Continuous Integration environment deployment

Oana-Anastasia MINCIU, Iulia-Lidia IACOB, Anca-Daniela IONIȚĂ, Ștefan MOCANU

University Politehnica of Bucharest

oana.minciu@upb.ro, iulia.iacob@upb.ro, anca.ionita@upb.ro, stefan.mocanu@upb.ro

Abstract: DevOps is a solution that software development industry relies upon to efficiently link development and operations processes. Even though it has been adopted relatively recently as a separate field, its growth has been accelerated by its undisputed usefulness, which has contributed to a solid reputation as a key part of the software development and delivery processes. The main purpose of DevOps is the automation of multiple and diverse activities related to development and operations inside a project, from modeling and implementing a Continuous Integration process to designing and provisioning a custom environment. Specifically, designing and implementing strategies for code integration, the provision of computing infrastructure, and system deployment were the ones that pushed the concept forward. Building and configuring a Continuous Integration environment can be a time-consuming and error-prone process as manual activities performed in development and operations, while also involving risks related to stability and availability. To combat these risks, provisioning and configuration operations can easily run on demand just as code integration operations can. These speed and reliability are what makes DevOps engineers valuable for any modern software project as automation becomes more and more ubiquitous in the software world.

Keywords: DevOps, Continuous Integration, Containers, Automation, Cluster.

Implementarea unui mediu de Integrare Continuă

Rezumat: DevOps este o soluție pe care industria de dezvoltare de software se bazează pentru a lega în mod eficient procesele de dezvoltare și cele operaționale. Chiar dacă a fost adoptat relativ recent ca domeniu separat, creșterea sa a fost accelerată de utilitatea incontestabilă, care a contribuit la o reputație solidă ca parte cheie a proceselor de dezvoltare și livrare de software. Scopul principal al DevOps este automatizarea unor activități multiple și variate, legate de dezvoltarea și operațiile din cadrul unui proiect, de la modelarea și implementarea unui proces de integrare continuă, până la proiectarea și furnizarea unui mediu personalizat. Mai exact, proiectarea și implementarea strategiilor pentru integrarea codului, furnizarea infrastructurii de calcul și implementarea sistemului au fost ceea ce a împins conceptul înainte. Construirea și configurarea unui mediu de integrare continuă poate fi un proces consumator de timp și predispus la erori, precum activitățile manuale executate în dezvoltare și operații, implicând totodată riscuri legate de stabilitate și disponibilitate. Pentru combaterea acestor riscuri, aprovizionarea și operațiile de configurare pot rula cu ușurință la cerere, la fel cum pot fi operațiile de integrare a codului. Viteza și fiabilitatea sunt ceea ce îi face pe inginerii DevOps valoroși pentru orice proiect software modern, pe măsură ce automatizarea devine din ce în ce mai omniprezentă în lumea software-ului.

Cuvinte cheie: DevOps, Integrare Continuă, Containere, Automatizare, Cluster.

1. Introduction

Driven by the current industry context of fast-evolving requirements, strong incentives have emerged to streamline and simplify the processes of development and delivery of software products. These led to a high interest in the implementation of automated processes that replace repetitive, monotonous, time-consuming activities. This paper belongs to the automation section of the field of information technology with a strong focus on software automation. This is the process of creating activities that perform repeatable operations that do not require human manual intervention or require minimal effort on the part of a user. Automation can be used to provision and manage software environments or resources from a particular environment. Its purpose is, of course, to save time, to minimize costs, but also to give users the opportunity to focus on essential activities that are not repeatable. Also, since the nature of the defined activities is independent of the human usage in a large percentage, the automation has major advantages in the operations of environment scaling, for obtaining results with small error impact and ensuring repeatability of resource provisioning.

https://doi.org/10.33436/v32i2y202206

Strongly related to the previous classification, the subject of the paper also falls under the practices that combine the development process with the operations part, known as DevOps, derived from joining them (Macarthy & Bass, 2020). These practices and tools embrace a model that combines several activities that would normally have been separated or considered as separate entities. These practices cover the complete life cycle management of software products, by linking the processes of development, testing, quality verification, installation and running of products, and even the generation of documentation based on templates, through automation techniques. Thus, DevOps is not a term with a specific, strong definition, but it can be considered an umbrella for all elements, activities, and tools that facilitate the rapid delivery of changes in software products and early detection of possible implementation errors in order to create a quality product built in the shortest possible time. DevOps activities are part of a diverse fast-evolving topic; thus, they may differ depending on the needs of the company. Considering this, it is difficult to define the required knowledge to prepare someone working with this topic, and practitioners are always required to learn through hands-on experience (Pang et al., 2020).

Among various aspects of DevOps, this paper is focused on the part of Continuous Integration, a practice in software development through which team members integrate their work frequently, even daily. Each integration process is checked by an automated build process with integrated tests meant to detect errors as quickly as possible. In the continuous integration phase, changes from a developer are merged and validated. The goal is to quickly validate these changes to the published code. The result is to identify any issues within the code and automatically notify the developer about this problem. The process detects when code changes are made and runs any associated compilation processes to prove that the code changes are buildable. It can also run specific tests to demonstrate that code changes work in isolation (function inputs produce the desired outputs, erroneous ones are appropriately signaled, and so on) (Laster, 2020). DevOps - Continuous Integration has grown in recent years and has become an essential part of the software development process. This can be done without implementing these practices, but depending on the size of the projects, the impact of delays caused by wasted time by manually performing processes can be considerable. There may be no major delivery delays for very small projects, but even for them there are benefits in implementing practices such as keeping a history of project progress and ensuring that if some developers abandon the project or are replaced, the processes defined by them are not lost.

