CRANE: a tool for deploying containerized applications in local environments. Lessons learned and proposed improvements.

José Miguel Silva Pavón[1][0009-0004-0240-9448], Franco Bellino[1][0009-0000-0286-4038], Patricia Bazán[3][0000-0001-6720-345X], Alejandra B. Lliteras[2,4][0000-0002-4148-1299], Nicolás del Rio[1][000-0002-0889-0752]

[1] UNLP, Facultad de Informática, [2] UNLP Facultad de Informática, LIFIA, [3] UNLP, Facultad de Informática, LINTI, [4] CICPBA
js.silva.010@gmail.com, fran85bellino@gmail.com, pbaz@info.unlp.edu.ar, alejandra.lliteras@lifia.info.unlp.edu.ar, ndelrio@info.unlp.edu.ar

Abstract. CRANE is a tool designed to simplify the local deployment of containerized applications, enabling developers and students to test distributed environments efficiently. Its lightweight design incorporates automatic scaling capabilities, making it a good solution for creating and deploying full application stacks within a controlled environment. Furthermore, CRANE aims to enhance DevOps skill development, streamlining the software development process within a continuous delivery framework. This paper introduces the implementation of CRANE as an API that manages the creation and deployment of Docker services while providing features for monitoring, defining scaling policies, and handling alerts. These functionalities support decision-making processes. Additionally, it discusses key lessons learned from the implementation, outlines observed results, and suggests potential improvements for future development.

Keywords: API REST, Local Deployment, DevOps, Docker, Scaling, Monitoring, Cloud Infrastructure, Training, Kubernetes.

1 Introduction

PaaS (Platform as a Service) services [1] have transformed the approach to software development in the digital era by providing preconfigured environments with tools that allow applications to be programmed and deployed directly in the cloud. This feature provides flexibility and convenience to developers, allowing them to focus on application logic without worrying about infrastructure management. It also facilitates the delivery of high-quality products, as their execution is not tied to a specific cloud, which aligns with current technologies.

However, in the early stages of development, it may be necessary to work in local environments. In these cases, cross-platform migration requires the creation of different configuration files, installation of specific dependencies, support for different library versions and adaptation to various operating systems, which also complicates effective performance testing.

To address these challenges, Docker containers [2] emerged, which make possible to run applications on any system with Docker installed, without depending on the

characteristics of the environment. However, managing, measuring and scaling these containers efficiently requires solutions such as Kubernetes [3][4], although their implementation involves a high consumption of computational resources. In addition, the configuration of containers demands static files that are often linked to local paths, which adds complexity to the process.

Given these needs, new roles have emerged within development teams, such as DevOps (Development and Operations) [5], which integrates previously separate functions, such as development, IT operations, quality engineering and security, allowing a comprehensive view of the system under development.

In this context, the present work proposes the implementation of CRANE [6], a low-cost solution aimed at the orchestration and local deployment of containers. This tool has been designed to simplify the management of containers in local environments, allowing their deployment in an agile and efficient way. CRANE automatically manages containers without the need for manual intervention by the developer, even in scenarios of low performance or host system failures.

The containers are interconnected through a network that enables real-time metrics collection and analysis. When a decrease in performance is detected, the system activates a set of predefined policies to respond appropriately to these events.

However, for different reasons, in early stages of development it may be necessary to work with this infrastructure in local environments. In these cases, the migration of the environment to different platforms requires different configuration files, as well as the installation of system-specific dependencies, library versions and the consideration of different operating systems. Effective performance testing is also difficult.

System management is performed by a REST API, which connects to the Docker client installed on the operating system via a Python library [7]. These tools together, and their organization in the CRANE architecture, allow performing the described operations in an efficient and automated way. One of the improvements proposed in this work, in relation to [18], is to replace the Python library with the native Docker API, thus avoiding intermediaries.

This paper is organized as follows: Section 2 describes the implementation of CRANE as a tool to simplify the deployment of applications in a local environment and also as a DevOps training platform. Section 3 describes the architecture, components and technology stack used in its implementation, as well as the description of an automated security component that is incorporated in [19] as an enhancement of [17]. In Section 4, lessons learned and proposed improvements are discussed. Finally, Section 5 presents conclusions and future work.

