Docker Executive Overview

Jennifer Morales (C)
Contingent Worker

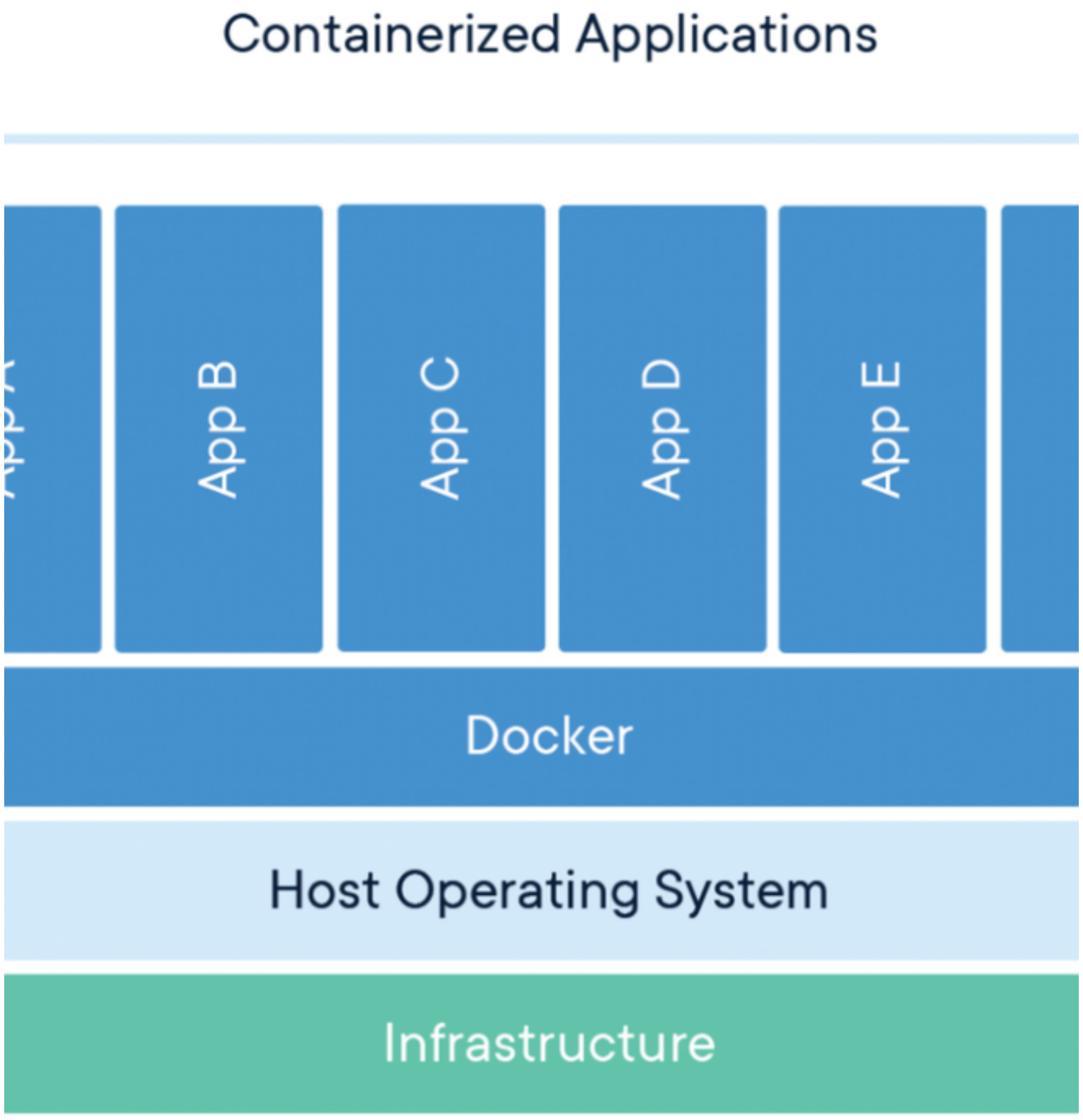
What is It?

