Software as a Service for content Website

Theoretical Study

Cloud Computing

- Intro
- Definition
- Why Cloud Computing?
- Characteristic
- Deployment Models
- Service Model
    - IaaS
        - Implementation Example: OpenStack
    - PaaS
    - SaaS
        - Intro
        - Definition
        - Live Examples
        - Business Consideration(Advantages)
        - Architecture
            - Traditional Web Application Architecture
            - Cloud Architecture for SaaS
                - Use case: AWS
                - Key Components
            - Multi-instance Architecture
            - Multi-tenant Architecture
            - Comparison between Multi-instance and Multi-tenant

- Economics of multitenancy life
- Implementations comparison
    - **Performance comparison between containers & virtual machines**
        - KVM
        - Linux containers
        - Benchmarking
            - CPU
            - Memory bandwidth
            - Random Memory Access
            - Network bandwidth
            - Block I/O
        - Conclusions
    - **Containers**
        - Intro
        - Containers types
    - **Kubernetes**
        - Compare kubernetes,Swarm Docker and Apache Mesos
        - More on Kubernetes
        - Kubernetes Architecture
        - Benefits of Kubernetes
        - Drawbacks of Kubernetes
        - Kubernetes - Cluster Architecture

        - Kubernetes - Master Machine Components
        - Kubernetes - Node Components
        - Kubernetes - Master and Node Structure
    - **Added Value**
    - **Tools**
        - **Native tools**
        - **Third-party tools**
- References

Cloud Computing

Introduction:

Cloud computing is a growing idea in the world of IT, born out of the necessity for computing on the go. It brings the user access to data, applications and storage that are not stored on their computer. For a very simple cloud computing overview, it can be understood as a delivery system that delivers computing the same way a power grid delivers electricity. To the average computer user it offers the advantage of delivering IT without the user having to have an in depth knowledge of the technology. Similar to the way a consumer can access electricity without being an electrician.

By delivering this computing, storage and applications as a service, not a product, the cloud offers both a cost and business advantage. The cloud moves all these services off-site to either a contractor, or a centralized facility. Centralizing the data allows the cost to be shared amongst all the users. The cloud accomplishes what IT is always seeking to; increase computing capabilities, without having to provide a new infrastructure. The possible uses of cloud computing are exponential. Users interface with the cloud through their web browser, eliminating the need for installing numerous software applications.

Definition: Resources on the internet.

OR: Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing

resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction. This cloud model is composed of five essential characteristics, three service models, and four deployment models.

Why Cloud Computing?

When compared to traditional software cloud computing offers:

1. Flexibility

Cloud-based services are ideal for businesses with growing or fluctuating bandwidth demands. If your needs increase it's easy to scale up your cloud capacity, drawing on the service's remote servers. Likewise, if you need to scale down again, the flexibility is baked into the service. This level of agility can give businesses using cloud computing a real advantage over competitors

2. Disaster recovery

Businesses of all sizes should be investing in robust disaster recovery, but for smaller businesses that lack the required cash and expertise, this is often more an ideal than the reality. Cloud is now helping more organisations buck that trend.

3. Automatic software updates

The beauty of cloud computing is that the servers are off-premise, out of sight and out of your hair. Suppliers take care of them for you and roll out regular software updates – including security updates – so you don't have to worry about wasting time maintaining the system yourself. Leaving you free to focus on the things that matter, like growing your business.

4. Capital-expenditure Free

Cloud computing cuts out the high cost of hardware. You simply pay as you go and enjoy a subscription-based model that's kind to your cash flow

5. Increased collaboration

When your teams can access, edit and share documents anytime, from anywhere, they're able to do more together, and do it better. Cloud-based workflow and file sharing apps help them make updates in real time and gives them full visibility of their collaborations.

6. Work from anywhere

With cloud computing, if you've got an internet connection you can be at work. And with most serious cloud services offering mobile apps, you're not restricted by which device you've got to hand.

7. Security

Lost laptops are a billion dollar business problem. And potentially greater than the loss of an expensive piece of kit is the loss of the sensitive data inside it. Cloud computing gives you greater security when this happens. Because your data is stored in the cloud, you can access it no matter what happens to your machine. And you can even remotely wipe data from lost laptops so it doesn't get into the wrong hands.

Characteristics:

1-On-demand self-service.

A consumer can unilaterally provision computing capabilities, such as server time and network storage, as needed automatically without requiring human interaction with each service provider.

2-Broad network access.

Capabilities are available over the network and accessed through standard mechanisms that promote use by heterogeneous thin or thick client platforms (e.g., mobile phones, tablets, laptops, and workstations).

3-Resource pooling.

The provider's computing resources are pooled to serve multiple consumers using a multi-tenant model, with different physical and virtual resources dynamically assigned and reassigned according to consumer demand. There is a sense of location independence in that the customer generally has no control or knowledge over the exact location of the provided resources but may be able to specify location at a higher level of abstraction (e.g., country, state, or datacenter). Examples of resources include storage, processing, memory, and network bandwidth.

4-Rapid elasticity.

Capabilities can be elastically provisioned and released, in some cases automatically, to scale rapidly outward and inward commensurate with demand. To the consumer, the capabilities available for provisioning often appear to be unlimited and can be appropriated in any quantity at any time.

5-Measured service.

Cloud systems automatically control and optimize resource use by leveraging a metering capability at some level of abstraction appropriate to the type of service (e.g., storage, processing, bandwidth, and active user accounts). Resource usage can be monitored, controlled, and reported, providing transparency for both the provider and consumer of the utilized service.

Deployment Models:

1-Private cloud.

The cloud infrastructure is provisioned for exclusive use by a single organization comprising multiple consumers (e.g., business units). It may be owned, managed, and operated by the organization, a third party, or some combination of them, and it may exist on or off premises.

2-Community cloud.

The cloud infrastructure is provisioned for exclusive use by a specific community of consumers from organizations that have shared concerns (e.g., mission, security requirements, policy, and compliance considerations). It may be owned, managed, and operated by one or more of the organizations in the community, a third party, or some combination of them, and it may exist on or off premises.

3-Public cloud.

The cloud infrastructure is provisioned for open use by the general public. It may be owned, managed, and operated by a business, academic, or government organization, or some combination of them. It exists on the premises of the cloud provider.

4-Hybrid cloud.

The cloud infrastructure is a composition of two or more distinct cloud infrastructures (private, community, or public) that remain unique entities, but are bound together by standardized or proprietary technology that enables data and application portability (e.g., cloud bursting for load balancing between clouds).

Service Models:

1-Infrastructure as a Service (IaaS).

The capability provided to the consumer is to provision processing, storage, networks, and other fundamental computing resources where the consumer is able to deploy and run arbitrary software, which can include operating systems and applications. The consumer does not manage or control the underlying cloud infrastructure but has control over operating systems, storage, and deployed applications; and possibly limited control of select networking components (e.g., host firewalls).

Implementation Example:

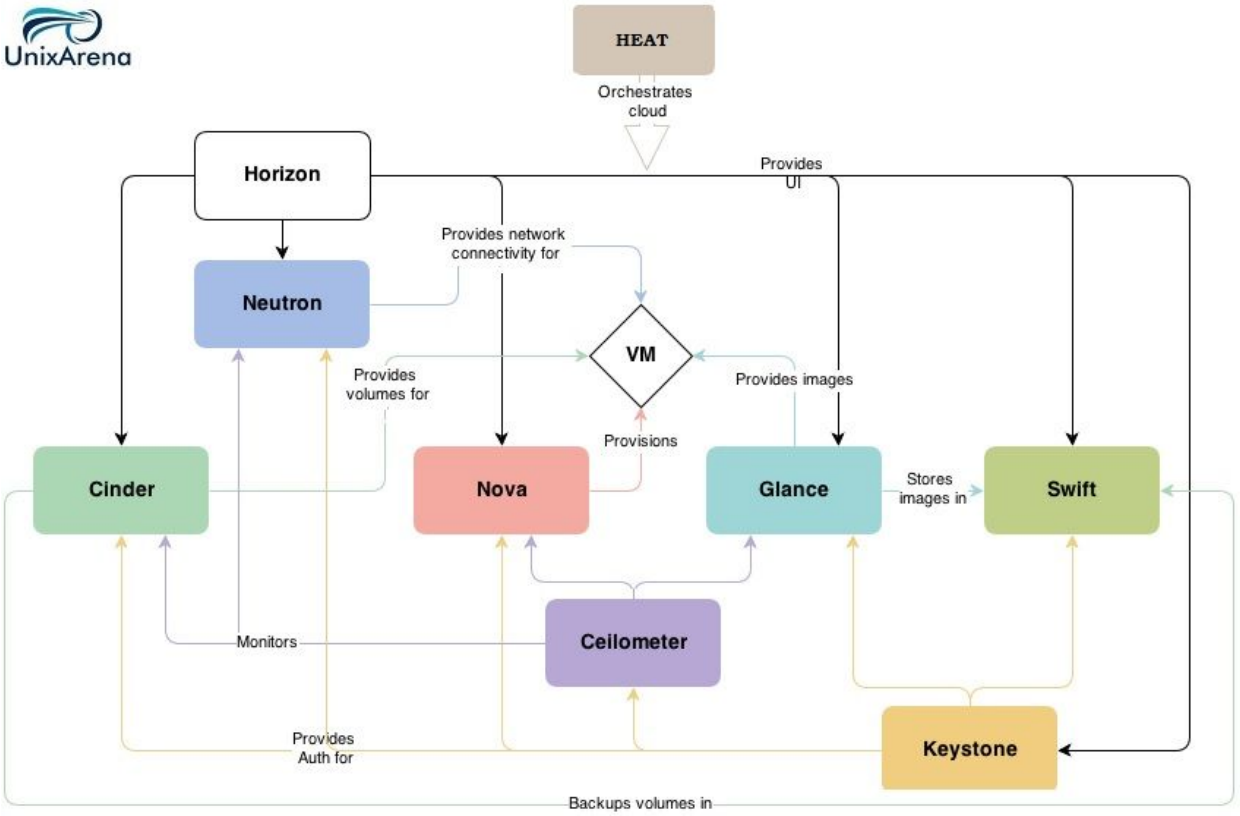
OpenStack

OpenStack is a cloud computing platform that controls large number of compute nodes , storage, and networking resources throughout a datacenter, all managed through a dashboard(Horizon) that gives administrators control while empowering their users to provision resources through a web interface. Openstack provides an Infrastructure-as-a-Service (IaaS) solution through a set of interrelated services

is the list of openstack Services , project name and description:

Service	Project name	Description	Requirement
Dashboard	Horizon	Web-Based Dashboard	Mandatory
Compute	Nova	Create virtual Machine & manage VM	Mandatory
Networking	Neutron	Software defined networking (Advanced Networking)	Optional
Object Storage	Swift	Store files & Directories	Optional
Block Storage	Cinder	Volume & Snapshot Management	Mandatory
Identity service	Keystone	Creating Projects/User/Roles/Token Management/Authentication	Mandatory
Image Service	Glance	To Manage OS Images	Optional
Telemetry	Ceilometer	Monitoring & Billing purpose	Optional
Orchestration	Heat	HOT(Heat Orchestration Template) based on YAML	Optional
Database Service	Trove	Database as a Service	Optional
Hadoop as Service	sahara	Hadoop as Service	Optional
Messaging	RabbitMQ	Messaging	Mandatory

This diagram shows how the openstack components are interconnected:



- **Nova** is a computing engine. It is used for deploying and managing large numbers of virtual machines.
- **Swift** is a storage system for objects and files.
- **Cinder** is a block storage component. It accesses specific locations on a disk drive.
- **Neutron** provides the networking capability.
- **Horizon** is the dashboard of Openstack. It is the only graphical interface (WEB UI)
- **Keystone** provides identity services. It is essentially a central list of all the users.

- **Glance** provides image services to OpenStack. In this case, "images" refers to images (or virtual copies) of hard disks.
- **Ceilometer** provides telemetry services, which allow the cloud to provide billing services to individual users of the cloud.
- **Heat** allows developers to store the requirements of a cloud application in a file that defines what resources are necessary for that application.

2-Platform as a Service (PaaS).

The capability provided to the consumer is to deploy onto the cloud infrastructure consumer-created or acquired applications created using programming languages, libraries, services, and tools supported by the provider.3 The consumer does not manage or control the underlying cloud infrastructure including network, servers, operating systems, or storage, but has control over the deployed applications and possibly configuration settings for the application-hosting environment.

3-Software as a Service (SaaS).

The capability provided to the consumer is to use the provider's applications running on a cloud infrastructure . The applications are accessible from various client devices through either a thin client interface, such as a web browser (e.g., web-based email), or a program interface. The consumer does not manage or control the underlying cloud infrastructure including network, servers, operating systems, storage, or

even individual application capabilities, with the possible exception of limited user specific application configuration settings.

