VIT®

Vellore Institute of Technology

(Deemed to be University under section 3 of UGC Act, 1956)

B.Tech. Winter Semester 2020-21

School Of Computer Science and Engineering

(SCOPE)

OPERATING SYSTEMS

FINAL PROJECT DOCUMENT

Team Members
Amit Kumar - 19BCE1281
Shyam Pipalia - 19BCE1152
Mandee Singh - 19BLC1059

Guide
Prof. Menka Pushpa

Cloud Storage With Terminal

Winter Sem 2020-21

- **Objective**
  - Currently there are many web-apps and softwares that provide cloud services. Some of them are free -to use while some are not.
  - Leading cloud storage providers are :
    - iCloud
    - Google Drive
    - Dropbox
    - One Drive
    - Box
    - Mi Cloud
  - All these cloud service systems are undoubtedly very effective and precise for day to day usage but, when it comes to mass data transfer or selective data transfer they are much less effective.
  - For example you have uploaded around 5000 small sized files each containing different file types, from all these files how would you select specific file types to move, copy or delete whatsoever.
  - There are several fields where current cloud storage systems lack such as,
    - Unpredictable Third-Party Apps

      - There are a number of apps and add-ons that work with Google Drive. Some first-party apps from Google work pretty well, but some third-party apps integrated with Google Drive are often bloated adware. Although all the third-party apps

that we found were free to install, many of the apps require payment to unlock all of their features.

- Google Drive permits you to use file storage collaboratively to edit spreadsheets, documents, drawings, presentations, forms, and more.You may like following drive tutorials:

- Privacy

- When you use a cloud provider, your data is no longer on your physical storage. So who is responsible for making sure that data is secure? That's a gray area that is still being figured out.

Solution Proposed.

- All the current cloud storage services contain very efficient and user-friendly Graphical User Interface(GUI) but, as stated above only GUI is not enough for certain types of tasks.

- To overcome this situation we have proposed to attach terminal service along with the graphical user interface.Giving access of the terminal to the user may contain certain potential security risks such like

  - Malicious or harmful activity from the user.
  - Server demadge.
  - Data loss
  - By certain commands users can decrease server health.
  - Privacy issues.
  - Loss of reliability

- We have tried to provide solutions for most of the problems listed above.

Technologies Used.

- GoLang

- Docker
- Containers
- Web Development tools
- HTML, CSS, JS, PHP, DATABASE
- Amazon Web Services(AWS)

Methodology:

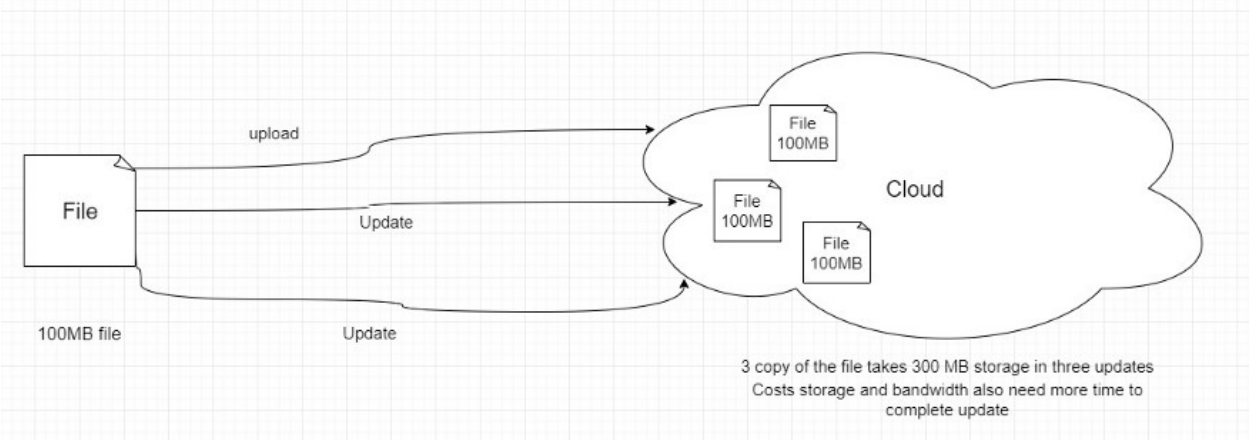
Users should be able to upload and download files/photos from any device. And the files will be synchronized in all the devices that the user is logged in.

