CI/CD With Containers

WRITTEN BY SHAY SHMELTZER, DIRECTOR OF PRODUCT MANAGEMENT, ORACLE CLOUD
WRITTEN BY ANDREA MORENA, DIRECTOR OF PRODUCT MANAGEMENT, ORACLE CLOUD

CONTENTS

CI/CD WITH CONTAINERS

As organizations struggle to accelerate the cycles of development and delivery of solutions for their customers, many are adopting Continuous Integration, Delivery, and Deployment. In this Refcard, we'll try and explain how containers can help you adopt and improve these methodologies, but first lets clear up some of the key concepts.

Continuous Integration (CI) refers to the practice of frequently merging new software using a single line of code. Continuous Delivery refers to the production of packaged software out of code in frequent cycles. Likewise, Continuous Deployment refers to the deployment the packaged software to a runtime platform in frequent cycles.

While each step of the CI/CD is independent from the others, and some teams choose to automate part of the CI/CD chain and manually do the other parts, a complete automation of the whole CI/CD pipeline delivers the most benefit.

A team that has a completely automated CI/CD cycle in place achieves a faster response cycle on code changes. Once code has been merged and deployed it is available for testing and verification, leading to accelerated DevOps cycles. Problems can be found earlier and working software can be delivered to customers faster.

It's worth mentioning that CI/CD automation doesn't necessarily mean deployment all the way to your production environment. You can also look at deployment to QA environment as the end point where automation takes your team, keeping deployment to production as a manual step.

Many teams who are adopting Agile rely on an automated CI/CD cycle to help them achieve the goals of quickly responding to customer demands and producing working, reliable software in short cycles.

A BRIEF HISTORY OF CONTAINERS

The idea of what we now call container technology first appeared in 2000 as FreeBSD jails, a technology that allows the partitioning of a FreeBSD system into multiple subsystems, or jails. Jails were developed as safe environments that a system administrator could share with multiple users inside or outside of an organization. In a jail, the intent was that processes get created in a modified chrooted environment — where access to the filesystem, networking, and users is virtualized — and could not escape or compromise the entire system. However, jails were limited in implementation, and methods for escaping jailed environments were eventually discovered.

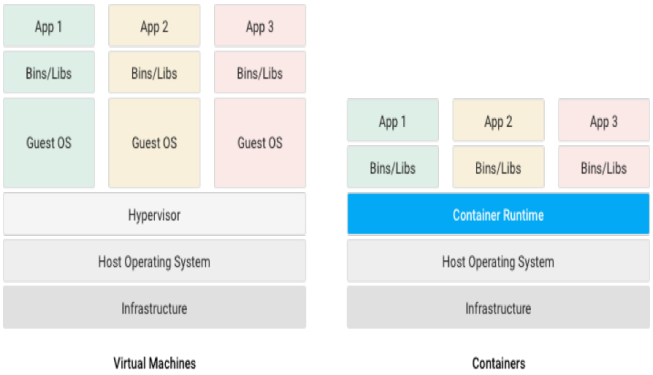
Very quickly, more technologies combined to make this isolated approach a reality. Control groups (cgroups) is a kernel feature that controls and limits resource usage for a process or groups of processes. And systemd, an initialization system that sets up the userspace and manages their processes, is used by cgroups to provide greater control over these isolated processes. Both of these technologies, while adding overall control for Linux, were the framework for how environments could successfully stay separated.

Advancements in kernal namespaces provided the next step for containers. With kernel namespaces, everything from process IDs to network names could be virtualized within the Linux kernel. One of the newer ones, User namespaces, "allow per-namespace mappings of user and group IDs. In the context of containers, this means that users and groups may have privileges for certain operations inside the container without having those privileges outside the container." The Linux Containers project (LXC) then added some much-needed tools, templates, libraries, and language bindings for these advancements — improving the user experience when using containers. LXC made it easy for users to start containers with a simple command line interface.

Put simply, a container consists of an entire runtime environment: an application (plus all its dependencies), libraries and other binaries, and configuration files needed to run it, bundled into one package.

WHY USE CONTAINERS?

Containers are a solution to the problem of how to get software to run reliably when moved from one computing environment to another. This could be from a developer's laptop to a test environment, from a staging environment into production, and perhaps from a physical machine in a data center to a virtual machine in a private or public cloud.



