time and base the decision on how they would like to expand their current Chaos Engineering practice. By recommending to start with T3, however, the idea is that all development teams interested in Chaos Engineering can get started with the same tool, and while they loop through the sophistication and expansion activities with T3, the process of introducing more of the tool map's tools can be given time to progress.

Chapter 6
Framework Evaluation

The validation of the framework was two-fold: the framework was applied iteratively to sample applications at ICA as it was built, and the finished version of it was used by stakeholders at ICA during an evaluation exercise. The findings from applying the framework are presented in section 6.1 and the findings from the final evaluation exercise are presented in section 6.2. Section 6.3 presents the results of the questionnaire in Appendix C regarding how employees at ICA perceive the need for Chaos Engineering. Finally, section 6.4 presents findings from the interviews which were conducted during the Why Chaos Engineering interview theme.

6.1 Applying the Framework

As described in subsection 3.1.5, only the discovery activity and the first step of the implementation activity were applied during this project. The activities were applied to `ica.se` and the back-end of ICA's e-commerce at separate meetings with participants from the applications' development teams. The participants were observed during this process to draw qualitative, but subjective, conclusions regarding the usability of the activities, as described in subsection 3.1.5. The participants were also encouraged to give oral feedback on how they perceived the activities. It was found that applying the discovery activity was helpful in two ways: it initially made adjustments and additions to the activity and its support documents, and the last time the activity was applied can be seen as a confirmation that the finished version of the activity is feasible. Examples of the changes applying the framework resulted in include:

- Adding the prioritization step when both participants considered the numbers of experiments in their applications' backlogs to be large (the backlogs contained 19 and 17 experiments, for `ica.se` and the e-commerce respectively)
- Rephrasing the conditions describing which application categories an application be-

longs to, from the form “The application is written in Java” to “Some part of the application is written in Java” as it was realized that the applications were large enough to use several cloud service providers and programming languages

- Adding a category for experiments that are not of interest to the development team to the Experiment Categories support document as the need for this was realized when the participants categorized the experiments

The examples of findings above show that the initial versions of the activity needed to be tailored to fit ICA’s applications. After that, applying the framework for the third and last time successfully generated experiment backlogs for both `ica.se` and the back-end of the e-commerce as well as nine improvement opportunities for the back-end of the e-commerce, corresponding to nine experiments which the participant from ICA’s e-commerce believed would be problematic for the application to handle. When it comes to improvement opportunities for `ica.se`, however, this output was blank.

Noteworthy is that during the fourth step of the discovery activity, where relevant experiments are to be categorized into one of five experiment categories, both applications had relatively large numbers of experiment to categorize (33 and 59 experiments, respectively). However, the participants were still able to categorize the experiments quickly and needed only a few seconds per experiment to decide which category it should belong to.

The full framework consists of nineteen steps, three outputs and eight support documents, and so the activity of applying the framework managed to test and thus validate approximately one-third of all steps, two-thirds of all outputs and five-eights of all support documents. The rest have not yet been evaluated on any ICA application.

6.2 Evaluation Exercise

At the evaluation exercise, a total of 17 people participated. They all work as process managers or as managers to process managers, except for four of them who have other managerial roles in related parts of the organisation. In other words, none of the participants have work tasks related directly to software development or software testing; instead, the process managers work to develop and maintain processes which are to be followed by the rest of the company, for instance regarding changes in the software, handovers between different parts of the organization and requests for new functionality. They were thus considered to be suitable participants when it comes to the logic, structure and accessibility of the framework, but it was understood that they perhaps would have less to say regarding the tools and Chaos Engineering principles which were included in the framework.

As mentioned in subsection 3.1.6, the exercise could not take place as planned and it was held over a video call instead. During the exercise, none of the attending process managers found any missing or redundant parts of the proposed Chaos Engineering framework, and all pieces of feedback received during the exercise agreed that the proposed framework seemed feasible. None of the participants sent in any additional feedback after the meeting.

6.3 Employee Attitudes Towards Chaos Engineering

Employees in ICA's IT department seem to have a generally positive attitude towards Chaos Engineering, both in general and applied at ICA specifically. After getting an introduction to Chaos Engineering, and in most cases also a demonstration of how Chaos Engineering can be implemented using the Chaos Monkey for Spring Boot tool, 23 people who work with IT at ICA were asked to state their level of agreement with two statements, ranging from 1 (I do not agree in the slightest) to 5 (I completely agree). The two statements were: "I see the benefits of Chaos Engineering in general" and "I see the benefits of Chaos Engineering somewhere at ICA". Figure 6.1 gives the distribution of the answers, showing a clearly positive ranking of the perceived benefits of Chaos Engineering. The average answers were 4.65 and 4.57, respectively, and 74 % and 65 % of the respondents answered 5 on the two questions, respectively.

A clear majority of the respondents (74 %) saw the same need for Chaos Engineering at ICA as at other companies. Some respondents (17 %) saw that the need for Chaos Engineering might not be as extensive at ICA as at other companies. Others (9 %) disagreed and saw more value of Chaos Engineering at ICA specifically than at an arbitrary company. This was, in some cases, motivated by the high degree of complexity in ICA's applications; that was, according to some respondents, why Chaos Engineering was specifically needed at ICA rather than at other companies.