Docker is a technology that allows encapsulated applications, web services, and or operating systems to be virtualized and run in very small containers. This is similar to the concept of a Virtual Machine from a functional perspective, but with significant and beneficial key technical differences. As a result of being able to run these components in a small container, Docker allows technical components to be compacted together within less infrastructure and with access to less resources (CPU & Memory). Additionally, smaller overall packages allows developers to more easily access exact copies of running software for development and testing purposes (i.e. no differences between software from an environment perspective).

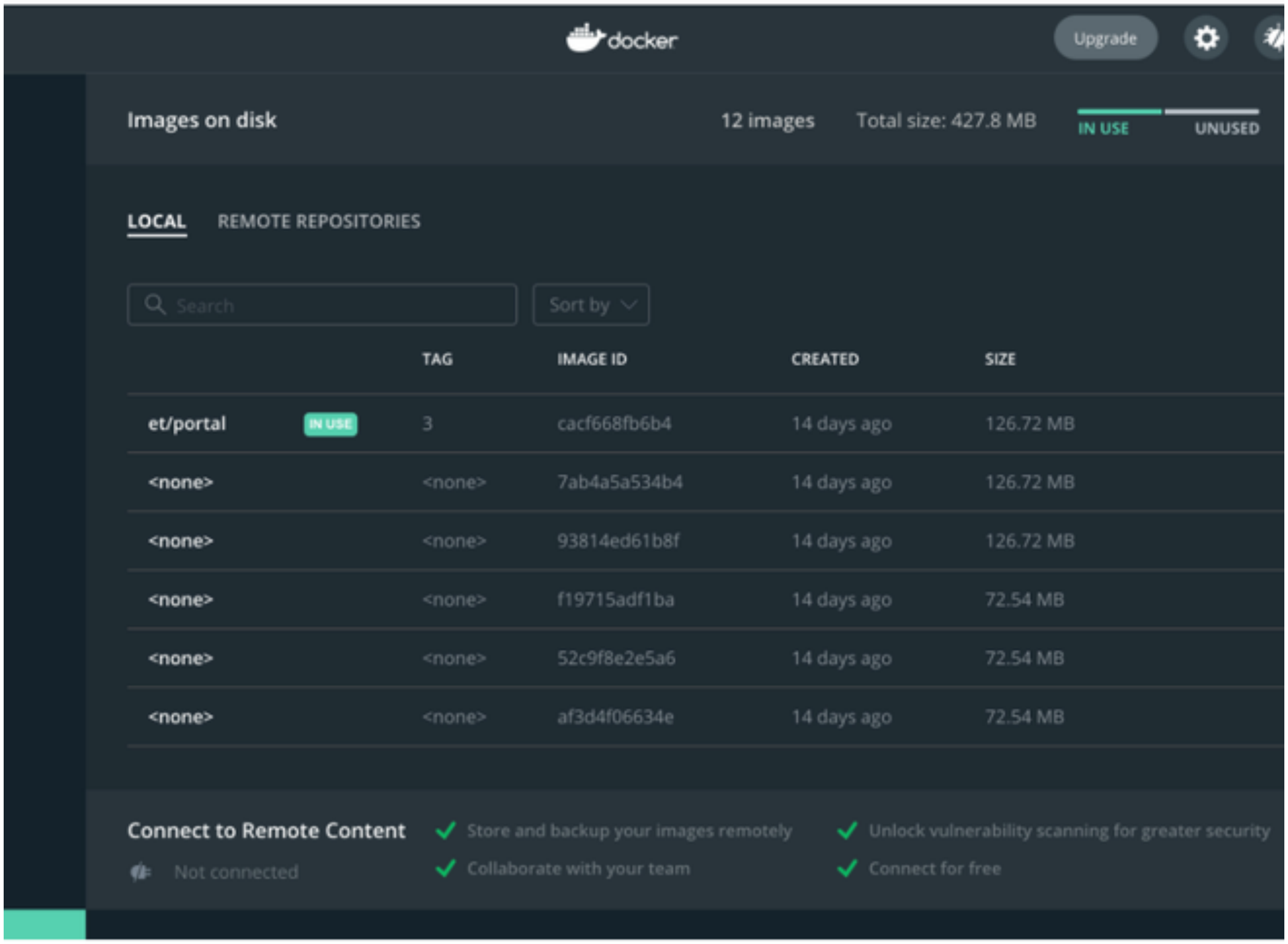
How Does it Work?