There are many container formats available. Docker is a popular open-source container format that is supported on many platforms.

WHAT ARE DOCKER AND KUBERNETES?

WHAT IS DOCKER?

The word "Docker" refers to several things. This includes an open source community project; tools from the open source project; Docker Inc., the company that is the primary supporter of that project; and the tools that the company formally supports. The fact that the technologies and the company share the same name can be confusing.

Docker as a technology added a lot of new concepts and tools — a simple command line interface for running and building new layered images, a server daemon, a library of pre-built container images, and the concept of a registry server. Combined, these technologies allowed users to quickly build new layered containers and easily share them with others.

HOW DOES DOCKER WORK?

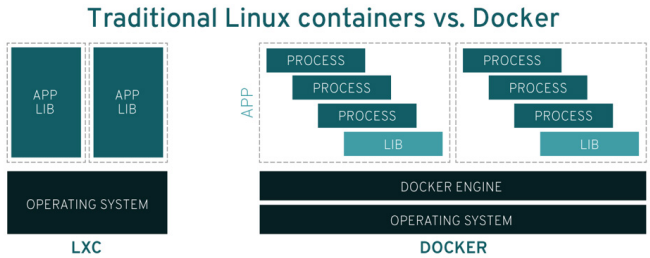
The Docker technology uses the Linux kernel and features of the kernel, like Cgroups and namespaces, to segregate processes so they can run independently. This independence is the intention of containers — the ability to run multiple processes and apps separately from one another to make better use of your infrastructure while retaining the security you would have with separate systems.

Container tools, including Docker, provide an image-based deployment model. This makes it easy to share an application, or set of services, with all of their dependencies across multiple environments. Docker also automates deploying the application (or combined sets of processes that make up an app) inside this container environment.

These tools built on top of Linux containers — what makes Docker user-friendly and unique — gives users unprecedented access to apps, the ability to rapidly deploy, and control over versions and version distribution.

IS DOCKER TECHNOLOGY THE SAME AS TRADITIONAL LINUX CONTAINERS?

No. Docker technology was initially built on top of the LXC technology — what most people associate with "traditional" Linux containers — though it's since moved away from that dependency. LXC was useful for lightweight virtualization, but it didn't have a great developer or user experience. The Docker technology brings more than the ability to run containers — it also eases the process of creating and building containers, shipping images, and versioning of images (among other things).



Traditional Linux containers use an init system that can manage multiple processes. This means entire applications can run as one. The Docker technology encourages applications to be broken down into their separate processes and provides the tools to do that. This granular approach has its advantages.

THE ADVANTAGES OF DOCKER CONTAINERS

- *Rapid application deployment* – containers include the minimal runtime requirements of the application, reducing their size and allowing them to be deployed quickly.

- *Portability across machines* – an application and all its dependencies can be bundled into a single container that is independent from the host version of Linux kernel, platform distribution, or deployment model. This container can be transferred to another machine that runs Docker, and executed there without compatibility issues.
- *Version control and component reuse* – you can track successive versions of a container, inspect differences, or roll back to previous versions. Containers reuse components from the preceding layers, which makes them noticeably lightweight. This supports an Agile development approach and helps make continuous integration and deployment (CI/CD) a reality from a tooling perspective.
- *Sharing* – you can use a remote repository to share your container with others. Oracle provides a registry in the cloud for this purpose, and it is also possible to configure your own private repository.
- *Lightweight footprint and minimal overhead* – Docker images are typically very small, which facilitates rapid delivery and reduces the time to deploy new application containers.
- *Simplified maintenance* – Docker reduces effort and risk of problems with application dependencies.

To allay fears of a single vendor controlling such an important technology, Docker Inc. contributed many of the underlying components to community-led open-source projects (runc is part of the Open Containers Initiative and containerd has been moved to the CNCF).

Today Docker, among other companies such as Oracle, are members of the Open Container Initiative (OCI) and are enabling an open industry standardization of container technologies.

ARE THERE LIMITATIONS TO USING DOCKER?

Docker, by itself, is very good at managing single containers. When you start using more and more containers and containerized apps, broken down into hundreds of pieces, management and orchestration can get very difficult. Eventually, you need to take a step back and group containers to deliver services — networking, security, telemetry, etc. — across all of your containers. That's where Kubernetes comes in.

WHAT IS KUBERNETES?