The rest of the paper is organized as follow: Section 2 presents a brief description of what Continuous Integration is and its advantages. Section 3 addresses the concept of software containers, providing their history, structure and uses. Section 4 continues with the subject of container orchestration through the Kubernetes technology, while also portraying how the continuous integration tool Jenkins can improve its performance using this technology. Section 5 describes how a continuous integration environment, as presented in the previous chapter, can be deployed. Section 6 concludes the paper with a summary of the presented concepts and the use case analysis.

2. Continuous Integration

As presented in the previous section, DevOps is an umbrella of practices and technologies related to process automation that combines development activities with operations. Specifically, the development and testing teams are the entities responsible for writing the source code, defining the test scenarios, and implementing them; the operations team complement them by packing and delivering the source code and by configuring and monitoring the running environment. According to DevOps methodology, these elements are modeled as a single entity capable to visualize and understand all steps and being more or less involved in most of them. It can be difficult for a person to follow this full role, so it is important to focus on certain elements, but with a clear view of the whole process, even if it is partially incomplete for some sections.

Continuous Integration is a key category in the DevOps chain of tools and practices (Laster, 2020). The word "*continuous*" does not mean a continuous run or execution but refers to operations that are always ready to run in the context of software creation. The software integration process is not a new problem, and for small projects with one or two developers it is not a problematic activity, but when considering a project with high complexity there is a growing need to integrate members' work to ensure the operation of the components together. Thus, waiting until the end of the project to carry out this operation will result in major quality problems, which in turn introduce delays in delivery.

Continuous Integration is usually used together with an agile software development process (Madhumathi, 2018). An organization will build a list of tasks that include a product roadmap. These tasks are then distributed to the members of the delivery software team. The use of Continuous Integration allows these software development activities to be developed independently and in parallel by the assigned developers. Once one of these tasks is completed, a developer will introduce the new version of the code into the integration system, which will be merged with the rest of the project. The most common representation of such a process is in the form of a pipeline. It can be very simply understood as the sequence of activities through which a single software unit can be delivered ready to be installed. The project team will choose what services they will use to build this, what technologies and what stages they will define, but it cannot be said that there is a single generic implementation of a pipeline, but only an abstract concept. A visual representation of a sequence of activities building a Continuous Integration pipeline is depicted in Figure 1.

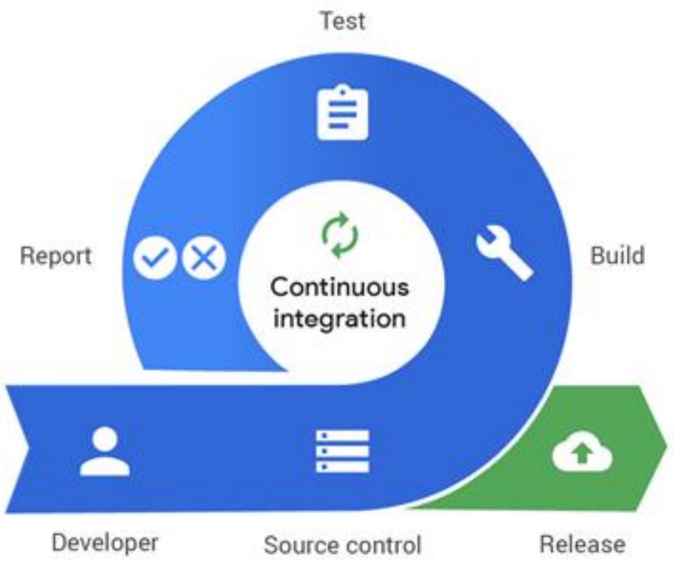


Figure 1. Continuous Integration (Pagerduty, 2020)

It is always necessary to juggle resources (developers, budget, time) within imposed limits (time required for delivery, deadline). As such, more time and budget need to be put behind the development of a solution that will save other long-term resources. Reducing costs and complexity is a crucial benefit of using a Continuous Integration pipeline. Many of their stages are very repetitive, but they are not easy to do, and it is noteworthy that they are prone to errors. Automating the process and delegating it to a defined process not only can free up valuable resources for developers, and product development activities but also can reduce the chances of errors while improving the team's responsiveness. In order to explore how to build an environment for Continuous Integration, the concept of software containers will be further addressed and how they can be managed.

2.1. Modeling a Continuous Integration process

There are many tools used for Continuous Integration, but probably the most famous one is Jenkins, a free and open-source platform built on Java. A few of its advantages are the flexibility in modeling processes, from using simple shell scripts to complex activities running on multiple physical or virtual nodes, but also the fact that it provides support for multiple tools, in code versioning for example git, svn, or clearcase. Jenkins works on a manager-worker architecture in order to delegate certain responsibilities and activities from the main node, but also to provide different environments according to the needs of a project. The master node is usually responsible for providing a graphical interface to the users, while the worker nodes will be the ones performing continuous integration tasks. This distributed computing model will allow the Jenkins master node to remain responsive to users through the graphical interface provided. At the same time, it also allows Jenkins to redirect the execution of activities to the connected nodes (McAllister, 2015) while ensuring that not a single node is overrun with activities and that the master node does not grow into a giant consumer of memory and processor time.

While the used tool can provide certain advantages, modeling of a continuous integration process should not depend on it. Regardless of the chosen environment, continuous integration uses a concept called “software pipeline” to represent a sequence of processes organized in stages. A pipeline object can be modified by users as needed, to achieve the desired results, and Jenkins offers the possibility of using this object to orchestrate long-term activities that can be performed even on multiple integration server agents. The definition of a Jenkins pipeline is written in a text file (called Jenkinsfile), either directly on the provided interface editor or stored in the source code version of a project (Jenkins, 2020). This is the foundation called "Pipeline-as-code" and treating pipeline processes as part of the application to be stored and revised like any other code allows control over the evolution of the process.

