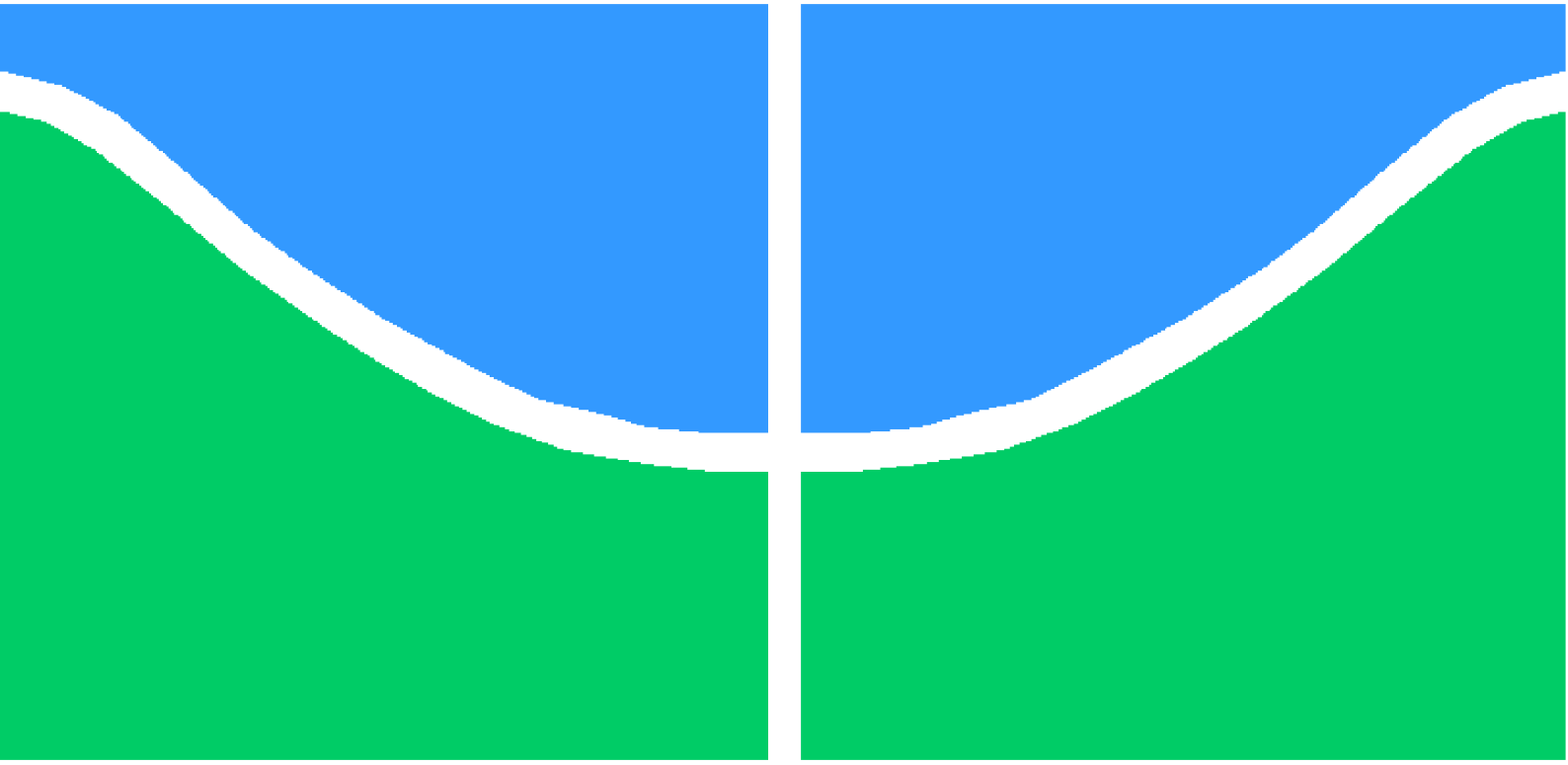


Universidade de Brasília - UnB

Faculdade UnB Gama - FGA

Engenharia de Software

DevOps Pipeline Optimization: Reduction in the Use of Computational Resources When Using Watchtower

Autor: Victor Correia de Moura

Orientador: Prof. Dr. Renato Coral Sampaio

Brasília, DF

2020

Victor Correia de Moura

DevOps Pipeline Optimization: Reduction in the Use of Computational Resources When Using Watchtower

Monografia submetida ao curso de graduação em (Engenharia de Software) da Universidade de Brasília, como requisito parcial para obtenção do Título de Bacharel em (Engenharia de Software).

Universidade de Brasília - UnB

Faculdade UnB Gama - FGA

Supervisor: Prof. Dr. Renato Coral Sampaio

Co-supervisor: Prof. Dra. Carla Silva Rocha Aguiar

Brasília, DF

2020

Victor Correia de Moura

DevOps Pipeline Optimization: Reduction in the Use of Computational Resources When Using Watchtower/ Victor Correia de Moura. – Brasília, DF, 2020-
60 p. : il. (algumas color.) ; 30 cm.

Supervisor: Prof. Dr. Renato Coral Sampaio

Trabalho de Conclusão de Curso – Universidade de Brasília - UnB
Faculdade UnB Gama - FGA , 2020.

1. DevOps. 2. Continuous Deploy. I. Prof. Dr. Renato Coral Sampaio. II. Universidade de Brasília. III. Faculdade UnB Gama. IV. DevOps Pipeline Optimization: Reduction in the Use of Computational Resources When Using Watchtower

CDU 02:141:005.6

Victor Correia de Moura

DevOps Pipeline Optimization: Reduction in the Use of Computational Resources When Using Watchtower

Monografia submetida ao curso de graduação em (Engenharia de Software) da Universidade de Brasília, como requisito parcial para obtenção do Título de Bacharel em (Engenharia de Software).

Trabalho aprovado. Brasília, DF, 15 de dezembro de 2020:

Prof. Dr. Renato Coral Sampaio
Orientador

Prof. Dra. Carla Silva Rocha Aguiar
Convidado 1

Prof. Dr. Tiago Alves da Fonseca
Convidado 2

Brasília, DF
2020

"God will not have his work made manifest by cowards".
(Ralph Waldo Emerson, Self-Reliance)

Abstract

This work details how software engineering has evolved into an era of continuous delivery and DevOps, in which Watchtower is inserted and operating with an active behaviour that affects the usage of computational resources. Watchtower is a continuous deployment tool that actively updates running Docker containers by polling the remote registry to detect image changes. Since polling requires it to constantly make pull operations and it is not guaranteed that there will be an image change on every pull, which would result in a container update, the effectiveness of this strategy is lower than if there was a pull only when changes are available. To prevent unnecessary pulls, this work proposes and presents the implementation of an HTTP API that exposes an endpoint that triggers the update operation, enabling external services to notify whenever a change has been uploaded and to go for proper container updates only when necessary, reducing the usage of computational resources.

Key-words: DevOps. Continuous Deploy. Watchtower.

Resumo

Este trabalho detalha como a engenharia de software evoluiu para uma era de entrega contínua e DevOps, na qual o Watchtower está inserido e operante com um comportamento ativo que afeta o uso de recursos computacionais. O Watchtower é uma ferramenta de *Deploy* Contínuo que, de maneira ativa, realiza atualizações em contêineres Docker em execução consultando, de forma periódica, o repositório remoto de imagens buscando por eventuais mudanças. Como a consulta periódica faz com que a ferramenta realize requisições frequentes sem que haja a garantia de que de fato houve uma mudança na imagem, o que resultaria na atualização do contêiner em execução, a efetividade desta estratégia é menor do que se houvesse uma requisição apenas quando mudanças na imagem estivessem disponíveis. Para evitar requisições desnecessárias, este trabalho propõe e apresenta a implementação de uma API HTTP que expõe uma rota que, quando requisitada, dá início ao processo busca por mudanças, permitindo com que serviços externos possam notificar sempre quando houver uma mudança na imagem e sinalizar que uma atualização de contêiner deve ser realizada, reduzindo o uso de recursos computacionais.

Palavras-chave: DevOps. *Deploy* Contínuo. Watchtower.

List of Figures

List of abbreviations and acronyms

API	Application Programming Interface
CLI	Command Line Interface
DP	Dynamic Programming
HDD	Hypothesis Driven Development
HTTP	Hypertext Transfer Protocol
LAPPIS	Laboratório Avançado de Pesquisa, Produção e Inovação em Software
OS	Operational System

Contents

1 Introduction

Software engineering has been one of the most evolving areas of knowledge in the latest times. New methodologies, tools and practices can emerge on every project, but few are widely adopted by the community. Certainly, we could say that one of the few is the adoption of DevOps practices. With the advance of agile methodologies, the need for continuously delivering value to the client has pushed development operations to a new level of automation, where Amazon, for example, deploys a new version of its famous website every 11.6 seconds on average, as described in (JENKINS, 2011).

To make automatic operations feasible, there are tools and architectural paradigms to guide teams to a standard and maintainable way of treating its services. An example of a technology popularized in the 21st century is Docker (DOCKER, 2019a), which aims to gather service orchestration and configuration management in a way that continuous deployment is a matter of swapping predefined containers.

Given its popularity, the Docker product family has motivated the creation of some robust tools, like Rancher (RANCHER, 2019b) and Kubernetes (KUBERNETES, 2019). Those tools empower teams to orchestrate and maintain thousands of containers, with load balanced requests, distributed persistent volumes, hosted in clusters with hundreds of machines. But, despite being open-source technologies, this kind of architecture demands a high service load to reach its full potential. That is the type of scenario that may be common in big tech companies environment, but it is not the reality for small startups or students that are looking for a way to continuously deploy a recently created proof of concept.

To fill the observed gap, an open-source community came up with Watchtower (CONTAINRRR, 2015), which is a single container that could be deployed, with Docker Compose, along with the other services. Its responsibility is to monitor, in the stack, the used Docker images, so it could keep all of the watched containers running with its latest available image. To be able to check whether the used image is the latest or not, Watchtower periodically pulls from the image registry to check if there is a match between the downloaded and the used images. If there is a newer, a new container is instantiated.

