# High-Performance Scientific Computing as a Service in the Cloud

by
Terryn Seaton

A thesis presented to the Honors College of Middle Tennessee State University in partial fulfillment of the requirements for graduation from the University Honors College

Summer 2021

# High-Performance Scientific Computing as a Service in the Cloud

by Terryn Seaton

APPROVED:

Dr. Joshua L. Phillips, Thesis Director
Associate Professor, Computer Science

Dr. Ennio Piano, Thesis Committee Chair
Associate Professor, Economics and University Honors College

## Dedication

I dedicate this thesis to my family and loved ones, with special gratitude to my parents, Terri Seaton and Joel Seaton; my sisters, Destiny Seaton and Jayden Seaton; and my grandmothers, Whitney Seaton and Deamie Priest, who instilled the values of education and hard work within me. Additionally, this thesis is dedicated to my cousins, Canaan Seaton and Charley Clanton, who supported me throughout my time at MTSU. Lastly, to my partner Alex Conyers, who has always loved and supported me through every endeavor. All my goals and accomplishments could not have been possible without their support.

## Acknowledgements

I would like to acknowledge all of the faculty and staff of the MTSU Honors College, especially Judy Albakry, for their support during my time at MTSU. I am sincerely grateful for the opportunity to have been a student within this prestigious institution. Additionally, I would like to thank my thesis advisor, Dr. Joshua Phillips, for guiding me through the writing process of this thesis. Lastly, thank you to my research team, Daniel Cox, Hannah Williams, and Jessica Wijaya, for holding me accountable and aiding me through our research project over the last year.

## Abstract

This thesis explores the process of developing and testing of a containerized Lightweight Directory Access Protocol (LDAP) server using a cloud-based resource model to successfully authenticate users as part of a high-performance computing (HPC) workflow for Middle Tennessee State University (MTSU). In industry, it is commonplace to have an HPC workflow, but not have the orchestration capabilities provided by a cloud computing resource model. As a result, the work done for this thesis serves as a functional prototype to the creation of a containerized LDAP server on a HPC cloud computing cluster. Future work can be done by MTSU to establish a workflow that will package all the development tools students use during their time at the university.

## Table of Contents

## List of Figures

## List of Symbols and Abbreviations

1. Cloud Computing Research Team: CCRT
2. Computer Science: CS
3. Central Processing Unit: CPU
4. High-Performance Computing: HPC
5. Lightweight Directory Access Protocol: LDAP
6. Message Passing Interface: MPI
7. Network File System: NFS
8. Operating System: OS
9. S-STEM Summer Research Experience: SRE
10. Virtual Machine: VM

## I. INTRODUCTION

Over the last several years, software containerization has become the industry standard for developers and researchers in the computer science (CS) field to implement and test software applications across multiple hardware and software configurations. Software containerization bundles the necessary files and software dependencies of an operating system (OS) and other applications into a centralized package. Software containerization is possible by utilizing containerization software, like Docker [1], and container orchestrating services, like Kubernetes [2]. Through this framework, containerized applications can be deployed and updated across the internet. From its inception, the workflow assumes a cloud-computing resource model. This architecture can scale the workflow to utilize additional hardware to provide the workflow with additional computational resources to aid in the computation. Some assumptions must be made in how to allocate these resources, but these assumptions typically are incompatible with traditional high-performance computing (HPC) workflows used by scientific computing applications. However, this architecture grants users the ability to deploy, test and decommission applications in different work environments quickly without leaving residual effects on the computational resources of the host systems.

A Docker container houses all the necessary components of an application and does not require software components to be installed individually on host systems, therefore providing end-users with massive productivity gains. Additionally, a container isn't computationally straining on its hosted system because of its intentionally scalable architecture. Therefore, containerization grants the end-user an environment for testing and developing of software applications across multiple combinations of hardware and software while still not exerting all of a single machine's computational resources to do so.

Students use multiple applications for their classes in the Middle Tennessee State University (MTSU) Computer Science Department. Technologies and languages, such as JupyterLab, Linux, Python, and C++, are all used throughout a student's tenure at the university. These applications may be used independently, but more complicated workflows, such as HPC applications, are challenging to install and configure on a host computer systems. Therefore, developing an orchestrated, container-based workflow for HPC applications would provide a significantly improved learning experience for students and teaching experience for faculty. In the chapters which follow, the development and testing of one of the initial components for providing students with a centralized, customizable, and lightweight HPC work environment for students through a cloud-based resource model is described. The primary goal of this thesis is to construct a Lightweight Directory Access Protocol (LDAP) server that could authenticate users as part of an orchestrated, containerized HPC workflow which packages all the necessary developmental tools for students. The LDAP serves as an authenticator for each containerized component of a scalable HPC cluster that the architecture is deployed on.

# II. BACKGROUND

*A. Virtual Machines*

In the simplest form, a virtual machine (VM) is merely a process (or program) running with special privileges on a host machine on top of a virtual disk image. With that in mind, a VM can be thought of as a virtualized computer. A VM is just like a physical computer because it needs an OS and other computational resources to function correctly. The VM is given a disk partition by the OS kernel on the host's device to provide the VM with the necessary operational requirements for storage, memory, and central processing unit (CPU) purposes. Virtualization is the technology that allows an application to divide computational resources on a physical machine among multiple VMs. Computational resources are either hardware or software components on a device that allow for a computer to run software applications. CPU usage, memory/storage usage, and network/internet usage are a few examples of computational resources. Lack of computational resources can be detrimental to the running and processing of software applications for a machine. Therefore, managing computational resources efficiently is pertinent in the computing world because mismanagement of resources can hinder an application's ability to run correctly and efficiently.