Figure 2 shows how Jenkins portrays an execution of a pipeline object and it highlights each stage separately along with its result. This is an excellent example of how a continuous integration pipeline is viewed during the modeling or how the development team sees it. It provides enough information as to what each stage should achieve but does not bother the user with the details of implementation. When modeling such a process the key steps are identified as individual stages along with acceptance criteria, however, each stage can be implemented and executed separately considering some pre-requirements: the test stage will need, of course, some artifacts generated by the build stage, while the deploy stage may require only one single artifact – the executable one. According to a dependence matrix between them, some will be mandatory, while others can be skipped, and this is the kind of flexibility available when choosing to model a continuous integration process. A single model can be created for an entire product, but any special component can alter the implementation while still respecting the provided template.

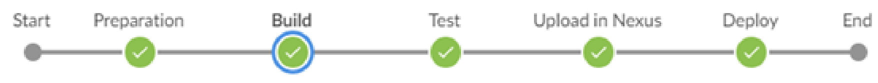


Figure 2. Jenkins Pipeline

3. Software containers

Container technology is one of the latest revolutionary elements in software development and operations (Shah & Dubaria, 2019; Sima et al., 2017). Many organizations view containers as means of improving the lifecycle management of programs developed through capabilities such as continuous integration and continuous delivery. The concept is not necessarily new, in the Linux environment, it has a long history and several implementations. Among the first is the well-known *chroot()*, which allows a process to run in an isolated environment, practically in its own file system. Another well-known type of container is LXC, short for *LinuX* Containers, which allows an entire operating system to run in one container, like a genuine virtual machine: software packages can be installed, upgrade operations can be performed, configurations can be changed. It allows the creation and isolation of multiple virtual environments on a single host system (Hardikar et al., 2021).

Containers ensure environmental independence by creating a complete running environment for the target program, including all its dependencies, and everything related to its configuration. It is these features that motivate the use of containers as easily and efficiently to automate the processes of compiling source code and generating artifacts, such as executable files. Thus, the adoption of containers in the process of Continuous Integration has grown with the advent of containers. A software container can be considered a rather abstract concept, and therefore, to make it easier to view, one can try a rather familiar analogy for users. The analogue is a transport container in the transport industry. The container completely revolutionized the transportation industry. The container is just a metal box with standardized dimensions, the length, width and height of each container following certain rules (Schenker et al., 2019).

Basically, a software container is nothing more than a running process, with some added encapsulation functions applied to it to keep it isolated from the host and other containers. One of the most important aspects of container isolation is that each container interacts with its own private file system; this file system is provided by an image. An image includes everything you need to run an application - code or executable, running media, dependencies, and any other required file system objects. The layered structure of a container image is portrayed in Figure 3: each image has a base layer to start from, on top of which others can be added to provide the desired tools and functionalities. It is a read-only object, however, each container started from an image will have its own writable layer so that multiple containers can use the same image simultaneously.

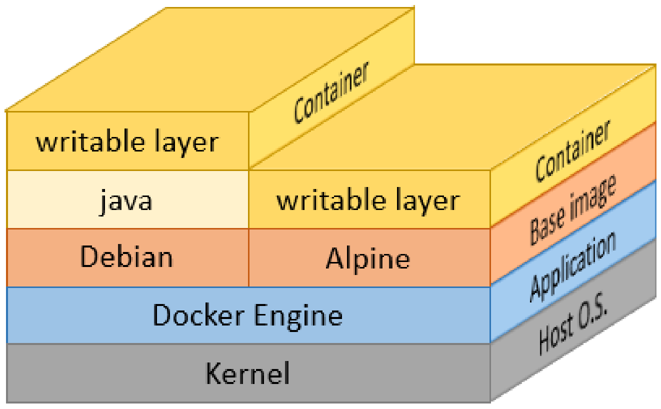


Figure 3. Docker image

A container can become the ideal environment for running continuous integration processes because it is ephemeral. One approach inspired by its very definition is to generate a new container for each activity or step specially tailored for that step so that the entire process runs smoothly, from code building, testing, and report generation to uploading artifacts to storage servers. Afterwards, the container will be destroyed along with any data it might have had in order to ensure a clean environment on the host machine. In this way, no trace of the integration process remains, and all resources are freed immediately after its execution ends. Considering the requirements of each project, the whole continuous integration process can be modeled to run inside containers, including even running the integration process in a *Docker* container. This kind of approach means dedicating even the development and deployment processes to be executed in containers and it is suited for applications and products designed to run in cloud environments, cloud computing being itself a solution for reducing costly hardware (Rădulescu & Rădulescu, 2017).

Continuous integration containers have dedicated *Docker* images built specifically for this purpose as they require certain tools used for code building, testing, artifacts generation, and even deployment in a development or production environment. Many factors need to be taken into consideration when designing images, because aiming at having everything in a single container, for example, all the necessary tools might lead to a too large image. Sometimes it cannot be avoided, but nonetheless, a continuous integration image should not exceed 2 gigabytes in size. However, the recommended approach is to make a different container for each step in the process, so that each individual image is very small as it is tailored for one specific step. When working with larger images the layered structure provides some advantages: individual tools can be placed in different layers so that when one tool needs to be changed the other layers are not affected, while also taking advantage of how container layers are stored separately – when downloading a *Docker* image from a storage server each layer is acquired individually, and if it already exists on the current machine it is not downloaded again.