Tooling

Docker sits on a host operating system and allows many containers to share resources. This is very different from Virtual Machines which each emulate their own instance of an entire bare metal system including the operating system.



Docker is typically accessed through the Docker Desktop Graphical User Interface (GUI) or a Command Line Interface (CLI).



		TAG	IMAGE ID	CREATED	SIZE
et/portal	IN USE	3	cacf668fb6b4	14 days ago	126.72 MB
<none>		<none>	7ab4a5a534b4	14 days ago	126.72 MB
<none>		<none>	93814ed61b8f	14 days ago	126.72 MB
<none>		<none>	f19715adf1ba	14 days ago	72.54 MB
<none>		<none>	52c9f8e2e5a6	14 days ago	72.54 MB
<none>		<none>	af3d4f06634e	14 days ago	72.54 MB

Connect to Remote Content
Not connected
✓ Store and backup your images remotely
✓ Unlock vulnerability scanning for greater security
✓ Collaborate with your team
✓ Connect for free

To use the CLI, a terminal is opened and the command "docker" is simply used with options. As an example, an entire MongoDB instance can be installed, launched, and exposed for immediate use locally with a single command:

```
1docker run -p 27017:27017 mongo 2
```

In the above example, you may be wondering how Docker knows where to look for the "mongo" image. By default, docker uses the public [Docker Hub](https://hub.docker.com) repository.

In this case, “mongo” referred to the official mongo image stored and versioned there.

Whats a Dockerfile?

- **Dockerfile:** The configuration build file used to tell docker how to structure and setup an image with your application
- **Docker-compose:** One level up from Dockerfile, this is a different kind of build configuration file that orchestrates the build out and execution of multiple Dockerfiles and images. This is often useful during development to spin up a complex interconnected stacks of services and technology with a single command.

Considerations

Docker is by nature ephemeral, meaning that it does not retain any information when shutdown. Another term for this concept is that it is stateless. This can present a challenge for developers who are accustomed to building stateful technologies. Docker requires a different approach to application development and architecture. One convenient reference to list out high level requirements that would be ideal for Docker is that of the 12-Factor Application.

“The twelve-factor app is a methodology for building software-as-a-service apps…”

Registries

When we create Docker images, they must be stored and versioned somewhere that all integrations, developers or customers can access. This is typically referred to as a Docker Registry. There is an established industry standard interface for how Docker interacts with Docker Registries. Specifically, the Docker tooling itself must be able to log into, access, and download images from the registry without any special modification or scripting. This is why it is both unusual and ill-advised for companies to simply zip up images and put them on a server for download. While this is technically possible, it breaks all of the integrations and infrastructure that has been built into the Docker ecosystem, rendering it less valuable and usable. There are a number of Docker Registry options available both SaaS and Self Hosted. Docker Hub, ECR, Quay, Arifactory JFrog and now GitHub are all examples of well known solutions - with Docker Hub as the original and most famous. These systems support both public and private registries.

Public registries provide docker images to anyone who wants them for free. This is similar to the open source model, though it does not expose the source code itself (at least not directly). Typically, software providers use public registries to spread trial or free versions of their software in order to then allow cross-sell to more lucrative licenses. Remember that Docker images are ephemeral, so they are rarely pre-populated with data in a public registry. This makes it possible to put the software out there for free but enforce a profitable usage pattern with licensed data access.

Private registries require a login to see or download Docker images. These are often used by the internal development teams of an organization so they can benefit from the Docker ecosystem without exposing in-progress solutions before they are ready.

