Aalto University
School of Science
Master's Programme in ICT Innovation

Adriaan Knapen

Chaos Engineering for Containerized Applications with Multi-Version Deployments

Master's Thesis
Espoo, February 1, 2021

Supervisors:	Professor Martin Monperrus, KTH Royal Institute of Technology
	Professor Antti Ylä-Jääski, Aalto University
Advisor:	Long Zhang M.Sc., KTH Royal Institute of Technology
	Tatu Kairi M.Sc., Eficode

Aalto University
School of Science

Aalto University
School of Science
Master's Programme in ICT Innovation

ABSTRACT OF
MASTER'S THESIS

Author:	Adriaan Knapen		
Title:	Chaos Engineering for Containerized Applications with Multi-Version Deployments		
Date:	February 1, 2021	**Pages:**	59
Major:	Cloud Computing and Service	**Code:**	SCI3081
Supervisors:	Professor Martin Monperrus Professor Antti Ylä-Jääski		
Advisor:	Long Zhang M.Sc. Tatu Kairi M.Sc.		

It is common practice to use versioning policies to create unique identifiers for applications running in production. Many times the schema used for the versioning identifier also includes information which allows one to infer the impact of the changes bundled in this new version. One well known and frequently used versioning policy is semantic versioning (SemVer), which, if properly adhered to, can be used to determine which different versions can be used interchangeably. However, many systems depending on the semantic versions of their applications do not explore the version compatibility in their applications. Therefore never obtaining confidence about whether the application adheres to the version compatibility promises made by adopting semantic versioning.

In this thesis, we present a novel approach to applying chaos engineering practices perturbing containerized application deployments. These perturbations create multi-version execution environments, where multiple different versions of the application are running concurrently. To achieve these multi-version deployments we created the `container-registry-proxy` (CRP), which applies proxy practices to perturb the container distribution protocol. The employed perturbation model targets the versioning of SemVer container images. When a container image is distributed, the perturbation engine serves any image compatible according to SemVer, picked at random. Consequently, deployments with multiple containers are likely to end up running multiple versions.

The CRP is evaluated against distributed deployments of two popular databases: PostgreSQL and Redis. We find that the PostgreSQL image for redundant PostgreSQL clusters distributed by Bitnami bundles additional tooling. Incompatibilities between these bundled tools were not reflected in the version identifier according to SemVer, causing multi-version deployments to encounter crashes and take longer before being able to handle requests. None of our multi-version deployments of Redis caused containers to crash nor did it take longer to be operational than version wise homogeneous deployments.

Keywords:	chaos engineering, multi-version, containers
Language:	English

Contents

1 Introduction **1**
1.1 Problem Statement 1
1.2 Context 2
1.3 Research Questions 2
1.4 Contributions 2
1.5 Ethics and Sustainability 3

2 Background **4**
2.1 Distributed Systems 4
2.2 Chaos Engineering 5
2.3 Dependency Versioning 6
2.4 Semantic Versioning 6
2.5 Containers 8
2.5.1 Containerized Service Orchestration 8
2.5.2 Container Image Distribution 9

3 Related Work on Multi-Version Deployment Perturbations **11**
3.1 Multi-version Execution 11
3.2 Software Diversity 12
3.3 Chaos Engineering 13
3.4 Version Compatibility 14

4 Design & Implementation of the CRP **16**
4.1 Goals 16
4.2 Design Overview 17
4.3 Container Image Distribution Perturbation 19
4.4 Monitoring Perturbations 21
4.5 Usage 21
4.6 Implementation 22
4.7 Plugins Available for the CRP 23

4.8 Summary 23

5 Experimental Evaluation Protocols 25
5.1 Applications 25
5.1.1 PostgreSQL 26
5.1.2 Redis 27
5.2 Experimental Setup 28
5.3 Perturbations 29
5.4 Metrics 30
5.5 Versions Under Test 31
5.6 Iteration Count 31
5.7 Summary 32

6 Experimental Results 34
6.1 Application Evaluations 34
6.1.1 PostgreSQL 34
6.1.2 Redis 36
6.2 Detected Defects 39
6.2.1 PostgreSQL 39
6.2.2 Redis 39
6.3 Performance Impact of the CRP 40
6.4 Summary 41

7 Discussion 42
7.1 Perturbation Possibilities for Container Deployments 42
7.2 Detecting Version Incompatibility using Chaos Engineering . 43
7.3 Runtime Overhead of the Perturbations 44

8 Conclusions 46
8.1 Recapitulation 46
8.2 Future Work 47

Bibliography 49

Chapter 1

Introduction

1.1 Problem Statement

It is common to continue making changes to software applications while the application is already running in production environments. These changes can consist of patches for detected faults and vulnerabilities, or add new features. As part of this process a continuous stream of new versions of this application emerge.

Deploying without downtime is increasingly being required. During such zero-downtime deployment to a newer version of an application there will be a moment where the older version of the application is handling request while the newer one is initializing to take over. Research into the behavior of an application while it is running in such multi-version deployment has not been extensively studied.

In this thesis we explore the potential of using chaos engineering practices to intentionally create multi-version deployments in production environments. Having a systematic method to experiment with multi-version deployments is useful to increase our confidence in the resilience of the system.

Growing adoption of practices like continuous deployment are making it increasingly common to release and deploy new versions of an application at a higher frequency. Moving the duration of release cycles from weeks or months to releasing multiple versions a day. With frequent deployments becoming more common increases the risk for multi-version deployments to accidentally occur. Therefore, raising the importance of a system withstand running as a multi-version deployment.

1.2 Context

This research focuses on applications deployed as containers. A container is a process of an application running on a host system, which is virtualized on the operating-system-level. A more elaborate introduction to containers, including tools to manage deployments of multiple applications running as containers, will be introduced in chapter 2: Background.

The industry standards governing the container formats and runtimes, created by the Open Container Initiative, form the basis for a whole range of tools which facilitate and manage deploying applications as containers. Such mechanism to facilitate creating container based deployments needs to cover distributing the binaries of the application and run these binaries as a container. Later we will refer to the process of distributing and running a binary as a container as a "container deployment mechanism".

1.3 Research Questions

This thesis will explore the following research questions (RQs):

RQ1 What aspect of container deployment mechanisms can be used to perturb a deployment into a multi-version deployment?

RQ2 To what extent can perturbing a container deployment mechanism facilitate detecting version incompatibility issues?

RQ3 What amount of runtime overhead is introduced by using the container perturbation mechanisms suggested in RQ2?

1.4 Contributions

This thesis makes the following contributions to the existing scientific literature:

- A novel approach to monitoring and perturbing the container image distribution process for container based applications.
- The Container Registry Proxy (CRP) is a generic implementation which facilitates observing and perturbing the container image distribution process. The Semantic Chaos plugin for the CRP, which enables one to

perturb the versions of images resolved in the container image distribution process. The source code of both the CRP with the Semantic Chaos plugin, and the evaluation experiment code and data are made publicly available at respectively `https://github.com/Addono/container-registry-proxy` and `https://github.com/Addono/thesis-experiment`.

- Experimental evaluation of the CRP and Semantic Chaos plugin, showing how this tool can be used to detect version incompatibility issues of multi-version deployments. Specifically, a version incompatibility issue is uncovered in the PostgreSQL images from Bitnami.

1.5 Ethics and Sustainability

The authors acknowledge the importance of open-access and open-source. Hence, all tools developed and results gathered as part of the research for this thesis is publicly released under the MIT license. In addition, all applications which were used to evaluate our proposed methodology on are open-source and publicly available as well. This allows anyone to reuse the tools developed during this thesis for any purpose they see fit and be able to replicate the conducted experiments.

In 2010 the electricity usage of data centers accounted for an estimated 1.1% to 1.5% of the worldwide electricity usage, whereas in a developed country like the US data centers are estimated to consume between 1.7% and 2.2% of the total electricity consumption of the nation [10]. A substantial amount of the energy used in data centers is wasted due to failures [17]. Our research explores novel methods to detect certain types of application failures, such that these can be resolved pro-actively. As such our research goals of detecting potential failure causing issues in applications yields applications less prone to fail, thus eliminating part of the energy wasted due to application failures.

Chapter 2

Background

2.1 Distributed Systems

A distributed system is a collection of independent computing elements that to their users appear as a single coherent system [39]. Here computing elements can either be an autonomous software process or a hardware device, and users can be people or applications. Since distributed systems appear to their users as a single coherent system it is often times necessary for the individual compute elements to collaborate.

There are several reasons why a distributed system might be desired or even required, including [21]:

Inherently distributed computations For some applications, like reaching consensus between various geographically dispersed parties, the task at hand is inherently distributed.

Resource sharing Often it is not practical nor cost-effective to completely replicate resources, such as entire databases or peripherals, for each computing element. In distributed systems, resources can be distributed over and shared between the compute elements, as to prevent the need for complete resource replication.

Improved reliability Replication in distributed systems has the potential to hide failure from individual resources, as other non-failing replicas can take over.

Scalability Distributed system can be used to incorporate scalability into the system. Scalability can be desired for several reasons, for one capacity or size in which adding more resources to the system has the goal to

increase the amount of users the system can serve. Another scalability dimension is geographic scalability, where the goal is to reduce the latency between the system and the users by geographically distributing the systems resources [28].

2.2 Chaos Engineering

Chaos engineering is a discipline which aims to increase confidence in the resilience of a distributed system running in production [1]. It advocates for taking an experimental approach to verifying that a system can withstand turbulent conditions.

A chaos engineering experiment consists out of four steps, according to the *Principles of chaos engineering* [33]:

1. First, the applications steady state is defined. The definition of the steady state should be a measurable output which reflects normal behavior of the system under test.

2. Secondly, device a hypothesis for the experiment stating that the steady state will be maintained in both the control and experimental group.

3. Thirdly, define one or more real-world events to perturb the application, e.g. network outages, disk corruption, or server outages.

4. Lastly, execute the experiment by perturbing the experiment group. Afterwards evaluate the difference in steady state between the experiment group and the control group.

The practice of chaos engineering emerged at Netflix [1], where they have been using this practice to improve the resilience of their production environments. Netflix developed several tools to adopt chaos engineering, some of which have become well known in the chaos engineering community. One of these tools is Chaos Monkey [27], which randomly terminates server instances in their production environment. Chaos Monkey was introduced to encourage Netflix engineers to build systems resilient against failure of individual machines [1].

Chaos engineering experiments are encouraged to run directly in production, or otherwise in environments as similar as possible to production environments. Accurately replicating the complexity and scale of production environments can be hard, therefore running chaos experiments on a test environment

gives less confidence in the resilience of the system running in production than directly running the experiment on production environments [2].

2.3 Dependency Versioning

For many modern software systems it has become common to re-use pre-existing components. In an effort to standardize these reusable components have many programming languages developed tools to bundle these components. We will use the term *packages* to denote these bundled components, however other names exist, including modules, components and libraries. In the last decade it has become common for programming languages to centrally store these packages in a *package repository* [15]. Packages needed by a software system are called *dependencies*. Client side tools called *package managers* interact with the package repository and retrieve the desired dependencies.

Software packages tend to evolve after they have been published. This could, among other reasons, to add new functionality, fix existing bugs or patch security issues. Hence, over time multiple versions of one package will exist. Therefore, package managers started to version their dependencies. Dependency versioning can either be a *fixed version* (e.g. "use version 1.0.0") or a *version range* (e.g. "use version 1.0.0 or newer"). Using fixed versioning results in more deterministic behavior than version ranges, as the package manager will always attempt to install the same version of the package. On the flip side, version ranges can increase the security of a system by automatically letting dependency versions resolve to the latest security patch [12]. In addition, using version ranges increases the freedom of a package manager as to decide what is the best version to install given some constraints. These constraints become increasingly complex when multiple dependencies of a software system depend on different versions of the same dependency. When a software system is a package itself, it could even result in cyclic dependencies, where a dependency of the software system depends on an older version of the software system. In such case, version ranges can become essential as to prevent version conflicts arising from these complex dependency networks.

