Bainomugisha, Engineer; Mwotil, Alex

Article

Crane Cloud: A resilient multi-cloud service abstraction layer for resource-constrained settings

Development Engineering

Provided in Cooperation with:
Elsevier

Suggested Citation: Bainomugisha, Engineer; Mwotil, Alex (2022) : Crane Cloud: A resilient multi-cloud service abstraction layer for resource-constrained settings, Development Engineering, ISSN 2352-7285, Elsevier, Amsterdam, Vol. 7, pp. 1-18, https://doi.org/10.1016/j.deveng.2022.100102

This Version is available at:
https://hdl.handle.net/10419/299118

Development Engineering 7 (2022) 100102

Contents lists available at ScienceDirect

Development Engineering

journal homepage: www.elsevier.com/locate/deveng

Crane Cloud: A resilient multi-cloud service abstraction layer for resource-constrained settings

Engineer Bainomugisha *, Alex Mwotil *

Department of Computer Science, College of Computing & Information Sciences, Makerere University, Kampala, 7062, Uganda

ARTICLE INFO

Keywords:
Microservices
Kubernetes
Containers
Orchestration
Low-resource settings
Portable cloud apps
Cloud native platforms

ABSTRACT

Developers and users situated in low-resource settings are faced with unique contextual and infrastructure challenges when accessing and consuming cloud-based services. In low-resource settings, access to cloud services and platforms is usually characterized by low-end computing devices and often unreliable and slow mobile broadband Internet connections. In this paper, we discuss key challenges for developing for and accessing cloud services in resource constrained settings, namely, (1) Frequent Internet partitions and bandwidth constraints, (2) Data jurisdiction restrictions, (3) Vendor lock-in, and (4) Poor quality of service. Inspired by these challenges, we propose a set of important design considerations and properties for a resilient multi-cloud service layer, that includes: (1) Containerization and orchestration of applications, (2) Application placement and replication, (3) Portability and multi-cloud migration, (4) Resilience to network partitions and bandwidth constraints, (5) Automated service discovery and load balancing, (6) Localized image registry, and (7) Support for platform monitoring and management. We present an implementation and validation case study, Crane Cloud, an open source multi-cloud service abstraction layer built on-top of Kubernetes that is designed with inherent support for resilience to network partitions, microservice orchestration (deployment, scaling and management of containerized applications), a localized image registry, support for migration of services between private and public clouds to avoid vendor lock-in issues and platform monitoring. We evaluate the performance and user experience of Crane Cloud by implementing and deploying a computational and bandwidth intensive machine learning system. The results show lower response times of the system on Crane Cloud compared with hosting on other public clouds. The Crane Cloud platform is serving as a cloud-service for students and developers in low-resource settings and also as an education platform for cloud computing.

1. Introduction

Cloud computing is now a popular model of delivering computing services over the Internet ("the cloud") with flexible pricing models such as pay per use. It provides flexible on-demand access to an elastic computing resource base and represents the infrastructure, software, platforms, storage and application containers as a cloud where users can provision and access services over a network. Cloud Computing has a large number of deployed solutions in education (Alabbadi, 2011; Liu et al., 2011; Sultan, 2010), big data computing (Hashem et al., 2015), health (Nkosi and Mekuria, 2010; Rolim et al., 2010), private sector and government (Kshetri, 2010; Zhang and Chen, 2010) domains.

Major global cloud platforms have their data centers concentrated in countries and regions where there is stable infrastructure and high reliability, performance and low latencies are guaranteed. Such cloud platforms make two broad assumptions: The first assumption is that since data centers run in regions with stable infrastructure, there is no special considerations when architecting cloud platforms to deal with unreliable Internet connectivity or frequent power cuts. The second assumption is that developers and users who consume cloud services have access to stable infrastructure to develop for or consume cloud-based services. From experiences developing for and consuming cloud services in low-resources settings, we find that these assumptions are not true. In such settings, challenges such as frequent Internet partitions, unannounced power shutdowns, poor quality of services, among others, are the rule rather than the exception (Fig. 1). The availability of high-speed Internet access is ranked the number one concern by African-based stakeholders as a key barrier to adoption and use of cloud computing in Africa (Maaref, 2012). Other concerns include vendor lock-in, data security and protection, and price. Both users and developers situated in low-resource settings are equally affected

* Corresponding author.
E-mail addresses: baino@mak.ac.ug (E. Bainomugisha), alex.mwotil@mak.ac.ug (A. Mwotil).
URL: https://ibaino.net (E. Bainomugisha).

https://doi.org/10.1016/j.deveng.2022.100102
Received 9 June 2022; Received in revised form 14 September 2022; Accepted 7 November 2022
Available online 17 November 2022

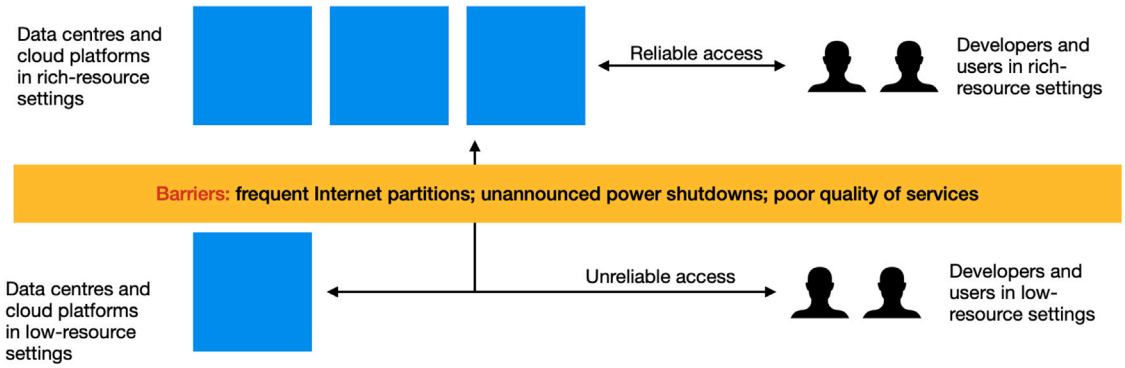

Fig. 1. Barriers to access of cloud services in low-resource settings.

by the above challenges when developing or consuming the cloud-based services. With the increased reliance on digital services especially for the attainment of global Sustainable Development Goals (SDGs), addressing barrier to adoption of cloud computing services will be critical.

The unanswered question is: *How can cloud platforms be designed to facilitate seamless access to cloud-services for users and developers situated in low-resource settings?* To address the above challenges and the arising research question, this paper presents:

1. Challenges and requirements for designing and operating a resilient multi-cloud model for low resource settings.
2. The design considerations and properties for implementing a resilient multi-cloud and bare-metal application cluster such as application state, networking, loadbalancing, monitoring and service exposure for external user access.
3. A prototype implementation of a resilient multi-cloud and bare-metal application cluster (Crane Cloud) which is a subset instantiation of the design and implementation options available. This will provide researchers and practitioners with a review point for further research and implementation cogitation respectively.
4. Evaluation of performance and user experience of Crane Cloud platform by implementing and deploying a computational and bandwidth intensive machine learning system that shows lower response time compared when hosted on other public clouds.

2. Requirements for a custom cloud-service layer for low-resource settings

As introduced above, low-resource settings are characterized by contextual challenges that present additional and new requirements for cloud platforms. To concretize these challenges, we use a real world scenario from low resource settings and present the requirements in Section 2.2.