Containers can be used just as they are or through a container orchestrator such as Kubernetes. This is a vast and interesting subject as besides using temporary agents it also allows the decoupling of resources from the main Jenkins node, like configuration files for tools (maven, npm, bower, git, proxy). Using a Kubernetes environment has many advantages, as it is a very popular platform for application deployment for both production and development environments, can be reused, and can share resources between multiple continuous integration processes. Contrary to the name, continuous integration activities are not executed continuously but rather they are always prepared to run whenever source code changes are published. However, the physical resources are only temporarily reserved and will not impact the development or testing processes executed in parallel to a great extent.

4. Kubernetes

Container orchestration consists of managing the life cycle of containers, especially in dynamic and large-scale environments. It fulfills the functions of automating the process of container launching, management, scaling, and network management. It also plays an important role in DevOps teams that can integrate container orchestration into the processes of Continuous Integration, Continuous Deployment, and Continuous Delivery. A simplified description of a container orchestrator can be seen in Figure 4. This technology can be viewed as a decision-making entity and a set of nodes that only provide resources. The entity that makes all the decisions that manage the life cycle of the containers can be on a separate node but can also exist directly on the resource nodes. It will be responsible for monitoring the status of all containers and nodes.

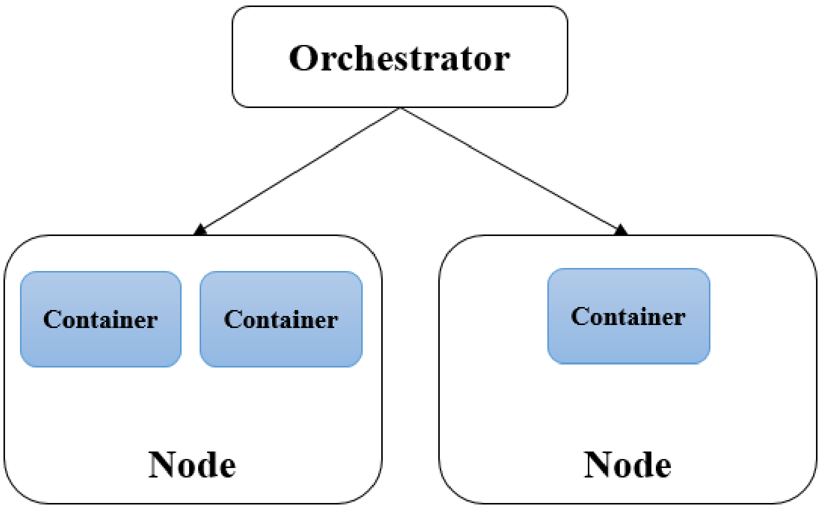


Figure 4. Container orchestrator

A well-known container orchestration tool is Kubernetes, an open-source technology that was originally developed by Google (Bernstein, 2014). Kubernetes works on the cluster model: a group of one or more physical or virtual machines capable of running containers Linux builds a *Kubernetes* cluster that can be used to orchestrate containers. These machines can be on a local network, or they can be in any type of cloud, even in different environments if they have access to each other from a network point of view. It does not matter what type of machine they are, physical or virtual, it cannot be said that one of them would be preferred because the choice depends on the time of connection to them. Because nodes provide the actual platform on which containers run, they must have software capable of running containers, such as *Docker*.

Kubernetes' central management entity is its HTTP API component with JSON serialization scheme. Like *Jenkins*, *Kubernetes* follows a master-slave architecture, with some nodes having control roles, while others are purely workers. The API points to the master nodes responsible for coordinating everything inside the cluster and represents the access path to the entire cluster. While *Kubernetes* is a container orchestrator, the smallest unit which can be deployed inside it is not a container, but rather a group of linked containers called a pod. This detail is important in the context of Continuous Integration because it provides the perfect environment to run multiple containers inside the same base environment, having access to each other's static resources.

4.1. Jenkins with Kubernetes

Jenkins can be integrated to work with Kubernetes fairly easily through a plugin developed for this purpose. Contrary to normal Jenkins nodes or agents, which are static resources represented by either physical servers or virtual machines, this plugin integrates with Kubernetes by creating temporary agents in the shape of pods. Considering both Jenkins and Kubernetes work with the master-slave concept this integration can be viewed as shifting the responsibility of agent scheduling from the Jenkins master to the Kubernetes master, as portrayed in Figure 5.

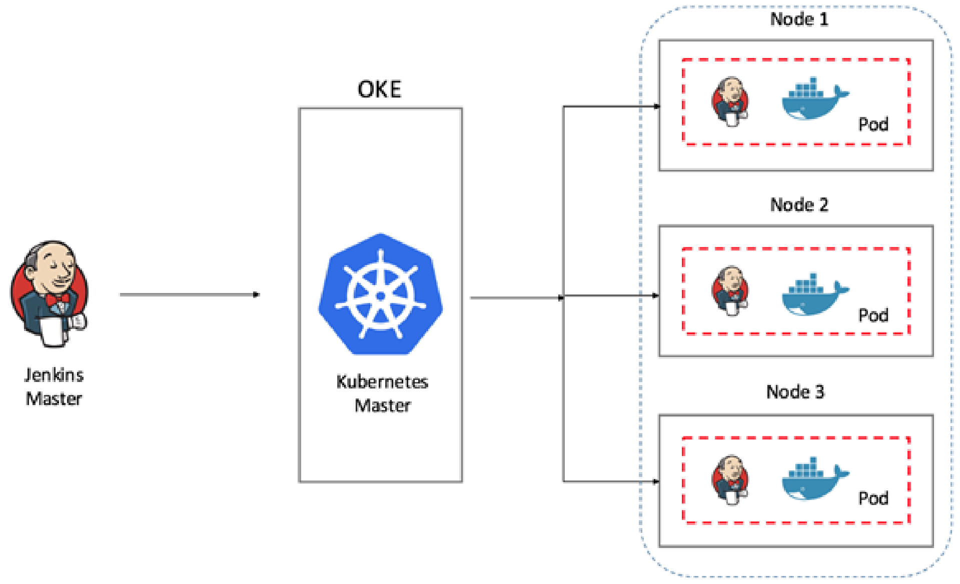


Figure 5. Temporary Kubernetes agents (Oracle, 2018)