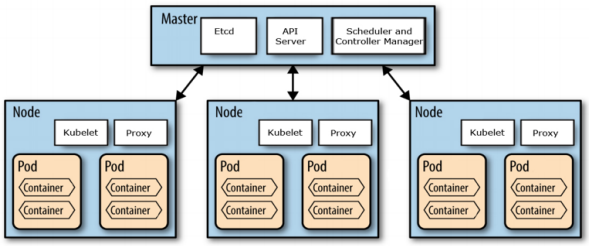
Kubernetes, or k8s, is an open source platform that automates Linux container operations. It eliminates many of the manual processes involved in deploying and scaling containerized applications. In other words, you can cluster together groups of hosts running Linux containers, and Kubernetes helps you easily and efficiently manage those clusters. These clusters can span hosts across public, private, or hybrid clouds.

Kubernetes was originally developed and designed by engineers at Google. Google was one of the early contributors to Linux container technology and has talked publicly about how everything at Google runs in containers. Google donated the Kubernetes project to the newly formed Cloud Native Computing Foundation in 2015.

WHY DO YOU NEED KUBERNETES?

A rudimentary application of Linux containers treats them as fast, efficient virtual machines. Once you scale this to a production environment and multiple applications, it's clear that you need multiple, collocated containers working together to deliver the individual services. This significantly multiplies the number of containers in your environment, and as those containers accumulate, the complexity also grows.

Real production applications span multiple containers. 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.



Master: The machine that controls Kubernetes nodes. This is where all task assignments originate.

Node: These machines perform the requested, assigned tasks. The Kubernetes master controls them.

Pod: A group of one or more containers deployed to a single node. All containers in a pod share an IP address, IPC, hostname, and other resources. Pods abstract network and storage away from the underlying container. This lets you move containers around the cluster more easily.

Replication controller: This controls how many identical copies of a pod should be running somewhere on the cluster.

Service: This decouples work definitions from the pods. Kubernetes service proxies automatically get service requests to the right pod — no matter where it moves to in the cluster or even if it's been replaced.

Kubelet: This service runs on nodes and reads the container manifests and ensures the defined containers are started and running.

kubectl: This is the command line configuration tool for Kubernetes.

WHAT CAN YOU DO WITH KUBERNETES?

- Orchestrate containers across multiple hosts.
- Make better use of hardware to maximize resources needed to run your enterprise apps.

- Control and automate application deployments and updates.
- Mount and add storage to run stateful apps.
- Scale containerized applications and their resources on the fly.
- Declaratively manage services, which guarantees the deployed applications are always running how you deployed them.
- Health-check and self-heal your apps with autoplacement, autorestart, autoreplication, and autoscaling.

THE RELATIONSHIP BETWEEN CI/CD AND CONTAINERS

The advent of continuous integration and continuous delivery (CI/CD) has revamped the traditional application development process in a drastic way. The CI/CD-based process involves DevOps — the paradigm shift that brings developers, QA engineers, and operations managers together on one platform. It generates frequent feedback at every stage and follows an automated process across your build, test, and production environments.

But can the CI/CD-based application development process be improved even further? The most modern approach is to use containerization to bring even more flexibility and benefits.

Bringing containerization to CI/CD has several advantages over the traditional software development process as well as the CI/CD-based software development process:

- Containers are a **lightweight** and ready to run **portable software**: optimum use of the infrastructure and much faster application deployment.
- **One-click infrastructure provisioning and decommissioning:** quick and dirty dev and test environment for CI/CD pipelines.
- **Faster and error-proof deployment**: removes all technical snags related to driver compatibility, library conflicts etc.
- **Reduce time and costs** associated with the dependency between Development and Operations (e.g for release management).
- **Eliminate restrictions** on using tools, frameworks, and testing suites. Enterprises have the freedom to choose whichever tools they want to use.
- **Automation**: The CI/CD framework over containers can automatically build, package, and deploy applications. This also completely obviates manual errors.