2.4 Semantic Versioning

Semantic versioning is a method of systematically constructing version identifiers, which are commonly used to indicate software releases. A semanti-

cally versioned release identifier explicitly communicates compatibility between versions as part of the notation of the version. Such release identifier consists of three integers separated by a full stop and optionally followed by pre-release or build metadata. These three integers denote the major, minor and patch version where this release belongs to. For example, version 1.2.3 would indicate the third patch release for the second minor release for the first major release.

The semantic versioning specifies whether the major, minor or patch number should be incremented for a new release [34]. Major releases are used for introducing non-backwards compatible changes, such change could consist of removing features, changes in configuration or alterations in the behavior of the system. Minor releases, also known as feature releases, are restricted to adding functionality whilst remaining backwards compatible with earlier versions in this major release. Patch releases can only introduce changes to the behavior of software which resolve deviations from the expected behavior of previously released features. As such, a patch release can result in non-backwards compatible changes when other systems have started to depend on the resolved deviations from the original intended specifications. Lastly, there is the possibility to add details on pre-release and build metadata, neither of which should be used to introduce new or changes to existing functionality. Pre-releases are used to track and release versions which are potentially not yet ready for adoption in production environments. Build metadata allows releasing multiple software artifacts of the same version with varying build configurations, for example targeting different operating systems, CPU architectures or using different compilers.

From these requirements imposed by semantic versioning we can deduce some properties on compatibility between versions which should for semantically versioned software [15]. For any semantic version $x.y.z$, with x as the major version, y as the minor version and z as the patch version, any version $a.b.c$ is compatible if both of the following conditions are met:

- $x = a$, as different major versions have no guarantee on being compatible.

- $y \leq b$ given that none of the features which are introduced in patches after y are used, as newer releases should only introduce optional new features.

2.5 Containers

Containers are a virtualization method on the OS level. A container uses the kernel from the host it is running on. OS specific features, like Linux namespaces, cgroups and chroot, are used to isolate containers from each other [23]. In contrast to hypervisor-based virtualization technologies, which provides each of the virtual environments emulated hardware, containers tend to be more lightweight. Performance wise outperform containers nearly always hypervisor-based virtual environments, as the overhead of containers is near-zero [16].

2.5.1 Containerized Service Orchestration

Docker is one of the implementations of containers on Linux, which appreciates growing user interest and has accrued widespread adoption [36]. While Docker has greatly contributed to bringing the potential of Linux containers to the cloud, it does not address managing the life cycle of containers. Special purpose container orchestration engines emerged to fill this gap. Kubernetes and Docker Swarm are container orchestration engines who enjoy the most adoption [20]. These container orchestration engines are able to use the resources available on multiple machines.

Container orchestration engines automate the process of provisioning and managing one or more containers, as to deliver one or more services. Tasks performed by the orchestration engine can be divided into three layers: resource management, scheduling and service management [20].

Resource management tasks focus on allocating a sufficient amount of resources, such as CPU, disk space, persistent storage, and configured IP addresses to each of the containers. Additionally, resource management tries to maximize the utilization of the available resources and minimizes interference caused by competition for resources.

The scheduling layer is responsible for efficiently allocating resources over the available machines. For example, replicating services over multiple machines for redundancy, automatically scaling services with the demand, and restarting failed containers when they emit an error.

Lastly, the service management layer offers more high-level features. This includes load-balancing incoming requests to a replicated service, isolating containers by application group and readiness checking to route traffic only to containers which are able to process requests.

One of the most popular container orchestration engines is Kubernetes.

Originally developed for internal use at Google where it has been in use for years in production before it was publicly released as open-source. Kubernetes uses a master/slave architecture to deploy applications over a cluster of machines, where each machine is called a *node*. Users submit the desired state of their applications to a master node. The master node then schedules containers onto both master and slave nodes.

One of the fundamental components in Kubernetes is the *Pod*. A pod is one or more containers scheduled together. Kubernetes comes with various other components to replicate pods, such as *ReplicaSet* and *StatefulSet*.

2.5.2 Container Image Distribution

The Open Container Initiative (OCI) is a project of The Linux Foundation, whose purpose is to create an open industry standard for containers. This initiative has published specifications for the image format of containers and their runtime. In addition, a specification on distributing images is in progress. This distribution specification is a standardization of the Docker Registry HTTP API V2 protocol. Our work targets the aforementioned Docker Registry protocol, however given that it is being raised to an industry standard it is likely to also be compatible with solutions from many other vendors.

Container images are stored in so-called container registries. Retrieving an image from such registry, often referred to as "image pulling", is done based on an image identifier. This image identifier is a name post-fixed with a version identifier. This version identifier can either be a "digest" or a "tag". A digest is a serialized hash result from the binary content of the image. Whereas a tag is a version string, which the container registry resolves to a digest. As such, digests are immutable, since it would require finding a hash collision, which given that the prescribed hash algorithm is SHA-256 has not yet shown to be feasible. Tags on the other hand are mutable, hence a registry implementation can allow overwriting the digest a tag is pointing to with another digest.

The process of retrieving an image by tag is shown in Figure 2.1 When an image is retrieved by tag, then first the client sends an HTTP GET request to resolve the tag to a digest. After the corresponding digest is received another HTTP GET request is made to retrieve the location of the binaries making up this image. Lastly, all binaries are downloaded and the client can construct the image.

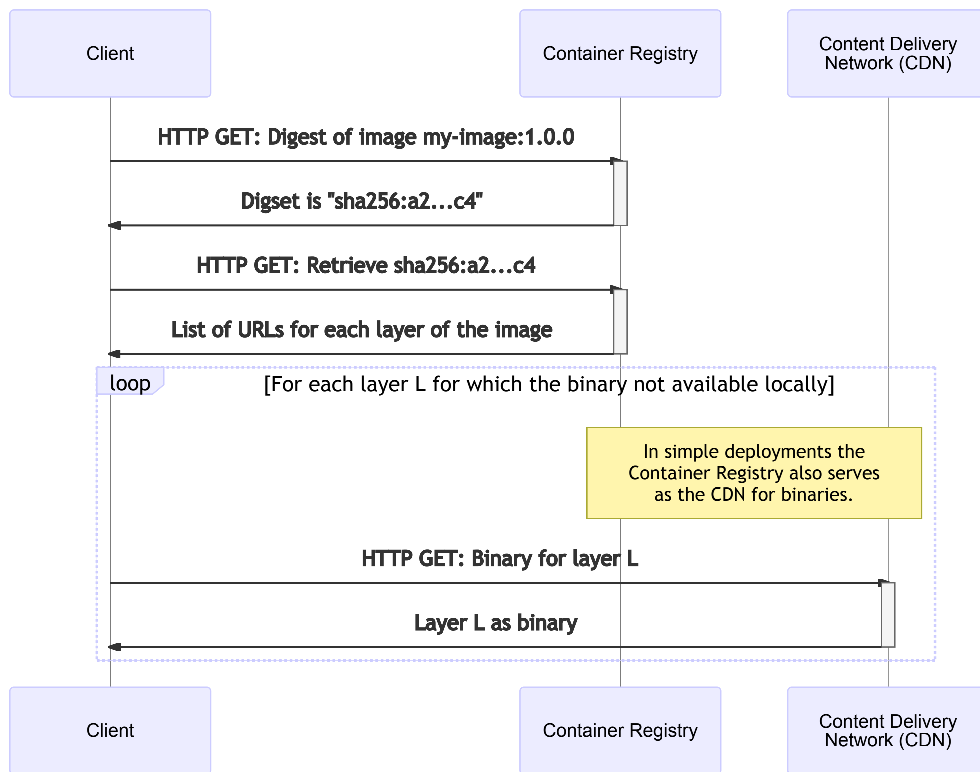


Figure 2.1: Sequence diagram for retrieving an image by tag from a container registry.

Chapter 3

Related Work on Multi-Version Deployment Perturbations

3.1 Multi-version Execution

In multi-version execution, also known as multi-variant execution, different versions of the same application are executed in parallel. Hosek et al. [19] develops a prototype called Mx, which runs multiple versions of a program concurrently for enhanced error recovery. When one of the programs encounters an error, then the state of one of the other programs which manages to surpass the erroneous point would be synchronized to the failing program and execution would continue. This approach allows deployments to create a functional application from multiple versions each with their own defects. Such situation could for example occur when upgrading an application to newer version. The new release might fix an existing bug while at the same time introducing new ones. Multi-version execution would then be able to combine the functional parts of both versions and prevent faults from emerging as failures.

Coppens et al. [8] uses multi-variant execution for security purposes. They created GHUMVEE, a multi-variant execution environment capable of running various realistic programs. GHUMVEE is security-oriented rather than reliability-oriented, such as Mx. By running different variants in parallel, it detects when one of the variants reaches a different state than the others, which causes the diverging variant to be terminated. This prevents external attackers from being able to exploit memory corruption vulnerabilities.

The work of Österlund et al. [30] creates a prototype called kMVX, which applies multi-variant execution on the kernel level. Which can prevent an attacker from exploiting bugs in the kernel. Evaluation of kMVX on several

popular server applications shows that the performance impact of enabling kMVX is at worst between 20% and 50%.

Other research on multi-version execution uses it to update cloud applications without downtime. Chen et al. [6] explores the possibility of updating applications by forking the currently running process and applying dynamic software updating to update the forked process. Once the update completes the original process terminates and only the updated fork remains. This allows both versions to be operational during the update and hence update the application without downtime. As part of this work they implemented Multivesion for Updating of Cloud (MUC), a prototype to update Linux applications by combining dynamic software updating and multi-version execution.

Pina et al. [31] improves upon MUC with Mvedsua, which again combines multi-version execution and dynamic software updating for enabling software updates without downtime. In addition, Mvedsua leverages multi-version execution to also improve the availability and reliability of the application. Mvedusua adds a mechanism which catches errors when the application is still running multiple versions. When an error is detected, the application rolls back to the point before the update.

These previously discussed applications of multi-version execution use multi-version execution to improve the resilience and security of an individual instance of an application. Our work focuses on improving the resilience of the deployment of a service as a whole, by means of redundancy within the deployment we are able to terminate and recreate individual instances.

3.2 Software Diversity

The field of software diversity has been actively studied since the 1970s. Initially the purpose of software diversity was to improve fault-tolerance, whereas improved application security is researched since the 1990s [3]. Since software is a result of human engineering is software diversity a man-made product. Hence, research into software diversity does not merely limit itself to how diversity improves certain properties of software, but also how to increase the diversity of software.

Some forms of software diversity emerge naturally from the method how software is developed. One form of natural software diversity occurs when multiple applications address the same problem, Oberheide et al. [29] runs several virus scanners in parallel as to improve the detection in malware and Gashi et al. [18] explores improving fault tolerance by using four different SQL database servers.

The continued development of a software application itself is also a form of natural software diversity. New versions of an application aim to resolve existing bugs and add features, however these updates can be incorrect and introduce new bugs [41]. Multi-version execution can be used to run different updates of the same application and use the bug free parts of either of them as to improve fault tolerance [19].

Our work explores actively increasing the diversity within application deployments which normally are predominantly version wise homogeneous. For this we leverage different versions of an application, these individual versions are by themselves already a form of natural software diversity, however their potential is not fully leveraged in deployments which are version wise homogeneous.

3.3 Chaos Engineering

The principles of chaos engineering have been applied to containerized applications. Inspired by Netflix' internal service Chaos Monkey [4], which randomly terminated virtual machines from their production services, a whole range of tools emerged replicating this behavior for container based deployments. For Kubernetes there is for example kube-monkey [38], PowerfulSeal [32], Litmus [25], chaoskube [24] and Chaos Mesh [5]. Whilst for Docker there is Pumba [22] and Blockade [40].

Simonsson et al. [35] build ChaosOrca, which takes a more granular approach by injects faults into individual system calls instead of perturbing a container or VM as a whole. As a result, the impact of perturbing certain system calls can be directly related to increases in system metrics like memory and cpu usage, and application metrics like increased response latency.