2.1. Motivating scenario

The Automated Plant Disease Diagnosis (APDD) system is a real-world case study of a machine learning system used by African farmers and agricultural scientists in low-resource settings to diagnose plant diseases (Mwebaze and Biehl, 2016). Specifically, it is designed to provide near real-time in-field diagnosis of plant diseases and identification of pests for cassava crops by farmers and agricultural experts in the East African region and national agricultural research organizations. Originally conceived as a monolithic system, the APDD is transitioning to a microservice architecture to optimize efficiency and work around the technology constraints (Fig. 2). It consists of several microservices including, (1) pest surveys module, (2) in-field automated pest identification and count (such as white-flies) on plant leaves, (3) rapid plant disease diagnosis, and (4) prediction and spatial analysis of plant disease incidences and spread.

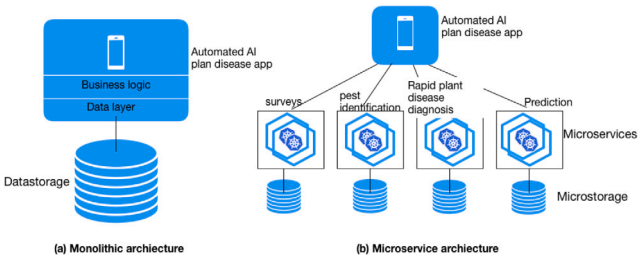

Fig. 2. Monolithic vs micro-service architecture for the automated plant disease diagnosis application.

The data pipeline for APDD system involves huge training datasets of over 300 GB including images of diseased and healthy plant leaves. The dataset is used for training and evaluation data for the machine learning models. The images are crowd-sourced from local farmers using mobile phones and can be stored on available public cloud platforms as well as on local storage to allow online training and evaluation of the machine learning models. The process flow typically involves uploading and downloading a huge datasets from the local storage and cloud systems. Such a dataset can take up to 24 h or so to upload to a public cloud service provider over slow intermittent connections that characterize many low-resource settings.

2.2. Scenario analysis and requirements

The above case study signals one of many other similar systems faced with technical hurdles when delivering cloud-based solutions in a low resource setting. To an extent, it reveals key requirements and challenges that can be addressed by containerized cloud-based systems that are developed in and for use in such environments. If not addressed, they are likely to be significant barriers to development, adoption and use of cloud-based solutions such as machine learning services for many developers, researchers, startups, students, users and organizations based in such settings. We argue that it should be possible for users situated in such settings to take advantage of the benefits offered by cloud-based computing models through creation of appropriate abstraction service layers. We discuss the key requirements and challenges below:

2.2.1. Frequent internet partitions and bandwidth constraints

Whereas the main cloud providers have set up cloud services on stable infrastructure, developers and users situated in low resource settings face major constraints when accessing their hosted applications and cloud platforms. In such settings, access to cloud services and platforms is often through low-end smartphones over slow and erratic

2G/3G/4G (von Wielligh et al., 2018) connections that are characterized by frequent network partitions and bandwidth constraints. As can be observed from the above scenario, a developer uploading training dataset of 300 GB images to a cloud platform could easily take several days over a slow connection. The in-field data collection by farmers to a public cloud repository can be very slow because of sporadic Internet connectivity in rural areas. Access speed to cloud services would be faster if the cloud data centers were located near to the users, however, 98.4% of the data centers of popular cloud providers are located overseas (outside Africa) (Calandro et al., 2018; Corneo et al., 2021). For instance, on the African continent, almost 70% of the content and services consumed by the Internet users is hosted and delivered from overseas data centers resulting in poor user experience due to high network latencies (Corneo et al., 2021). Furthermore, mobile data costs in Africa are significantly high in real and GDP-relative terms in the world with the prices averaging US$ 7.04 per 1 GB (Ecobank Research, 2018; Gillwald and Mothobi, 2019) and speeds as low as 56 kbit/s. However, it is important to note that these issues are similar in any other location with similar resource constraints. The design and setup of most public cloud platforms assume stable infrastructure across the users and leave the issues of connectivity challenges to the application developers. This shifts the burden and unnecessary complexity to the application developers who must consider offering different function-trimmed variations of their app services for users situated in low-resource settings, for instance, Facebook Lite app (Shankar, 2015), WhatsApp Lite (DigitBin, 2019), Uber Lite (Uber, 2021), Google Go (Google LLC, 2019) and Gmail Basic (Google LLC, 2021) for slow Internet connections and low-end devices.

The above issues form the motivation of the research work in this paper. Cloud platforms need to be designed and optimized to work under frequent Internet partitions and bandwidth constraints challenges since they are the rule rather than the exception in low-resource settings.

2.2.2. Data jurisdiction restrictions

Many countries are coming up with new laws and guidelines to regulate the location of data and access. Countries have recently enacted laws that enforce data sovereignty and prevent data from leaving the country's boundaries. This becomes difficult to implement in regions where public clouds do not have physical presence. For instance, on the African continent where public cloud data centers are still sparse, it becomes almost impossible to comply with such policies. The microservice architecture and cloud orchestration platforms such as Kubernetes (Burns et al., 2016) promises potential remedies to this challenge. In the above scenario, the APDD system can be broken into independent microservices each with different data jurisdiction policies. For instance, the data storage and management microservice needs to be enforced to remain within the boundaries of the country while the plant disease prediction service can run in a public cloud without restrictions and benefit from the rich machine learning libraries and tools. Such a setup would require a multi-cloud environment that spawns boundaries with support for data jurisdiction policies specific to a microservice and use case.

The popularity of cloud computing solutions has introduced gaps in key processes of the data management cycle (collection, storage, analysis and use/reuse). Most cloud solutions do not provide controls over where data should be stored and in cases where there is no infrastructure presence, users have to make exceptions at the expense of prescribed hosting recommendations. Cloud providers also distribute content over spatial infrastructure located in different regions to maintain the cloud Quality of Service (QoS) along dimensions of performance, availability and reliability. This leads to silos of data spanning different geographical regions that users may have no idea or control of.

Governments are now adopting data localization where a nation's data is collected, processed, and/or stored inside the country and data sovereignty where data is subject to the laws and governance structures within the nation it is collected. This has led countries and regions to enact Data Protection and Privacy laws such as the European Union (EU)'s operational General Data Protection Regulation (GDPR) that impose stringent policy controls on the use of Personally identifiable information (PII). In Africa, over 80% of the countries have data protection laws with varying degrees of enforcement (Daigle, 2021).

2.2.3. Vendor lock-in

Vendor lock-in, which is a user difficulty of switching from one vendor to another, is regarded as one of the major deterrents in the adoption of cloud by developers as well as small and medium-sized enterprises (SMEs) (Sahandi et al., 2013). These user categories may not have the local computing resources to run their workloads and most often resort to cloud providers but flexibility to switch/shift between providers is one of their desirable properties. Other than vendor lock-in, there are other variations including product lock-in, version lock-in, architecture lock-in, platform lock-in, skills lock-in and mental lock-in (Hohpe, 2019). Public clouds offer provider-specific proprietary solutions to meet the market demands and this has resulted in an interoperability, integration and portability downside across the cloud divide. Consequently, the applications developed for a specific cloud provider such as Amazon Web Services (AWS) may not work out-of-the box with another cloud provider such as IBM cloud due to inherent dependencies of the underlying IT infrastructure (hardware and software), cloud semantics and non-standardized APIs (Opara-Martins et al., 2016; Kratzke et al., 2014). The migration of cloud services from one provider to another usually requires major reworks on the application that may be catastrophic for mission-critical systems. For instance, the APDD case study may use vendor-specific machine learning libraries and tools making it difficult to migrate to another cloud when there is need.

