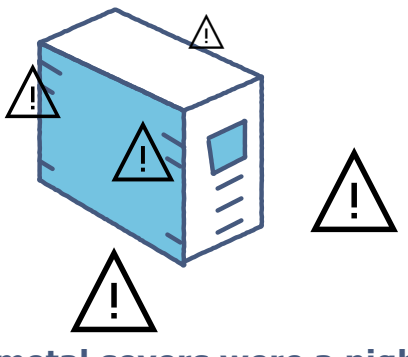
A Short History of Deployment Processes

In this lesson, we will peek into the past and go through a short history of deployment processes.

WE'LL COVER THE FOLLOWING

The Beginning

In the beginning, there were no package managers. There were no JAR, WAR, RPM, DEB, and other package formats. Package managers typically maintain a database of software dependencies and version information to prevent software mismatches and missing prerequisites. The best we could do at that time was to zip files that form a release. More likely, we'd manually copy files from one place to another. When this practice is combined with bare-metal servers which were intended to last forever, the result was living hell. After some time, no one knew what was installed on the servers. Constant overwrites, reconfigurations, package installations, and mutable types of actions resulted in unstable, unreliable, and undocumented software running on top of countless OS patches.



Bare metal severs were a nightmare

Configuration Management Tools

The emergence of configuration management tools (e.g., CFEngine, Chef, Puppet, and so on) helped to decrease the mess. Still, they improved OS setups and maintenance, more than deployments of new releases. They were never designed to do that even though the companies behind them quickly realized that it would be financially beneficial to extend their scope.

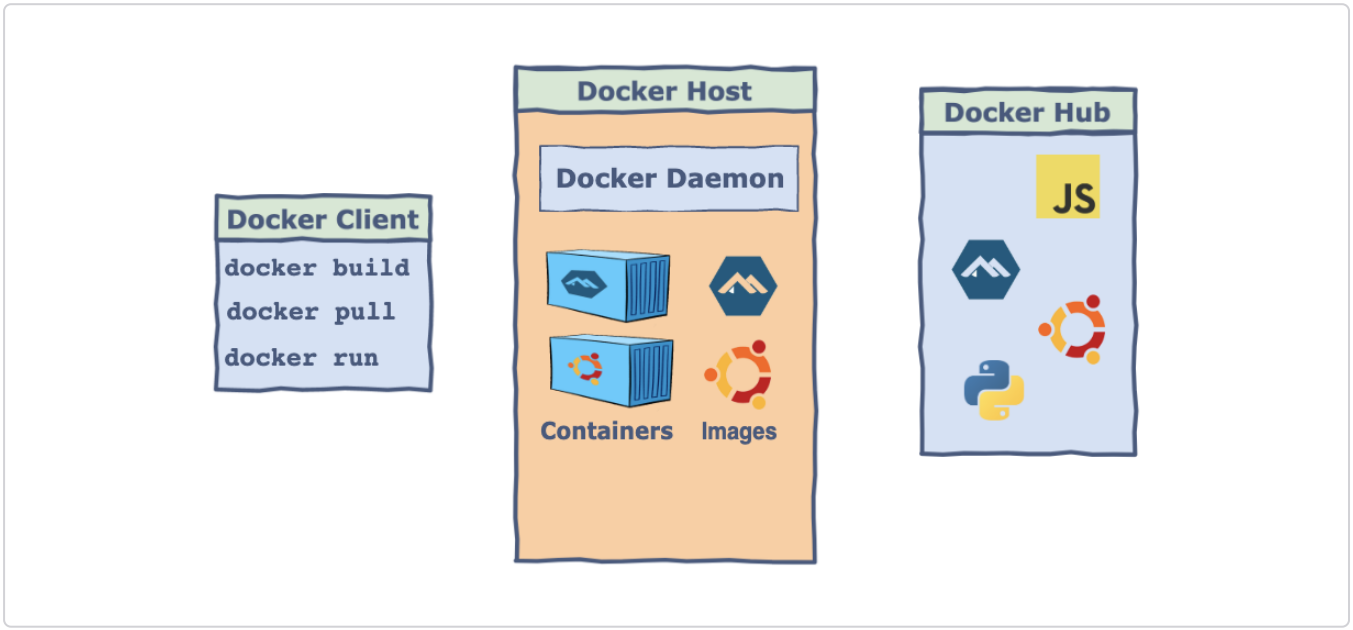
Even with configuration management tools, the problems with having multiple services running on the same server persisted. Different services might have different needs, and sometimes those needs clash. One might need JDK6 and the other JDK7. A new release of the first one might require JDK to be upgraded to a new version, but that might affect some other service on the same server. Conflicts and operational complexity were so common that many companies would choose to standardize. As we discussed, standardization is an innovation killer. The more we standardize, the less room there is for coming up with better solutions. Even if that's not a problem, standardization with clear isolation means that it is very complicated to upgrade something. Effects could be unforeseen and the sheer work involved to upgrade everything at once is so significant that many choose not to upgrade for a long time (if ever). Many end up stuck with old stacks for a long time.

