Crane: a local deployment tool for containerized applications

Jose Arcidiacono[1] [0000-0003-0300-5213], Patricia Bazán[2] [0000-0001-6720-345X], Nicolás del Río[3] [0000-0002-0889-0752] and Alejandra B. Lliteras[4, 5] [0000-0002-4148-1299]

[1] Universidad Nacional de La Plata, Facultad de Informática, LINTI, Calle 50 esquina 120S/N 2° piso, La Plata. Argentina
jarcidiacono@linti.unlp.edu.ar

[2] Universidad Nacional de La Plata, Facultad de Informática, LINTI, Calle 50 esquina 120S/N 2° piso, La Plata. Argentina
pbaz@info.unlp.edu.ar

[3] Universidad Nacional de La Plata, Facultad de Informática, Calle 50 esquina 120S/N, La Plata. Argentina
ndelrio@info.unlp.edu.ar

[4] Universidad Nacional de La Plata, Facultad de Informática, LIFIA, Calle 50 esquina 120S/N, 1º piso, La Plata. Argentina

[5] CICPBA, Calle 526 e/ 10 y 11, La Plata, Argentina
alejandra.lliteras@lifia.info.unlp.edu.ar

Abstract. Application deployment as one of the software development stages has become more complex in the presence of distributed architectures that involve a variety of tools, and, with them, configuration differences, versioning and communication protocols. Even when cloud services have contributed a solution in this sense, it is still difficult to deploy distributed applications in on-premise environments.

The container concept as packages that include the application code, its dependencies, libraries and services required for its correct execution, turns out to be an alternative for streamlining application deployment and it allows taking the virtualization concept to the operative system. However, it adds a software layer that requires monitoring and management.

There are robust solutions for administering and monitoring containers but they also require computing resources that sometimes exceed the capacity of the average computer used for development, and they make local deployment difficult.

In this work, Crane, a tool for local deployment of containerized applications is presented. This tool has the characteristic of being lightweight, of general purpose and with automatic scaling capacities, which differentiates it from the Minikube tool, which allows some local Kubernetes API testing and is used mainly for the development of new features for the latter.

Keywords: Middleware Framework, Container Deployment, Distributed Services.

1 Introduction

The responsibility to deploy and monitor applications traditionally fell on an administrator that knew about infrastructure and networking, and was dedicated specifically to this task. With the advent of cloud services (PaaS, Platform as a Service) [1] monitoring and metrics became available to developers. In addition, with the advancement of continuous integration (CI) [2] and continuous delivery (CD) [3], a trend that proposes to test and deploy code as it is written, the deployment became closer to developers [4].

One aspect to be considered when deploying is the difference that exists between the developing, testing and production environments. This can lead to version problems –tool or library related-, configuration differences –e.g. connection timeouts-, and others.

One way to unify the environments is to use Docker[1], a container virtualization platform that allows creating "images" that include dependencies and configuration for an application. From a Docker image identical containers can be created, always with the same dependencies and initial configuration. Additionally, these containers can be parametrized. This means that each container can receive configurations as URLs to connect to, ports to open, and others. This configuration is in container creation time and it allows adapting (in a static way) the environment using console arguments or a simple text file. There are cloud platforms that support container use in production environments, both as unitary applications and as interconnected services.

When adding multiple containers the need to monitor and orchestrate them, that is, to define their start order and the dependencies between them, appears. One option to carry out these tasks automatically is Kubernetes[2], a container orchestration, monitoring and scaling platform. Kubernetes has the disadvantage of needing at least three virtual machines in order to work, and more machines can be added in order to increase the cluster count [5].

These specifications exceed the resources available in an average development machine, and because of that, are not suited for local deployment. There is a lightweight version, Minikube[3], that allows some local testing of the Kubernetes API and it is used mainly for development of new features for the main project.

In this context we present Crane, a local deployment tool for Docker-containerized applications that, unlike Minikube, is a general-purpose tool and it has an automatic scaling feature. The presented work is organized as follows: 1- In Section "Container management architecture precedents" some container management solutions with scaling are describer. 2- In Section "Design evolution of Crane", our container management tool, Crane, is presented. 3 –Section "Conclusions and future work" closes this work and extension points are enumerated.