As stated above, private registries are primarily used as internal tooling; however, it is possible to use a private Docker registry to distribute containers to customers and avoid making the software publicly available. This can be done by using custom accounts or SSO to allow customers access to the private repository. It should be noted that this is not a common practice and key safeguards may be missing. For example, most private repositories are a single pool of images, meaning that once a customer has access to the registry, they can see all the images in that registry. In some solutions it is possible to limit permissions to the images present - meaning that while a customer can see an image, they may not be able to download the image. One workaround is to use a distinct private repo per customer; however, this can be difficult to scale as it requires individual configuration per customer. This also can create complexity within any automated pipeline that customer or Vertex has defined to access the images, which is a key part of the Docker ecosystem value proposition. Nevertheless, solutions are possible for this approach.

Security

Docker in and of itself is generally a secure solution; however, the overall ecosystem does afford opportunities for security issues. Specifically, the fact that public registries exist with public docker images that anyone can download means that you don't always know what is coming along for the ride in your container. Thankfully, this potential issue is easily mitigated through simple steps:

- When possible, create and use your own base images
- When using public images, ensure they are from trusted providers - in Docker Hub for example, this is indicated with an "official image" flag
- Implement container security scanning functionality as part of your registry, or use a registry that provides this feature
- Ensure your orchestration solution is monitoring ports and deployments for unusual activity while feeding that information to a dashboard for observation and where possible alerts

Docker itself has vulnerability scanning functionality available. Additionally, a large number of vendors and software providers exist that offer enhanced security features, scanning and reporting as well. Often these solutions are built into the registry.

Limitations

As interesting an innovation Docker has been, there are key limitations that should be understood.

Docker does not itself provide commercial functionality to an existing product. While from a distribution perspective, "Dockerization" may be desired or even requested by a customer, this process enables automated infrastructure and deployment, it does not itself offer any new functionality. Ideally, a “dockerized” solution should function exactly the same as a “non-dockerized” solution from an experience perspective.

Docker has technology stack limitations. Originally, Docker was built for the linux OS ecosystem, both from a development perspective and an image runtime perspective. Today, all major development platforms (Mac, Windows, & Linux) are supported for the creation and maintenance of Docker images (the development perspective). From an image runtime perspective, by far the operating system of choice is linux. There are Windows containers and you can run windows applications in them; however, even Microsoft is moving away from this container approach. Windows containers require a lot of supporting components which swell the size of the overall image back to VM levels, removing many of the benefits that would otherwise be available. With the release of .NET Core and official Microsoft Linux distributions, it is far more common and beneficial to focus on technologies that ultimately run within Linux containers.

Today, Docker does not natively, nor through any industry standard practice, run on mobile operating systems such as iOS or Android. While it is true the iOS is part of the Unix/Linux family tree and Android a descendent of Linux, these solutions are not designed to explicitly run Docker. Mobile operating systems have a variety of tools that allow web specific code such as html, javascript, css, java, and others to be compiled to native versions of those applications that the OS can manage. This is very different than simply running a Docker container. There are 4 major reasons why:

1. Docker is a backend server/infrastructure technology while mobile applications are client side technologies. It might be technically possible to house Docker inside an Android device through some form of technical wizardry/hack; however, it would effectively be a lone server without any of the common interfaces that make servers usable such as http based APIs. Mobile operating systems are not designed to expose ports and services the same way a server in AWS or Azure might.
2. iOS is explicitly designed to stop applications from interacting with the same ease that they may on a locally controlled server. Only recently has

an API been released that even lets applications interact; and this uses a different interface standard than normal web technologies.

3. Mobile devices are not networked such that a local UI can be generated and served on the browser, which is what you would need to be able to do in order to access a website hosted locally via Docker on a device for example.
4. Docker has no established standard protocol, interface, or methodology to render content to the native operating system of a mobile device.