The hypervisor, also called virtual machine monitor (VMM), serves as the divider from the host's computational resources from the virtual machines' computational resources. Hypervisors create and run VMs by managing physical computational resources on the host's machine. The hypervisor can manage multiple VMs simultaneously on a single physical device [3].

Hypervisors must be installed on a physical device, such as a desktop or a server, therefore, taking up physical space on that host machine. The resource requirement for VMs orchestrated by hypervisors is a hindrance for the host machine and the VM itself because of the system's inability to be scaled to multiple physical devices. The hypervisor architecture is limited to a sole physical host machine, thus setting a physical limit to the computational resources that the VM can utilize to run software applications.

## *B. Cloud Clusters*

This section highlights the key components necessary to create a virtualized containerization cluster that is hosted as a service on a cloud. Docker, Kubernetes, and Helm work closely together to enable the cloud architecture for this thesis. Cloud computing technologies are workflows capable of sharing computational resources amongst worker nodes within the so-called cloud network across the internet. A cloud network is an IT infrastructure that allows nodes (physical host machines) to share data and computational resources amongst one another to create a more scalable and efficient work environment compared to its virtualization alternative.

*1) Docker*: Virtual containerization has gained widespread support in recent years because of services like Docker and Kubernetes. Docker is a software containerization service that creates containers from images configured through a Dockerfile. Docker describes a container as:

> A standard unit of software that packages up code and all its dependencies, so the application runs quickly and reliably from one computing environment to another. A Docker container image is a lightweight, standalone, executable package of

> software that includes everything needed to run an application: code, runtime, system tools, system libraries, and settings. [4]

Thus, containers through Docker provide an accessible medium for users to deploy applications in a standard, lightweight, and secure manner [4]. Docker containers allow users to create images on their machines to provide a test environment for different operating systems while still maintaining a non-computationally heavy workflow compared to VMs orchestrated by hypervisors. However, containers and virtual machines run by hypervisors differ in how scalable and portable each system is. While VMs and containers are similar in use-cases, their main differences lie in their effect on their host machines and their capabilities of hosting cloud computing technologies. A VM is an emulation of another, entirely different computer, virtually.  The VM packages the OS, the CPU, and the memory of its emulated device onto the physical hardware where it is being hosted. In comparison, a container just runs within the OS of another machine. Because of this crucial difference, the VM is a much more computationally expensive application than a container. Therefore, VM is less capable of maintaining and scaling cloud computing workflows compared to containers. For developers, the focus for their application is to test whether their application will function properly on various OSs; therefore, it is more beneficial and applicable to try that application through a container because of the lightweight, scalable characteristics that they possess. Figure 1 [5] visualizes how a traditional deployment, virtualized deployment, and a containerized deployment operate. As seen from Figure 1, traditional deployments build applications on an OS built upon the host machine’s hardware. When using the conventional deployment method to construct applications, there are limitations to how flexible and scalable the application can be because of the limited computational resources available to the machine that the application is built upon. Also, a traditional deployment does not possess a method to divide resources between an application and the physical device hosting the application. Traditional deployments typically do not function as efficiently for HPC workflows due to the physical limitations of computational resources available to the application. However, one key advantage for the traditional deployment is the ease of use for users to create simple applications designed for the specific architecture they are developing their application on.

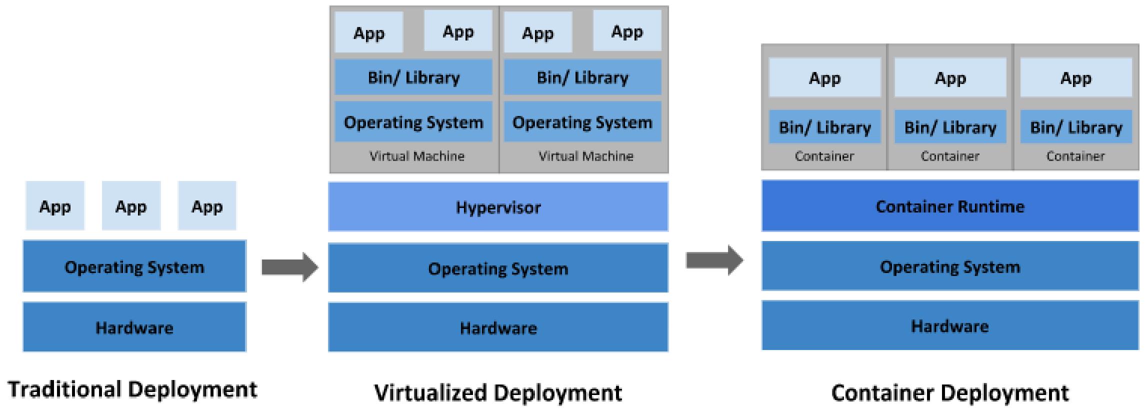

Figure 1: Comparison of Traditional Deployments, Virtualized Deployments, and Container Deployments

Source: Adapted with permission from [5]

In contrast to a traditional deployment, a virtualized deployment requires a hypervisor to manage the computational resources of the VMs that will run the applications that are being developed. Therefore, VMs are deployed onto a hypervisor so that the VMs can compute multiple applications in parallel to one another. Each VM utilizes the computational resources allocated to serve as a separate machine from the physical device. As seen in Figure 1, the hypervisor and VMs used for this architecture are built upon the host machine’s OS and hardware. Virtualized deployments are more efficient than traditional deployments due to the separation of different VMs that run multiple applications simultaneously. Through utilizing multiple VMs, the virtualized deployment is more capable of computing HPC workflows more than a traditional deployment. While a virtualized deployment is better suited to run HPC applications than a traditional deployment, one drawback is that the VMs and the hypervisor require computational resources to operate; therefore, hindering the resources readily available to the application being run.