---

1 https://www.docker.com/

2 https://kubernetes.io/

3 https://minikube.sigs.k8s.io/docs/

2 Container management architecture precedents

Even when container management and scaling is a new area and one in constant evolution, the investigation leading to this work reported an existing vacancy in terms of using this technology in one computer, through a management tool with capacity to scale for the administrator.

The state of art studied in this matter was exhaustive and went through several topics, but in order to address the Crane proposal, several solutions that allow container management and their scaling were surveyed and two of them were chosen for description and analysis: 1- SWITCH system [6] and 2- COCOS architecture [7], which is a Kubernetes extension. This selection is due both of them being complete architecture definitions addressing the problem.

2.1 SWITCH

SWITCH is an automatic scaling system for container based adaptable applications. In the article [6] presenting it, different metric types are studied for both vertical scaling (that is, resources increase) and horizontal scaling (instance count increase) [8] and an algorithm that adapts to different applications and resource usage patterns is proposed.

The SWITCH architecture consists of a load balancer to distribute the requests between the application instances, two monitoring mechanisms (container-level monitoring agent and application-level monitoring agent), a time series database, an alerting mechanism, and an adapting component that responds to such alerts. Additionally, a graphic user interface to configure thresholds and analyzing events is included.

Based on the metrics collected by the monitoring agents, both horizontal scaling (more containers) and vertical scaling (more resources in the same node) is performed.

As for the technologies mentioned in the article, the load balancer is HAProxy[4], the database is Apache Cassandra[5], and the monitoring, alerting and adapting component are developed in Java[6].

2.2 COCOS

Meanwhile, the COCOS architecture assumes that the monitoring part is already solved and it focuses on the control of containers. It has three control levels: container level [8], virtual machine level and cluster level [9].

In order to scale the containers, application metrics (response times, workload) that come from the containers themselves are used. The latter can also have Adapta-

4 http://www.haproxy.org/
5 https://cassandra.apache.org/_/index.htm
6 https://www.java.com

tion Hooks, code portions that allow the application to respond to scaling taking advantage of the new resources (for instance, increasing the thread count).

At the virtual machine level, there is a Supervisor for every one of the machines that handles the resource requests from containers (vertical scaling) and communicates with the next level in order to coordinate container scaling.

Lastly, at the cluster level there is an Orchestrator that manages the horizontal scaling of the containers (that is, the creation of new instances of containers and virtual machines).

3 Design evolution of Crane

DEHIA is a workflow manager for human-intervened data collection [10]. Its architecture is based on microservices. In a first delivery attempt at a local server, it proved to be complex and hard to replicate because of its various components and technologies.

A possible solution to this problem is containerization [8]. The simplest component, a gateway, was chosen to start. This component has no functional dependencies to the other components and it has no internal state.

The gateway component only needs one open port (in order to receive the requests it has to redirect) and it expects a small set of parameters. Because of that, it was viable to deploy it automatically. For this purpose it was decided to develop an automatic Docker deployment tool (named "Crane"), with the addition of scaling the application on-demand creating new instances.

3.1 Instances load balancing

Crane must be capable of interacting directly with the container platform, so that a first version was developed as a console script (bash, specifically) that uses Docker commands.

As a consequence of the new instance creation feature, it was necessary to add a load balancer that would make transparent the use of the component. For this purpose, a HTTP proxy (NGINX[7]) with load balancing capabilities, was used.

For portability and ease of use reasons, it was decided to use and configure the containerized version of the proxy. That way, this version of the tool is capable of creating not only instances, but also the proxy itself, configured for each component. NGINX was used because it communicates via HTTP. However, this restricts the tool to applications that share this feature. In addition to that, the application must be load-balancing-compatible (by having separate persistence means, by synchronizing them or not having them).

Inside the same script, an internal network is created for the containers to communicate with each other, leaving outbound access to the load balancer entrance. Fig.