➝ Server-side Component Design:

Our user in this system can upload and download files. The user uploads files from the client application/browser, and the server will store them. And user can download updated files from the server.

➝ Upload/Download File:

From the figure, we can see that if we upload the file with full size, it will cost us storage and bandwidth. And also, latency will be increased to complete upload or download.

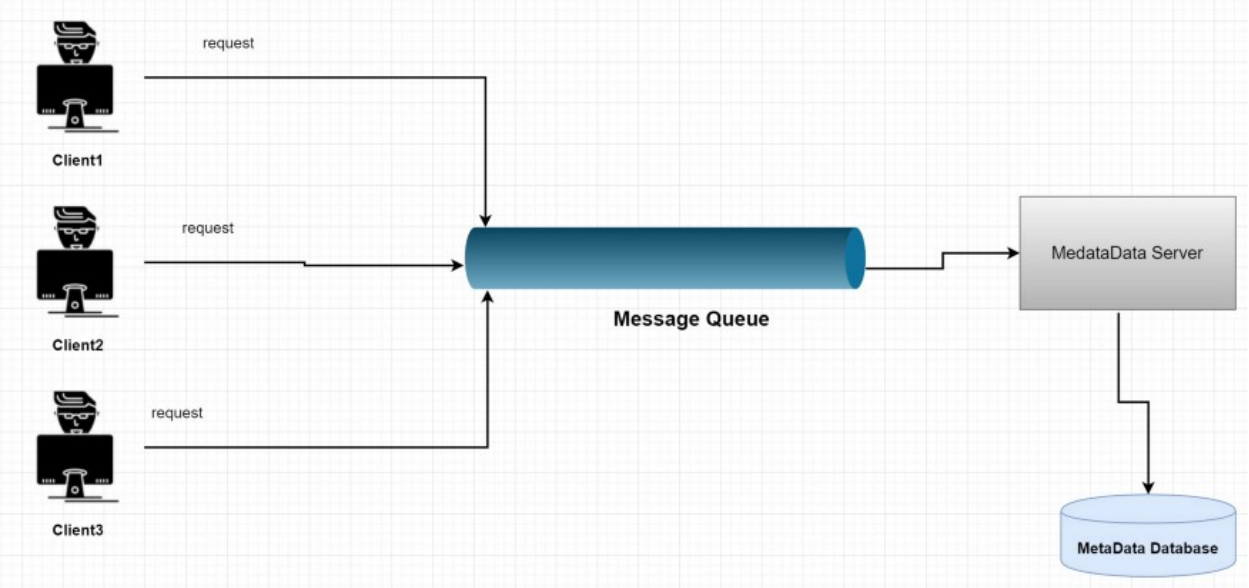


➝ **Synchronization:**

Now the client updates a file from a device; there needs to be a component that process updates and applies the change to other devices. It needs to sync the client's local Database and remote Metadata DB. MetaData server can perform the job to manage metadata and synchronize the user's files.

➝ **Message Queue:**

Now think about it; such a huge amount of users are uploading files simultaneously, how the server can handle such a large number of requests. To be able to handle such a huge amount of requests, we may use a message queue between client and server.



The message queue provides temporary message storage when the destination program is busy or not connected. It provides an asynchronous communications protocol. It is a system that puts a message onto a queue and does not require an immediate response to continue processing. RabbitMQ, Apache Kafka, etc. are some of the examples of the messaging queue.

In case of a message queue, messages will be deleted from the queue once received by a client. So, we need to create several Response Queues for each subscribed device of the client.

For a massive amount of users, we need a scalable message Queue that supports asynchronous message-based communication between client and synchronization service. The service should be able to efficiently store any number of messages in a highly available, reliable, and scalable queue. Example: apache Kafka, rabbitMQ, etc

➔ Cloud Storage:

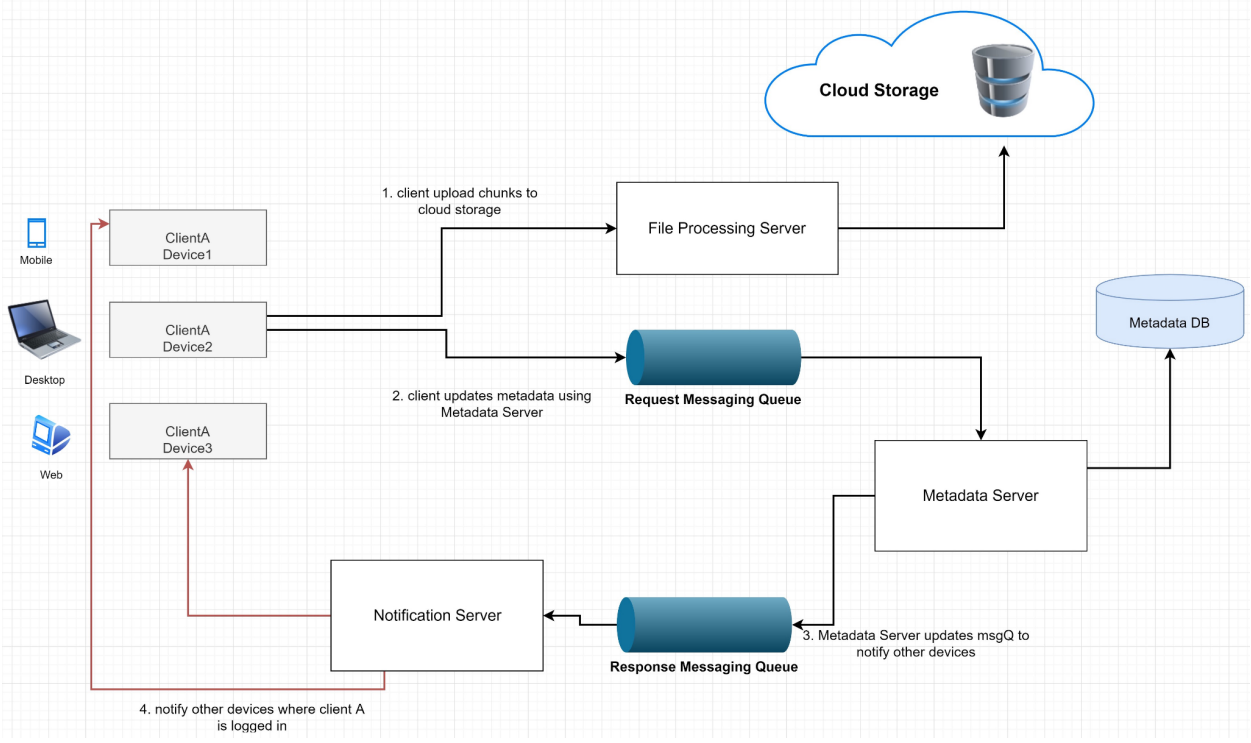
Nowadays, there are many platforms and operating systems like smartphones, laptops, personal computers, etc. They provide mobile access from any place at any time.

If you keep files in the local storage of your laptop and you are going out but want to use it on your mobile phone, how can you get the data? That's why we need cloud storage as a solution.

It stores files(chunks) uploaded by the users. Clients can interact with the storage through File Processing Server to send and receive objects from it. It holds only the files; Metadata DB keeps the data of the chunk size and numbers of a file.

File processing Workflow:

Client A uploads chunk to cloud storage. Client A updates metadata and commits changes in MetadataDB using the Metadata server. The client gets confirmation, and notifications are sent to other devices of the same user. Other devices receive metadata changes and download updated chunks from cloud storage.



➝ Security:

In a file-sharing service, the privacy and security of user data are essential. To handle this, we can store the permissions of each file in the metadata database to give perm what files are visible or modifiable by which user.

➝ **Client-Side:**

The client application(web or mobile) transfers all files that users upload in cloud storage. The application will upload, download, or modify files to cloud storage. A client can update metadata like rename file name, edit a file, etc.

● OS principles and algorithms used.

- **What is containerization?**

Containerization is defined as a form of operating system virtualization, through which applications are run in isolated user spaces called containers, all using the same shared operating system (OS). A container is essentially a fully packaged and portable computing environment:

- Everything an application needs to run – its binaries, libraries, configuration files and dependencies – is encapsulated and isolated in its container.
- The container itself is abstracted away from the host OS, with only limited access to underlying resources – much like a lightweight virtual machine (VM).
- As a result, the containerized application can be run on various types of infrastructure—on bare metal, within VMs, and in the cloud—without needing to refactor it for each environment.

That's because there's less overhead during startup and no need to set up a separate guest OS for each application since they all share the same OS kernel. Because of this high efficiency, containerization is commonly used for packaging up the many individual microservices that make up modern apps. Citrix uses containerization with CPX, an application delivery controller (ADC) that supports more scalable, agile and portable application delivery.

How Does Containerization Actually Work?

