ADCAIJ: Advances in Distributed Computing and Artificial Intelligence Journal
Regular Issue, Vol. 11 N. 4 (2022), 395-409
eISSN: 2255-2863
DOI: https://doi.org/10.14201/adcaij.28351

# Containerization and its Architectures: A Study

Satya Bhushan Verma[a], Brijesh Pandey[b], and Bineet Kumar Gupta[C]

[a,c] Shri Ramswaroop Memorial University, Barabanki, India
[b] Goel Institute of Technology & Management, Lucknow, India
[a] satyabverma1@gmail.com, [b] brijesh84academics@gmail.com, [c] bkguptacs@gmail.com

KEYWORDS

*Cloud Computing; storage; containerization; Docker; LXC*

ABSTRACT

*Containerization is a technique for the lightweight virtualization of programmes in cloud computing, which leads to the widespread use of cloud computing. It has a positive impact on both the development and deployment of software. Containers can be divided into two groups based on their setup. The Application Container and the System Container are two types of containers. A container is a user-space that is contained within another container, while a system container is a user-space that is contained within another container. This study compares and contrasts several container architectures and their organisation in micro-hosting environments for containers.*

## 1. Introduction

Containerization is a method for the lightweight virtualization of programs, which contributes to the widespread adoption of cloud computing. As described in C. Pahl et al., orchestrating and deploying containers separately and in groups has been a significant issue (C. Pahl 2015). Numerous studies on container knowledge in the cloud have been conducted to detect trends, knowledge gaps, and future directions. The studies provide a comparative picture of the status of research by identifying, classifying, and synthesizing the available data. The systematic mapping study (SMS) was used in this research to help define and structure new topics of inquiry.

Both the development and deployment processes benefit from the use of containers. For example, cloud architecture is evolving toward DevOps approaches, allowing for a continuous advancement and distribution pipeline based on containers and orchestration that incorporates cloud-native architectural results (Brunnert et al., 2015).

According to the findings, containers may offer continuous progress in the cloud when employed with cloud-native platform progress and distribution facilities, however, they require sophisticated orchestration provision. Thus, orchestration methods based on containers emerge as a tool for orchestrating compute in cloud-based, grouped systems. The findings of this research indicate that such methods are realized to strike an equilibrium between the necessity for reasonable quality control, e.g., optimal source usage and performance, which is a price issue in the cloud (due to its utility assessing code).

This research article aims to assist researchers working in software engineering, dispersed arrangements, and cloud computing. A systematic research presentation establishes a knowledge foundation on which to build theories and explanations, evaluate study consequences, and define future scopes. Additionally, it helps practitioners who engage in this activity by using existing tools and technology.

As virtualization enables the virtualization of computer, storage, and networking components, it is the foundation of Cloud computing. Through the use of a specialized software layer known as a hypervisor, virtualization enables the creation of several separate execution environments known as Virtual Machines (VM) that can be easily moved from one Cloud to another. KVM, Xen, WM-Ware, Virtual Box, and Virtual PC are some hypervisor virtualization software solutions. Linux Container Virtualization has recently emerged as a lightweight alternative to HVV (LCV). This technique divides the actual resources of a computer into several separate user-space instances. Docker, Kubernetes, OpenVZ, and Virtuozzo are just a few examples of LCV software solutions.

The primary distinction between HVV and LCV is that HVV abstracts the visitor operating systems, while LCV bonds a single OS kernel across containers. In a nutshell, HVV establishes a computer hardware concept layer, while LCV utilizes system calls. Containers and virtual machines are equivalent from the user's perspective. Additionally, since LCV consumes fewer hardware resources than HVV, it enables more containers on a single physical host than VMs.

## 1.1. Container Configurations

Containers are available in two distinct configurations.

**1.1.1. Application Container:** An individual container is purpose-built to execute a single kind of application;

**1.1.2. System Container:** A user-space is contained inside another container. LCV enables greater adaptability in terms of service design, optimization, and administration.

The following section compares HVV with LCV.

- Start-Up-Time. For the start-up phase, LCV is quicker than HVV.
- Dynamic-Runtime Control. LCV enables the host OS to start or stop a containerized program, while HVV usually needs the host OS to establish a virtual network/serial connection.
- Speed. Only experiments conducted under particular conditions are capable of determining the latency and throughput overhead.
- Isolation. In comparison to LCV, HVV has a high level of isolation.
- Flash Memory Consumption. LCV enables the OS core and other portions of the user's environment, while HVV does not allow sharing due to separated VM images.
- Assigning or separating resources dynamically. Both solutions are capable of implementing this functionality.
- Direct computer hardware access. In contrast to HVV, LCV needs specialized drivers in the host OS core to entree the device's computer peripheral. We conclude that, on the basis of this study, LCV triumphs over HVV.

## 1.2. Responsibilities for Resource Supervision on the Internet of Things-Cloud

IoT Cloud service providers can employ both HVV and LCV models to deliver their services. The Internet of Things devices, which run custom software, are connected to a Cloud stage that can manage heterogeneous identifying data from several sensors. Due of its limited computational capability, an IoT device typically only runs primitive programmes for detecting and actuating. A Cloud data centre, on the other hand, conducts heavy computational tasks such as storing and processing sensor data. Several projects have looked into LCV-based applications for IoT devices to take advantage of both HVV and LCV benefits. Figure 1 depicts an IoT Cloud that makes use of both HVV and LCV knowledge. The structure of the Internet of Things is in charge of the following:

- Instantiation;
- Guaranteeing consistency, QoS, safety, and scalability;
- Supervision and optimization of IoTaaS.