Even though Watchtower gracefully deploys newer images, it can be observed that, since it is periodically pulling from the registry, there is an overuse of computational resources, as not every pull results in a deployment. As a result, the amount of processing and exchanged data over time can be higher than expected. To reduce the usage of computational resources, it is necessary to avoid unnecessary image pulls by notifying Watchtower when to update the running containers, instead of relying on its periodical

polling behaviour.

Besides affecting computational resources usage, image pull limits can be imposed by the image storage host. So a polling strategy, in which multiple pulls are potentially being requested in a short period of time, can be made unfeasible.

This work proposes an architectural change in which an Application Programming Interface (API) is created, and coupled to Watchtower, with at least one endpoint available for external services to send a request, so it can expect other services to notify when to update, instead of actively checking if any change has been made.

1.1 Objectives

The main goal of this work is to design the new behaviour of Watchtower so, in the continuous deployment stage, it does not request images when there is no change in the registry, assuring an efficient usage of computational resources and avoiding eventual image pulling quotas. To develop this approach, the following goals should be achieved:

- Create an experiment which shows the current behaviour;
- Design the new behaviour;
- Implement the designed approach and make it available to the Watchtower community;
- Create an experiment that allows to show differences between resource usage and results for comparison between both behaviours;

1.2 Work Structure

This work will present itself by defining a background, that introduces relevant and necessary topics to understand the problematic, a methodology, describing which and how experiments should be made to achieve and understand the results and discussions, and a conclusion, to summarize and describe the impact and the future of the developed solution.

2 Background

2.1 DevOps

As discussed in (ERICH; AMRIT, 2017), there could be many definitions for DevOps. The lack of quantitative and well defined advantages often hide the fact that teams are actually applying it in a daily basis. To fully understand what DevOps means, it is necessary to understand in what context it became a reality and what are its impacts.

The adopted concept, based in (DYCK; PENNERS; LICHTER, 2015), is the one defined in (LEITE et al., 2019):

> *DevOps is a collaborative and multidisciplinary effort within an organization to automate continuous delivery of new software versions, while guaranteeing their correctness and reliability.*

One of the reasons that make DevOps a multi defined concept is the existence of different point of views. One can be an engineer, a manager or a researcher and still benefit from this culture. Since DevOps can be seen as practices evolving many agents, these have to have an impression of what it is.

From the engineer perspective, DevOps can be an extremely technical and well paid position in which the necessity to work with multiple low level tools is a fact. Also, it could be necessary for a software architect to adapt the services design so it can be continuously deployed.

Still, a company could demand its managers to introduce DevOps concepts in the existing pipelines, monitor the results of already used DevOps practices or, along with the engineers, decide on which tool set should be adopted by the company's collaborators.

Lastly, researchers conduct studies on how DevOps practices are being implemented, whipping up discussions about it and producing materials that can be used by managers, to support and align taken decisions, or engineers, to empower themselves with the state of the art concepts, allowing the good use of existing technical solutions or the production of newer ones.

DevOps can also be seen as an evolution of agile methodologies. As is the practice in these methodologies, software development should be divided by constant, atomic, iterations driven by customer reviews. Technically, it has been a challenge to truly implement a pipeline that made this process feasible. It is historically known that moving a software to production can be a tense experience, since it relies on the alignment between

developers, code and environment. DevOps has emerged as a solution to the recognized methodological gap between development and deployment.

2.2 Containers and Container images

In the context of this work, containers will be seen as a standardized, lightweight, packaged software unit defined in (DOCKER, 2019b). Containers are defined by container images, that describe what files, dependencies and environments the container will pack. When Container images are executed, they become containers and, when a container is running, it exists as a volatile entity in the system. In other words, if a container self changes during runtime, it won't persist these changes after a reboot.

Similar to virtual machines, containers provide the operating system tool set to the running application. The main difference is how this abstraction is made. Virtual machines require a hypervisor layer, that is responsible for translating system calls made by the virtualized hardware to understandable calls made to the used hardware infrastructure. With a lighter solution, containers are seen as processes from the base operational system (OS) perspective, so the OS kernel is shared along with other applications or containers. The peculiarities of this setup are particular for each container engine.

2.3 Docker and Docker Compose

According to (DOCKER, 2019b), Docker is a technology in which containers and containers images can be based on. A container that is Docker based can be called as a Docker container. The same applies to images and Docker images. Since an engine is necessary to run containers, Docker containers must exist over the Docker Engine, which is installable in the base OS.

In many use cases, multiple containers are run at the same time, with separate environments and dependencies, but still representing a single application or service. For example, an API based service that requires an alongside database can be represented by two containers:

- An API container, with all the business rules and features;
- A database container, with the persistent data collected by the API.

To support developers with similar situations, in which is necessary to maintain multiple alongside containers, there is a Docker family product called Docker Compose, that collects container configurations, also available through a command line interface

(CLI), in a single *.yml* file. Even though it is possible not to use it, command line operations are poorly versioned and they can vary depending on which environment the containers are being run. For this reason, Docker Compose is well accepted in the Docker community.

It is important to notice that, even though Docker Compose is a powerful management configuring tool, it doesn't solve all of the problems. Distributed operations and load balancing, for example, aren't simply supported by Docker Compose. For this reason, it is known that, if a service needs scalability, Docker compose might not be the optimal tool for the production environment. When more robust orchestration strategies are needed, other tools, as Docker Swarm (DOCKERDOCS, 2019c), should be used.

Docker Compose focuses on a niche that include the following, but not limited to, situations:

- Development environments, to ensure the app dependencies are the same as in other environments;
- Small scale production environments, with affordable operations for a small startup or a student group.

As a Docker user, one should understand concepts as Docker layers, Docker tags and Docker volumes since they are part of the usual workflow when one is dealing with the tool. These three are defined in the containerizing process, which is where the developer defines files and dependencies to be encapsulated into the container. (DOCKERDOCS, 2019d) provides a guide to help first-time users to configure a Docker based environment.

From setting up to testing, the stages involved are:

1. Defining the base image;
2. Defining which files and dependencies should be in the container;
3. Defining the command which first starts the application;
4. Building the image;
5. Testing the image.

The first three stages demand the developer to edit a Dockerfile, which is the file that contains the instructions necessary to build a specific Docker Image. Each instruction in the Dockerfile represents a layer in the Docker Image, as defined in (DOCKERDOCS, 2019a).

This layer-based approach, also referred as Build cache, is used by the Docker Engine to better manage computational resources when dealing with image building,

uploading and downloading. Since Dockerfile is a versionable artifact of the project, it is expected for it to be modified over time. These modifications may or may not impact the whole image — e.g., a single file **ADD** instruction (which stands for copying a file from the local folder to the container folder) at the end of the Dockerfile won't impact on how the previous states have been executed. In the other hand, if a **FROM** instruction (which defines the base image that is used in the described by the Dockerfile) is changed, the whole build could be impacted.

If layers didn't exist, the naive approach would imply in the whole image rebuild on every change, which could be unnecessarily costly when simpler modifications occur. With layers, it is possible to reconstruct only from the altered layer and thereafter. Similarly, image downloading (or pushing) and uploading (or pulling) presents the same behaviour — if an image is partially downloaded or uploaded, only the subsequent layers will be transferred.

Also in the three first stages, a developer should declare if the container will be mapped with a Docker volume or not. Docker volumes are a way to create an interface between a running container and a local system folder. Since containers are naturally volatile, changes to files won't persist unless a volume is set. With this approach, it is possible to replicate file changes that occur inside the container in a local storage device, allowing persistence to happen. Without Docker volumes, a persistent database should never be executed inside a Docker container, since the data would never be permanent in a hardware device.

In the last two stages, it is required for the user to understand how Docker Tags work. Since most applications source codes are versioned, Docker attempts to provide full versioning with editable configurations files, as Dockerfile and Docker Compose's *.yml*. With built images, there is a trickier situation, since the resultant files are an agglomerate of binary layers. To allow image versioning, Docker provides tags, which can be associated to the images. The tags are text labels that identify a specific build of a Dockerfile. This feature allows developers to control which version of the application is packed into an image, since a tag could be any sequence of allowed characters - as the hash of a git commit. For all situations, when no tag is provided, Docker automatically sets the label **latest** to it.

2.4 Docker Registries

As defined in (DOCKERDOCS, 2019b), a Docker Registry is a delivery system which holds Docker Images and make them available to clients who need an image repository. Registries are implemented with Docker's tags and layers schema, taking advantage of the design potential. Since Docker is an open-source technology, one can deploy it's

own registry. But, for users who are looking for a no maintenance solution, Docker provides a ready-to-use platform called DockerHub, a fully managed registry with some extra features, implemented over the default Docker Registry API.

With these registries, it is possible to download and upload Docker images, as shown in Listings 2.1 and 2.2:

```
$ docker pull debian:latest
```

Listing 2.1 – Downloading Debian's latest Docker Image from DockerHub

```
$ docker push lappis/coach:tais
```

Listing 2.2 – Uploading an image named **coach**, labeled as **tais**, to DockerHub's **lappis** organization

Despite making some extra features available over the basic registry solution, DockerHub is not fully free-to-use (DOCKER, 2020). Paid plans enable multiple tools, as:

- Private image storage;
- Unlimited API calls.

Whenever a client pulls an image, for instance, requests are made to the DockerHub API in order to complete the operation. According to the official documentation, a pull request consumes at least one unit of the requester's usage quota (DOCKERDOCS, 2020). By the date of this work, for free tier users, there is a limit of 200 image pulls per 6 hours (DOCKER, 2020). If this limit is reached, a warning is sent to the requester and it won't be allowed to pull the image. No limits were being applied until Nov, 2. 2020.

Usage quotas are applied based on the user doing the pull, not on the pulled image or its owner. So, even if an image is stored in a private paid tier registry, a free tier user won't be allowed to unlimitedly pull it.