The *Jenkins* master is connecting to these temporary agents in the same way as usual agents, via the JNLP (Java Network Launch Protocol). However, before being able to request temporary agents, *Jenkins* will require connection credentials and configuration to the API of *Kubernetes*. An option to add a *Kubernetes* cloud instance will appear in the *Jenkins* nodes section, where connection information will be specified: the *Kubernetes* API link, the service account information for *Jenkins* (the corresponding certificate used to access the API), the namespace it has access to, and the link to the *Jenkins* server so that the pod objects can connect back to the master node. *Jenkins* will create all pods starting from already defined templates, either in the cloud definition or in the Continuous Integration pipeline code. Due to the JNLP protocol requirements, each pod will need to have a container capable of using this protocol to connect to the *Jenkins* master, this pod being named jnlp, otherwise, the pod will not be able to report back to the master and the agent will be virtually useless.

For the usual *Jenkins* users, nothing actually changes when they want to select a node, and the visual representation of a pipeline is identical. Normal nodes are selected via labels when running a pipeline and *Kubernetes* agents are selected in the same manner, as the pod templates will have an associated label. What needs to be changed is the implementation approach: each stage will need to explicitly select a certain container by name after being assigned a *Kubernetes* pod. Using *Kubernetes* to provide temporary *Jenkins* agents is a graceful solution to increasing the optimization of resource consumption in the context of Continuous Integration processes, as they do not require a vast amount of resources continuously, but rather sporadically.

5. Deploying the environment

Continuous Integration is defined by the automation of manual processes, which is the base of the solution described in the current section. If continuous integration activities can be automated, then the deployment of such an environment can also be performed using automated operations. Manual deployment of a continuous integration environment is performed by installing and configuring each server from scratch, followed by ensuring their connection also manually. However, this method is both time-*consuming* and prone to human errors. Therefore, an automated solution for the environment deployment is preferred to counter the mentioned disadvantages, but also to address another significant aspect, namely the repeatability of the deployment operations. Unforeseen events such as the loss of a machine or corruption of disks can affect a continuous integration environment, so an automated solution is essential in order to quickly provision a new environment identical to the initial one.

The environment components have already been presented above: all machines will have Linux as the operating system, a *Jenkins* master server will need to be installed, a *Kubernetes* cluster as well, and a management server for *Kubernetes*. The *Jenkins* installation is the simplest part of such a deployment, while for *Kubernetes* there are multiple solutions, from installing all components manually to free automated solutions such as *kubespray* or *Rancher*. Due to its simplicity from the user's point of view, *Rancher* will be considered in this approach. The automation technology considered here is Ansible, an extremely simple IT automation engine that automates cloud provisioning, configuration management, application deployment, inter-service orchestration, and many other IT needs (Redhat, 2020).

Ansible only requires an environment with python to run and works by connecting to the desired machines and transferring small programs called Ansible modules. These programs are written to be resource models of the desired state of the system. Then Ansible executes these modules (by default via SSH directly on the target machines) and removes them upon completion. Basically, Ansible merely describes the desired state of a system (for example specifies that the vim editor must be in the installed state) and, if it is not in the expected state, it runs the module to reach the required state (it will call the Linux package manager module to install the vim editor if it does not already exist, while if it is installed it will just skip that step). The solution uses Ansible to prepare machines by installing *Docker* as a container running medium, configuring *Docker* with a private image source if necessary, installing desired dependencies, starting the installation of the *Kubernetes* cluster on machines, generating elements for *Jenkins* to use *Kubernetes* as a medium: a separated environment inside the cluster, a login account. It is also desired to explore the start-up of the *Jenkins* master node through Ansible and its configuration.

To prepare the environment with Ansible, it is assumed that the user already has a set of physical or virtual machines with an already installed Ubuntu operating system, version 18.04 or newer. The solution does not install or generate machines, but only aims to configure and use them in order to create an environment with *Jenkins*, *Kubernetes*, *Rancher* Server. To perform all the steps, Ansible activities were logically separated for each component. The requested environment will require the following elements: a machine used for installation (it will need to have access to all the other machines listed afterwards, as its purpose is to be able to connect via SSH to all and run the necessary operations), *Jenkins* machine (the *Jenkins* master server will be installed on it as a *Docker* container), *Rancher* machine (the place where the *Rancher* server application will also be installed in a *Docker* container to monitor the *Kubernetes* cluster), *Kubernetes* machine set (a set of one or more machines that will host the *Kubernetes* cluster).