Additionally, since IoT devices include LCV capabilities, IoTaaS may be deployed in many dispersed containers. The information collected from IoT gadgets is standardized and saved on the Cloud stage, activating actuators linked to IoT gadgets. HVV and LCV create a film of abstraction that conceals all corporal proficiencies. Typically, IoT cloud workers provide both patterns, but only LCV is utilized in IoT devices. To dynamically offer services to their customers, the operators of IoT clouds

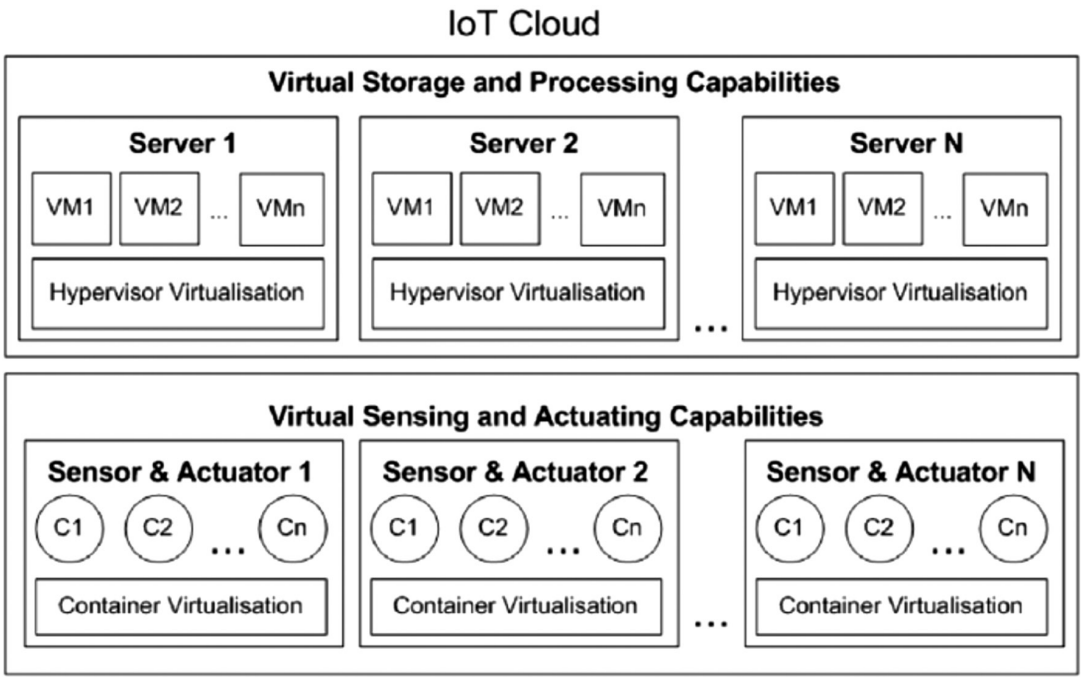


*Figure. 1. A demonstration of an Internet of Things cloud that makes use of both HVV and LCV virtualization know-how (Celesti et al. 2019).*

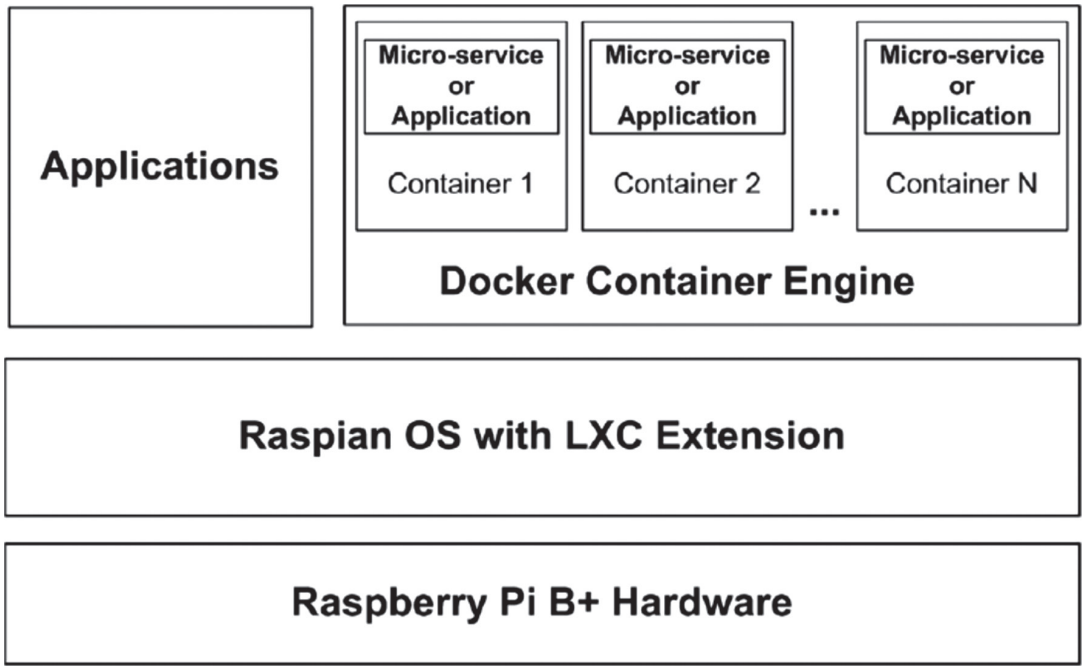


*Figure. 2. Container virtualization software planning based on Docker and running on a Raspberry Pi B+ board Antonio (Celesti et al. 2019).*

install a variety of containers and virtual machines (VMs) in their infrastructure. It enables the worker to reorganize, improve, and move its computer-generated resources. The operator of the IoT Cloud may fulfil any service allocation request made by its customers by using this infrastructure. IoT Clouds may execute various activities due to LCV functional to SBCs (Single Board Computers).