2.5 Continuous Deploy and Continuous Integration

Closely aligned with Agile Methodologies, Continuous Integration and Deploy have allowed companies to shift from a more traditional way of working to a more dynamic and organic manner of delivering value, as detailed in (OLSSON; ALAHYARI; BOSCH, 2012). In the context of this work, the mentioned article considerations will be used as a baseline.

That said, and given the fast paced modern software development, Continuous Integration is a practice that allows teams to continuously integrate reliable code to its product. To achieve that, fresh modifications are submitted to a verification process to assure correctness and reliability. Automated tests are, usually, the chosen approach to approve or reject a modification. With Continuous Integration, it is expected from a team to always have a reliable deliverable version of its software.

Differently from code integration, deployment relates more to truly delivering the fresh produced changes to the final user. As a further step from Continuous Integration, a company that implements Continuous Deployment usually delivers new software versions to its customers more frequently.

As noticed, many automations are required to achieve these practices in an enterprise. From that need, different platforms have been created to empower developers with the tools needed to continuously integrate and deploy code. Some examples of Continuous Integration platforms are:

- CircleCI (CIRCLECI, 2019);
- GitLabCI (GITLABDOCS, 2019);
- TravisCI (TRAVISCI, 2019).

In these platforms, it is possible to arrange jobs and scripts that execute the verification stages in the software production pipeline. E.g: A continuous integration pipeline that approves a merge request if, and only if, the unit tests stage finishes with no failure.

It is also possible, specially in Docker based projects, to attach, in the end of the pipeline, a stage to publish the latest built project's Docker image, as seen in Tais (LAPPIS, 2017), an open-source chatbot developed by LAPPIS (Laboratório Avançado de Pesquisa, Produção e Inovação em Software). With this behavior, there is always a published and deployable version of the software.

Since the automation level allows developers to design any stage, as long as it can be abstracted into a script, Continuous Deployment can be implemented with a single stage in the Continuous Integration pipeline. A materialization of this situation is the one used by Lappis Learning project (LAPPIS, 2018a), also developed by LAPPIS. The Continuous Integration pipeline is given by the following steps:

1. Build, in which the Docker image is built and pushed to the registry;
2. Test, in which the code static analysis and unit tests are executed;
3. Deploy, in which the Rancher's environment is instructed to pull the latest image and deploy it.

As most pipelines, if a step fails, the remaining won't be executed. This way, a deploy never happens if a build or a test fails. Also, it is noticeable that a deploy only actually happens when the deployment environment changes its state. In Lappis Learning's example, a different container in execution would be considered as a state change. When this change happens along with the concepts of Continuous Integration, there is Continuous Deployment.

2.6 Watchtower and Moby

Watchtower is a tool, written in Golang (GOLANG, 2019), used with Docker Compose, to allow continuous container deployment by simply pushing a new image to the registry (CONTAINRRR, 2015). The monitoring process happens as followed:

1. A monitoring interval is given to Watchtower (e.g., 30 seconds);
2. On each interval, Watchtower pulls the images used in the containers of interest;
3. Given the pulled images, there happens a check to assure none of the used are different from the freshly pulled;
4. If there is a difference, a SIGTERM is sent to the container and a new one is deployed with the latest image.

The containers of interest are those who attend the following requirements:

- Is in the same Docker Compose stack as the Watchtower's container;
- Is labeled as a monitored container, as described in (CONTAINRRR, 2019c).

Since Watchtower's approach is to always deploy the latest built Docker image, it is possible to implement a continuous deployment pipeline by simply adding a step, in any continuous integration platform, to push the latest built image to a registry reachable to Watchtower. With the push, it's possible to assume that there will be a deployment shortly after, depending on the chosen monitoring interval.

To manipulate Docker containers, Watchtower uses the Moby Project, which is a toolset, developed by Docker, to support developers with low level container operations (MOBY, 2017). Initially, Moby was part of the Docker Engine, but, eventually, the community noticed that the containerization API itself was generic enough to work with other engines. At the date of this work, Docker Engine depends on Moby project, instead of containing it. This setup is simplified due to the way Golang allows developers to manager their project dependencies. Moby is reachable by simply importing the source code available in GitHub, as shown in the code snippet below:

```
package main

import (
  "context"
  "fmt"
  "github.com/docker/docker/api/types"
  "github.com/docker/docker/client"
)
```

Listing 2.3 – Importing Moby package in a Golang project

It is important to notice that, even though the import refers to the Docker package, Moby is being imported. At the moment of this work, the transition from Docker API to Moby API is in progress. The code base has been migrated but the import statements are still referencing the old repository.

With Moby, it is possible to perform actions to manipulate Docker containers by calling the API provided interfaces. To list all running containers, for example, Moby client package can be used:

```
func main() {
  cli := client.NewClientWithOpts(
      client.WithVersion("1.40")
  )
  containers := cli.ContainerList(
    context.Background(),
    types.ContainerListOptions{}
  )
  for _, container := range containers {
    fmt.Printf("%s\n", container.Image)
  }
}
```

Listing 2.4 – Using Moby client to list running containers

Due to Watchtower's architecture, developers are not required to master the peculiarities of the Moby Project in order to reuse existent actions to manipulate running Docker containers - e.g. Watchtower's **container** package provides an interface named **Client**, that provides useful operations to developers, as **ListContainers**, **HasNewImage** and **StopContainer**.

Another important information about Watchtower is that it is an open-source software and its source code is fully available on its GitHub project (CONTAINRRR,

2015). With access to the code base, it is possible to edit it and insert any wanted behavior to the tool. If a code implementation is in the interest of the community, it can be pushed to the core Git branch to be made available in the next release.

2.7 Existing Solutions

2.7.1 GitLabCI and GitHub integration

GitHub users, until the creation of GitHub Actions (GITHUB, 2019), had to use external tools to set up their continuous integration pipeline. One of the solutions available was GitLabCI (GITLABDOCS, 2019). To set up a pipeline with GitLabCI for a GitHub project, one had to mirror the repository in GitLab and, only then, plug GitLabCI for it to monitor for changes and trigger the pipeline whenever a new commit was pushed.

GitLab's mirror schema had two approaches to trigger an update in the mirror:

- Periodically pull from GitHub repository, even if no change has been made;
- Receive requests, through GitLabCI API, from GitHub's webhook whenever a commit was pushed.

The webhook approach used to present better response time to a commit, since there was an immediate trigger to GitLab's mirror to pull from GitHub whenever a new commit was pushed. The strategy in which the pull was periodically made could present worst results due to the fact that not always the commit push action was immediately followed by the mirror's pull, since these events had not been related to each other, like in the webhook solution.

Developers with no permissions to set up the GitHub webhook had to go for the other approach. In practice, that meant that the CI pipeline would not trigger immediately after each commit push, increasing the time the development team had to wait for the CI result and delaying pull requests reviews and approvals, since these events also rely on the CI result.

Given the nonexistence of these delays with the webhook, it can be considered an optimal solution to this integration.

2.7.2 Rancher upgrade calls in GitLabCI

Rancher (RANCHER, 2019b) has been one of the tools used by developers to allow container orchestration in production environment. One of its features is a set of webhooks, available through Rancher API, to allow action triggering via HTTP requests.

With these endpoints, the GitLabCI community created a CI image called **rancher-gitlab-deploy** (CDRX et al., 2019), which packed utilities to help setting up the CI pipeline with ease. One of the available commands was **upgrade**, which made a post to Rancher API triggering an update, of a deployed container, with the latest available image. This schema allowed continuous deployment of applications, since the webhook call could be made in the end of the CI pipeline, after no failures in the previous steps and in only when the job referred to a specific branch.

This approach was used in Lappis Learning Project (LAPPIS, 2018a). In Listing 2.5, it is possible to see the code snippet, available in (LAPPIS, 2018b), of the continuous integration stage in which there was the implicit call to the Rancher API webhook.

When the job referred to a **master** branch commit and the pipeline reached the **deploy** stage without failures, the previously set environment variables were used as arguments to identify the specific service that should be upgraded. If the upgrade ended with failures, the job had a failed verdict and vice versa.

```
deploy to production:
  stage: deploy
  image: cdrx/rancher-gitlab-deploy
  environment: prod
  script:
    - upgrade --environment $RANCHER_ENVIRONMENT \
              --stack $RANCHER_STACK \
              --service salic-ml \
              --debug
  only:
    - master
  tags:
    - docker
```

Listing 2.5 – Triggering a container upgrade in Rancher in GitLabCI (LAPPIS, 2018b)

3 Methodology

This chapter presents the methodology and the necessary concepts to reproduce the process used to achieve the same results as this work.

3.1 Contributing to an open-source software

Before contributing to any software project, one must gather the needed permissions to change the docs or the code base. As this work includes a contribution to an open-source software, the methodology should be adapted to fit the development flow of a community contributor, including the permissions to do it. According to (GITHUB, 2020c), in order to help newcomers and standardize contributions, typical open-source projects usually provide the following files:

- *LICENSE*;
- *README*;
- *CONTRIBUTING*;
- *CODE_OF_CONDUCT*.

An open-source software *LICENSE* file is, by definition, what makes it open-source in reality. Without an open-source license, a project cannot be considered an open-source software. Watchtower is licensed under the Apache License 2.0, which allows commercial use, modification, distribution, patent use and private use (CONTAINRRR, 2018). This license ensures that any person can modify Watchtower's code base, making this work legally feasible.

Even though the license allows one to change the code base, this work's proposed code increment might positively impact current Watchtower users. To make its way to the core code base, a contribution must be reviewed and approved by the project maintainers. Usually, maintainers require contributors to follow the guidelines defined in *CONTRIBUTING* and *CODE_OF_CONDUCT*.

Watchtower's *CONTRIBUTING* states that, to contribute to the project, Go 1.11 and Docker will be required as part of the development kit (CONTAINRRR, 2020b). Besides that, it also includes a guide on how to build and test a local Watchtower Docker image. This document will be used as the technical baseline of any code contributions in the scope of this work.