SaaS

Software as a Service

Intro

Cloud application services, or Software as a Service (SaaS), represent the largest cloud market and are still growing quickly. SaaS uses the web to deliver applications that are managed by a third-party vendor and whose interface is accessed on the clients' side. Most SaaS applications can be run directly from a web browser without any downloads or installations required, although some require plugins.

Because of the web delivery model, SaaS eliminates the need to install and run applications on individual computers. With SaaS, it's easy for enterprises to streamline their maintenance and support, because everything can be managed by vendors: applications, runtime, data, middleware, OSes, virtualization, servers, storage and networking

---

Definition

Software- as- a –Service (SaaS) is an application delivery model that enables users to utilize a software solution over the Internet. SaaS revenue models are typically subscription based, where users pay a fixed recurring fee over a period of time (often monthly or annually).

Live Examples

- cloud-based Microsoft Office 365

  Signature Microsoft productivity applications such as Word, Excel and PowerPoint are longtime staples of the workplace, but the cloud-based Microsoft Office 365 dramatically expands the Office suite's parameters. Users now may create, edit and share content from any PC, Mac, iOS, Android or Windows device in real-time, connect with colleagues and customers across a range of tools from email to video conferencing and leverage a range of collaborative technologies supporting secure interactions both inside and outside of the organization.

- Google Apps

  Google long ago expanded beyond its search and advertising roots to offer businesses a comprehensive suite of productivity tools. Google Apps includes custom professional email (complete with spam protection), shared calendars and video meetings alongside Google Drive. A cloud-based document storage solution, Google Drive enables staffers to access files from any device and share them instantly with colleagues, in the process eliminating email attachments as well as the hassles of merging different versions.

- Amazon Web Services

  Amazon, too, has evolved beyond its core e-commerce platform to support the on-demand delivery of cloud-based IT resources and applications, bolstered by pay-as-you-go pricing options. Amazon Web Services currently encompasses more than 70 services in all, including computing, storage, networking,

database, analytics, deployment, management and tools for the Internet of Things.

Business Consideration(**Advantages**)

SaaS offerings present a number of unique challenges when compared to traditional software product models. Specifically, providers must take into account the following:

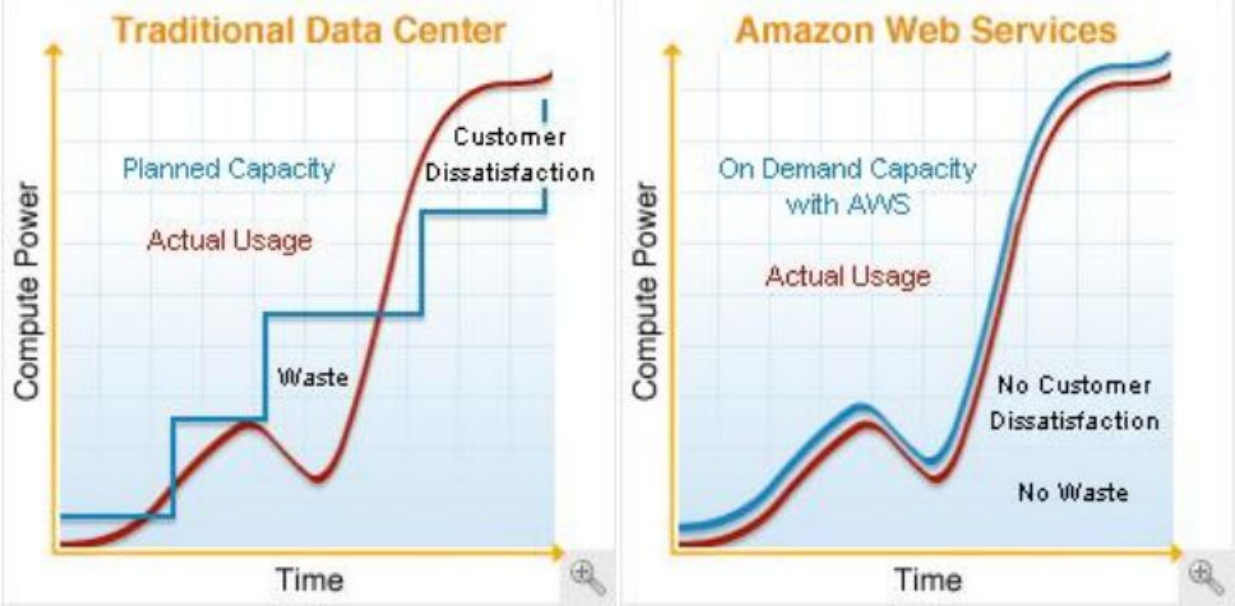
- Managing Cash Flow

  Subscription pricing models, in contrast to perpetual licensing models, don't have large, one time, upfront license revenue coming in as customers on-board. Instead, customers pay smaller, recurring fees over the lifetime of the contract. When choosing a hosting platform, this is an important difference to keep in mind. Traditional hosting environments require large initial capital outlays to procure required infrastructure before a single customer can be onboarded. Forecasting demand well ahead of actual usage becomes necessary, which in turn exposes two distinct business risks: If the business performs better than expected, there may not be enough servers on hand to support new paying customers. On the other hand, if demand is over-forecasted, you will have low asset utilization, high fixed operating expenses, and ultimately increase the required time duration to recognize positive cash flow for a given customer

  Example Used: Amazon Web Service

  AWS enables providers to avoid the expense of owning servers or operating data centers. Our service enables SaaS providers to add and remove resources as needed based on the real-time demands of their business. This significantly

decreases forecasting risk, and improves cash flow positions by only paying for resources when they are actually needed. AWS services can be consumed without minimum usage contracts or long-term commitments, helping providers retain maximum business flexibility.



- Reduce Platform Support

  A SaaS delivery model allows an ISV to dedicate their valuable engineering resources to develop technology that leads to greater innovation and differentiation of their solution in the market. With the distributed client server computing model, the end user takes on the responsibility of building the platform required to support the business application. As such, there are an infinite number of configurations that an ISV must “certify” to in order to support their customers.
  Instead of spending tens of millions of dollars on sustaining engineering and legacy support, SaaS allows an ISV to focus on one platform of their choosing.

- Increase Sales Velocity and Customer Satisfaction

No longer does an enterprise sales executive have to wait 4 weeks for a customer to acquire hardware, operating system licenses and data center space followed by installation and configuration of the business application to start an evaluation. The reality is that cloud computing allows ISVs to deliver software in a truly on demand manner. Reduce sales cycles from months to days. Reduce time to value for the customer from months to hours.

- Managing Uptime for a Global Customer Base

  A highly reliable and available IT infrastructure requires SaaS providers to not only maintain reliable storage and backup devices, but also operate a reliable network with redundant networking devices, transit connections, and physical connections between data centers. In addition to backup and reliable networking, SaaS providers must also have a tested, working solution for disaster recovery. This includes deploying data and applications across multiple data centers – either with failure resilient software or in a more traditional hot/cold standby approach. To achieve realistic disaster recovery, all of the data centers and servers involved have to be constantly utilized; if they sit idle, it's almost certain they won't function as desired when activated from a cold start. SaaS providers need to account for both the cost and the complexity of this redundancy when evaluating their deployment.

- Providing a Secure Environment

  Another direct cost for SaaS providers running their applications is ensuring the confidentiality, integrity, and availability of business critical data. Examples of security costs for SaaS providers include capital expenditures for network security devices, security software licenses, staffing of an information security organization, costs associated with

information security regulatory compliance, physical security requirements, smart cards for access control, and so on.

- Overall Cost

  Example Used: Amazon Web Service

  These include cost of the sizable IT infrastructure teams that are needed to handle the “heavy lifting” – managing heterogeneous hardware and the related supply chain, staying up-to-date on data center design, negotiating contracts, dealing with legacy software, operating data centers, moving facilities, scaling and managing physical growth, etc. – all the things that AWS’s services handle on behalf of SaaS providers.

Architecture

SaaS architectures are often variations of the classic three-tier web application hosting model. Design priorities are typically reliability, security, availability, performance, and cost.

Traditional Web Application Architecture

This traditional web hosting architecture is built around a common three-tier web application model that separates the architecture into presentation, application and persistence layers. This architecture has already been designed to scale out by adding additional hosts at the persistence or application layers and has built-in performance, failover and availability features.

Exterior Firewall
Hardware or Software Solution to open standard Ports (80, 443)

Web Load Balancer
Hardware or Software solution to distribute traffic over web servers

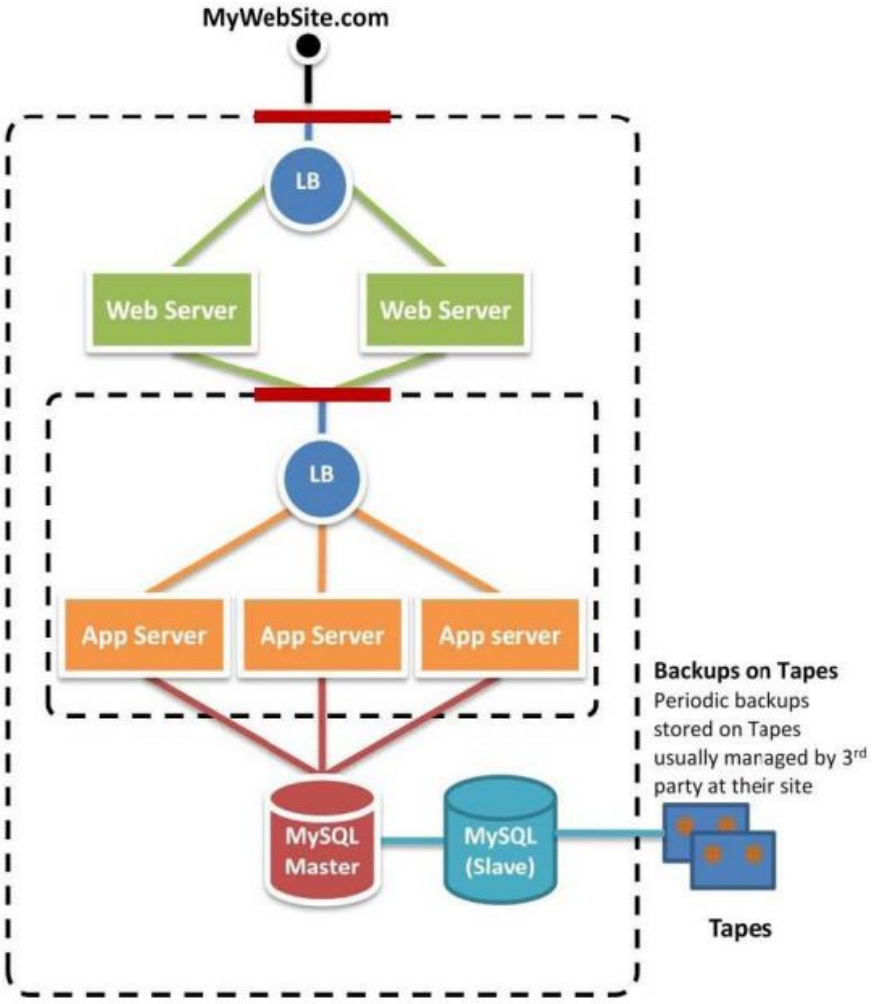
Web Tier
Fleet of machines handling HTTP requests.