[7] https://nginx.org/en/

1 shows this version of the tool, in which a console script (bash) interacts directly with Docker through console commands.

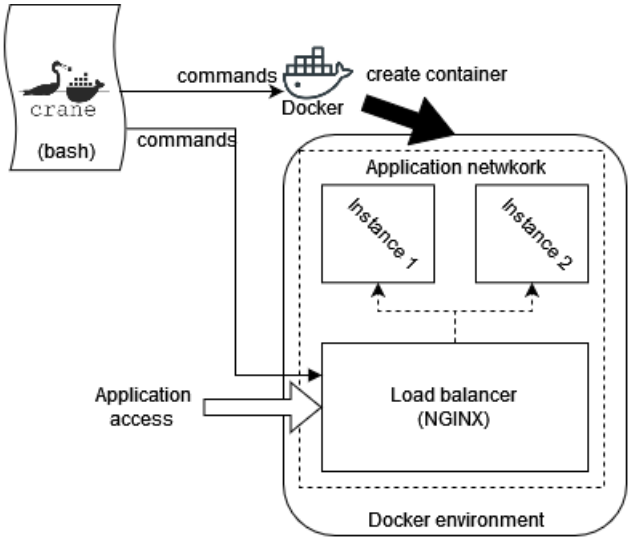

Fig. 1. First version of the tool, in Bash scripting

In Fig. 1, the Docker actions (black filled arrow), as a result of the commands sent by the script (simple arrow), create each of the containers inside the Docker environment, that is, the Load Balancer and the Instances.

Crate also interacts directly with the load balancer in order to notify it about the new instances created (also with a simple arrow) and that way directs correctly the requests that come from the outside (white filled arrow).

Inside the Docker environment, the load balancer allows access to both instances in a transparent way (dashed-line arrow).

Lastly, all the application containers belong to the same network (in a dashed-line square). This means that it could be multiple applications, each running in its own network and isolated from the others.

In order to modify the configuration of the load balancer each time an instance is created, a small script written in Python that waits for configuration updates was added. This updates are sent by the script itself via HTTP (specifically with the curl[8] command, a library and application for this purpose) and they affect the proxy configuration when it reloads as part of the script.

In short, to this point there is a console script that starts with three parameters: the name of the Docker image to be instantiated, the port where it must listen, and the container start parameters. When it starts, the container creates the following elements:

[8] https://curl.se/

- An inter-container network [11] where only the containers related to the application (instances and load balancer) will be connected.
- The first instance of the application, connected to the aforementioned network, but with no external access.
- The load balancer, configured to direct all the requests to the first instance. The Docker image of the load balancer was modified in order to receive the network location of the first instance as a parameter, and it also includes the Python script previously mentioned. It has two external access points: the application port that was received as a parameter by the console script, and the configuration port that waits for scaling instructions.

Table 1 shows each startup option for the script.

Table 1. Startup options for the Bash version of Crane

Option	Description
start (image, port, parameters)	It creates a inter-container network, the first instance of the application and the preconfigured load balancer
scale (identifier)	It creates a new instance of the application and configures the load balancer via HTTP
descale (identifier, instance)	It deletes the indicated instance and reconfigures the load balancer via HTTP

3.2 Container automatic scaling

Scaling this way is not practical because of two reasons: firstly, the administrator must decide when to scale in a manual way. Secondly, there are no metrics for the administrator to make such a decision.

Because of that, it was decided to make a second version of Crane that would scale automatically with an approximated rule (which is not in scope of this work) and that would also employ application use metrics in order to make the decision.

At this point it's interesting to notice that there are two types of use metrics [6]: 1- application metrics, including request count and response times, and others, and 2- infrastructure metrics, including CPU, memory, storage and network usage levels, and others.

For the first case, it can be measured by taking information in the application itself or at the load balancer, and it can be improved by scaling the application. For the second case, it can be measured by asking Docker or the operative system for information, and it can be improved by scaling resources (thinking of a virtual machine with elastic provisioning of CPU and memory).