Zhang et al. [42] created ChaosMachine, a chaos engineering system whose perturbations target the exception-handling capabilities of Java applications. It differentiates between performing “falsification experiments” and “exploration experiments”. In the former experiment type the developer added annotations to the code with hypotheses on the behavior of the exception-handling logic, these annotations are then used by ChaosMachine to evaluate whether these hypotheses hold at runtime. Exploration experiments do not depend on annotations in the code, instead ChaosMachine discovers new hypotheses by reporting on the observed behavior exception-handling code blocks.

Earlier research on Chaos Engineering attempts to perturb the steady state of the system by injecting faults, such as terminating applications or increasing network latency, as to evaluate the resilience of a system against such faults.

Our research deviates by perturbing the version deployed for an application, instead of injecting faults, as to create multi-version deployments.

3.4 Version Compatibility

Compatibility between different versions of a software module is widely studied in the context of versioning policies of dependencies managed by package managers. Cossette et al. [9] studies the effectiveness of tools to automatically propose changes to adopt newer versions of Java libraries, concluding that for the five packages they studied only 20% of the upgrade recommendations are correct. In addition, they observed that the majority of the releases of newer versions of the libraries did not document changes to the API, neither in the release notes or by means of comments in the source code. Consequently, developer documentation is insufficient to solely depend on when determining the impact of adopting newer versions.

Dietrich et al. [14] examines for 109 Java based applications whether newer versions of their dependencies would be compatible, they conclude that for 75% of the version upgrades the public API of the dependency was changed, however these changes only caused issues for 8 out of the 109 applications. In addition, Dietrich et al. notices that API versioning practices were not properly adhered to as a means to reliably determine whether a version change would be compatible, since incompatible API changes were also introduced in minor and patch releases.

Raemaekers et al. [34] conducts a study on 22,000 Java libraries by analyzing 150,000 JAR binaries for binary compatibility with previous versions. They find that one third of the releases introduce a breaking change, these breaking changes are with equal frequency released as minor or major release, hence leaving developers little certainty to rely on version identification to detect compatibility.

Dietrich et al. [15] explores how applications declare their dependencies in different programming languages and package managers by analyzing 70 million dependencies. For all the package managers are the majority of the dependencies not fixed to a specific version, instead package managers have the freedom to pick a suitable version from a range of available versions.

Mujahid et al. [26] propose a method to identify version compatibility issues by leveraging the available test suites from dependent packages. Running the test suites from dependent packages multiple times with different versions of the package under test can be used to detect backwards incompatible changes between versions.

Here we covered research revolving around detecting version incompatibilities between software artifacts. Detecting these version incompatibilities is done either by manual inspection or automated tools. The methods to automatically detect version incompatibilities suffer from low accuracy or are only able to detect a specific type incompatibilities. This thesis proposes a novel method to the compatibility of different versions of a software application on an experimental basis.

Chapter 4

Design & Implementation of the CRP

This chapter covers the goal, design and implementation of the Container Registry Proxy (CRP). The CRP is a tool which gives fine-grained control over which version of a containerized application is deployed.

4.1 Goals

Software which has adopted semantic versioning, which is introduced in section 2.4, for their releases give their users guarantees on which versions are compatible with each other. This guarantee on compatibility should allow the user to change the version of the software they are consuming to a different compatible version. In extend, deployments featuring multiple instances of the same application might consist of various compatible versions. There are many cases where such situation would occur, for example when an application modifies persistent storage instantiated by instances using other version of the software, or in application clusters which during a rolling release run several different versions at the same time.

The goal of the CRP is to be able to validate whether an application adheres to the semantic versioning guaranties it promised. Specifically, we aim to construct a method to validate the following steady state hypothesis for our software under test:

> Given that an application uses semantic versioning, when deploying such application it should not malfunction when potentially different but compatible versions are used.

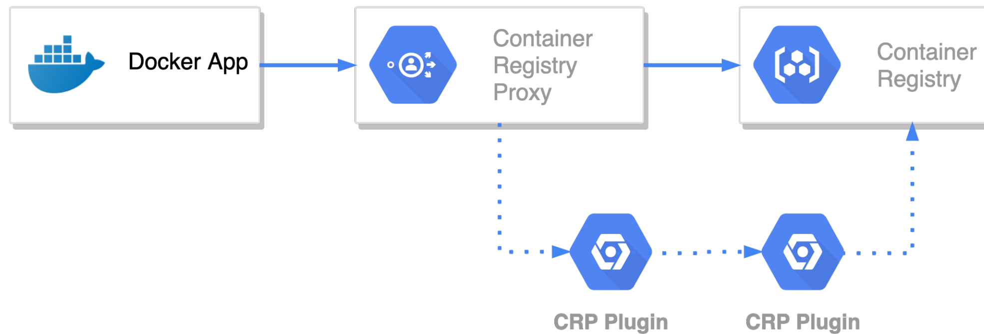


Figure 4.1: Request flow when using the Container Registry Proxy to retrieve container images from a container registry.

Reusability acts as a secondary goal for the tool. Ideally, neither the application nor the container orchestration platform used to deploy the containers needs to be modified in order to support it.

4.2 Design Overview

The Container Registry Proxy (CRP) functions as a proxy, which exposes a similar interface as a container registry. Figure 4.1 shows the interactions made between the CRP and its environment. Requests made from a container engine to an instance of the CRP will forward requests to an existing container registry. By default, all requests to the CRP will be directly forwarded to the container registry, hence functionally it would be identical as to retrieving container images directly from the registry. This default behavior can be modified by launching the CRP with plugins. They can modify or drop requests before they are forwarded to the container registry.

A high level layout of the components making up the CRP is shown in Figure 4.2. The proxy is started from the command line, which hands over execution to the entrypoint of the CRP. The entrypoint parses command line parameters and environment variables used to configure the plugin. This configuration is used to instantiate all plugins and starts the HTTP server. The HTTP server component actively listens for requests. Requests arriving at the HTTP server are parsed into an internally standardized format and then handed over to the enabled plugins. Each plugin can decide to alter or terminate the request before the request is forwarded to the container registry. Logging is responsibility for each of the individual components, which are centrally collected by sending them to the standard output.

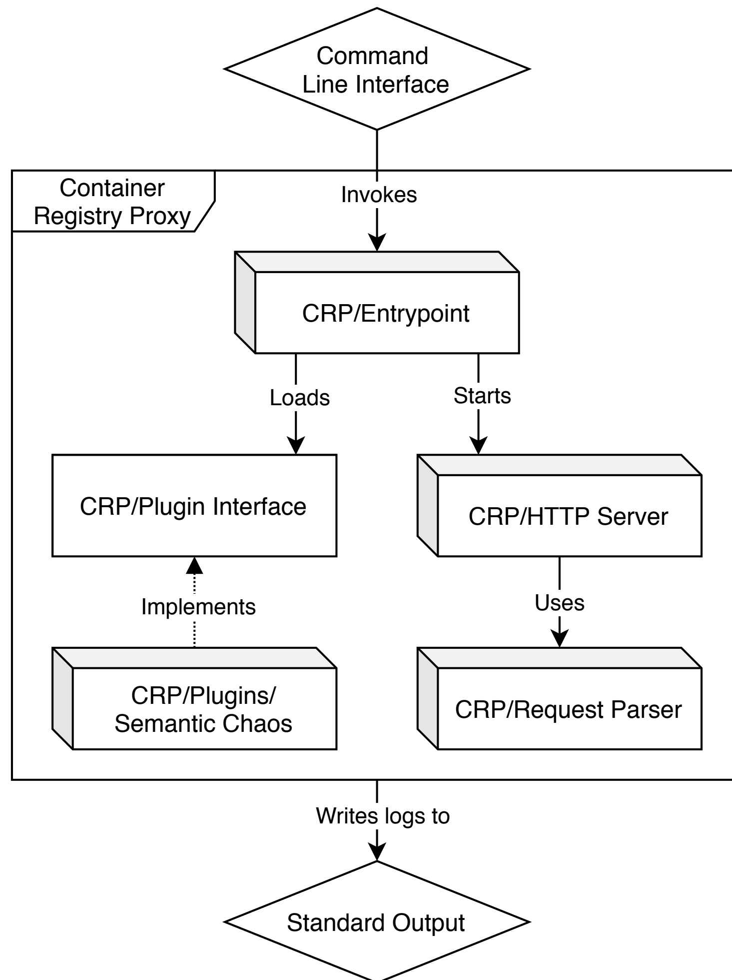


Figure 4.2: High level component diagram of the Container Registry Proxy.

4.3 Container Image Distribution Perturbation

This section assumes the reader understands the mechanisms behind distributing containers, which has been introduced in subsection 2.5.2: Container Image Distribution.

In order to perturb the distribution of containers it suffices to modify the HTTP traffic from and to the container registry. Modifying HTTP traffic can be done by routing all traffic through a proxy. This thesis is focused on modifying the first request, which resolves a tag to a digest, as later requests only deal with digest and hence cannot be easily modified. During the first request we have full control over the image to be returned.

The Semantic Chaos plugin developed for the Container Registry Proxy modifies the process of resolving tags to digests. An example of the request flow for a modified request is visualized in Figure 4.3.

When a request arrives at the CRP, it is handed over to the Semantic Chaos plugin before it is forwarded to the container registry. The Semantic Chaos plugin retrieves all tags available at the container registry for the image specified in the original request. These tags are filtered such that merely the tags remain which are compatible according to semantic versioning. One of these compatible tags is picked at random. The request is forwarded to the container registry with the original tag substituted with the one selected by the plugin. As such, the result from the container registry will be the digest for our newly selected tag, which is transparently forwarded to our client. The client now believes that the digest corresponding with the requested tag is in fact the digest of the tag selected by the Semantic Chaos plugin. In subsequent requests, the client will use the image of the digest of the tag selected by the plugin.

The chaos caused by the Semantic Chaos plugin originates from this plugin replacing the obtained image with the image of a compatible tag picked at random. Current container image registries resolve a tag to one version of the image. This relationship is mutable, hence over time a tag can be overwritten and refer to different versions. Consequently, requests made at exactly the same time will obtain the same version of the image. In practice, the mutability of tags could cause multi-version deployments. However, often times tags are only sporadically mutated, hence multi-version deployments occur infrequently and makes errors occurring because of version incompatibilities hard to reproduce. By intentionally causing these multi-version deployments to occur frequently we reap the benefits of chaos engineering. It establishes certainty that the application is able to function in multi-version deployments.

In the process of replacing the version neither the agent requesting the

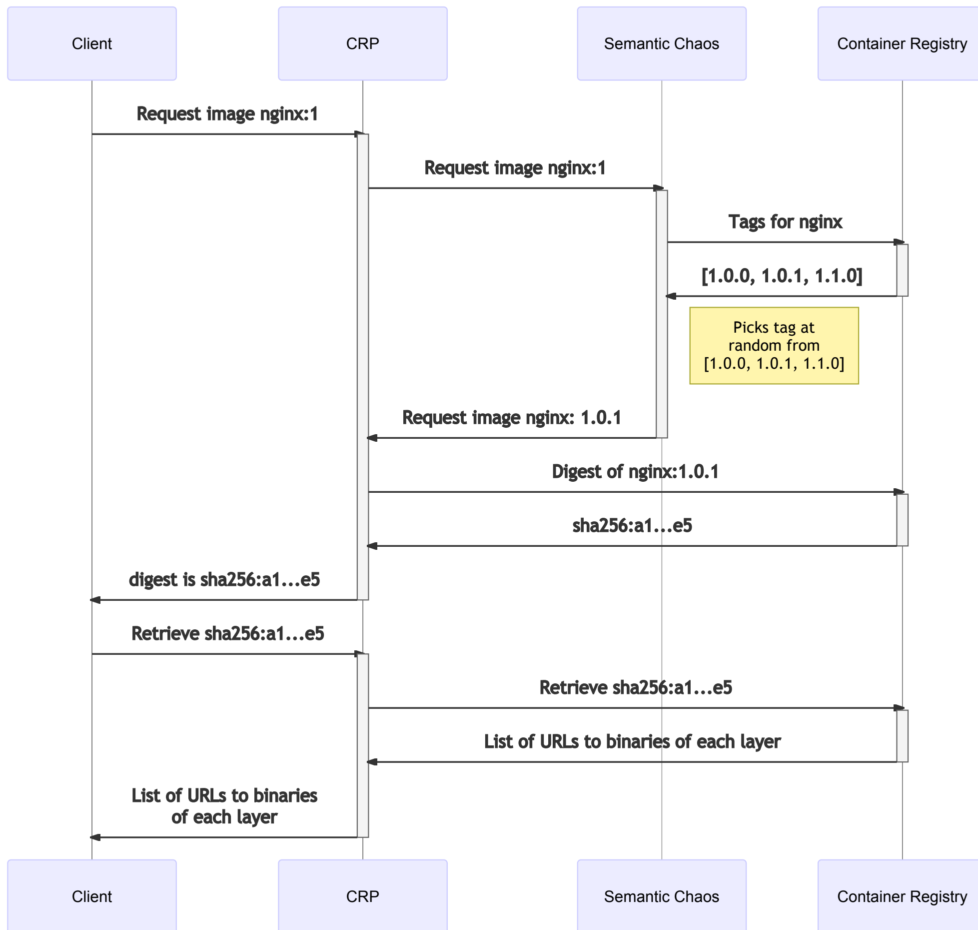


Figure 4.3: Sequence diagram of the request flow when retrieving an image through the Container Registry Proxy with the Semantic Chaos plugin enabled.

image nor the container registry serving the request is aware that the version was substituted. All interactions with the CRP remains compliant with the interface exposed by the container registry. Hence, no modifications to either the agent or the container registry need to be made in order to support the usage of the CRP, which allows adopting the CRP in a wide range of different use-cases and environments.