Backend Firewall limits access to application tier from web tier

App Load Balancer
Hardware or Software solution to spread traffic over app servers

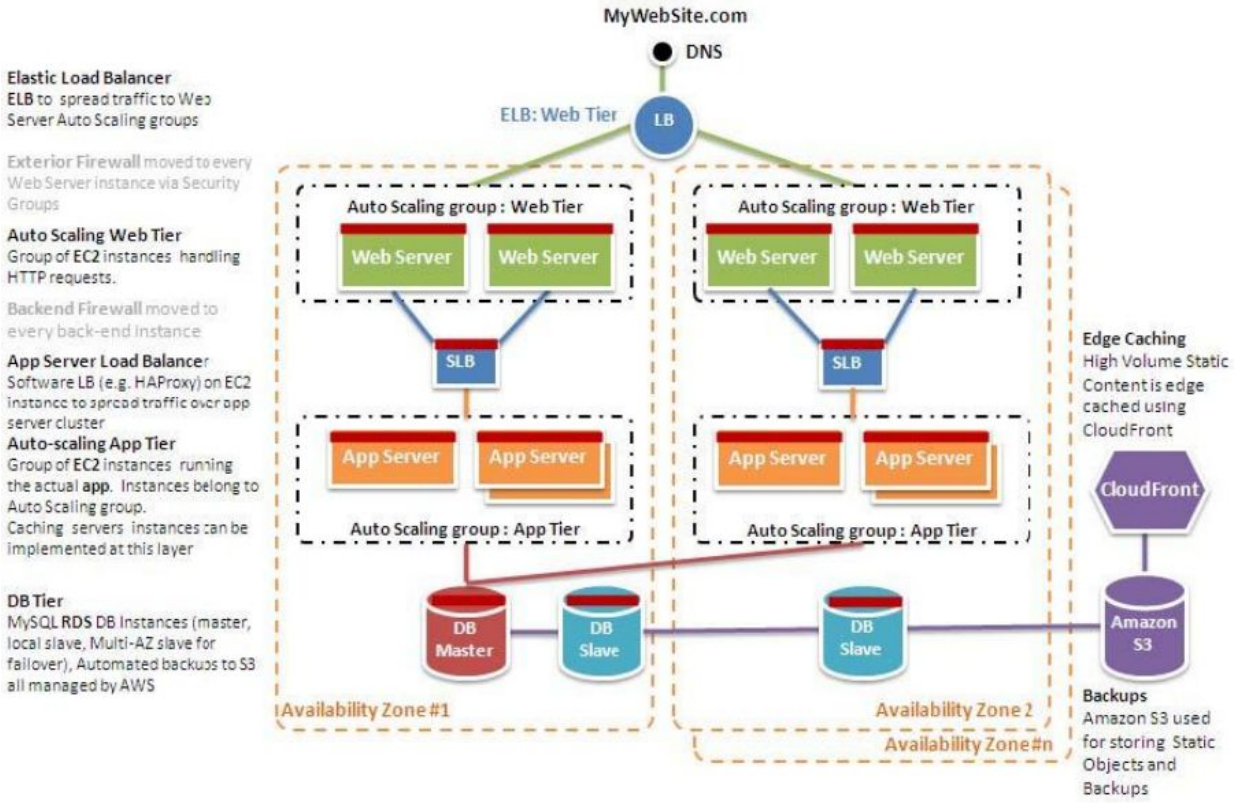
App Server Tier
Fleet of machines handling Application specific workloads
Caching server machines can be implemented at this layer

Data Tier
Database Server machines with master and local running separately, Network storage for Static objects



Cloud Architecture for SaaS

Use case: AWS (Amazon Web Services)



Key Components of an AWS Web Hosting Architecture

- Edge caching
- Elastic IPs and DNS
- Controlling access to hosts
    - Unlike a traditional web hosting model, there is no border firewall that controls all traffic into the data center. Instead, every EC2 host has a local firewall that you can configure for access.

- Load balancing across clusters
- Finding other hosts and services
    - Since most web application architectures have a database server, which is always on, this is a common repository for configuration. Using this model newly added hosts can request the list of necessary endpoints for

communications from the database as part of a bootstrapping phase.

- Caching within the web application
- Database configuration, backup and failover
- Storage and backup of data and assets
- Auto-scaling the fleet
    - One of the key differences between the AWS web architecture and the traditional hosting model is the ability to dynamically scale the web application fleet on-demand to handle increased or decreased traffic.
    - For example, if the web servers are reporting greater than 80% CPU utilization then an additional web server could be quickly deployed and then added to the load balancer host. Once the web server responds correctly to the URL checks coming from the load balancer then it would take its place in the load rotation and future requests can now be served by this newly deployed web server. This same capability can be used for handling failover scenarios as well. For example, if the monitoring component has not received status updates from Load Balancer #1 then the Load Balancer #2 could be reconfigured to add the hosts behind Load Balancer #1 for a period of time. Once a new host has been deployed to replace Load Balancer #1 then Load Balancer #2 can again be reconfigured to have its original set of hosts.

- Failover with AWS
    - Another key advantage when using AWS versus traditional web hosting is the availability of simple to manage availability zones that give the web application developer access to multiple datacenters for the deployment of virtual hosts. As can be seen in the AWS

web hosting architecture, it is recommended to spread EC2 hosts across multiple AZs since this provides for an easy solution to making your web application fault tolerant.

Multi-instance Architecture

By contrast, a multi instance architecture uses one application instance per client. With the multi-instance model, organizations must devote their time to efficiently creating and managing multiple application instances

---

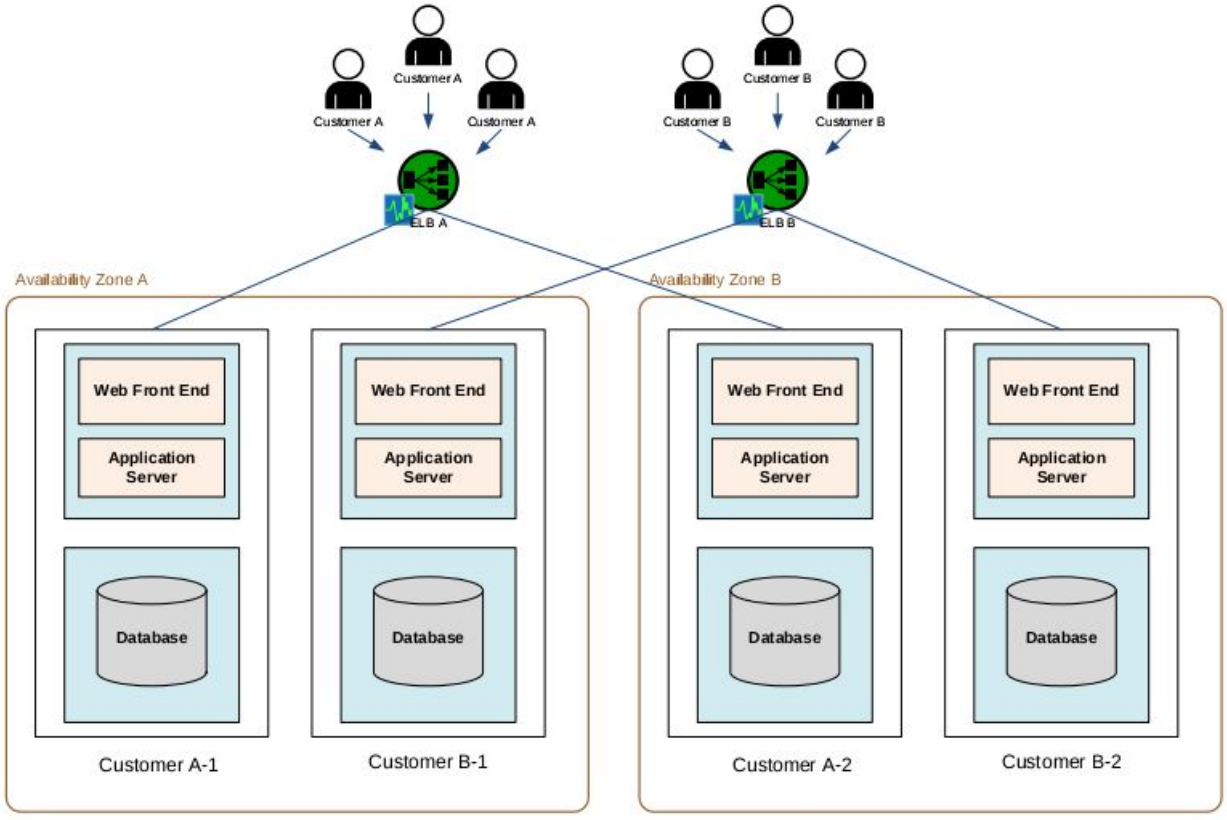
Multi-tenant Architecture

A software architecture in which a single instance of software runs on a server and serves multiple tenants. A tenant is a group of users who share a common access with specific privileges to the software instance. With a multitenant architecture, a software application is designed to provide every tenant a dedicated share of the instance - including its data, configuration, user management, tenant individual functionality and non-functional properties. Multitenancy contrasts with multi-instance architectures, where separate software instances operate on behalf of different tenants.

Comparison between Multi-instance and Multi-tenant

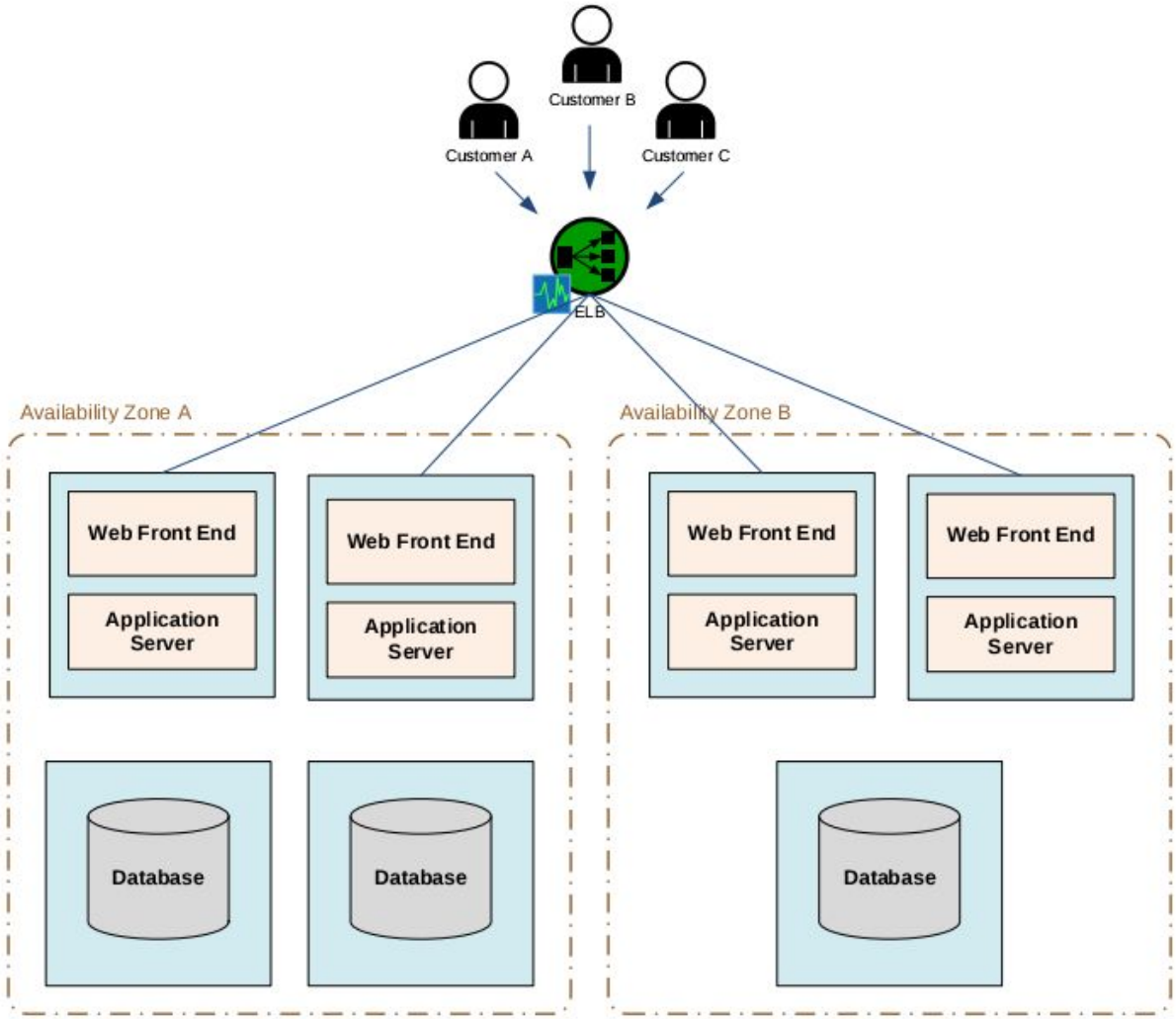
The fundamental architectural consideration when designing a SaaS application is deciding how to handle the issue of multi-tenancy. One way of handling multi-tenancy is to rely on virtualization as the mechanism for keeping application modules separated for each customer. Another option is to have a single application that has logical separation of data built and enforced in each architectural layer.

The following diagram depicts the strategy of provisioning a separate SaaS environment for each customer that leverages the best practices highlighted in this document:



This approach will make sense for a SaaS application that will be constructed from subsystems of an existing singletenant application or packaged software.

If the SaaS application is being built from the ground up, it may make more sense to create a single application that has multi-tenancy baked into every layer of the architecture. This approach is likely to make more efficient use of available resources and allow for more flexibility when operating the infrastructure. That said, it is likely going to be significantly more difficult to design and implement from a software development perspective. The following diagram depicts this approach:



Economics of multitenancy life

- **Cost savings**

  Multitenancy allows for cost savings over and above the basic economies of scale achievable from consolidating IT resources into a single operation. An application instance usually incurs a certain amount of memory and processing overhead which can be substantial when multiplied by many customers, especially if the customers are small. Multitenancy reduces this overhead by amortizing it over many customers. Further cost savings may come from licensing costs of the underlying software (such as operating systems and database management systems). Put crudely, if you can run everything on a single software instance, you only have to buy one software license. The cost savings can be eclipsed by the difficulty of scaling the single instance as demand grows - increasing the performance of the instance on a single server can only be done by buying faster hardware, such as fast CPUs, more memory, and faster disk systems, and typically these costs grow faster than if the load was split between multiple servers with roughly the same aggregate capacity. In addition, development of multitenant systems is more complex, and security testing is more stringent owing to the fact that multiple customers' data is being co-mingled.