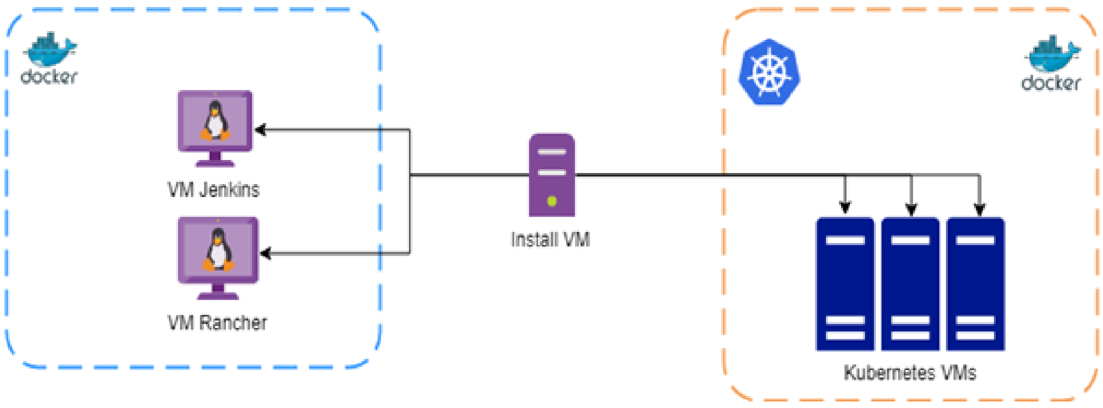
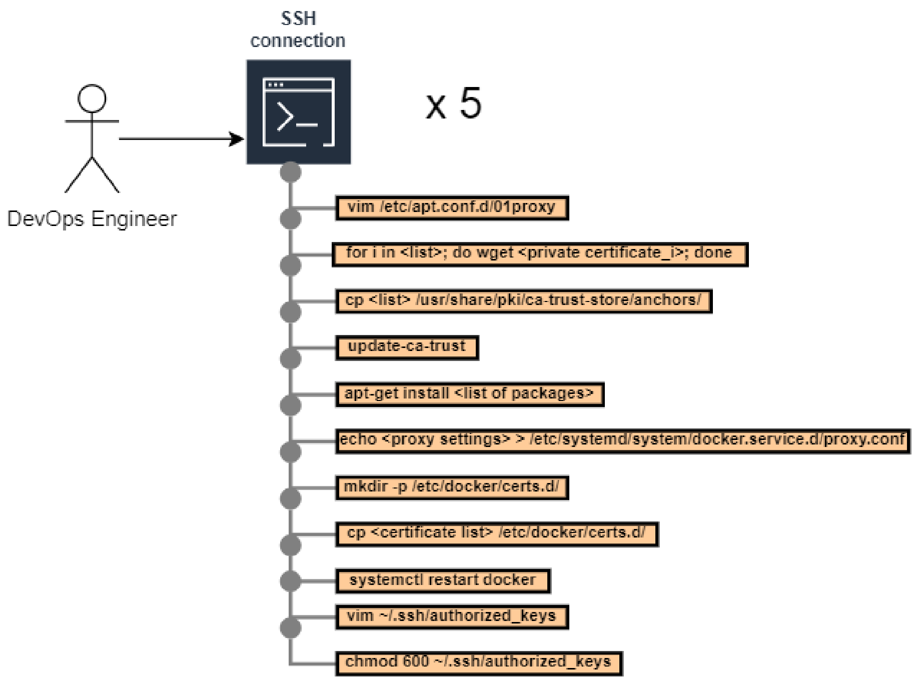


Figure 6. Required resource environment

Figure 6 portrays the machines' requirements considering the following aspects: *Jenkins* and *Rancher* can be installed on the same node, but were represented separately, while for *Kubernetes* the example contains three nodes, however the number will depend on the user's requirements. A set of three machines would be the recommendation in order to have more than one master node, but the installation will work with a single node if necessary. As already specified, each of these components is to be installed and configured in its own logical segments of the installation, while the common elements are all provided in the first section, including installing and configuring the *Docker* engine service, providing access from certain entities via SSH keys to all machines and configuring certain special requirements such as proxy or private certificate installation in the operating system and the *Docker* engine.

The actual configuration is designed to run sequentially, first with the configuration of all machines, then one by one the *Kubernetes* cluster, *Jenkins* master and *Rancher* server are installed. While the last three need to be executed in this order, since *Jenkins* needs connection credentials to an existing *Kubernetes* cluster and the *Rancher* server can be an optional element that also needs access to the cluster, the configuration sequence can be executed in parallel on all machines if desired. Ansible provides flexibility in how the configuration is called and allows deactivating certain sequences based on user input, an ability that emphasizes its importance in DevOps activities for software automation.

A use case can portray where this kind of environment is important and why an automated deployment is essential. The actual case study is represented by a city traffic monitoring and management system, which is to be developed from scratch, by a large team seeking to follow a microservice architecture to separate several functions. The project's objective is to centralize physical equipment data, from traffic cameras, air quality monitors, traffic signals, communication devices, to monitor and streamline the traffic in a city. Besides considering cars, bicycles, other vehicles and pedestrians, the project aims to integrate with other external or internal transport systems, such as trains or subway, to build a complete picture of the city traffic and identify possible bottlenecks or other issues. One approach is to have different components for collecting field data from sensors and cameras, for processing and organizing the data, generating solutions for streamlining traffic, integrating with other systems such as the train or subway ones, for sending commands to field elements and so on. This project has an extensive component architecture which is perfect for using microservices and software containers, therefore the Continuous Integration environment for it will benefit of the technologies and automation solution described in this paper. One more important aspect that makes this scenario to be the most suitable consists of the security requirements for developing a safety critical project: this kind of project should be developed from within a restricted environment, with limited access to internet via a proxy server and with private resources servers associated with private certificates.



```
vim /etc/apt.conf.d/01proxy
for i in <list>; do wget <private certificate_i>; done
cp <list> /usr/share/pki/ca-trust-store/anchors/
update-ca-trust
apt-get install <list of packages>
echo <proxy settings> > /etc/systemd/system/docker.service.d/proxy.conf
mkdir -p /etc/docker/certs.d/
cp <certificate list> /etc/docker/certs.d/
systemctl restart docker
vim ~/.ssh/authorized_keys
chmod 600 ~/.ssh/authorized_keys
```