- Deployment of Dispersed IoTaaS. It is probable to configure dispersed IoTaaS by combining containers installed on a variety of IoT gadgets.
- IoTaaS Repositioning and Optimization. An IoT Cloud may move a container from one IoT gadget to another in order to implement load balancing techniques;
- IoTaaS Consolidation. Using a location-aware IoTaaS constructed by merging the containers deployed on various IoT devices makes it feasible to impose software consolidation methods based on the application logic. This opens up new revenue streams for IoT Cloud workers in the areas of cost-effective asset movement and optimization, energy conservation, and on-demand resource supply.

## 2. Container Architectures and their Organization

Virtualization is used in cloud computing to provide elasticity in large-scale shared resources. At the infrastructure layer, virtual machines (VMs) usually serve as the backbone. By contrast, containerization enables lightweight virtualization by customizing containers as application correspondences

from separate images (often obtained from an image source) that use fewer properties and time. Additionally, they enable more interoperable application packing, which is required for cloud-based moveable and interoperable package applications. Containerization is built to create, test, and deploy programs over a distributed network of computers and link these containers. Containers, therefore, solve Cloud PaaS-related issues. Given the Cloud's overall significance, it is critical to have a consolidated picture of ongoing operations.

## 2.1. Basis of Container Technology

Container stores that are packed, self-reliant, ready-to-distribute requests and, if essential, middleware and commercial logic (in the form of binaries and libraries), are required to execute applications. Container-based apparatuses such as Docker are based on container machines, which serve as moveable containers for programs. As a consequence, in multi-tier systems, it is necessary to handle container dependencies. In a tiered plan, an orchestration plan may specify components, their needs, and their lifetime. After that, a PaaS cloud may execute the processes through mediators (such as a container appliance). As a result, PaaS clouds may enable the placement of container-based applications. Their coordinated development, deployment, and continuing administration are referred to as orchestration in this context.

Numerous container systems are built on the Linux LXC framework. Current Linux versions, part of the Linux container scheme LXC, have core features such as namespaces and groups that allow

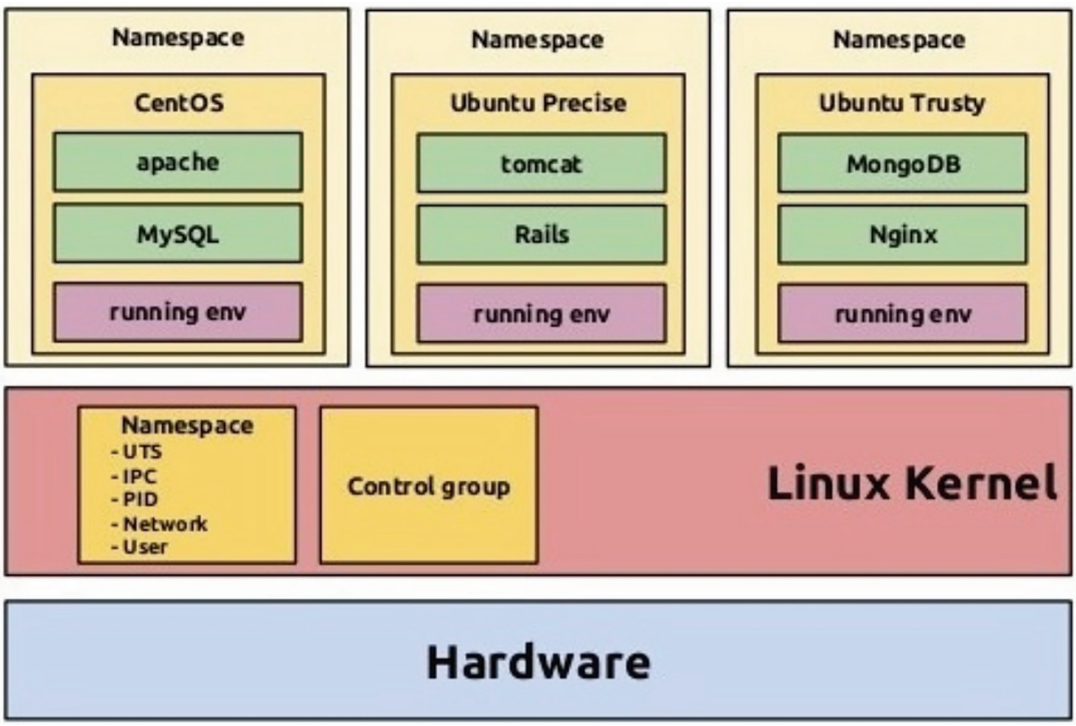


*Figure 3: Linux Containers or LXC Containers*

processes to be isolated on a joint operating system [S5]. Docker is now the maximum common container explanation and has been used to demonstrate containerization. A Docker copy comprises layered file systems analogous to the Linux virtualization stack's LXC instruments; Docker employs a combination mount to add a writeable file structure on top of the read-only file structure. This enables the coexistence of several read-only file systems. This feature enables the creation of new pictures by layering existing ones. Only the container's top layer is editable.