A container deployment possesses a different combination of attributes between traditional deployments and virtualized deployments. As observed from Figure 1, a container deployment builds the containers that run applications directly onto the host machine’s OS and hardware, comparable to a traditional deployment. A containerized deployment utilizes containers constructed directly onto the OS to run HPC applications, similar to how virtualized deployments use VM to run HPC applications. Given that containerized deployments don’t require a hypervisor or VMs to compute HPC applications, they are the most suitable for running HPC applications compared to traditional and virtualized deployments due to their lightweight footprint on computational resources.

To use Docker, it must first be downloaded and installed onto the host machine. During that installation, Docker Desktop is downloaded and installed on the host machine as well. Docker Desktop is the centralized application needed to control the setting of Docker that runs on the host machine’s command line or terminal. Upon registering for an account through Docker, users are permitted to use Docker Hub. Docker Hub is an open-source cloud-based platform that stores images for containers. Through Docker Hub, users have a widespread open-sourced platform to pull and run different Docker images that other users create. Therefore, users are not necessarily required to create their own containers' images. Users can utilize Docker Hub to help develop their versions of container images that other users built on Docker Hub, pre-built images.

*2) Kubernetes:* Kubernetes is an orchestrator for numerous Docker containers. Kubernetes serves a wide range of use-cases for the architecture of this thesis. Load balancing, storage management, secret regulation, and the distribution and management of containers on the cluster are all different functionalities Kubernetes provides to a cluster.

Load balancing for a Kubernetes cluster is necessary to not overload a single node on the cluster with an application that may be too computationally expensive for a single node. The load-balancing characteristic of Kubernetes will distribute the workload of an application across all the nodes on a cluster to even the amount of computing for which each node is responsible. Therefore, each node performs an evenly distributed amount of computing to limit the probability of a single node failing due to being overworked. Mounting files for a cluster gives the cluster access to a file already created on a host machine, or it stores a file created on the cluster to the host machine to allow the host machine to access that mounted file or data.

A secret on a Kubernetes cluster is sensitive information that the system's administrator can only access and manage. The secrets on a cluster are used to provide the entire cluster with shared data that can be used for many different reasons. LDAP credentials, OAuth tokens, or SSH keys are all common uses for secrets on a cluster.

Kubernetes is responsible for the deployment of a cluster of containers. With this comes the responsibility to update, add, or delete containers when needed. It is Kubernetes' responsibility to ensure the cluster continues to operate when any one of these actions is required. Kubernetes allows for these actions to occur while maintaining the overall health of the cluster. Kubernetes does not need the complete disposal of the cluster to perform any one of these actions to a single container. However, suppose the desired action is to perform any one of these tasks to the entire cluster. In that case, a newer separate cluster will need to be deployed with the updated containers' image. Then the older cluster can be torn down and edited as desired.

There are multiple ways Kubernetes can be installed onto a host's system. One option is to use the Kubernetes client/server application called 'kind,' packaged into the Docker Desktop application.  To do this, download and install Docker onto a host's machine, and this will install Docker, Docker Desktop, and Kubernetes. Another option is to separately download and install Kubernetes as a standalone option called 'minikube.'

*3.) Helm:* Helm is a package manager for Kubernetes applications. Helm is responsible for the running and deployment of multiple Kubernetes .yaml files. Helm can be thought of as the orchestrator for multiple Kubernetes deployments, similar to how Kubernetes is the orchestrator for numerous Docker containers. Helm coordinates the installation of deployments (multi-container services) on a Kubernetes cluster. Because of Helm, installing, upgrading, and deleting Kubernetes applications can be achieved while maintaining the state of the overall Helm deployment if desired.  Helm can be installed onto a host's machine through scripts, the command line, or package managers, such as Chocolatey or Homebrew.

*C. High-Performance Computing Clusters*

Cloud architectures can potentially become HPC workflows by balancing computational resources more effectively than a virtualization architecture. An HPC cluster is multiple servers/applications/nodes that compute applications significantly quicker than traditional computing methods. An HPC cluster is broken down into three main components that make the cluster's functionality: computing, storage, and communication between nodes. HPC workflows are capable of hosting Message Passing Interface (MPI) applications. MPI applications are software that possesses the ability to utilize parallel processing. Parallel processing is leveraged inside an MPI application to enable the program to distribute work across computational resources to create more efficient software. Additionally, parallel processing allows a machine to utilize multiple cores on a CPU to compute a given application. Slurm, LDAP, network file system (NFS), and Singularity [6] are additional applications that can potentially be used by HPC environments.

*1.) Slurm:* Slurm is an open-source job scheduler used in HPC workflows because it allows users to submit a script to run across available compute node. A compute node grants administrative or non-administrative access to users using the workflow. Also, Slurm "Provides a framework for starting, executing, and monitoring work (normally a parallel job) on the set of allocated nodes" [7]. Typically, Slurm communicates through munge and then application communicate using MPI. Lastly, if multiple jobs need to be run, Slurm is responsible for queuing jobs to ensure that the jobs are computed in the first-in-first-out order. This serves as a computational resource manager for the nodes on the cluster. Queuing jobs on the workflow is essential so that the nodes on the system aren't overloaded with jobs at any given moment.

