www.ijecs.in
International Journal Of Engineering And Computer Science
Volume 13 Issue 05 May 2024, Page No. 26172-26177
ISSN: 2319-7242 DOI: 10.18535/ijecs/v13i05.4821

Continuous Integration and Delivery of Software products: Pipeline implementation

Munkhtsetseg Namsraidorj[1], Sanchirjav Lkhaasuren[1*], Bagabold Gendensuren[1], Javkhlan Rentsendorj[1], Amirlan Enkhtur[2]

[1]School of Information Technology and Electronics,National University of Mongolia
[2]Jefferson Community and Technical College, USA
[*]Corresponding Author Sanchirjav Lkhaasuren

Abstract
Continuous practices, i.e., continuous integration, delivery, and deployment, are the software development industry practices that enable organizations to frequently and reliably release new features and products. CI/CD is the best practice for teams using a DevOps methodology and is best suited to agile methodologies. CI/CD automates code integration and delivery, allowing software development teams to focus more on business requirements. Implementing CI/CD and automating the delivery process from application development to the live environment using pipelines, reducing the time and errors during manual delivery, the previously mentioned high-quality and reliable software products are delivered to the end user.
This work will automate the workflow of software products from development to supply and delivery using pipelines.

Keywords— *Continuous Integration, Continuous Delivery, Continuous Deployment, CI/CD, Kubernetes, Jenkins*

1. Foundation

As software usage proliferates, delivering updates and enhancements quickly and reliably is critical. Continuous integration (CI), continuous delivery (CD), and continuous delivery (CD) continue to be the best-tested methods for organizations, enabling the rapid and secure delivery of high-quality software. CI/CD includes the culture, operating principles, and methodologies software development teams use to deliver code changes to end users more frequently and reliably[1].

A. Software life cycle

The software development lifecycle (SDLC) is a step-by-step process that helps development teams efficiently build the highest quality software at the lowest cost. Teams follow the SDLC to help them plan, analyze, design, test, deploy, and maintain software. The SDLC also helps teams ensure that the software meets stakeholder requirements and adheres to their organization’s standards for quality, security, and compliance.

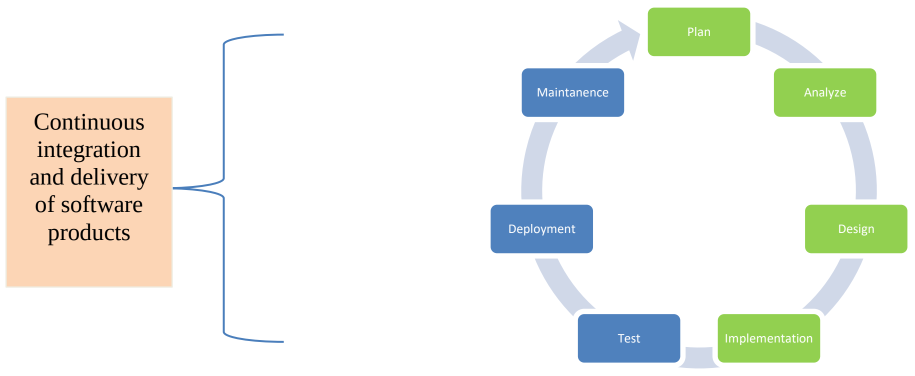


Figure 1. The software development lifecycle (SDLC)

B. Containers vs Virtual Machines-VMs

Containers and Virtual Machines are 2 approaches to packaging computing environments that combine various IT components and isolate them from the rest of the system. The main difference between the 2 is what components are isolated, which in turn affects the scale and portability of each approach[2].