HOW TO BUILD A CONTAINER ARTIFACT AND DEPLOY TO KUBERNETES

DEFINE A BUILD JOB

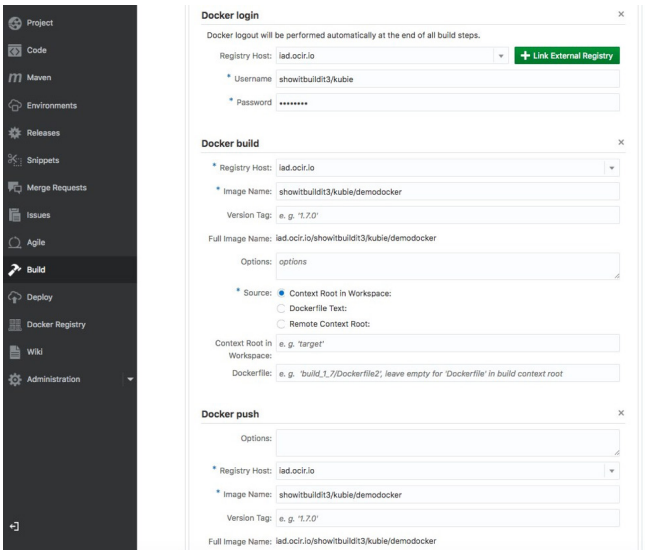
Build jobs are the way you define automated tasks that your CI/CD platforms execute. For this Refcard, we'll use Oracle Developer Cloud Service as an example, which provides a CI/CD engine that orchestrates and executes these build jobs. Developer Cloud Service can also automate CI/CD for various types of software deliverables, but in this section, we'll focus on automating the cycle for Docker containers.

A build job for Docker containers can include several steps – all of them leverage the Docker command line to execute activities on your definition files and the images generated from them.

First, you'll want to build the Docker image to verify that it is configured properly. In the build step, you configure where the definition files are located (for example, in your Git repository's root directory). You might want to add a specific tag to your image so you can easily manage multiple versions. This can be done either as part of the build or in an independent build step.

Next. you might want to publish the built Docker image to a repository of images – which might mean that you'll want to first login into that repository. There are public repositories, such as DockerHub, as well as private repositories, such as the one provided by the Oracle Cloud Infrastructure Registry.

In Developer Cloud Service, you can sequence all of these steps in a single build job or break them into separate jobs. Developer Cloud Service also supports a variety of other Docker commands, and all of them can be defined in a declarative way, reducing chances of errors in your build scripts.



A simple set of steps defined in a declarative way in Developer Cloud Service to publish a Docker image to Oracle's Docker Registry

Another option is to just write a shell script with the Docker commands manually and execute it in your CI/CD flow.

TIE THE BUILD JOB TO YOUR GIT REPOSITORY

Adopting the "Infrastructure as Code" approach means that your

Docker container definition files should reside in a version management repository, like any other piece of code your team produces.

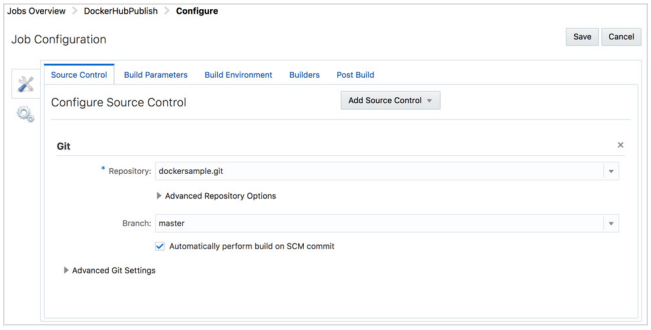
In Developer Cloud Service, you can tie your build jobs to hook up with your Git-based source code repositories. You can dedicate specific build jobs for specific branches of your code.

AUTOMATE EXECUTION OF BUILD JOB

Usually, you'll aim for the build job to automatically execute whenever someone changes the code that your CI/CD relies on. This will mean hooking up your source repository with your build job through your CI/CD engine.

Some teams automate execution of Docker builds based on a schedule. For example every night, after the team is done with that day's development effort, a build job will pick up the latest changes and deploy them on QA instances.

In Oracle Developer Cloud Service, for example, you can check a box that tells a build job to automatically start when a change has been made to a specific branch of your Git repository. You'll usually associate this with changes to the master branch, which will contain code that is "production ready" and has passed peer review. Alternatively, you can set a schedule for your build jobs to automate execution based on a specific schedule.



Build job configured to pick code from a specific branch of the git ository and execute automatically when changes have been done to this branch.

AUTOMATE KUBERNETES CLUSTER DEPLOYMENT WITH TERRAFORM

Now that we have created the Docker containers and published them, the next step is getting them deployed and running on a Kubernetes Cluster. First, we'll want to provision this environment.