*2.) Lightweight Directory Access Protocol:* An LDAP server is a virtual protocol, similar to a database, that houses user information as an ordered structure for storage and retrieval. The LDAP is the centralized location for the storage of permissions and accesses for users on the cluster. An LDAP is necessary for a cluster to authenticate users to their accounts and for nodes to authenticate into the cluster. Typically, LDAPs are deployed to HPC clusters to manage users' accounts. For nodes to communicate with one another in a cluster, the workflow must have the means to authenticate nodes to the centralized system; therefore, an LDAP is essential for users of the cluster and is vital to the operation of the cluster itself. When a user or node attempts to authenticate into a system or website using specified credentials, the user or node will send a token request to the LDAP server; next, the LDAP authenticates the provided credentials with the directory's stored entries to give the user or node the appropriate accesses the cluster has assigned to them. Lastly, the response message is sent to the requester from the LDAP, either granting or denying service permission for the requested application.

*3.) Network File System:* A NFS is a client/server architecture that allows users to share files across a network. An NFS is an essential component of an HPC workflow because it grants users the ability to store files on the server and access those files as though they were stored locally to their host machine.  Also, an NFS authorizes the HPC workflow to share files and data between multiple users on the cluster. An NFS is responsible for making all or just a portion of the files on the server accessible to all users on the cluster or just a subsection of users on the cluster. For this reason, the NFS grants users the ability to have personal files that are only accessible to them.

*4.) Singularity:* Singularity is similar to Docker in the sense that both are containerization applications. However, Singularity allows for the creation of containers locally in user space. This means that, unlike Docker, users do not need administrator permissions on the system or cluster to deploy containers to the host machine (administration permissions are usually needed to create Docker images). Through Singularity, MPI is achievable on a single host's system with the assistance of Slurm to schedule jobs on the machine. Currently in industry, Singularity is utilized on physical HPC clusters; however, it is presently not integrated into the cloud computing architecture with Docker and/or Kubernetes.

## *D. High-Performance Cloud Computing Clusters*

Given that cloud computing applications can potentially host the components of HPC clusters and applications, it is now possible for the two paradigms to be combined to create a high-performance cloud computing cluster. While both cloud computing applications and HPC applications are current technology in the industry, the combination of the two allows for the workflow to be orchestrated as a cloud computing resource while still maintaining the MPI characteristics of an HPC cluster. Adding, deleting, and updating user information and nodes becomes attainable through the HPC applications being deployed across a cloud computing network. As a result, this creates a workflow that allows end-users to easily attain MPI applications while still granting the system administrator the controls needed to maintain the overall cluster for the framework.

# III. METHODOLOGY

## *A. Cloud Computing Team*

The 2020 S-STEM Summer Research Experience (SRE) was a four-week team-based internship program created at MTSU to provide students with an opportunity to develop unique skills that were not in the university's academic requirements. The program gives students a chance to become familiar with modernized technologies, e.g., Docker, Kubernetes, and Helm. Additionally, the experience teaches students how to collaborate with other teammates; also, the experience teaches students how to present one's progress on work in a professional environment. The internship began in mid-May of 2020 and continued through late July of 2020. According to the S-STEM Scholarship Program in Computer Science [8], "Working as a student on a summer research experience project will be a part-time appointment for two months (~150 hours total)".

For the internship, the "Cloud Computing Infrastructure" Research Team's (CCRT) goal was to offer students an opportunity to gain experience in cloud computing platforms capable of virtualizing applications. In addition, participants were to test and deploy virtualized applications of MTSU's current workflow to enhance the user experience for CS students at the university. The program's initial research began by creating a functioning MPI-enabled Kubernetes deployment that could run a containerized scientific application, Gromacs [9]. Gromacs is software used to simulate molecular dynamics and is widely used in the field of theoretical biophysics. The work conducted in this study demonstrated the positive capabilities that HPC can have in specific test environments. To illustrate the performance gain that an HPC cluster has to offer, each team member conducted computationally intensive tests for Gromacs simulations both with and without the utilization of HPC.

After completing the internship (September of 2020), the CCRT applied as a collective unit for an MTSU Undergraduate Research Experience and Creative Activity (URECA) Grant. The CCRT applied for the grant to continue the research they had previously established during the summer internship. The application process for the URECA Grant entails each student to write a two-page max essay that proposes the topic to be researched by the group and by the team members individually.

Upon the application's review from the URECA organization, the organization notified the CCRT that they had been awarded a Silver level URECA grant for the Fall of 2020 and the Spring of 2021. A Silver level URECA grant provides funds to each team member to conduct one hundred hours of research over the topic introduced during the application process per semester of research.

The CCRT's research topic for the URECA program was to deploy a container-within-container workflow that the university could eventually administer on their current infrastructure.  The goal of the CCRT's proposed research topic was to develop a cloud architecture that packages the developmental tools needed as a CS student at MTSU for HPC application development and testing. The CCRT divided the project into designated components to build the proposed architecture.

## *B. LDAP Development*

While other project components will be discussed in sections, the main contribution to the CCRT's research focus in this thesis was to develop a containerized

Lightweight Directory Access Protocol (LDAP) server over a Kubernetes cluster. The construction process began with researching and testing how to establish a base Docker container used to house the LDAP's base configuration. To create a Docker container, the user must download and install the Docker Desktop [10] application to use Docker on the command line or terminal window. Following installation, the user must carefully construct a Docker image to create a Docker container configured to the user's needs. A Docker image is a series of commands built by a Dockerfile. The image acts as the base configuration for the constructed Docker container.