By contrast, the *CODE_OF_CONDUCT* includes behavioral recommendations in order to keep the contributing environment inclusive, respectful and welcoming (CONTAINRRR, 2020a). This document will be used as the ethical baseline of communications with any Watchtower maintainer, contributor or user in the scope of this work.

Besides following the contribution guidelines, to achieve the objectives of this work, the code base must be understood so new features can be implemented. The understanding of the source code will also be used as a guideline for architectural patterns and code styling. Watchtower's source code is available in its official GitHub repository (CONTAINRRR, 2015).

To contribute to Watchtower's main code base, the following steps will be necessary:

1. Forking the official Watchtower repository;
2. Creating an issue to make the contribution visible to the community;
3. Creating a Pull Request to the main repository branch in order to insert the changes into the next release.

The first step is necessary to enable the pushing of unstable code to a remote repository during the development phase. Instructions on forking a GitHub repository are defined by (GITHUB, 2020b). The latest two steps are needed to collect feedback from the community on the proposed feature and to request its merge into the core code base, respectively, as defined by (GITHUB, 2020a). Due to the needed setup to enable this process, which is how the majority of Watchtower's contributions are made, it can be assumed that Git (GIT, 2019) will be used as the versioning protocol of any code within the scope of this work.

3.2 Architectural Design

Given that the code contribution guidelines and tools are well defined and understood, it will be necessary to design the actual code solution to run the tests and prove, or not, that the expected results were, or weren't, found. To better define the architectural design stage, it can be divided into three processes:

1. Learning from existing solutions;
2. Defining technical and non-technical characteristics of the solution;
3. Designing and implementing the solution.

3.2.1 Learning from existing solutions

As mentioned in the background chapter, there are similar solutions in existing tools. Rancher 1.6 (RANCHER, 2019b), for instance, has an HTTP API implemented which allows third-parties to send a request to trigger an image update. However, because of the context in which Rancher is inserted, there may be architectural decisions that were made to fit more sophisticated use cases, as a deployment which allows a rolling back if a failure is detected. These must not directly drive how Watchtower's solution will be made.

In this stage, it is important to list possible inspirations and document how they can be adapted to Watchtower's context. Also, given that many solutions may not be written in Golang, as Watchtower, it can be more relevant to study how the features behave over how they have been implemented.

3.2.2 Defining technical and non-technical characteristics of the solution

Aligned with the results of the first stage, it comes the step in which Watchtower's context is brought up alongside found solutions and a custom designed implementation is idealized.

The goal is to reduce complex inspirations into a more controlled environment, such as Docker Compose's, without the need for high levels of scalability or reliability. This exercise must have, as an output, the list of the technical and non-technical requirements of the solution, even though the expected nature of the final implementation (an individual, voluntary, open-source contribution) does not make it mandatory.

3.2.3 Designing and implementing the solution

Due to the code contribution methodology, community feedback is expected by the time an issue or a pull request is created on the official repository. So the forking of the repository and the opening of an issue or pull request must occur by the beginning of this stage.

Supported by the community, the needed resources (technical and non-technical characteristics) should be gathered, making it possible to design the actual implementation. The desired output is a modified version of Watchtower that implements the designed feature.

3.3 Experiment design

In order to monitor any improvements, there must be a controlled environment to collect metrics that can indicate whether the increment has had a positive impact or not. The design of this experiment must consider the following aspects:

- It must represent a Watchtower real continuous deployment setup;
- It must isolate unwanted variables, as deployment frequency, for instance.

Even though the latest item can appear to be obvious in a scientific context, the first one is specially important due to the objectives of this work. Without a real use case, the implementation wouldn't add value up to the software itself, which can increase the chances of the contribution not to be accepted by the community. The non acceptance of the contribution results in one of the objectives not being achieved.

4 Results and Discussions

This chapter presents a feature proposal, how it has been implemented and its results.

4.1 Watchtower API proposal

Given Watchtower's default behaviour, it is reasonable to assume it makes unnecessary requests to Docker image registries: it pulls images in a fixed frequency and, not necessarily, there will be an image change in each pull. There may occur pulls, from the registry, of unchanged images, which can be faced as an inefficient behaviour.

Under the successes of related works, mentioned on the background chapter, it is possible visualize an hypothesis that states: an API which allows Watchtower to be notified when to pull would result in a more efficient operation of the tool.

Given the hypothesis above, the next step is to make the problem and the probable solution visible to the community in order to get feedback and assess the interest in accepting the increments.

4.1.1 Joining the Community

Joining a community is not always a straightforward process that can be taught or well defined. As each project defines the contribution process based on its own environment, the strategies have to be adapted to the context in order to be successful. As it strongly depends on the personality of the agents, the approach can vary.

As Watchtower's core community was composed by a dozen regular contributors and maintainers, it sounded more adequate to, at first, get in touch individually with the core maintainer, which was also the most active contributor, to look for some kind of mentorship and explain that the initial goal was to develop a scientific research that included a code contribution to Watchtower. As it can be seen in Appendix A, this first approach via email was an indicative that the increment could be a valuable feature to the project.

After getting in touch with the first member of the community, the next step was to create an issue and expose the feature idea not only to the core maintainer, but also to current users and contributors. Through a GitHub issue (MOURA, 2020b), and a GitHub pull request (MOURA, 2020a), it was possible not only to expose the HTTP API proposal, but also, as planned, to collect community feedback on technical and non-technical aspects

of the solution.

It can also be seen that, even though the feature had been better explained through the issue, the community got more engaged through the pull request, where there was an actual code implementation to review. So, to take advantage of the engagement, potentializing and fomenting feedbacks, the pull request description was edited to include the completed and remaining sub-tasks and make them more explicit to those who were interested in knowing what and how Watchtower would change after the acceptance of the increments. Also, there was added a link to the issue if a pull request reviewer wanted to check the full and detailed feature proposal.

4.1.2 Watchtower API Design and Implementation

As mentioned in the background chapter, Watchtower is a continuous deployment tool, written in Golang. Also, the designed hypothesis states that, with HTTP requests, it would be possible to notify Watchtower when to pull a new version of the deployed image. Given these statements, it's possible to assume that the optimal implementation of the solution would be a Golang HTTP web API.

This solution has been inspired in the Rancher 1.6 HTTP API (RANCHER, 2019a). Despite being a powerful tool to control the orchestrator, it presents features that are not in the interest of this work, as:

- Updating a single service in the stack by specifying a label;
- Updating services with possible Rollback (deploying a container keeping the old as a backup).

Those can be seen as specific features to the context in which Rancher is inserted. Watchtower itself doesn't even present the necessary back-end to enable these functionalities and it is not a goal of this work to implement them, as discussed by a few contributors in the pull request comment section (GITHUB, 2020a). On the other hand, some aspects of Rancher API were also discussed and are of great interest:

- Access control through a request token;
- Each stack has its own endpoint, as each Watchtower instance can have its own API;
- API versioning through custom endpoints ("/v1", "/v2").

Besides analyzing existent solutions, it was important to fit them into Watchtower's context. One of the core features this work aims to substitute with the proposed solution

is the interval based registry monitoring. Even though this work assumes it is not an optimal solution, there may be some use cases in which its use is made necessary. If there isn't a way to expose an API endpoint to external services, for example, Watchtower will only be useful if its interval based behaviour keeps untouched. For this reason, it is important not to completely substitute this feature, but to create another option to the users. A possibility was to make the API an optional behaviour that would be enabled when the user explicitly tells it to.

Another important aspect is not to change Watchtower's settings interface. To customize Watchtower's behaviour, the user can adjust its settings through environment variables and command line arguments. So, to enable and configure the HTTP API, the same interface should be used.

Given this brief study and the adopted methodology, the preliminary list of requirements states that the developed API should:

- Be build with Golang and made available as a module to Watchtower;
- Make use of existent Watchtower functionalities;
- Not completely substitute the default interval based behaviour;
- Not be enabled by default not to impact current users;
- Be enabled by the user through command line arguments and environment variables;
- Keep versioning of its interfaces through specific endpoints ("/v1", "/v2").

To build a solution with the listed requirements, it seemed important to dominate the involved technologies and tools. Therefore, it was created a project called Go Webhook Server (MOURA, 2019a), which is an HTTP web API built with Golang. The code snippet in 4.1 shows how it works.

As it can be noticed, there are two functions: **main** and **handleWebhook**. The **main** function is the one that initializes the program and sets up the API, which exposes an endpoint called "/webhook" and listens to requests made to the 8080 port. It also binds a callback function named **handleWebhook** to the exposed endpoint. As a result, the exposed API will be able to receive requests made to a custom set endpoint and process them with a custom made function.

```go
func handleWebhook(w http.ResponseWriter, r *http.Request) {
  fmt.Printf("headers: %v\n", r.Header)

  _, err := io.Copy(os.Stdout, r.Body)
  if err != nil {
    log.Println(err)
    return
  }
}

func main() {
    log.Println("server started")
  http.HandleFunc("/webhook", handleWebhook)
  log.Fatal(http.ListenAndServe(":8080", nil))
}
```

Listing 4.1 – Golang HTTP API Example (MOURA, 2019a)

So, with the initially designed technical implementation, it is essential to understand Watchtower's code base structure so the contribution can be actually implemented on the correct place.