As Crane is designed for local deployment in a personal computer, and there is no way of automatically increasing the resources, infrastructure metrics will not be considered.

To the moment, there wasn't a mechanism to collect application metrics, and because of that after researching the features of NGINX a module[9] was found. This module allows to keep track of the metrics that could be of interest for the tool (connection count, response times). Then, a new Docker image for the load balancer was designed. This image compiles NGINX with the nginx-module-vts module and includes the Python remote configuration script. In addition to that, the module was configured to listen in a third port (being the first one the access to the application and the second one the remote configuration) from where it returns the aforementioned metrics.

For the second part, that is, to have an approximated rule based in metrics from the load balancer in order to know when to scale, the following condition was set up: "when the sum of the average per second (rate) of the request count received in the last five minutes exceeds the value 0.1". This threshold set on 0.1, and also the five-minute window, are empirical values that allow for a high but manually achievable request rate to trigger the scaling (so it can be noticed).

This condition is useful when the right tool is available. This tool would, on one hand, keep track of the historical requests and, on the other hand, detect when the mentioned condition is met.

Prometheus and Alertmanager. In order to implement this automatic scaling, Prometheus[10], a time series database was used. This means that its specific functionality is to keep track of historical data in order to calculate statistics. On the other hand, it allows to set up alerts that are triggered when a particular condition is met. This "triggered" state implies that the alert is on the alert list in the "firing" state (active).

Prometheus has an external module called Alertmanager[11] whose purpose is to detect the alerts emitted by Prometheus and then trigger notifications through several mechanisms, of which the "webhooks" one is of interest.

A webhook is a URL, provided by the side interested in the alert, where it waits for an Alertmanager notification. This mechanism implies that the URL, that in this case would belong to the console deployment script, must be accessible by Alertmanager. In order to take advantage of using Alertmanager it was noticed that the script would need server capacities in order to receive the alerts, which makes the tool more complex. Fig. 2 shows the message sequence between the components that lead to the scaling of a container.

---

9 https://github.com/vozlt/nginx-module-vts

10 https://prometheus.io/

11 https://prometheus.io/docs/alerting/latest/alertmanager/

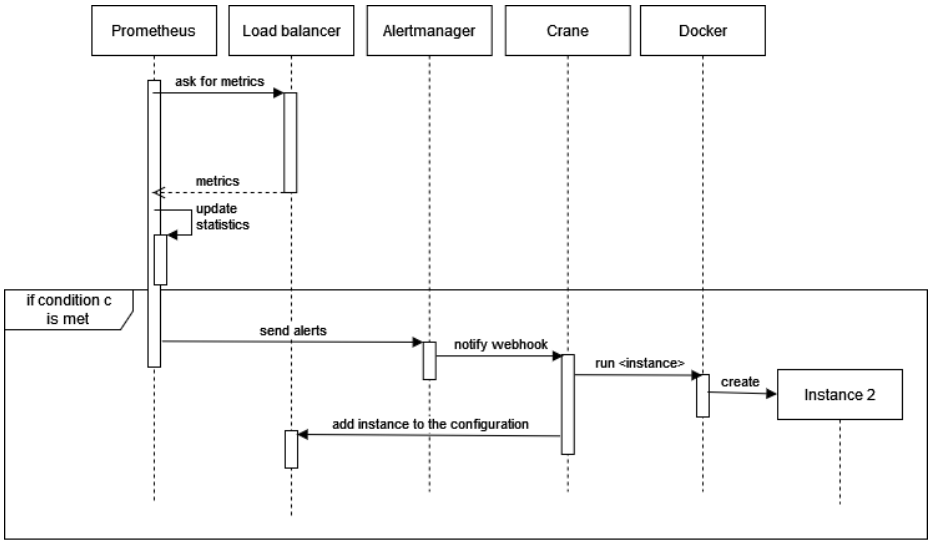

Fig. 2. – Automatic instance creation process

3.3 Detected implementation problems

When adding Prometheus and Alertmanager, a set of problems were detected, and that led to rethink the design of Crane from scratch.