- **Data aggregation/data mining**

  One of the most compelling reasons for vendors/ISVs to utilize multitenancy is for the inherent data aggregation benefits. Instead of collecting data from multiple data sources, with potentially different

database schemas, all data for all customers is stored in a single database schema. Thus, running queries across customers, mining data, and looking for trends is much simpler. This reason is probably overhyped as one of the core multitenancy requirements is the need to prevent Service Provider access to customer (tenant) information. Further, it is common to separate the operational database from the mining database (usually because of different workload characteristics), thus weakening the argument even more.

- **Complexity**

  Because of the additional customization complexity and the need to maintain per-tenant metadata, multitenant applications require a larger development effort.

- **Release management**

  Multitenancy simplifies the release management process. In a traditional release management process, packages containing code and database changes are distributed to client desktop and/or server machines; in the single-instance case, this would be one server machine per customer. These packages then have to be installed on each individual machine. With the multitenant model, the package typically only needs to be installed on a single server. This greatly simplifies the release management process, and the scale is no longer dependent on the number of customers.

  At the same time, multitenancy increases the risks and impacts inherent in applying a new release version. As there is a single software instance serving multiple tenants, an update on this instance may cause downtime for all tenants even if the update is

requested and useful for only one tenant. Also, some bugs and issues resulted from applying the new release could manifest in other tenants' personalized view of the application. Because of possible downtime, the moment of applying the release may be restricted depending on time usage schedule of more than one tenant.

Implementations comparison

Performance **comparison** between

Containers & **Virtual machines**

while achieving isolation and resource control

Isolation and resource control for cloud applications has traditionally been achieve through the use of virtual machines.
Deploying applications in a VM results in reduced performance due to the extra levels of abstraction. In a cloud environment, this results in loss efficiency for the infrastructure.

Newer advances in **container-based virtualization** simplifies
the deployment of applications while isolating them from one another.

Results show that containers result in equal or better performance than VM in almost all cases.

Isolation refers to the requirement that the execution of one workload cannot affect the execution of
a different workload on the same system.
Resource control refers to the ability to constrain a workload to a specific set of resources.

Virtual machines are used extensively in cloud comput-ing; today the concept of infrastructure as a service (IaaS) is largely synonymous with VMs.
For example, Amazon EC2 makes VMs available to customers and it also runs services like databases inside VMs. Many PaaS and SaaS providers are built on IaaS which implies that they run all their work-loads in VMs. Since virtually all cloud workloads are currently running in VMs, **VM performance is a crucial** component of overall cloud performance.

Once a hypervisor has added overhead a higher layer **cannot** remove it, so such overhead becomes a pervasive tax on cloud performance.

Container-based virtualization presents an interesting alternative to virtual machines in the cloud. Although the concepts underlying containers such as namespaces are very
mature, only recently have containers been adopted and standardized in mainstream operating systems, leading to a
renaissance in the use of containers to provide isolation and
resource control.

Linux is the preferred operating system for the cloud due to its zero price, large ecosystem, good hardware support, good performance, and reliability.
The kernel namespaces feature needed to implement containers in Linux has only become mature in the last few years since it was first discussed. Within the last year, **Docker**
has emerged as a standard runtime, image format, and build
system for Linux containers.

This comparison use Docker as a Container and KVM(kernel virtual machine) as a Virtual Machine.

KVM

Kernel Virtual Machine (KVM) is a feature of Linux that allows Linux to act as a type 1 hypervisor, running an unmodified guest operating system (OS) inside a Linux process. KVM uses hardware virtualization featuresin recent processors to reduce complexity and overhead

KVM supports both emulated I/O devices through QEMU and
paravirtual I/O devices using virtio. The combination of hardware acceleration and paravirtual I/O is designed to reduce virtualization overhead to very low levels.

KVM supports live migration, allowing physical servers or even
whole data centers to be evacuated for maintenance without
disrupting the guest OS.
KVM is also easy to use via management tools such as libvirt.

Because a VM has a static number of virtual CPUs (vCPUs) and a fixed amount of RAM, its resource

consumption is naturally bounded. A vCPU cannot use more than one real CPU worth of cycles and each page of vRAM maps to at most one page of physical RAM (plus the nested page table). KVM can resize VMs while running by “hotplugging” and “ballooning” vCPUs and vRAM, although this requires support from the guest OS and is rarely used in the cloud.

Because each VM is a process, all normal Linux resource management facilities like scheduling and cgroups (described in more detail later) apply to VMs. This simplifies implementation and administration of the hypervisor
but complicates resource management inside the guest OS.

Operating systems generally assume that CPUs are always
running and memory has relatively fixed access time, but under KVM vCPUs can be descheduled without notification and virtual RAM can be swapped out, causing performance anomalies that can be hard to debug.

VMs also have two levels of allocation and scheduling: one in the **hypervisor** and one in the guest OS. Many cloud providers eliminate these problems by not

overcommitting resources, pinning each vCPU to a physical CPU, and locking all virtual RAM into real RAM. This essentially eliminates scheduling in the hypervisor. Such fixed resource allocation also simplifies billing.

VMs naturally provide a certain level of isolation and security because of their narrow interface; the only way a VM can communicate with the outside world is through a limited number of hypercalls or emulated devices, both of which are controlled by the hypervisor. This is not a panacea, since a few hypervisor privilege escalation vulnerabilities have been discovered that could allow a guest OS to "break out" of its VM "sandbox".

While VMs excel at isolation, they add overhead when sharing data between guests or between the guest and hypervisor. Usually such sharing requires fairly expensive marshaling and hypercalls. In the cloud, VMs generally access storage through emulated block devices backed by image files; creating, updating, and deploying such disk images can be time-consuming and collections of disk images with

mostly-duplicate contents can waste storage space.

Linux containers

Rather than running a full OS on virtual hardware, container-based virtualization modifies an existing OS to provide extra isolation. Generally this involves adding a container ID to every process and adding new access control checks to every system call. Thus containers can be viewed as another level of access control in addition to the user and group permission system.

Linux containers are a concept built on the kernel namespaces feature, originally motivated by difficulties in dealing with high performance computing clusters. This feature,accessed by the clone() system call, allows creating separate instances of previously-global namespaces. Linux implements filesystem, PID, network, user, IPC, and hostname namespaces.

For example, each filesystem namespace has its own root directory and mount table, similar to chroot() but more powerful.

Namespaces can be used in many different ways, but the most common approach is to create an isolated container that

has no visibility or access to objects outside the container.

Processes running inside the container appear to be running
on a normal Linux system although they are sharing the underlying kernel with processes located in other namespaces.
Containers can nest hierarchically, although this capability has not been much explored.

Unlike a VM which runs a full operating system, a container can contain as little as a single process. A container that behaves like a full OS and runs init, inetd, sshd, syslogd, cron, etc is called a **system container** while one that only runs an application is called an **application container**.

Both types are useful in different circumstances. Since an **application container** does not waste RAM on redundant management processes it generally consumes less RAM than an
equivalent system container or VM. **Application containers**
generally do not have separate IP addresses, which can be an
advantage in environments of address scarcity.

If total isolation is not desired, it is easy to share some resources among containers. For example, bind mounts allow
a directory to appear in multiple containers, possibly in different locations. This is implemented efficiently in the Linux
VFS layer. Communication between containers or between
a container and the host (which is really just a parent names-
pace) is as efficient as normal Linux IPC.

The Linux control groups (**cgroups**) subsystem is used to group processes and manage their aggregate resource consumption. It is commonly used to limit the memory and CPU consumption of containers.

A **container** can be resized by simply changing the limits of its corresponding cgroup.
Cgroups also provide a reliable way of terminating all processes inside a container. Because a containerized Linux system only has one kernel and the kernel has full visibility into the containers there is only one level of resource allocation
and scheduling.

An **unsolved** aspect of container resource management is the fact that processes running inside a container are not aware of their resource limits. For example, a process can see all the CPUs in the system even if it is only allowed to run on a subset of them; the same applies to memory. If an application attempts to automatically tune itself by allocating resources based on the total system resources available it may over-allocate when running in a resource-constrained container.

Securing containers tends to be simpler than managing Unix permissions because the **container** cannot access what it cannot see and thus the potential for accidentally over-broad permissions is greatly reduced. When using user namespaces, the root user inside the container is not treated as root outside the container, adding additional security. The primary type of security vulnerability in containers is system calls that are not namespace-aware and thus can introduce accidental leakage between containers.

Because the Linux system call API set is huge, the process of auditing every system call for namespace-related bugs is still ongoing.
Such bugs can be mitigated (at the cost of potential application incompatibility) by whitelisting system calls using Seccomp.

Several management tools are available for Linux containers, including LXC,systemd-nspawn, lmctfy, Warden, and Docker

Due to its feature set and ease of use, Docker has rapidly become the standard management tool and image format for containers. A key feature of Docker not present in most other container tools is layered filesystem images, usually powered by **AUFS (Another UnionFS)**. AUFS provides a layered stack of filesystems and allows reuse of these lay-
ers between containers reducing space usage and simplifying
filesystem management. A single OS image can be used as
a basis for many containers while allowing each container to have its own overlay of modified files (e.g., app binaries and configuration files). In many cases, Docker container

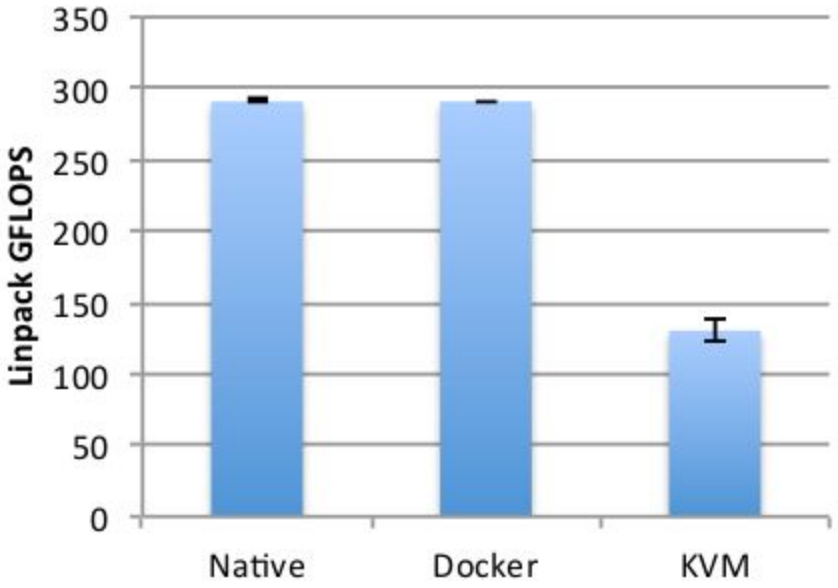
images require less disk space and I/O than equivalent VM
disk images. This leads to faster deployment in the cloud since images usually have to be copied over the network to
local disk before the VM or container can start.

Although this paper focuses on steady-state performance, other measurements have shown than containers can start much faster than VMs (less than 1 second compared to
11 seconds on our hardware) because unlike VMs, contain-
ers do not need to boot another copy of the operating system.
In theory CRIU [1] can perform live migration of containers,
but it may be faster to kill a container and start a new one.

Benchmarking

CPU—Linpack

LINPACK is a software library for performing numerical linear algebra on digital computers.