The vendor lock-in challenge emphasizes the need for new abstraction layers to alleviate the difficulty of migrating applications between clouds. New platforms and architectures such as Kubernetes (Burns et al., 2016) offer new possibilities to implement a vendor neutral layer on top of public and private clouds. However, the current offerings of managed Kubernetes layers assume migration of services in situations where there is stable connectivity and infrastructure and are not designed for data centers that may be characterized by frequent network partitions and bandwidth constraints.

2.2.4. Poor quality of service

In cases where cloud services are offered from data centers located overseas and far from the consumers, user experience can be poor compared to closely-located content. For a user located in Africa, the average round trip time to reach cloud content ranges from 70 ms for proximal services to over 250 ms for content such as in North America and Europe (Corneo et al., 2021). Coupled with high costs of Internet access, limited bandwidth and high latencies mainly introduced by distance, the content load times are high and this negatively impacts the user experience. Consider for example in the APDD case study where a cloud-based service for farmers is situated on a public cloud with a data center situated in the North America while the target farmers are located on the African continent. The longer the distance, the higher the number of intermediary links which can act as failure points (bottlenecks) and potentially introduce network packet losses. Furthermore, there are applications that are delay-sensitive and these require optimal and stringent quality of service parameter values such as low latency, low jitter and minimal or no packet loss for best performance. Currently, public cloud providers attempt to solve this challenge by moving services closer to the user. This approach however assumes presence of data centers closer to the user. Unfortunately this is not always the case for users located in regions where public cloud data centers are sparse.

In the next section, we present the design options that need to be considered when developing a multi-cloud service abstraction layer to address the above challenges particularly in low resource settings. In the subsequent sections, we demonstrated the instantiation of the design considerations in a practical open source cloud project.

3. Design considerations for a resilient multi-cloud service

In this section, we present the design considerations and properties for a resilient multi-cloud service layer that is envisioned to meet the above requirements, namely, (1) Frequent Internet partitions and bandwidth constraints, (2) Data jurisdiction restrictions, (3) Vendor lock-in, and (4) Poor quality of service. In Section 4, we shall present the first prototype implementation that instantiates some of these design considerations (DC) and properties:

3.1. DC 1: Containerization and orchestration of applications

Many organizations are recently adopting microservice architectures in place of traditional monolithic architectures so as to truly reap from the benefits of modern cloud services. Microservice architectures involve collaborations between different fine-grained and independently deployable modules usually without a centralized controller to achieve the desired overall functionality of the system (Nadareishvili et al., 2016; Knoche and Hasselbring, 2019). Driven by application features such as scalability, agility, performance and fault-tolerance, microservice architectures involve autonomous software development teams independently working to build loosely coupled application features and employing collaborative workflows and automation tools from version control systems to full scale production deployment (Hasselbring and Steinacker, 2017). A number of popular technology companies such as Uber, Spotify, Netflix, Amazon and Ebay are now using microservices at the core of their business processes and have achieved differing levels of reliability and scalability in their services (Knoche and Hasselbring, 2019). As part of the inceptor team of the microservice terminology, Fowler and Lewis (2014) identified the following key properties and benefits of microservice applications:

1. *Autonomous software components*: A complex system is decomposed into service-specific pluggable components along business service lines allowing for each service deployment and modification without impacting other functional facets of the application. The units are small, granular, manageable and loosely coupled and they communicate using well defined interfaces based on platform-independent data formats and technologies such as HTTP/REST or messaging solutions such as Kafka or RabbitMQ (Knoche and Hasselbring, 2019). Microservice architectures thrive on the notion of 'small size' with reductions in the scope of the problem, task completion time, feedback response time and the size of the deployment unit. This in turn translates to an application's resilience to cascading failures, easier maintenance and seamless deployment. In addition, the decomposition yields small coherent components that are easy to understand and debug.
2. *Decentralization*: The services (components) are distributed, as there is no central controller and may store only data related to the supported business domain or different instances of the same database technology. Monolithic architectures usually have a single logical database for a range of applications. The teams are also decentralized and can adopt appropriate standards that allow them to deliver the business domain functionality without reinventing the wheel. The teams are responsible for the build it/run it cycle of the service and this improves on the quality of the code, fastens the deployment process, promotes component reusability and isolates the impact of changes on the schema.
3. *Technology independence*: The microservices can be built using different tools such as frameworks, programming languages and databases given the architecture supports heterogeneous technologies. The frontend service and reporting tools could, for example, be developed using a User Interface (UI) framework such as Node.js,[1] the backend service could be written using Java,[2] a real-time component could use C++,[3] the persistent data storage mechanisms could employ MongoDB.[4] The developers are at liberty and can independently choose a technology stack that best fits the work at hand hence placing the responsibility of development, maintenance and generally ownership to the teams.
4. *Automation*: One of the modern software development concepts is Continuous Integration (CI), Continuous Delivery is where an application is manually deployed and Continuous Deployment (CD) where an application is automatically deployed. This has immensely changed how developers and testers ship software culminating into the famous CI/CD pipeline depicted in Fig. 3. In CI, the development teams implement changes and write to version control systems with automated build and test scripts and in CD, the application is deployed/shipped to either a staging or production environment. This has been spurred by the introduction of Software Developers (Dev) and IT Operations (Ops) generally termed as DevOps model for software development: an end-to-end model for fast delivery of reliable applications and services involving cultures(by and for the people), automation (testing, feedback, deployment and performance benchmarks), quality measurement and sharing of ideas, processes and tools (Hüttermann, 2012). In an incremental migration and architectural refactoring of a commercial mobile backend (monolithic application) as a service, Balalaie et al. (2016) noted that the microservice architecture is an enabler for use of DevOps. Automation does not imply management overhead on introduction of new applications (Dragoni et al., 2017) but rather increased agility and reliability. In low resource settings, further automation of the containerization and deployment processes can ultimately enhance adoption of cloud computing (Mwotil et al., 2022).

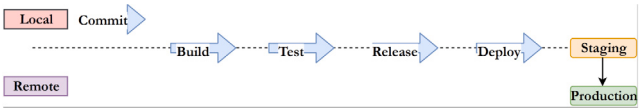

Fig. 3. CI/CD pipeline.

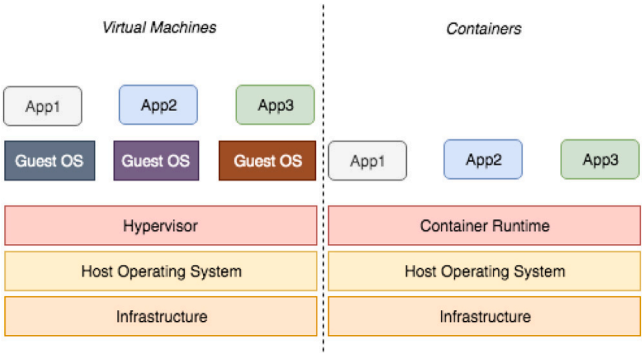

Fig. 4. Virtual machines vs containers.

