Docker is an open-source system of software containers (shifted to open source in the first part of 2013). Its popularity grew steadily and exploded in 2014. By the middle of 2015, more than 300 million container downloads had occurred (according to Docker’s home page). The project has been “starred” more than 25,000 times on Github, forked more than 6,500 times, and received 1,000 contributions.

Docker is not a virtualization technology, it’s an application delivery technology.

To understand Docker, we need to understand containers

Shared Operating Systems

Containers vs. Virtual Machines

Docker is so popular that some industry analysts are talking about possibly doing away with VMs altogether in favor of container technologies such as Docker and LXC. However, using Docker or not depends on the specific needs of a particular project. Docker is not always the answer.

Containers work best when a specific application to handle an individual task is needed. VMs have more of a macro view and can run on most modern operating systems. Containers are the answer in some situations while VMs are the answer in others — and, in some cases, both should be deployed at the same time. Cloud architects need to understand when and where each type of deployment is best for their organizations.

Other complaints about Docker include:

Builds and deployments are slow and unpredictable

Some Docker supporters advocate the putting of data into containers, a practice that would make it difficult to backup or clone the data

Docker is not developer-friendly

There is a high learning curve for Linux novices

It has a touchy and unpredictable command-line interface

The security is weaker than VMs

When a technology like Docker skyrockets in popularity, there is a tendency to want to use it in every potential use case. In reality, Docker should be used only when it is the best solution to the specific problem at hand.

Docker shares a lot of the host operating system resources. It uses a layered filesystem (AuFS; advanced multi layered unification filesystem) and manages networking.

AuFS is a layered file system, so you can have a read only part and a write part which are merged together. One could have the common parts of the operating system as read only (and shared amongst all of your containers) and then give each container its own mount for writing.

So, let's say you have a 1 GB container image; if you wanted to use a full VM, you would need to have 1 GB times x number of VMs you want. With Docker and AuFS you can share the bulk of the 1 GB between all the containers and if you have 1000 containers you still might only have a little over 1 GB of space for the containers OS (assuming they are all running the same OS image).

A full virtualized system gets its own set of resources allocated to it, and does minimal sharing. You get more isolation, but it is much heavier (requires more resources). With Docker you get less isolation, but the containers are lightweight (require fewer resources). So you could easily run thousands of containers on a host, and it won't even blink.

There are pros and cons for each type of virtualized system. If you want full isolation with guaranteed resources, a full VM is the way to go. If you just want to isolate processes from each other and want to run a ton of them on a reasonably sized host, then Docker/LXC/runC seems to be the way to go.

Why is deploying software to a docker image (if that's the right term) easier than simply deploying to a consistent production environment?

Deploying a consistent production environment is easier said than done. Even if you use tools like Chef and Puppet, there are always OS updates and other things that change between hosts and environments.

Docker gives you the ability to snapshot the OS into a shared image, and makes it easy to deploy on other Docker hosts. Locally, dev, qa, prod, etc.: all the same image. Sure you can do this with other tools, but not nearly as easily or fast.

This is great for testing; let's say you have thousands of tests that need to connect to a database, and each test needs a pristine copy of the database and will make changes to the data. The classic approach to this is to reset the database after every test either with custom code or with tools like Flyway - this can be very time-consuming and means that tests must be run serially. However, with Docker you could create an image of your database and run up one instance per test, and then run all the tests in parallel since you know they will all be running against the same snapshot of the database. Since the tests are running in parallel and in Docker containers they could run all on the same box at the same time and should finish much faster. Try doing that with a full VM.

From comments...

Interesting! I suppose I'm still confused by the notion of "snapshot[ting] the OS". How does one do that without, well, making an image of the OS?

Well, let's see if I can explain. You start with a base image, and then make your changes, and commit those changes using docker, and it creates an image. This image contains only the differences from the base. When you want to run your image, you also need the base, and it layers your image on top of the base using a layered file system: as mentioned above, Docker uses AUFS. AUFS merges the different layers together and you get what you want; you just need to run it. You can keep adding more and more images (layers) and it will continue to only save the diffs. Since Docker typically builds on top of ready-made images from a registry, you rarely have to "snapshot" the whole OS yourself.