**W13 – Build Container Image**

| **Criteria** |
| --- |
| **Execute Programming Code Assignment**  Complete Build Container Image Assignment |
| **Interview Readiness**  **What does it mean to create a Docker image and why do we use Docker images?**  A container image is an immutable, static file containing the dependencies for the creation of a container. These dependencies may include a single executable binary file, system libraries, system tools, environment variables, and other required platform settings. Container images result from an application's containerization and are typically stored in container registries, where they can be downloaded and run as an isolated process. Static here means that the container image is not running, it's not being executed, it's only the packaged files and metadata.  Docker is the most used containerization application worldwide.  A container image is a lightweight, standalone and an executable package of software. An ‘image’ includes everything needed to run an application, its code, runtime, the system tools , the system libraries, and the settings.  In contrast to a "container image" that is the stored static contents, a "container" normally refers to the running instance, the thing that is being executed. A container is run from a container image.  Container images become containers at runtime. When using Docker, an image becomes a container when it runs on the Docker Engine.  A container is a standard unit of software that packages up code and all its dependencies. The application runs quickly and reliably in any computing environment.  How are Docker Images created? Through a Dockerfile. A Dockerfile is a text document that contains all the commands a user could call on the command line to assemble an image. So, a Dockerfile is a recipe for building Docker images (a requirement file is needed), running a separate build command produces the Docker image from that recipe. |
| **Interview Readiness**  **Please explain what the difference is between a Container and a Virtual Machine.**  It may be helpful to understand containers as the latest point on the continuum of IT infrastructure automation and abstraction.  In traditional infrastructure, applications run on a physical server and grab all the resources they can get. This leaves you the choice of running multiple applications on a single server and hoping one doesn’t hog resources at the expense of the others or dedicating one server per application, this wastes resources and doesn’t scale.  Virtual machines (VMs) are servers abstracted from the actual computer hardware, enabling you to run multiple VMs on one physical server or a single VM that spans more than one physical server. Each VM runs its own OS instance, and you can isolate each application in its own VM, reducing the chance that applications running on the same underlying physical hardware will impact each other. VMs make better use of resources and are much easier and more cost-effective to scale than traditional infrastructure. VMs are disposable: when you no longer need to run the application, you take down the VM.  Containers take this abstraction to a next level: in addition to sharing the underlying virtualized hardware, they share an underlying, virtualized OS kernel as well. Containers offer the same isolation, scalability, and disposability of VMs, but because they don’t carry the payload of their own OS instance, they’re lighter weight (that is, they take up less space) than VMs. They’re more resource-efficient: they let you run more applications on fewer machines (virtual and physical), with fewer OS instances. Containers are more easily portable across desktop, data center, and cloud environments. And they’re an excellent fit for Agile and DevOps development practices.  The key differentiator between containers and virtual machines is that virtual machines virtualize an entire machine down to the hardware layers and containers only virtualize software layers above the operating system level.  A VM is an emulation of a physical computer. VMs enable teams to run what appear to be multiple machines, with multiple operating systems, on a single computer. VMs interact with physical computers by using lightweight software layers called hypervisors. Hypervisors can separate VMs from one another and allocate processors, memory, and storage among them. VMs are also known as virtual servers, virtual server instances and virtual private servers.  Containers are a lighter-weight, more agile way of handling virtualization; since they don't use a hypervisor, one can enjoy faster resource provisioning and speedier availability of new applications.  Rather than spinning up an entire virtual machine, containerization packages together everything needed to run a single application or microservice (along with runtime libraries they need to run). The container includes all the code, its dependencies and even the operating system itself. This enables applications to run almost anywhere: a desktop computer, a traditional IT infrastructure or the cloud.  Just like virtual machines, containers allow developers to improve CPU and memory utilization of physical machines. Containers go even further, however, because they also enable [microservice](https://www.ibm.com/cloud/learn/microservices) architectures, where application components can be deployed and scaled more granularly. |
| **Interview Readiness**  **What are 5 examples of container orchestration tools (please list tools)?**  As containers proliferated today, an organization might have hundreds or thousands of them, and operations teams needed to schedule and automate container deployment, networking, scalability, and availability. And so, the container orchestration market was born.  Container orchestration software allows developers to deploy multiple containers for implementation within applications. These tools help IT administrators automate the process of running instances, provisioning hosts, and linking containers. These tools assist in optimizing orchestration procedures and extending the lifecycle of applications containing multiple containers. They can also facilitate deployment, identify failed container implementations, and manage application configurations.  According to Cloud Native Computing Foundation’s Cloud Native Landscape, there are more than 109 tools to manage containers, but 89% are using different forms of Kubernetes.  Container orchestration tools:  Kubernetes  Openshift,  Hasicorp Nomad,  Docker Swarm,  Rancher,  Mesos,  Managed container orchestration services:  GKE, Google Kubernetes Engine,  Google Cloud Run,  EKS, AWS Elastic Kubernetes Service,  ECS, AWS EC2 Container Service,  AWS Fargate,  Azurre AKS Service,  Azurre Managed Openshift Service,  Mirantis Kubernetes Engine |
| **Interview Readiness**  **How does a Docker image differ from a Docker container?**  An image is a set of archive files needed to run (including any of its runtime dependencies) containers and its process. You could see it as a form of template on which you can create an unlimited number of containers.  After creating a docker file (Dockerfile) with the instructions for Docker of how to create an image, you need to build the image, this is done using “docker build” command.  A container image is an immutable, static file containing the dependencies for the creation of a container. These dependencies may include a single executable binary file, system libraries, system tools, environment variables, and other required platform settings. Container images result from an application's containerization and are typically stored in container registries, where they can be downloaded and run as an isolated process  After building an image, you will need to run the container, this is done using the “docker run” command.  A container is a running process with resource and capability constraints managed by a computer’s operating system. The files available to the container process are packaged as a container image. Containers run adjacent to each other on the same machine, but typically the operating system prevents the separate container processes from interacting with each other. A Docker container has an exposed end point; it runs on a particular port. The code used to start a container is referred to as a container runtime. |