Ratnadeep Rajendra Kharade, Hadi Adineh and Dieter Uckelmann

Comparing Service-Oriented System Management Solutions in Remote and Virtual Laboratory Environments

Abstract

Digitalized laboratories are gaining importance in the higher education sector. Students are being provided with remote access to physical laboratory infrastructures as well as online access to virtual labs. Due to the complexity of systems in digital laboratory environments, it is often difficult to manage the applications efficiently. Moreover, there can be multiple types of laboratories with different system configurations. These laboratories need different management solutions based on the heterogeneity of lab systems. Therefore, different approaches are needed to create deployable software units which support multiple architectures.

We compare a microservices approach and monolithic architectures. As regards production deployment, virtualization and containerization along with their benefits and disadvantages are considered. In our research, we compared Docker solutions as well as the main Kubernetes tools like Minikube, Kubeadm, K3S, and Microk8s. Our goal is to identify solutions that are easy to manage even in heterogeneous hardware environments. Security, high availability, and compatibility with digitalized laboratories are also considered.

Keywords

Remote Laboratories, Digitalization, Microservices

1 Introduction

Digitalization is rapidly changing the world and many sectors are benefitting from it (Rodriguez-Andina et al., 2010). Digitalization in education is also being adopted in this wave. In lab-based education, digitalized laboratories, like remote laboratories and virtual laboratories are gaining more and

more importance. Corresponding applications vary from basic web-based dashboards to highly domain-specific software (Taivalsaari & Mikkonen, 2018). Those applications can be designed based on monolithic architectures or microservices approaches.

University labs are using a wide variety of hardware components. Some of the laboratories are specially designed for specific requirement. Unfortunately, hardware heterogeneity increases the complexity of system design and management. However, modern system management solutions can handle different hardware architectures, e.g., servers with Intel or AMD processors or IoT-compatible devices such as Raspberry PI with ARM architecture.

Considering monolithic architectures as well as microservices approaches in homogeneous and heterogeneous environments, the aim of this research is to come up with a suitable solution based on the system requirements in remote and virtual laboratories. Our findings will be beneficial for other digital laboratories.

As part of the Open Digital Lab 4 You (DigiLab4U) project (Pfeiffer & Uckelmann, 2019) and the concept of research based education, the requirements mentioned here have been researched by senior and student researchers (Kharade, 2021) at the University of Applied Sciences Stuttgart (HFT Stuttgart).

2 Background

Various architectures ranging from monolithic to microservices are compared along with their benefits in this research. The benefit of monolithic architecture is that its development can be faster in the initial phases (Kalske et al., 2017). However, the complexity of hardware architecture and problems arising due to an increase in codebase size could be mentioned as two major challenges with monolithic architecture. This poses challenges in the updating and scaling of application. On the other hand, microservices are small software units that run independently and have a single responsibility. Since microservices are loosely coupled, application scaling and deployments can be carried out independently (Kalske et al., 2017).

Traditionally, the deployment of the application used to be done directly on the underlying infrastructure, the host OS. Virtualization enables the creation of isolated virtual machines on a single hardware system. In contrast, containers virtualize only the file system, whereas VMs virtualize the entire operating system. A container engine is basically a software piece that takes requests from users, including options from the command line,

pulls images, and executes the container from the perspective of the end user (McCarty, 2018). Compared to other container engines (e.g., RKT, CRIO and LXD), Docker provides additional features such as building the images and signing (Baker, 2020). Therefore, in this research for a laboratory environment, Docker is used as one of the use cases for container runtime. By taking advantage of Docker's methodologies for shipping, testing, and deploying code quickly, the delay between writing code and running it in production can be significantly reduced (Official Docker Documentation).

Docker Swarm, Apache Mesos, and Kubernetes are some of the popular Container Orchestration Tools. Jawarneh et al. (Al Jawarneh et al., 2019) has done a comprehensive comparison of these tools based on major functionality groups such as resource management, scheduling, and service management.