2 CRANE as a tool for simplification and DevOps training

DevOps is a methodology that combines the world of software development (Dev) and IT operations (Ops) to accelerate the software lifecycle and ensure continuous delivery of quality software. Its main objective focuses on automating processes and increasing collaboration between teams, promoting a culture of shared responsibility throughout the life of the application.

This methodology began to be used around 2007, when the software development and IT operations communities became concerned about the traditional software writing model where developers write the code and work independently from the operations team, who are in charge of implementing and supporting it. This is why the term "DevOps" is a combination of the words development and operations because it reflects the process of integrating these disciplines in a continuous and unified process.

A DevOps team consists of developers and IT operators working collaboratively throughout the product lifecycle, with the goal of increasing the speed and quality of the software implementation. It is a new way of working, a cultural shift, that has significant implications for the teams and the organizations they work for.

Under this approach, development and operations teams are no longer separate. In many cases, these teams are integrated into a single entity in which engineers collaborate at all stages of the application lifecycle, ranging from development and assessment to implementation and operations. This has resulted in professionals possessing a diverse set of multidisciplinary skills, promoting closer and more efficient collaboration.

Development and operations teams use tools to automate and accelerate the various processes to increase reliability. These tools help teams address important fundamentals such as continuous integration, continuous delivery, automation and collaboration.

Because of the "continuous nature" of this methodology, practitioners use the infinite loop to show how the phases of the DevOps lifecycle relate to each other. Although they appear to flow in a sequential fashion, the loop symbolizes the need for constant collaboration and iterative improvement throughout the lifecycle.

2.1 Difficulties and challenges of on-premises vs. cloud testing

Software development in local environments, while presenting significant advantages, also brings a number of limitations that can complicate the process of writing and deploying software. The initial setup of the programming environment requires developers to install and configure all the necessary tools, a process that consumes considerable time and effort. It is also very common that variability between the local and production development environments can cause software that works properly in the local environment to behave differently in the final production environment.

This problem is exacerbated by the diversity of operating systems used on technical equipment, such as Linux, MacOS or Windows [18], whose differences can greatly affect software execution, resulting in incompatibilities that impact program functionality on different platforms. Dependency management introduces another layer of complexity because variations in dependency versions between systems can make the software operational for some users, but not for others.

Adding to these challenges, effective team collaboration can be hindered in local environments because changes made by one developer may not be instantly available to the rest of the team. Even so, testing in local environments is essential, especially for economic reasons.

Despite the aforementioned complications, testing locally is much cheaper compared to testing in the cloud, where costs escalate rapidly based on resource usage and test duration. This last point is very important for startups and companies with limited budgets that need to carefully manage their operating expenses.

Finally, resource constraints in local environments, while challenging, also force developers to optimize software to work well with limited resources. However, while local deployment may be simpler, adapting the software for optimal performance in a production environment requires extensive modifications and work after deployment. These factors together make writing software in local environments a more complicated but important option for the software development lifecycle.

2.2 Design assumptions

CRANE was born from the design assumptions of [6], and evolved in [8] [9] in response to the need to quickly deploy locally a component-based application with multiple instances, test the behavior against critical errors and also migrate that application in different environments. That is, to simulate a cloud infrastructure in a local environment that does not demand a significant amount of hardware resources.

These design assumptions lead to the following question as a developer: *Are you ready to develop and understand applications running in a distributed environment on a cloud infrastructure?*

Developing software in a local environment versus the cloud involves carefully evaluating several critical factors such as cost, scalability, performance and security. On-premises development can give more fine-grained control over the infrastructure and, in some cases, lower upfront costs. On the other hand, opting for the cloud offers benefits such as greater scalability and flexibility, as well as providing access to advanced technologies that may not be available or feasible locally.

CRANE's approach is to integrate the advantages of both approaches, implementing technologies such as virtualization, automation and resource orchestration and create a local environment that mimics the capabilities of the cloud.

This solution gives developers the advantages of the scalability and flexibility of the cloud, while maintaining comprehensive control over the infrastructure and managing costs more cost-effectively.

CRANE is an educational tool that contains an ecosystem to simulate a cloud infrastructure locally, allowing the user to practice basic development and operations (DevOps) concepts such as: 1- Create and interact with containers, 2- Integrate real-time metrics and alerting systems, 3- Implement decision making based on predefined rules, 4- Create multiple instances of an application to test its performance and 5- Reduce costs when testing an application.

