Automated Cloud Infrastructure, Continuous Integration and Continuous Delivery using Docker with Robust Container Security

Somya Garg
Ernst & Young, LLC
somya.garg@ey.com

Satvik Garg
Jaypee University of Information Technology
satvikgarg99@gmail.com

Abstract

Recent industry trends have determined that cloud computing has proven to be a very demanding and lucrative field. Regardless of the size of an organization, they are transitioning to the cloud platform for multiple reasons. Despite cloud providers effort to make the transition smooth, we firmly believe that we have not reached to that point yet, but with the proper set of tools and leveraging best practices, we can certainly make this transition efficient and robust. There are various essential layers that all are working closely together to make cloud automation and platform orchestration successful. This paper provides a brief cloud technology overview and guideline on how cloud computing technology is transforming industrial automation for the future, enhancing productivity and cost optimization. This paper also examines a strategy for effective deployment and maintenance so that automated resources can be managed effectively, provisioned rapidly and released with little to no admin efforts.

Keywords— docker; security; cloud infrastructure; continuous integration; continuous delivery

1. Introduction

Corporations are not required to make an upfront investment to buy expensive hardware and the initial phase of building the infrastructure since the advent of cloud service providers' pay-as-you-go model. The use of virtualization and orchestration has reduced the time required to deploy these computing resources from days to a few minutes. However, for building a cloud-based infrastructure, this is not enough. The engineering teams need to build a strategy for effective deployment and maintenance so that the automated resources can be managed in such a way that they can be leveraged efficiently, provisioned rapidly and released with fewer admin efforts. In [1] and [2] authors have explained how IaaS transition has changed the industry and an overview on cloud system disaster recovery based on Infrastructure as a Code (IaaC) concept. There are multiple architectures just like these developed based on the organization's needs.

According to the Gartner Industry trends [3], Gartner predicts that by 2025, 80% of enterprises will shut down their traditional data centers - it is safe to say that major institutions have already started this transition. In making this transition so adoptive, containerization plays a key role along with the open-source technologies have driven the necessary cloud orchestration. Given it is such a significant trend and diversity of organizations needs, automation techniques are not yet been standardized. Open-source IT automation tools such as Terraform and Terragrunt are being utilized to automate various cloud infrastructures that previously required a significant amount of manual work. Automation not only is a viable option for the organizations from a cost savings point of view, but it plays a massive role in continuous development and delivery [4]. Authors in [5] and [6] have demonstrated that how Continuous integration (CI) and continuous delivery (CD) is not only a practice, it is a culture that development team has to adopt in order to succeed in a highly competitive environment. This culture aligns operating principles and includes methods that enable development teams to deliver high-quality code more frequently and reliably. The methodology is also called the CI/CD pipeline and is one of the common DevOps practices.

This paper will focus on plan, code, deploy, and testing phase of the automation process. Additionally, we discuss how Continuous Integration (CI) and Continuous Distribution (CD) are an essential part of automation and how to implement CI/CD using Docker containers. Additionally, this paper talks about how to implement Docker Security to make automation robust and scalable.

2 Continuous Integration

Continuous Integration is a software development process. The right side of the Figure:(1) explains how the developer's code is continuously merged into a central repository, which also acts as a source control system, several times a day. Each check-in is verified through the automated test and build processes to ensure success in integration between builds. By integrating early and often, development teams can prevent running into large integration's issues between the code updates that are being worked on.

Developers can Check-In code frequently, this allows developer teams to check if the code passes testing phase or not consistently. The practice of Continuous Integration helps development teams by faster issue discovery, which means the issues and errors in code are found quickly through the automated test. The

978-1-7281-1198-8/19/$31.00 ©2019 IEEE
DOI 10.1109/MIPR.2019.00094

IEEE computer society

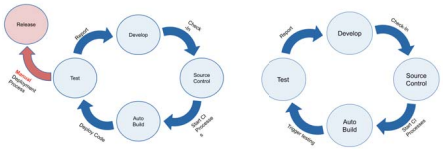

Figure 1: Continuous Integration Overview

other benefit is, it reduces the possibilities of integration issues, large scale integration issues are less familiar with the adoption of CI. This streamlines development and makes it transparent, it increases the visibility of work that has been done throughout the team and helps everyone stay on schedule. This boosts developers' productivity, less time is spent on bug-fixes and integration's issues. The author Stolberg [7] [8] explained how a continuous integration system could support agile software development and testing environment.