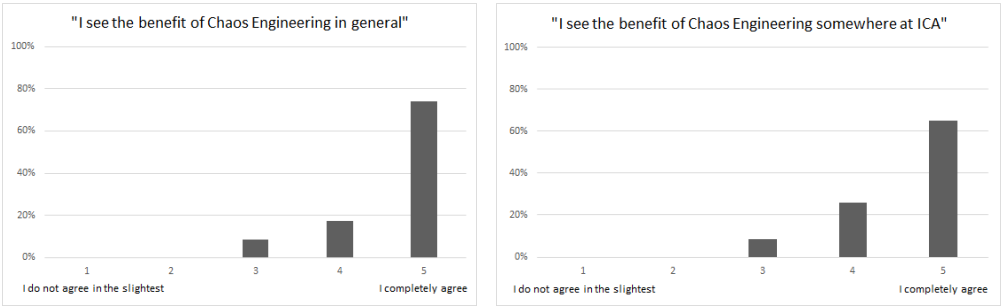


Figure 6.1: Distribution of how IT employees at ICA perceive the benefits of Chaos Engineering.

6.4 Employee Attitudes Towards Incidents

The obvious reason for implementing Chaos Engineering is that it aims to improve a software application's resilience, which has the potential of reducing the application's number of incidents. The interviews which were conducted during the Why Chaos Engineering interview theme revealed several examples of why the interviewees considered this to be of importance to ICA. These examples can be grouped into the following qualitative categories:

- **Economic reasons**. Some incidents are critical enough to severely hinder ICA's business and lead to not only decreased incomes but also increased costs, for instance if they

cause stores and warehouses to malfunction or if they make ICA's e-shop go down. It is also perceived that such losses are difficult to recover; in grocery retail, customers can be assumed to often choose a competitor rather than wait if they for some reason cannot purchase their groceries.

- **Branding reasons**. Severe incidents can damage how ICA is perceived as a company by media and the general public.
- **Loyalty reasons**. Incidents which affect ICA's end customers can cause frustration and thus a tendency for the customers to favour competitors over ICA. Frustration can, naturally, also be experienced by the store owners who so heavily rely on ICA.
- **Legal reasons**. Since ICA contains a segment for banking operations, it answers to the Swedish Financial Supervisory Authority. This supervising organization has legal reliability requirements which ICA is obliged to fulfil.
- **Psychological reasons**. Incidents can cause stress and dramatically increase workloads for employees. Frequent incidents can also decrease general employee motivation, and long-lasting incidents come with problematic matters of importance such as appropriately switching off employees and adhering to the Swedish Working Hours Act.
- **Competitive reasons**. The employees who handle the incidents have other tasks during the incident-free times. This means that when major incidents occur, the general development at ICA will halter and slow down, which by extension can delay releases and thus introduce a risk of falling behind competitors.
- **Political reasons**. Stakeholders in the ICA IT department find it a possibility that the IT department will be outsourced if a sufficient quality cannot be maintained in the IT services that are delivered. There is thus a perceived need to motivate management why the current structure should be trusted to continue with their operations, for instance by minimizing the number of incidents.

During several of the interviews there were explicit mentions of a general desire to go from being reactive to being proactive when it comes to handling incidents. Chaos Engineering was numerous times mentioned as a good step in this direction.

Chapter 7
Discussion and Conclusions

This chapter discusses and concludes the results presented in this report. As mentioned in the introduction of the report, the project had the purpose of examining how the resilience of the software systems developed at ICA could be improved by using tools and techniques for Chaos Engineering. In line with this, the report aimed to gather how principles for Chaos Engineering are best implemented at ICA in a so-called framework, which can be used by separate development teams in the organisation to implement Chaos Engineering for the software applications they develop. The framework proposes a recommended way to work with Chaos Engineering that is continuous and also extensible, which makes it possible for development teams to start small and then increase their practices gradually. The way of working is based on a total of twelve open source Chaos Engineering tools, which can be used for different purposes since the tools differ in terms of functionality and which applications they can be used to test. As presented in Chapter 5, the framework consists of four activities to perform to start using Chaos Engineering: *discovery*, which defines a backlog of Chaos Experiments suitable for a specific application as well as initial improvement opportunities for the application's resilience and monitoring, *implementation*, which helps to implement one experiment from the backlog using one of the twelve tools of the framework, *sophistication*, which matures already implemented experiments, and finally *expansion*, which adds more experiments to the current Chaos Engineering practice. It is recommended to regularly repeat all activities except discovery; implementation should be repeated when a new tool is about to be used for the application under test, and sophistication should be repeated between every pair of experiments that is added when the expansion activity is performed. The way of working is only a suggestion and has been produced as a sort of lighthouse project, aiming to introduce the discipline of Chaos Engineering in the organization. There are, naturally, other ways to work with Chaos Engineering as well, and the results presented in this report should only be viewed as recommendations.

This chapter starts with section 7.1 which contains a discussion on this project's contribution in relation to previous and related work. Section 7.2 discusses the validation activities performed during the project, before section 7.3 reflects further on related work and presents

an evaluation of the information they contain. Section 7.4 states some recommendations for future work when it comes to implementing Chaos Engineering at ICA, before answers to the thesis's research questions are summarized in section 7.5.