4.4 Monitoring Perturbations

Monitoring the actions taken by CRP plugins enabled one to observe the perturbations made to the process of container image distribution. Besides knowing what changed it is also relevant to know the rationale behind why a change is made. Hence, the CRP advocates for all plugins to use logging to record the changes they make, including supporting details to indicate why a plugin made a perturbation.

The Semantic Chaos plugin tracks the following properties for each of the requests to show why a perturbation was or was not made:

Timestamp: The timestamp when the request was made.

Requested tag: The tag which was part of the original incoming request.

Potential tags: A list of all tags which would be a candidate to replace the requested tag.

Selected tag: One of the tags from the potential tags which replaces the requested tag in the forwarded request.

4.5 Usage

To adopt the Container Registry Proxy in an existing setup, two steps need to be taken. The first step is deploying the CRP, which will be a separate service. Deployment instructions for the CRP are bundled with the source code. Supported deployment methods include running it as an individual container and running it as a Node.JS application. The CRP does not need to be deployed on the same infrastructure as where the target registry or containing consuming clients reside. The only requirement is that the network topology needs to be such that clients can access the proxy and the proxy can access the target registry. For publicly accessible registries, it would suffice to host the CRP as a publicly accessible service on any cloud or private infrastructure provider.

When deploying the CRP one can specify which plugins need to be enabled. Without any plugins, the CRP will not modify requests and merely forward them.

The second step is to modify the configuration of existing clients to retrieve their containers from our CRP instance instead of directly from the registry. Container images retrieved from registries are referenced by prefixing the hostname of the registry with the desired container image identifier. Updating the configuration by prefixing the image identifiers instead with the hostname of the CRP instance suffices to let the container image distribution process use the CRP instead of directly contacting the registry. One can choose to gradually adopt the CRP by only updating the configuration for a subset of their deployed containers.

4.6 Implementation

The CRP is implemented in TypeScript and consists of 775 lines of code. From the total code-base 288 lines (29%) are used for unit tests, which are written using the Jest[1] testing framework. The TypeScript source code is compiled to JavaScript and distributed as a container image on Docker Hub and package onto the npm Registry. The project targets to run on the current Node.js active long-term support release, which is version 12, although newer releases from version 13 and 14 pass our automated tests as well. The Semantic Chaos plugin is shipped as part of the distributed executables and hence available on every deployment. For plugins not part of the distributed executables it is still possible to include them by configuring the CRP with the path to this custom plugin.

Most of the CRP uses build-in functionality from Node.js, for example the HTTP server functionality uses the Node.js HTTP Server class. Two external packages are required as dependencies. Axios[2] for making HTTP requests, such as the semantic chaos plugin receiving all available tags from the registry. The other dependency is Commander[3], a library that simplifies building command line tools, which is used to parse the command line arguments received during startup and document the available command line arguments.

The source code and usage instructions of the Container Registry Proxy is released under the MIT license. A complete list of the different distributions of the CRP, including where to retrieve the source code, is shown in Table 4.1.

[1] https://jestjs.io/

[2] https://www.npmjs.com/package/axios

[3] https://www.npmjs.com/package/commander

4.7 Plugins Available for the CRP

The Container Registry Proxy differentiates between two types of plugins, namely build-in plugins and custom plugins. The differentiating factor between the two is that build-in plugins are included in the source code and hence available for every distribution of the CRP. Whereas custom plugins need to be readable by the CRP as external files.

Currently, one build-in plugin is shipped with the CRP, namely the Semantic Chaos plugin. The Semantic Chaos plugin perturbs the container image distribution process for images following semantic versioning for their tags. With this plugin it is possible to specify a range of tags, for which the returned image is picked at random from the tags in this range. The details of the workings of the Semantic Chaos plugin and how it affects the container image distribution process is discussed in section 4.3.

Custom plugins are plain JavaScript files exporting an object containing the logic of the plugin. These are loaded in by the CRP at runtime from the file system. The CRP contains type hints for plugins, which can be used to aid development of custom plugins. An example custom plugin is released[4] as to illustrate the process of creating custom plugins.

4.8 Summary

This chapter introduced the Container Registry Proxy (CRP), our novel tool to perturb traffic from and going to container image registries. The CRP comes bundled with the Semantic Chaos plugin, a plugin to perturb the process of retrieving images by returning any compatible version rather than always the exact requested version. We discussed the design principles behind the CRP, how it is implemented and how it can be used for chaos engineering purposes using this Semantic Chaos plugin.

[4] https://github.com/Addono/container-registry-proxy-custom-plugin-example

Artifact Type	**Platform**	**URL**	**Description**
Source Code	Github	https://github.com/ Addono/container-registry-proxy	Git based version control and release tracker for the source code of the CRP.
Container Images	Docker Hub	https://hub.docker.com/ addono/container-registry-proxy	Container images containing the CRP as a command line tool.
Node.js Package	npm	https://www.npmjs.com/ package/container-registry-proxy	Package for Node.js for using the CRP as a command line tool or adding it as a dependency for the development of plugins.

Table 4.1: List of sources where the official development artifacts of the CRP are accessible.

Chapter 5

Experimental Evaluation Protocols

5.1 Applications

This section introduces the applications under test we will be using to evaluate the effectiveness of the CRP (which was introduced in chapter 4) for container image distribution perturbation. The applications under test are selected using the following criteria:

Semantic versioning: Our application under tests needs to use semantic versioning to version their dependencies. Because our container image distribution perturbation plugin for the CRP, the Semantic Chaos plugin, only supports semantic versioning. In practice, we observed that most applications who have their versioning mechanism well-defined are using semantic versioning.

Open source: We opt for open source applications under permissive licenses for two reasons. Firstly, using open source software for our application under test enables others to have access to the same applications when replicating our results. Secondly, open-source projects tend to communicate and document more publicly, which eases understanding their release policy and allows us to potentially more easily triage encountered inconsistencies.

Widely used: To show that the obtained results are meaningful is it required for the application under test to be widely used. We quantify "widely

used" as an application with over a billion container image pulls on Docker Hub[1].

Given these criteria we selected PostgreSQL and Redis as our applications under test. Both applications will be introduced in detail in the remainder of this section.

5.1.1 PostgreSQL

The first application under test we will be using for our experiments is PostgreSQL, which is also known as Postgres. PostgreSQL is one of the two leading open source relational database management systems (RDBMS). To date, many companies worldwide are using PostgreSQL in production [13]. PostgreSQL is chosen over the other vastly popular RDBMS MySQL, as PostgreSQL finds its origin in the academic world at the University of California, Berkeley.

As to match high-availability and high-performance scenarios, PostgreSQL will be deployed in a distributed fashion as a cluster with active automated fail-over mechanisms enabled. We will be deploying the `postgres-ha` Helm chart[2] maintained by Bitnami, as their library of open-source deployment charts are well maintained and cover a wide range of well known open-source projects. Other similar deployment charts are the PostgreSQL operators by Zalando[3] and CruncyData[4], both come with extensive support for additional features which aid maintaining the PostgreSQL deployment, such as automated backups. Each of these deployment mechanisms slightly differs in the architecture of the deployment, as there are multiple tools and configuration options to achieve data replication and master election within the cluster.

In the process of selecting the deployment method used in our experiments we require that the used images are semantically versioned, as this is a prerequisite for using the Semantic Chaos plugin, which eliminates the operator made by Zalando. The CrunchyData operator does not include support for deployments using Helm, instead the operator imperatively builds the deployment configuration. This additional layer makes it harder to comprehend and be certain about what is going to be deployed. Therefore we chose the Bitnami charts for creating the deployments under test in our experiments.

[1] `https://hub.docker.com`

[2] `https://github.com/bitnami/charts/tree/master/bitnami/postgresql-ha`

[3] `https://github.com/zalando/postgres-operator`

[4] `https://github.com/CrunchyData/postgres-operator`

Bitnami's chart for redundant PostgreSQL deployments uses container images build by Bitnami. The PostgreSQL image is bundled with a tool called `repmgr`, which is responsible for replicating the data between the different instances of PostgreSQL. Such replication manager tool is necessary for redundant deployments, since it keeps the data of the different nodes synchronized.

The workload[5] used to validate the working of the cluster comprises of a series of simple queries which create a new table, insert a row into this new table and attempts to read it. Intentionally this workload avoids new cutting edge features, as to make sure that all features used are available in all versions under test.

5.1.2 Redis

The other application under test is Redis[6], a distributed in-memory key-value store, which is widely used in academic research and enterprise environments [7]. Redis differs from traditional relational database management systems as it values performance over persistence and has a different approach to retrieving data. Data is read and written primarily to the in-memory datastore, which can be configured for persistence by writing to disk. Writing the changes made to the in-memory data back to disk is done asynchronously on a regular interval, which improves performance as operations do not depend on disk access at the cost of potentially losing recently modified data on system failure.

Redis offers multiple forms of data replication, either using a single master or multi-master. In a configuration with a single master all writes go through this single master node while additional slave nodes offload the master. The other option is to create a so-called Redis Cluster. In a Redis Cluster there are multiple masters each with their own slave nodes. Communication between these masters is done in a completely decentralized setup using the Gossip protocol [7].

For our experiments we will be using a Redis deployment using the `redis` Helm chart[7] provided by Bitnami. Which is configured to deploy three Redis instances, one acting as master while the other two are slave nodes. The workload used to validate the functioning of the Redis deployment consists of writing and reading a random value. This random value is first written to

[5] https://github.com/Addono/thesis-experiment/blob/main/evaluation-tool/postgresql-ha/load.sql

[6] https://redis.io/

[7] https://github.com/bitnami/charts/tree/master/bitnami/redis

the master node and then read from the node under test. A node is considered functional if the read value matches the value which was written.

5.2 Experimental Setup

For the experiment any platform to run and orchestrate containers could be used. This experiment uses Kubernetes as the container orchestration platform, because of its widespread adoption. Many managed Kubernetes services exist, these experiments were conducted on clusters managed by DigitalOcean running Kubernetes version 1.18.6. The cluster consisted out of 3 nodes, each with one virtual CPU and 2 gigabytes of memory. Each deployment runs against a newly provisioned cluster, as later deployments would be able to benefit from images cached in earlier deployments.

The application under test is sourced from a community maintained deployment configuration, which is distributed as a Helm chart. These community maintained deployment charts are available for various well known and vastly used open-source applications. Hence making them realistic deployment configurations for real-world deployments.