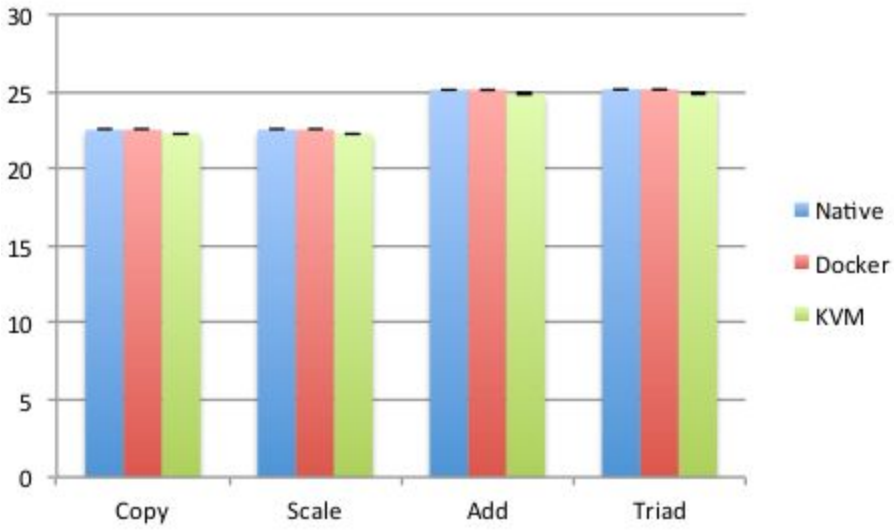


Memory bandwidth—Stream

The STREAM benchmark is a simple synthetic benchmark
program that measures sustainable memory bandwidth when
performing simple operations on vectors

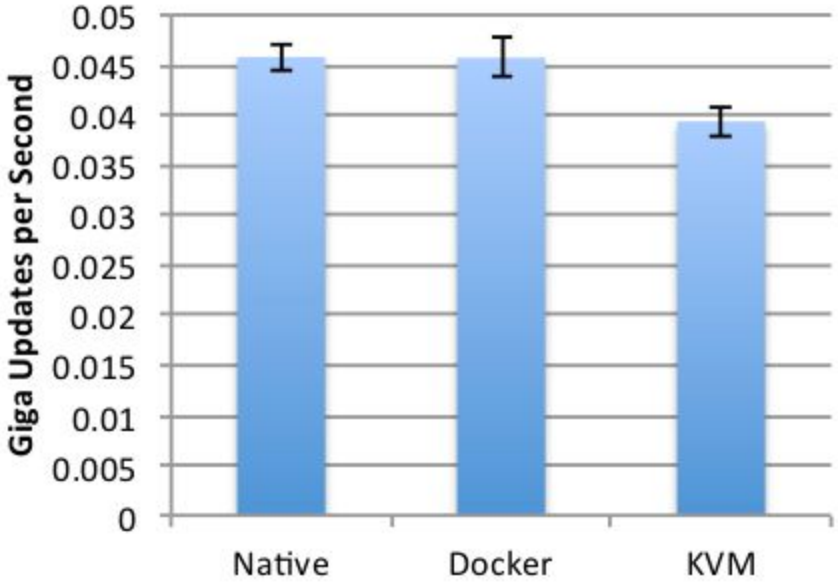
Table 1. Stream components

Name	Kernel	Bytes per iteration	FLOPS per iteration
COPY	$a[i] = b[i]$	16	0
SCALE	$a[i] = q * b[i]$	16	1
ADD	$a[i] = b[i] + c[i]$	24	1
TRIAD	$a[i] = b[i] + q * c[i]$	24	2



Random Memory Access—RandomAccess

The Stream benchmark stresses the memory subsystem in a regular manner, permitting hardware prefetchers to bring in data from memory before it is used in computation. In contrast, the RandomAccess benchmark is specially designed to stress random memory performance. The benchmark initializes a large section of memory as its working set, that is orders of magnitude larger than the reach of the caches or the TLB.



Network bandwidth—nuttcp

The study used the nuttcp tool to measure network bandwidth between the system under test and an identical machine connected using a direct 10 Gbps Ethernet link between two Mellanox ConnectX-2 EN NICs. We applied standard network tuning for 10 Gbps networking such as enabling TCP
window scaling and increasing socket buffer sizes.

As shown in Figure 6, Docker attaches all containers on the host to a bridge and connects the bridge to the network via NAT. In our KVM configuration we use virtio and vhost to minimize virtualization overhead.

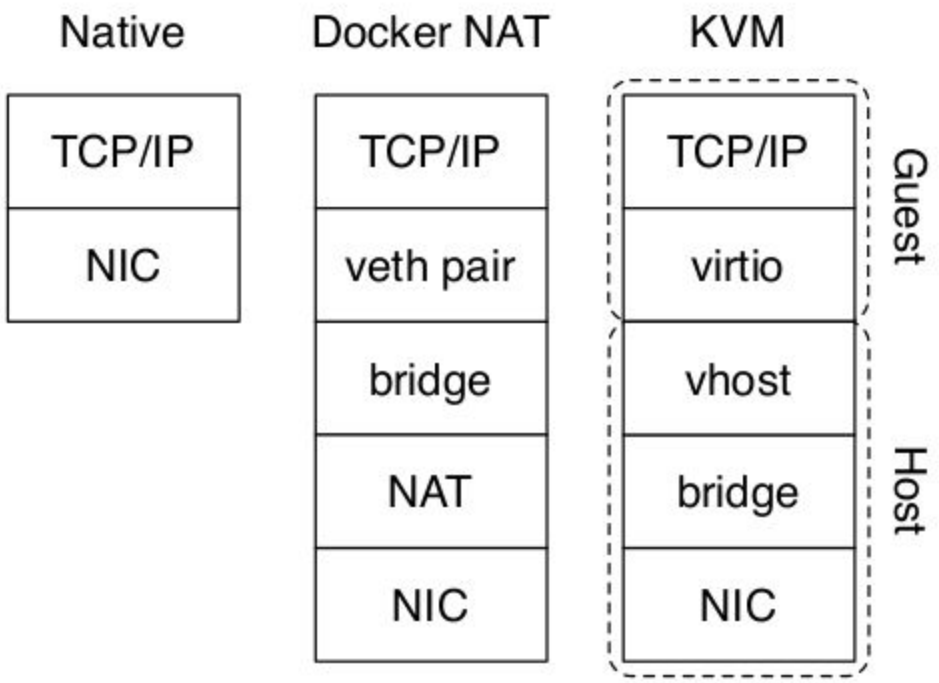
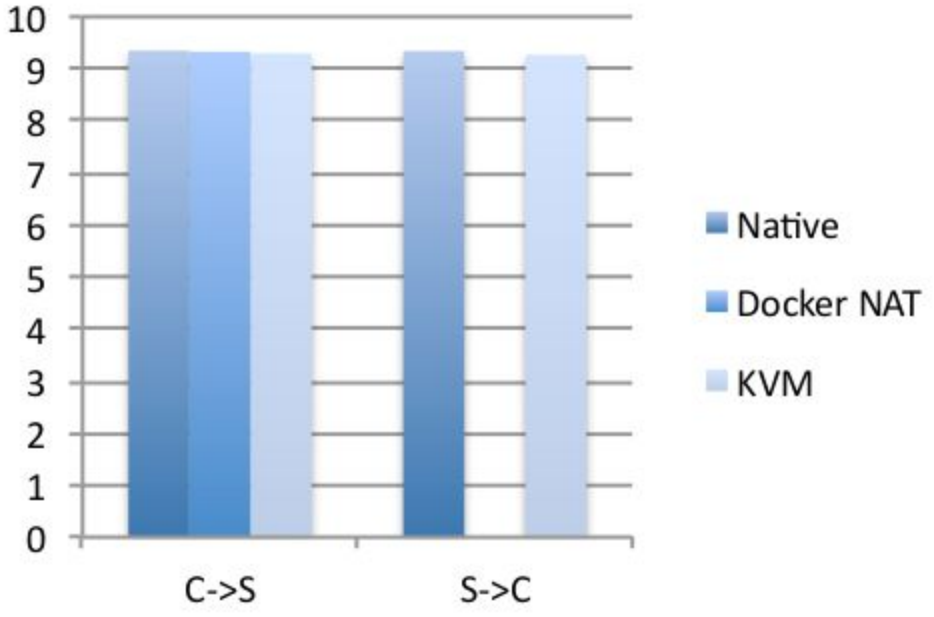


Figure 6. Network configurations



Block I/O—fio

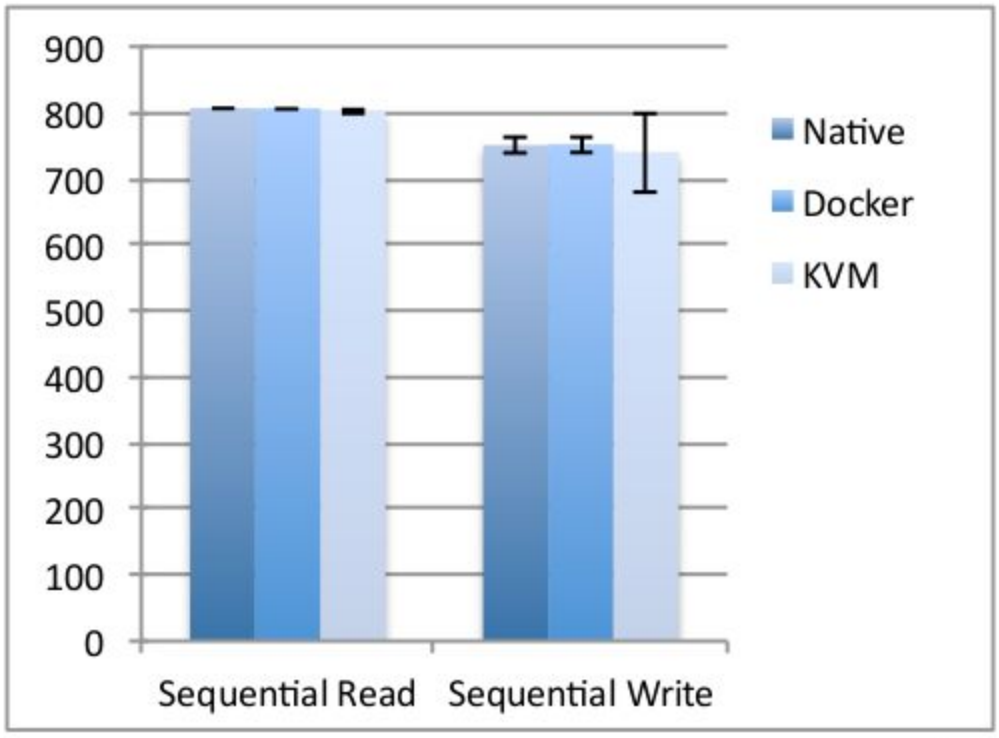


Figure 10. Sequential I/O throughput (MB/s).

Figure 10 shows sequential read and write performance averaged over 60 seconds using a typical 1 MB I/O size. Docker and KVM introduce negligible overhead in this case,although KVM has roughly four times the performance variance as the other cases.

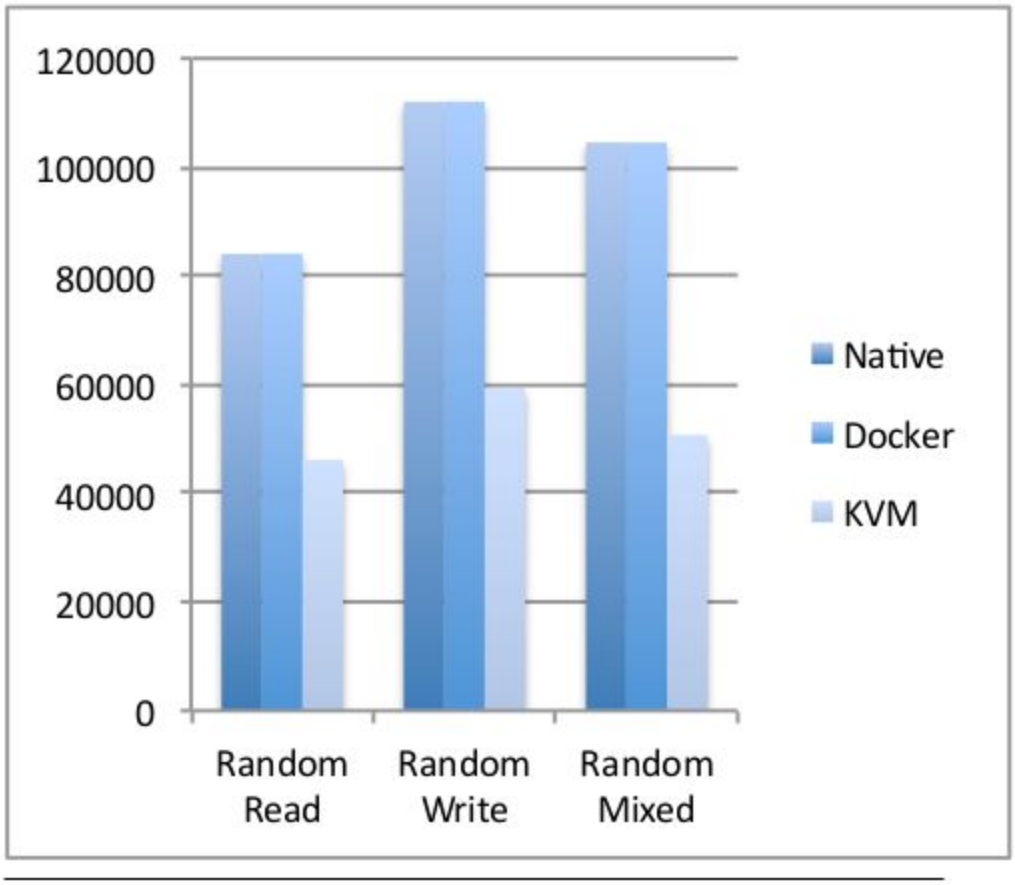


Figure 11. Random I/O throughput (IOPS).

Figure 11 shows the performance of random read, write and mixed (70% read, 30% write) workloads using a 4 kB block size and concurrency of 128, which we experimentally determined provides maximum performance for this particular SSD. As we would expect, Docker introduces no overhead compared to Linux, but KVM delivers only half as many IOPS because each I/O operation must go through QEMU. While the VM's absolute performance is still quite high, it uses more CPU cycles per I/O operation, leaving less CPU available for application work.

Conclusions

Both VMs and containers are mature technology that have benefited from a decade of incremental hardware and software optimizations. In general, Docker equals or exceeds KVM performance in every case we tested. The study results show that both KVM and Docker introduce negligible over-head for CPU and memory performance (except in extreme cases). For I/O-intensive workloads, both forms of virtualization should be used carefully.