7.1 The Framework's Contribution

Related work on Chaos Engineering, some of which is summarized in Chapter 2 of this report, presents several principles of Chaos Engineering, such as minimizing the blast radius of Chaos Experiments and experimenting directly on production traffic. Related work also gives information on how Chaos Experiments can be carried out in four steps: defining steady state, hypothesizing about steady state, simulating real-world events and finally, proving or disproving the hypothesis. What the framework proposed in this report contributes is a way to combine these principles and Chaos Experiments into a more long-term way of working with Chaos Engineering that is tailored to suit the operations of a specific company. The framework has its focus on a higher level than single Chaos Experiments, and it can answer questions such as *which* experiments to perform, in which *order* they should be performed, and how and when they can be *improved*.

The benefits of gathering recommendations for practice in a framework like the one proposed in this report may for instance include a more simple way for separate development teams to get started with Chaos Engineering even if they have no prior experience in the discipline. It also forms a common base for discussion and reference, and can thus enable people from different parts of the company, for instance different development teams, to speak the same language regarding the discipline. By establishing a common way of working, an opportunity is furthermore created to maintain and evolve a quality in the way the discipline is implemented throughout the organization, which perhaps can decrease the number of misinterpretations of what the discipline consists of and how it should be implemented. In addition to this, the activities of the framework were designed with the aim of fulfilling as many of the Chaos Engineering principles found in related work as possible. For instance, the discovery activity ensures that the overall principle of only running experiments which the application under test should handle is followed, and it also introduces the principle of prioritizing Chaos Experiments based on likelihood and impact. The implementation activity gives guidelines on how to define steady state with both technical metrics and business metrics at the system boundary, and instructs framework users to consider the experiments' blast radius when configuring them. The way the implementation, sophistication and expansion activities are recommended to be repeated enables teams to adhere to the principle of starting small and increasing incrementally. The sophistication activity adds the Chaos Maturity Model to the framework, and it also helps teams to constantly aim to automate experiments and run them in the application's production environment. Finally, if the steady state was not defined with both technical metrics and business metrics in the implementation activity, the sophistication activity also helps development teams to correct this. Without a framework, it is plausible that at least some of these principles would have been missed by teams who unaided had to implement Chaos Engineering for their application.

When it comes to Chaos Engineering tools, the results presented in Chapter 4 can be useful as an overview of some of the currently available open source tools. The tool map in Figure 4.6 answers questions such as which problems it is possible to simulate using Chaos

Engineering and which applications it is possible to simulate these problems for. It can in other words concretize what the discipline actually consists of, from an operational and practical point of view.

The interview results presented in section 6.4 in the previous chapter suggest that any discipline or practice which is successful in reducing the number of incidents in ICA's software system would be welcomed by ICA employees. Furthermore, the introductory chapter of this report mentioned that ICA has a long-term goal of completely eliminating critical incidents in ICA's applications while at the same time maintaining a fast software development, and a desire to go from being reactive to being proactive could be seen as a part of this goal. As explicitly mentioned in several of the interviews, Chaos Engineering is positively associated with this change, and the questionnaire results presented in section 6.3 seem to confirm that Chaos Engineering is perceived as valuable at ICA. These results can, naturally, be considered to be promising when it comes to future work with Chaos Engineering at ICA. For instance, as mentioned in section 5.6, new tools or concepts are at ICA tried in real development teams to investigate the value of them before they are introduced; perhaps a positive employee attitude can help this process progress after the end of this master's thesis project.

7.2 Validation of the Framework

As described in section 6.1, applying progressive versions of the discovery activity three times resulted in experiment backlogs for both `ica.se` and ICA's e-commerce but improvement opportunities for only the e-commerce. The lack of improvement opportunities for `ica.se` can be interpreted in different ways. It can for instance mean that the participant from `ica.se` overrated the website's resilience or simply believed that no experiment in the framework would be able to challenge it, but it can also mean that the framework is unable to generate improvement opportunities for some types of applications. Why the output was blank will remain unanswered in this report, but the fact that the framework found some improvement opportunities for the back-end of the e-commerce (in the form of nine experiments for which the application lacks resilience mechanisms) is interpreted as an indication that improvement opportunities can be found for at least some software applications when the activity is performed.

Also as described in section 6.1, it was found that the process of categorizing experiments into the five categories in the Experiment Categories support document could be finished quickly for both applications. This could indicate that the total number of experiments presented in the tool map, 87, is not large enough to make the proposed activities impossible or impractical to perform. Even if an application belongs to all eight application categories, meaning that all 87 experiments have to be categorized, it should be possible to finish this process in a matter of minutes.

The main conclusions drawn from the activity of applying the framework were thus that the five steps of the discovery activity and the first step of the implementation activity are feasible, that they can generate outputs successfully for at least some software applications and that their five support documents are possible to use. Regarding the rest of the activities and their outputs and support documents, no validating conclusions can be drawn.

When it comes to the evaluation exercise, it seems highly plausible that holding the exercise digitally rather than physically negatively affected the amount of feedback the exercise

was able to generate, since none of the participants were able to perform the activities of the framework themselves. With that said, the clear majority of the participants work as process managers and can thus be assumed to be familiar with what is required in order for a proposed process to be feasible. It is therefore arguable that at least one of them would have been able to spot an obvious mistake in the logic behind a process when it is presented to them, and in this case, none of them saw any lacking or redundant parts in the proposed framework, regarding neither the steps of the activities nor their support documents and outputs. This can be seen as an indication that the framework's logic will work as an ICA process. However, due to the way the exercise had to be carried out, no further interpretations of its results will be made.