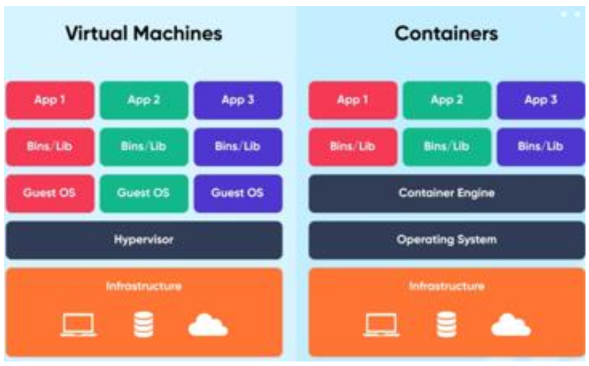


Figure 2. Virtual Machine vs Containers

A container is a unit of software that holds together all the components and functionalities needed for an application to run. Most modern applications are made up of multiple containers that each perform a specific function. One of the significant factors which contribute to the success of containers is portability[3]. Much like interconnecting LEGO™ blocks, the individual containers can easily be exchanged and moved around to different environments. Docker, an open-source platform for building, deploying, and managing containerized applications, has played a major role in the evolution of container technology over the years. Virtual machines play a crucial role in cloud computing, emulating physical computers by running operating systems in isolated instances. Multiple VMs are commonly hosted on a single server, with a hypervisor acting as a lightweight software layer positioned between the physical host and the VMs[4]. This hypervisor efficiently manages access to resources, enabling virtual machines to function as distinct servers while offering enhanced flexibility and agility. Initially gaining popularity in the 2000s due to consolidation and cost saving initiatives, the use of VMs have evolved over time.

C. Cloud-native vs Traditional IT

Beyond the technological differences, comparing containers to virtual machines is a proxy comparison between modern cloud-native IT practices and traditional IT architectures. Emerging IT practices (cloud-native development, CI/CD, and DevOps) are possible because workloads are split up into the smallest possible serviceable units—usually a function or microservice–and run in isolation, where they are independently developed, deployed, managed, and scaled. These small units are best packaged in containers, which allow multiple teams to work on individual parts of an app or service without interrupting or threatening code packaged in other containers. Traditional IT architectures (monolithic and legacy) keep every aspect of a workload tightly coupled and unable to function without the greater architecture in place. Because aspects cannot be split up, they need to be packaged as a whole unit within a larger environment, often a VM.It was once common to build and run an entire app within a VM, though having all the code and dependencies in one place led to oversized VMs that experienced cascading failures and downtime when pushing updates.

2. Research Method

Continuous software engineering is an emerging area of research and practice. It refers to develop, deploy and get quick feedback from software and customer in a very rapid cycle [5]. Continuous software engineering involves three phases: Business Strategy and Planning, Development and Operations. This study focuses on only three software development activities: continuous integration, continuous delivery and continuous deployment. Figure 1 shows the relationship between these concepts.

A. Continuous Delivery (CDE), Continuous Deployment (CD) and Continuous Integration (CI)

Continuous Delivery (CDE)

CDE is aimed at ensuring an application is always at production-ready state after successfully passing automated tests and quality checks [6]. CDE employs a set of practices e.g., CI, and deployment automation to deliver software automatically to a production-like environment. According to [6], this practice offers several benefits such as reduced deployment risk, lower costs and getting user feedback faster. Figure 1 indicates that having continuous delivery practice requires continuous integration practice.

Continuous Deployment (CD)

Continuous Deployment (CD) practice goes a step further and automatically and continuously deploys the application to production or customer environments. There is robust debate in academic and industrial circles about defining and distinguishing between continuous deployment and continuous delivery. What differentiates continuous deployment from continuous delivery is a production environment (i.e., actual customers): the goal of continuous deployment practice is to automatically and steadily deploy every change into the production environment.

Continuous Integration (CI)

Continuous Integration (CI) focuses on automating the process of integrating code changes from multiple developers, allowing for early detection and resolution of integration problems. Continuous Delivery (Continues Delivery) extends CI and prepares code changes to be deployed to the real environment or prepares a new version and lays the foundation for Continuous Delivery (Continuous deployment) [6]. Continuous Delivery is the next step in continuous Delivery, which is used to deliver new releases reliably and securely in real-world environments.

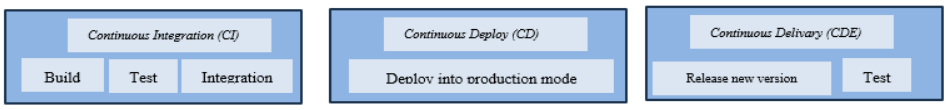

Continuous Integration (CI)
Build
Test
Integration
Continuous Deploy (CD)
Deploy into production mode
Continuous Delivery (CDE)
Release new version
Test

Figure 3. Stage of Continuous integration, Deployment and Delivery’s

B. **To automate CI/CD pipeline**

A CI/CD methodology cannot be implemented without an automated pipeline. This implementation involves the use of a variety of software tools, which are designed to allow for continuous integration (CI), delivery (CD), and environment customization during delivery, as well as all necessary system customizations[7].