Containerization enables the transition from standalone containerized apps to clusters of container hosts capable of running containerized applications across cluster hosts. Containers' inherent compatibility facilitates the latter. Clusters of linked container hosts are formed from individual container hosts. Each cluster is comprised of numerous nodes (hosts). Application facilities are reasonable clusters of containers belonging to the identical image. Scaling an application over many host nodes is enabled via application services. Volumes are used to provide persistence methods in applications that need them. These volumes may be mounted for storage in containers. The use of links enables the connection and communication of two or more containers. Orchestration provision for inter-container statement, connections, and provision assembly is required to deploy and manage these container clusters (S. Soltesz et al. 2007).

## 2.2. Cloud-Based Container Designs

Container orchestration entails more than just starting, stopping, and shifting applications (i.e., containers) among servers. The process of constructing and continuously sustaining groupings of container-based computer program applications that may be geographically scattered is known as orchestration. Container orchestration allows customers to specify how the containers in the Cloud will be coordinated when they install a multi-container request. Container orchestration is the process of deploying and maintaining containers as a single entity. It makes sure that containers are always available, scalable, and connected. Container construction in the cloud is simply a form of orchestration within a distributed cloud environment. Cloud architecture can be conceived of as a layered and scattered structure, with core infrastructure, platform, and software application layers spread across several cloud environments (Antonio Celesti et al. 2019). Container technologies have the potential to assist. As such, container technologies are critical in the imminent application administration, particularly in the context of Cloud PaaS. Container-based architectures, which have recently gained popularity, may be implemented in this cloud platform. Given this shift in architectural style, secondary research may assist practitioners in selecting the best technology.

# 3. Related Works

Virtualization of resources usually entails installing a software layer on top of the host operating system to manage numerous resources. These virtual machines (VMs) may be thought of as their execution environment. Numerous techniques are used in the process of virtualization (M. Xavier et al 2013). Hypervisor-based virtualization is one common approach. KVM and VMware are two well-known virtualization systems based on hypervisors. A virtual machine monitor must be placed on top of the underlying physical system to take advantage of this technology. Additionally, each virtual machine supports (separate) guest operating systems. This virtualization strategy is feasible for a single host operating system to sustain many visitor operating systems (M. Xavier et al. 2013).

A different approach is represented by container-based virtualization. The hardware resources are partitioned in this approach by implementing many (safe) isolations. Guest processes acquire abstractions instantly using container-based technologies since they function directly at the operating system (OS) level via the virtualization layer.Typically, in container-based systems, one OS core is joint across virtual occurrences. Users perceive containers as autonomous operating systems capable of operating hardware and software self-sufficiently (S. Soltesz et al. 2007).

According to Biederman (E.W. Biederman, 2006), the isolation feature of containers is handled by kernel namespaces. This is a Linux kernel distinctive property that enables developments to achieve the required abstraction stages. Although containers do not communicate external with a namespace layer, there is a separation between the Host OS and Invitee Processes, and an individual container has its operating system. Biederman (E.W. Biederman, 2006) states that namespaces separate file systems, process IDs, networks, and inter-process communication. However, container-based virtualization solutions impose restrictions on resource usage according to process groups.

To be more specific, in container-based virtualization, c groups are responsible for allocating priority to CPU, memory, and I/O usage. However, some technologies that make use of containers consistently manage their resources using cgroups. By using container-based solutions, it is possible to dynamically deploy and consume microservices in packaged hosting settings. Microservice patterns are not a novel concept in the world of software architecture. They are now generally acknowledged as a cost-effective method of developing applications. Before the microservices design, the standard approach to service development was to mainly build monolithic systems. From a practical standpoint, this necessitates the creation of a unified platform capable of managing everything. Many of these problems may be resolved in Cloud settings by using scripting methods that support Infrastructure-as-a-Service (IaaS), Platform-as-a-Service (PaaS), and Software-as-a-Service (SaaS) (SaaS). Such explanations, however, face difficulties when several ad hoc plans and groups are engaged, each with its different software and information requirements, as is the case when community-specific virtual apparatus images are not accessible for the Cloud. Lightweight Cloud-based solutions are advantageous in these settings. Microservices are an example of this kind of architecture.

The microservice architecture is based on the idea of "split and conquer". Micro-services fundamentally substitute a single big code base with smaller code basics managed by minor, agile teams. These codebases communicate through a sole API. The advantages of this approach are that each group may operate independently and be protected/disconnected from one another—a process referred to as exemption cycles. However, in certain instances, such as when services from various organizations are interdependent, these exemption cycles may be linked. There is currently a plethora of container-based micro hosting providers. Docker, CoreOS, and LXC are the most well-known of them. Docker simplifies the process of encapsulating an application in a container, along with any necessary accessories for operation.