Each container is an executable package of software, running on top of a host OS. A host(s) may support many containers (tens, hundreds or even thousands) concurrently, such as in the case of a complex microservices architecture that uses numerous containerized ADCs. This setup works because all containers run minimal, resource-isolated processes that others cannot access.

Think of a containerized application as the top layer of a multi-tier cake:

1. At the bottom, there is the hardware of the infrastructure in question, including its CPU(s), disk storage and network interfaces.
2. Above that, there is the host OS and its kernel – the latter serves as a bridge between the software of the OS and the hardware of the underlying system.
3. The container engine and its minimal guest OS, which are particular to the containerization technology being used, sit atop the host OS.
4. At the very top are the binaries and libraries (bins/libs) for each application and the apps themselves, running in their isolated user spaces (containers).

Cgroups and LXC

Containerization as we know it evolved from cgroups, a feature for isolating and controlling resource usage (e.g., how much CPU and RAM and how many threads a given process can access) within the Linux kernel. Cgroups became Linux containers (LXC), with more advanced features for namespace isolation of components, such as routing tables and file systems. An LXC container can do things such as:

- Mount a file system.
- Run commands as root.
- Obtain an IP address.

It performs these actions in its own private user space. While it includes the special bins/libs for each application, an LXC container does not package up the OS kernel or any hardware, meaning it is very lightweight and can be run in large numbers even on relatively limited machines.

Docker and OCI

LXC serves as the basis for Docker, which launched in 2013 and quickly became the most popular container technology – effectively an industry standard, although the specifications set by the Open Container Initiative (OCI) have since become central to containerization. Docker is a contributor to the OCI specs, which specify standards for the image formats and runtimes that container engines use. Someone booting a container, Docker or otherwise, can expect an identical experience regardless of the computing environment. The same set of containers can be run and scaled whether the user is on a Linux distribution or even Microsoft

Windows. This cross-platform compatibility is essential to today's digital workspaces, in which workers rely on multiple devices, OSes and interfaces to get things done.

What Differentiates Containerization from Virtualization?

The most distinctive feature of containerization is that it happens at the OS level, with all containers sharing one kernel. That is not the case with virtualization via virtual machines (VMs):

- A VM runs on top of a hypervisor, which is specialized hardware, software or firmware for operating VMs on a host machine, like a server or laptop.
- Via the hypervisor, every VM is assigned not only the essential bins/libs, but also a virtualized hardware stack including CPUs, storage and network adapters.
- To run all of that, each VM relies on a full-fledged guest OS. The hypervisor itself may be run from the host's machine OS or as a bare-metal application.

Like containerization, traditional virtualization allows for full isolation of applications so that they run independently of each other using actual resources from the underlying infrastructure. But the differences are more important:

- There is significant overhead involved, due to all VMs requiring their own guest OSes and virtualized kernels, plus the need for a heavy extra layer (the hypervisor) between them and the host.
- The hypervisor can also introduce additional performance issues, especially when it is running on a host OS, for example on Ubuntu.
- Because of the high overall resource overhead, a host machine that might be able to comfortably run 10 or more containers could struggle to support a single VM.

Still, running multiple VMs from relatively powerful hardware is still a common paradigm in application development and deployment. Digital workspaces commonly feature both virtualization and containerization, toward the common goal of making applications as readily available and scalable as possible to employees.

What Are the Main Benefits of Containerization?

Containerized apps can be readily delivered to users in a digital workspace. More specifically, containerizing a microservices-based application, a set of Citrix ADCs or a database (among other possibilities) has a broad spectrum of distinctive benefits, ranging from superior agility during software development to easier cost controls.

More agile, DevOps-oriented software development

Compared to VMs, containers are simpler to set up, whether a team is using a UNIX-like OS or Windows. The necessary developer tools are universal and easy to

use, allowing for the quick development, packaging and deployment of containerized applications across OSes. DevOps engineers and teams can (and do) leverage containerization technologies to accelerate their workflows.

Less overhead and lower costs than VMs

A container doesn't require a full guest OS or a hypervisor. That reduced overhead translates into more than just faster boot times, smaller memory footprints and generally better performance, though. It also helps trim costs, since organizations can reduce some of their server and licensing costs, which would have otherwise gone toward supporting a heavier deployment of multiple VMs. In this way, containers enable greater server efficiency and cost-effectiveness.