7.3 Information Evaluation

This section presents some aspects of source criticism regarding the related work on Chaos Engineering which this report builds on. Note that while it is always appropriate to be critical when reading and using sources of information, it was decided to trust all pieces of related work used in this report, at least when it comes to the specific way in which they were used. This section merely presents some possible objections to this, objections which were taken into account during the project.

Firstly, many resources treat Chaos Engineering from the perspective of Netflix, a sort of pioneer when it comes to the discipline since Netflix is where it originated around ten years ago. Aspects to take into account include whether or not their implementation of Chaos Engineering is specific to just their organization, or if the results can be used from the perspective of other organizations as well. Some pieces of related work explicitly argue for the latter, and to quote one of them: "While the tools that we have written may be specific to Netflix's environment, we believe the principles are widely applicable to other contexts" [7]. It was decided to agree with this point of view. However, when reading these pieces of literature, it should be taken into account that Netflix can differ from other applications, for instance when it comes to their peak times in relation to common business hours.

Secondly, during the tool evaluation, information from each tool's website and documentation was heavily used, in other words information written by the tool developers. An obvious source of criticism is therefore that these pieces of information can be biased; it should arguably lie in the best interest of the developers to highlight the most positive aspects of their own tools. However, since the tool evaluation was performed with a focus on functionality and areas of use, in other words rather quantitative pieces of information, the information was still deemed reliable enough to use.

Thirdly, not only books, journal articles and similar types of resources were used, but also a few blog posts and news posts. However, they were not used in critical ways which affect the results; instead, they were used as examples or to give some piece of trivia aiming to make the report more readable. Therefore, they were still deemed possible to use.

7.4 Recommendations for Future Work

As this project was conducted as a master's thesis project, it was dimensioned to be finished within 20 weeks. This limited the project in several ways and this section presents some possible starting points for future work, continuing where this project left off.

As described in section 3.2, this project only considered open source tools during the tool evaluation. A future project may benefit from examining other types of tools as well. Also, as mentioned in section 4.1, a total of 27 tools were considered during the tool evaluation, and it was decided to conclude the search for additional open source tools when this number had been reached in order to limit the scope of the project. In a future study, additional open source Chaos Engineering tools might be discovered — perhaps tools which were missed during this project, or tools which have been developed since — and it is possible that they overlap the 27 examined tools and that they thus can have the potential of replacing any of the twelve tools which were ultimately selected for the framework. They could also add new application and functionality categories to the tool map.

In section 4.2, irrelevant application categories were defined to be GCP, Cloud Foundry, DC/OS, Spinnaker and Go. However, the consulted IT architect believed it a possibility that some development teams at ICA might use Cloud Foundry, thus introducing a risk for an incorrect tool elimination. In a future project, the need for Chaos Engineering in Cloud Foundry environments at ICA could be investigated further. Also, the IT architect pointed out that while no ICA applications were written in Go at the time of the project, there had been talk of starting to use it as a programming language sometime in the future. If this happens, Chaos Engineering tools for testing Go applications could become useful, indicating that this project's definition of irrelevant categories should be reconsidered. It is furthermore noteworthy that only one IT architect was available for consultation regarding these matters, and there is thus of course a possibility that his colleagues would disagree with the recommendations he made during the project.

Also, it was only possible to validate the discovery activity and the first step of the implementation activity during the project. It may be possible for the activities of the framework to act as a foundation for a formal ICA process for Chaos Engineering in the future, but before that, the rest of the activities should be evaluated on some sample ICA applications as well. This work could be carried out for instance by one of the company's process managers, who could act as a functional owner of the discipline. He or she could then develop the activities into a process and integrate it with necessary parts of the organization.

Finally, when it comes to introducing Chaos Engineering tools at ICA, it was only recommended to start the process described in section 5.6 for the tool Chaos Toolkit. Therefore, future projects need to actually initialize the process of introducing Chaos Toolkit, answering questions such as who should be responsible for it and how the tool best is provided to development teams, and they also need to consider the other eleven tools which were selected for the framework.

7.5 Answering the Research Questions

This section summarizes the project's findings by giving answers to the three research questions defined in section 1.2.

7.5.1 RQ1

Question: How can Chaos Engineering improve the resilience of software systems at ICA?

Answer: By following the way of working proposed in terms of the four activities in Chapter 5, the following sets of findings can be defined:

1. Ideas for yet unimplemented resilience mechanisms. As illustrated in Figure 5.2, these improvement opportunities come from the discovery activity, when experiments are categorized as experiments for which the application lacks resilience mechanisms.

2. Ideas for how to improve monitoring solutions. As the previous set of findings, these improvement opportunities also come from the discovery activity when experiments are categorized, but when they are categorized as experiments for which the application lacks monitoring solutions.

3. Problems in already implemented resilience mechanisms. These are found whenever experiments result in disproved hypotheses.