When deploying the application all containers will be fetched through the Container Registry Proxy, except for the reference runs which directly fetches the containers from the original container registry. The Container Registry Proxy is launched with the Semantic Chaos plugin, which enables it to perturb the container image retrieval process. The proxy is deployed as a separate application outside the cluster on Heroku as a container, it is publicly accessible hence every node in our Kubernetes cluster can directly access the proxy.

Some minor modifications are made to the deployment configuration of our application. For one, the hostname of the container registry where the images are stored is replaced with the hostname where the Container Registry Proxy is hosted. In addition, the so called “pullPolicy” configuration variable is set to “Always”. This change ensures that every time a container is deployed it contacts the container registry to check if a newer version of the image is available. The default setting for the pull policy configuration would cache the results indefinitely and hence not hand over control to the CRP. It is not strictly necessary to modify this configuration value, as node rebuilds would also purge the cache, however invoking a node rebuild as a means to clear the cache would also affect all other applications deployed onto this node.

During the experiment we monitor the deployment process and test whether the deployment process is successful, yielding a deployed application which is functional. With regard to the deployment process will we be looking into

how long it takes for the deployment to succeed and how often parts of the deployment were retried. For a deployment to be considered successful, it is required that all containers specified by the deployment are running without errors and the application can handle a test workload.

The process of running the experiments is automated. The code which was is used to run the experiments is publicly accessible at `https://github.com/Addono/thesis-experiment`.

5.3 Perturbations

The experiment is executed multiple times, each with a varying degree of what version ranges are considered compatible. Different ranges of version compatibility are based on semantic versioning, which was introduced in section 2.4. This yields the following test cases which will be evaluated during the experiment:

Reference run: During the reference run the CRP it not used. This gives a base point to compare deployments using the CRP with deployments without using the CRP.

Perturbations disabled: The tests with perturbation disabled is using the CRP with the Semantic Chaos plugin enabled, however the default version requested by the deployment will also be the version which is deployed.

Perturb build version: Only the specific build of a version are perturbed. Thus all containers of one image will be running the same patch release, but potentially different builds. For example, different images can be build for the same release when different base images are used.

Perturb build and patch versions: When in addition to the build also the patch version is perturb, then also different releases will be mixed.

Perturb build, patch and minor versions: Varying the build, patch and minor version gives the maximum amount of freedom to the Semantic Chaos plugin while still operating within the compatibility guarantees from semantic versioning. When perturbing minor versions, caution needs to be taken as to avoid usage of features newly introduced in one of these minor releases. As for such features are not required to be backwards compatible. In order to ensure none of the newly introduced features are used, the workload to verify the application with is selected based on the lowest minor version under test.

The Container Registry Proxy, which is introduced in chapter 4: Design & Implementation of the CRP, is used with the Semantic Chaos plugin to inject the perturbations during the experiments. Version 4.4.6 of the CRP are used during the experiments. The CRP is hosted on the Heroku cloud platform on a node with 512 MB of RAM and a shared CPU.

5.4 Metrics

In order to evaluate the resilience of our application against multi-version deployments will we observe the deployment process. The following metrics will be collected during the deployment of the application:

Time to Complete Initialization (TCI): The amount of time until all containers part of the deployment report that they are initialized and available to receive traffic.

Time to First Request (TFR): Measures the amount of time needed for the deployment until it successfully processes its first workload request. The time is measured from the moment the deployment process initiated.

Time to All Containers Handle Requests (TACHR): The amount of time until all containers able to handle requests successfully handled one request. TFR is a more relaxed version of this metric, as TFR is satisfied when one of the request handling containers is operational, whereas TACHR demands that all containers are operational.

Amount of Container Restarts (ACR): Malfunctioning containers are restarted by Kubernetes. The reason why a container malfunctions can be manifold and can occur both when starting the application and at runtime.

The collected metrics will be used to answer research question 2 and 3, which were introduced in section 1.3: Research Questions. RQ2 focuses on the effectiveness of our practices of applying chaos engineering with multi-version deployments to detect version incompatibilities. Longer times to deploy and more frequent container restarts when introducing our perturbations suggest version incompatibility. To answer RQ3, increases time to deploy observed between the reference runs and tests with perturbations give insight in the overhead introduced by the CRP.

Experiment	PostgreSQL	Redis	Total
Reference Run	50	50	100
Perturbation Disabled	50	50	100
Vary Build	15	15	30
Vary Patch	15	15	30
Vary Minor	15	N/A	15
Total	145	130	275

Table 5.1: Amount of times each of the experiments was executed for each of the applications under test.

5.5 Versions Under Test

The specific versions of the Helm charts and container images used in each of the experiments can be found in Table 5.2. The three most recent major releases of Redis, namely 4, 5 and 6, do not feature minor releases. Hence the experiment to vary the minor release would effectively be equivalent to varying the patch version, therefore this part of the experiment will be omitted for Redis.

5.6 Iteration Count

The amount of times each of the experiments is executed can be found in Table 5.1. The exact amount of times each of the experiments is repeated depends on the experiment type. Each of the control and perturbations disabled tests are executed 50 times. With two applications under test and two experiment types then yield a total of $2 \cdot 2 \cdot 50 = 200$ deployments without the perturbation engine enabled.

Experiments with the perturbation engine enabled are executed 15 times for each of the three experiment type and two application under test. Fewer deployments are made for each of the experiments with perturbation enabled compared to the experiments without perturbation, because the expected performance impact of the CRP tested in the latter set of tests is substantially lower than the expected variation in the metrics caused by perturbing the deployment. Given that the "Vary Minor" experiment is excluded for Redis, as discussed in section 5.5: Versions Under Test, add the experiments with perturbation another $(3 + 2) \cdot 15 = 75$ experiment runs.

5.7 Summary

This chapter summarized how we will be evaluating the performance and effectiveness of the Container Registry Proxy with the Semantic Chaos plugin. PostgreSQL and Redis are selected as the applications under test, which will be deployed multiple times with varying degrees of perturbation freedom for the Semantic Chaos plugin. For each of these deployments we will collect various metrics on the time it takes for the deployment to be able to handle requests and how many times containers needed to be restarted. The results of our experimental evaluation are presented in chapter 6: Experimental Results.

Application Under Test	Helm Chart	Container Name	Container Version Range			
			Control Test & Perturbations Disabled	Vary Build	Vary Patch	Vary Minor
PostgreSQL	bitnami/postgresql-ha:3.2.7	bitnami/postgresql-repmgr	11.8.0-debian-10-r13	11.8.0	11.8	11
		bitnami/pgpool	4.1.2-debian-10-r5	4.1.2	4.1	4
Redis	bitnami/redis:10.7.16	bitnami/redis	6.0.6-debian-10-r10	6.0.6	6.0	N/A

Table 5.2: The versions under test for each of the experiment types.

Chapter 6

Experimental Results

This chapter presents the results from running the experiments. First it breaks down the collected metrics per application under test. Then section 6.3 compares the test results from the experiments with and without the Container Registry Proxy, as to give insight into the impact of the CRP on the deployment performance metrics.

6.1 Application Evaluations

The collected metrics (introduced in section 5.4: Metrics) focus on detecting whether faults during the deployment occur, however they do not give insight into the cause of these faults. Therefore, if there were major deviations, then the deployments were manually inspected in an attempt to diagnose the cause. This section looks at the obtained results for each of the applications under test and covers the insights gained from manual inspection.

6.1.1 PostgreSQL

The deployment time metrics for PostgreSQL during the reference runs and perturbation disabled tests are visualized in Figure 6.1. The boxplots shows the range between the lower and upper quartile with colored boxes, where the line dividing these boxes is the median. The whiskers of the boxplots denote a confidence interval, which considers all points more than 1.5 times the interquartile range - the distance between the lower and upper quartile - from the lower and upper quartile as outliers. These outliers are rendered as diamonds outside the area covered by the whiskers. Besides the boxplot the figure shows each of the attained test points using a semi-transparent scatterplot. Similar as

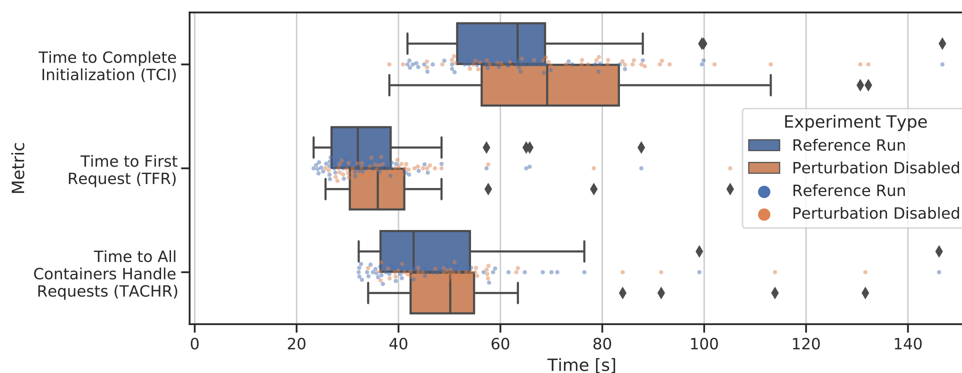


Figure 6.1: Comparing the results of the reference runs versus the results for the perturbation disabled tests for PostgreSQL.

to what is concluded in section 6.3 is also the median of each of the metrics higher during the tests with perturbation disabled compared to the reference runs.

Figure 6.2 shows individual plots for each of the metrics. These plots all render the data similarly as previously discussed for Figure 6.1. Instead of denoting the different experiments using color are these now placed on the y-axis. The coloring is instead used to denote the application under test.

Figure 6.2b and Figure 6.2c show that the median of the TACHR and TCI metric respectively doubles for PostgreSQL when the perturbation engine varies the patch level compared to the case where the perturbation engine is disabled. The results from these figures show an even stronger effect for the case where the perturbation engine also disturbs the deployed minor version, as the median metric more than quadruples. No such increase is seen for perturbations on the deployed build version.

Welch's t-test is applied as to establish the statistical significance of the increase of the metrics in the tests perturbing the patch and minor version. The result of Welch's t-test comparing the results from the perturbations disabled test with the experiments perturbing up till the build or patch level are shown in Table 6.2. This test has as its null hypothesis H_0 that the mean of the perturbation disabled experiments is the same as the mean of the experiments with perturbation enabled. The alternative hypothesis H_1 states that the means of these two data sets differs. For all metrics the resulting p-value < 0.05, hence we can conclude that the vary patch and vary minor tests have different means than the tests with perturbation disabled. The p-value for the experiments which merely perturb the build version do not have a p-value smaller than 0.05, hence confirming the observation from the plots that these results

	Vary Build	Vary Patch	Vary Minor
Time to All Containers Handle Requests (TACHR) [s]	0.241434	7.9298e-07	1.8536e-10
Time to Complete Initialization (TCI) [s]	0.996325	2.7714e-08	4.0639e-09
Time to First Request (TFR) [s]	0.540063	6.3000e-04	7.2377e-03

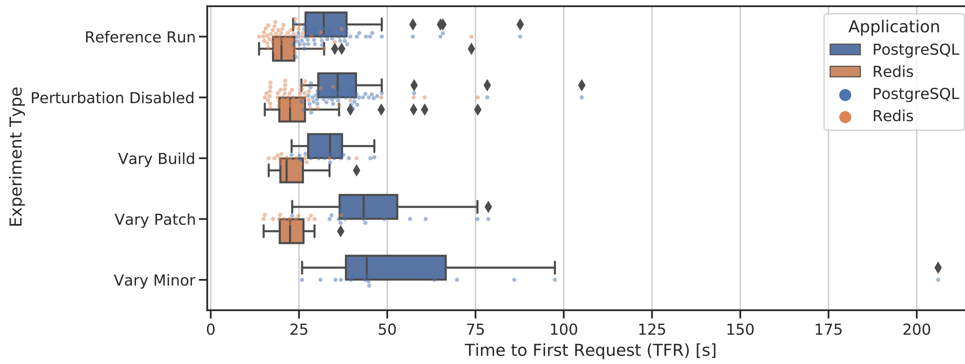
Table 6.1: The p-value for Welch's t-test for PostgreSQLevaluating whether the mean of the perturbation disabled test differs from the experiments with the perturbation engine disabled.

seem similar to the perturbation disabled test.