Excellent portability across digital workspaces

Containers make the ideal of "write once, run anywhere" a reality. Each container has been abstracted from the host OS and will run the same in any location. As such, it can be written for one host environment and then ported and deployed to another, as long as the new host supports the container technologies and OSes in question. Linux containers account for a big share of all deployed containers and can be ported across different Linux-based OSes whether they're on-prem or in the cloud. On Windows, Linux containers can be reliably run inside a Linux VM or through Hyper-V isolation. Such compatibility supports digital workspaces, in which numerous clouds, devices and workflows intersect.

Fault isolation for applications and microservices

If one container fails, others sharing the OS kernel are not affected, thanks to the user space isolation between them. That benefits microservices-based applications, in which potentially many different components support a larger program. Microservices within specific containers can be repaired, redeployed and scaled without causing downtime of the application

Easier management through orchestration

Container orchestration via a solution such as Kubernetes platform makes it practical to manage containerized apps and services at scale. Using Kubernetes, it's possible to automate rollouts and rollbacks, orchestrate storage systems, perform load balancing and restart any failing containers. Kubernetes is compatible with many container engines including Docker and OCI-compliant ones.

What Applications and Services Are Commonly Containerized?

A container may support almost any type of application that in previous eras would have been traditionally virtualized or run natively on a machine. At the same time, there are several computing paradigms that are especially well-suited to containerization, including:

- Microservices: A microservices architecture can be efficiently configured as a set of containers operating in tandem and spun-up and decommissioned as needed.
- Databases: Database shards can be containerized and each app given its own dedicated database instead of needing to connect all of them to a monolithic database.
- Web servers: Spinning up a web server within a container requires just a few command line inputs to get started, plus it avoids the need to run the server directly on the host.
- Containers within VMs: Containers may be run within VMs, usually to maximize hardware utilization, talk to specific services in the VM and/or increase security.
- Citrix ADCs: An ADC manages the performance and security of an app. When containerized, it makes L4-L7 services more readily available in DevOps environments.

How Is Containerization Related to Microservices and Container Orchestration?

Microservices

The microservices that comprise an application may be packaged and deployed in containers and managed on scalable cloud infrastructure. Key benefits of microservice containerization include minimal overhead, independently scaling, and easy management via a container orchestrator such as Kubernetes.

Citrix ADC can help with the transition from monolithic to microservices-based applications. More specifically, it assists admins, developers and site reliability engineers with networking issues such as traffic management and shifting from monolithic to microservices-based architectures.

When microservices are packaged in containers that are deployed at scale, a container orchestration platform is necessary for managing the life cycles of containers.

Container Orchestration

Kubernetes is the most prominent container orchestration platform. Originally developed by Google, it has seen been open-sourced and is now managed by the Cloud Native Computing Foundation.

While it isn't the only such solution, its feature set is indicative of what a modern container orchestrator should be capable of, as it can:

- Expose containers by DNS name or IP address.
- Handle load balancing and traffic distribution for containers.
- Automatically mount local and cloud-based storage.
- Allocate specific CPU and RAM resources to containers and then fit them onto nodes.

- Replace or kill problematic containers without jeopardizing application performance and uptime.
- Manage sensitive information like password and tokens without rebuilding containers.
- Change the state of containers and roll back old containers to replace them with new ones.

What is Docker?