The construction of a Kubernetes container is similar to how a user would create a Docker container. The user begins the container construction process by first downloading and installing Kubernetes. Next, the user must configure a .yaml file with the desired specification of the Kubernetes container. The configured .yaml file will serve as the configuration. Once the .yaml file is error-free, Kubernetes can deploy the Kubernetes container. Fortunately, the time spent researching how to create a base image in Kubernetes and Docker was relatively minimal to the scope of this thesis because of the prior knowledge gained from the 2020 S-STEM SRE. Therefore, during this period, the research for this thesis allocated most of the time spent in the research phase learning how an LDAP server works and how to configure a functioning LDAP for the CCRT's workflow properly.

There are several different approaches to building a containerized LDAP server. One option includes constructing both a Docker/Kubernetes image from scratch or without the assistance of previous work in the field, while another option is to employ a pre-built image. A pre-built image is a Docker/Kubernetes image developed by a company or an individual constructed to meet his or her needs for the container. Creating an image from scratch requires the researcher to identify the necessary components of a Dockerfile (for Docker). This process can be tedious and time-consuming to locate similarities in functioning LDAPs. Finding specific components to configure an LDAP isn't necessarily difficult because of the wide range of resources across the internet to help guide the process. However, the task can become significantly more challenging when building the configuration into an image due to how containers operate. The design of an LDAP server on a physical machine differs from that of an LDAP server configured to be inside a container. Traditionally, LDAPs created on a physical machine can vary from their containerized counterpart in a wide range of different ways. For example, handling firewall access and handling access rights for users are two separate ways in how the two architectures might operate differently. Another option to construct a containerized LDAP server for the project would be to utilize a pre-built image to fit into the scope of the project.

Initially, time was allocated to learning the high-level concepts of how an LDAP server is configured and functions for a workflow. A large portion of time was also allocated for identifying the necessary .yaml file configurations needed to build a functioning containerized LDAP server. A careful comparison and contrasting of different LDAP configurable values between functioning and nonfunctioning servers was performed. By identifying similarities in LDAP files, the required environment variables, and dependencies that the Kubernetes image needed to build a containerized LDAP server that functioned correctly could be understood. After this initial phase, it was determined that utilizing a configurable, pre-built image for deploying a containerized

LDAP server, along with slight modifications, could decrease the production time needed for the containerized server. So, by trial-and-error testing of different pre-built images, a functional proof of concept LDAP was created to authenticate with an Ubuntu client and verify authentication with a Jupyterlab client.  The bitnami/openldap Docker image by Bitnami [11] incorporates the base functionalities needed for the scope of the project. The decision to utilize Bitnami's image was decided upon quickly because of the customization and configuration capabilities that the image offers to system administrators. Once the pre-built Docker image was integrated, the next step was to configure a Kubernetes configuration file (.yaml) to control and use the base Docker image. The process to configure a Kubernetes image built upon a Docker image is similar to configuring a Docker image. The configuration of the Kubernetes image requires that the researcher configure the dependencies, hooks, and environment variables accordingly to achieve the desired objective. Alternatively, the CCRT's workflow could potentially utilize a pre-built Helm deployment as well but none was found to be currently available. The researcher initially began by locating the necessary components to build a Kubernetes image that would orchestrate Bitnami's Docker image. However, just like with a Docker image, locating the necessary parts can be tedious and monotonous, so the researcher concentrated on finding a pre-built Kubernetes image instead.

Campuzano's article from 2020 [12] teaches readers how to build a containerized OpenLDAP server using a pre-built Docker image on Kubernetes. The tutorial says "OpenLDAP is the open-source solution for LDAP (Lightweight Directory Access Protocol). It is a protocol used to store and retrieve data from a hierarchical directory structure such as in databases" [12]. Campuzano's tutorial was notably different from other works in the field because he successfully built an OpenLDAP system via Kubernetes, not just on a single physical machine. By creating the OpenLDAP server on a Kubernetes cluster, the system administrator can have the entire workflow be managed by a central across the cluster node while balancing the overall computational resource consumption of the service. Also, Kubernetes allows the system administrator to manage the containerized applications, i.e., an OpenLDAP server, as a cloud service. Through the steps in the article, a containerized LDAP server could now be deployed in the cloud. Multiple entries for user credentials were created for the server in the form of secrets. A secret is specific data on a Kubernetes cluster kept confidential and utilized by the entire cluster.

After employing Campuzano's tutorial, the next objective was to verify that the LDAP server could authenticate a user from a worker node in the cluster. Verifying user authentication requires the LDAP server to be deployed to the workflow's cluster before being instantiated by a client user. Also, the cluster's secret must be created before the instantiation of a user node.

First, the "Authentication and Authorization" documentation by Jupyterhub for Kubernetes [13] was used to construct a Jupyterhub client container that could authenticate as a user node LDAP server that runs from the workflows cluster. Jupterhub is a development platform that packages some of the software development tools that CS students at MTSU use in classes; therefore, Jupyterhub is an ideal indicator as to whether the LDAP was functioning correctly or not.  By utilizing the guide provided by Jupyerhub, the researcher constructed a Jupyterhub image. Once the Jupyterhub image is running, the IP and port number to the LDAP server must be configured. Next, a pop-up box will appear, then the user can enter authentication credentials (username and password) on the server. Once access is granted, it is clear that the containerized LDAP server is functioning correctly.