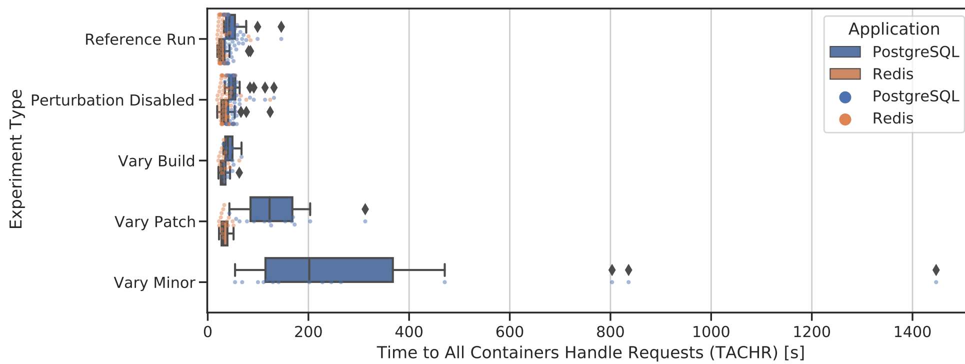
6.1.2 Redis

Figure 6.3 plots the results for the reference run and perturbation disabled test for Redis. The first quartile, median and third quartile are denoted by respectively the left side, middle line and right side of each of the colored boxes. For each of the metrics the first quartile, median and third quartile is higher during the test with the perturbation engine disabled compared to the reference runs. A confidence range of 1.5 times the inter-quartile range is denoted by the whiskers on either side of the colored boxes.

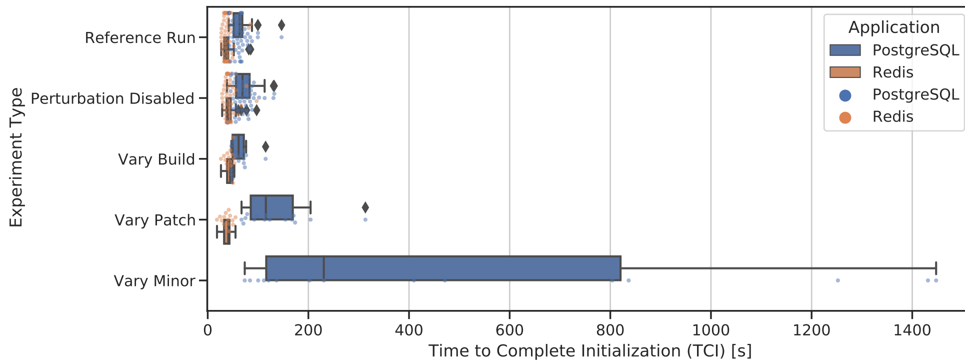
Plots in Figure 6.2 include the results for the experiments with the perturbation engine enabled. Figure 6.2d shows that none of the deployments, both with and without perturbation, encountered containers which needed to be restarted. In addition, all other metrics seem to only differ very little between the perturbation disabled test and all tests with perturbation enabled. A statistical significance test is conducted to verify that perturbations did not vastly influence these metrics. The result of Welch's t-test comparing the results from the perturbations disabled test with the experiments perturbing up till the build or patch level are shown in Table 6.2. This test has as its null hypothesis H_0 that the mean of the perturbation disabled experiments is the same as the mean of the experiments with perturbation enabled. The alternative hypothesis H_1 states that the means of these two data sets differs. None of the resulting p-values is smaller than 0.05, hence we can not reject the null hypothesis and hence do not have proof that the perturbations significantly impacted the metrics.



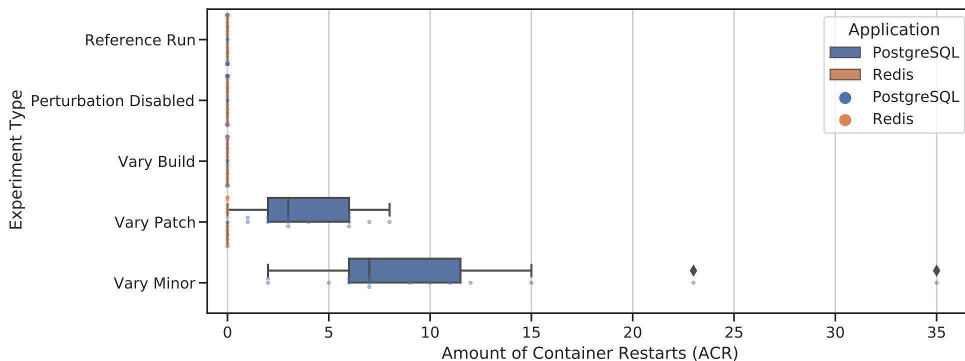
(a) Plot showing the distribution for the Time to First Request (TFR) metric.



(b) Plot showing the distribution for the Time to All Containers Handle Requests (TACHR) metric.



(c) Plot showing the distribution for the Time to Complete Initialization (TCI) metric.



(d) Plot showing the distribution for the Amount of Container Restarts (ACR) metric.

Figure 6.2: Plots for each of the collected metrics by experiment type and application under test.

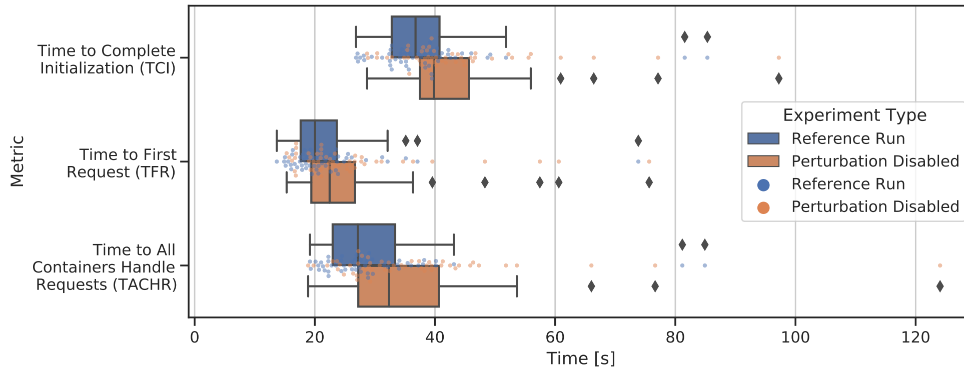


Figure 6.3: Comparing the results of the reference runs versus the results for the perturbation disabled tests for Redis.

	Vary Build	**Vary Patch**
Time to All Containers Handle Requests (TACHR) [s]	0.515164	0.355291
Time to Complete Initialization (TCI) [s]	0.179562	0.896623
Time to First Request (TFR) [s]	0.514524	0.658152

Table 6.2: The p-value for Welch's t-test for Redis evaluating whether the mean of the perturbation disabled test differs from the experiments with the perturbation engine disabled.

6.2 Detected Defects

This section goes over the defects detected by using the Semantic Chaos plugin to perturb deployments of PostgreSQL and Redis.

6.2.1 PostgreSQL

Figure 6.2d shows that none of the experiments encountered containers needing to be restarted, except for PostgreSQL deployments with perturbations on patch or minor level. A manual inspection on the logs of the terminated containers is performed, as to find an explanation which caused the restart. The logs reveal that the container restarts originated from a version incompatibility between the data replication manager. The PostgreSQL image contains both PostgreSQL and the data replication manager `repmgr`. During one of the patch releases of the PostgreSQL image, the bundled version of the replication manager was changed from 5.0 to 5.1. As part of the initialization process of the PostgreSQL image is letting the replication manager connect to the other running containers. When it is connecting to the other containers it checks if its own version of the replication manager is compatible with the one running in the other container. If these versions are not compatible, then the initialization is terminated and the container exits unsuccessfully.

After a container is terminated it is automatically recreated as part of the default error recovery mechanisms of Kubernetes. The newly created container retrieves a random image from the CRP. This container will crash and be recreated until the container image it retrieves through the CRP contains a version of the replication manager compatible with what is already running in the cluster.

Experiments which also perturbed the deployed minor version observed older versions of the replication manager, down to version 4.4. Thus, even if the replication manager would adhere strictly to semantic versioning, then upgrading it to a new major version should have caused a new major release for the PostgreSQL image bundling `repmgr`.

6.2.2 Redis

In subsection 6.1.2 it is observed that there is no significant effect on the deployment metrics with different levels of perturbation intensity. Hence, there is no evidence in our metrics which suggest that the multi-version deployment of Redis encounter faults not present in version wise homogeneous deploy-

	Time to All Containers Handle Requests (TACHR)		Time to First Request (TFR)		Time to Complete Initialization (TCI)	
	Postgres	**Redis**	**Postgres**	**Redis**	**Postgres**	**Redis**
Reference Run [s]	48.75	30.16	35.34	22.09	64.15	38.65
Perturbation Disabled [s]	52.68	36.68	38.48	26.02	71.76	43.25
Absolute Increase [s]	3.92	6.53	3.15	3.93	7.61	4.61
Relative Increase	8.05%	21.64%	8.91%	17.77%	11.87%	11.92%

Table 6.3: The increase of the mean response or initialization time introduced when employing the Container Registry Proxy. Absolute and relative increases regard the difference between the reference tests and perturbation disabled tests.

ments. This is not to say that Redis does not contain any faults, merely that our workload did not trigger them, neither with and without perturbation into multi-version deployments.

6.3 Performance Impact of the CRP

The two experiment types with the perturbation engine disabled or omitted from the experiment setup isolate the effect of the Container Registry Proxy and the Semantic Chaos plugin. Specifically, the "Perturbation Disabled" tests uses the CRP with Semantic Chaos plugin, however it is configured such that it will not perturb the container image distribution process. During the "Reference Run" the CRP is not used and instead images are retrieved directly from the container registry.

Table 6.3 displays the increase in the mean response and initialization time when using the Container Registry Proxy. For each of the metrics and applications under test this mean increases. The absolute increase in deploy time metrics ranges 3.15 to 7.61 seconds, whereas relative increases are between 8% to 21%. Comparing the applications under test shows that the relative increase in the mean of each of the metrics for Redis is higher than for PostgreSQL.

	PostgreSQL	**Redis**
Time to All Containers Handle Requests (TACHR) [s]	0.300372	0.030039
Time to Complete Initialization (TCI) [s]	0.053723	0.048184
Time to First Request (TFR) [s]	0.219115	0.066115

Table 6.4: The individual p-value for Welch's t-test with the hypothesis that the mean of the Reference Run is equal to the mean of the Perturbation Disabled experiment.

As to establish the statistical significance of this increase, a Welch's t-test is conducted on each of the metrics and applications under test. The null hypothesis H_0 for this test assumes that the mean is the same between the Reference Run and Perturbation Disabled, while the alternative hypothesis H_1 asserts that the mean is different. Table 6.4 shows the individual p-values for Welch's t-tests between each combination of deploy time metric and application under test. For Redis, two out of the three metrics achieve statistical significance with a p-value smaller than 0.05. PostgreSQL does not achieve statistical significance on any of the individual metrics.

6.4 Summary

This chapter presented the results from running the evaluation experiment, which was introduced in chapter 5. The performance impact of the Container Registry Proxy increased the deploy time metrics on average between 8% and 22%.

Perturbing the deployments of the applications under test had a significant impact on PostgreSQL, however none was observed for Redis. Different patch versions of PostgreSQL showed to be incompatible and as a result caused containers to fail to initialize. This version incompatibility issue was traced down to an application bundled into the PostgreSQL container image. The versions of this bundled application could differ minor or major versions between different PostgreSQL images. Deploying containers with a minor or major difference of this bundled application would cause the container to terminate as part of the initialization process.

Chapter 7

Discussion

7.1 Perturbation Possibilities for Container Deployments

A method to perturb the deployment mechanism of container based application deployments into multi-version deployments is by targeting the container image distribution process. The container image distribution protocol is based on HTTP and is standardized by the Open Container Initiative (OCI) of The Linux Foundation. Targeting the image distribution protocol makes the approach highly generic and container orchestration platform agnostic.