Containers

Intro:

Containers are not virtual machines. Containers, at their heart, are a way to fence an application off from other areas or applications on your servers. Applications inside a container have direct access to your underlying server hardware without the overhead of virtualization. They also have the benefit of being portable and self-contained.

With containers, all of your hardware can receive the same configuration, and the containers are self-contained apps on top of a pile of computer power.

Developers no longer need to be at odds with engineers. A developer can create, test, and build an app all in a container. Then they simply hand the container over to the engineering staff or container management platform. The engineer or container management service just needs to know what network and storage requirements the container(s) need. They really don't need much knowledge about the container's function -- just that it runs. This allows increased speed to market, and it lowers engineering costs by bypassing the server-level engineering normally needed to launch applications.
In general, containers allow your team to work better together and to iterate/develop faster.

Containers are an awesome tool, but success with containers is directly related to an engineer's or a business leader's ability to understand them and use them correctly.

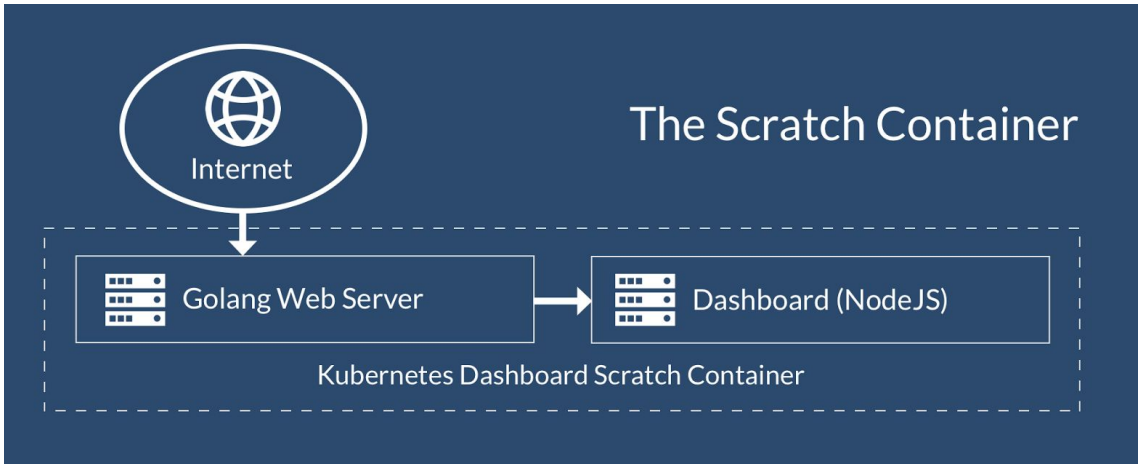
The most progressive organizations have already used containers to reduce costs dramatically and improve their infrastructure performance. The container space has matured at a prodigious rate, and many organizations, like

Netflix and Spotify, have already invested and reaped the benefits of a containerized infrastructure.

Containers Types:

The Scratch Container

You can see here that we are able to run our whole application in one very fast scratch container:



The “Container OS”

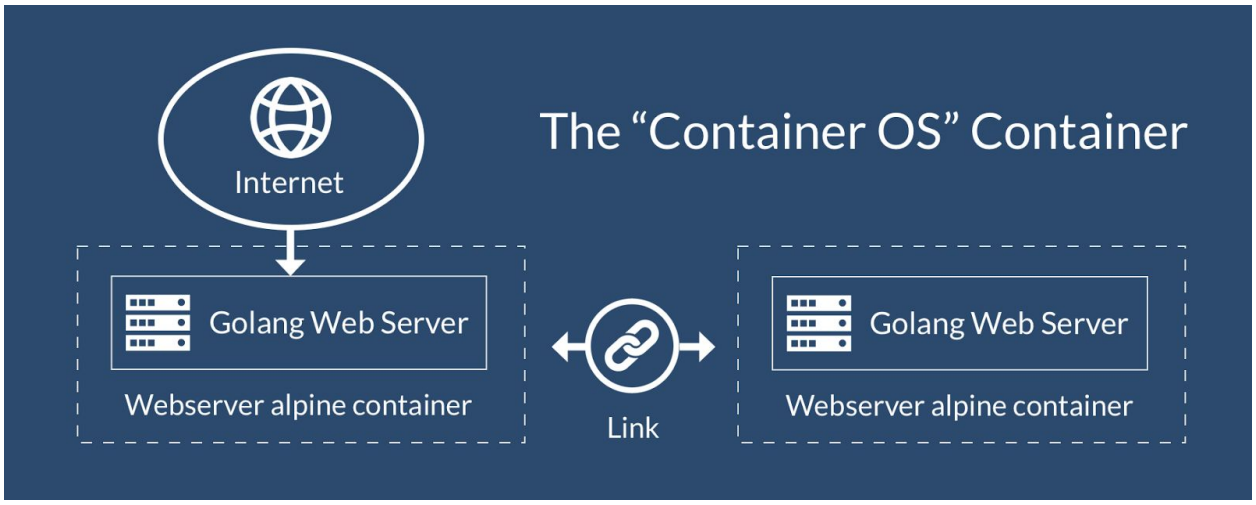
These types of containers are a great option if you can’t get your application running in a scratch environment. It’s important to remember that a container should ideally do just one thing well: run a database, a web server, etc. When running single applications like this, we have no need for many of the default higher-level operating system

functions that can steal CPU and RAM we would otherwise like devoted to our application.

Unlike scratch containers, most of these container images provide a package management tool that allows you to install dependencies. A great example of one of these operating systems is **Alpine OS.**

These containers also have the virtue of being **very small** -- usually **5-8 megabytes** in size. However, dependencies can make these container sizes balloon quickly when they are not kept in check. After installing a typical NodeJS or Rails environment, it is not uncommon to see your container size balloon to 50 or 100 megabytes or more, and this can severely impact container performance.

Let's use our dashboard example from the scratch container to see how it would look in this type. Remember, we only run one application per container, but we can use Docker container linking to link them together into one object. Things get a bit more complicated, but our applications is still fairly fast:



The “Full OS” Container

The Full Operating System container is similar to the Container OS container, but it has all of the features you expect of a full OS -- access to SSH, init, multiple shells, etc. These come at a steep cost to performance.

An example of one of these operating systems would be Ubuntu, or Red Hat server. These types of containers can range from 300-800 megabytes in size. When your container management system deploys this container hundreds or thousands of times a day/week, you can see how the data transfer size can get cumbersome.

The good news is major operating system providers are aware of these issues, and many are releasing smaller

and more compact container-specific versions. These new versions can range from 50-200 megabytes in size, and it appears they are making them smaller and smaller as the container space matures.
I am not going to include a diagram here because it would look the same as the Alpine diagram above. The only exception is that this type of container is the least performant of the bunch.

Kubernetes

Compare kubernetes,swarm Docker and Apache Mesos

While all three technologies make it possible to use containers to deploy, manage, and scale applications, in reality they each solve for different things and are rooted in very different contexts. In fact, none of these three widely adopted toolchains is completely like the others.

Instead of comparing the overlapping features of these fast-evolving technologies, let’s revisit each project’s original mission, architectures, and how they can complement and interact with each other.

Docker

Docker Inc., today started as a Platform-as-a-Service startup named dotCloud. The dotCloud team found that managing dependencies and binaries across many applications and customers required significant effort. So they combined some of the capabilities of Linux cgroups and namespaces into a single and easy to use package so that applications can consistently run on any infrastructure. This package is the Docker image, which provides the following capabilities:

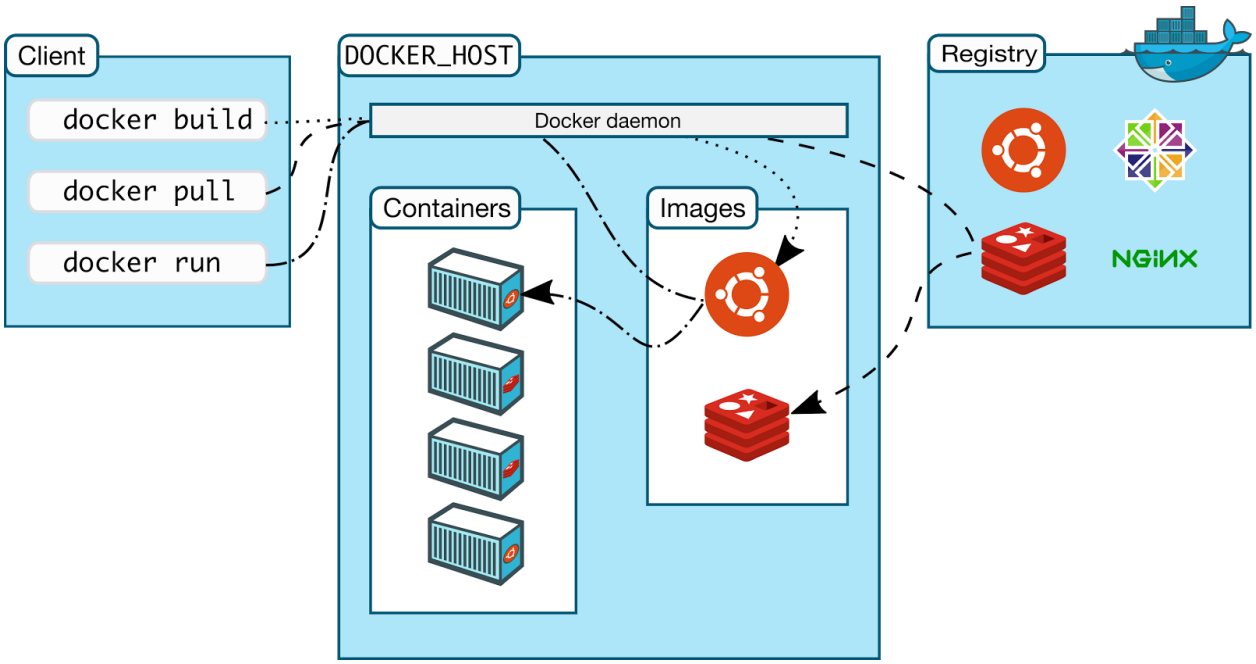
- Packages the application and the libraries in a single package (the Docker Image), so applications can consistently be deployed across many environments;
- Provides Git-like semantics, such as “docker push”, “docker commit” to make it easy for application developers to quickly adopt the new technology and incorporate it in their existing workflows;
- Define Docker images as immutable layers, enabling immutable infrastructure. Committed changes are stored as an individual read-only layers, making it easy to re-use images and track changes. Layers also save disk space and network traffic by only transporting the updates instead of entire images;
- Run Docker containers by instantiating the immutable image with a writable layer that can temporarily store

runtime changes, making it easy to deploy and scale multiple instances of the applications quickly.

Docker grew in popularity, and developers started to move from running containers on their laptops to running them in production. Additional tooling was needed to coordinate these containers across multiple machines, known as container orchestration.

As Docker moved to commercialize the open source file format, the company also started introducing tools to complement the core Docker file format and runtime engine, including:

- Docker hub for public storage of Docker images;
- Docker registry for storing it on-premise;
- Docker cloud, a managed service for building and running containers;
- Docker datacenter as a commercial offering embodying many Docker technologies.



*

Docker’s insight to encapsulate software and its dependencies in a single package have been a game changer for the software industry; the same way mp3’s helped to reshape the music industry. The Docker file format became the industry standard, and leading container technology vendors (including Docker, Google, Pivotal, Mesosphere and many others) formed the Cloud Native Computing Foundation (CNCF) and Open Container Initiative (OCI). Today, CNCF and OCI aim to ensure interoperability and standardized interfaces across container technologies and ensure that any Docker container, built using any tools, can run on any runtime or infrastructure.

Apache Mesos

Apache Mesos started as a UC Berkeley project to create a next-generation cluster manager, and apply the lessons learned from cloud-scale, distributed computing infrastructures such as Google's Borg and Facebook's Tupperware. While Borg and Tupperware had a monolithic architecture and were closed-source proprietary technologies tied to physical infrastructure, Mesos introduced a modular architecture, an open source development approach, and was designed to be completely independent from the underlying infrastructure. Mesos was quickly adopted by Twitter, Apple(Siri), Yelp, Uber, Netflix, and many leading technology companies to support everything from microservices, big data and real time analytics, to elastic scaling.

As a cluster manager, Mesos was architected to solve for a very different set of challenges:

- Abstract data center resources into a single pool to simplify resource allocation while providing a consistent application and operational experience across private or public clouds;

- Colocate diverse workloads on the same infrastructure such analytics, stateless microservices, distributed data services and traditional apps to improve utilization and reduce cost and footprint;
- Automate day-two operations for application-specific tasks such as deployment, self healing, scaling, and upgrades; providing a highly available fault tolerant infrastructure;
- Provide evergreen extensibility to run new application and technologies without modifying the cluster manager or any of the existing applications built on top of it;
- Elastically scale the application and the underlying infrastructure from a handful, to tens, to tens of thousands of nodes.