At the application development and deployment levels, developers require appropriate abstractions to efficiently package and deploy microservices on a cloud infrastructure. Recently, containers have

[1] https://nodejs.org
[2] https://www.java.com
[3] https://www.cplusplus.com
[4] https://www.mongodb.com

emerged as a novel abstraction to deploy microservices as opposed to traditional virtual machine approaches. Containers form the basic deployment units to release and ship microservices (Nadareishvili et al., 2016) given their modular design which closely maps with the functional decomposition of a complex business application. The lightweight nature of containerized applications coupled with functional and configuration encapsulation facilitates replication and portability, are cost-efficient and have a reduced overhead on the operation and maintenance line. It is for these reasons that containers emerged as the most suitable packaging toolset for microservices.

A container is a lightweight "virtual machine" based on the Linux Kernel Extension (LXC)[5] *technology that allows running of many (up to hundreds) isolated and autonomous Linux environments on a single server or virtual machine*. It is a collection of communicating components such as application code, runtime, system tools, system libraries and settings required to run an application. Container history dates back to the early 2000's with the introduction of FreeBSD jails (Kamp and Watson, 2000). The FreeBSD jails provide a logical isolation environment through sandboxing for system features such as the filesystem, users and network mimicking a virtual machine but running on the same operating system (Hope, 2002). In comparison with virtual machines, containers consume much less computing resources and take the least provisioning time because virtual machines require different instances of the operating system (guest OS) while containers use the same host operating system as shown in Fig. 4. CoreOSrkt,[6] Mesos Containerizer,[7] LXC Linux Containers, OpenVZ[8] and containerd[9] are examples of containerization technologies but Docker[10] is by far the most popular.

Fully leveraging the elasticity of using container-based virtualization requires *automated orchestration* and cluster management tools such as Kubernetes,[11] Nomad,[12] Docker Swarm[13] and DC/OS[14] that provide an abstraction layer between computing resource pools and the applications. These tools provide a number of attributes important to implement service discovery, scalability, load balancing, service replication and provisioning of replicas across multiple compute nodes (Dragoni et al., 2017; Knoche and Hasselbring, 2019). Despite its complexity such as in installation, Kubernetes is the most widely adopted and powerful container orchestration tool owing to its immense scalability, performance and advanced automation features. It has inbuilt monitoring and logging libraries and processes which are lacking in the other tools (Modak et al., 2018). In Kubernetes, the applications are packaged as containers and wrapped in a pod (a group of containers that form the basic deployable Kubernetes unit), which can then be deployed via a declarative manifest (YAML[15]) file. In this file, the user describes the desired state of the application such as name, replica count, labels, storage mounts and exposed ports and the deployment controller works to ensure that this state is maintained at all times for example by replacing pods that fail or are evicted from their nodes.

3.2. DC 2: Application placement and replication

In a multi-cluster(node) environment, placement involves determining what cluster(node) should host an application based on factors such as application affinities, resource (storage, memory and network) availability, user preferences and data jurisdictions. The clusters(nodes) could be located in zones/sites with varying availability and regulatory constraints. Application replication ensures that there is operational business continuity in the face of downtime as a result of computing equipment failures, natural disasters, planned maintenance operations (Levijarvi and Mitzev, 2015), power outages, unreliable network connectivity, limited bandwidth and utilization surges. Replication can further be used to realize scalability, availability and fault-tolerance of an application under scheduled or unplanned downtime periods. The replica count, the number of clones of an application, depends on the reliability assurance as a requirement for an application and also the popularity of the service in a cluster(node) region. Replication strategies fall into two broad categories:

1. Static replication strategy where the number of nodes and replicas is defined beforehand
2. Dynamic replication strategy where replicas are automatically created or destroyed based on changes such as user density, performance, storage utilization, loadbalancing features and bandwidth consumption.

For the rest of this section, we shall consider the four microservices for the Automated Plant Disease Diagnosis (APDD) system: Surveys, pest identification, rapid plant disease diagnosis and prediction. A resilient multi-cloud service should provide an adaptive service replication approach that considers the following attributes:

3.2.1. User defined and cost-sensitive replication policies

The cloud service should operate only within the user-defined replication limits but also ensuring minimal replication costs between the clusters and the target nodes. In the deployment of APDD, the user may specify a replication limit of 3: the cloud service should ensure that there are three instances of APDD microservices available at all times. Additionally, the replication approach in cases of downtime should consider the replication costs such as the impact on the network performance whenever provisioning is required. It should also be noted that the cloud service provider may impose restrictions based on, for example resource availability, which the user adheres to but regardless, the user will operate on a higher level of abstraction.

Fig. 5 shows a 3-cluster cloud service located in different regions as shown by the link latencies. The user specified a replication count of 3 (classic 3-replica replication strategy (Li et al., 2012)) and the services are initially deployed on each of the three clusters. Cluster 2 experiences a downtime and this requires scheduling of the four microservices into another cluster. The decision on which cluster the services will be provisioned on should consider the transmission cost and this will ultimately be Cluster 1. Considering the costs that the fixed-replica count strategy may impose, Li et al. (2011) presents a dynamic cost-effective replication algorithm for data in cloud data centers. This approach requires computation of a reliability requirement value which informs the replica count. The default replica count is 1 and this will be scaled upwards to meet the reliability requirement of a data intensive system. The initial costs of this approach are significantly low but may increase exponentially with provisioning of more replicas. This algorithm can further be enriched by introducing user-defined boundary limits while ensuring the reliability requirement is maintained.

3.2.2. Quality of service (QoS) and high availability

Some applications in heterogeneous clouds have higher QoS requirements in comparison with others for example a critical medical diagnosis system that should operate under stringent availability and consistency constraints. In distributed cloud environments that support service geo-replication, maintaining consistency and performance consecutively is desirable but not fully achievable according to the CAP's theorem (Brewer, 2000). Consistency may be achieved but at the expense of degraded system performance. A number of research works have been published in the line of QoS and high availability.

[5] https://linuxcontainers.org
[6] https://coreos.com/rkt/
[7] http://mesos.apache.org/documentation/latest/containerizers/
[8] https://openvz.orghttps://openvz.org
[9] https://containerd.io
[10] https://www.docker.com
[11] https://kubernetes.io
[12] https://www.nomadproject.io
[13] https://docs.docker.com/engine/swarm/
[14] https://dcos.io
[15] https://yaml.org