3 CRANE: architecture, components and implementation of the solution

CRANE is based on the implementation of two components: a Back-End for the creation and deployment of Docker containers [10] and a Front-End for the end user.

The Back-End component uses Python and FastAPI to generate an API that allows the management of Docker services. Among them the creation, deployment and monitoring of containers, as well as the definition of scaling policies and the management of alerts.

The Front-End component, through the CRANE API, allows the user to interact with the tool transparently and independently of the place where it is executed. The interaction with the user is done through an intuitive and easy-to-use web interface, designed to simplify the user experience and reduce the learning curve. This last component is beyond the scope of this paper and will be addressed in future work.

The CRANE Back-End infrastructure is composed of several interconnected components that work together to provide high availability and efficient response to critical events.

The user communicates with CRANE via a REST API and can create containers and adjust scaling rules depending on their own analysis and/or previously defined rules. You have control of exposed application ports, container traffic and can clone application configurations to generate new instances faster.

Being portable and easily accessible, it can be deployed on any system by simply executing a command. At the same time, the generated Docker Compose files are available to be taken to any other machine without having to take CRANE out of the local development environment.

3.1 – Technology Stack

This section briefly describes each of the technologies used for the proposed implementation and describes how the components of the solution relate to each other.

Docker [10] is an open platform designed to automate the deployment of containerized applications efficiently. It facilitates the separation of applications from the underlying infrastructure, thus enabling faster software delivery.

A reverse proxy [11] is a type of proxy server that acts as an intermediary between clients and one or more web servers. Unlike a regular proxy that is positioned between the client and the target server, a reverse proxy is positioned between the client and one or more web servers. Traefik [12] is used.

Prometheus [13] is an open-source system monitoring and alerting toolkit originally created at SoundCloud (SoundCloud is a Swedish-founded German headquartered audio streaming service) but now maintained independently of any company. It collects and stores its metrics as time-series data, i.e., metric information is stored with the timestamp at which it was recorded, along with optional key-value pairs called tags.

Alertmanager [14] is a Prometheus component that manages the generated alerts and allows applying certain actions. For this project, it was fundamental because it allows action to be taken in the event of any variation in the performance or availability of the active services. The way Alertmanager works depends on the rules defined in Prometheus, it is in charge of analyzing if any of the thresholds defined in the file called rules.yaml are met and to act with the configuration provided by the user. In the proposed implementation, a communication through automated messages (webhook) was used, in which Alertmanager notifies Crane by means of a call to the Alert receiving endpoint when it detects any anomaly.

The integration of Prometheus with CRANE brought benefits as it allows to abstract from several complications when defining an alert system, providing the following advantages: 1 - Receive alerts from Prometheus or other monitoring systems that are configured to send alerts through it, 2 - Can discard identical alerts and group similar

alerts into single notifications. This helps reduce the noise generated by multiple similar alerts, 3 - Can send alerts to specific recipients based on certain criteria such as alert labels, priorities, or user-defined settings. This allows sending alert notifications to the right operations teams or appropriate communication channels, 4 - Can send alert notifications to different communication channels such as emails, chat systems (Slack, Microsoft Teams), ticketing systems (PagerDuty, JIRA), and other notification services through custom integrations, and 5 - Offers the ability to temporarily mute or suppress certain alerts to avoid notification overload during planned maintenance periods or known incidents.

Open Policy Agent (OPA) [15] is an open-source project that provides a platform for policy authorization and evaluation. OPA is designed to help development teams design and enforce security, access and other types of policies in their applications and services.

OPA is based on the definition of policies in a declarative language called Rego. These policies can address many use cases, such as access control, data validation, network authorization, and are all implemented through an API exposed to the application, which can be queried to verify, among several functions, whether the user has permissions to access a resource.

Fig. 1 shows the architecture of the implemented solution and the way its components dialogue.