Mesos has a unique ability to individually manage a diverse set of workloads — including traditional applications such as Java, stateless Docker microservices, batch jobs, real-time analytics, and stateful distributed data services. Mesos' broad workload coverage comes from its two-level architecture, which enables "application-aware" scheduling. Application-aware scheduling is accomplished by encapsulating the application-specific operational logic in a "Mesos framework" (analogous to a runbook in operations). Mesos

Master, the resource manager, then offers these frameworks fractions of the underlying infrastructure while maintaining isolation. This approach allows each workload to have its own purpose-built application scheduler that understands its specific operational requirements for deployment, scaling and upgrade. Application schedulers are also independently developed, managed and updated, allowing Mesos to be highly extensible and support new workloads or add more operational capabilities over time.

MESOS TWO-LEVEL SCHEDULER ARCHITECTURE

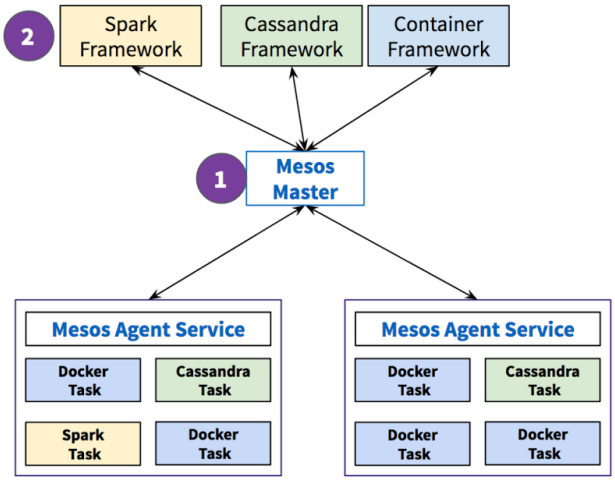
Two-level Scheduling

1 **Mesos Master and Agent**

- Abstracts data center resources into one pool
- Offers & tracks resources across all workloads
- Guarantees isolation
- Restarts workloads on node or task failure

2 **Mesos Framework**

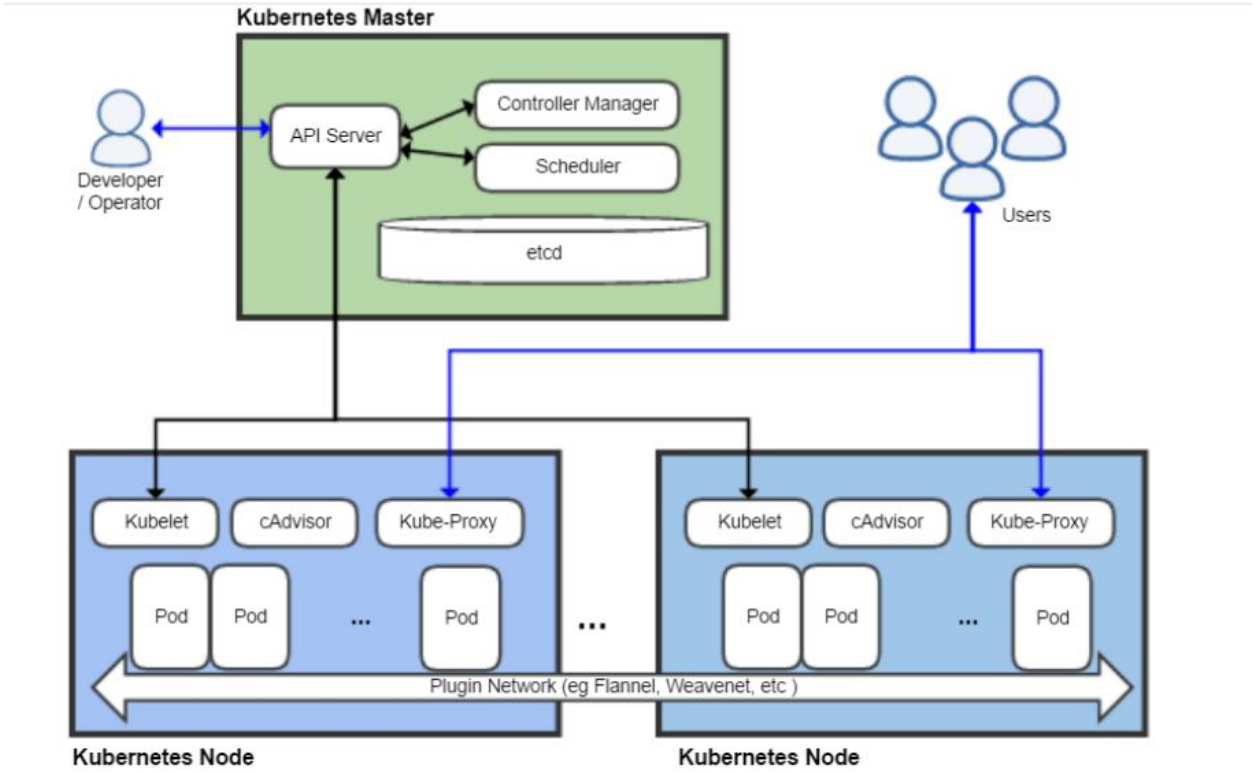
- Consume resources
- Deploys tasks
- Provide application specific logic for deployment, recovery, upgrade..etc



Kubernetes

Google recognized the potential of the Docker image early on and sought to deliver container orchestration “as-a-service” on the Google Cloud Platform. Google had tremendous experience with containers (they introduced cgroups in Linux) but existing internal container and distributed computing tools like Borg were directly coupled to their infrastructure. So, instead of using any code from their existing systems, Google designed Kubernetes from scratch to orchestrate Docker containers. Kubernetes was released in February 2015 with the following goals and considerations:

- Empower application developers with a powerful tool for Docker container orchestration without having to interact with the underlying infrastructure;
- Provide standard deployment interface and primitives for a consistent app deployment experience and APIs across clouds;
- Build on a Modular API core that allows vendors to integrate systems around the core Kubernetes technology.



By March 2016, Google donated Kubernetes to CNCF, and remains today the lead contributor to the project (followed by Redhat, CoreOS and others).

Kubernetes was very attractive for application developers, as it reduced their dependency on infrastructure and operations teams. Vendors also liked Kubernetes because it provided an easy way to embrace the container movement and provide a commercial solution to the operational challenges of running your own Kubernetes deployment (which remains a non-trivial exercise).

Kubernetes is also attractive because it is open source under the CNCF, in contrast to Docker Swarm which, though open source, is tightly controlled by Docker, Inc.

Kubernetes' core strength is providing application developers powerful tools for orchestrating Docker containers.

More on Kubernetes

Kubernetes تمكن من التعامل مع الشبكات تلقائيا، والتخزين،و autoscaling، logs، alerting، وما إلى ذلك لجميع الحاويات . وتكلفة الصيانة منخفضة للغاية. بمجرد إعدادها وتكوينها بشكل صحيح، يمكن التوقع أن الوقت الخاص بتشغيل التطبيقات الخاصة بك مع وقت التوقف منخفضة للغاية، والأداء العظيم، وكما تقلص إلى حد كبير للتدخل الدعم والصيانة.

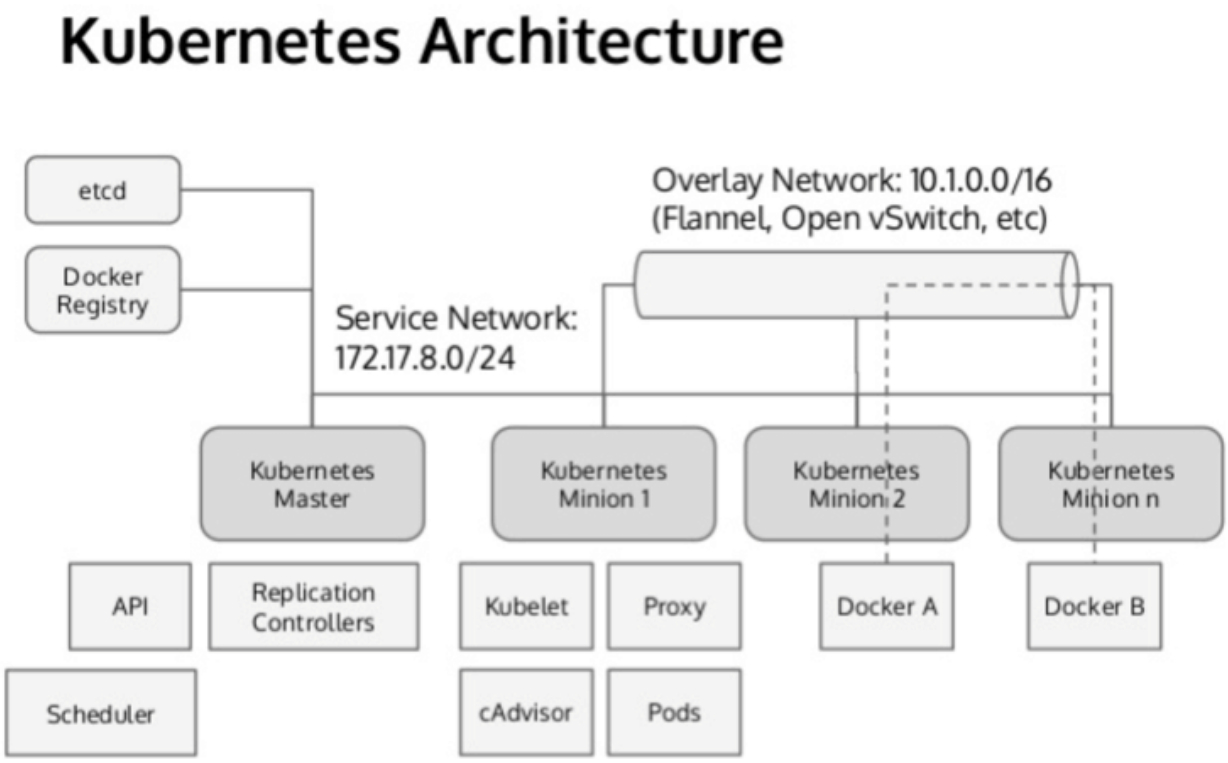
إن Kubernetes مستقرة تمامة ومناسبة بشكل خاص لمنتجات Microsoft

It is important to understand that without containers Kubernetes would not exist. At a high-level, containers look like virtual machines, but the one major difference is that containers share the host system's kernel with other containers.

This is a major differentiating factor as containers share the OS on the physical hosts making them easy to move. You are also able to fit many more containers on a host than VMs because they share the kernel, libraries and binaries. For example, a VM might weigh 20GB, a container running the same application may weigh 200MB.

Containers (whether it is Docker or CoreOS' rkt offering) allow developers to seamlessly focus on their application in runtime. Gone are the days where messy infrastructure issues interfere with agile application development. With containers, developers need only to think of how to properly scale, deploy and manage their newly written application but existing container offerings today are not capable of managing, scheduling and deploying containers across a multi-node environment by themselves.

Kubernetes Architecture:



Pod: The most basic unit is a pod, when a container is assigned it is never actually assigned to physical hardware, instead it is assigned to a pod. Pods typically house containers that all influence or serve a single application. This means all containers within a pod can share volumes and IP space allowing them to be scaled as a single application.

Services: As one begins to build a more robust containerization strategy the concept of “services” become important. A service functions as a resource balancer and conduit from a container to other

containers. This allows for back-end containers to speak with front-end applications through a single, stable access point. This feature allows for consumer ease of use and scalability.

Replication Controllers: These units are tasked with ensuring there are the desired amount of a said container are up and running at any giving point. Let's say there is a kernel panic and a container (or group of containers) were to crash, it is the responsibility of the RC to spin up a replicated pod until the original pod is booted back up. Once the original pod is up and running again, the RC oversees killing the replicated pod.
Virtualization Analogy:
Thinking about a technology that we all know today let's examine Kubernetes through the lens of virtualization. The Master in Kubernetes functions similarly to the vCenter in VMware. The Master is aware of the nodes in the cluster and the capabilities of the nodes. Furthermore, the Master schedules and places Pods similarly to how the vCenter do

---

Benefits of Kubernetes:

While Kubernetes is not the only container management platform on the market (think Docker Swarm or Mesos) it is the industry favorite. Why is that? At the highest-level Kubernetes is gaining traction because it provides a platform that allows containerized applications to be written once and run on all types of cloud infrastructure. It abstracts away all the messy infrastructure differences that exist amongst public cloud providers and whatever infrastructure that powers your private cloud. Furthermore, Kubernetes is progressing to allow developers to run any application they see fit on Kubernetes.ld deploy a VM to a vSphere host. The Pod functions very similarly to the vApp as they house multiple containers across one flat network. Containers are like the VM as they are isolated from their counterparts except if they are given a defined path on the network. Finally, the Replication Controller functions as HA because the RC continuously polls the environment to ensure the proper number of pods are running and if the number is short, it will schedule a new instance.

With Kubernetes, developers can deploy applications quickly and without the risk associated with more traditional platforms (think horizontal scaling across a multi-OS environment), scale applications on the fly, and better resource allocation.