Kubernetes (Kubernetes.io, 2020) is an open-source, extendable, portable platform for managing containerized workloads and services, aiding both automation and declarative configuration. It has a vast and rapidly evolving ecosystem. Kubernetes services, support, and tools are commonly obtainable. For this research, Kubernetes is further evaluated in terms of managing applications in a laboratory environment, and the recent container orchestration tools are compared and evaluated.

3 Software Systems in a laboratory environment

There are different architectures for a software system, from traditional monolithic architectures to the modern microservices architecture. Laboratory environments can be different in terms of hardware and software complexity and number of devices. So, each laboratory environment can have different requirements in terms of system architecture.

Similarly, based on laboratory requirements, system environments can also differ. There can be traditional servers or virtual machines or use of containerization.

3.1 System Architecture

A typical software system consists of various components such as web user interface, back end, and database application.

Traditionally, software systems were designed using a Monolithic architecture. It provides a unified model in context to software design.

3.1.1 Monolithic Architecture and its Challenges

Monolithic software is constructed as one unit. Monolithic software is structured to be self-contained; elements of the package are connected and are dependent on each other. These packages have a high coupling between them. To execute the code, all components must be always available in a tightly coupled system. The benefit of monolithic architectures is that their development can be faster in their initial phases (Kalske et al., 2017). Monolithic architectures are best suited for laboratories which have very few hardware and software components and do not need frequent upgrades.

Even though monolithic types of software are simple and straightforward to develop, they have some downsides. There are two major categories of challenges with monolithic architecture which are relevant for the software systems in a complex laboratory or IoT setup.

a. Challenges with Hardware Architecture Complexity

Laboratory environments may consist of different devices which have various applications. These applications can be independent of each other and are distributed throughout the infrastructure. For example, a digital laboratory can have some applications which enable robotic movements using a Raspberry Pi, while the UI application and database applications run on high-end servers. Moreover, suppose that a lab needs to run other applications requiring high performance computing, virtual reality, artificial intelligence, or machine learning algorithms as well as data processing on GPUs. Since monolithic architectures are tightly coupled in nature, they are not particularly useful in IoT environments, where multiple applications run independently.

Also, different devices in an IoT infrastructure need different programming languages based on the functionality they need to achieve. This makes monolithic architectures unsuitable for deployment in complex laboratory environments as they are mostly based on the same programming context. For example, monolithic software based on tomcat server hosts uses interface, business logic as well database access in the same environment.

b. Challenges posed by the Software Development Process

The challenges with monolithic architectures escalate as the codebase increases in size. It is more difficult to incorporate new features and improvements to existing features as the developer must find the right place to apply changes (Kalske et al., 2017).

Along with that, whenever a change is required in the software, the whole software is affected and needs to be redeployed, which increases the downtime of software even though it is not required for other functionalities.

3.1.2 Microservices Approach

Microservices are small software units that run independently and have a single responsibility, as shown in Figure 1. Microservices are loosely coupled in nature and focus on one utility. Loose coupling enables developers to make individual changes to microservices without impacting the rest of the codebase. Since microservices are not connected to each other, application scaling and deployments can be carried out independently (Kalske et al., 2017).

Complex laboratory environments can benefit by incorporating microservices architecture. Complex laboratory systems consist of independent software units which have varied purposes, such as data collections from sensors, actuation, data processing, and user interface. These functionalities work independently. Also, platform-dependent microservices can be developed based on hardware architecture, for example software based on ARM architecture for Raspberry Pi and AMD for software based on intel servers. Development and scaling of these applications can be done independently without affecting other applications. This enables the development of IoT systems with wide availability.

3.2 System Environment

Traditionally, the deployment of the applications used to be done directly on the underlying infrastructure. In this approach, the applications are deployed directly on the host Operating System (OS) and access the system resources directly via host OS processes. Multiple applications are installed on an OS. These applications are exposed using ports on the host operating system.

3.2.1 Virtualization

Virtualization enables the creation of isolated Virtual Machines (VMs) on a single hardware system. These virtual machines have their own operating systems, also referred to as guest operating systems. This approach isolates software applications inside VM and limits resource usage per application.