Portability between installations. The difficulty level of reinstalling the full tool was considered.

Install the tool in another installation of the same operative system (Debian 11) or a Linux system with bash installed used to take just copying the scripts and installing Docker.

However, adding Prometheus and Alertmanager implies installing and configuring them, which is not trivial. One way to automate the configuration is to use the containerized versions of both tools, preconfigured to be instantiated by the script automatically. This led to an apparent conflict with the use of webhooks: if the tool is installed in the host system and Alertmanager is installed inside a container, the latter cannot send the notification because Docker containers have no access to the host system ports.

After more detailed research, it was found that from Docker version 20.10 (end of 2020) this is possible using a special DNS name and the –add-host parameter [12], so that the idea of containerize the module could continue.

For most of the components the containerized version was used, except the console script, which has to interact with Docker in the host system. Three alternatives were considered [13]:

- Docker-outside-of-Docker: it involves sharing the Docker socket with a container that has permissions on it. In this way Docker can be manipulated directly from a container. As the socket requires administrator rights on the host, it is a risky alternative.
- Docker-in-Docker [14]: it involves running Docker inside a container that generates another container inside itself. For this to work, the main container has to be in "privileged mode" (with administrator rights), which could also lead to one of the containerized applications to take control of the local host. Minikube uses this approach.
- Simply install the tool in the host, using the Docker HTTP interface to communicate with it.

Given the complexity and risks that the first two alternatives involve, the third one was chosen.

Prometheus must be configured so it can read the metrics of each load balancer (assuming multiple different applications) and store the alert condition.

In addition, Alertmanager must be configured to detect the alerts and to send the notification in the matching webhook. All this is done through configuration files internal to the filesystem of each container.

A first approach to the modification of these files from the tool was the use of Docker volumes[12], that allow synchronizing a file from the host system with another from the filesystem of the container.

These two modifications (the containerization of Prometheus and Alertmanager, and making use of volumes) allowed installing the tool from Debian 11 to Ubuntu 18.04 without reconfiguring it.

Figure 3 shows the evolution of the tool adding Prometheus and Alertmanager with their respective volumes.

[12] https://docs.docker.com/storage/volumes/

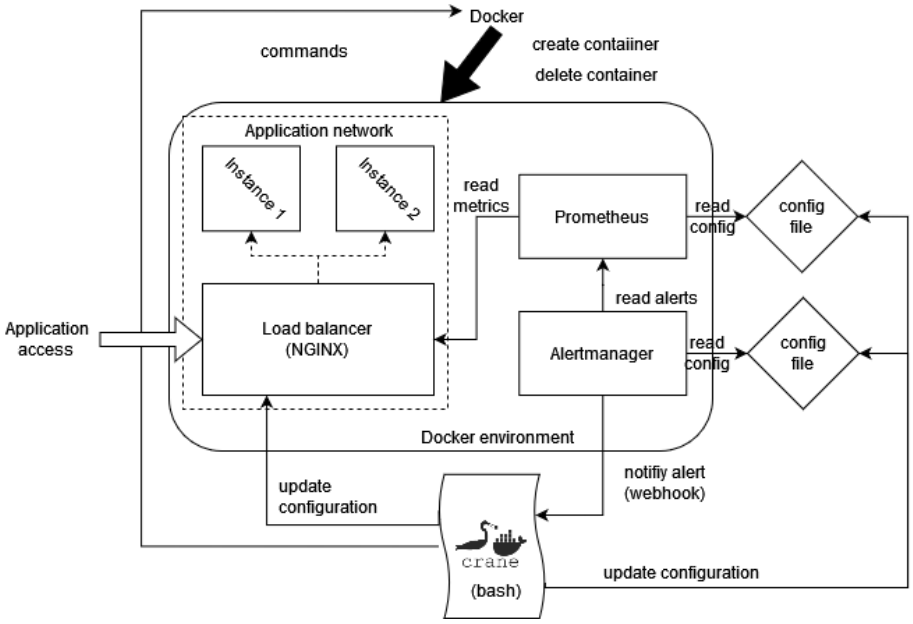

Fig. 3. Second version of Crane using Docker volumes

It can be noticed how the volumes require the files to be outside of the Docker environment and, because of that, they have access to the host filesystem.