By the time of this work, Watchtower was structured in multiple packages as shown in Figure 1:

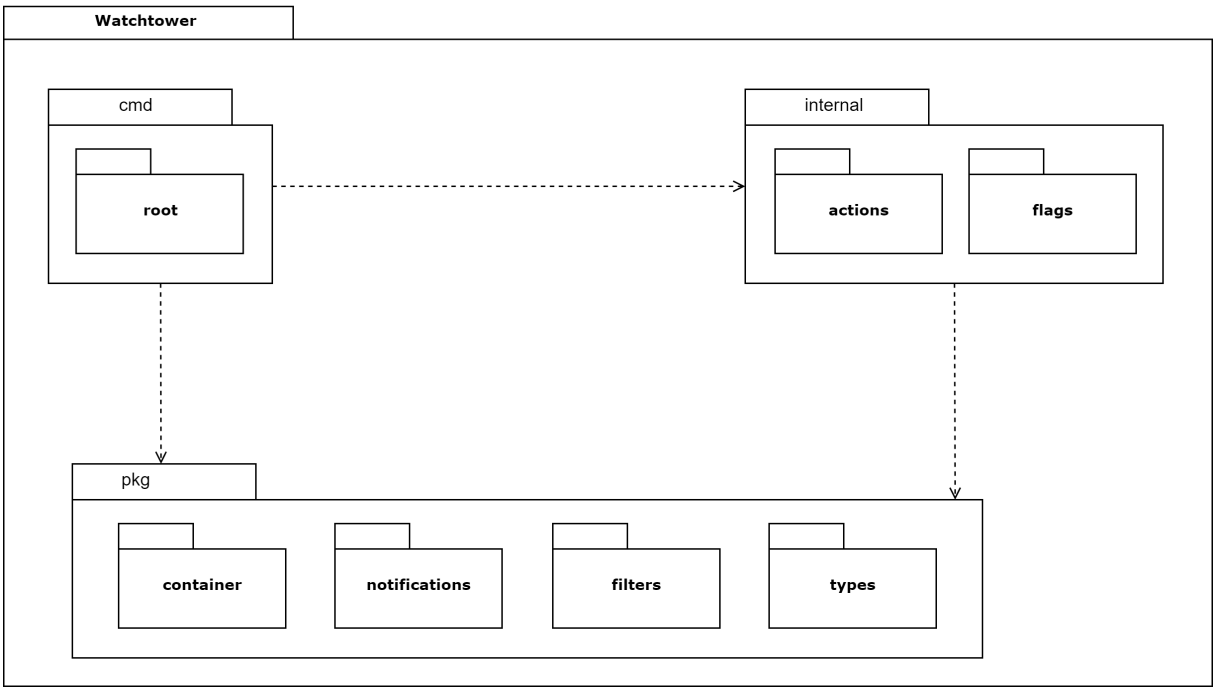


Figure 1 – Watchtower 0.3.11 Package Diagram

The **cmd** package was the main package. It made use of **internal** and **pkg** to:

- Parse arguments from command line and environment variables;
- Configure the instance behaviour based on the parsed arguments;
- Initialize the update scheduler.

Most of Watchtower's logic was placed under **pkg**, as **internal** contained only the declaration of acceptable arguments, internal static actions that prepared the running environment and utility functions. The **pkg** package grouped all of the needed sub-packages that enabled the core features. For instance:

- **container** included the integration with the Moby Project (MOBY, 2017) and exposed a structure called **Container**, which was a Docker container abstraction in Golang;
- **notifications** included the integration with external messaging services to notify users whenever a new update had been made;
- **filters** included multiple filters that could be used to fetch the monitored containers based on its labels - e.g, the filter **FilterByEnableLabel**, used to fetch all containers that had a label that indicated that it should be monitored;
- **types** included custom built structures that worked as interface from multiple other entities in the code - e.g., the type **FilterableContainer**, used as an interface that added utility methods to the original **Container** type.

As the proposed HTTP API would be implemented as a function that exposed a feature, instead of an utility, the chosen approach to its inclusion into Watchtower architecture was to develop a sub-package called **api** that would be placed under **pkg**, as shown in Figure 2:

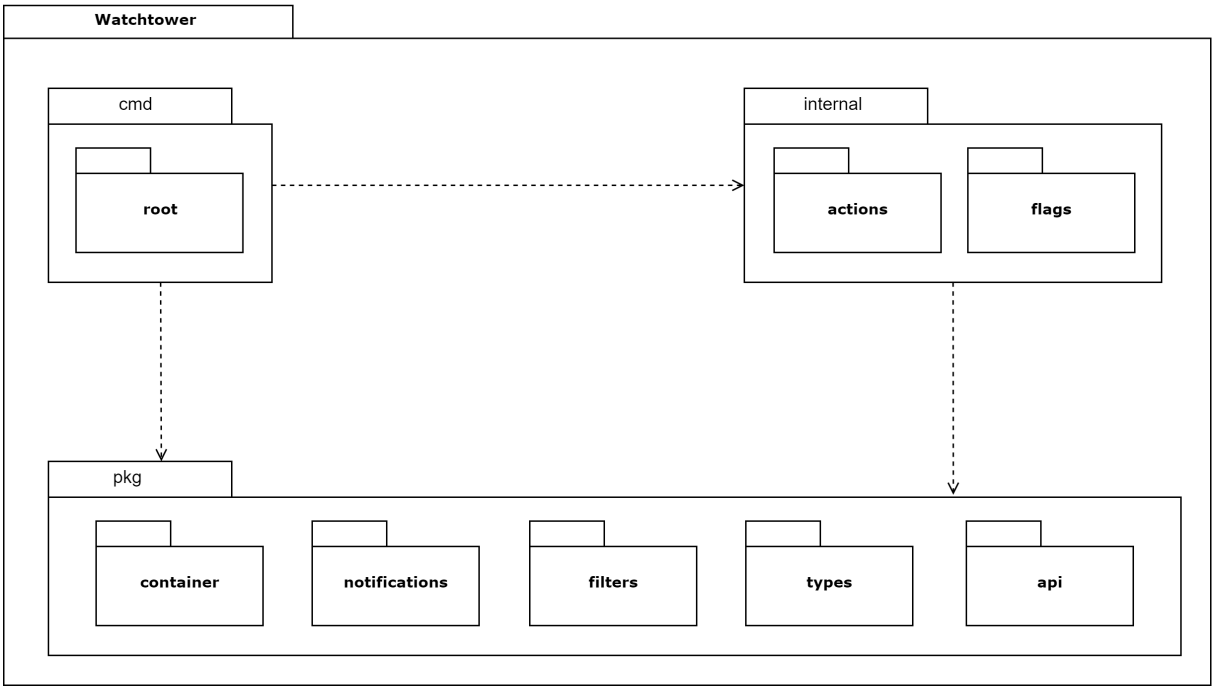


Figure 2 – Watchtower 1.0.0 Package Diagram

In the **api** sub-package, a function named **SetupHTTPUpdates** was implemented.

```
1 func SetupHTTPUpdates(apiToken string, updateFunction func()) error
```

Listing 4.2 – SetupHTTPUpdates signature (CONTAINRRR, 2020c)

As shown in Listing 4.2, **SetupHTTPUpdates** expected the following arguments:

- **apiToken**: a *string* type value that contained the authentication token that each incoming request should include in its headers;
- **updateFunction**: a *func* type value that contained the function to be run whenever an authorized request reached the instance.

With the listed arguments, **SetupHTTPUpdates** exposed the endpoint *"/v1/update"* that, when requested, would authenticate the request with the given token and execute the given update function if an update wasn't already running, as shown in Listing 4.3:

```
http.HandleFunc(
    "/v1/update",
    func(w http.ResponseWriter, r *http.Request) {
     log.Info(
          "Updates triggered by HTTP API request."
     )

     if r.Header.Get("Token") != apiToken {
      log.Println("Invalid token. Not updating.")
    return
     }

     log.Println("Valid token found. Attempting to update.")

     select {
      case chanValue := <-lock:
          defer func() { lock <- chanValue }()
          updateFunction()
      default:
          log.Debug("Another update already running.")
     }

    }
)
```

Listing 4.3 – SetupHTTPUpdates function (CONTAINRRR, 2020c)

The *http.HandleFunc* call, shown in Listing 4.3, receives two arguments. The first is a *string* type that defines the path in which this handle will be located. The second is the function that will handle the incoming requests. Notice that the shown handle function is an anonymous function that calls the **updateFunction** in a given moment. This setup was needed due to a few technical peculiarities.

The first one is related to the arguments received by the handler function. The **updateFunction** arguments couldn't be expected to receive HTTP requests related types, as *http.ResponseWriter* and **http.Request*. So an intermediate function that received these types of argument was needed. Imposing that the **updateFunction** would need to accept HTTP related types would move the request processing related responsibilities out from the **api** package. This decision impacted on the software's maintainability, as future contributors won't have to deal with HTTP related types outside the HTTP API sub-package.

Besides improving the maintainability, it was necessary to deal with the asynchronous nature of HTTP requests. When a request is being processed, there is no guar-

antee that the processing will be finished when the next request arrives. As the container update operation should be atomic, there cannot be two **updateFunction** calls running simultaneously for the same group of containers, so the defined anonymous function also had to deal with this.

To prevent multiple updates from running at the same time, a Golang type called *Channel* was used to control the execution of the multiple possible **updateFunction** calls. Whenever a request reaches the Golang HTTP server, defined by the native package *http*, a Goroutine is spawned to run the handler function. Goroutines are implemented as lightweight threads managed by the Go runtime (GOLANG, 2020b). As defined in (GOLANG, 2020a), Channels are shareable among multiple Goroutines and can be used to lock the execution of other operations if one is already running. Whenever the handler is run, it attempts to capture an integer value from the *Channel*. If there is a valid value to fetch, it is removed from the *Channel*, preventing other handlers to capture it, and, after the operation is done, it is returned into the *Channel* again. If there is no valid value to fetch, the handler won't proceed with the **updateFunction** call, preventing it to run concurrently.

After implementing the HTTP API core within the **api** package, it was necessary to change the **cmd** and **internal** packages to include a new argument that could be used to enable the implemented feature and disable the default interval based monitoring. As shown in Listing 4.4, there were created two new possible arguments for Watchtower:

- A boolean argument which presence would enable the HTTP API mode;
- A string argument which would contain the authorization token to be matched with each incoming request authorization header.