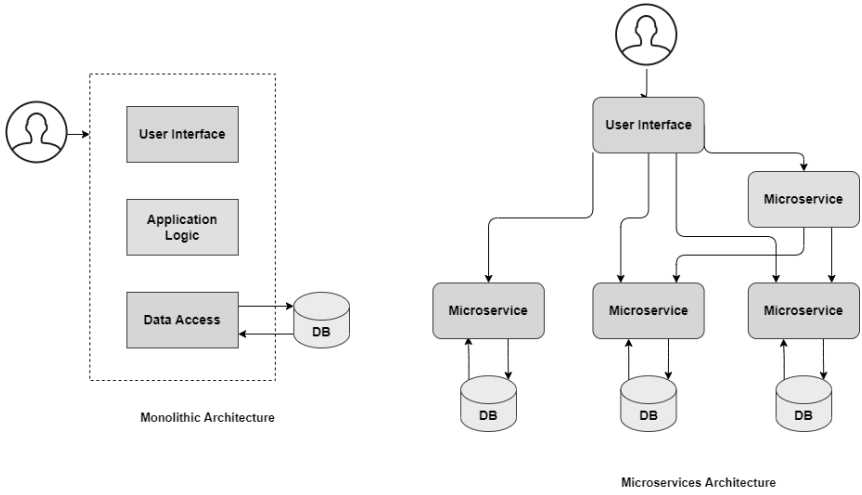

Figure 1: Monolithic vs. Microservices architectures

System encompassing virtualization has components such as Host OS, hypervisor, and VM, as displayed in Figure 2.

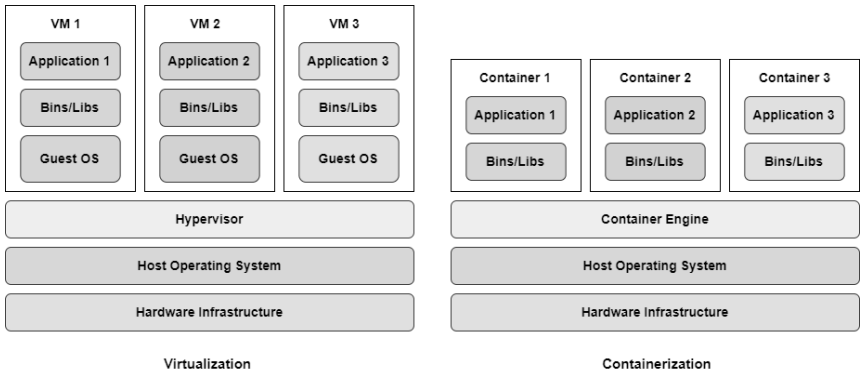

Figure 2: Virtualization and Containerization

The benefit of VMs is that, in a laboratory environment, virtualized deployments can be done on servers with greater processing power and memory.

So VMs are suitable for applications that are resource-intensive. Also, VMs are more secure because applications are deployed in the Guest OS.

The challenge with virtualization is that it is not suitable on edge devices as the guest OS on VM is itself heavyweight and a large chunk of the resources must be assigned to VMs. So VMs are not suitable in labs with low-end servers and applications needing few resources.

3.2.2 Containerization

The term containerization originates from shipping containers, where all goods are packaged within containers and shipped across the world. Software containers are used to pack software along with its dependencies. Software containers provide an isolated environment for an application which also contains the required packages, dependencies, and libraries, as displayed in Figure 2.

Software containers are platform independent. Containers try to solve the dev-ops problem through abstraction and platform independence in various environments such as 'development' and 'production.'

A container is a runtime instance of an image. The image is a blueprint of a container which is never running. An image contains file systems and source codes. Many containers can be spawned from the same image.

3.2.3 Containers vs. VMs

Containers virtualize only the file system, whereas VMs virtualize entire operating systems. Containers share the kernel with the host OS. VMs create a new virtual kernel. Containers require a lower amount of storage, are lightweight, and take less time to boot up. On the other hand, VMs need mode storage as the OS and programs are not only installed and run, but are also heavyweight and even take more time to boot.

