History of Containers and Its Impact on Software Development and Deployment

Ibrahim Abdelkareem - 2023-05-08

Containerization technologies transformed how software applications are developed, deployed, and managed. Containers provide significant value to developers, enabling them to build, test, and deploy software efficiently. The history of containers goes back to the 1960s when virtualization was first introduced at IBM. This essay discusses the history of containers before Docker and how it became one of the most important technologies for DevOps. I'll also discuss the impact of containers, Docker and other containeraization technologies on the software development industry, including container orchestration and Kubernetes.

The Beginning

In the 1960s, IBM introduced virtualization technology in its mainframe systems to enable multiple applications to run on a single system. This technology was a useful for developers as it allowed them to run multiple applications on a single system, which reduced the cost of hardware and increased the efficiency of the system.

IBM achieved that via time-sharing where every application is given a slice of time on the system's CPU to execute its tasks before the CPU move on to the next application in the queue. This technology has its limitations, and applications had to be designed with time-sharing in mind. These limitations pushed IBM to invent virtual machines (VMs) as we know it today, which allows multiple isolated environments, each with its own operating system and resources, to run on a single physical machine. Virtual machines was a breakthrough for software development at that time as it allowed far greater utilization of physical resources.

chroot and Moving Past Virtualization

In the 1970s, Unix introduced the `chroot` command that changes the root directory for a process and its child processes which allows a process to run in an isolated environment with a restricted view of the file system. `chroot` provided a basic level of isolation for processes but had limitations. e.g., Processes running in a `chroot` could still access the

host system's resources which is a security concern, and there was no way to limit their resource usage.

Even with these limitations and its basic level of isolation, `chroot` gives an advantage over virtual machines, which requires developers to spin a new virtual machine with its own OS if they want to run a process in isolation.

Addressing chroot Limitations with BSD Jails

In late 1990s, a new technology called BSD Jails came to light in order to address the limitations with chroot. BSD Jails is built upon the chroot concept and extends it with features. e.g., resource, network and user isolation so that multiple isolated environments could co-live on a single host, each with its own IP address, network stack, and file system. This made it possible to run multiple applications on a single system without interfering with each other.

Solaris Zones

After the success the BDS Jails brought, there still some challenges, e.g., how can we create a network of multiple jails? How can we limit the resources utilized by jails? That's why in 2004 Sun Microsystems introduced Solaris Zones, which builds upon the concept of BSD Jails by providing a more advanced and robust solution for isolating processes and applications.

Some of its key features are the support for virtual networking so that multiple zones can communicate independently from the outside world and the support for resource management that enables developers to allocate a specific amount of CPU time, memory, disk space, and network bandwidth, which helps ensure that the resources of one zone do not interfere with those of another, and live migration feature which allows running zones to be migrated from one physical host to another without downtime. This feature is very useful, especially for load balancing and disaster recovery scenarios.

Open VZ

Following the success of Solaris Zones, Virtuozzo announced OpenVZ in 2005 which is a similar technology to Solaris Zones which enables multiple isolated environments to run on a single host. However, there are some differences between the two technologies.

One of the main differences between OpenVZ and Solaris Zones is the isolation level they provide. Solaris Zones provides complete isolation at the kernel level, where each zone has

its own instance of the Solaris kernel. This provides high security and reliability, but the downside is that you must use it with the Solaris Operating System. OpenVZ, on the other hand, uses a shared kernel model, where all containers on a host share the same kernel, which provides less isolation, but makes it far more flexible as it allows it to run on a wider range of Linux distributions.

OpenVZ also manages resource differently than Solaris Zones, as it uses a hierarchical resource management system that allows resources to be allocated to groups of containers, rather than individual containers which provides better flexibility and control over resource allocation.

OpenVZ also supports checkpointing and live migration, similar to Solaris Zones. Checkpointing allows a running container to be saved to disk, while live migration enables a running container to be moved from one host to another without downtime.

Linux Containers

The idea of containers started to get more attention with the introduction of Linux Containers (LXC) in 2008. LXC is an open-source technology that's similar to OpenVZ and Solaris Zones, it provides a lightweight virtualization solution that enables applications to be isolated from each other and the host operating system with a user-friendly cli to create, manage, and monitor containers. It's also compatible with wide range of Linux Distributions, as developers can use it with any Linux distribution that supports cgroups and namespaces. That brings us to one of the key features of LXC which is its support for multiple containerization technologies as it uses Linux Kernel Features such as control groups or namespaces, or a combination of both.

Control groups (cgroups) is a kernel feature that enables fine-grained control over system resources e.g., CPU, memory, and I/O. While namespaces, provide a way to isolate containers at the process level, by creating separate namespaces for each container's process tree, network stack, and file system.

In addition to that, like Solaris Zones & OpenVZ. LXC supports live migration, which allows running containers to be moved from one host to another without downtime.

The Emergence of Docker

In 2013, Docker was born. Initially it was built on top of LXC but later switched to its own containerization backend called libcontainer it quickly gained popularity among developers

and DevOps due to its ease of use, portability, and support for container images. It also provided a developer-friendly and intuitive cli to make it easy to work with.

Container images allowed developers to write code to specify a blueprint of what should be included in a container when it gets created. The container image is used later on any machine with Docker installed to create and run a container or to be a base image to another image (e.g., image that will run ubuntu used as a base for another image for ubuntu with java preinstalled). Container images enabled developers of packaging container imagesin the CI/CD to later share and distribute it across different environment, usually via systems called Container Registeries (e.g., Azure Container Registery, AWS Elastic Container Registery, …etc.).