This is accomplished using a focused collection of apparatuses and combined Application Programming Interface Guidance knowhows that include kernel-level structures, such as Linux containers, control clusters, and a copy-on-write file arrangement. Assembling a container's file arrangement, Docker relies on the Advanced Multi-Layer Unification File System (AMULFS) (AuFS). AuFS supports the exact superimposition of one or more accessible file systems. It allows Docker to make use of the images necessary for the container's basis. For example, a user may use one Ubuntu image as the basis for many more containers. Docker makes use of a single copy of Ubuntu by using the Advanced Multi-Layer Unification File Scheme. This significantly lowers the amount of storage used and the amount of RAM used by containers due to their fast launch. AuFS also offers one additional advantage: it supports image

versioning. Each new edition is tagged as diff, a file ordering feature that indicates the variance between two records. This enables the reduction of the amount of picture files, for example.

Another advantage of AuFS is that it allows to track the changes made to image versions—similar to how software expansion code versioning schemes work. CoreOS is a very new Linux delivery that was created to function tons of software structures. This technique enables a slimmed-down Linux core intending to minimize outlays to the utmost. Additionally, CoreOS provides capabilities for ensuring redundancy and safeguarding against system failure. CoreOS has announced an original rocket container runtime (rkt), which implements the submission container standard. This technology's primary objective is to create a customized container model.

Additionally, Amazon Machine Images are supported by this method. The rocket container runtime is an alternative to Docker, including enhanced safety and other requirements for server-based production. The rocket container runtime conforms to the Submission Container standard, which defines a new collection of container presentations that make them readily transportable or moveable. Docker executes each process through a daemon, which does not offer the same level of assurance as Rocket in terms of security. To address this issue, some have proposed that Docker be entirely rebuilt.

LXC is integrated into the standard Linux kernel, and the project allows for the management of container and operating system images via instrumentation. Containers may be thought of as lightweight operating systems that run alongside the host operating system. Because containers do not emulate the hardware film, they may run near-native speeds without additional overhead performance. In their normal operation, apps and web loads are created and installed in a specific configuration on bare-metal servers for challenging reasons. For instance, simultaneous installation and configuration of PHP, MySQL, Nginx, and Drupal are possible. Currently, however, programs are installed on a specific computer and cannot be easily moved. A virtual computer may be installed and relocated, and although this provides the illusion of portability, performance suffers as a result. The LXC container provides near-bare-metal performance and the flexibility to move easily across systems by encapsulating comparable stacks in containers. The increased speed and flexibility of an LXC container provide the appearance of a dedicated server. Clones, backups, and snapshots of containers are possible. LXC simplifies container management and adds new levels of freedom to the execution and deployment of applications. Flockport enables the rapid deployment of web stacks and applications packaged in LXC containers on any Linux-based computer. Flexibility and throughput are directly supported by LXC containers, while Flockport is a mechanism for distributing and using LXC containers.

Numerous academics have focused on narrowing the performance and optimization gap between virtualization and non-virtualized methods. Typically, these studies employ a unique approach and a set of instruments. Additionally, the compositions of the methods that have been studied and compared differ. Recent research articles, for example, analyze and compare the opening between inherent systems and their envisaged counterparts and the presentation disparities between container- and hypervisor-based virtualization methods. We acknowledge that this is a rapidly evolving area, and as a result, some of the older articles use out-of-date tools. Additionally, contemporary visualization technology has not been studied. In a recent paper, Hwang et al. evaluated four distinct hypervisor-based virtualization systems (J. Hwang, S. Zeng, T. Wood 2013). The authors did not find a hypervisor that performed much better. As a result, they recommended that customers use various software and hardware stages in Cloud services to meet their needs.

Abdellatief et al. (E. Abdellatief, N. Abdelbaki 2013) compared the presentation of VMware, Microsoft Hyper-V, and Citrix Xen in various situations. The authors utilized a modified SQL instance technique to conduct the assessment. They created millions of items, consumers, and orders using this technique.

(S. Varrette et al. 2013) conducted a similar study using different technologies and testbed environments. The authors conducted tests with high-speed computing using kernel-based computer-generated machines instead of Microsoft Hyper-V. Their research examined power usage, energy efficiency, and scalability.

Although the evidence for virtualization overheads was inconsistent, (S. Varrette et al. 2013) established that the virtualization film for virtually every hypervisor substantially impacted the presentation of virtualized systems, particularly for high-performance computation spheres. Current articles compare and contrast hypervisors with container methods. According to Dua et al. (R. Dua, A.R. Raja, D. Kakadia, 2014), Containers are gaining popularity to enable PaaS facilities. Estrada et al. benchmarked both KVM and Xen-based virtualization technologies, notably KVM, Xen, and LXC (Z. Estrada, et al. 2014). Their primary approach was to compare and contrast the runtime performance of each of the technologies they listed. Their tests and benchmarks were motivated by the goal of facilitating the development of series-based applications.

Heroku, a platform-as-a-service company pioneered containers to professionally and repeatably organize apps. Rather than treating the container as a computer-generated server, Heroku defines it as a procedure with enhanced separation characteristics. As a result, container-based application deployment provides a low-overhead solution with almost the same isolation as virtual machines. As with normal processes, it also includes resource sharing capabilities. Google's infrastructure largely relied on such containers. Additionally, Docker provided a standardized image format for application containers and administration tools for them.