4. Verification of working resilience mechanisms. These are generated whenever experiments result in proved hypotheses.

The first set of findings can improve resilience by giving software engineers at ICA suggestions of resilience mechanisms that might have been missed or unheard of when developing the application under test. The third set of findings is a result of carrying out a Chaos Experiment that failed; since a prerequisite for carrying out a Chaos Experiment is that it is believed that the application under test should be able to withstand the performed Chaos Engineering attack, a failed experiment should always come as a surprise and thus give a new improvement opportunity. The second and fourth set of findings will not improve software resilience on their own. However, the second set of findings can increase the number of Chaos Experiments which can be run, which can improve resilience by extension, and the fourth set of findings can verify that the software already has a certain degree of resilience.

Apart from this, adopting Chaos Engineering in an organization can come with general benefits which also can improve software resilience, but in less tangible ways. A selection of such advantages are discussed in subsection 2.7.1.

7.5.2 RQ2

Question: What are the necessary building blocks of a Chaos Engineering framework, and how can they be implemented?

Answer: The proposed framework consists of four activities, divided into steps to follow, support documents to use and outputs to produce. The activities provide support to *discover* appropriate Chaos Experiments for a specific application, to *implement* such experiments by using Chaos Engineering tools, to *sophisticate* the current Chaos Engineering practice to make it more valid, and to *expand* the current Chaos Engineering practice to make it wider in terms of number of experiments. In total, the four activities comprise a number of nineteen steps, eight support documents and three outputs.

All of the framework's support documents and a description of the framework itself have been gathered as digital resources on a SharePoint site. This is considered to be sufficient to follow the proposed way of working in any desired way. As an additional resource and a suggested implementation of the framework, an Excel file was created and given functionality using Visual Basic for Applications, and this file was also added to the SharePoint site. It illustrates one way to implement the framework and has automated some of the activities' steps.

7.5.3 RQ3

Question: Where is Chaos Engineering suitable to use at ICA, and can it be provided as a centralized service?

Answer: The answer this thesis can give to where Chaos Engineering is suitable to use at ICA is somewhat limited by the project's delimitations. Assuming that the usage of a tool is necessary in order for Chaos Engineering to be suitable, and considering only the open source tools which were considered during this project, the application under test needs to fulfil at least one of the following conditions to be testable with Chaos Engineering:

- It has to be containerized with Docker
- It has to be managed with Kubernetes or OpenShift
- It has to be running on AWS or Azure, or in a private cloud environment
- It has to be written in Java
- It has to communicate with other services over the Internet

This is a necessary condition but not a sufficient one. In order for a Chaos Experiment to be executable for the application under test, the application also has to have at least one resilience mechanism which deals with the problem simulated by the experiment, it has to have a way to monitor the effects of the experiment, and there has to be an interest in the application's development team to investigate what would happen to the application if the problem simulated by the experiment happened in a real scenario in the application's production environment. If a future project examines other types of Chaos Engineering tools, or a way to implement Chaos Engineering that is not dependent on using a tool, it is possible that this answer can be widened. A common and general perception, however, is that the only condition an application needs to fulfil in order for some form of Chaos Engineering to be suitable is that it is a distributed application.

When it comes to the topic of providing Chaos Engineering as a centralized service, the answer can be seen from two perspectives: a functional one and a technical one. Functionally, the proposed framework constitutes a common way of working shared by all development teams at ICA. By letting ICA process managers validate the framework's contents, it seems likely that the activities of the framework can be further developed into a formal ICA process which can be used by various development teams. Technically, providing tool support for Chaos Engineering in a centralized way can be done in several ways, for instance as a self-service or a bought service, as discussed in section 5.6. The question as to which way is the

most appropriate one is answered towards the end of ICA's tool introduction process and is thus not part of the scope of this project, but it can be concluded that it should be possible. What to include in the service will depend on the specific tools of the framework; this is also investigated late in ICA's tool introduction process and is thus also outside of the scope of this project.

To conclude, it seems that the discovery activity is feasible at ICA and that it can successfully generate useful outputs for future resilience testing of specific software applications developed in the organization. The implementation, sophistication and expansion activities have not yet been fully validated and no conclusions regarding their feasibility can therefore be drawn. It is, however, considered to be reasonable that all four activities, after some future validation work, can act as a foundation to develop a more formal ICA process for Chaos Engineering.

References

[1] Andrew S. Tanenbaum and Maarten Van Steen. *Distributed Systems: Principles and Paradigms*. Pearson Education, Inc., 2 edition, 2007.

[2] Principles of Chaos Engineering. `http://principlesofchaos.org/?lang=ENcontent`, May 2018. Accessed: 2019-11-04.

[3] ICA Gruppen AB. `https://www.icagruppen.se/`. Accessed: 2019-11-28.

[4] Ali Basiri, Lorin Hochstein, Nora Jones, and Haley Tucker. Automating Chaos Experiments in Production. *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice*, pages 31–40, 2019.

[5] Michael Smith. DevOps Foundations: Chaos Engineering, LinkedIn Learning Course. `https://www.linkedin.com/learning/devops-foundations-chaos-engineering/organized-chaos`, July 2019. Accessed: 2019-11-15.

[6] ITIC. One Hour of Downtime Costs > $100K For 95% of Enterprises. `https://itic-corp.com/blog/2013/07/one-hour-of-downtime-costs-100k-for-95-of-enterprises/`, July 2013. Accessed: 2019-12-04.