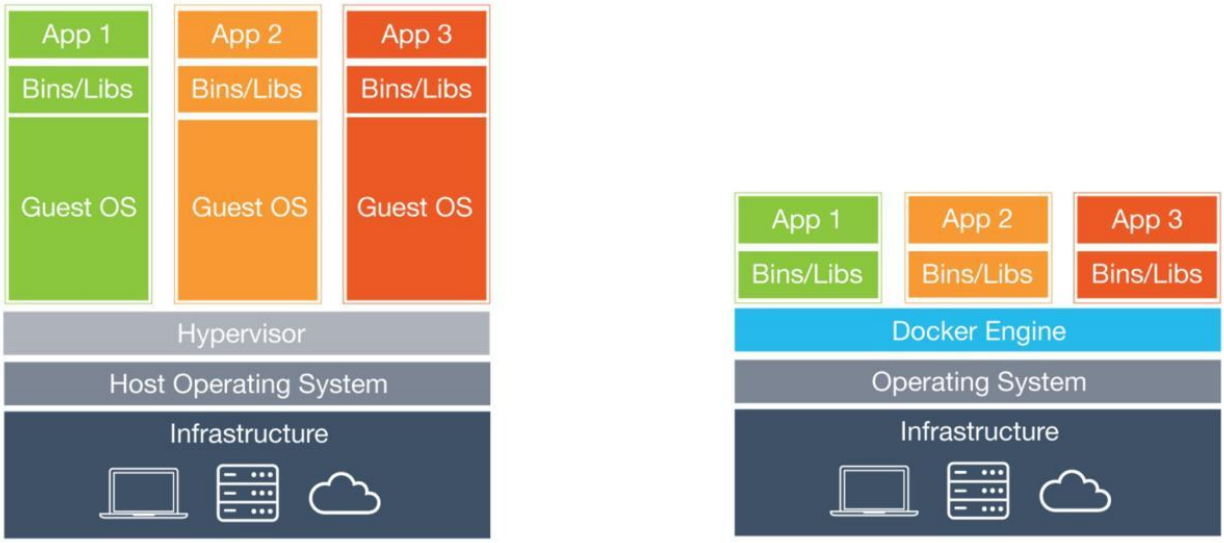
Docker is a software platform for building applications based on *containers* — small and lightweight execution environments that make shared use of the operating system kernel but otherwise run in isolation from one another. While containers as a concept have been around for some time, Docker, an open source project launched in 2013, helped popularize the technology, and has helped drive the trend towards containerization and *microservices* in software development that has come to be known as cloud-native development.

What are containers?

One of the goals of modern software development is to keep applications on the same host or cluster isolated from one another so they don't unduly interfere with each other's operation or maintenance. This can be difficult, thanks to the packages, libraries, and other software components required for them to run. One solution to this problem has been virtual machines,

which keep applications on the same hardware entirely separate, and reduce conflicts among software components and competition for hardware resources to a minimum. But virtual machines are bulky—each requires its own OS, so is typically gigabytes in size—and difficult to maintain and upgrade.

Containers, by contrast, isolate applications' execution environments from one another, but share the underlying OS kernel. They're typically measured in megabytes, use far fewer resources than VMs, and start up almost immediately. They can be packed far more densely on the same hardware and spun up and down en masse with far less effort and overhead. Containers provide a highly efficient and highly granular mechanism for combining software components into the kinds of application and service stacks needed in a modern enterprise, and for keeping those software components updated and maintained.



What is Docker?

Docker is an open source project that makes it easy to create containers and container-based apps. Originally built for Linux, Docker now runs on Windows and MacOS as well. To understand how Docker works, let's take a

look at some of the components you would use to create Docker-containerized applications.

Dockerfile

Each Docker container starts with a Dockerfile. A Dockerfile is a text file written in an easy-to-understand syntax that includes the instructions to build a Docker image (more on that in a moment). A Dockerfile specifies the operating system that will underlie the container, along with the languages, environmental variables, file locations, network ports, and other components it needs—and, of course, what the container will actually be doing once we run it.

Paige Niedringhaus over at ITNext has a good breakdown of the syntax of a Dockerfile.

Docker image

Once you have your Dockerfile written, you invoke the Docker build utility to create an image based on that Dockerfile. Whereas the Dockerfile is the set of instructions that tells build how to make the image, a Docker image is a portable file containing the specifications for which software components the container will run and how. Because a Dockerfile will probably include instructions about grabbing some software packages from online repositories, you should take care to explicitly specify the proper versions, or else your Dockerfile might produce inconsistent images depending on when it's invoked. But once an image is created, it's static. Codefresh offers a look at how to build an image in more detail.

Docker run

Docker's run utility is the command that actually launches a container. Each container is an instance of an image. Containers are designed to be transient and temporary, but they can be stopped and restarted, which launches the container into the same state as when it was stopped. Further, multiple container instances of the same image can be run simultaneously (as long as each container has a unique name). The Code Review has a

great breakdown of the different options for the run command, to give you a feel for how it works.

Docker Hub

While building containers is easy, don't get the idea that you'll need to build each and every one of your images from scratch. Docker Hub is a SaaS repository for sharing and managing containers, where you will find official Docker images from open-source projects and software vendors and unofficial images from the general public. You can download container images containing useful code, or upload your own, share them openly, or make them private instead. You can also create a local Docker registry if you prefer. (Docker Hub has had problems in the past with images that were uploaded with backdoors built into them.)