The reduction in hardware usage is another reason companies rave about Kubernetes. Some companies report a 40-50% reduction in the need for hardware due to the lightweight nature of containers and the ability to more quickly kill unused entities (than traditional architectures)

Drawbacks of Kubernetes:

Kubernetes functions as a third-party management system for the containers. Drawbacks and growing pains experienced by container technologies themselves will have an impact on the service Kubernetes can provide.

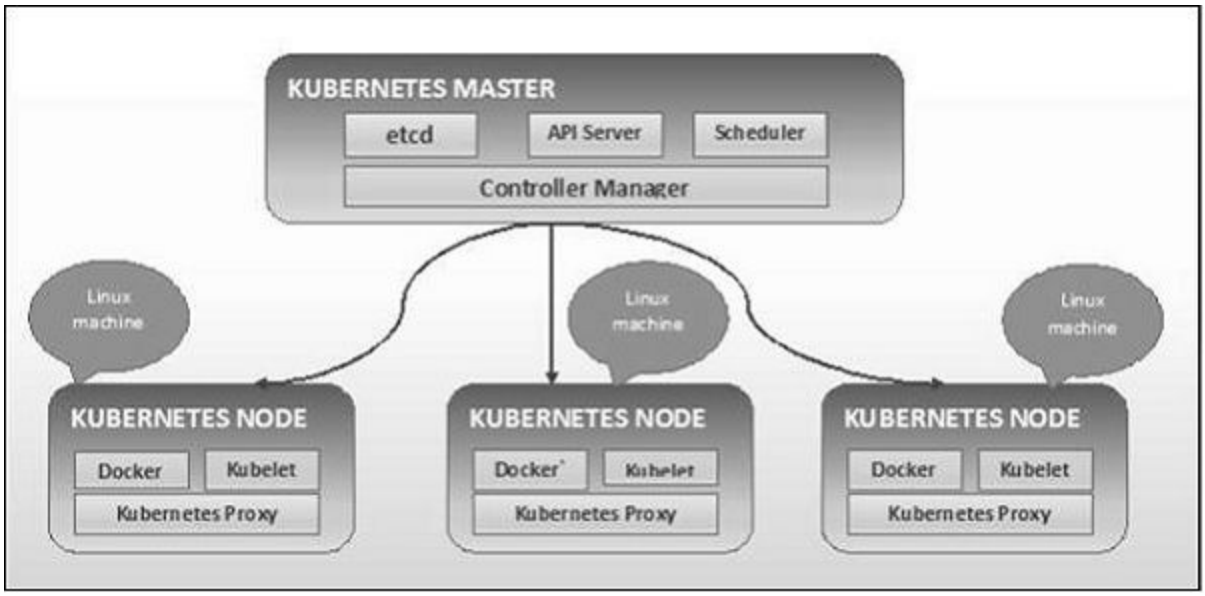
Kubernetes capability of automating deployment, scaling of application, and operations of application containers across clusters. It is capable of creating container centric infrastructure.

some of the important features of Kubernetes:

- Continues development, integration and deployment
- Containerized infrastructure
- Application-centric management
- Auto-scalable infrastructure

- Environment consistency across development testing and production
- Loosely coupled infrastructure, where each component can act as a separate unit
- Higher density of resource utilization
- Predictable infrastructure which is going to be created

Kubernetes - Cluster Architecture



Kubernetes - Master Machine Components:

Following are the components of Kubernetes Master Machine.

- **etcd**

  It stores the configuration information which can be used by each of the nodes in the cluster. It is a high availability key value store that can be distributed among multiple nodes. It is accessible only by Kubernetes API server as it may have some sensitive information. It is a distributed key value Store which is accessible to all.

- **API Server**

  Kubernetes is an API server which provides all the operation on cluster using the API. API server implements an interface, which means different tools and libraries can readily communicate with it. Kubeconfig is a package along with the server side tools that can be used for communication. It exposes Kubernetes API.

- **Controller Manager**

  This component is responsible for most of the collectors that regulates the state of cluster and performs a task. In general, it can be considered as a daemon which runs in nonterminating loop and is responsible for collecting and sending information to API server. It works toward getting the shared state of cluster and then make changes to bring the current status of the server to the desired state. The key

controllers are replication controller, endpoint controller, namespace controller, and service account controller. The controller manager runs different kind of controllers to handle nodes, endpoints, etc.

- **Scheduler**

  This is one of the key components of Kubernetes master. It is a service in master responsible for distributing the workload. It is responsible for tracking utilization of working load on cluster nodes and then placing the workload on which resources are available and accept the workload. In other words, this is the mechanism responsible for allocating pods to available nodes. The scheduler is responsible for workload utilization and allocating pod to new node.

- **etcd**

  It stores the configuration information which can be used by each of the nodes in the cluster. It is a high availability key value store that can be distributed among multiple nodes. It is accessible only by Kubernetes API server as it may have some sensitive information. It is a distributed key value Store which is accessible to all.

- **API Server**

  Kubernetes is an API server which provides all the operation on cluster using the API. API server

implements an interface, which means different tools and libraries can readily communicate with it. Kubeconfig is a package along with the server side tools that can be used for communication. It exposes Kubernetes API.

- **Controller Manager**

  This component is responsible for most of the collectors that regulates the state of cluster and performs a task. In general, it can be considered as a daemon which runs in nonterminating loop and is responsible for collecting and sending information to API server. It works toward getting the shared state of cluster and then make changes to bring the current status of the server to the desired state. The key controllers are replication controller, endpoint controller, namespace controller, and service account controller. The controller manager runs different kind of controllers to handle nodes, endpoints, etc.

- **Scheduler**

  This is one of the key components of Kubernetes master. It is a service in master responsible for distributing the workload. It is responsible for tracking utilization of working load on cluster nodes and then placing the workload on which resources are available and accept the workload. In other words, this is the mechanism responsible for allocating pods to

available nodes. The scheduler is responsible for workload utilization and allocating pod to new node.

Kubernetes - Node Components

Following are the key components of Node server which are necessary to communicate with Kubernetes master.

- **Docker**

  The first requirement of each node is Docker which helps in running the encapsulated application containers in a relatively isolated but lightweight operating environment.

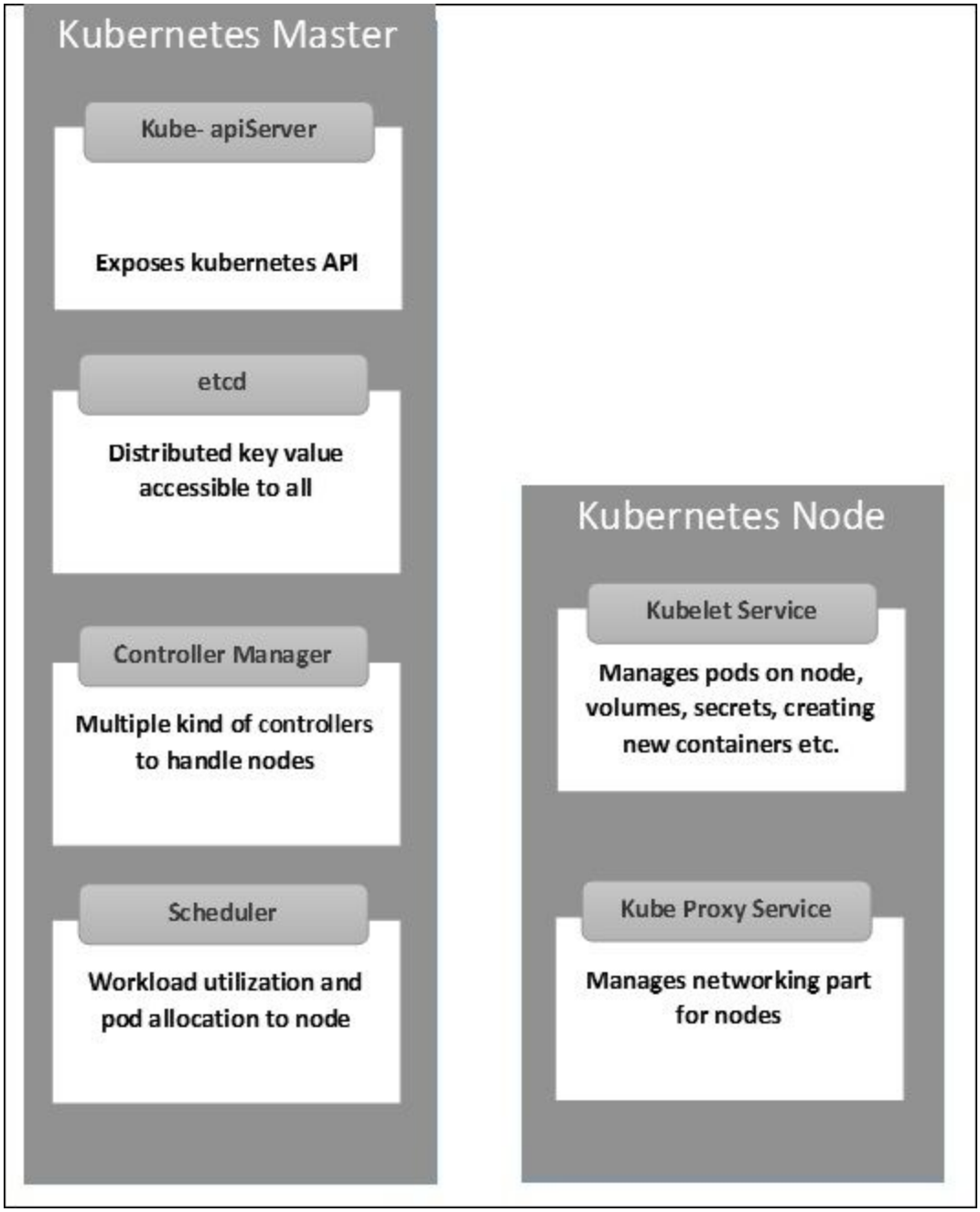
- **Kubelet Service**

  This is a small service in each node responsible for relaying information to and from control plane service. It interacts with etcd store to read configuration details and wright values. This communicates with the master component to receive commands and work. The kubelet process then assumes responsibility for maintaining the state of work and the node server. It manages network rules, port forwarding, etc.

- **Kubernetes Proxy Service**

This is a proxy service which runs on each node and helps in making services available to the external host. It helps in forwarding the request to correct containers and is capable of performing primitive load balancing. It makes sure that the networking environment is predictable and accessible and at the same time it is isolated as well. It manages pods on node, volumes, secrets, creating new containers' health checkup, etc.

Kubernetes - Master and Node Structure



Added Value

Use containers to achieve Software as a service model
Which leads to

1. Faster performance of web applications than traditional web application deployments and service models
2. Less time for failover of servers
3. The ability to make horizontal scaling of your web application at ease
4. Many options for data security and encrypting methods

Tools

Kubernetes contains several built-in tools to help you work with the Kubernetes system, and also supports third-party tooling.

Native Tools

Kubernetes contains the following built-in tools:

- Kubectl
  kubectl is the command line tool for Kubernetes. It controls the Kubernetes cluster manager.

- Kubeadm

  kubeadm is the command line tool for easily provisioning a secure Kubernetes cluster on top of physical or cloud servers or virtual machines (currently in alpha).
- Kubefed

  kubefed is the command line tool to help you administrate your federated clusters.
- Minikube

  minikube is a tool that makes it easy to run a single-node Kubernetes cluster locally on your workstation for development and testing purposes.

- Dashboard

  Dashboard, the web-based user interface of Kubernetes, allows you to deploy containerized applications to a Kubernetes cluster, troubleshoot them, and manage the cluster and its resources itself.
  Third-Party Tools

Third-party tools

Kubernetes supports various third-party tools. These include, but are not limited to:

- Helm

  Kubernetes Helm is a tool for managing packages of pre-configured Kubernetes resources, aka Kubernetes charts.
  Use Helm to:
  - Find and use popular software packaged as Kubernetes charts
  - Share your own applications as Kubernetes charts
  - Create reproducible builds of your Kubernetes applications
  - Intelligently manage your Kubernetes manifest files
  - Manage releases of Helm packages

References:

http://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-145.pdf

https://apprenda.com/library/cloud/

https://www.salesforce.com/uk/blog/2015/11/why-move-to-the-cloud-10-benefits-of-cloud-computing.html

https://apprenda.com/library/paas/iaas-paas-saas-explained-compared/

https://d36cz9buwru1tt.cloudfront.net/SaaS_whitepaper.pdf

https://en.wikipedia.org/wiki/Multitenancy

https://domino.research.ibm.com/library/cyberdig.nsf/papers/0929052195DD819C85257D2300681E7B/$File/rc25482.pdf

https://www.slideshare.net/imesh/an-introduction-to-kubernetes

https://turbonomic.com/blog/on-technology/kubernetes-its-past-present-and-future/

https://stackshare.io/kubernetes

https://stackshare.io/openstack

https://mesosphere.com/blog/docker-vs-kubernetes-vs-apache-mesos/

https://kubernetes.io/docs/tools/