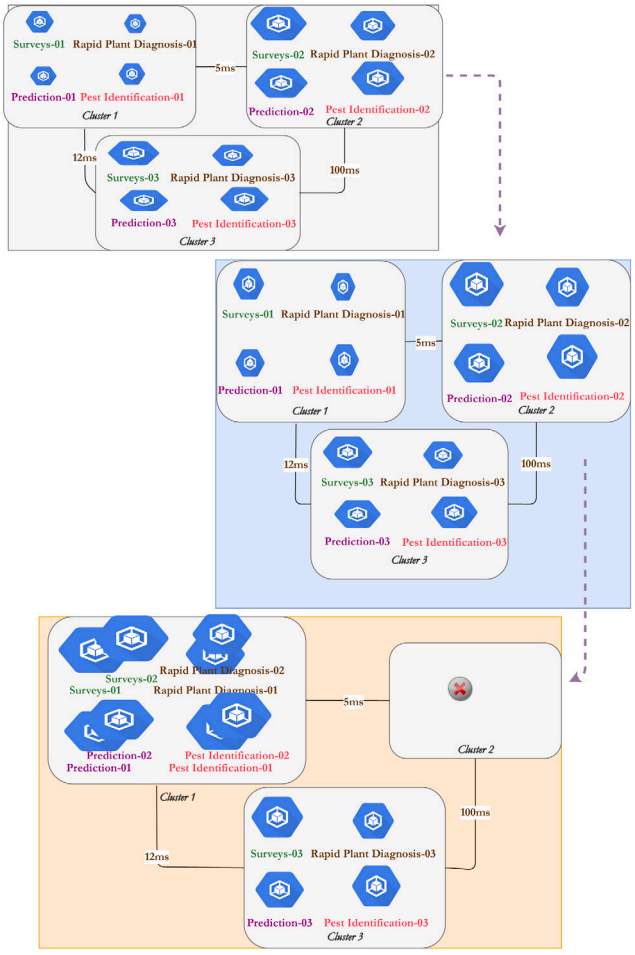

Fig. 5. Automated Plant Disease Diagnosis System with user-defined replication factor 3.

Esteves et al. (2012) developed a novel QoS consistency model for geo-replicated data in cloud computing. In this model, the consistency of an application module is either dynamically strengthened or weakened based on its criticality needs. This requires an in-memory cache system that has a monitor/control component, session manager, scheduler and a QoS engine. These components manage the replication of a service based on the cache statistics specific to the application. Gao and Diao (2010) proposed a lazy master update propagation model for transaction-based systems in cloud computing. This requires a master (registry host) replication site with replica management, an update receiver, propagator and executor components. The components manage replication through synchronization and message sends (transmission, receipt and update) with different sites. The update instruction is first committed at the master node and later replicated to other sites. This model also uses an immediate update propagation variant for data access to ensure that users only access a fully replicated service. The success of this approach requires a highly available master site and geographically nearby secondary sites as the updates directly impact network performance. Boru et al. (2015) asserted that most cloud applications interact with database systems that may be located locally or remotely and data queries are sent to locations nearest to the user for enhanced availability and improved user experience. The model presented is intended to minimize energy consumption, bandwidth utilization and communication delays in the network. The energy consumption model aggregates power usage from computing servers as well as core, aggregation, and access switches to build an optimal profile.

3.2.3. Location-aware

Cloud-enabled applications necessitate different modes of delivering system functionality to the end users for improved experience and this includes location awareness: applications and data should follow the users. Application replication models should dynamically consider locations of the end-users based on their density, proximity and mobility and perform desirable provisioning in situ. Location awareness may also be in line with the QoS requirements expected (requires continuous monitoring for agreed QoS and replication should allow for reconfiguration of available resources so that minimum QoS requirements are achievable) of an application. If the main consumers of the APDD are in Uganda, then provisioning of the application should be done on the nearest clusters. At a certain point in time, the APDD consumers may be in South Africa and this requires resource reconfiguration and provisioning to serve the new user environment. Location-aware systems should strive to achieve fairness in cost vis-a-vis performance (Shi et al., 2020) in multi-cloud setups.

The design of a location aware system requires request tracking based on the Internet Protocol (IP) address, monitoring components, location sensing and prediction technologies and assorted geolocation APIs so that user requests take advantage of nearby computing servers to carry out demanding tasks. This allows users to have a more contextual and fulfilling experience while drastically reducing the costs of delivering compute, network and storage resources. It should also be noted that location-aware systems may impose serious privacy issues and should be handled appropriately.

3.3. DC 3: Portability and multi-cloud migration

Portability in cloud computing can be defined as the ability for movement of applications, workloads, processes and data from one cloud environment to another with least disruption, whether manually or automatically. The least disruption should translate to lowest possible cost, effort and time. The movement of one service, such as the one instance of the prediction microservice for automated plant diagnosis system from Cluster 2 to Cluster 1 as shown in Fig. 5, should cause minimal or no downtime and should not compromise the QoS attributes tagged to overall operation of the system. As noted earlier, cloud computing offers significant benefits such as scalability, disaster recovery, mobility and cost reduction in operation of an organization's IT infrastructure. This is evidenced in the introduction of different cloud computing technologies and deployments to make it easy for organizations to embrace and adopt this new wave of handling compute, storage and network workloads. One of the pertinent issues in the adoption of cloud computing is vendor lock-in (lack of portability and interoperability across cloud platforms) where providers work with specific technologies such as tools and programming interfaces. Given the different deployment models and the cloud service models, organizations should be able to move cloud services from one provider to another without worries of complexities and infrastructure dependence.

Bozman and Chen (2010) identified standardized programming interface, abstraction layers and management capabilities as some of the key enablers for portability and service migration between cloud providers. A standardized programming interface includes programming toolsets to support application movement, the abstraction layers insulates users from infrastructure complexities and dependencies and the management tools provide interfaces for operational activities such as application deployment, monitoring and troubleshooting. However, adopting a standardized approach to portability is a myth as it is extremely difficult for providers to agree and adopt a unified set of standards. This requires major rework of the proprietary APIs and file formats, and this also destroys the competition spirit which has been very effective in delivering high quality cloud services for the market (Gonidis et al., 2012).

Most of the research work geared to support portability across different cloud environments such as mOSAIC[16] (Open-Source API and Platform for Multiple Clouds), Open Cloud Computing Interface (OCCI) have focused on abstraction layers and management tools and container-centric solutions. Containerization allows an application to be built once, placed inside a container image or series of images for a multi-service application and running it on any host operating system that supports the containerization technology in perspective such as Docker. It should however be noted that achieving full portability out-of-the-box and application storage persistence using containers has some limitations such as no support for cross operating system support - *a containerized Linux application requires a Linux host operating system, a windows one requires a Windows operating system*. Despite this limitation, containerization is a big step in ensuring applications can run uniformly and consistently across a plethora of computing platforms or cloud environments.

3.4. DC 4: Resilience to network partitions and bandwidth constraints

Low-resource environments usually experience unreliable and intermittent Internet connections due to power failures, few or no network redundancy points and the low Internet penetration hindering access. In a multi-cluster setup, this can result in network partitions where some clusters are totally unreachable for prolonged periods of time. This setup also requires additional bandwidth to support, for example, synchronization of services across these fault domains that could be distantly located. Network partitions can lead to ruinous system failures, some of which leaves the system in a continuous error state, that capitulates into data loss, corruption, unavailability and inconsistency, broken locks and system crashes (Alquraan et al., 2018).

A stateful application is a data-driven application that requires persistent storage across a set of multi-cloud clusters with strict data consistency demands. To allow for this, there is a need for consensus algorithms to ensure that cluster states are globally consistent thus providing for dynamic leader election approach (the cloud cluster to act as the leader and handle writes), replication for cluster consistency and safety in ensuring that client requests are served with the correct results (Ongaro and Ousterhout, 2014) in faces of complete, partial and simplex network partitions (Alquraan et al., 2018). *Distributed consensus algorithms* are a well-studied research problem with a popular convergence in Raft,[17] an implementation of Paxos, an easily understandable and practically implementable algorithm that guarantees a shared state among multiple servers for full operation of the system. Raft achieves this by decomposing the consensus problem into leader election, log replication and safety as independent tasks.