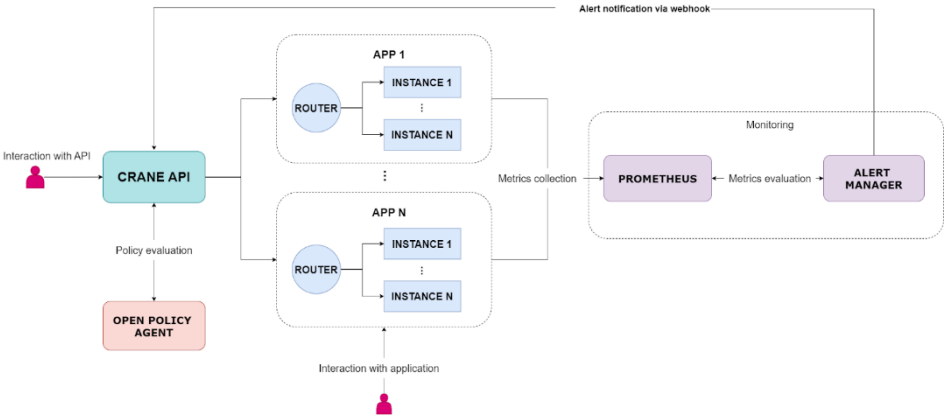

Fig. 1. CRANE components and their interaction.

CRANE API is a component written in Python that coordinates and manages several vital functions. It acts as the main link to the Python API, using the Python on Whales library to perform create, read, update and delete (CRUD) operations for containerized applications. In addition, it generates application instances, receives and manages alerts and manages user security.

The application instances (APP), referred to as Instance 1 to N, are interconnected within a virtual private network. This allows secure and isolated communication between the application instances. In turn, the application router communicates with Prometheus using a global network to provide access to the container's network metric.

Prometheus, acts as the monitoring system, responsible for reading and collecting metrics from the application instances. These metrics provide crucial information about the status and performance of the application, essential for automated decision making.

Alertmanager integrates with Prometheus to read alerts generated based on the collected metrics. Alertmanager processes and manages these alerts, and if certain conditions are met, triggers a webhook alert to CRANE.

Webhook alerts are HTTP requests triggered by Alertmanager when specific conditions are detected that require attention. This triggers a process in CRANE to query for possible actions and update the configuration as needed.

Open Policy Agent (OPA), acts as an authorization and policy control layer, which receives queries from CRANE and determines allowable actions according to defined policies. This mechanism ensures that any changes or automated responses are aligned with business rules and security requirements.

3.2- Starting CRANE, creating an application and managing alerts

This section presents scenarios of use of the proposed implementation.

The first scenario to be presented consists of starting Crane using the `uvicorn app:app -reload` command.

Within the CRANE API the following processes are executed: a) Authentication and registration: CRANE receives the request and extracts the user_id from the included token, integrating it to the application object, b) Storage: The application is registered in the database, c) Name management: A unique name is generated for the application, combining the name supplied by the user with an auto-incremental ID, to prevent name collisions in the host, d) Container configuration: A docker-compose. yml file is created with the default proxy configuration and the specifications provided by the user. This file is stored temporarily, e) Deployment of containers: Using the Python on Whales library, we proceed to build and deploy the environment with Docker Compose, f) Integration with Prometheus: Once the container is running, the port assigned to the router in the Prometheus network is obtained and the scrape is updated to allow reading metrics from the new application, g) Restarting the monitoring system: The monitoring stack is restarted to apply and refresh the configuration changes and, h) Temporary file management: Finally, if the REMOVE_TEMP_FILES constant is enabled, it is deleted, i.e., if the REMOVE_TEMP_FILES constant is enabled, it is deleted..

Regarding the creation of applications, the user has at his disposal all the configurations that Docker currently accepts, there is no limitation to create applications, as long as Python on Whales is kept updated with the latest Docker features. In addition, once any service is generated, use is made of the CRUD offered by the CRANE API to query, modify and delete the generated applications. In turn, users with administrator role have at their disposal the metrics, alerts and policies configurations to evolve the API and adapt it to their specific needs.

Fig. 2 contains a screenshot showing the instances generated with CRANE in the upper left part. In the lower left part is the terminal with CRANE running. It is also observed when a container is created and the endpoints that are receiving the request (CRANE status). In the right part, the body of the postman that receives the parameters for the creation of the instance is displayed.