In a complex laboratory environment with multiple edge devices, containers are a better option than VMs as edge devices require fewer system resources.

3.2.4 Container engines

A container engine is basically a software piece that takes requests from users, including options from the command line, pulls images, and executes the container from the perspective of the end user (McCarty, 2018).

Many container engines are available for running containers, such as Docker, RKT, CRI-O, and LXD. Along with that, various cloud providers such as Google GCP, Microsoft Azure, and Amazon AWS have their own

container engines, which utilize container images compliant with Docker or the open container initiative (McCarty, 2018).

Compared to other container engines, Docker provides additional features such as building the images and signing (Baker, 2020). Therefore, in this paper for a laboratory environment we used Docker as one of the use cases for container runtime.

3.2.5 Docker

Docker is a popular container technology provider. Using Docker, it is easier to create, deploy, and run applications. Docker is an open platform for developing, shipping, and running applications. Docker enables separation of applications from infrastructure, so software can be delivered fast. By taking advantage of Docker's methodologies for shipping, testing, and deploying code quickly, the delay between writing code and running it in production can be significantly reduced.

3.3 Container Orchestration

Automatic container deployments, management, scaling, and networking are achieved through the Container Orchestration process. It is feasible to deploy the same application in different environments and without any architecture modifications (RedHat, 2019). Along with managing a container's life cycle, container orchestration tools also support the integration of continuous integration and continuous deployment workflows.

A container orchestration tool's major tasks are the management of system resources, scheduling of applications, and management of services (Al Jawarneh et al., 2019). Container orchestration tools manage the underlying infrastructure for setting up the applications and provisions communications between them even though they are distributed.

Although containers are a better solution, it gets difficult to manage applications when there are many servers distributed in a laboratory environment, where there are distributed servers. Managing and networking challenges can be overcome using container orchestration.

3.3.1 Container Orchestration Tools

Over the last few years, many container orchestration platforms have been developed in the industry. Although they all perform the same simple container automation task, they run in various ways and have been developed

for various usage scenarios. The most popular tools are Kubernetes (k8s), Docker Swarm, and Apache Mesos.

These tools are compared by Jawarneh et al. (Al Jawarneh et al., 2019), as shown in Table 1 and Table 2, and we found out that Kubernetes is best suited to a laboratory environment.

Table 1: Comparison of container orchestration tools based on resource utilization (Al Jawarneh et al., 2019)

Functionality	Docker Swarm	Apache Mesos	Kubernetes
Memory	yes	yes	yes
CPU	yes	yes	yes
GPU	no	partial	no
Disk Space	no	yes	no
Volume	yes	yes	yes
Persistent Volume	no	partial	partial
Port	yes	yes	yes
IP	partial	partial	partial

Table 2: Comparison of container orchestration tools based on features (Al Jawarneh et al., 2019)

Functionality	Docker Swarm	Apache Mesos	Kubernetes
Memory	yes	yes	yes
CPU	yes	yes	yes
GPU	no	partial	no
Disk Space	no	yes	no
Volume	yes	yes	yes
Persistent Volume	no	partial	partial
Port	yes	yes	yes
IP	partial	partial	partial

3.3.2 Kubernetes

Kubernetes is an open-source, extendable, portable platform for managing containerized workloads and services, aiding both automation and declarative configuration. It has a vast, speedily evolving ecosystem. Kubernetes' services, support, and tools are commonly obtainable.

The platform generally referred to as K8s or Kube has started taking the place of Development and Operations (DevOps) in recent years by helping developers to speed up the coding along with best practices, also to speed up deployments, automated testing, and updates. Moreover, working on Kube helps developers to manage apps as well as services with almost zero downtime. Also providing self-healing abilities, Kubernetes has the ability to detect and restart services if a process fails inside a container (Kubernetes.io, 2020).