# 4. Comparison of Micro-hosting Environments

Each container-based knowledge comes with a particular set of advantages and disadvantages. This section discusses some of the main structures of the leading container-based knowledge and their potential effect on performance. While CoreOs Rocket was mentioned in the preceding segment, no presentation assessment was performed since CoreOS has not yet published an official edition of their produce.

## 4.1. Docker

Docker uses numerous Linux kernel capabilities to enable the isolation of containers (Docker 2021).

### 4.1.1. Namespaces

Containers are deployed using Docker namespaces. Docker makes use of a variety of different kinds of namespaces to accomplish its job of isolating containers (M. Xavier et al. 2013):

- Docker containers are based on pid, which guarantees that no process in one container may influence processes in another;
- It makes use of the network to accomplish system interfaces, or more specifically, to isolate the system's networking resources;
- ipc is used to isolate particular inter-process communication (IPC) properties, including System V IPC things and POSIX communication queues. This implies that each IPC namespace has its set of properties for interprocess communication.

Docker uses the mnt namespace to provide processes with their view of a file structure and its mount facts.

- Throughout, identifiers for the kernel and version are isolated.

### 4.1.2. Control Groups

Docker uses cgroups to enable prevailing containers to segment presented hardware properties. It is conceivable that some of these resources may be limited in their usage at any moment. With Docker, some arduous tasks associated with low-level OS virtualization are abstracted away, saving users from enduring these tasks. According to Docker documentation, it aims to create an environment that is similar to that of the application's running environment no matter where it continues to run.Despite different hardware configurations, the software behaves exactly the same. The possibility of working with the software under its original intended conditions is perhaps even more appealing to developers since most unforeseen issues are not present under a different operating system. As a result, the operations team can ship the software to the final destination and expect it to behave similarly if the software runs properly in the container.

- The union file scheme is a kind of file scheme that works by creating films that serve as the foundation for containers.
- The container format may be thought of as a container that encompasses all of the preceding methods.

## 4.2. LXC

Linux Container is a container-based virtualization solution that allows for the rapid development of lightweight Linux containers via a standardized and extensible API and related implementations (Linux Containers 2021). By contrast, Docker is a container-based application development platform. They have several characteristics but also have a great deal in common. To begin, LXC is a method for virtualizing the operating system at the level of individual Linux containers on a sole LXC host. It does not utilize a computer-generated machine but rather creates a computer-generated environment with its CPU, memory, blocked I/O, system, and resource management system. This is accomplished through the LXC host's Linux kernel's namespaces and cgroups capabilities. As a chroot, but with much greater separation. LXC supports a broad range of virtual network and device types. Second, Docker uses fixed layers to facilitate the reuse of well-structured architectures, albeit at the expense of complexity and performance. Constraints on the number of applications per container limit the potential for usage. LXC allows the formation of single- or multi-threaded programs.

Additionally, LXC enables the creation of numerous system containers, which may be replicas of a single sub-volume that can be accessed through a btrfs file. This LXC capability enables it to address complex file system-level problems. Thirdly, LXC provides a comprehensive set of tools and rights for both container design and execution. In conclusion, LXC allows for the construction of poor containers, ensuring that non-root operators may figure containers. This functionality is currently unavailable in Docker.

## 4.3. CoreOS rocket

Rocket (Zhanibek Kozhirbayev and Richard O. Sinnott 2017) is a container-based technology developed by CoreOS as a substitute for Docker. Both Rocket and Docker automate application deployment via virtual containers that may run autonomously depending on the host's features. While Docker

has evolved into a complex environment capable of supporting various needs and activities, Rocket is designed to execute basic tasks while maintaining a safe environment for application deployment. Rocket is a command-line tool that allows to execute application containers, descriptions of graphic designs. Rocket focuses on CoreOS's application container standard, a collection of descriptions that enables easy migration of containers. As Polvi acknowledged, Rocket may be tougher to use than Docker, as Docker streamlines the complete process of container construction through its informative boundary.

# 5. Evaluation parameters

There are many angles from which technologies may be compared, most notably in terms of performance. To assess container-based technologies from an overhead viewpoint, it was essential to comprehend (measure) the outlays associated with non-virtualized settings. The study in this section examined various performance metrics, including CPU and memory performance, system bandwidth and inexpression, and storage outlays. Multiple experiments were performed 15 times in each benchmarking phase to determine the precision and reliability of the different findings. Timing data was collected on an average and standard deviation basis (Zhanibek Kozhirbayev and Richard O. Sinnott 2017).

- CPU performance: The first scenario to evaluate the CPU performance was based on use of a compressor.
- Disk I/O performance: A key aspect of performance is the evaluation of disk I/O performance, specifically, volumes given as a non-ephemeral storage were attached to instances.
- Memory performance: The evaluation of Memory I/O performance is presented, The benchmark tool used to test the microhosting environments was the STREAM software (STREAM 2021). STREAM assesses memory throughput utilizing straightforward vector kernel procedures.
- Network I/O performance: The evaluation of Network I/O performance is presented.

# 6. Container Designs and their Management

The cloud enables the resistance of large-scale collective resources via the use of virtualization methods (P. Mell and T. Grance 2011). At the infrastructure layer, virtual machines (VMs) usually serve as the backbone. By comparison, containerization enables lightweight virtualization by customizing containers as application correspondences from separate images (often obtained from an image source) that use fewer possessions and time. Additionally, they enable more interoperable claim packaging, which is required for cloud-based transferable and interoperable package applications (C. Pahl 2015). Containerization is predicated on creating, testing, and deploying applications across many servers and linking these containers. Containers, therefore, solve cloud PaaS-related issues. Given the cloud's overall significance, it is critical to have a consolidated picture of ongoing operations.