[7] Ali Basiri, Aaron Blohowiak, Lorin Hochstein, Nora Jones, and Casey Rosenthal. *Chaos Engineering: Building Confidence in System Behavior Through Experiments*. O'Reilly Media, Inc., August 2017.

[8] Kshirasagar Naik and Priyadarshi Tripathy. *Software Testing and Quality Assurance: Theory and Practice*. Wiley, 2008.

[9] CabForward. The Difference Between Reliable and Resilient Software. `https://cabforward.com/the-difference-between-reliable-and-resilient-software/`, June 2015. Accessed: 2019-11-14.

[10] Rakesh Kumar Lenka, Kabita Manjari Nayak, and Sarthak Padhi. Fault Injection Techniques - A Brief Review. *2018 International Conference on Advances in Computing, Communication Control and Networking*, pages 832–837, October 2018.

[11] Software Testing Fundamentals. Gray Box Testing. http://softwaretestingfundamentals.com/gray-box-testing/, December 2010. Accessed: 2019-12-02.

[12] Fredric Paul. Chaos Engineering Explained. https://blog.newrelic.com/engineering/chaos-engineering-explained/, 2019. Accessed: 2019-11-07.

[13] Jose Fermoso. Netflix Goes Down For 11 hours, Wall Street Pushes Its Stock to All-Time High. https://www.wired.com/2008/03/netflix-goes-do/, March 2008. Accessed: 2019-11-13.

[14] Ashley Rodriguez. Indian Commuters Are Coping With Their Terrible Commutes By Watching Netflix. https://qz.com/india/989659/, May 2017.

[15] William Goddard. Chaos Engineering - A Complete Introduction. https://www.itchronicles.com/software-development/chaos-engineering-introduction/, March 2019. Accessed: 2019-11-13.

[16] Ali Basiri, Niosha Behnam, Ruud de Rooij, Lorin Hochstein, Luke Kosewski, Justin Reynolds, and Casey Rosenthal. Chaos Engineering. *IEEE Software*, 33(3):35–41, May 2016.

[17] Sathiya Shunmugasundaram. Continuous Chaos — Introducing Chaos Engineering into DevOps Practices. https://medium.com/capital-one-tech/continuous-chaos-introducing-chaos-engineering-into-devops-practices-75757e1cca6d, July 2018.

[18] Edaena Salinas. Tammy Bütow on Chaos Engineering. *IEEE Software*, 35(5):125–128, Sep 2018.

[19] Jacob Winch. Testing in Production: Rethinking the Conventional Deployment Pipeline. https://www.theguardian.com/info/developer-blog/2016/dec/20/testing-in-production-rethinking-the-conventional-deployment-pipeline, December 2016. Accessed: 2019-11-05.

[20] Ali Basiri, Lorin Hochstein, Nora Jones, Casey Rosenthal, and Haley Tucker. The Business Case for Chaos Engineering. *IEEE Cloud Computing*, 5(3):45–54, June 2018.

[21] Mathias Lafeldt. The Limitations of Chaos Engineering. https://sharpend.io/the-limitations-of-chaos-engineering/, January 2018. Accessed: 2019-11-04.

[22] Rasmus Ros and Per Runeson. Continuous Experimentation and A/B Testing: A Mapping Study. *2018 IEEE/ACM 4th International Workshop on Rapid Continuous Software Engineering*, pages 35 – 41, 2018.

[23] Jan Bosch, Pavel Dmitriev, Aleksander Fabijan, and Helena Homström Olsson. The Evolution of Continuous Experimentation in Software Product Development: From Data to a Data-Driven Organization at Scale. *2017 IEEE/ACM 39th International Conference on Software Engineering*, pages 770–780, 2017.

[24] Aaron Strong. Containerization vs. virtualization: What's the difference? https://www.burwood.com/blog-archive/containerization-vs-virtualization, November 2019. Accessed: 2019-12-13.

[25] IBM. Virtualization: A Complete Guide. https://www.ibm.com/cloud/learn/virtualization-a-complete-guide, June 2019. Accessed: 2019-12-13.

[26] Kubernetes. What is Kubernetes. https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/. Accessed: 2019-12-13.

[27] Margaret Rouse. Cloud Service Provider. https://searchitchannel.techtarget.com/definition/cloud-service-provider-cloud-provider, April 2018. Accessed: 2019-12-16.

[28] Guru99. What is AWS? Amazon Cloud Services Tutorial. https://www.guru99.com/what-is-aws.html. Accessed: 2020-01-09.

[29] Cynthia Harvey and Andy Patrizio. AWS vs. Azure vs. Google: Cloud Comparison [2019 Update]. https://www.datamation.com/cloud-computing/aws-vs-azure-vs-google-cloud-comparison.html, January 2019. Accessed: 2019-12-16.

[30] Hemant Sharma. AWS Tutorial: Introduction to Cloud Computing. https://www.edureka.co/blog/amazon-aws-tutorial/, May 2019. Accessed: 2019-12-16.

[31] IntelliPaat. AWS vs Azure vs Google - Detailed Cloud Comparison. https://intellipaat.com/blog/aws-vs-azure-vs-google-cloud/, December 2019. Accessed: 2019-12-16.

[32] Elizabeth Bjarnason, Emelie Engström, Martin Höst, Per Runeson, and Margaret-Anne Storey. Using a Visual Abstract as a Lens for Communicating and Promoting Design Science Research in Software Engineering. *2017 ACM/IEEE International Symposium on ESEM*, pages 181 – 186, 2017.