Fig. 2 - Creation of instances in CRANE

Finally, in case of receiving alerts through the Alertmanager webhook, CRANE proceeds as follows: 1- determines the severity depending on whether the alert indicates FIRING or RESOLVED condition, 2- locates in the database the identifier of the affected service that generated the alert, 3- consults OPA sending alert type, severity and name of the affected service so that OPA evaluates the corresponding policies and returns the action to be executed (escalate, de-escalate, restart container) and 4 - executes the action determined by OPA.

Fig. 3 -Application scaling due to high demand

Fig. 3 contains in the upper right part a simulation of high load with Artillery [16]. In the upper left part, the alert activated in red. In the lower right, the CRANE console

receiving the HTTP request for new instance creation. And finally in the lower left, the new whoami [1] instance.

3.3 - Automated container security

Container security is a very important aspect because they share the same operating system kernel and resources with the host and other containers. When new containers are created, different considerations and best practices must be taken into account to secure them because otherwise vulnerabilities, data breaches or other critical security problems can be introduced in the equipment and other containers in the system.

The following recommendations help to mitigate many of these problems and were applied to perform the practical part of this work in order to meet the proposed security objectives [19]:

1. Use official or trusted images from verified repositories. Always verify that they are up to date and come from trusted sources.
2. Use vulnerability scanning tools to analyze images for potential security issues. Tools such as Clair[2], Trivy[3] or Anchore Engine[4] can help identify known vulnerabilities.
3. Use "Latest" versions preferably to keep images and containers updated with the latest security fixes, as long as they do not generate some kind of incompatibility in the service stack.
4. Apply the principle of least privilege, running containers with the minimum permissions necessary to perform their functions. Restricting access to resources and privileges that are not needed.
5. Use resource limits to prevent a container from consuming too many system resources and thus avoid inconsistency on the part of the Host.
6. Ensure that containers are well isolated from each other and from the host system by limiting interactions between them and using appropriate network policies.

As an improvement for [17], different alternatives were analyzed to add to the implementation a component to detect vulnerabilities in the containers and Trivy was chosen, a vulnerability scanning solution that was specifically designed for container environments and offers a detailed analysis of the images, looking for possible security risks. This scan not only detects known vulnerabilities in the libraries and dependencies used in the image, but also provides information on available fixes and security best practices.

Trivy can be coupled into development and deployment flows, allowing vulnerabilities to be proactively identified before they become security issues in production environments. Figure 4 shows how Trivy is incorporated into the CRANE architecture.

---

[1] Tiny Go webserver that prints OS information and HTTP request to output

[2] https://clairproject.org/

[3] https://trivy.dev/latest/

[4] https://anchore.com/blog/anchore-engine/

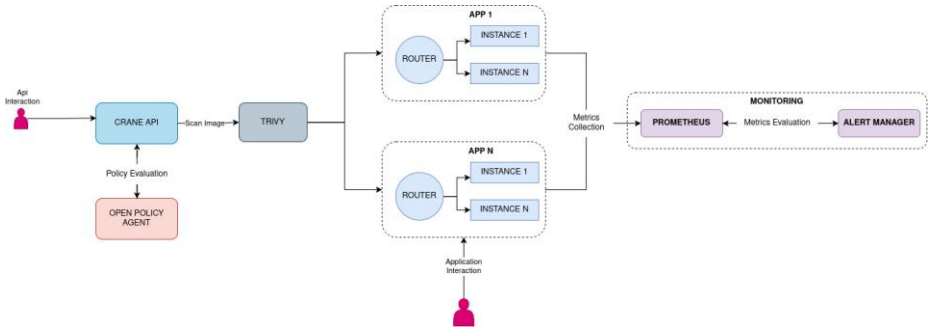

Fig. 4- CRANE components and their interaction, with automated container security

CRANE, basing its core on docker, needs the images used to be as secure as possible for different reasons:

- Early identification of potential vulnerabilities in container images and thus avoiding security risks before deploying to production environments.
- Compliance with security standards and regulations.
- Raising awareness among students and developers about good security practices.
- Protection of sensitive and critical data against attacks and security breaches.

4- Discussion of improvements from lessons learned

This section presents some proposals for improvements arising from the implementation experience and lessons learned. Firstly, the overload and complexity of the code is analyzed, secondly, the possibility of enhancing the efficiency and security of CRANE from a vulnerability scanning tool is presented and finally, the analysis of an alternative to Docker for container management is presented.