To develop scalable and portable application deployments that can be managed, scheduled, and maintained easily, it's easy to notice why it's becoming the go-to technology of preference. Kubernetes can be integrated with all of the top free cloud offerings and can be used on-site in a corporate data hub. Its features, like cross-functionality and heterogeneous cloud support, are the reason behind making this platform standard in container orchestration (Kubernetes.io, 2020).

3.3.3 Microk8s

Microk8s is Kubernetes distribution developed by Canonical. Microk8s supports a single node as well as a multi-node Kubernetes cluster in the production environment. Microk8s is also a minimal version of Kubernetes and is suited to all kinds of servers, such as high-end workstations to IoT devices. Microk8s' installation is simple. Microk8s supports all popular add-ons, which are disabled initially and can be enabled when required. This makes Microk8s lightweight and suitable for any environment.

3.3.4 Suitable Kubernetes for a Laboratory Environment

The laboratory environment is a combination of multiple devices with different architectures, such as ARM and AMD. Also, there are varying resources per device. For example, devices such as Raspberry Pi have less storage, memory, and CPU than high-end workstations such as Intel based CPUs. Due to these parameters, it becomes challenging to come up with proper tools in such a heterogeneous environment. Also, ease of setup and available features are important parameters.

Based on the comparison of the tools, Microk8s and K3S are best suited for the laboratory environment and support multiple architectures.

4 Discussion and Sample Scenarios

Each digital laboratory has its own requirements and criteria as well as different hardware and software configurations. Some of them are used for performing experiments by accessing hardware remotely, while others are purely virtual ones. Based on these features the preferred solution for each laboratory can be chosen. These solutions could be adopted with personal computers, high-end servers or even lower-end Single-Board Computers (SBC) such as Raspberry PIs. Although there is no unique solution for all digital laboratories, this paper can help them to find the suitable solution. For example, here are some scenarios and solutions.

Scenario 1: To digitalize the RFID laboratory at the University of Applied Sciences Stuttgart (HFT Stuttgart) in order to enable remote experiments, as discussed in (Pfeiffer et al., 2022), there is a RFID measuring device which is designed to be operated remotely. The device controller software needs to be run during the remote experiment runtime and part of its Graphical User Interface (GUI) is intended to be shared with remote users. Thus, we could only run this program on a real or virtualized Microsoft Windows operating system, because this GUI dependency makes it too sophisticated to be containerized.

Scenario 2: In the second scenario, we found that the DigiLab4U Laboratory Management System (LabMS) (Adineh et al., 2022) is fully compatible with containerization. A LabMS is an application with which to manage laboratory instruments, as well as enhance remote operation while cooperating with DigiLab4U shared services. LabMS applications have been mainly developed based on DigiLab4U LabMS libraries and are potentially adoptable in dockerization scenarios.

Scenario 3: Consider that there is a virtual laboratory through which to execute and evaluate Artificial Intelligence (AI), Machine Learning (ML), or data analytics algorithms. These algorithms consume high performance computing resources as well as Graphical Processing Units (GPUs) on the server's side. As shown in other research, like that by Xu et al. (2017), by dockerizing part of GPUs' computation, not only could the benefits of containerization be significantly minimized, but in some cases so could the overheads.

5 Conclusions and Future works

In this paper, we prepared different solutions for system management in digital laboratories (remote and virtual laboratories) which have a different hardware and software complexity. The results show that when there is one application or service to be run, it is suitable for deployment on a single computing machine (e.g., PC, server, Raspberry Pi) as a normal running application. If there are multiple services to be run in one machine, implementing the Docker solution enables us to have more control over application management. Moreover, with the help of Kubernetes, laboratories with multiple machines in a distributed system can be managed more efficiently. Based on our findings, Kubernetes distribution Microk8s can handle multiple applications gracefully in laboratory environments with heterogeneous computing machines (PCs and Raspberry PIs). Mikrok8s is also easy to use in a laboratory environment as compared to other Kubernetes distributions.

As regards future work, evaluating the results of this research in a purely virtual laboratory as well as a virtual reality provider is considered. Moreover, this research could be continued by providing a Kubernetes-based solution with which to manage all DigiLab4U services. Finally, we want to evaluate the architecture mentioned in a real laboratory and compare the results in terms of deployment speed, failure recovery, and security.