## 6.1. Container Knowledge Principles

Container stores are packed, self-sufficient, ready-to-disperse components of applications and, if required, middleware and commercial logic (in the form of binaries and reference library), which are required to execute applications. Container-based apparatuses such as Docker are based on container

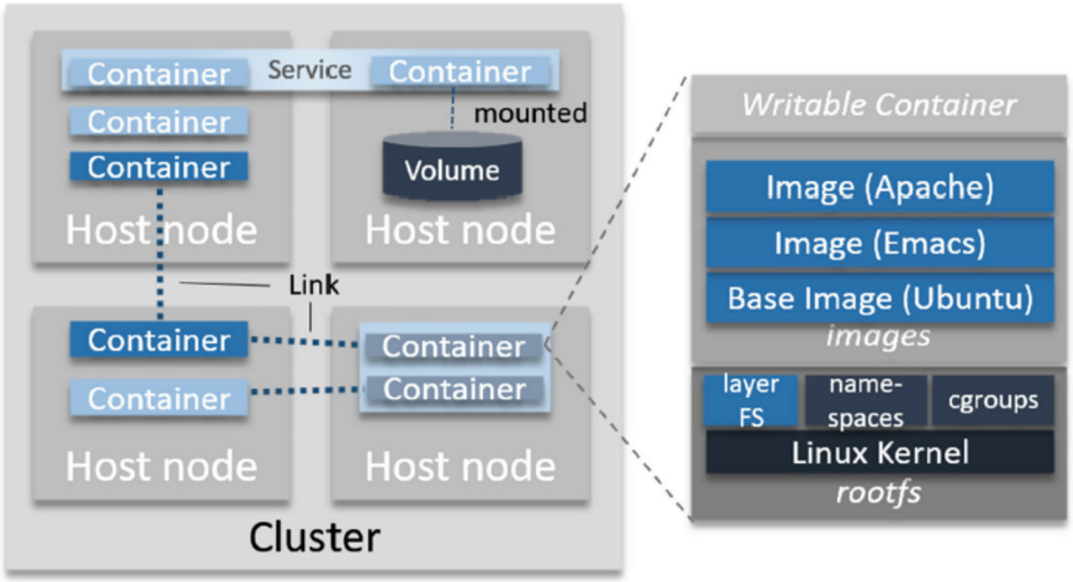


*Figure 4: Container Cluster Architectures, (Claus Pahl, Antonio Brogi, Jacopo Soldani, and Pooyan Jamshidi 2019)*

appliances, which serve as transferable containers for programs. As a consequence, in multi-tier systems, it is necessary to handle container dependencies. In a tiered plan, an orchestration plan may specify components, their needs, and their lifetime. After that, a PaaS cloud may execute the plan's processes through mediators (such as a container appliance). As a result, PaaS clouds may enable the positioning of container-based applications. Their coordinated development, deployment, and continuing administration are referred to as orchestration in this context.

Numerous container systems are built on the Linux LXC framework. Current Linux versions, a portion of the Linux container scheme LXC, have kernel features such as namespaces and cgroups that allow processes to be isolated on a collective operating system [S5]. Docker is now the utmost common container explanation and has been used to demonstrate containerization. A Docker image comprises tiered file systems analogous to the Linux virtualization stack, which utilizes the LXC instruments, as shown in Figure 3. Docker enhances a writeable file arrangement on top of the read-only file system via a union mount. This enables the coexistence of several read-only file arrangements. This feature enables the creation of new pictures by layering existing ones. Only the container's top layer is editable. Containerization enables the transition from single containerized apps to clusters of container hosts capable of running containerized applications across cluster hosts. Containers' inherent compatibility facilitates the latter. Individual container hosts are organized in clusters, as shown in Fig. 3. Individual clusters are comprised of numerous nodes (hosts). Application facilities are reasonable clusters of containers belonging to the identical image. Scaling an application over many host nodes is enabled via application services.

Volumes are used to provide persistence methods in applications that need them. These volumes may be mounted for storage in containers. The use of links enables the connection and communication of two or more containers. Orchestration provision for inter-container statements, links, and facility assemblies is required (P. Mell and T. Grance 2011).

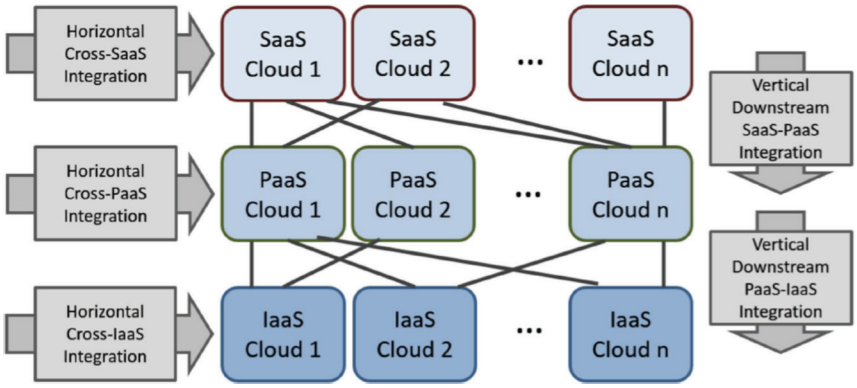


*Figure 5: Cloud reference Architecture Model (Claus Pahl, Antonio Brogi, Jacopo Soldani, and Pooyan Jamshidi 2019)*