4.1. Code overhead and complexity.

The use of a wrapper to interact with Docker, as proposed in [17], provides flexibility and ease of use for the user, but at the same time, introduces unnecessary overhead and complexity on the code, which can cause that, in the face of new changes or improvements, or new people working on the project, the code becomes difficult to read, explain and maintain.

Thus, one of the improvements arising from the experience of implementing CRANE is the simplification of the abstraction layers to achieve a lighter and more efficient system. In this sense, direct interaction with the Docker API is proposed to lighten the system, eliminating intermediaries and reducing response time in container creation, deployment and monitoring operations. Direct interaction allows for better

resource management and provides greater control over advanced Docker configurations.

Working directly with the Docker API represents an improvement in terms of maintenance, because it allows updates and new functionalities to be incorporated faster, without depending on the compatibility of third-party libraries.

4.2. Enhancing the efficiency and security of CRANE

Another improvement opportunity that could enhance the efficiency and security of CRANE is the incorporation of Trivy as a vulnerability scanning tool within an automated pipeline, since it would allow performing security tests every time an application is deployed using CRANE, as described in Section 3.

Implementing Trivy in the pipeline would contribute to finding vulnerabilities in Docker images before they are put into production, allowing deployed applications to meet the required security standards.

It is proposed that this security test be run automatically at the time of deployment with CRANE, allowing any bugs or vulnerabilities to be identified and reported to the developer in real time.

4.3. Container management.

Another of the improvements analyzed is the possibility of integrating Containerd[5] as an alternative to Docker for container management proposed in [17].

Containerd, being a lighter and more optimized runtime for container execution, in relation to Docker[6], would reduce the overhead in the system and improve the efficiency in container management. Its adoption would allow a better integration with security tools such as Trivy, as it supports the execution of scans directly on its images, without the need to rely completely on Docker [20].

Containerd has native support in Kubernetes, allowing a possible extension of the system towards more complex architectures. This is very important as Kubernetes has deprecated direct support for Docker, opting instead for containerd, which is the underlying runtime used by Docker but in a lighter version optimized for container orchestration.

By using Containerd in CRANE, the dependency on Docker as runtime would be reduced, allowing a possible future integration with Kubernetes and other orchestration platforms, optimizing the performance of the system by eliminating unnecessary processes that Docker has.

---

5 https://containerd.io/

6 https://www.docker.com/blog/containerd-vs-docker/

5- Conclusions and future works

The main objective of this work was to effectively implement the CRANE design proposal [6]. First, the design was technically validated and the technological alternatives for each component were analyzed.

The first prototype was developed in Python following Crane's structure and with it the communication of each part was validated. At the same time, Python FastAPI plugins were included, as well as the interaction of libraries to communicate with Docker.

Then, the integration with Traefik for traffic management, Prometheus for metrics monitoring, Alertmanager for alert management, and Open Policy Agent for the application of scaling and security policies was performed.

Finally, data models were integrated to apply an authentication and authorization layer over the designed API.

Beyond its technical utility, this solution is intended to have significant value in the educational environment. Students could use it to learn about service development and deployment practices, but also for the simulation of DevOps practices in local environments, a process that, as explained throughout this paper, is very complex and costly fundamentally if not analyzed in early time.

The combination of the proposed enhancements would strengthen both the security and performance of the system resulting in a more reliable, scalable, readable and efficient workflow and would result in a more optimal use in educational and testing environments, where users are in the process of learning about DevOps best practices and container security.

For future work, we propose the creation of a Front-End using React. This Front-End will consume all implemented REST endpoints, from the creation and deployment of Docker services to metering, scaling policy definition and alert management. This will provide users with a complete interface to communicate with all of Crane's core functions.

The purpose of this Front-End is to make CRANE accessible and efficient for a wide range of users, from beginners to experts. It not only aims to make the technical process simpler, but also to foster understanding and learning about containerized service management in a local environment.

The alerting system can also be enhanced by using container metrics, such as memory and CPU usage, for its scaling policies and other parameters, as currently, metrics are taken directly from router traffic.