In resource-constrained environments, Internet traffic could be categorized into different priority classes such as sensitive(cluster replication), best-effort (service access) and undesired (other Internet traffic) to closely correspond to high, medium and low priority traffic which can then be dynamically allocated bandwidth. Assignment of optimal bandwidth to the different traffic classes will ensure that the defined QoS attributes such as availability and consistency of an application deployed on the clusters is achievable. This requires a fast and rigorous classification algorithm and considerable dynamic changes on the network. Another approach would be to route user requests to the closest cluster hence avoiding upstream bandwidth costs and limitations and also improved user experience. Multi-cluster support for application and data replication to achieve consistency, availability and tolerance to network partitions over Wide Area Networks (WANs) and especially geographically distant network points is still an open area of research. This requires a good and redundant connectivity between communication endpoints and a compromise in application properties such as data consistency, availability and user experience. In all this, a fair concession for near-efficient application demands should be achieved.

[16] https://occi-wg.org
[17] https://raft.github.io

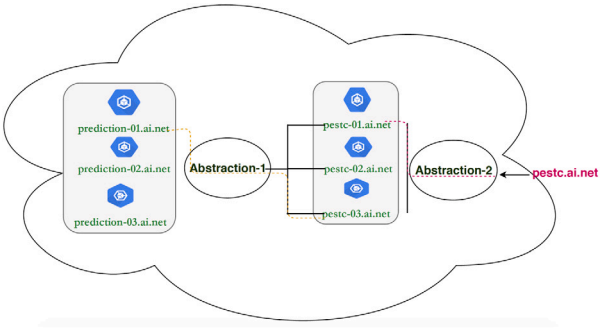

Fig. 6. DC 5: Service discovery and loadbalancer for the Automated Plant Disease Diagnosis System.

3.5. DC 5: Service discovery and load balancing

In a multi-cloud environment, applications may need to be scaled up by increasing application instances for improved user experience or scaled down by destroying excess instances to limit compute costs. In certain scenarios, an application may need to be moved from one cloud provider to another and rescheduled on a particular node in the new location. In the process, application settings such as its Internet Protocol (IP) addresses and Domain Name Service (DNS) attributes may change and this necessitates an update to all reliant services in order to maintain application availability. Service discovery is the ability of a client or service component to discover healthy and available services (providers) that it can connect and communicate with. In addition, an application with multiple instances of different services spread across different clouds requires an internal and external load balancer solution to prevent network and node overload which in turn translates to optimal usage of computing resources. The internal load balancers consider microservice communication inside a cluster (cloud provider) while external load balancers consider routing of client requests to user facing endpoints of an application. In Fig. 6, *Abstraction-1* and *Abstraction-2* represents the internal and external load balancers respectively. The prediction microservice and the pest identification microservice have to communicate and each has multiple instances and it is the role of the internal load balancer to route the traffic to the appropriate instances. A user visiting *pestc.ai.net* has no knowledge of what instance will respond to the request as this is abstracted by the external load balancer. *pestc-01.ai.net* at a certain point may be destroyed or rescheduled in another node with possibly a new IP address and name and the abstraction layers have to update their registries so that the request/response cycle is always complete.

Kubernetes implements two options of service discovery: one based on environment variables available and the preferable DNS-based service discovery. Using environment variables, a service is identified by the IP address and port it is running on for example *PREDICTIONS SERVICE HOST = 10.233.11.2* and *PREDICTIONS SERVICE PORT = 5000*. In DNS, the services are identified by names that are specific to the cluster namespace that the application is deployed in for example, prediction-01.dev.cluster1.local and *prediction-02.prod.cluster2.local* to identify two instances of the prediction service deployed in the *dev* and *prod* namespaces of *Cluster 1* and *Cluster 2* respectively. The DNS option is more flexible as the environment variables are fixed for the lifetime of the service and changes require a redeployment of the application and/or service. Service discovery outside the Kubernetes cluster requires services to be exposed through NodePort, Ingress and LoadBalancer alternatives.

Most service discovery mechanisms in distributed computing and service-oriented architectures employ a centralized approach where a central server(s) maintains information about all the services that includes access credentials, protocols, version numbers, service location and environment details. The service discovery process involves

a distributed client querying the central registry for location and information of other services either using a client-side discovery (a client queries the service registry, selects an available instance and makes a request) or server-side discovery (a router acting on behalf of the client queries the service registry and forwards the request to an available instance) implementation. The popular centralized service registry/discovery solutions include Netflix's Eureka,[18] CoreOS's highly available etcd[19] key–value distributed datastore, consul[20] and Apache ZooKeeper.[21] The major drawbacks of centralizing the service registry/discovery are the introduction of points of failure, performance bottlenecks and possible network congestion. Distributing the nodes providing these services and ensuring there are multiple instances in a consistent way usually suffices. Zhou and Shi (2010) and Ranjan et al. (2010) proposed an unstructured Peer-to-Peer(P2P)-enabled service discovery method for cloud environments based on Distributed Hash Tables (DHTs) with a decentralized index system. The peers maintain their own services and descriptions and a semantic-based matching rule is used to map the user requirement expressed in the query message to the desired service.

3.6. DC 6: Localized image registry

An image is an immutable file built according to instructions and can only be extended by building a layer on top of it. A container is an instance of an image with instructions on how to execute an application and operate as isolated environments with ability to interact with other containers and the host environment through well defined interfaces (Jaramillo et al., 2016). To fully effect containerization, an image of the application is created and pushed to a local or remote image registry through which a user, such as a DevOps engineer, can now pull and create an instance of it in a container host. A container image registry provides a storage location and distribution portal for images some with multiple versions identified by tags. Docker Hub,[22] Google's GCR,[23] Azure ACR[24] and Amazon ECR[25] are some examples of popular public/private image repositories.

According to Zhang et al. (2017), image pulling costs, workload network transition costs (such as bandwidth, latencies and connection limits) and energy conservation can significantly affect the scheduling and deployment of applications into container hosts. If the images are stored on the local container host, then the cost is negligible otherwise extra costs shall be incurred depending on the sizes of the layers, image location and bandwidth restrictions involved in fetching the image from a remote repository. In resource-constrained settings, the image may be located thousands of kilometers from the local container hosts and this negatively impacts deployment, for example in cases of downtime where provisioning on another local cluster instance to ensure availability is required. Imagine a *10GB* image file located on https://hub.docker.com to be provisioned on a container host in Uganda with a dedicated upstream bandwidth limit of *2Mbps*. On average, there is a network latency of *357 ms* between https://hub.docker.com and Uganda. To fetch this image file, it will take close to *12 minutes* and this can have a huge negative impact on the availability QoS requirement. To reduce container schedule (download) times, images may need to be distributed across different cloud providers and located very close to the container hosts.