Docker images consist of layers. Each layer is a specific set of changes or additions to the base image. This layered architecture enables Docker images to be lightweight and efficient, with only the necessary changes being stored and shared.

Another key feature of Docker was its support for container orchestration, which enabled multiple containers to be managed and scaled together as a single unit. Docker provided its own orchestration tools, e.g., Docker Compose and Docker Swarm, to enable developers to define complex application deployment strategies and execute them easily.

All what's mentioned above are reasons why Docker gained popularity and adoption quickly compared to its predecessors. While each technology contributed to having containerization as we know it today, Docker provided the most developer friendly and easy way of working with containers.

Docker innovations also set the first stone for more technologies to rise such as container orchestration (e.g., Mesos, Yarn and Kubernetes), serverless technologies, and more.

Containerd

As mentioned in the previous section, docker initially relied on LXC to run containers and then switched to its implementation that's called libcontainer. This didn't last forever as Docker later developed Containerd to replace libcontainer, which is a lightweight and flexible runtime that can run containers in different environments while providing a consistent API for managing containers.

Docker donated containerd to the Cloud Native Computing Foundation (CNCF) in 2017. The CNCF is an open-source organization focused on developing and promoting cloud-native technologies. The CNCF is part of the Linux Foundation, and its members include many of the world's largest technology companies.

By donating containerd to the CNCF, Docker ensured that containerd would continue to be developed and supported as an open-source project and that it'll be used as a foundational technology in the broader ecosystem of cloud-native technologies e.g., Kubernetes, Amazon Elastic Container Service for Kubernetes (EKS), Istio, Podman, …etc.

Despite the existence of other container runtimes, Containerd nowadays can be considered the standard of how to run containers because it provides a common interface for managing containers, which makes it easier for different tools and platforms to work together. This common interface is known as the Container Runtime Interface (CRI), which is used by Kubernetes and other container orchestration platforms to manage containers. Also, containerd is highly modular and can be used in a wide variety of environments. It can run on many operating systems, including Linux, Windows, and macOS, and it can be used with many different container image formats, including Docker images, OCI images, and others.

The CNCF has also developed several other containerization-related projects, including the Container Networking Interface (CNI) and the Container Storage Interface (CSI), which help to standardize networking and storage for containers.

Container orchestration

While containers are helpful, they're challenging to manage in complex and large-scale deployments. The difficulty of managing container grow as the number of deployed containers grows. This difficulity is due to many reasons such as:

- **Resource Management:** Containers are often deployed across multiple hosts, which makes it difficult to manage resources effectively at the organization level.
- **Scalability:** Containers need to be deployed, scaled up and down based on load and demand quickly and efficiently. High Availability: Containerized applications need to be highly available and withstand failures of individual containers, hosts, or entire data centers.
- **Networking:** Containers need to be connected to each other and to the outside world, and managing networking can be a complex task.
- **Security:** Containers can be vulnerable to security threats, especially when they're deployed across multiple hosts or in public cloud environments. All these reasons sparked the need for tools to simplify and automate managing containers in complex and large-scale deployments.

Container orchestration tools provide a framework for automating many of the tasks for containers management to enable DevOps of managing realworld high-scale applications,

including:

- Automate the process of deploying containers across multiple hosts, making it easy to distribute containers across a cluster of machines.
- Enable containers to communicate with each other and with the outside world, and can help ensure that network traffic is routed efficiently.
- Providing a unified platform for managing containers, virtual machines, and other types of resources.
- Load balancing by distributing network traffic automatically across a cluster of containers.
- Automate the process of scaling containers up and down in response to changes in demand, and ensure that applications are always running at optimal capacity.
- Automatic failover and self-healing ensure that containerized applications remain available even in the face of hardware or software failures.

Apache Mesos

In 2011, Apache Mesos, one of the earliest container orchestration tools. Mesos provided a unified platform for managing containerized and non-containerized workloads, and can scale to manage thousands of nodes in a single cluster.

Apache Mesos abstracts resources like CPU, memory, and storage away from physical machines and provides them as a shared pool to applications running on top of it. It uses a scheduler-based architecture, which allows it to handle the resource allocation across a large number of machines to tasks in a highly efficient and flexible manner. Mesos has a master node which schedules tasks and slave nodes which executes these tasks.

Kubernetes

In 2014, Google released Kubernetes, which quickly became the dominant container orchestration tool. Kubernetes was built on top of Google’s internal container orchestration system, Borg. Kubernetes provides a unified platform for deploying, scaling, and managing containerized applications across a cluster of machines.

Kubernetes builds on the ideas of previous container orchestration systems, such as Apache Mesos and Docker Swarm, but takes them to the next level with a more comprehensive and extensible architecture, a wider range of features, and a strong focus on automation and ease of use.

Kubernetes is a distributed system of worker nodes that are responsible for running containers and executing workloads. Each node runs a container runtime, such as Docker or containerd, and communicates with the Kubernetes control plane to receive instructions, report status, and perform other tasks.

The Kubernetes control plane consists of several components, including the Kubernetes API server, etcd (distributed key-value database), the Kubernetes scheduler, and the Kubernetes controller manager. These components work together to manage the state of the cluster, schedule workloads onto nodes, monitor and repair nodes and containers, and expose the Kubernetes API to external clients.

Kubernetes has a declarative model, which allows developers to specify the desired state of their applications and infrastructure in a YAML or JSON file called a manifest. Kubernetes then uses this manifest to create and manage the necessary resources, such as pods, services, and deployments, to ensure that the actual state of the system matches the desired state.

Due to all the advantages it brings Kubernetes became the standard for container orchestration and is part of the CNCF.