Figure 7. Manual configuration steps

To sustain the development process, a Continuous Integration environment needs to be deployed. Usually, the DevOps activities inside a project are performed by one or few members of the team, but not all have the same attributions, so the actual deployment will be ensured by one person in most cases. Figure 7 portrays how to manually configure the environment described in Figure 6. One person will need to do the following tasks: start an SSH connection to all five machines, modify the operating system package manager configuration to include proxy servers, download all private certificates and install them one by one at system level in each target machine. Next, the package requirements need to be addressed by installing all necessary dependencies, like docker and runc. Afterwards, docker also needs to be configured so as to use a proxy for internet connection or otherwise, a private repository. Finally, extra ssh connection keys are added inside the *~/.ssh/authorized_keys* file on each target machine to provide future access. All these tasks are performed for all targets and can take from twenty minutes to half an hour per machine, depending on how many operating system packages are installed and on the network performance; therefore, in total, about an hour or two will be spent only on the configuring activities. Using an automated solution will at least cut this time in half, even if all configurations are executed sequentially, because a lot of time is spent modifying files, switching contexts and checking that no mistakes are introduced by the user.

The last three steps are already inherently fast or automated solutions by definition, using Rancher for the Kubernetes cluster and Docker containers for running the Jenkins and Rancher servers; however, time can still be saved in these stages, by eliminating manual command executions, extracting credentials and generating the Jenkins resource space inside the cluster. The total time spent here is of approximately twenty minutes, and it can be reduced by a few minutes. A big impact can be observed by using the presented solution, as a functional Continuous Integration environment can be deployed in less then an hour, versus at least an hour and a half manually (in the most ideal circumstances where the user makes little to no mistakes in manual operations). Also, using Jenkins and shared pipeline libraries along with a deployment solution provides an important advantage in the event that the existing environment is corrupted or lost: a new one can be deployed and all Jenkins activities will be generated from *Jenkinsfiles* again in a short amount of time. Of course, the previous history is lost, but the development and testing teams are not blocked for several hours.

6. Conclusion

For this study, an analysis on projects similar to the use case scenario already presented was performed based on the previous experience with traffic management and field elements monitoring systems developed in a restricted environment, so as to build a complete picture of required technologies, important principles, limitations and resulting problems. The projects consisted of large development teams, a few tens of people, where most time delays were caused by the restricted environment; people often forgot to configure proxy settings, made mistakes in the configuration, or did not install private servers' certificates, resulting in access issues to both private and public resources. Consequently, a functional Continuous Integration environment was absolutely essential to the projects, and a temporary loss or malfunction introduced large delays in both the development and delivery processes.

Continuous Integration along with continuous delivery and deployment are practices used to accelerate or increase the frequency of releasing new features of software products in a reliable way (Shahin et al., 2017). Regardless of the chosen technologies, continuous integration practices focus on approaches and challenges to improve the development process by reducing build, test, and delivery time; they aim to automate time-consuming manual repeatable tasks which are also prone to human error. A continuous integration environment is built upon a set of tools and technologies selected and configured according to the requirements of an organization or of a project, thus usually there are no two environments alike, introducing the need for a stable and configurable automated deployment solution to ensure that the provisioning is fast and repeatable.

Continuous Integration environments are provided by solutions that require payment for services such as Gitlab CI and Bitbucket Pipelines, however, this paper does not aim to provide a new Continuous Integration environment, but a way to install and configure such an environment, as most documentations of tools are designed only for ideal cases, or examples of manual installation of various components. The ideal environment, with unrestricted internet access and using only public domain resource servers is not necessarily the most common situation, and the lack of official documentation of recommendations for these situations or their vague presentation creates many problems in the installation of many technologies regardless of their popularity.

The created solution provides the possibility of automatically deploying a configurable environment and Ansible was chosen for its development as it is a very flexible, powerful, widespread technology and implicitly with an extended support platform. Alternatively, bash or python could be used to write the automation code, but it would have taken a slightly more complicated approach, while Ansible is made exactly for the desired scenario: automating machine configuration. *Jenkins* and *Rancher* were chosen as tools because they are free and similar to Ansible, they have a stable support platform and are flexible. *Jenkins* in particular offers a lot of flexibility, as it can offer a Continuous Integration platform for projects with different version systems, *git*, *clearcase*, *svn*, and its functionality is extended by plugins. However, the key is still the software container technology, a broad topic with a continuously growing popularity. Regardless of the adjacent technologies and tools, it is believed that the focus on containers for resource management will be present for a very long time to come.

REFERENCES

1. Bernstein, D. (2014). *Containers and Cloud: From LXC to Docker to Kubernetes*. IEEE Cloud Computing, 1(3), 81-84. DOI 10.1109/MCC.2014.51.

2. Hardikar, S., Ahirwar, P. & Rajan, S. (2021). *Containerization: Cloud Computing based Inspiration Technology for Adoption through Docker and Kubernetes*. In Second International Conference on Electronics and Sustainable Communication Systems (ICESC), pp. 1996-2003. DOI: 10.1109/ICESC51422.2021.9532917.

3. Jenkins (2020). *Pipeline*. Available at: https://www.jenkins.io/doc/book/pipeline/, last access April 22, 2022.

4. Laster, B. (2020). *Continuous Integration vs. Continuous Delivery vs. Continuous Deployment*. O'Reilly Media Incorporated.

5. Macarthy, R. W. & Bass, J. M. (2020). *An Empirical Taxonomy of DevOps in Practice.* In 46th Euromicro Conference on Software Engineering and Advanced Applications (SEAA), pp. 221-228. DOI: 10.1109/SEAA51224.2020.00046.

6. Madhumathi, R. (2018). *The Relevance of Container Monitoring Towards Container Intelligence.* In 9th International Conference on Computing, Communication and Networking Technologies (ICCCNT), pp. 1-5. DOI: 10.1109/ICCCNT.2018.8493766.