And, as shown in 4.5, the presence of each of the new arguments would impact on how the **cmd** package sets the instance behaviour. The **runUpdatesWithNotifications** function was the scheduled function for running interval based updates. As the goal is to adapt the behaviour to run the same process but without the scheduling, it has been set as the **updateFunction** argument of the HTTP API handler function. So, whenever a new request reaches the server, the update will be run as if it had been normally scheduled, without great differences from the default behaviour other than the update trigger.

```
flags.BoolP(
  "http-api",
  "",
  viper.GetBool("WATCHTOWER_HTTP_API"),
  "Runs Watchtower in HTTP API mode,
      so that image updates must be
      triggered by a request"
    )

flags.StringP(
  "http-api-token",
  "",
  viper.GetString("WATCHTOWER_HTTP_API_TOKEN"),
  "Sets an authentication token to
      HTTP API requests."
    )
```

Listing 4.4 – HTTP API Watchtower flags (CONTAINRRR, 2020b)

```
if httpAPI {
    apiToken, _ := c.PersistentFlags().GetString(
        "http-api-token"
    )

    if err := api.SetupHTTPUpdates(
        apiToken,
        func() {
            runUpdatesWithNotifications(filter)
        }); err != nil {
          log.Fatal(err)
          os.Exit(1)
    }
  api.WaitForHTTPUpdates()
}
```

Listing 4.5 – HTTP API impact on Watchtower's root file (CONTAINRRR, 2020d)

Along with the code increments, some documentation to assure users have the needed resources to make use of the HTTP API was also written. The documentation included an usage example as shown in Appendix B.

After reviewing the code, members of the community considered that the feature was mature enough to be included in Watchtower v1.0.0. The code was merged and the feature made available to all users.

4.1.3 Experiment Development

As defined previously, there must be an experiment which involves two deployment strategies:

- Traditional Watchtower interval-based deployment;
- Modified Watchtower API-based deployment.

For a deployment to happen, there must be a deployable application. To set it up, there was created a tool called Python Watchtower Deployable (MOURA, 2019b), which is an isolated, containerized and independent Python 3.7 application that runs in a loop and keeps the container alive. The only purpose of this application is to be deployable by Watchtower.

With an application to deploy, it is possible to design the testing framework. Since Watchtower requires an image hosted in a Docker Registry, a Docker Compose file and it usually is set up alongside a continuous integration pipeline, these must exist.

The Watchtower Experiment 1 (MOURA, 2019c) repository has a GitLab mirror, a GitLabCI configuration file and a Docker Compose setup which runs Python Watchtower Deployable and Watchtower in the same stack. Its GitLabCI configuration, as shown in 4.6, presents only one stage and one job, responsible for building and pushing the deployable application image to DockerHub.

It is important to notice a few peculiarities of this pipeline. In order to build and push an image to DockerHub, the running job must execute Docker CLI. Since GitLabCI runs their jobs inside Docker containers, there must be the use of a service called **docker:dind**. This service, as explained in (DOCKER, 2019c), allows the use of Docker inside a Docker container.

Another aspect of the job is the fact that, in order to login to DockerHub via CLI, there is the use of environment variables, so no critical information is written in plain text and made public. When the pipeline finishes with no failures, the expected result is the latest version of Python Watchtower Deployable made available in its DockerHub organization.

With the image uploaded to the registry, it becomes usable in any Docker Compose configuration file, as shown in Listing 4.7, with a 30s interval based configuration, and in Listing 4.8, with the HTTP API configuration.

```
image: docker:stable

stages:
  - build

services:
  - docker:dind

build and push to registry:
  stage: build
  script:
    - docker login -u $DOCKERHUB_USERNAME -p $DOCKERHUB_PASSWORD
    - docker build -t victorcmoura/python-watchtower-deployable .
    - docker push victorcmoura/python-watchtower-deployable
```

Listing 4.6 – Watchtower Experiment 1 GitLabCI configuration file (MOURA, 2019c)

```
version: '3'

services:
  python-watchtower-test:
    image: victorcmoura/python-watchtower-deployable
    environment:
      - PYTHONUNBUFFERED=1
    labels:
      - "com.centurylinklabs.watchtower.enable=true"

  watchtower:
    image: containrrr/watchtower
    volumes:
      - /var/run/docker.sock:/var/run/docker.sock
    command: --interval 30
    labels:
      - "com.centurylinklabs.watchtower.enable=false"
```

Listing 4.7 – Watchtower Experiment 1 Docker Compose configuration file for interval based updates (MOURA, 2019c)

```
version: '3'

services:
  python-watchtower-test:
    image: victorcmoura/python-watchtower-deployable
    environment:
      - PYTHONUNBUFFERED=1
    labels:
      - "com.centurylinklabs.watchtower.enable=true"

  watchtower:
    image: victorcmoura/watchtower
    volumes:
      - /var/run/docker.sock:/var/run/docker.sock
    command: --debug --http-api
    environment:
      - WATCHTOWER_HTTP_API_TOKEN=lk123g
    labels:
      - "com.centurylinklabs.watchtower.enable=false"
    ports:
      - 8080:8080
```

Listing 4.8 – Watchtower Experiment 1 Docker Compose configuration file for HTTP API based updates (MOURA, 2019c)

Notice the **PYTHONUNBUFFERED** environment variable, set to disable Python's standard output buffer and allow text to be printed by the container while it is on an infinite loop, and Watchtower related settings:

- –**debug** argument: to allow the monitoring of Watchtower's behaviour during runtime (CONTAINRRR, 2019a);
- –**interval** argument: to allow the customization of the monitoring interval (CONTAINRRR, 2019b);
- –**http-api** argument: to enable the HTTP API operation mode (CONTAINRRR, 2020a);
- **WATCHTOWER_HTTP_API_TOKEN** environment variable: to set the authorization token to be matched with the incoming request authorization headers (CONTAINRRR, 2020a);
- **enable** labels: to set which services should be monitored by Watchtower (CONTAINRRR, 2019d).

With this setup, it is possible to run the containers with Docker Compose, commit to the GitHub repository and monitor, through Watchtower logs, when and how the continuous deployment is given. Through these logs, it is possible to measure the following metrics:

Data exchanged to decide for a deployment or not (kilobytes of data): It is possible to measure how much data has been downloaded from DockerHub and how much data has reached Watchtower's HTTP API by using command line tools that monitor the network interfaces and running processes. This metric doesn't consider the downloaded image data, only the requests made prior to the container substitution operation.

Deployments made by Watchtower (number of container substitutions after an update): The number of deployments can be measured by counting the event occurrences on the debug log.

Deployment attempts made by Watchtower (number of updates): The number of deployment attempts can also be measured by counting the event occurrences on the debug log.

Deployments made to deployment attempts ratio (number of container substitutions/number of updates): Derived from two other metrics, this aims to measure the relation between application deployments and update operations made by Watchtower. Values closer to 1 represent a more efficient operation mode, as most deployment attempts result in an actual deployment.

4.2 Experiment Results

After designing and implementing the experiment, a cloud hosted virtual machine was deployed to be used as the testing platform. The hardware specifications were:

- **CPU:** 1 vCPU of a shared Intel Xeon E5 v3 (Haswell) with 2.3GHz base clock;
- **RAM:** 600 MB;
- **Storage:** 10 GB SSD;
- **Network:** Egress up to 1 Gbps and Ingress of up to 20 Gbps.

Measurements were divided into two testing phases: The first one, that was used to measure the **Data exchanged to decide for a deployment or not**, and the second, to measure the remaining metrics.

In order to isolate the measurement of **Data exchanged to decide for a deployment or not**, no changes to the deployed Docker image were made in this first

testing phase. Without changes, all of the data exchanged by Watchtower and Docker processes would contain only requests exchanged to decide for a deployment or not - and always resulting in a non update scenario. To measure the HTTP API version without a deployment pipeline, the trigger requests were manually executed through a command line tool.

During the first and the second testing phases, none of the hardware limits were reached and, therefore, may not have impacted on the final measurements.

4.2.1 Testing Results

To measure the data exchange values, the command line tool *nethogs* (RABOOF, 2020) was used to monitor the *dockerd* and *watchtower* processes, which were the local Docker Daemon and Watchtower processes, respectively. The command line tool captures the sum of data received and sent by each process.

Given that:

- Each sum S_t represents the exchanged data until the time period with index t;
- The difference $S_t - S_{t-1}$ represents the data exchanged in an operation that occurred between times t and $t-1$;
- n is the number of operations to be measured.

The function $f(n)$, shown in 4.1, that represents the average data exchange per strategy can be defined as:

$$f(n) = \frac{\sum_{t=1}^{n+1} S_t - S_{t-1}}{n} \tag{4.1}$$

For **Data exchanged to decide for a deployment or not (megabytes of data)**, it should be noticed that the HTTP API strategy value included the data exchanged by the trigger request, which was, on average, a 0.001 MB sized request.

For the deployment related metrics, the measurements were made in fixed periods of 10min with a single change to the deployed image during the operation, so each value corresponds to the amount of events during a total operation time of 10 minutes and assuming that the remote registry received only a single image change during this time frame.

Table 1 contains all of the measured results for each operation strategy.

4.2.2 Experiment conclusion

Analysing only **Average data exchanged per operation**, the HTTP API strategy presented more data exchanging than the polling strategy. This result was already expected, as with the HTTP API strategy, there is the inclusion of the extra trigger request required by the solution to begin a deployment attempt.

Despite requiring more data to be exchanged, the HTTP API approach, as expected, made less deployment attempts than the polling strategy. For the 10 minutes time frame, there was a reduction of 95%, which resulted in the best efficiency based on the **Deployments made to deployment attempts ratio** value of 1.00, as the developed strategy makes a deployment attempt only when a new image push is made to the remote registry.

Given the results and the technical requirements of the setup, it is possible to assume the following statements:

- The polling interval strategy makes one update attempt every 30s;
- The HTTP API strategy makes one update attempt only when a new image is pushed to the remote registry;
- There won't be changes, during the operation time frame, on how much data is exchanged during each update attempt operation, as the Watchtower image is not being updated.