[33] Per Runeson, Emelie Engström, and Margaret-Anne Storey. The Design Science Paradigm as a Frame for Empirical Software Engineering. In Michael Felderer and Guilherme Horta Travassos, editors, *Contemporary Empirical Methods in Software Engineering*. Springer, 2020.

[34] Jens Andersson and Maria Kihl Palm. *Datakommunikation och nätverk*. Studentlitteratur AB, 1 edition, 2013.

Appendices

Appendix A
Interview Guides and Questions

This appendix contains the interview guides and interview questions used for the thesis's interviews, which were organized in terms of the interview themes ICA IT, Why Chaos Engineering and Chaos Engineering Areas. While the ICA IT interviews were openly structured, the interviews of the remaining two themes were semi-structured. Therefore, the ICA IT interviews were led only by a interview guide containing a set of question areas, while the interviews of the Why Chaos Engineering and Chaos Engineering Areas themes were guided by sets of actual questions. All interviews were held in Swedish, but the guides and questions have been translated into English in this appendix.

All interviews followed the same general structure and consisted of four phases: context specification, opening part, main part and conclusion. The context specification was always carried out in the same way, regardless of interview theme, and it followed the following steps:

- Present the interview: a short summary of how the interview was to be structured
- Ask for questions or comments
- Present the interviewer
- Introduce Chaos Engineering, if the interviewee was not already familiar with the discipline
- Introduce the interview theme
- Explain how the results of the interview were to be used
- Ask for questions or comments

The opening part of every interview consisted the following steps:

- Ask about role at ICA, and, if not implicitly answered, ask about previous roles at and outside of ICA

- Ask about work tasks a regular day
- Ask about work tasks a critical day
- Ask about education

The conclusions of the interviews were carried out identically regardless of interview theme as well. The conclusions started by repeating how the interviewee's answers had been interpreted while encouraging interruptions whenever there was a wish for a change in or an addition to one of the answers. Then final comments were asked for, before the interviewee was thanked for his or her time and the interview was ended.

The only part of the interviews where they differed depending on their theme was, in other words, the main parts of the interviews. The sections A.1 through A.3 present these main parts of all three interview themes.

A.1 ICA IT

The following question areas were discussed:

- The interviewee's organization: what does it do?
- The organization's place at ICA: how does it relate to the rest of the IT department?
- The organization's products: what does it deliver?
- The organization's customers: who use the delivered products?
- The organization's size: how big is it, in terms of employees and/or budget?
- The organization's structure: which roles exist in the organization?

A.2 Why Chaos Engineering

The following open questions were asked:

- What happens when a system you work with stops working? Who are affected? How are you affected?
- What is the cost of downtime for ICA? What are the consequences?
- When Amazon's website goes down it costs the company a tremendous amount of money due to lost incomes. Is there any difference if ICA's applications go down?
- How often do incidents occur? How are they prioritized? How do incidents of different priorities differ?
- How long does it take, would you say, to handle a critical incident? What happens during that time?
- As a user, how are you affected by an incident?

- As an incident handler or general employee, how are you affected by an incident?
- What would an incident-free system mean, for you and for the users?
- Why is it difficult to have a week free from critical incidents?
- Tell me about the worst incident you have experienced at ICA.
- Would you say that ICA has good incident handling? Does ICA have good mechanisms for reliability in their applications?
- What would the benefits be if the number of incidents was decreased?
- What is your impression of Chaos Engineering in relation to what we have discussed today?

The following closed questions were asked:

- On a scale of 1-5, how much would you agree with the statement "I see the benefits of Chaos Engineering in general"? (1 = I do not agree in the slightest, 5 = I completely agree)
- On a scale of 1-5, how much would you agree with the statement "I see the benefits of Chaos Engineering somewhere at ICA"? (1 = I do not agree in the slightest, 5 = I completely agree)

A.3 Chaos Engineering Areas

As an introduction, the question areas given in section A.1 were discussed. Then, the following open questions were asked:

- How are your software systems structured, in terms of architecture, platforms, tools, cloud service providers and similar aspects?
- Which mechanisms for reliability have been implemented in the systems?
- How are the systems tested?
- Have there been any problems with the software systems? What caused them?
- What is your impression of Chaos Engineering in relation to what we have discussed today?
- Which Chaos Engineering attacks do you consider to be suitable for ICA's software systems?
- What is your view on automated as opposed to manual Chaos Engineering at ICA?
- Should Chaos Engineering be included in many software systems at ICA or do you consider its use more limited? If limited, why and to which systems?

- If you were to search for suitable systems to test with Chaos Engineering, where would you look?

The following closed questions were asked:

- On a scale of 1-5, how much would you agree with the statement "I see the benefits of Chaos Engineering in general"? (1 = I do not agree in the slightest, 5 = I completely agree)
- On a scale of 1-5, how much would you agree with the statement "I see the benefits of Chaos Engineering somewhere at ICA"? (1 = I do not agree in the slightest, 5 = I completely agree)

Appendix B
Experimentation Protocol

This appendix contains the experimentation protocol used as part of the tool evaluation process.