## 6.2. Cloud-Based Container Designs

Container orchestration encompasses more than just starting and stopping programs (i.e., containers) and relocating them across servers. Orchestration is defined as the process of establishing and constantly maintaining clusters of container-based package applications that may be geographically dispersed. When users install a multi-container claim, container orchestration enables them to specify how the containers in the cloud are to be coordinated. Container orchestration encompasses container deployment and the administration of many containers as a single entity. It ensures that containers are available, scalable, and networked. Cloud-based container building is essentially a method of orchestration inside a distributed cloud atmosphere. The cloud may be thought of as dispersed and tiered, as shown in Figure 2, with core infrastructure, stage, and application layers spread over many cloud environments (A. Brogi, et al. 2016). Container technologies have the potential to assist. As such, container technologies will be critical in the imminent application administration, particularly in cloud PaaS. Containers enable this cloud framework to implement recent and popular microservice-based architectures (N. Kratzke 2015, J. Lewis and M. Fowler, 2014). Given this shift in architectural style, secondary research may assist practitioners in selecting the best technology.

# 7. Conclusion

Containerization is a lightweight virtualization of any program in cloud computing, which contributes to the extensive acceptance of cloud computing. It provides many advantages for both the development and deployment processes. Containers are categorized in two distinct configurations. Application Container and system container, the Application Container is a purpose-built individual container for the execution of a single kind of application, and system container is a user-space contained inside another container. In this paper, various container architectures and their organization, have been described. This paper has also presented a comparison of micro-hosting environments of containers.

# 8. References


Abdellatief, E., Abdelbaki, N. 2013, Performance evaluation and comparison of the top market virtualization hypervisors, in: 2013 8th International Conference on Computer Engineering & Systems, (ICCES), IEEE.

Biederman E.W., 2006, Multiple instances of the global linux namespaces, in: Proceedings of the Linux.

Brogi A. et al. 2016, SeaClouds: An Open Reference Architecture for Multi-Cloud Governance. Cham, Switzerland: Springer, pp. 334–338.

Celesti A. et al. 2019, A study on container virtualization for guarantee quality of service in Cloud-of-Things, Future Generation Computer Systems 99, pp. 356–364

Docker 2021, Build, Ship, and Run Any App, Anywhere, viewed 1 September 2021, https://www.docker.com.

Dua, R., Raja, A.R. and Kakadia, D. 2014, Virtualization vs containerization to support PaaS, in: 2014 IEEE International Conference on Cloud Engineering, (IC2E), IEEE.

Estrada, Z. et al. 2014, A performance evaluation of sequence alignment software in virtualized environments, in: 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, (CCGrid), IEEE, 2014.

Hwang, J., Zeng S. and Wood, T. 2013, A component-based performance comparison of four hypervisors, in: IFIP/IEEE International Symposium on Integrated Network Management, (IM 2013), IEEE, 2013.

Kratzke, N. 2015, "About microservices, containers and their underestimated impact on network performance" in Proc. 6th Int. Conf. Cloud Computing, pp. 165–169.

Lewis J. and Fowler, M. 2014. Microservices, http://martinfowler.com/articles/microservices.html

Linux Containers 2021, viewed 1 September 2021, URL http://linuxcontainers.org.Xavier M. et al. 2013, Performance evaluation of container-based virtualization for high performance computing environments, in: 21st Euromicro International Conference on Parallel, Distributed and Network-Based Processing, (PDP), IEEE, 2013.

Liu, C., Loo, B. T. and Mao Y. 2011, "Declarative automated cloud resource orchestration" in Proc. 2nd ACM Symp. Cloud Comput., Art. no. 26.

Mell P. and Grance, T. 2011, "The NIST definition of cloud computing" Recommendations Nat. Inst. Standards Technol., Special Publication, 800-145, http://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-145.pdf

Pahl C. 2015, "Containerization and the PaaS Cloud" IEEE Cloud Comput., vol. 2, no. 3, pp. 24–31.

Pahl, C., Brogi, A., Soldani, J. and Jamshidi, P. 2019, Cloud Container Technologies: A State-of-the-Art Review, IEEE Transactions on Cloud Computing, Vol. 7, No. 3.

Soltesz, S. et al. 2007, Container-based operating system virtualization: a scalable, high-performance alternative tohypervisors, SIGOPS Oper. Syst. Rev. 41 (3) 275–287.

Stream 2021, viewed 1 September 2021, http://www.cs.virginia.edu/stream/ref.html.

Varrette, S. et al. 2013, HPC performance and energy-efficiency of Xen, KVMand VMware hypervisors, in: 25th International Symposium on Computer Architecture and High Performance Computing, (SBAC-PAD), IEEE, 2013.

Xavier M. et al 2013, Performance evaluation of container-based virtualization for high performance computing environments, in: 21st Euromicro International Conference on Parallel, Distributed and Network-Based Processing, (PDP), IEEE.

Zhanibek Kozhirbayev and Richard O. Sinnott 2017, A performance comparison of container-based technologies for the Cloud, Future Generation Computer Systems 68, pp 175–182.