Given these statements, with 1 image push per day to the remote registry and the gathered values, it is possible to calculate the possible results for **Data exchanged to decide for a deployment or not** with multiple time frames of operation, shown in Table 2.

The calculated values show that, for long term operations, the benefit of the HTTP API approach can be significantly greater, with up to 99.963% less data exchanged in a year of operation.

4.2.3 Challenges and Limitations

Even though the planned scope has been covered and the solution has presented significant improvements, much of the community started to use Watchtower due to its active behaviour. So, as the HTTP API is not enabled by default, those who came to notice it are, by the time of this work, still a minority. Despite of that, newcomers with specific use cases, that require them to use the implementation, are being oriented to do so and it is expected for the number of adopters to grow over time.

Also, one limitation of the developed feature is the need for a network configuration that allows the host to receive requests from outside sources. As the HTTP API has been conceived to make Watchtower reachable by external services, as public continuous integration platforms, a public IP and the correct firewall adjustments might be required. Without the permission or possibility to expose the trigger endpoint, the feature cannot be used.

Table 1: Collected metrics for each operation strategy

-	**Polling strategy**	**HTTP API strategy**
Average data exchanged per operation	0.025 MB	0.026 MB
Deployment attempts made by Watchtower	20	1
Deployments made by Watchtower	1	1
Deployments made to deployment attempts ratio	0.05	1.00

Table 2: Data exchanged in multiple time frames for each operation strategy

Time frame	**Data exchanged (Polling Strategy)**	**Data exchanged (HTTP API)**
1 day	72.0 MB	0.026 MB
1 month (30 days)	2160.0 MB	0.78 MB
1 year (365 days)	26280.0 MB	9.49 MB

5 Conclusion

As expected, the HTTP API presented itself as a more performative approach than the default polling strategy, as there are not unnecessary update attempts that do not result in a deployment of a new image. With the spawning of the update operation occurring only when trigger requests have to be processed and less usage of network resources, the solution has been proven to be beneficial in real production stacks monitored by Watchtower.

Since Watchtower is an open-source software, the developed feature can be extended by any member of the community. The possible future improvements of this work includes:

- Single container updates through individual endpoints, so one can update a single container without triggering a deployment attempt for all of the monitored containers;
- An endpoint to rollback to older containers, enabling quick reverts if the updated container malfunctions;
- Endpoints to fetch Watchtower metadata, which could be used by monitoring solutions to provide real time metrics to external clients.

Besides future endpoint implementations, the authorization back-end can also be improved to provide dynamic access tokens, which could be updated on each client session, temporary permissions and different access levels for each client.

Also, due to the architectural and usability decisions taken during the development phase, current users of all versions of Watchtower won't need to make any changes to their already working setups and, users who are running versions 1.0 or greater can enable the HTTP API feature to benefit from the improvements designed, implemented and shown in this work.

Bibliography

CDRX et al. *Rancher GitLab Deployment Tool*. 2019. Available at: <https://github.com/cdrx/rancher-gitlab-deploy>. Accessed on: Oct, 28. 2019. Cited on page 28.

CIRCLECI. *CircleCI*. 2019. Available at: <https://circleci.com/>. Accessed on: Oct, 14. 2019. Cited on page 24.

CONTAINRRR. *Watchtower*. 2015. Available at: <https://github.com/containrrr/watchtower>. Accessed on: Sep, 6. 2019. Cited 4 times on pages 17, 25, 27, and 30.

CONTAINRRR. *Watchtower License*. 2018. Available at: <https://github.com/containrrr/watchtower/blob/master/LICENSE.md>. Accessed on: Nov, 24. 2020. Cited on page 29.

CONTAINRRR. *Watchtower Arguments - Debug*. 2019. Available at: <https://containrrr.github.io/watchtower/arguments/#debug>. Accessed on: Oct, 31. 2019. Cited on page 44.

CONTAINRRR. *Watchtower Arguments - Poll Interval*. 2019. Available at: <https://containrrr.github.io/watchtower/arguments/#poll_interval>. Accessed on: Oct, 31. 2019. Cited on page 44.

CONTAINRRR. *Watchtower Container Selection*. 2019. Available at: <https://github.com/containrrr/watchtower/blob/master/docs/container-selection.md>. Cited on page 25.

CONTAINRRR. *Watchtower Container Selection*. 2019. Available at: <https://containrrr.github.io/watchtower/container-selection/>. Accessed on: Oct, 31. 2019. Cited on page 44.

CONTAINRRR. *Watchtower Code of Conduct*. 2020. Available at: <https://github.com/containrrr/watchtower/blob/master/code_of_conduct.md>. Accessed on: Nov, 24. 2020. Cited on page 30.

CONTAINRRR. *Watchtower Contributing*. 2020. Available at: <https://github.com/containrrr/watchtower/blob/master/CONTRIBUTING.md>. Accessed on: Nov, 24. 2020. Cited on page 29.

CONTAINRRR. *Watchtower HTTP API Mode Documentation*. 2020. Available at: <https://containrrr.dev/watchtower/http-api-mode/>. Accessed on: Nov, 28. 2020. Cited 3 times on pages 11, 44, and 60.

CONTAINRRR. *Watchtower's Flags Package*. 2020. Available at: <https://github.com/containrrr/watchtower/blob/master/internal/flags/flags.go>. Accessed on: Nov, 28. 2020. Cited on page 41.

CONTAINRRR. *Watchtower's HTTP API Package*. 2020. Available at: <https://github.com/containrrr/watchtower/blob/master/pkg/api/api.go>. Accessed on: Nov, 28. 2020. Cited 2 times on pages 38 and 39.

CONTAINRRR. *Watchtower's Root Package*. 2020. Available at: <https://github.com/containrrr/watchtower/blob/master/cmd/root.go>. Accessed on: Nov, 28. 2020. Cited on page 41.

DOCKER. *Docker*. 2019. Available at: <https://www.docker.com/>. Accessed on: Nov, 3. 2019. Cited on page 17.

DOCKER. *What is a Container?* 2019. Available at: <https://www.docker.com/resources/what-container>. Accessed on: Oct, 9. 2019. Cited on page 20.

DOCKER. *What is Docker in Docker?* 2019. Available at: <https://hub.docker.com/_/docker>. Accessed on: Oct, 31. 2019. Cited on page 42.

DOCKER. *Docker Pricing Subscriptions for Individuals Teams*. 2020. Available at: <https://www.docker.com/pricing>. Accessed on: Nov, 18. 2020. Cited on page 23.

DOCKERDOCS. *Best practices for writing Dockerfiles*. 2019. Available at: <https://docs.docker.com/develop/develop-images/dockerfile_best-practices/>. Accessed on: Oct, 28. 2019. Cited on page 21.

DOCKERDOCS. *Deploy a registry server*. 2019. Available at: <https://docs.docker.com/registry>. Accessed on: Sep, 6. 2019. Cited on page 22.

DOCKERDOCS. *Docker Swarm*. 2019. Available at: <https://docs.docker.com/engine/swarm/>. Accessed on: Oct, 9. 2019. Cited on page 21.

DOCKERDOCS. *Get Started, Part 2: Containerizing an Application*. 2019. Available at: <https://docs.docker.com/get-started/part2/>. Accessed on: Oct, 28. 2019. Cited on page 21.

DOCKERDOCS. *Download rate limit*. 2020. Available at: <https://docs.docker.com/docker-hub/download-rate-limit/>. Accessed on: Nov, 18. 2020. Cited on page 23.

DYCK, A.; PENNERS, R.; LICHTER, H. Towards definitions for release engineering and devops. *IEEE/ACM 3rd International Workshop on Release Engineering*, 2015. Available at: <https://ieeexplore.ieee.org/document/7169442>. Accessed on: Oct, 9. 2019. Cited on page 19.

ERICH, F.; AMRIT, C. A qualitative study of devops usage in practice. *Journal of Software: Evolution and Process*, 2017. Available at: <https://onlinelibrary.wiley.com/toc/20477481/2017/29/6>. Accessed on: Sep, 17. 2019. Cited on page 19.

GIT. *Git*. 2019. Available at: <https://git-scm.com/>. Accessed on: Nov, 18. 2019. Cited on page 30.

GITHUB. *GitHub Actions*. 2019. Available at: <https://github.com/features/actions>. Accessed on: Oct, 28. 2019. Cited on page 27.

GITHUB. *Collaborating with Issues and Pull Requests*. 2020. Available at: <https://docs.github.com/en/free-pro-team@latest/github/collaborating-with-issues-and-pull-requests>. Accessed on: Nov, 24. 2020. Cited 2 times on pages 30 and 34.

GITHUB. *Fork a Repo*. 2020. Available at: <https://docs.github.com/en/free-pro-team@latest/github/getting-started-with-github/fork-a-repo>. Accessed on: Nov, 24. 2020. Cited on page 30.

GITHUB. *How to Contribute to Open Source*. 2020. Available at: <https://opensource.guide/how-to-contribute/>. Accessed on: Nov, 24. 2020. Cited on page 29.

GITLABDOCS. *GitLab CI/CD*. 2019. Available at: <https://docs.gitlab.com/ee/ci/>. Accessed on: Oct, 14. 2019. Cited 2 times on pages 24 and 27.

GOLANG. *The Go Programming Language*. 2019. Available at: <https://golang.org/>. Accessed on: Nov, 18. 2019. Cited on page 25.

GOLANG. *A Tour of Go - Channels*. 2020. Available at: <https://tour.golang.org/concurrency/2>. Accessed on: Nov, 28. 2020. Cited on page 40.

GOLANG. *A Tour of Go - Goroutines*. 2020. Available at: <https://tour.golang.org/concurrency/1>. Accessed on: Nov, 28. 2020. Cited on page 40.

JENKINS, J. *Velocity 2011: Jon Jenkins, "Velocity Culture"*. 2011. Available at: <https://www.youtube.com/watch?v=dxk8b9rSKOo>. Accessed on: Oct, 9. 2019. Cited on page 17.