7. McAllister, J. (2015). *Mastering Jenkins*. Packt Publishing.

8. Oracle (2018). *Deploy Jenkins on Oracle Cloud Infrastructure Container Engine for Kubernetes (OKE).* Available at: https://blogs.oracle.com/cloud-infrastructure/deploy-jenkins-on-oke, last access April 20, 2022.

9. Pagerduty (2020). *What is Continuous Integration*. Available at: https://www.pagerduty.com/resources/learn/what-is-continuous-integration/, last access April 20, 2022.

10. Pang, C., Hindle, A. & Barbosa, D. (2020). *Understanding DevOps Education with Grounded Theory*. In IEEE/ACM 42nd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion), pp. 260-261.

11. Rădulescu, C. Z. & Rădulescu, D. M. (2017). *Atribute și Metrici Asociate pentru Evaluarea Calității Serviciilor Cloud Computing*. Revista Română de Informatică și Automatică (Romanian Journal of Information Technology and Automatic Control), 27(2), 17-30.

12. Redhat. (2020). *Overview How Ansible Works*. Available at: https://www.ansible.com/overview/how-ansible-works, last access April 22, 2022.

13. Schenker, G. N, Saito, H., Lee, H.-C. C & Hsu, K.-J. C. (2019). *Getting Started with Containerization.* Packt Publishing.

14. Shah, J. & Dubaria, D. (2019). *Building Modern Clouds: Using Docker, Kubernetes & Google Cloud Platform.* In IEEE 9th Annual Computing and Communication Workshop and Conference (CCWC), pp. 0184-0189. DOI: 10.1109/CCWC.2019.8666479.

15. Shahin, M., Ali Babar, M. & Zhu, L. (2017). *Continuous Integration, Delivery and Deployment: A Systematic Review on Approaches, Tools, Challenges and Practices*. IEEE Access, 5, 3909-3943. DOI: 10.1109/ACCESS.2017.2685629.

16. Sima, V., Hartescu, F. & Stanciu, A. (2017). *Noi Aplicații de Calcul pentru Identificarea Sistemelor*. Revista Română de Informatică și Automatică (Romanian Journal of Information Technology and Automatic Control), 27(2), 53-61.

Oana-Anastasia MINCIU is Ph.D. student at Politehnica University of Bucharest with the main field of study in DevOps and Cloud Computing, having 3 years' experience in the field. The main areas of interest are Continuous Integration/Delivery, software containers, Infrastructure as Code, and software automation.

Oana-Anastasia MINCIU este doctorand la Universitatea Politehnica din București cu domeniul principal de studiu în DevOps și Cloud Computing, având o experiență de 3 ani în domeniu. Principalele domenii de interes sunt Continuous Integration/Delivery, containerele software, Infrastructure as Code, software de automatizare.

Iulia-Lidia IACOB is a Lecturer within the Faculty of Automatic Control and Computer Science at Politehnica University of Bucharest. She received the Ph.D. degree in System Engineering from the Politehnica University of Bucharest in 2013. She has more than 10 years of professional experience both in education and IT&C companies and organizations. The main areas of interest for her research activity include: service science, business process modeling and management, manufacturing, augmented reality, Cloud Computing and DevOps.

Iulia-Lidia IACOB este șef de lucrări la Facultatea de Automatică și Calculatoare din cadrul Universității Politehnica din București. A obținut titlul de doctor în Ingineria Sistemelor la Universitatea Politehnica din București în anul 2013. Are experiență de peste 10 ani atât în domeniul academic, cât și în companii și organizații IT&C. Principalele domenii de interes pentru activitatea de cercetare sunt: știința serviciului, modelarea și managementul proceselor de lucru, realitate augmentată, Cloud Computing și DevOps.

Anca-Daniela IONIȚĂ, Ph.D., is a tenured Professor with the University Politehnica of Bucharest, Faculty of Automatic Control and Computer Science, Automatic Control and Industrial Informatics Department. She was a researcher at Politecnico di Torino and University Joseph Fourier, Grenoble, with Copernicus and Marie Curie grants. Her research interests are: IT services for hazard management, migration to service-oriented systems, and model-driven engineering.

Anca-Daniela IONIȚĂ este professor titular la Universitatea Politehnica din București, Facultatea de Automatică și Calculatoare, Departamentul Automatică și Informatică Industrială. A fost cercetător la Politehnica din Torino și Universitatea Joseph Fourier, Grenoble, cu burse Copernicus și Marie Curie. Domeniile sale de cercetare sunt: servicii IT pentru managementul hazardurilor, migrarea la sisteme orientate pe servicii și inginerie dirijată de modele.

Ştefan MOCANU graduated the Faculty of Automatic Control and Computer Science, Faculty of Automatic Control and Computer Science and the Open Architecture Systems Master within the same faculty. Since 2005 he holds a Ph.D. degree based on a doctoral stage at the Faculty of Automatic Control and Computer Science. Currently he is tenured Associate Professor within the Automatic Control and Industrial Informatics Department where he is in charge with teaching and research activities related to IT&C.

Ştefan MOCANU a absolvit Facultatea de Automatică și Calculatoare, Departamentul Automatică și Informatică Industrială și masterul în Sisteme cu Arhitectură Deschisă din cadrul aceleiaşi facultăţi. Din 2005 deţine titlul de doctor în urma stagiului de doctorat efectuat tot în cadrul Facultăţii de Automatică și Calculatoare. În prezent este conferenţiar în cadrul aceleiași facultăți, unde desfăşoară activităţi didactice şi de cercetare în domeniul IT&C.