Once a JupyterHub client could authenticate as a user on the LDAP server, development of an Ubuntu [14] container that could authenticate as a user on the LDAP server was performed. The Ubuntu client serves as an alternate means of validation that the LDAP can adequately authenticate a user's credentials. Through two different validation methods, one can safely assume that the service does function as intended. So, to create a base Ubuntu image, Downey's guide to Creating a Simple Kubernetes Debug Pod [15] was followed. Once the base Ubuntu image was up and running, the following Ubuntu packages were downloaded: libnss-ldap, libpam-ldap, and ldap-utils. After downloading the necessary packages, the dependencies of the downloaded packages were directed to the location of the LDAP's cluster. All that is required once the packages are configured, the last step to validating an Ubuntu client could authenticate as a user on the LDAP server was to switch accounts from the terminal window. Upon inputting the client credentials of one of the entries on the LDAP, one can validate through two different methods that the LDAP is functioning correctly.

Each of the other members on the CCRT was responsible for developing their components of the workflow. Daniel Cox constructed the Slurm scheduler for the CCRT's project. The Slurm scheduler acts as the HPC job scheduler. The scheduler in an HPC workflow controls the computational resources utilized in the workflow to allow for MPI applications to run, such as Singularity. The Slurm scheduler was developed by configuring a Dockerfile with the necessary configurations for a SLURM scheduler.

Next, a base Docker image was created that a Kubernetes container would run on the Kubernetes cluster. Once functioning correctly, an MPI application is directed to the scheduler by a containerized Singularity service.

Hannah Williams developed the containerized version of Singularity in a Kubernetes container, similar to how Daniel developed the SLURM scheduler. The containers created inside the cluster would have message passing interface (MPI) capabilities; therefore, allowing containers to handle parallel processing for desired applications or jobs. During the developmental process of the CCRT's project, Daniel collaborated closely with Hannah to create a test environment to have a SLURM scheduler send an MPI application from a pre-configured Singularity image. The collaborative efforts between team members played a vital role in ensuring that their respective components of the CCRT's project were functioning correctly because the two applications work so closely with one another.

Jessica Wijaya designed the network file system (NFS) to aid in constructing the CCRT's project. A network file system is necessary for HPC clusters because they NFS allows nodes to share files amongst one another. Furthermore, an NFS server is responsible for managing files (updating, storing, or deleting) files inside a remote machine. The NFS allows the user to modify or store files located on the cluster. By providing the capabilities to store information from inside a container to the host system, the user can create a working version of their work and then keep the file on the host machine and know with certainty that the file works for the test environment.

Once each component in the team project was complete and functioning correctly, the only necessary step for completing the workflow was to merge the separate parts of the project. Each component (LDAP, SLURM, Singularity, and NFS) plays a vital role in the overall functionality of the CCRTs project. While each element of the project operates correctly separately, the CCRT's collaborative efforts created a more beneficial deployment that the university could utilize for its students with more fine-tuning and development.

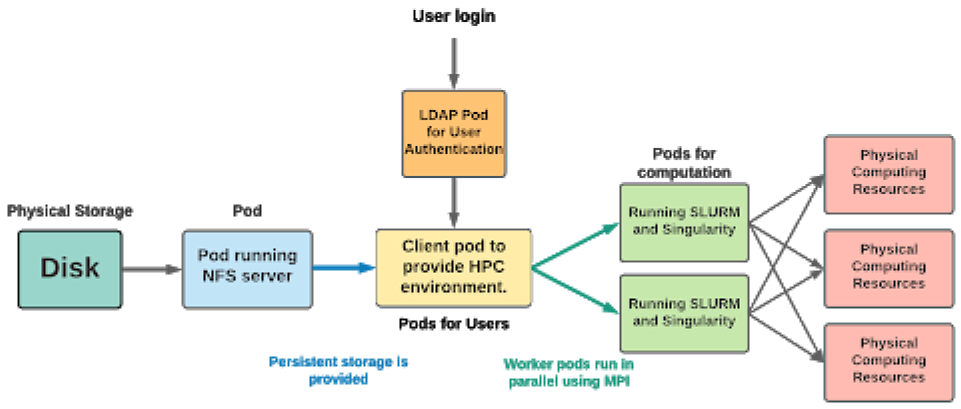
User login
LDAP Pod for User Authentication
Physical Storage
Disk
Pod
Pod running NFS server
Client pod to provide HPC environment.
Pods for Users
Pods for computation
Running SLURM and Singularity
Running SLURM and Singularity
Physical Computing Resources
Physical Computing Resources
Physical Computing Resources
Persistent storage is provided
Worker pods run in parallel using MPI

Figure 2: Schema of Workflow for CCRT's Project

The CCRTs project, when combined, can be visualized in Figure 2. As seen in the diagram, a user instantiates the HPC cluster by providing login credentials to the LDAP container. From there, the LDAP either grants or denies access to the rest of the cluster. Upon entering approved login credentials, the user is given a container with access to the files on the cluster and has storage capabilities due to the persistent storage provided by the NFS server. Additionally, users can utilize the Slurm and Singularity containers provided on the cluster to create worker nodes that can run parallel through MPI to compute HPC applications. The Slurm and Singularity containers are responsible for maintaining the physical computing resources needed to run applications on the cluster.