Docker Vs. VM

Here are some primary differences from a use perspective:

Category	Docker	Virtual Machine
Size	Docker images and containers are typical very small, though they can be large if necessary. Advances in this technology also allow for the concept of micro-containers which may be 10MB or less. More typically you will see containers in the 100mb to 250mb range.	Virtual Machines are by comparison massive, typically requiring Gigabytes.
Startup Time	Because Docker shares resources (something we will discuss in the next section), containerized runtimes can start very quickly, with the only limiting factor being the software having been	Virtual Machines mimic bare metal systems and as such must go through the full OS startup before even attempting to start the

	containerized. All of the virtual "hardware" is already operational.	underlying application or service. This means they are much slower to start.
Resource Sharing	Docker sits on top of an existing hardware and operating system (the host) and shares resources within this host. This allows different containers to access data, networking, and other elements if desired, or to be securely isolated if preferred. Because Docker shares these host resources between containers, it does not require copies of them for each container, leading to the small size and fast startup times.	Conversely, Virtual Machines do not share resources. They can be mapped together or networked together as you would machines on any network, but doing so is complicated and requires the coordination of distinct isolated components (the VMs). This also means VMs are large and slow to start relatively speaking.
		Virtual Machines are

Orchestration & Infrastructure	Docker can be run as a standalone system on a machine (virtual or otherwise), or it can be run as part of an orchestration system such as Kubernetes, Mesos, Rancher, or ECS. Orchestration systems are designed to full leverage the size and speed of Docker, compacting many containers into as little a space as possible, automatically scaling up or down the containers, managing routing, managing logging/tracing, handling security, and more. Kubernetes has become the leading industry adopted solution for Docker orchestration.	"orchestrated" using more traditional solutions within cloud infrastructure. They use distinct services and components such as VPCs, load balancers, mounted volumes, terraform and other cloud offerings to define the functionality of auto scaled, routed, and networked systems within a broader cloud environment that is at the end of the day, a series of individual running virtual machines. This is more time consuming and difficult to	

		maintain than the out of box solutions provided by Docker Orchestration Software.
Productivity	Docker allows developers to become significantly more productive once they understand the tools available to them. Docker is now available on Windows, Mac, and Linux. It is easy to install and work with. It allows developers to quickly spin up entire stacks of operating systems and services all locally and networked with a few commands. Developers can also quickly package software versions to share for testing and deployment without concern about how it will be compiled or with what settings, as these are all part of the container.	Virtual Machines are a useful tool but require specialized software, licenses, and often higher end hardware to match the same productivity & scale offered by Docker. As discussed above, VMs are also slower in terms of spinning up and down systems, something that happens frequently in development. This is why VMs have never really taken off as a consistent

		tool for the development process itself.

What is the Competition Doing With It?

Docker and associated orchestration technologies are a signifiant component of all modern Cloud and SaaS technologies. Mature and fully realized container strategies allow SaaS infrastructure costs to reduce dramatically, both increasing the overall profit margin while also allow for more competitive pricing. While we can not audit competitor development environments directly, it is inconceivable that they would not all be somewhere along the path of fully containerized infrastructure. Companies like Vertex that have a significant legacy software footprint will be slower to move toward the benefits that containerization provides, and we can use that knowledge to understand where some of our competitors may be on the journey.

One simple metric to show this investment is to look at the frequency with which experience and skills associated to Docker are listed in competitor job listings. A simple google search of any competitor followed by docker (e.g. ‘Avalara Docker’) will show that almost all technical positions now list as requirement Docker and Kubernetes experience.

What is the Potential Intersection With Tax?

Docker itself does not intersect with tax or specific tax market opportunities. It does however provide opportunities to think about new ways to make Tax or Tax Adjacent software available to customers while reducing the overall costs associated with developing and maintaining software. In doing so, Docker forces us to ask questions around where the value of the software comes from, for example:

- Is the software code the valuable part, or the possible data that can be run. Do I really need to license both?
- How should customers get access to our software? Does it really require a license to download?

- What does it cost to run our current cloud infrastructure, what might it cost if we converted fully to containers?
- If we create an Edge docker solution which runs on our customer's infrastructure, who has ultimate accesses and responsibility for maintenance of that running image?
- etc...

Next Steps

- Continue the investment internally to transition technology toward containers in order to streamline R&D costs
- Continue to investigate new deployment and partnership opportunities that are made possible via Docker

Other Links

- Official Docker Site
- AWS Overview