The Container Registry Proxy (CRP), introduced in chapter 4: Design & Implementation of the CRP, is a working example of how perturbing the container image distribution process can be used for chaos engineering purposes. With the CRP it is possible to both monitor and modify requests made to a container image registry. Perturbing the container image distribution process by means of deploying a proxy in front an existing container registry offers some key advantages. For one, the solution can be used with existing container registries, which is crucial in environments where the container registry cannot be modified, as for example would be the case when it is hosted by a third-party. By extension this makes it agnostic of the specific implementation used by the container registry, as it merely requires it to adhere the standard set by the OCI. Secondly, by adding a proxy one maintains a single source of truth shared between applications deployed with and without chaos engineering enabled. Lastly, adopting chaos engineering in the container image protocol through a proxy reduces the impact to an existing deployment to merely a configuration change of a single value.

The Semantic Chaos plugin was developed for the CRP, which perturbs the

container image distribution process as to create multi-version deployments in a controlled fashion. During our experiments we successfully used the CRP with the Semantic Chaos plugin to deploy functional multi-versions of both Redis and PostgreSQL a total of 175 times. This shows that perturbing the container image distribution process by using a proxy is both viable and practical.

7.2 Detecting Version Incompatibility using Chaos Engineering

Version incompatibility issues in software application deployments emerge when different versions of the same application running in parallel cause faults. Systematic versioning of applications can be done such that the versioning conveys intent about which versions are able to parallel. One of these versioning methods is semantic versioning, which is widely adopted.

This thesis explored the possibility of using chaos engineering to perturb the versions running in the deployment of a distributed application. These deployments then consist of multiple versions of the same application, whereas normally such deployment would be homogeneous. The perturbation engine developed for this research, namely the Semantic Chaos plugin for the Container Registry Proxy which was introduced in chapter 4, allows the creation of such multi-version deployment while respecting version compatibility according to semantic versioning.

The effectiveness of detecting version incompatibility issues using chaos engineering with the Semantic Chaos plugin was validated by using it to create numerous multi-version deployments for two widely used databases. These two applications under test comprise PostgreSQL and Redis. Various metrics were collected for each of the deployments, measuring the time it takes for the application deployment to become functional and the amount of container restarts required to get the application to a functional state.

From the results of the experiments emerged a pattern where the PostgreSQL deployments saw a vast increase in all metrics with increasing freedom for the perturbation engine, whilst no such pattern emerged for Redis. Manual inspection of the logs revealed that a build-in version check of an application bundled with the PostgreSQL application terminated containers when certain multi-version environments emerged. The version of this bundled tool was updated without this backwards compatibility breaking change being reflected in the version of the container image it was bundled into. As

a result, the multi-version deployments revealed a violation of the versioning policy of the PostgreSQL image offered by Bitnami.

Redis on the contrary did not see a similar trend as PostgreSQL. Results for all metrics stayed relatively similar and none of the container deployments failed. Therefore, Redis seems to adhere to its versioning policy and seems suitable for multi-version deployments.

7.3 Runtime Overhead of the Perturbations

The overhead introduced by adopting the Container Registry Proxy with the Semantic Chaos plugin is twofold. On the one hand there is the cost of running the Container Registry Proxy. During the experiments the CRP was deployed on the smallest node available by Heroku, which gave it merely 512 MB of memory. With the CRP being able to run on such small amount of resources makes its resource usage insignificant compared to the applications intended to be deployed with the CRP.

The low amount of resources needed by the CRP can be attributed to the fact that it is merely a simple HTTP proxy. Requests are modified in-flight and handed over to the target container image repository, which handles more complex issues like access control and storing container images.

On the other hand, there is cost involved by using the CRP, because it slows down deployments by increasing the time it takes to retrieve container images. Retrieving container images is slowed down because it adds another hop and stalls ongoing requests when the Semantic Chaos plugin is retrieving all available tags. The effect of adding the CRP to a deployment is visible when comparing the Reference Runs with the Perturbation Disabled tests. These tests only differ on the usage of the CRP. The Reference Run directly connects to the container image registry, whereas the Perturbation Disabled test will use the CRP with Semantic Chaos plugin. For the latter test the container images requested to the CRP are so specific that according to semantic versioning the Semantic Chaos plugin is not able to perturb the returned images.

Adding the CRP resulted in an increase in mean response time between 8% and 22%, varying between application under test and the type of metrics measured. The statistical confidence of equal mean deployment time of all tests combined results in a p-value of 0.002994, which gives a strong confidence that the CRP in-fact caused the mean of the deployment time metrics to increase. This effect was particularly strong for Redis as the application under test.

The direct effect of adding a CRP to a deployment configuration are insignificant compared to the impact of perturbing deployments with version incompatibility issues. Perturbing the PostgreSQL deployment can easily cause the deployment metrics to increase more than the 22% seen without perturbing. For the TACHR and TCI metric the average doubled and quadrupled when perturbing up to respectively the patch and the minor level.

Chapter 8

Conclusions

This chapter starts with a summary of the work presented in this thesis. Then potential future work for future research is suggested.

8.1 Recapitulation

This thesis explored how chaos engineering practices can be used to create multi-version deployments for applications deployed as containers. The container image distribution protocol is HTTP based and resolves a container image identifier in a couple of requests to the container registry into the binaries of this image. Perturbing the container deployment mechanism into a multi-version deployment (RQ1) can be achieved by manipulating how container image identifiers are resolved to the binaries by the container registry. For this purpose we developed the Container Registry Proxy (CRP), a tool agnostic proxy server to observe and modify the container image distribution protocol.

By default, the proxy does not alter the behavior of the container registry, as to facilitate extending the functionality at the end-users discretion. A plugin system is added to the CRP, which allow controlling the modifications made to the traffic passing through the proxy. For our chaos engineering purposes we developed the Semantic Chaos plugin, which modifies the behavior of the container registry to serve an image at random from a range of versions deemed compatible by semantic versioning.

The effectiveness of the CRP with the Semantic Chaos plugin was validated through an experiment in which two open-source applications, PostgreSQL and Redis, were deployed various times. Specifically, RQ2 revolved around detecting version incompatibility issues - faults caused by running different versions of an application which are supposedly compatible. Major de-

viations of the deployment metrics in the PostgreSQL deployments were related to a version compatibility issue present in the specific distribution of the PostgreSQL image used during the experiment. No version incompatibility issues were found for Redis.

Another objective of the experiment was to measure the performance impact the CRP has on deploying applications, which answers RQ3. The experiments showed with high confidence that the CRP did increase the mean deployment time for some metrics. Overall, the observed metrics increased between 8% and 12% for Redis and 11% and 22% for PostgreSQL. The CRP merely impacts the container image distribution process, hence this additional overhead is limited merely to applications being deployed.

8.2 Future Work

Observing and modifying the container distribution process is an exciting new direction for research. Tools such as the Container Registry Proxy are generic enough to act as a starting point for future research endeavors.

The work presented in this thesis focuses on applying proxy practices to create multi-version deployments, however the concept of manipulating the container distribution process could also be used for other purposes. A proxy like the CRP could be used to prevent pulling certain images, for example to prevent images going into production which have known vulnerabilities or originate from unverified sources. Such additional security layer could be added to existing solutions without having to facilitate interactions with the specific implementations for the container registry or consumer of the container images.

The CRP could be extended with additional monitoring features, for example to facilitate usage analytics or satisfy auditing requirements. Data on when certain images are pulled could give insight into the amount of time it takes for applications to adopt a new version of a container image since it has been released.

To the best of our knowledge, all applications of the container distribution model use the container registry as a static artifact storage, which only supports pulling images already present at the container registry. With a proxy in front of the container registry it becomes possible to intercept this pull event and act upon it while temporarily stalling the container pull request.

For example, intercepting the pull event could be used to serve containers which are created or augmented on-demand. The proxy can make modifications to the requested container, e.g. injecting a license key specific to the user

requesting the image.

Many container images contain dependencies, when a container image is created these dependencies are pinned. This introduces technical lag, as newer versions of these dependencies are released with additional security and bug fixes, however the image will still bundle the original version. When intercepting the pull event, one could update the dependencies bundled within the container image to the latest version just before serving the image, which reduces technical lag [11].

Another use-case for augmenting container images would be to add tools to an existing image or create images on-demand. One could for example add additional tools for observability purposes, similar as what POBS [43] does for image build files for Java based applications. Being able to augment images on-demand simplifies the process of adopting augmented images, as it removes the need of creating all required augmented images in advance.

Our experimental evaluation was limited to observing deployments for two individual applications. For one, it would be interesting to attempt to reproduce the effectiveness of our work on applications other than databases. Also, it should be noted that the deployments evaluated during our experiments are relatively simple compared to distributed applications with many moving parts. Such more complex deployments could for example be found in a system using the micro-services architecture. There are micro-service based systems in production featuring thousands of services [37]. The micro-services architecture allows individual services to adopt their own versioning and release policy. Such vastly complex system would be an interesting candidate for more in-depth for multi-version experimentation.

Bibliography

[1] Ali Basiri et al. "Chaos Engineering". In: *IEEE Software* 33.3 (May 2016), pp. 35–41. ISSN: 0740-7459, 1937-4194. DOI: 10.1109/MS.2016.60. URL: https://ieeexplore.ieee.org/document/7436642/ (visited on 02/04/2020).

[2] Ali Basiri et al. *Chaos Engineering Upgraded*. Medium. Apr. 19, 2017. URL: https://netflixtechblog.com/chaos-engineering-upgraded-878d341f15fa (visited on 02/18/2020).

[3] Benoit Baudry and Martin Monperrus. "The Multiple Facets of Software Diversity: Recent Developments in Year 2000 and Beyond". In: *ACM Computing Surveys* 48.1 (Sept. 29, 2015), pp. 1–26. ISSN: 0360-0300, 1557-7341. DOI: 10.1145/2807593. URL: https://dl.acm.org/doi/10.1145/2807593 (visited on 09/02/2020).

[4] Netflix Technology Blog. *The Netflix Simian Army*. Medium. July 19, 2011. URL: https://netflixtechblog.com/the-netflix-simian-army-16e57fbab116 (visited on 09/29/2020).

[5] *chaos-mesh/chaos-mesh: A Chaos Engineering Platform for Kubernetes.* URL: https://github.com/chaos-mesh/chaos-mesh (visited on 09/29/2020).

[6] Feng Chen et al. "Multi-version Execution for the Dynamic Updating of Cloud Applications". In: *2015 IEEE 39th Annual Computer Software and Applications Conference*. 2015 IEEE 39th Annual Computer Software and Applications Conference (COMPSAC). Taichung, Taiwan: IEEE, July 2015, pp. 185–190. ISBN: 978-1-4673-6564-2. DOI: 10.1109/COMPSAC.2015.130. URL: http://ieeexplore.ieee.org/document/7273617/ (visited on 08/12/2020).

[7] Shanshan Chen et al. "Towards Scalable and Reliable In-Memory Storage System: A Case Study with Redis". In: *2016 IEEE Trustcom/BigDataSE/ISPA*. 2016 IEEE Trustcom/BigDataSE/ISPA. Tianjin, China: IEEE, Aug. 2016,

pp. 1660–1667. ISBN: 978-1-5090-3205-1. DOI: 10.1109/TrustCom.2016.0255. URL: http://ieeexplore.ieee.org/document/7847138/ (visited on 08/05/2020).

[8] Bart Coppens, Bjorn De Sutter, and Stijn Volckaert. "Multi-variant execution environments". In: *The Continuing Arms Race: Code-Reuse Attacks and Defenses*. Ed. by Per Larsen and Ahmad-Reza Sadeghi. ACM, Mar. 1, 2018, pp. 211–258. ISBN: 978-1-970001-83-9. DOI: 10.1145/3129743.3129752. URL: https://dl.acm.org/citation.cfm?id=3129752 (visited on 10/09/2020).