3 Continuous Delivery

Continuous Delivery is a software engineering policy that aims to reduce the time needed to produce software. The left side of Figure:(1) focuses on deploying smaller changes in a more frequent manner. The Releases must go through the automated testing steps and a final User Acceptance test. All the Changes will be ready to be released through a manual process once tests are passed. The manual process allows for changes to be released only when deemed necessary. The author states in [9], that the organizations that have adopted the CD in their systems have reported remarkable benefits. The benefit could be reduced deployment risk, if the developers release smaller changes more often, all of which have to pass through automated and user acceptance changes, there are fewer chances of an issue reaching production. Also, the factor could be increased user feedback, and every code change must go through user acceptance before being released. The code release is also a manual process that must be agreed upon by all stakeholders. The other notable feature is quicker reaction time, more frequent releases allow teams to react to technology trends and release to market quicker. The practice of Continuous Delivery helps development teams by the faster feature implementation; it means to release smaller changes more frequently, the overall product can evolve at a quicker rate.

However, adopting Continuous Delivery can be very challenging for an ample number of reasons, as mentioned in [10]. Companies need to make sure they obtain an agreement from an organization's stakeholders whose goals may be conflicting from each other. Obtaining continuous support in a dynamic enterprise culture is difficult. The other challenge is to manage the development team's propulsion as the application migration to the Continuous Delivery can take a considerable time investment. To conquer these challenges [11], companies can provide proper training and coaching to the development teams and explain to them how this implementation can be beneficial in the future; constitute a dedicated DevOps team which help in developing the DevOps culture; make sure to kick-start the process with the easily deploy-able applications.

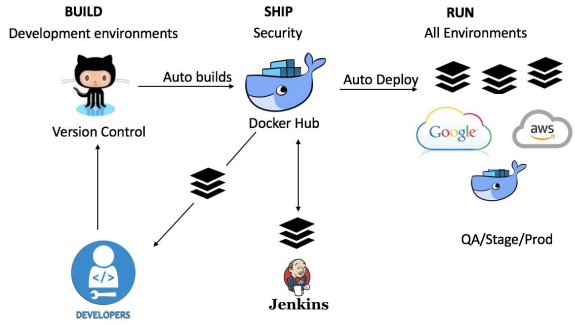

Figure 2: CI/CD Architecture using Docker

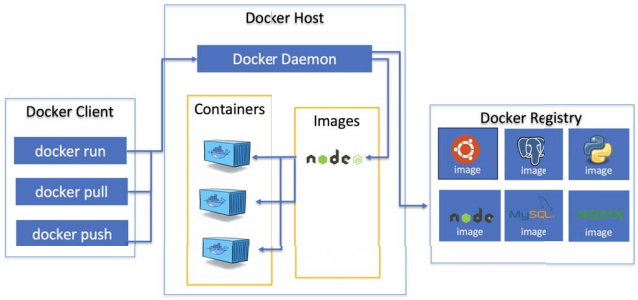

Figure 3: Docker Architecture

4 CI/CD Using Docker

The visual presentation of CI/CD using Docker is displayed in Figure:(2). Docker will verify if the required information is provided in the container to build the application. Docker performs the instructions listed in the Dockerfile. The first step is to load with the Docker image (locally or Docker downloads it). Docker finishes the build and reports the outcome. Docker caches the results of executing a line in the Dockerfile for use in a future build. If something has changed, then Docker will invalidate the current and all following lines to ensure everything is up-to-date. The command *–no-cache=true* can be used to instruct Docker not to use the cache as part of the build.

4.1 *Architectural Components*

In order to implement this architecture, following compute resources are must be provided:

- Docker host running commercially supported Docker Engine and the Docker hub components
- Jenkins master running commercially supported Docker Engine and the Jenkins platform in the docker container
- Jenkins slave running commercially supported Docker Engine with SSHD enabled and a Java runtime installed, (In a production environment there will be likely multiple slaves nodes to make the architecture highly available and highly scalable)

- There will be a Github repository which will be used to hosts the application and will be tested along with the Dockerfile to build the application
- Last step would be, the Jenkins master will use the Github plugin that will let Jenkins job to be prompted when a change is pushed to the repository by the developer.