Most organizations implement CI/CD pipelines by automating steps in the DevOps methodology. Key stages of microservice development, continuous testing, code-based infrastructure changes, and deploying containerized systems to real environments are automated through pipelines. The steps outlined here make continuous delivery (CD) easier.

Continuous Integration stage of the pipeline

CI allows the development, testing, and integration of code written by a developer to be automated. This is done with code versioning systems and software tools that can implement CI. Also, continuous integration pipeline automation can be different for each organization. Because it fully integrates with the code version control methodology used by developers.Continuous Deploy stage of the pipeline

In this section, you will develop a new version of the integrated code using CI and prepare it for delivery to the real environment. At this stage, it is most appropriate to check the quality of the software. Tests such as User Test (UAT), System Integration Test (SIT), and Vulnerability (VA) are automatically performed using pipelines to prevent the newly developed changes from making any errors in the real environment[7].

Continuous Delivery stage of the pipeline

The newly developed version updates the software used in the actual environment. In this process, there are cases where base and infrastructure changes are made twice. While doing this manually, there are instances where errors occur, but when continuous delivery is done using a pipeline, new development that has passed all the necessary tests can be rolled out into a live environment without errors.

3. Implementation

As mentioned earlier, CI/CD best aligns with the DevOps methodology, where a single software product is integrated, delivered, and delivered using a pipeline based on the DevOps methodology. It uses Azure DevOps, a code version control system, Jenkins to implement CI/CD, and Kubernetes, an environment where applications can run using container technology. The kubelet then continuously collects the status of those containers from Docker and aggregates that information in the control plane. Docker pulls containers onto that node and starts and stops those containers. The difference when using Kubernetes with Docker is that an automated system asks Docker to do those things instead of the admin doing so manually on all nodes for all containers.

In order to meet changing business needs, your development team needs to be able to rapidly build new applications and services. Cloud-native development starts with microservices in containers, which enables faster development and makes it easier to transform and optimize existing applications.

A. Application development with Kubernetes

Production apps span multiple containers, and those containers must be deployed across multiple server hosts. Kubernetes gives you the orchestration and management capabilities required to deploy containers, at scale, for these workloads.

Kubernetes orchestration allows you to build application services that span multiple containers, schedule those containers across a cluster, scale those containers, and manage the health of those containers over time. With Kubernetes you can take effective steps toward better IT security.
Kubernetes also needs to integrate with networking, storage, security, telemetry, and other services to provide a comprehensive container infrastructure.

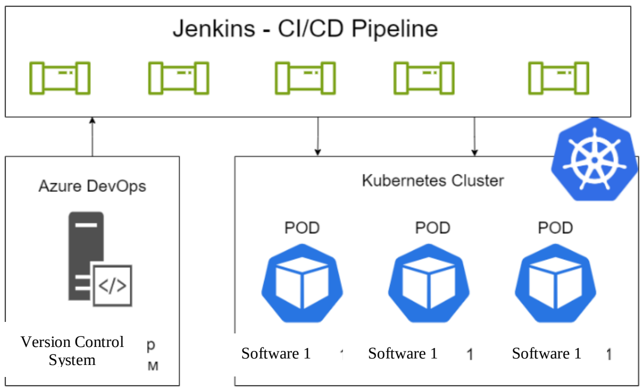


Figure 4. Architecture diagram

Many organizations use CI/CD pipelines in conjunction with cloud technologies. Software delivered using CI/CD often has a microservice architecture, which now uses container technologies. Kubernetes is an integrated management system for containers and a robust system for orchestrating software in a reliable and scalable manner[5]. One of the most important capabilities of Kubernetes is auto-scaling, which can scale horizontally and extensively. It is also adjusted using the CD.