[9] Bradley E. Cossette and Robert J. Walker. "Seeking the ground truth: a retroactive study on the evolution and migration of software libraries". In: *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering - FSE '12*. the ACM SIGSOFT 20th International Symposium. Cary, North Carolina: ACM Press, 2012, p. 1. ISBN: 978-1-4503-1614-9. DOI: 10.1145/2393596.2393661. URL: http://dl.acm.org/citation.cfm?doid=2393596.2393661 (visited on 09/30/2020).

[10] Miyuru Dayarathna, Yonggang Wen, and Rui Fan. "Data Center Energy Consumption Modeling: A Survey". In: *IEEE Communications Surveys Tutorials* 18.1 (2016), pp. 732–794. ISSN: 1553-877X. DOI: 10.1109/COMST.2015.2481183.

[11] Alexandre Decan, Tom Mens, and Eleni Constantinou. "On the Evolution of Technical Lag in the npm Package Dependency Network". In: *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 2018 IEEE International Conference on Software Maintenance and Evolution (ICSME). Sept. 2018, pp. 404–414. DOI: 10.1109/ICSME.2018.00050.

[12] Erik Derr et al. "Keep me Updated: An Empirical Study of Third-Party Library Updatability on Android". In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. CCS '17: 2017 ACM SIGSAC Conference on Computer and Communications Security. Dallas Texas USA: ACM, Oct. 30, 2017, pp. 2187–2200. ISBN: 978-1-4503-4946-8. DOI: 10.1145/3133956.3134059. URL: https://dl.acm.org/doi/10.1145/3133956.3134059 (visited on 05/22/2020).

[13] M. Di Giacomo. "MySQL: lessons learned on a digital library". In: *IEEE Software* 22.3 (May 2005), pp. 10–13. ISSN: 0740-7459, 1937-4194. DOI: 10.1109/MS.2005.71. URL: https://ieeexplore.ieee.org/document/1438321/ (visited on 07/09/2020).

[14] Jens Dietrich, Kamil Jezek, and Premek Brada. "Broken promises: An empirical study into evolution problems in Java programs caused by library upgrades". In: *2014 Software Evolution Week - IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE)*. 2014 Software Evolution Week - IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE). Feb. 2014, pp. 64–73. DOI: 10.1109/CSMR-WCRE.2014.6747226.

[15] Jens Dietrich et al. "Dependency Versioning in the Wild". In: *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. 2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR). Montreal, QC, Canada: IEEE, May 2019, pp. 349–359. ISBN: 978-1-72813-412-3. DOI: 10.1109/MSR.2019.00061. URL: https://ieeexplore.ieee.org/document/8816809/ (visited on 04/24/2020).

[16] Wes Felter et al. "An updated performance comparison of virtual machines and Linux containers". In: *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS). Philadelphia, PA, USA: IEEE, Mar. 2015, pp. 171–172. ISBN: 978-1-4799-1957-4. DOI: 10.1109/ISPASS.2015.7095802. URL: http://ieeexplore.ieee.org/document/7095802/ (visited on 10/05/2020).

[17] Peter Garraghan, Paul Townend, and Jie Xu. "An Empirical Failure-Analysis of a Large-Scale Cloud Computing Environment". In: *2014 IEEE 15th International Symposium on High-Assurance Systems Engineering*. 2014 IEEE 15th International Symposium on High-Assurance Systems Engineering (HASE). Miami Beach, FL, USA: IEEE, Jan. 2014, pp. 113–120. ISBN: 978-1-4799-3466-9 978-1-4799-3465-2. DOI: 10.1109/HASE.2014.24. URL: http://ieeexplore.ieee.org/document/6754595/ (visited on 03/27/2020).

[18] Ilir Gashi, Peter Popov, and Lorenzo Strigini. "Fault Tolerance via Diversity for Off-the-Shelf Products: A Study with SQL Database Servers". In: *IEEE Transactions on Dependable and Secure Computing* 4.4 (Oct.

2007), pp. 280–294. ISSN: 1941-0018. DOI: 10.1109/TDSC.2007.70208.

[19] Petr Hosek and Cristian Cadar. "Safe software updates via multi-version execution". In: *2013 35th International Conference on Software Engineering (ICSE)*. 2013 35th International Conference on Software Engineering (ICSE). San Francisco, CA, USA: IEEE, May 2013, pp. 612–621. ISBN: 978-1-4673-3076-3 978-1-4673-3073-2. DOI: 10.1109/ICSE.2013.6606607. URL: http://ieeexplore.ieee.org/document/6606607/ (visited on 08/12/2020).

[20] Isam Mashhour Al Jawarneh et al. "Container Orchestration Engines: A Thorough Functional and Performance Comparison". In: *ICC 2019 - 2019 IEEE International Conference on Communications (ICC)*. ICC 2019 - 2019 IEEE International Conference on Communications (ICC). May 2019, pp. 1–6. DOI: 10.1109/ICC.2019.8762053.

[21] Ajay D. Kshemkalyani and Mukesh Singhal. *Distributed computing: principles, algorithms, and systems*. Cambridge ; New York: Cambridge University Press, 2008. 736 pp. ISBN: 978-0-521-87634-6.

[22] Alexei Ledenev. *alexei-led/pumba: Chaos testing, network emulation and stress testing tool for containers*. URL: https://github.com/alexei-led/pumba (visited on 09/29/2020).

[23] Wubin Li, Ali Kanso, and Abdelouahed Gherbi. "Leveraging Linux Containers to Achieve High Availability for Cloud Services". In: *2015 IEEE International Conference on Cloud Engineering*. 2015 IEEE International Conference on Cloud Engineering (IC2E). Tempe, AZ, USA: IEEE, Mar. 2015, pp. 76–83. ISBN: 978-1-4799-8218-9. DOI: 10.1109/IC2E.2015.17. URL: http://ieeexplore.ieee.org/document/7092902/ (visited on 10/05/2020).

[24] Martin Linkhorst. *linki/chaoskube: Periodically kills random pods in your Kubernetes cluster.* URL: https://github.com/linki/chaoskube (visited on 09/29/2020).

[25] *litmuschaos/litmus: Litmus helps Kubernetes SREs and developers practice chaos engineering in a Kubernetes native way.* URL: https://github.com/litmuschaos/litmus (visited on 09/29/2020).

[26] Suhaib Mujahid et al. “Using Others’ Tests to Identify Breaking Updates”. In: *Proceedings of the 17th International Conference on Mining Software Repositories*. MSR ’20: 17th International Conference on Mining Software Repositories. Seoul Republic of Korea: ACM, June 29, 2020, pp. 466–476. ISBN: 978-1-4503-7517-7. DOI: 10.1145/3379597.3387476. URL: https://dl.acm.org/doi/10.1145/3379597.3387476 (visited on 09/29/2020).

[27] *Netflix/chaosmonkey*. 2016. URL: https://github.com/Netflix/chaosmonkey (visited on 10/15/2020).

[28] B. Clifford Neuman. “Scale in Distributed Systems”. In: *IEEE Computer Society Press*. Readings in Distributed Computing Systems (1994). URL: https://cs.uwaterloo.ca/~brecht/courses/epfl/Possible-Readings/general/scale-dist-sys-neuman-readings-dcs-1994.pdf (visited on 03/09/2020).

[29] Jon Oberheide, Evan Cooke, and Farnam Jahanian. “CloudAV: N-Version Antivirus in the Network Cloud”. In: (2008), p. 16.

[30] Sebastian Österlund et al. “kMVX: Detecting Kernel Information Leaks with Multi-variant Execution”. In: *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS ’19: Architectural Support for Programming Languages and Operating Systems. Providence RI USA: ACM, Apr. 4, 2019, pp. 559–572. ISBN: 978-1-4503-6240-5. DOI: 10.1145/3297858.3304054. URL: https://dl.acm.org/doi/10.1145/3297858.3304054 (visited on 10/09/2020).

[31] Luís Pina et al. “MVEDSUA: Higher Availability Dynamic Software Updates via Multi-Version Execution”. In: *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS ’19: Architectural Support for Programming Languages and Operating Systems. Providence RI USA: ACM, Apr. 4, 2019, pp. 573–585. ISBN: 978-1-4503-6240-5. DOI: 10.1145/3297858.3304063. URL: https://dl.acm.org/doi/10.1145/3297858.3304063 (visited on 10/09/2020).

[32] *powerfulseal/powerfulseal: A powerful testing tool for Kubernetes clusters.* URL: https://github.com/powerfulseal/powerfulseal (visited on 09/29/2020).

[33] *PRINCIPLES OF CHAOS ENGINEERING - Principles of chaos engineering*. 2018. URL: https://principlesofchaos.org/ (visited on 10/15/2020).

[34] Steven Raemaekers, Arie van Deursen, and Joost Visser. "Semantic Versioning versus Breaking Changes: A Study of the Maven Repository". In: *2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation*. 2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation (SCAM). Victoria, BC, Canada: IEEE, Sept. 2014, pp. 215–224. ISBN: 978-1-4799-6148-1. DOI: 10.1109/SCAM.2014.30. URL: http://ieeexplore.ieee.org/document/6975655/ (visited on 09/28/2020).

[35] Jesper Simonsson et al. "Observability and Chaos Engineering on System Calls for Containerized Applications in Docker". In: *arXiv:1907.13039 [cs]* (July 2, 2020). arXiv: 1907.13039. URL: http://arxiv.org/abs/1907.13039 (visited on 09/28/2020).

[36] Sachchidanand Singh and Nirmala Singh. "Containers & Docker: Emerging roles & future of Cloud technology". In: *2016 2nd International Conference on Applied and Theoretical Computing and Communication Technology (iCATccT)*. 2016 2nd International Conference on Applied and Theoretical Computing and Communication Technology (iCATccT). Bangalore, India: IEEE, 2016, pp. 804–807. ISBN: 978-1-5090-2399-8. DOI: 10.1109/ICATCCT.2016.7912109. URL: http://ieeexplore.ieee.org/document/7912109/ (visited on 10/05/2020).

[37] Andy Singleton. "The Economics of Microservices". In: *IEEE Cloud Computing* 3.5 (Sept. 2016), pp. 16–20. ISSN: 2325-6095. DOI: 10.1109/MCC.2016.109. URL: http://ieeexplore.ieee.org/document/7742218/ (visited on 09/30/2020).

[38] Ayush Sobti. *asobti/kube-monkey: An implementation of Netflix's Chaos Monkey for Kubernetes clusters*. URL: https://github.com/asobti/kube-monkey (visited on 09/29/2020).

[39] Maarten van Steen and Andrew S. Tanenbaum. *Distributed systems*. Third edition (Version 3.01 (2017)). London: Pearson Education, 2017. 582 pp. ISBN: 978-1-5430-5738-6 978-90-815406-2-9.

[40] *worstcase/blockade: Docker-based utility for testing network failures and partitions in distributed applications*. URL: https://github.com/worstcase/blockade (visited on 09/29/2020).

[41] Zuoning Yin et al. "How do fixes become bugs?" In: *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering - SIGSOFT/FSE '11*. the 19th ACM SIGSOFT symposium and the 13th European conference. Szeged, Hungary: ACM Press, 2011, p. 26. ISBN: 978-1-4503-0443-6. DOI: 10.1145/2025113.2025121. URL: http://dl.acm.org/citation.cfm?doid=2025113.2025121 (visited on 09/15/2020).

[42] Long Zhang et al. "A Chaos Engineering System for Live Analysis and Falsification of Exception-handling in the JVM". In: *IEEE Transactions on Software Engineering* (2019), pp. 1–1. ISSN: 0098-5589, 1939-3520, 2326-3881. DOI: 10.1109/TSE.2019.2954871. arXiv: 1805.05246. URL: http://arxiv.org/abs/1805.05246 (visited on 02/26/2020).

[43] Long Zhang et al. "Automatic Observability for Dockerized Java Applications". In: *arXiv:1912.06914 [cs]* (Aug. 25, 2020). arXiv: 1912.06914. URL: http://arxiv.org/abs/1912.06914 (visited on 09/29/2020).