References

1. Rani, D., & Ranjan, R. K. (2014). A comparative study of SaaS, PaaS and IaaS in cloud computing. International Journal of Advanced Research in Computer Science and Software Engineering, 4(6).
2. 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.
3. Kubernetes - https://kubernetes.io/es/docs/concepts/overview/what-is-kubernetes/

4. Burns, B., Beda, J., Hightower, K., & Evenson, L. (2022). Kubernetes: up and running. " O'Reilly Media, Inc.".
5. Ebert, C., Gallardo, G., Hernantes, J., & Serrano, N. (2016). DevOps. Ieee Software, 33(3), 94-100.
6. Arcidiacono, J., Bazán, P., del Río, N., & Lliteras, A. B. (2022). Crane: A Local Deployment Tool for Containerized Applications. In Conference on Cloud Computing, Big Data & Emerging Topics (pp. 58-71). Springer, Cham.
7. Python - https://www.python.org/doc/
8. Silva Pavon, J. M., Bellino, F., Bazán, P. A., Lliteras, A. B., Arcidiacono, J., & Rio, N. D. (2023). Despliegue de aplicaciones contenerizadas: un caso de implementación basado en Crane. In XXV Workshop de Investigadores en Ciencias de la Computación (Junín, 13 y 14 de abril de 2023)
9. Silva Pavón, J. M., Bellino, F., Bazán, P. A., Lliteras, A. B., & Rio, N. D. (2023). CRANE: simplificando el despliegue de aplicaciones contenerizadas en entornos locales. In XXIX Congreso Argentino de Ciencias de la Computación (CACIC) (Luján, 9 al 12 de octubre de 2023).
10. Docker - https://docs.docker.com/
11. Sommerlad, P. (2003, June). Reverse Proxy Patterns. In EuroPLoP (pp. 431-458).
12. Traefik - https://doc.traefik.io/traefik/
13. Prometheus Documentation - https://prometheus.io/docs/introduction/overview/
14. Alertmanager Documentation - https://prometheus.io/docs/alerting/latest/alertmanager/
15. Open Policy Agente Documentation - https://www.openpolicyagent.org/
16. Artillery.io https://www.artillery.io/
17. Silva Pavón J, Bellino F., Bazán P., Lliteras A., del Rio N. (2024). Implementing CRANE: a tool for simple deployment of containerized applications in local environments. Argentina. La Plata 2024. CACIC 2024. http://sedici.unlp.edu.ar/handle/10915/172755. Pag 921-930
18. Adekotujo, A., Odumabo, A., Adedokun, A., & Aiyeniko, O. (2020). A comparative study of operating systems: Case of windows, unix, linux, mac, android and ios. International Journal of Computer Applications, 176(39), 16-23.
19. Silva Pavón, J. M., & Bellido, F. (2024). CRANE: Simplificando el despliegue de aplicaciones contenerizadas en entornos locales (Tesis de Grado, Universidad Nacional de La Plata). http://sedici.unlp.edu.ar/handle/10915/168369
20. Kaiser, S., Haq, M. S., Tosun, A. Ş., & Korkmaz, T. (2022). Container technologies for arm architecture: A comprehensive survey of the state-of-the-art. IEEE Access, 10, 84853-84881.

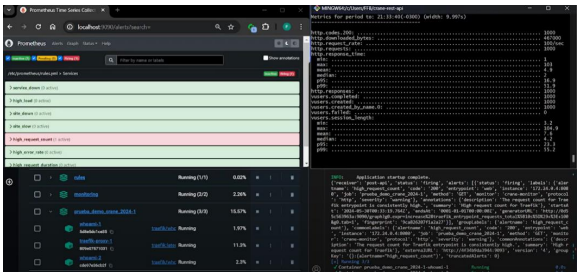

Fig. 2

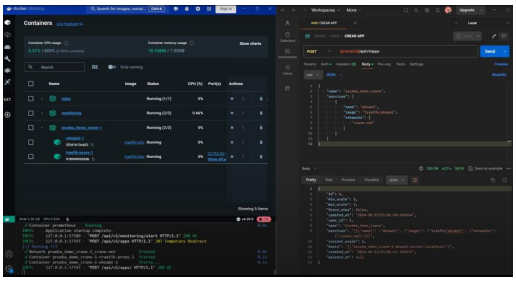

Fig. 3