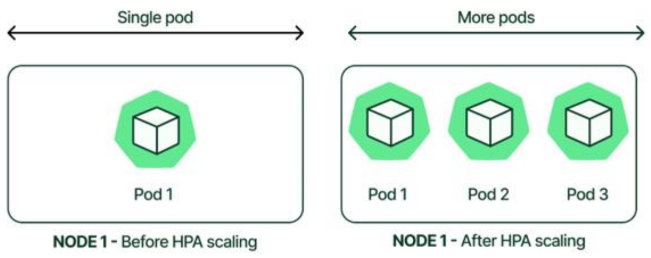


Figure 5. The Kubernetes environment is powered by software called Pods, and this figure shows scalability.

B. CI/CD Pipeline and Jenkins

Jenkins is a powerful open-source automation tool that builds pipelines to accelerate continuous software development and delivery. It uses a scripting language called Groovy. Jenkins is a powerful tool that allows you to run any command on your server or another server using Secure Shell Host (SSH).

Pipeline structure

Newly developed software goes through the following pipelines. Figure *6* shows. Software flowing through this pipeline needs to run as a container and run in a Kubernetes environment by creating an image from the developer's branch. After that, a test will be performed, and if it passes successfully, the newly written code will be merged into the Master branch code. After that, an image will be prepared and run from the main branch code again, and after passing several tests successfully, it will be prepared to start in the real environment. If the results of the previous stage of testing are considered satisfactory, the engineer managing the pipeline will automatically enter the approval to make changes in the production mode*[9]*.

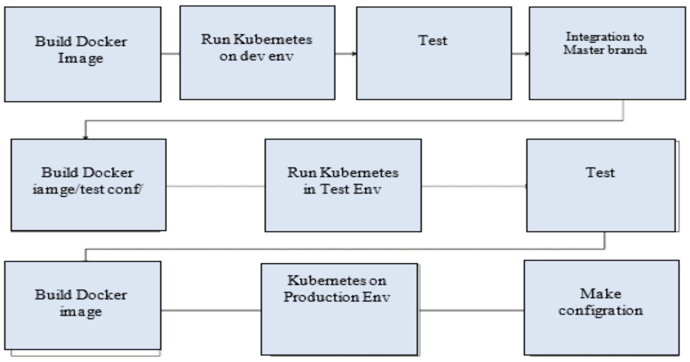

Build Docker Image
Run Kubernetes on dev env
Test
Integration to Master branch
Build Docker iamge/test conf/
Run Kubernetes in Test Env
Test
Build Docker image
Kubernetes on Production Env
Make configration

Figure 6. Pipeline structure

Pipeline implementation

The Jenkins system has its own user web page for monitoring. The source code of the software and the source code of the CI/CD pipeline are located in the Repo menu of the Azure DevOps system, which can be configured via the Jenkins system user web page. This sets the path where the original pipeline code is stored and the keywords used to perform the necessary infrastructure configuration.

However, the pipeline is written using Groovy, a structured scripting language. Figure 7 shows the stages of the self-written pipeline.

Figure 7. Pipeline operation

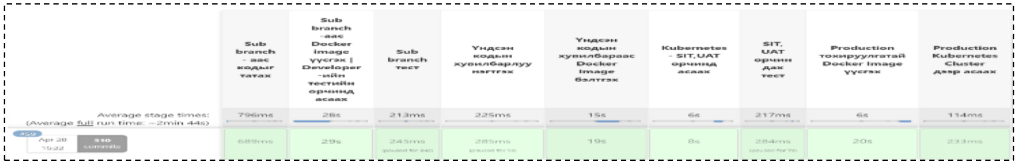
C. Resuls

This pipeline has been tested on two different software. One is a Node application, and the other is a Flask application. Each program has different pipeline times. The table below compares system-dependent phase times for each program.

Table 1. Compares system-dependent phase times

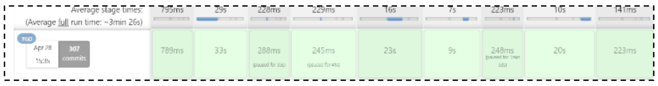
№	Software	Build Docker image	Run Kubernetes on Dev	Build Docker image	Run Kubernetes on SIT,UAT
1	NodeApp	29 сек	7 сек	23 сек	9 сек
2	Flask App	33 сек	10 сек	19 сек	8 сек

Creating a Docker Image from a non-core version of the code is 23 seconds for NodeApp and 29 seconds for flaskApp. This time is independent of the pipeline and depends on the size of the application and the framework used. In the same way, the duration of other stages directly depends on the program's size. Figure 6 and Figure 7. However, the quality and capacity of the pipeline can be determined by the following factors: These include: 1. Depending on the test results and quality results, it can be defined as stopping the delivery of the program or returning the program depending on the normality or abnormality of the program after the pipeline has been delivered to the real environment.