For the development of the CCRT's project, each team member was responsible for ensuring the other members of the team were informed with the progress that was made on their specific component to the project. To accomplish this, each team member created a Github repository. The team members' repositories acted as a version history for the work that they had done on their specific components. Additionally, the repository allowed the team members to share files between one another for the development and integration for the combining of individual components to create a centralized functioning prototype of the team's overall project. The README.md file in the team's shared repository teaches users how to build a simple Singularity/MPI application to run on the team's deployment. The Github repository for each team member as well as the CCRT's repository is listed below:

- Terryn Seaton - https://github.com/tseaton2/thesis
- Daniel Cox – https://github.com/danielJC06/Slurm_cluster
- Jessica Wijaya - https://github.com/jcwijaya/nfs-server
- Hannah Williams - https://github.com/hnw2y/HPC
- Joshua L. Phillips (faculty mentor) - https://github.com/mtsu-cs-summer-research/cloud-infrastructure-2020

# IV. RESULTS

The knowledge gained from the SRE proved invaluable for completing this thesis work. During this experience, the CCRT learned some of the base functionalities of the Docker and Kubernetes applications. In the early weeks of this program, the CCRT dedicated the most time to learning how Docker and Kubernetes operate. Additionally, the researchers discovered how Docker and Kubernetes aided the development process. Each team member spent multiple hours completing one Gromacs simulation workflow without HPC resources. In the following team meeting, the group discussed how their Gromacs simulation process went and how long each simulation took to complete. The consensus from the CCRT was that each test took multiple hours to add. During the meeting, the CCRT conducted the same simulations as previously tested, but now with the assistance of an HPC cluster. When performing the Gromacs simulations on an HPC cluster, each simulation that had once taken hours to complete without an HPC cluster can now take just a few minutes. The CCRT conducted all the same simulations as previously tested before during the length of the one-hour meeting that day. Additionally, the SRE program taught students how to collaborate to complete the desired project while doing so in a professional environment. For this work, the team members discussed and cooperated to conduct the tedious task of simulating molecular dynamics both with and without HPC technologies.

Initial attempts to create the LDAP container component of the project focused on the development of the required container image. This work did not necessarily produce a functional image but did provide insights into the relevant software configuration needed to deploy such a container. As such, a pre-built LDAP image was subsequently found to provide the required functionality for the purposes of the project. For example, full customization of all user account information is possible and necessary if accounts are to persist over time. However, since cloud-based deployments are often created, used, and then disposed, the need for persistent user account data is removed. Instead, these deployments only require a containerized authentication mechanism for all of the HPC-related containers in the deployment. The Bitnami pre-built images proved sufficient to meet these requirements. Still, all HPC containers need to also be configured to utilize this centralized resource and therefore most of the effort was spent on integration of the LDAP server with the other cluster components.  It would have been more beneficial to utilize an already configured Docker and Kubernetes container at the beginning of this research phase to increase the time available for this thesis implementation and testing. However, during that time, there was no way of knowing the challenges that would arise while creating a Docker and Kubernetes container from scratch. When creating new components for deployments, careful consideration of the needs and use-cases of the deployment can aid in understanding whether pre-built or custom images will suffice, but even then it can still be the case that certain considerations are overlooked and result in a change of plans.

While creating the containerized LDAP server without the assistance of a pre-built image during the implementation and testing phases of this thesis, multiple unforeseen challenges occurred. Because of this, there were several key takeaways during this time. A containerized LDAP is merely just a .yaml file that can be properly configured to host a server on a cloud architecture. With that said, to properly configure the container, the correct configurations must first be identified to create the bare minimum configurations of the server. The comparing and contrasting of different already configured LDAP can aid in the development of the containerized model. However, upon doing so, it can be found that the required base configurations of non-containerized LDAPs and containerized LDAP are not a one-to-one comparison because of differences in network infrastructure between the two architectures. Some environment variables that are required by the containerized LDAP were not required by its non-containerized counterpart.

Even though the base requirements for the containerized LDAP server and the non-containerized LDAP server weren't verbatim, the experience provided insight into how to use a pre-built image to suit the use-cases of the project. Through the testing of different configurations between the two models, a better understanding of how to set and configure the required components of the pre-built image was gained. Additionally, the experience gained through comparing the two architectures provided sound knowledge of what was being asked by the environment variables that needed to be set when using the pre-built image.

Another critical idea learned during the implementation and testing phase was that a host machines' OS could often affect the overall installation of the Docker application with the standalone Kubernetes service.  While testing, Kubernetes would at times be inconsistent with its service. There were multiple instances where Kubernetes would completely stop functioning. During these instances, Kubernetes would provide very minimal output to help debug the issue. The most common bug with the Kubernetes service typically made itself evident after a Kubernetes container had already been created and destroyed. The problem would occur after the destruction of one container and creating a second Kubernetes container. Upon doing so, Kubernetes would create the container appropriately; however, when trying to connect to the internet in the container, connection issues would begin to arise for the container. This prevented the Kubernetes container from updating and retrieving the necessary information to remain up to date. Once the container was created and could not be corrected, several hours might be spent debugging different workflow components. Eventually, after testing the individual parts of the workflow, completely uninstalling, and reinstalling the Docker application with the Kubernetes standalone service was necessary. After that, the Kubernetes container was restarted to find that the container was fully functioning after removing the Docker application from the host machine. While the issues seemed like they were corrected, the same problems continued to arise during the implementation and testing phases. However, when the bugs reappeared, additional testing was conducted on the Docker Desktop application to find any glaring issues. Through this, it was determined that the Docker component of the application was functioning correctly. The Kubernetes standalone service for the Docker Desktop, through further testing, was determined to be the culprit for what had been causing the connection issues for the containers. Unfortunately, the problem with the Kubernetes standalone server/client could not be resolved.