Getting a Kubernetes cluster up and running, let alone a production-ready one, has not historically been quite as straightforward. While purists (and those learning Kubernetes) might choose to stand up a Kubernetes cluster the hard way – most of us are looking for easy and automated ways to make this happen. There have been a (large) number of projects from the vendor and Kubernetes community in this area, many in various stages of ongoing development.

The Terraform Kubernetes Installer is an open source Terraform template for easily standing up a Kubernetes Cluster on Oracle Cloud Infrastructure (OCI). This allows customers to combine the production-grade container orchestration of Kubernetes with the control, security, and predictable performance of a cloud platform.

WHAT IT DOES

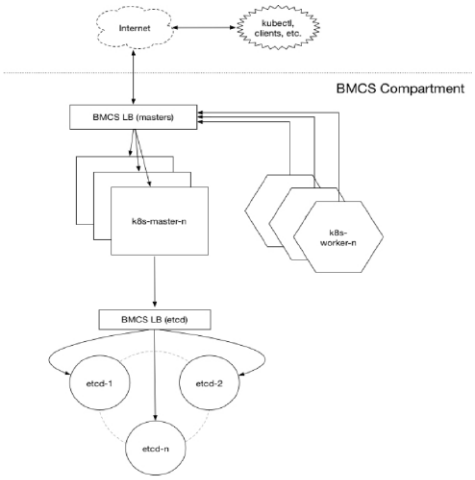
The Terraform Kubernetes Installer provides a set of Terraform modules and sample base configurations to provision and configure a highly available and configurable Kubernetes cluster on Oracle Cloud Infrastructure (OCI). This includes a Virtual Cloud Network (VCN) and subnets, instances for the Kubernetes control plane to run on, and Load Balancers to front-end the etcd and Kubernetes master clusters.

The base configuration supports a number of input variables that allow you to specify the Kubernetes master and node shapes/sizes and how they are placed across the underlying availability domains (ADs).

You can specify Bare Metal shapes (no hypervisor!) in addition to VM shapes to leverage the full power and performance of OCI for your Kubernetes clusters. The nodes are also labeled intelligently, like with the Availability Domain, to support Kubernetes multi-zone deployments so that the Kubernetes scheduler can spread pods across availability domains. You can also add and remove nodes from your cluster using Terraform as documented in the README.

If your requirements extend beyond the base configuration, the modules can also be used to form your own customized configuration.

PROVISION A KUBERNETES CLUSTER VIA CLI



Prerequisites

1. Download and install [Terraform] (terraform.io/) (v0.10.3 or later).
2. Download and install the [OCI Terraform Provider] (github.com/oracle/terraform-provider-oci/releases) (v2.0.0 r later).
3. Create an Terraform configuration file at `~/.terraformrc` that specifies the path to the OCI provider.
4. Ensure you have Kubectl installed if you plan to interact with the cluster locally.

CUSTOMIZE THE CONFIGURATION

Create a *terraform.tfvars* file in the project root that specifies your configuration.

```
# start from the included example
$ cp terraform.example.tfvars terraform.tfvars
```

- Set mandatory OCI input variables relating to your tenancy, user, and compartment.
- Override optional input variables to customize the default configuration.

Deploy the Kubernetes cluster

```
$ terraform init
$ terraform plan
$ terraform apply
```

ACCESS THE CLUSTER

Typically, this takes around 5 minutes after the terraform apply and will vary depending on the overall configuration, instance counts, and shapes. A working kubeconfig can be found in the ./generated folder or generated on the fly using the kubeconfig Terraform output variable.

PROVISION A KUBERNETES CLUSTER AUTOMATICALLY IN A CI / CD PIPELINE

Oracle Developer Cloud Service supports HashiCorp Terraform in the build pipeline to provision Oracle Cloud Infrastructure as part of the build pipeline's automation.

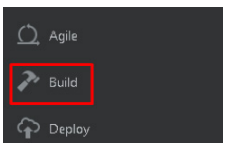
To execute the Terraform scripts that, for example, provision a Kubernetes cluster as part of a CI/CD pipeline, you need to upload the scripts to the Git repository.