Portability between different operating systems. It is of interest to increase the portability of the tool, taking advantage of Docker's multiplatform feature.

Two problems were considered: 1- bash console scripts are not instantly compatible with other systems like Windows (although there is a Linux subsystem, the intent was not to depend on the console) and 2- Docker volumes depend on the filesystem. Even when all major operating systems supporting Docker also support volumes, they are not necessarily compatible. [13]

The first problem was solved in two steps: first, the tool was migrated to use the Docker HTTP interface[14] through curl in the console. This decoupled the commands given to Docker from console commands. In a second step, the Python Docker SDK was used. This involved rewriting the tool in Python, which is multiplatform. At the moment the tool requires installing a Python interpreter, which although portable requires configuration and installing libraries. By using Cython[15], a Python extension for C language, a small self-contained executable can be generated.

This new version of the tool makes use of Flask[16], a minimalistic framework form web application development. The console format was abandoned for a REST service

[13] Since 2019 Kubernetes address this problem by adopting the CSI standard, which presents an unified API for the volume storage

[14] https://docs.docker.com/engine/api/

[15] https://cython.org/

[16] https://flask.palletsprojects.com/en/2.0.x/

that allows (via HTTP) both creating "components" (each one a containerized application) from the same parameters that the previous version (image, port and parameters), and receiving Alertmanager alerts. In order to do that, an object model that considers container images, instances and other managed containers (Prometheus and Alertmanager) was adopted. A self-contained SQLite[17] database was also added. The SQLalchemy[18] ORM was used to map classes to tables. Fig. 4 shows the object model of the tool.

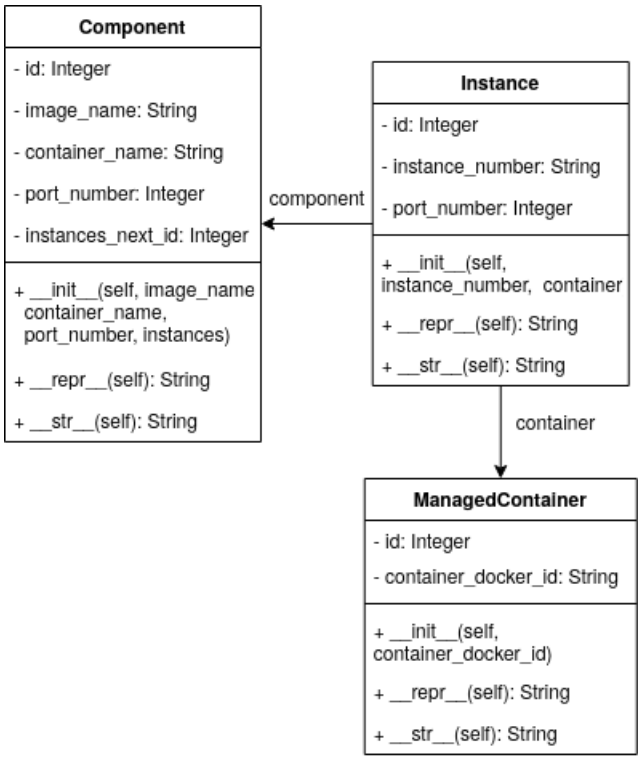

Fig. 4. Object model of the tool (Python version)

Even when using files generally brings compatibility problems, sharing the file between installations is not planned.

Fig. 4 shows the classes for the components, their instances and the class for the managed containers, that is used to clean all the containers (instances, load balancer, Prometheus, Alertmanager) when closing or restarting the application.

The second problem, related to volumes, was solved by deleting them and opening an HTTP interface that receives configuration messages. This interface consists of a Python script which also uses Flask. That way, the file modifying mechanism be-

[17] https://www.sqlite.org/index.html
[18] https://www.sqlalchemy.org/

comes encapsulated in the container, and the communication is carried out through a standard mechanism such as HTTP.