3.7. DC 7: Platform monitoring and management

Monitoring is a critical and essential aspect of managing any IT infrastructure. Systems are susceptible to failure and without monitoring, it is difficult to ascertain the causes of failure and even anticipate for future ones. Compared to traditional monolithic applications, monitoring of microservice applications requires intensive service reporting features especially given their distributed nature (services run as independent processes on possibly geographically different hosts) and dynamic behavior. Monitoring aids users in understanding the overall health of an application, gain insight into the performance of constituent services of an application and to ensure that APIs are available and performing as expected. The monitoring metrics divided into platform/host (CPU, RAM, threads and database connections) and application metrics (service availability, service and API endpoint latency, success of API endpoints, API endpoint response times, API request clients, errors and exceptions) should be collected at each stage of the deployment pipeline. Haselböck and Weinreich (2017) identifies four areas for microservice monitoring based on monitoring activities of information generation, processing, dissemination and presentation: *Generation and collection of monitoring data*, *storage, hosting and distribution of monitoring data*, *processing of the data to obtain platform and application metrics* and *presentation of need-to-know information via a dashboard to the relevant stakeholders*. In addition, a real-time monitoring component of a production-ready microservices application to detect current and imminent failures due to changes in key metrics is necessary.

A number of monitoring tools and frameworks exist but most are either native (Amazon Cloudwatch,[26] Azure Monitor,[27] Google Cloud's Operations Suite[28]) or virtualization type specific (such as cAdvisor[29]) or commercial (such as Datadog[30] and Dynatrace.[31] Given a plethora of monitoring options available and complexities of monitoring microservice applications, a monitoring framework should be designed to capture, report and alert stakeholders on performance and failures of an application based on critical metric data. Noor et al. (2019) presents a framework for monitoring microservice-oriented cloud applications in heterogeneous virtualization environments. It is composed of mainly two components: a monitoring agent *(a cloud platform-independent software component that collects information from a microservice)* and a monitoring manager *(a software component that receives monitoring information from agents in heterogeneous cloud environments)*.

4. Crane Cloud: an implementation of a resilient multi-cloud service layer

This section presents Crane Cloud, a first-cut prototype instantiation of design properties and considerations for a multi-cloud service layer presented in Section 3 and summarized in Table 1. Motivated by the unique requirements for low-resource settings in Section 2.2, Crane Cloud is an open source project that attempts to encapsulate the intricacies of operating heterogeneous application clusters into a highly available unified platform for management and monitoring of a microservice application lifecycle. The target users of the platform include developers, researchers, students, and startups located in resource constrained environments. The public Github repository for the project is available on Github *https://github.com/crane-cloud/*.

[18] https://github.com/Netflix/eureka
[19] https://github.com/etcd-io/etcd
[20] https://www.consul.io
[21] https://zookeeper.apache.org
[22] https://hub.docker.com
[23] https://cloud.google.com/container-registry
[24] https://azure.microsoft.com/en-us/services/container-registry/
[25] https://aws.amazon.com/ecr/
[26] https://aws.amazon.com/cloudwatch/
[27] https://azure.microsoft.com/en-us/services/monitor/
[28] https://cloud.google.com/products/operations
[29] https://github.com/google/cadvisor
[30] https://www.datadoghq.com
[31] https://www.dynatrace.com

Table 1
Design considerations for a resilient multi-cloud service model.

ID	Design Consideration
DC 1	Containerization and orchestration of applications
DC 2	Application placement and replication
DC 3	Portability and multi-cloud migration
DC 4	Resilience to network partitions and bandwidth
DC 5	Service discovery and load balancing
DC 6	Localized image registry
DC 7	Platform monitoring and management

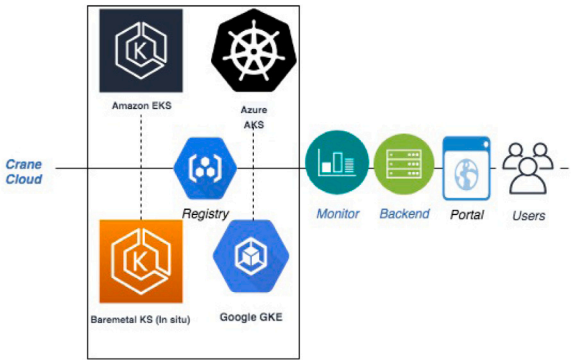

Fig. 7. *Crane Cloud* components.

4.1. Architecture and overview

Crane Cloud is an open source multi-cloud service layer designed to enable developers, organizations and researchers to set up reliable cloud-services in low resource setting. The Crane Cloud software layer was conceived to address the key hurdles of operating a cloud-service platform in resource constrained environments characterized by challenges identified in Section 2.2. Its main ingredients include resilience to network partitions, support for microservice orchestration, support for migration of services between private and public clouds to avoid vendor lock-in issues, seamless downtime and network traffic load distribution, monitoring metrics, and tools for transforming existing non-cloud compliant services into compliant cloud services. The multi-cloud service layer has five components (managed portal, authentication and authorization, monitoring and billing, local registry and the backend service) purposely designed taking into consideration features described in Section 3 and are shown in Fig. 7.

4.1.1. Multi-cloud cluster support

Crane Cloud enables harmonization of clusters from different cloud providers (public or private) and bare metal environments. It is designed to provide for easy migration, replication and loadbalancing of services across different clusters and cloud providers to ensure high availability and improve the general user experience. The cloud providers are chosen based on location, API service offering and costs of running workloads in their data centers.

4.1.2. Managed portal

The managed portal provides an interface for access to the rest of the abstracted Crane Cloud multi-cloud components. Developers can deploy and access their application services, monitor resources and running services, manage users and access the local private registry. Administrators can monitor the different clusters and nodes, view project details such as resource usage and its users, add or remove clusters, manage projects and accounts in the system and ensure that the clustered environment is performing optimally viz-a-viz the running services. The portal is a window to Crane Cloud features for management of resources in a clustered computing environment. As a developer, infrastructure setup complexities and application deployment intricacies are eliminated and focus shifts to software functionality. As a service consumer (user), the availability of a service and user experience regardless of location and underlying technologies is paramount.

4.1.3. Authentication and authorization

Crane Cloud uses the concept of user projects to closely map with a cluster namespace. A namespace is a logical environment that supports resource management for users working in a team or across teams. In Crane Cloud, access to the cluster resources requires valid credentials and the right privileges mediated by an API. This involves creation of projects and accounts that correspond to specific privilege levels in the cluster. This ensures that users can only access what may be required to perform functions within the cluster without affecting other projects. Two types of accounts are supported: Project user accounts that are managed outside the cluster and service accounts inside the cluster. The service accounts are directly used to manage resources in the cluster while the project user accounts are mapped to service accounts but usually with defined privileges/roles over a namespace.

4.1.4. Platform and service monitoring support

Monitoring is integral to the overall operation of Crane Cloud in terms of infrastructure and the distributed services hosted to ensure the QoS attributes are in check. The infrastructure includes the nodes while the services are the client applications and supporting tools. Monitoring coupled with an alert system also ensures that possible failures are averted early on before turning catastrophic. More specifically, the cluster monitoring involves the state of the cluster (collection of nodes) which is a constituent of node resource utilization parameters such as network bandwidth, disk utilization, CPU, and memory utilization while the service metrics include CPU, network, and memory usage irrespective of the nodes they are running on.

4.1.5. Local registry

The localized and replicated registry provides a platform through which users can easily upload, store and deploy their applications fast. The registry also provides trust signing and vulnerability scanning of container images. This significantly reduces the costs of upload and download of container images from public registries. The registry is locally available on all the clusters and container images are replicated on all. A local registry also ensures that administrators have more control over it.