	Sub branch - аас кодыг татах	Sub branch -аас Docker image үүсгэх | Developer -ийн тестийн орчинд асаах	Sub branch тест	Үндсэн кодын хувилбарлуу нэгтгэх	Үндсэн кодын хувилбараас Docker image бэлтгэх	Kubernetes - SIT, UAT орчинд асаах	SIT, UAT орчин дах тест	Production тохируулгатай Docker image үүсгэх	Production Kubernetes Cluster дээр асаах
Average stage times: (Average full run time: ~2min 44s)	796ms	28s	213ms	225ms	15s	6s	217ms	6s	114ms
Apr 28 15:22	689ms	29s	245ms	285ms	19s	8s	284ms	20s	233ms

Figure 8. Flask App software



Average stage times: (Average full run time: ~3min 26s)	795ms	29s	228ms	229ms	16s	7s	223ms	10s	141ms
Apr 28 15:05	789ms	33s	280ms	245ms	23s	9s	248ms	20s	223ms

Figure 9. NodeJs App software

5.Conclusion

The main purpose of this work is the pipeline, which can be used in any workflow of your choice. In general, the beginning of the pipeline should start with a non-main version of the code, merge with the main version, and release the main version into the real world. The previously mentioned pipeline will deploy a non-core code version to the developer environment and run it as a test, deciding whether to merge development into the core version. This will be the basis for introducing a quality and error-free system. The following important part of the pipeline is to deploy and test the main release code in the SIT and UAT environments. This stage can be done manually, and depending on the results of the tests, it will be decided whether the development will enter the real environment. This step prevents errors that might occur in a real environment. Refinement of the last critical step, the deployment stage, allows for delivery with minimal downtime during code deployment to the live environment without human error. Embracing CI/CD implementation as a standard will deliver software development to the end user more reliably, securely, and quickly. We can see this in pictures 7, 8, and 9.

Reference

1. M. Fowler. "Continuous Integration. Available at: http://martinfowler.com/articles/continuousIntegration.html [Last accessed: 21 October 2015]." 21/10/2015; http://martinfowler.com/articles/continuousIntegration.html.
2. M. Shahin, M. A. Babar, and L. Zhu, "Continuous integration, delivery and deployment: a systematic review on approaches, tools, challenges and practices," IEEE Access, vol. 5, pp. 3909–3943, 2017.
3. K. Gallaba, "Improving the robustness and efficiency of continuous integration and deployment," in 2019 IEEE International Conference on Software Maintenance and Evolution (ICSME). IEEE, pp. 619–623
4. B. Fitzgerald, and K.-J. Stol, ―Continuous Software Engineering: A Roadmap and Agenda,‖ *Journal of Systems and Software,* vol. 123, 2017.
5. Ska, Yasmine & Publications, Research. (2019). A Study And Analysis Of Continuous Delivery, Continuous Integration In Software Development Environment. Ssrn Electronic Journal. 6. 96-107.
6. Arachchi S.A.I.B.S., Perera I. Continuous Integration And Continuous Delivery Pipeline Automation For Agile Software Project Management; Proceedings Of The 2018 Moratuwa Engineering Research Conference (Mercon); Moratuwa, Sri Lanka. 30 May–1 June 2018.
7. Maryam S., Javdani G.T., Rasool S. Quality Aspects Of Continuous Delivery In Practice. *Int. J. Adv. Comput.Sci. Appl.* 2018;**9**:210–212
8. Burns B., Grant B., Oppenheimer D., Brewer E., Wilkes J. Borg, Omega And Kubernetes. *Commun. Acm.* 2016;**14**:70–93.
9. Andrawos M., Helmich M. *Cloud Native Programming With Golang: Develop Microservice-Based High Performance Web Apps For The Cloud With Go.* Packt Publishing Ltd.; Birmingham, Uk: 2017