PUSHING SCRIPTS TO GIT REPOSITORY ON ORACLE DEVELOPER CLOUD

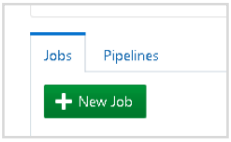
```
Command_prompt:> cd <path to the Terraform script
folder>
Command_prompt:> git init
Command_prompt:> git add –all
Command_prompt:> git commit –m "<some commit
message>"
Command_prompt:> git remote add origin <Developer
cloud Git repository HTTPS URL>
Command_prompt:>  git push origin master
```

HOW TO CREATE A TERRAFORM BUILD JOB

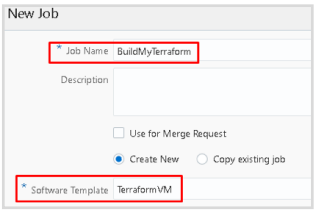
1. Open your Developer cloud Service Project
2. Go to Build



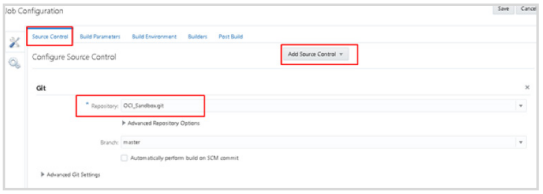
3. Add 'New Job'



4. Give it a name and choose your software template

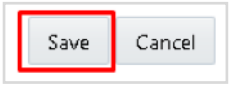


5. Configure your Job
6. In 'Source Control' add your Source Control Git and select your > Terraform Repository

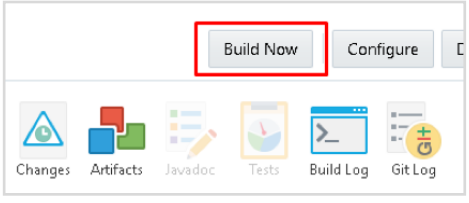


7. Go to the 'Builders' tab and add a 'Builder' Unix Shell Builder. > Here you type your needed commands.

8. Go to the 'Post Build' tab and under 'Post Build Actions' select > 'Artifact Archiver' and type what file(s) you want to add to the > Archive after the build.
9. Now Save it.



10. Now you can run your job.



11. You can use the 'Build Log' to check what is going on and the Compute Service Console to see the output.

DEPLOY CONTAINER ARTIFACTS FROM A REGISTRY TO KUBERNETES (TARGET ENVIRONMENT)

To deploy Docker images to a Kubernetes environment, you'll use the kubectl command line and create scripts that you can then run as part of your CD process.

The most important command is the kubectl create command, which deploys an application from a definition file. You can use the kubectl get commands to monitor the nodes, services, and pods you create. Note that you'll want to have access to the kubeconfig file for the cluster you are working on.

Below you can see an example shell script that was used to deploy a Docker image to a cluster.

```
kubectl get nodes
kubectl create -f nodejs_micro.yaml
sleep 120
kubectl get services nodejsmicroJODU-k8s-service
kubectl get pods
kubectl describe pods
```

In Developer Cloud Service you can define a pipeline which links the build job that builds the Docker image with a second build job that uses a shell script (similar to the one above) to deploy that image to the cluster.

SUMMARY

Containers improve and simplify the Continuous Integration and Continuous Delivery cycle. Combining a cloud runtime platform for Docker and Kubernetes with an end-to-end DevOps automation platform allows you to effectively leverage these technologies, thereby improving your team's development and deployment cycles.

Written by **Shay Shmeltzer** , Director of Product Management, Oracle Cloud Development Tools

Shay Shmeltzer is Director of Product Management for Oracle Cloud Development Tools. He is focused on helping developers simplify and streamline their work leveraging Oracle solutions. Shay frequently presents at industry events, publishes many articles, and regularly blogs at blogs.oracle.com/author/shay-shmeltzer. You can also find him on Twitter at @JDevShay.

Written by **Andrea Morena** , Director of Product Management,Oracle Cloud

Andrea Morena is a Director of Product Management for Oracle Cloud with 25 years' experience working for leading IT & Cloud vendors, including Sun Microsystems, Red Hat, Huawei and Oracle. In his current role at Oracle, he supports Oracle Cloud sales, demand, and adoption providing regional product management activities. He is an expert-CXO level communicator and influencer including Technical and Business Decision Makers. @AndreaMorena5

DZone, Inc.
150 Preston Executive Dr. Cary, NC 27513
888.678.0399 919.678.0300