Experimentation Protocol		
General information	Name	
	Developer	
	First released in (year)	
	Written in (language)	
Availability	Code available at	
	Documentation available at	
Functionality	Types of attacks	
	How to install tool	
	Ways to control tool	
	Other Functionality	
	Lacking Functionality	
Environment	Prerequisites	
	Suitable infrastructures and platforms	
	Supported languages/application types	
Community	Additional material (such as a set of finished Chaos Experiments) exists	
	Tool is still maintained	

Appendix C
Questionnaire

The questionnaire which accompanied the Chaos Engineering demonstrations held for various stakeholders at ICA contained five questions, to which answers where completely optional. The reason for this was that not every person was considered relevant for all five questions, because of their more operational roles. However, answers to the two last questions, regarding the perceived benefits of Chaos Engineering, were highly encouraged due to the simple nature of the questions. Also, since the respondents had just been given a demonstration of Chaos Engineering, it was assumed that every respondent had a clear enough idea of the principles of Chaos Engineering to give their opinions on its benefits.

The five questions were given in Swedish, but they have been translated into English in this appendix. The questions were:

1. These areas or systems at ICA are the first ones that come to mind when I think of reliability testing or Chaos Engineering:

2. I am aware of the following reliability mechanisms in those areas or systems:

3. These attacks are the first ones that come to mind when I think of Chaos Engineering in relation to ICA: (The answer does not need to include any of the three attacks demonstrated today)

4. I see the benefits of Chaos Engineering in general: (Circle a number between 1 and 5 where 1 = “I do not agree in the slightest” and 5 = “I agree completely”)

5. I see the benefits of Chaos Engineering somewhere at ICA: (Circle a number between 1 and 5 where 1 = “I do not agree in the slightest” and 5 = “I agree completely”)

INSTITUTIONEN FÖR DATAVETENSKAP | LUNDS TEKNISKA HÖGSKOLA | PRESENTERAD 2020-04-28

EXAMENSARBETE Building a Framework for Chaos Engineering
STUDENT Hugo Jernberg
HANDLEDARE Per Runeson (LTH), Michael Adis (ICA Gruppen AB)
EXAMINATOR Emelie Engström (LTH)

Chaos Engineering: Konsten att undvika kaos genom att orsaka kaos med flit

POPULÄRVETENSKAPLIG SAMMANFATTNING **Hugo Jernberg**

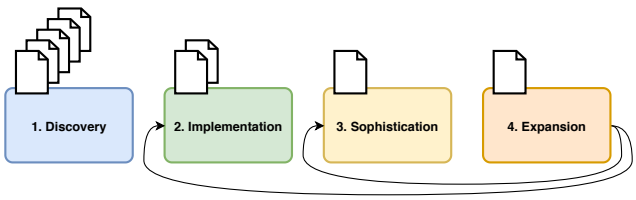
Det kan låta som en dålig idé att med flit orsaka kaos i en programvaruapplikation. Det är inte riskfritt, och att göra det på fel sätt kan få ödesdigra konsekvenser — men om det görs på rätt sätt, kan belöningen vara en mer tillförlitlig applikation.

I en perfekt värld skulle det gå att förutspå effekterna av alla tänkbara problem, men i verkligheten är det svårare — många system, kanske framför allt programvarusystem, tenderar att vara för komplexa för att det ska vara möjligt. På grund av detta kan man välja att tillämpa disciplinen Chaos Engineering.

Chaos Engineering går ut på att utföra experiment, där man med flit inför ett problem i sin programvara för att se hur den beter sig. Om det visar sig att programvaran slutar fungera när problemet införs, då har man identifierat en svaghet. När den svagheten sedan är åtgärdad är effekten att programvaran är mer tillförlitlig och bättre förberedd ifall problemet vi införde med flit någon gång inträffar naturligt.

I mitt examensarbete har jag föreslagit ett ramverk för hur man på ett säkert sätt kan arbeta med Chaos Engineering på ICA Gruppen AB, en grupp av företag vars kärnverksamhet är dagligvaruhandel. Ramverket består av fyra aktiviteter som stöttas av stöddokument, och genom att följa aktiviteterna och samtidigt ta hjälp av stöddokumenten är målet att definiera en lämplig Chaos Engineering-metodik för en viss programvaruapplikation. En sådan metodik grundas i användandet av olika Chaos Engineering-verktyg. Ett trettiotal verktyg utvärderades därför under examensarbetets gång, varav tolv valdes ut till ramverket.

De fyra aktiviteterna i ramverket visas i figuren nedan tillsammans med sina stöddokument. Den första aktiviteten i ramverket (discovery) bygger en lista av problem som är lämpliga att införa i en viss applikation. Den andra aktiviteten (implementation) påbörjar användandet av ett av de tolv utvalda verktygen. Den tredje aktiviteten (sophistication) grundas i en mognadsmodell för Chaos Engineering och syftar till att förbättra hur disciplinen används. Slutligen ämnar den fjärde aktiviteten (expansion) att utöka antalet problem som införs i applikationen. Det rekommenderas att upprepa de tre sista aktiviteterna regelbundet, och regler för när dessa upprepningar är passande är också en del av det föreslagna ramverket.



Tanken är att ramverket ska göra det enklare att komma igång med Chaos Engineering. Kanske kan det leda till att ICA:s applikationer blir bättre förberedda — så att de kan stå emot vad de än utsätts för i dagens turbulenta värld.