Course of action in case of an alert. In case of an alert, the course of action is fixed: add a container. It was considered that it would be desirable to be able to describe, in a policy form, which is the action to take in case of an alert. Multiple alerts can also be considered, some for scaling and other for down-scaling.

For instance, it could be defined that if the condition mentioned in the previous sections ("when the sum of the average per second (rate) of the request count received in the last five minutes exceeds the value 0.1") a container will be added, but if the threshold is reached in less than two minutes two containers could be added. It also could be defined that if the value is below a certain threshold (0.1, for instance) for a certain time (five minutes, for instance), a container will be removed (if there is more than one). An interesting point is which the selection criteria are when a container has to be deleted. In this case, for simplicity's sake, the oldest container is deleted.

In order to achieve this, a policy management tool called Open Policy Agent[19] was added. It consists of a server capable of making decisions based on an input, a policy and stored data related to a topic. In this case the topic is "actions to perform when an alert fires" and the data is pairs of "alert – action". The actions have the format *[direction] [count]*. For instance, if it is desired to scale up in three containers, the action would take the form *upscale 3*. If the desired behavior is to reduce in one container the total count when other alert fires, the action would be *downscale 1*. The policy simply extracts the action according to the input, which is an alert name.

Difficulty of use. Given that the tool became a web server, in order to access its functionality it is needed to make HTTP requests to its REST interface. There are three ways to achieve this: 1- Using curl from the console, 2- Using an HTTP client with graphic user interface as Postman[20] o 3- Writing a client and integrating it in another system.

The third alternative was chosen and it was decided to develop a graphic user interface in React[21], which allows access to the functionality provided by the tool, and it loads at the same time the alerts in Prometheus and their respective actions in Open Policy Agent. This decision is due to the first alternative going back to the console format, which is rudimentary and the format to avoid, and the second one requiring external tools and extended knowledge of the REST interface of the tool. However, the tool allows any of the three alternatives, and, in fact, a second graphical interface with different requirements could be developed using the same tool.

Fig. 6 shows a new version that includes a graphic user interface and it leaves out Docker volumes for the Prometheus and Alertmanager configuration.

---

[19] http://openpolicyagent.org/
[20] https://www.postman.com/
[21] https://reactjs.org/