4.2 *Architecture work-flow*

In this section we will go over the Continous Integration displayed in Figure:(2):

- Developers builds and pushes code to Github
- Github will use a webhook to notify Jenkins of the recent change
- Jenkins will pull the Github repository, including the Dockerfile describing the image along with the application
- Jenkins will then build the Docker image of that application on the Jenkins slave node
- Jenkins will run the Docker container on the slave node and will execute the appropriate tests
- If the tests are successful, the docker image is then pushed to the Docker Hub with the appropriate version number

4.3 *Dockerfile*

```
FROM scratch
MAINTAINER FIRSTNAME LASTNAME
ARG git_repository="Your git repository"
ARG git_branch="Git branch"
ARG git_commit="unknown"
ARG built_on="Timestamp"
LABEL git.repository=$git_repository
LABEL git.commit=$git_commit
LABEL git.branch=$git_branch
LABEL build.dockerfile=/Dockerfile
LABEL build.on=$built_on
EXPOSE 12345
COPY ./Dockerfile /Dockerfile
ADD bin/linux/helloservice /
CMD ["/helloservice"]
```

NOTE*: The ARG values must be replaced by your git repository values for the CI/CD pipeline implementation

4.4 *Dockerfile.build*

```
FROM golang:latest
MAINTAINER FIRSTNAME LASTNAME

ENV GOOS Linux
ENV CGO_ENABLED 0
RUN mkdir -p
    /go/src/github.com/user/helloservice
WORKDIR
    /go/src/github.com/user/helloservice
```

5 Docker Container Security

According to the author [12] the increasing need for shorter development cycles, Continuous Integration, and Continuous Delivery, and cost savings in the cloud infrastructures led to the rise of containers, which are more reliable and efficient than the traditional virtualization technology. Docker containers are widely adopted by majority of companies who are automating their IT infrastructure. Docker containers provide agility, developers of applications can avoid worrying about differences in user, environments and the availability of dependencies. Docker containers are very portable; developers can build software locally, knowing that it will run identically regardless of the host environment. With the docker containers, you can have a Control of your systems; operations engineers can concentrate on networking, resources, and up-time and spend less time configuring environments and battling system dependencies.

5.1 *Docker Architecture*

Docker is implemented using a client-server architecture, conceptual docker architecture is presented in Figure:(3).

- Docker client talks to the Docker Daemon
- Docker Daemon runs on a host machine
- A user interacts with Docker Client and does not directly interact with a Daemon

In order to make sure that docker architecture is implemented and designed efficiently we must follow some best practices. In following section we will discuss some best practices to follow.

5.2 Container Security Best Practices

- Namespaces: Docker automatically creates a set of namespaces and control groups for that container, it provides isolation from other containers.
- Control Groups (aka cgroups): They are responsible for resource accounting and limiting resource limits help against DoS attacks. An overutilized (or attacked) container cannot take resources from other containers on the host. The cgroups such as memory, CPU, devices, blkio which helps in securing docker containers.
- Docker Daemon: Running the daemon requires root which makes this an attack vector. Make sure only to run docker on the host. All other services should be in containers hosted by docker. Admin tools like SSH and monitoring are okay to be on the host. Make sure to restrict access, dont expose outside your network. Make sure to use Authentication and authorization plugins to control access to the daemon.
- Linux Kernel Capabilities and Security: Docker containers are started with a limited set of capabilities by default.

Some capabilities allow for more fine-grained control than root/non-root. The containers should almost never need root. On bare metal (or VMs), things like ssh and cron would need root, but with containers, they are handled by the supporting infrastructure. When they do need root, it will have far fewer privileges than common root. The developers can also deny mount operations and access to raw sockets.

There are several things to keep in mind; containers are generally secure as long as you run processes inside as non-root. Third party containers might not be safe; you are trusting whoever made that image. The command below will warn you if you try to pull down an unsigned image.

```
sudo export DOCKER_CONTENT_TRUST=1
```

Instead of using third-party containers, use a private registry like Docker Trusted Registry. Also, make sure to do a Vulnerability scanning, the images can be automated for known vulnerabilities. There are various tools in the market like [13][14][15] AppArmor, SELinux, GRSEC and similar hardening tools that companies are using at an ample amount.