Acknowledgements

The DigiLab4U project, on which this paper is based, was funded by the Federal Ministry of Education and Research (BMBF), Germany under the funding code 16DHB2112. The responsibility for the content of this publication lies with the authors.

References

Adineh, H., Galli, M., Heinemann, B., Höhner, N., Mezzogori, D., Ehlenz, M., & Uckelmann, D. (2022). Challenges and Solutions to Integrate Remote Laboratories in a Cross-University Network. In M. E. Auer, K. R. Bhimavaram, & X.-G. Yue (eds.), *Lecture Notes in Networks and Systems. Online Engineering and Society 4.0* (vol. 298, pp. 189–202). Springer International Publishing. https://doi.org/10.1007/978-3-030-82529-4_19

Al Jawarneh, I. M., Bellavista, P., Bosi, F., Foschini, L., Martuscelli, G., Montanari, R., & Palopoli, A. (2019). Container orchestration engines: A thorough functional and performance comparison. In *ICC 2019-2019 IEEE International Conference on Communications (ICC)*. Symposium conducted at the meeting of IEEE.

Baker, E. (2020). *A Comprehensive Container Runtime Comparison*. https://www.capitalone.com/tech/cloud/container-runtime/

Kalske, M., Mäkitalo, N., & Mikkonen, T. (2017). Challenges when moving from monolith to microservice architecture. In *International Conference on Web Engineering*. Symposium conducted at the meeting of Springer.

Kharade, R. R. (2021). *System Management, Recovery and Security in Laboratory Environment*. Stuttgart University of Applied Sciences (HFT Stuttgart), Stuttgart, Germany.

Kubernetes.io. (2020). *Kubernetes*. https://kubernetes.io/docs/home/

McCarty, S. (2018). *A Practical Introduction to Container Terminology*. https://developers.redhat.com/blog/2018/02/22/container-terminologypractical-introduction/

Official Docker Documentation. *Docker overview*. https://docs.docker.com/get-started/overview/

Pfeiffer, A., Adineh, H., & Uckelmann, D. (2022). Aligning Technic with Didactic — A Remote Laboratory Infrastructure for Study, Teaching and Research. In M. E. Auer, K. R. Bhimavaram, & X.-G. Yue (eds.), *Lecture Notes in Networks and Systems. Online Engineering and Society 4.0* (vol. 298, pp. 78–86). Springer International Publishing. https://doi.org/10.1007/978-3-030-82529-4_8

RedHat. (2019). *What is container orchestration?* https://www.redhat.com/en/topics/containers/what-is-container-orchestration

Rodriguez-Andina, J. J., Gomes, L., & Bogosyan, S. (2010). Current Trends in Industrial Electronics Education. *IEEE Transactions on Industrial Electronics*, *10*(57), 3245–3252.

Taivalsaari, A., & Mikkonen, T. (2018). A taxonomy of IoT client architectures. *IEEE Software*, *35*(3), 83–88.

Xu, P., Shi, S., & Chu, X. (2017). Performance Evaluation of Deep Learning Tools in Docker Containers. In *2017 3rd International Conference on Big Data Computing and Communications (BIGCOM)*. IEEE. https://doi.org/10.1109/bigcom.2017.32

Authors

Ratnadeep Rajendra Kharade
HFT Stuttgart
Schellingstraße 24
70174 Stuttgart
https://de.linkedin.com/in/ratnadeep-rajendra-kharade-a0597a18
ratnadeep.kharade@outlook.com

Hadi AdinehDINEH
HFT Stuttgart
Schellingstr. 24
70174 Stuttgart
https://www.hft-stuttgart.de/p/hadi-adineh
hadi.adineh@hft-stuttgart.de

Prof. Dr.-Ing. Dieter Uckelmann
HFT Stuttgart
Schellingstr. 24
70174 Stuttgart
https://www.hft-stuttgart.de/p/dieter-uckelmann
dieter.uckelmann@hft-stuttgart.de