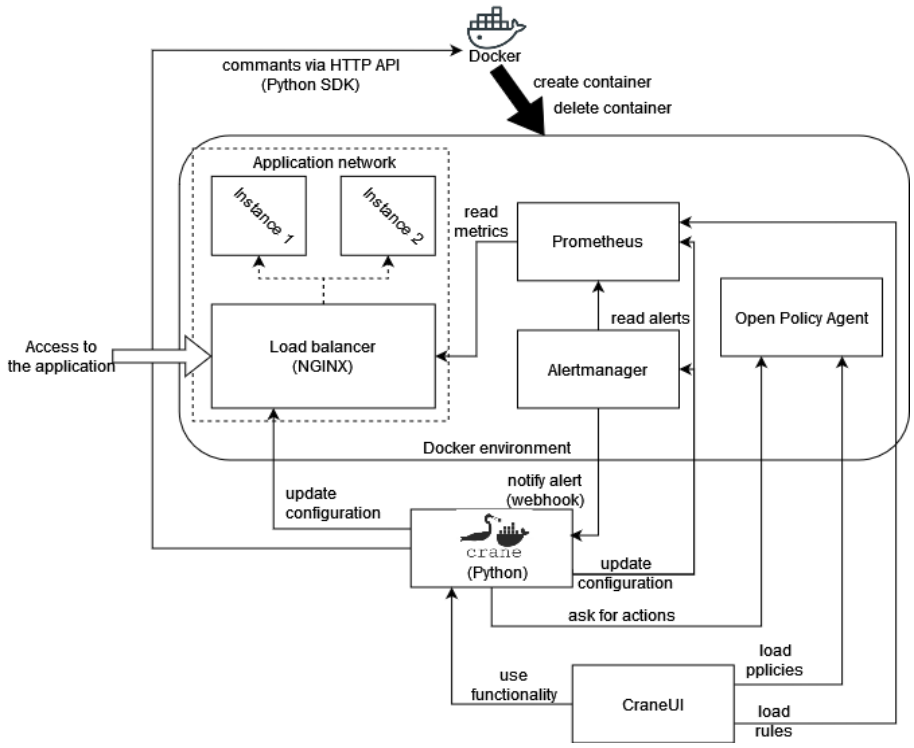

Fig. 5. – Third version of Crane with graphical user interface

In this version, the responsibility to load the rules and policies moves to the graphical interface, which uses the functionality of the Python service through its HTTP interface. It can be seen how the configuration update (arrow from Crane Python to Prometheus and Alert manager) is made directly to the container, which has a small service listening, and not through files, which used to tie the implementation to the operative system.

4 Conclusions and future work

A tool for creating containers automatically was presented. It adds load balancing and automatic scaling with configurable rules. Even though it is not in scope of this work, the construction of said rules were thoroughly studied for several application styles by other authors, as in [6].

An interesting point to extend is the possibility of extending the capacities of the tool in order to add automatic deployment of new versions of the Docker images.

In addition, it could add support for multiple ports for each container, or the support for non-HTTP ports (this is limited due the use of NGINX). In regard to contain-

er parametrization, currently they are listed as console arguments, but support for file input could be added.

At the moment, a local deployment is considered, but if a virtual machine deploy were to be made, other tools as Packer[22] could be used for vertical scaling, and Ansible[23] for automatic configuration. Finally, regarding the distribution of the tool, it could be turned into a packaged and uploaded to repositories as Chocolatey[24] (Windows) or the Debian and Ubuntu repositories.

References

1. Loukides, M. (2012). What is DevOps?. " O'Reilly Media, Inc.".
2. Fowler, M., & Foemmel, M. (2006). Continuous integration.
3. Leszko, R. (2017). Continuous Delivery with Docker and Jenkins. Packt Publishing Ltd.
4. Virmani, M. (2015, May). Understanding DevOps & bridging the gap from continuous integration to continuous delivery. In Fifth international conference on the innovative computing technology (intech 2015) (pp. 78-82). IEEE. In press.
5. Oracle (2021). "Chapter 3 Host Requirements". URL: https://docs.oracle.com/en/operating-systems/olcne/1.1/relnotes/hosts.html
6. Taherizadeh, S., & Stankovski, V. (2019). Dynamic multi-level auto-scaling rules for containerized applications. The Computer Journal, 62(2), 174-197. In press.
7. Baresi, L., & Quattrocchi, G. (2020, March). Cocos: A scalable architecture for containerized heterogeneous systems. In 2020 IEEE International Conference on Software Architecture (ICSA) (pp. 103-113). IEEE. In press.
8. Bullington-McGuire, R. and Dennis, A.K. and Schwartz, M. (2020). Docker for Developers: Develop and run your application with Docker containers using DevOps tools for continuous delivery. Packt Publishing.
9. Erl, T., Puttini, R., & Mahmood, Z. (2013). Cloud computing: concepts, technology, & architecture. Pearson Education.
10. Arcidiacono (2020). "DEHIA: una plataforma liviana para definir y ejecutar actividades con intervención humana basadas en workflows (*DEHIA: a lightweight platform to define and execute human intervention activities based on workflows*). "). Degree thesis. Facultad de Informática, UNLP
11. Docker (2022). "Networking with standalone containers" URL: https://docs.docker.com/network/network-tutorial-standalone/
12. Docker (2020). "Docker Engine release notes" URL: https://docs.docker.com/engine/release-notes/#20100
13. Nestybox (2019). "Secure Docker-in-Docker with System Containers". URL: https://blog.nestybox.com/2019/09/14/dind.html
14. Docker (2013). "Docker can now run within Docker". URL: https://www.docker.com/blog/docker-can-now-run-within-docker

[22] https://www.packer.io/
[23] https://www.ansible.com/
[24] https://chocolatey.org/