The Need of the Hour

We needed process isolation that does not require a separate VM for each service. At the same time, we had to come up with an immutable way to deploy software. Mutability was distracting us from our goal to have reliable environments. With the emergence of virtual machines, immutability became feasible. Instead of deploying releases by doing updates at runtime, we could create new VMs with not only OS and patches but also our own software baked in. Each time we wanted to release something, we could create a new image, and instantiate as many VMs as we need. We could do immutable rolling updates. Still, not many of us did that. It was too expensive, both regarding resources as well as time. The process was too long. Even if that would not matter, having a separate VM for each service would result in too much unused CPU and memory.

Docker and Containers

Fortunately, Linux got namespaces, cgroups, and other things that are together known as containers. They were lightweight, fast, and cheap. They provided process isolation and quite a few other benefits. Unfortunately, they were not easy to use. Even though they've been around for a while, only a handful of companies had the know-how required for their beneficial utilization. We had to wait for Docker to emerge to make containers easy to use and thus accessible to all.

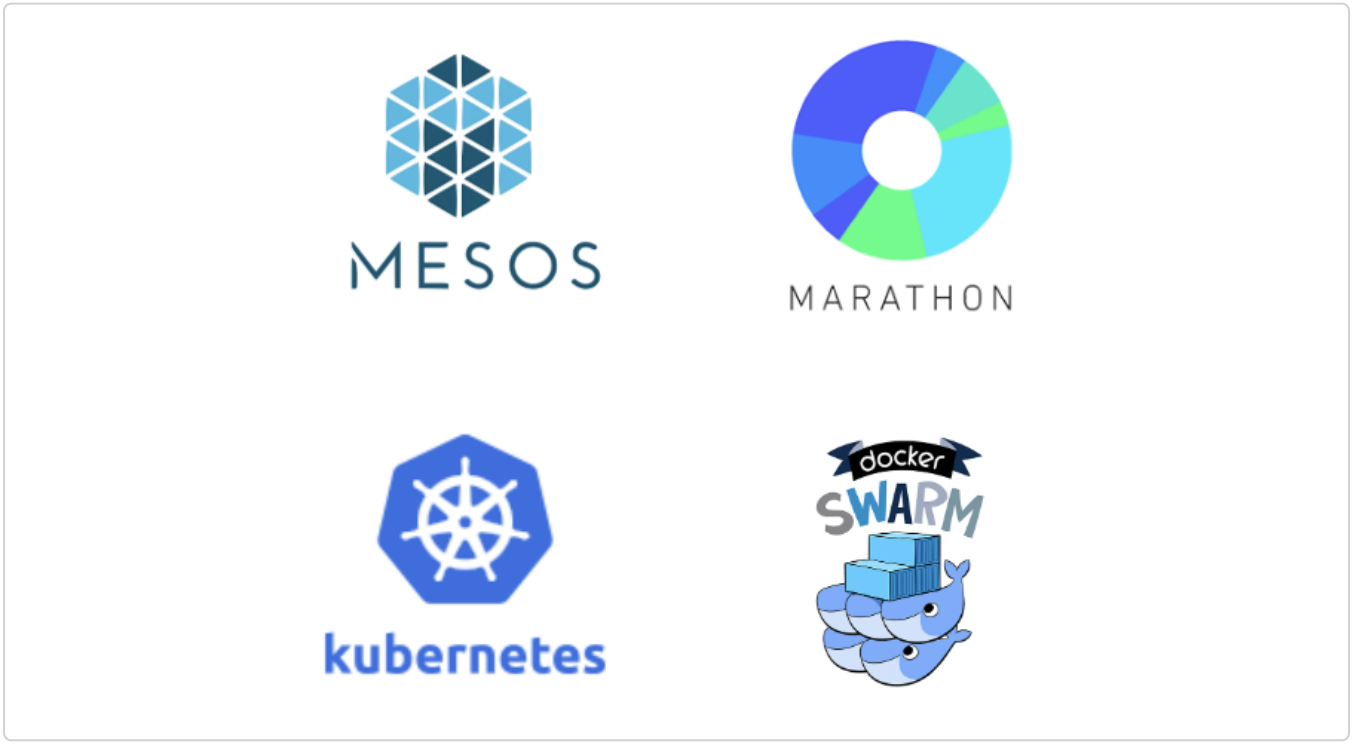


Today, **containers** are the preferable way to package and deploy services. They are the answer to immutability we were so desperately trying to

They are the answer to immutability we were so desperately trying to implement. They provide necessary isolation of processes, optimized resource utilization, and quite a few other benefits. And yet, we already realized that we need much more.

Why Container Schedulers?

It's not enough to run containers. We need to be able to scale them, to make them fault tolerant, to provide transparent communication across a cluster, and many other things. Containers are only a low-level piece of the puzzle. The real benefits are obtained with tools that sit on top of containers. Those tools are today known as container schedulers. They are our interface. We do not manage containers, they do.



In case you are not already using one of the container schedulers, you might be wondering what they are.

In the next lesson, we will get familiar with the container schedulers.