KUBERNETES. *Kubernetes Homepage*. 2019. Available at: <https://kubernetes.io/pt/>. Accessed on: Oct, 28. 2019. Cited on page 17.

LAPPIS. *Tais*. 2017. Available at: <https://github.com/lappis-unb/tais>. Accessed on: Oct, 14. 2019. Cited on page 24.

LAPPIS. *Lappis Learning*. 2018. Available at: <https://github.com/lappis-unb/lappis-learning>. Accessed on: Oct, 14. 2019. Cited 2 times on pages 24 and 28.

LAPPIS. *Lappis Learning GitLabCI configs*. 2018. Available at: <https://github.com/lappis-unb/lappis-learning/blob/master/.gitlab-ci.yml>. Accessed on: Oct, 28. 2019. Cited on page 28.

LEITE, L. et al. A survey of devops concepts and challenges. 2019. Available at: <https://arxiv.org/abs/1909.05409>. Accessed on: Oct, 7. 2019. Cited on page 19.

MOBY. *Moby Project*. 2017. Available at: <https://github.com/moby/moby>. Accessed on: Oct, 11. 2019. Cited 2 times on pages 25 and 37.

MOURA, V. *Go Webhook Server*. 2019. Available at: <https://github.com/victorcmoura/go-webhook-server>. Accessed on: Nov, 1. 2019. Cited 2 times on pages 35 and 36.

MOURA, V. *Python Watchtower Deployable*. 2019. Available at: <https://hub.docker.com/r/victorcmoura/python-watchtower-deployable>. Accessed on: Oct, 31. 2019. Cited on page 42.

MOURA, V. *Watchtower Experiment 1*. 2019. Available at: <https://github.com/victorcmoura/watchtower-experiment-1>. Accessed on: Oct, 31. 2019. Cited 3 times on pages 42, 43, and 44.

MOURA, V. *Watchtower HTTP API based updates*. 2020. Available at: <https://github.com/containrrr/watchtower/pull/432>. Accessed on: Nov, 25. 2020. Cited on page 33.

MOURA, V. *Watchtower HTTP API proposal*. 2020. Available at: <https://github.com/containrrr/watchtower/issues/429>. Accessed on: Nov, 25. 2020. Cited on page 33.

OLSSON, H. H.; ALAHYARI, H.; BOSCH, J. Climbing the "stairway to heaven" a multiple-case study exploring barriers in the transition from agile development towards continuous deployment of software. *2012 38th Euromicro Conference on Software Engineering and Advanced Applications*, Gothenburg, Sweden, 2012. Cited on page 23.

RABOOF. *Nethogs*. 2020. Available at: <https://github.com/raboof/nethogs>. Accessed on: Nov, 29. 2020. Cited on page 46.

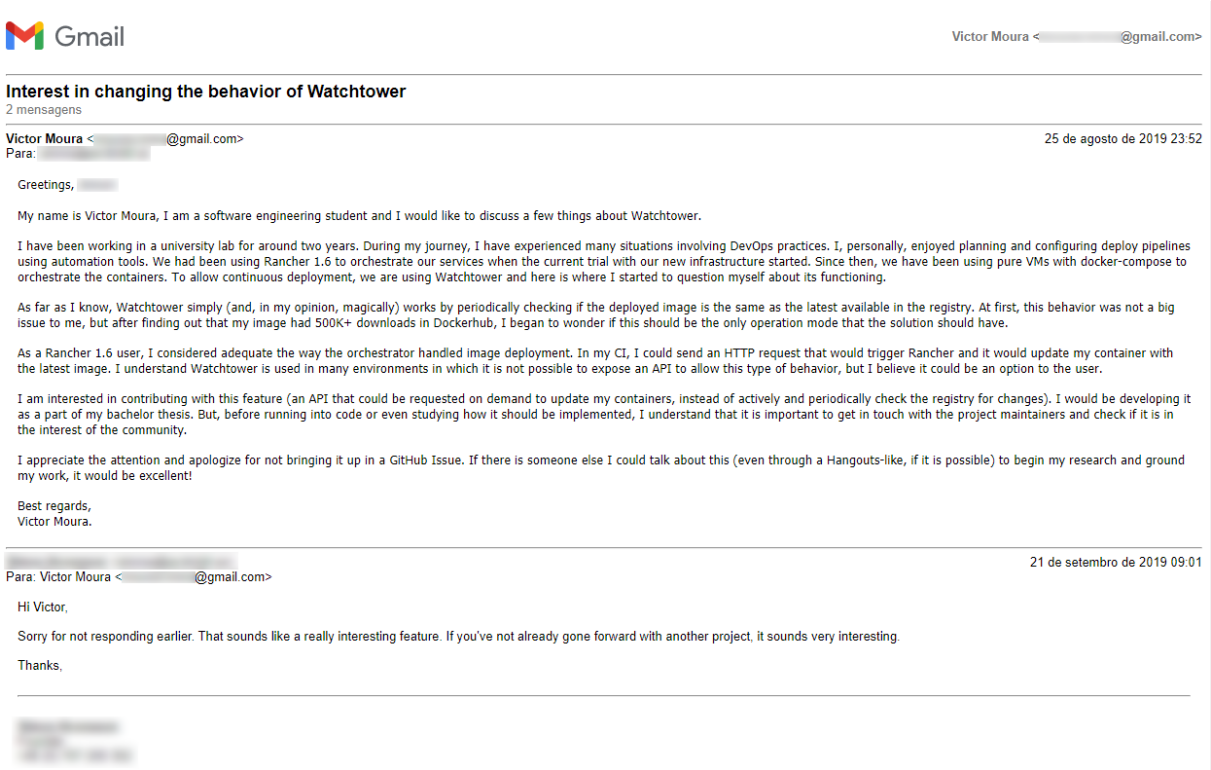
RANCHER. *How to use the Rancher API*. 2019. Available at: <https://rancher.com/docs/rancher/v1.6/en/api/v2-beta/>. Accessed on: Nov, 18. 2019. Cited on page 34.

RANCHER. *Overview of Rancher*. 2019. Available at: <https://rancher.com/docs/rancher/v1.4/en/>. Accessed on: Oct, 28. 2019. Cited 3 times on pages 17, 27, and 31.

TRAVISCI. *TravisCI*. 2019. Available at: <https://travis-ci.org/>. Accessed on: Oct, 14. 2019. Cited on page 24.

Appendix

APPENDIX A – First Approach to the Community



Gmail — Victor Moura <[redacted]@gmail.com>

Interest in changing the behavior of Watchtower

2 mensagens

Victor Moura <[redacted]@gmail.com> — 25 de agosto de 2019 23:52
Para: [redacted]

Greetings, [redacted]

My name is Victor Moura, I am a software engineering student and I would like to discuss a few things about Watchtower.

I have been working in a university lab for around two years. During my journey, I have experienced many situations involving DevOps practices. I, personally, enjoyed planning and configuring deploy pipelines using automation tools. We had been using Rancher 1.6 to orchestrate our services when the current trial with our new infrastructure started. Since then, we have been using pure VMs with docker-compose to orchestrate the containers. To allow continuous deployment, we are using Watchtower and here is where I started to question myself about its functioning.

As far as I know, Watchtower simply (and, in my opinion, magically) works by periodically checking if the deployed image is the same as the latest available in the registry. At first, this behavior was not a big issue to me, but after finding out that my image had 500K+ downloads in Dockerhub, I began to wonder if this should be the only operation mode that the solution should have.

As a Rancher 1.6 user, I considered adequate the way the orchestrator handled image deployment. In my CI, I could send an HTTP request that would trigger Rancher and it would update my container with the latest image. I understand Watchtower is used in many environments in which it is not possible to expose an API to allow this type of behavior, but I believe it could be an option to the user.

I am interested in contributing with this feature (an API that could be requested on demand to update my containers, instead of actively and periodically check the registry for changes). I would be developing it as a part of my bachelor thesis. But, before running into code or even studying how it should be implemented, I understand that it is important to get in touch with the project maintainers and check if it is in the interest of the community.

I appreciate the attention and apologize for not bringing it up in a GitHub Issue. If there is someone else I could talk about this (even through a Hangouts-like, if it is possible) to begin my research and ground my work, it would be excellent!

Best regards,
Victor Moura.

[redacted] — 21 de setembro de 2019 09:01
Para: Victor Moura <[redacted]@gmail.com>

Hi Victor,

Sorry for not responding earlier. That sounds like a really interesting feature. If you've not already gone forward with another project, it sounds very interesting.

Thanks,

[redacted]

Figure 3 – Email to the Core Maintainer

APPENDIX B – Designing and Implementing the Solution

Http api mode

Watchtower provides an HTTP API mode that enables an HTTP endpoint that can be requested to trigger container updating. The current available endpoint list is:

- `/v1/update` - triggers an update for all of the containers monitored by this Watchtower instance.

To enable this mode, use the flag `--http-api`. For example, in a Docker Compose config file:

```
version: '3'

services:
  app-monitored-by-watchtower:
    image: myapps/monitored-by-watchtower
    labels:
      - "com.centurylinklabs.watchtower.enable=true"

  watchtower:
    image: containrrr/watchtower
    volumes:
      - /var/run/docker.sock:/var/run/docker.sock
    command: --debug --http-api
    environment:
      - WATCHTOWER_HTTP_API_TOKEN=mytoken
    labels:
      - "com.centurylinklabs.watchtower.enable=false"
    ports:
      - 8080:8080
```

Notice that there is an environment variable named WATCHTOWER_HTTP_API_TOKEN. To prevent external services from accidentally triggering image updates, all of the requests have to contain a "Token" field, valued as the token defined in WATCHTOWER_HTTP_API_TOKEN, in their headers. In this case, there is a port bind to the host machine, allowing to request localhost:8080 to reach Watchtower. The following `curl` command would trigger an image update:

```
curl -H "Token: mytoken" localhost:8080/v1/update
```

Figure 4 – Watchtower HTTP API Documentation (CONTAINRRR, 2020a)