A key benefit to of this work is that MTSU's Computer Science Department now has the foundation of a workflow that will bundle all of the HPC developmental tools a student will need into one package. While the HPC workflow isn't production-ready, the groundwork of the project has been completed by this thesis and the CCRT's research project. Currently, the workflow can authenticate a user’s account through a Jupyterhub client or an Ubuntu client. Meaning, a user can authenticate with their credentials to their account on the school's LDAP server by deploying an Ubuntu Kubernetes container or by deploying a Jupyterhub client container. Upon submitting their credentials, the LDAP will verify that the user's credentials are correct and then either grant the user access to their specific permissions on the server or deny the user access because of an invalid username or password. Also, the HPC cluster can be managed and orchestrated through Docker/Kubernetes as a cloud-based architecture. Additionally, the HPC cluster allows for users to create personalized work environments to test and run HPC applications on the cluster. Also, users can edit, save, or share files stored in the HPC cluster.

# V. DISCUSSION

A containerized, cloud-based LDAP server can serve multiple functions. One important use case is allowing users to authenticate their credentials through the server to log into the services hosted on a HPC Kubernetes deployment. Additionally, the LDAP server acts as the centralized authenticator for containers and services in the HPC cluster to facilitate HPC applications. The work described above contributes to the development of a Kubernetes workflow that users can deploy, quickly and efficiently, to create a complete HPC environment for debugging and testing parallel applications across various hardware and software combinations.

The results indicate progress made toward developing a workflow that the university can utilize to provide students and faculty with a more streamlined approach for deploying the necessary HPC development environment and tools throughout their time at the university. While the current workflow developed by the Cloud Infrastructure CCRT doesn’t provide a production model of the desired workflow, the current workflow does serve as a functional prototype for the desired end goal. As a result, it can serve as a good foundation for continued development work to provide students and faculty with a more efficient and effective way to manage their developmental tools. For example, the work will allow faculty in the department to spend less time going over the software installation and setups in their classrooms. This is accomplished by the development tools being all packaged into stand-alone, orchestrated deployments. If students are taught how to use the packaging applications (Docker/Kubernetes) at the beginning of their academic career at the university then potentially any such deployments may be deployed using the same skills as for all others. After the initial learning phase of the application, students would be free to obtain their necessary development tools through the application.

The current model built by the CCRT hasn’t been extensively tested to rid the workflow of any security holes. Therefore, in the future, additional testing will be needed to provide a secure workflow for the university. Also, while all the components of the desired end goal have been constructed, the individual HPC applications that will be ran on the system will need to be stored in an accessible repository to provide all the packaged applications to students in a centralized location.

During the development process, the CCRT faced numerous issues surrounding the Docker/Kubernetes applications with regards to how the deployment ran on their personal machines. Oftentimes, the application would break and supply little to no feedback as to what was causing the issues. These issues were persistent throughout the project. The only consistent solution for these issues was the uninstalling and reinstalling of the Docker/Kubernetes application. With that in mind, future work surrounding these issues could be addressed. While the issue is potentially due to variations in different operating systems, a consistent solution will need to be discovered to provide these HPC workflow models with a more reliable and stable version of themselves.

# REFERENCES

[1] “Empowering App Development for Developers,” Docker. https://www.docker.com/ (accessed June 10, 2020).

[2] “Production-Grade Container Orchestration,” Kubernetes. https://kubernetes.io/ (accessed January 23, 2021).

[3] “What is a Hypervisor?” Red Hat. https://www.redhat.com/en/topics/virtualization/what-is-a-hypervisor (accessed July 1, 2021).

[4] “What is a Container?” Docker. https://www.docker.com/resources/what-container (accessed June 19, 2020).

[5] “What is Kubernetes?” Kubernetes. https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/ (accessed October 10, 2020).

[6] “Singularity,” Singularity.io. https://sylabs.io/singularity/ (accessed May 17, 2021).

[7] “Slurm Workload Manager,” SchedMD. https://slurm.schedmd.com/overview.html (accessed May 14, 2021).

[8] Z. Dong.“S-STEM Scholarship Program in Computer Science,” Middle Tennessee State University. https://www.cs.mtsu.edu/~s-stem/research.php (accessed May 23, 2020).

[9] “Gromacs,” Gromacs. https://www.gromacs.org/ (accessed June 17, 2020).

[10] “Docker Desktop,” Docker. https://www.docker.com/products/docker-desktop (accessed June 15, 2020).

[11] Bitnami. “bitnami/openldap,” Docker Hub. https://hub.docker.com/r/bitnami/openldap (accessed March 10, 2021).

[12] R. Campuzano. “Create An OpenLDAP server with Bitnami Containers on Kubernetes,” Bitnami by VMWare. https://docs.bitnami.com/tutorials/create-openldap-server-kubernetes/ (accessed March 10, 2021).

[13] “Authentication and Authorization,” Jupyterhub for Kubernetes. https://zero-to-jupyterhub.readthedocs.io/en/latest/administrator/authentication.html (accessed April 2, 2021).

[14] “Ubuntu,” Ubuntu. https://ubuntu.com/ (accessed April 12, 2021).

[15] T. Downey. “Creating a Simple Kubernetes Debug Pod,” Downey. https://downey.io/notes/dev/ubuntu-sleep-pod-yaml/ (accessed April 12, 2021).