Docker Engine

Docker Engine is the core of Docker, the underlying client-server technology that creates and runs the containers. Generally speaking, when someone says *Docker* generically and isn't talking about the company or the overall project, they mean Docker Engine. There are two different versions of Docker Engine on offer: Docker Engine Enterprise and Docker Engine Community.

Docker Community Edition

Docker released its *Enterprise Edition* in 2017, but its original offering, renamed Docker Community Edition, remains open source and free of charge, and did not lose any features in the process. Instead, Enterprise Edition, which costs $1,500 per node per year, added advanced management features including controls for cluster and image management, and vulnerability monitoring. The BoxBoat blog has a rundown of the differences between the editions.

How Docker conquered the container world

The idea that a given process can be run with some degree of isolation from the rest of its operating environment has been built into Unix operating systems such as BSD and Solaris for decades. The original Linux container technology, LXC, is an OS-level virtualization method for running multiple isolated Linux systems on a single host. LXC was made possible by two Linux features: namespaces, which wrap a set of system resources and present them to a process to make it look like they are dedicated to that process; and cgroups, which govern the isolation and usage of system resources, such as CPU and memory, for a group of processes.

Containers decouple applications from operating systems, which means that users can have a clean and minimal Linux operating system and run everything else in one or more isolated container. And because the operating system is abstracted away from containers, you can move a container across any Linux server that supports the container runtime environment.

Docker introduced several significant changes to LXC that make containers more portable and flexible to use. Using Docker containers, you can deploy, replicate, move, and back up a workload even more quickly and easily than you can do so using virtual machines. Docker brings cloud-like flexibility to any infrastructure capable of running containers. Docker's container image tools were also an advance over LXC, allowing a developer to build libraries of images, compose applications from multiple images, and launch those containers and applications on local or remote infrastructure.

Docker Compose, Docker Swarm, and Kubernetes

Docker also makes it easier to coordinate behaviors between containers, and thus build application stacks by hitching containers together. Docker Compose was created by Docker to simplify the process of developing and testing multi-container applications. It's a command-line tool, reminiscent of the Docker client, that takes in a specially formatted descriptor file to assemble applications out of multiple containers and run them in concert on a single host.

Docker advantages

Docker containers provide a way to build enterprise and line-of-business applications that are easier to assemble, maintain, and move around than their conventional counterparts.

Docker containers enable isolation and throttling

Docker containers keep apps isolated not only from each other, but from the underlying system. This not only makes for a cleaner software stack, but makes it easier to dictate how a given containerized application uses system resources—CPU, GPU, memory, I/O, networking, and so on. It also makes it easier to ensure that data and code are kept separate. (See "Docker containers are stateless and immutable," below.)

Docker containers enable portability

A Docker container runs on any machine that supports the container's runtime environment. Applications don't have to be tied to the host operating system, so both the application environment and the underlying operating environment can be kept clean and minimal.

For instance, a MySQL for Linux container will run on most any Linux system that supports containers. All of the dependencies for the app are typically delivered in the same container.

Container-based apps can be moved easily from on-prem systems to cloud environments or from developers' laptops to servers, as long as the target system supports Docker and any of the third-party tools that might be in use with it, such as Kubernetes (see "Docker containers ease orchestration and scaling," below).

Normally, Docker container images must be built for a specific platform. A Windows container, for instance, will not run on Linux and vice versa. Previously, one way around this limitation was to launch a virtual machine that ran an instance of the needed operating system, and run the container in the virtual machine.

However, the Docker team has since devised a more elegant solution, called manifests, which allow images for multiple operating systems to be packed side-by-side in the same image. Manifests are still considered experimental, but they hint at how containers might become a cross-platform application solution as well as a cross-environment one.

Docker containers enable composability

Most business applications consist of several separate components organized into a stack—a web server, a database, an in-memory cache. Containers make it possible to compose these pieces into a functional unit with easily changeable parts. Each piece is provided by a different container and can be maintained, updated, swapped out, and modified independently of the others.

This is essentially the microservices model of application design. By dividing application functionality into separate, self-contained services, the microservices model offers an antidote to slow traditional development processes and inflexible monolithic apps. Lightweight and portable containers make it easier to build and maintain microservices-based applications.

Docker containers ease orchestration and scaling

Because containers are lightweight and impose little overhead, it's possible to launch many more of them on a given system. But containers can also be used to scale an application across clusters of systems, and to ramp services up or down to meet spikes in demand or to conserve resources.