5.3 Industry recommended tools for Docker Security

- Docker Bench for Security: A script that checks for dozens of best practices for using Docker in production, it also checks Host Configuration and Docker Daemon configuration.
- CoreOSs Clair: scan images locally or on a public image repository for security vulnerabilities.
- Docker Security Scanning: it helps in vulnerability scanning for docker hosted containers.
- AppArmor/SELinux: it is not specific to Docker or containers but is another layer of defense.

6 Conclusion

In this paper we began from understanding the industry trend to transition from traditional on-premise data-center to cloud and why cloud infrastructure automation is a must if developer teams want to produce better, faster, and scalable products. Then we discussed the CI/CD pipeline using docker container and advantages of docker security and how best practices can be followed for optimal result. We also looked at the implementation of CI/CD using docker container and how it can be leveraged for the development team of any size. In future, we plan to make the boilerplate code available to open source community that will help developers to get started even faster.

References

[1] R. Moreno-Vozmediano, R. S. Montero, and I. M. Llorente, "Iaas cloud architecture: From virtualized datacenters to federated cloud infrastructures," *Computer*, vol. 45, no. 12, pp. 65–72, Dec 2012.

[2] O. Lavriv, M. Klymash, G. Grynkevych, O. Tkachenko, and V. Vasylenko, "Method of cloud system disaster recovery based on" infrastructure as a code" concept," in *Advanced Trends in Radioelecrtronics, Telecommunications and Computer Engineering (TCSET), 2018 14th International Conference on*, IEEE. IEEE, 2018, pp. 1139–1142.

[3] "Gartner top 10 trends impacting infrastructure operations for 2019," https://www.gartner.com/smarterwithgartner/top-10-trends-impacting-infrastructure-and-operations-for-2019/, accessed January, 2019.

[4] L. Williams and A. Cockburn, "Agile software development: it's about feedback and change," *Computer*, vol. 36, no. 6, pp. 39–43, June 2003.

[5] "What is ci/cd? continuous integration and continuous delivery explained," https://www.infoworld.com/article/3271126/ci-cd/what-is-cicd-continuous-integration-and-continuous-delivery-explained. html, accessed January, 2019.

[6] N. Rathod and A. Surve, "Test orchestration a framework for continuous integration and continuous deployment," pp. 1–5, Jan 2015.

[7] S. Stolberg, "Enabling agile testing through continuous integration," in *2009 Agile Conference*, IEEE. IEEE, Aug 2009, pp. 369–374.

[8] B. Fitzgerald and K.-J. Stol, "Continuous software engineering: A roadmap and agenda," vol. 123. Science Direct, 2017, pp. 176 – 189. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0164121215001430

[9] L. Chen, "Continuous delivery: Overcoming adoption challenges," *Journal of Systems and Software*, vol. 128, pp. 72 – 86, 2017. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0164121217300353

[10] E. Laukkanen, J. Itkonen, and C. Lassenius, "Problems, causes and solutions when adopting continuous deliverya systematic literature review," *Information and Software Technology*, vol. 82, pp. 55 – 79, 2017. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0950584916302324

[11] B. Adams, S. Bellomo, C. Bird, T. Marshall-Keim, F. Khomh, and K. Moir, "The practice and future of release engineering: A roundtable with three release engineers," *IEEE Software*, vol. 32, no. 2, pp. 42–49, Mar 2015.

[12] T. Combe, A. Martin, and R. D. Pietro, "To docker or not to docker: A security perspective," *IEEE Cloud Computing*, vol. 3, no. 5, pp. 54–62, Sep. 2016.

[13] "Docker and apparmor: 0.000 foot-view," https://medium.com/lucjuggery/docker-apparmor-30-000-foot-view-60c5a5deb7b, Article, accessed January, 2019.

[14] "Docker and selinux: 0.000 foot-view," https://medium.com/lucjuggery/docker-selinux-30-000-foot-view-30f6ef7f621, Article, accessed January, 2019.

[15] "Docker security best practices," https://blog.sqreen.io/docker-security/, Article, accessed January, 2019.