The most enterprise-grade versions of the tools for deployment, managing, and scaling containers are provided by way of third-party projects. Chief among them is Google's Kubernetes, a system for automating how containers are deployed and scaled, but also how they're connected together, load-balanced, and managed. Kubernetes also provides ways to create and re-use multi-container application definitions or "Helm charts," so that complex app stacks can be built and managed on demand.

Docker also includes its own built-in orchestration system, Swarm mode, which is still used for cases that are less demanding. That said, Kubernetes has become something of the default choice; in fact, Kubernetes is bundled with Docker Enterprise Edition.

Docker caveats

Containers solve a great many problems, but they aren't cure-alls. Some of their shortcomings are by design, while others are byproducts of their design.

Docker containers are not virtual machines

The most common conceptual mistake people make with containers is to equate them with virtual machines. However, because containers and virtual machines use different isolation mechanisms, they have distinctly different advantages and disadvantages.

Virtual machines provide a high degree of isolation for processes, since they run in their own instance of an operating system. That operating system doesn't have to be the same as the one run on the host, either. A Windows virtual machine can run on a Linux hypervisor and vice versa.

Containers, by contrast, use controlled portions of the host operating system's resources; many applications share the same OS kernel, in a highly managed way. As a result, containerized apps aren't as thoroughly isolated as virtual machines, but they provide enough isolation for the vast majority of workloads.

- **Work done**
    - **Amit Kumar**
        - **Terminal Request**

            When a user requests for a terminal,

We access the user's container, mount his storage to it and provide the terminal of the container to the user, since the terminal is of a jailed environment, the user cannot cause any harm to the actual server.

This container has a base image of ubuntu in it and has its own file system, it a small computer in itself.

You can install anything and run anything in it like running cpp code and java code, anything that can be done in a normal computer

Due to the above a lot of security issues occur, what if the user starts to use our terminal in more ways than just for managing the files. For this very reason we have limited the memory and processing power usage that a user can take up that too for a short time

To project the terminal we have used authentication and the concept of NGROK to do the tunelling process, which uses web sockets internally.

Before mounting and projecting the terminal a lot of checks are made:

1. Is the user authenticated?
2. Are the ports available?
3. Lock the terminal with the provided temporary username and password so that no one else accessed it
4. Is the terminal sharable? If no, move with the above steps, if yes, make sure other users are able to access it as well.

After all the users have left it is made sure that the terminal is shutdown within 60 seconds.

○ Shyam Pipalia

■ Front-end development.

- Frontend Development contains user interface, In which user can request terminal view, upload, move, delete and create files as suited.
- Giving access to terminal to user is a risky task
- Frontend is developed in such a way that whenever a user logs in he is trapped inside of the container whatever files he create or whatever commands he passes stays in the container In this way server security is implemented
- Mount files from container to the server.
- Different permissions level for the files i.e. View only , write permissions etc..
- Files can be selectively compressed from frontend itself

■ Backend scripting.

- Initially we have decided to allocate 1 GB of the space per user. To Implement this,
    - Before the user completes its upload process, if the uploaded file is larger than 1 GB then the backend will deny the upload process and throw an error.
    - If the file is smaller than 1 GB but already the user has consumed their 1GB then also it will throw an error.

- If the whole space is less than 1 GB and total size after the upload exceeds the 1GB limit then also the user can't upload the file.This all functionalities are implemented by PHP scripting.

○ Mandeep Singh:

- **In terms of front end:**
  - As our application starts with authentication of user, login and sign up pages were designed and implemented.
  - Small component like navbar was designed and implemented.
- **Backend Development:**
  - Registration of new users and storing their credentials in our database.
  - Logging users to our site after successful verification.
  - Ensuring that all users have a unique username and email for smooth functioning and preventing unnecessary collisions.
  - Passing data of a user to other webpages with the help of sessions and cookies.
  - Creating a new user directory after every successful new registration.

References

- R. R. Muntz, "Operating systems," in *Computer*, vol. 7, no. 6, pp. 21-21, June 1974, doi: 10.1109/MC.1974.6323579.

- X. Ju and H. Zou, "Operating System Robustness Forecast and Selection," *2008 19th International Symposium on Software Reliability Engineering (ISSRE)*, 2008, pp. 107-116, doi: 10.1109/ISSRE.2008.10.

- Steenken, Dirk & Voss, Stefan & Stahlbock, Robert. (2004). Container terminal operation and operations research - A classification and literature review. OR Spectrum. 26. 3-49. 10.1007/s00291-003-0157-z.