4.1.6. Crane backend

The Crane Backend is the heartbeat of Crane Cloud providing abstractions and hooks for a number of services. The backend ensures that applications are appropriately scheduled on a cluster(s) or nodes, creation of service endpoints for communication between parts of an application and also ensure users can access the application, management of volumes for deployment of stateful applications, location-aware and user preference deployment of applications. It also provides endpoints for management of container images through the registry API and monitoring of deployed services.

4.2. Implementation

Crane Cloud is implemented using a combination of tools ranging from the design of the managed portal to the backend and from monitoring, registry and persistent volume management to the container orchestration. In most cases, open source solutions were experimented and used as much as possible.

Table 2
A comparison of major container orchestration implementation tools.

Feature	Apache Mesos	Docker Swarm	Kubernetes	Nomad
Community (As of September 2022)	Has a medium active community with *18,178* Github commits and *322* contributors[a]	Started in 2014, Docker Swarm has a relatively smaller community with *3,570* Github commits and *177* contributors[b]	Larger community with over *110,144* Github commits and *3,238* contributors[c] making it one of the most active open source projects	Released in 2015 by HashiCorp, Nomad is gaining traction. Has over *23,707* Github commits and *576* contributors[d]
Open Source	Yes	Yes but with an enterprise edition that detriments the open source version	Yes	Yes
Scalability & Flexibility	Automatic scaling but may require definitions in the application.	Manual scaling	Automatic scaling based on resource utilization	Dynamic & resource-centric
Fault tolerance	Yes	Yes	Yes	Yes
Monitoring	Has a diagnostic utility for health and other metrics but requires queries and aggregation through APIs	Uses basic out-of-the-box tools and supports other 3rd-party logging and monitoring tools	Uses inbuilt tools for logging and monitoring with support for third party integrations to keep track of logs and other performance metrics	Run-time metrics available for use with external monitoring tools

[a] https://github.com/apache/mesos.
[b] https://github.com/docker/classicswarm.
[c] https://github.com/kubernetes/kubernetes.
[d] https://github.com/hashicorp/nomad.

4.2.1. Container orchestration

Container orchestration tools are used to automate the deployment, management, scaling, and networking of containerized applications. These tools provide an abstraction layer between pools of resources and the application containers that run on those resources. Kubernetes, HashiCorp Nomad (Sabharwal et al., 2021), Docker swarm and Apache Mesos are the most popular tools with the former taking a fair share of the cloud-native market. With an impressively large community and functionality, backed by Cloud Native Computing Foundation (CNCF) and its open source nature, Crane Cloud uses Kubernetes for container orchestration. We considered five factors in selecting the most viable tool for our setup: Community (Support), Open Source, Scalability & Flexibility, Fault tolerance and Monitoring support. As shown in the comparison Table 2, Kubernetes is an open source project that impressively commands the cloud-native market with an adoption rate of 50% in the past 6 months and 87% market penetration supporting application scalability, fault tolerance and has inbuilt monitoring and logging tools and hence was the preferred choice for container orchestration implementation of Crane Cloud. Additionally, Kubernetes provides automated scheduling of applications, self healing capabilities, automated roll-out and rollback, service loadbalancing and a higher density of resource utilization.

4.2.2. Developer tools

The managed portal (frontend) was implemented using React.js,[32] a fast, scalable, and simple JavaScript library for building user interfaces created and open-sourced by Facebook. It uses the component-based architecture and declarative approach hence simplifying the debugging process. It allows creation of simple reusable and stateful components which can be composed to build complex user interfaces. The Crane Cloud backend was implemented as a REpresentational State Transfer (REST) API, using Python Flask,[33] for ease of use and integration while ensuring effective maintenance. Python Flask was used because it is simple, flexible and lightweight and now considered as one of the most popular Python web application frameworks by the programming community. PostgreSQL[34] is an open source object-relational database management system that Crane Cloud uses to maintain state for its internal workings. Applying multi-version concurrency control (MVCC) which allows several concurrent read/write operations, PostgreSQL can handle multiple tasks simultaneously and efficiently. In addition, PostgreSQL is SQL standards-compliant, highly programmable and extensible by many third-party tools and libraries.

4.2.3. Image registry

In implementation of the image registry, Crane Cloud considered open source extensible tools that can secure, scan and sign container images and also support replication across clusters. Harbor[35] perfectly fitted into the picture, providing an extensible API that the backend service would easily consume. Harbor delivers a consistent experience across multiple clouds and works best for environments that may not want to rely on public registries but rather a private one packaged as an add-on. Harbor additionally provides features such as access control on registry images, image vulnerability scanners, image storage and replication using a clustering mechanism. Crane Cloud a is multi-cloud service layer that can work with cloud providers in different regions and availability zones and a zonal scalable registry with a replication service is cardinal.

4.2.4. Stateful applications

Containerization technologies were originally designed to support stateless applications but considerable efforts have now been made to also support stateful applications owing to community adoption and contribution. This enables organizations to work with data-driven and legacy applications while leveraging the portability, scalability and highly available features of containers. Traditionally, Kubernetes used to provide support for manual attachment of cloud-backed storage to applications limiting usage outside the cloud provider but cloud native storage solutions have now been advanced. Crane Cloud uses OpenEBS,[36] an open source Container Attached Storage (CAS) solution developed using the microservice architecture. Distributed, monolithic or streaming, OpenEBS allows deployment of storage technologies and optimizations appropriate to an application type using different storage engines. Additionally, OpenEBS is a multi-cloud storage solution that shares the same philosophy of Crane Cloud borderless computing.

[32] https://reactjs.org
[33] https://github.com/pallets/flask/
[34] https://www.postgresql.org
[35] https://goharbor.io
[36] https://openebs.io

Table 3
Mapping of Crane Cloud components and design considerations.

	Crane Cloud Component	Design Consideration	Requirements and Challenges
1	Multi-cloud Cluster Support	Containerization and container orchestration engines.	Vendor lock-in, Poor quality of service
2	Managed Portal	Manual service deployment, placement and replication, monitoring and alerts portal	Data jurisdiction restrictions
3	Authentication and Authorization	Portability and multi-cloud migration	Cloud resource security
4	Crane Backend	Application containerization and orchestration, Ingress, Service discovery, scheduling, replication and loadbalancing for QoS, location-aware and user defined policy management.	Frequent Internet partitions and bandwidth constraints, Poor quality of service
5	Crane Registry	Localized image registry	Poor quality of service
6	Crane Monitor	Platform monitoring, alerts and management	Poor quality of service

4.2.5. Monitoring

Crane Cloud uses a combination of the inbuilt Kubernetes Metrics server and Prometheus.[37] The Metrics server is a cluster-wide aggregator of container resource metrics such as container CPU and memory usage exposed on each cluster node and available through the Metrics API. Prometheus is an open source time series database optimized to store monitoring metrics using a periodic data pull model and provides API endpoints. Providing basic visualizations and dashboards, Prometheus can be deployed alongside dedicated visualization and dashboard solutions from React.js libraries. Using the Prometheus APIs, visualizations can be generated and customized for users.

4.2.6. Mapping of Crane Cloud, design considerations and challenges addressed

The Table 3 shows a mapping between components of Crane Cloud, the design considerations and low-resource computing environment challenges addressed. The registry, monitor and backend are continuously being refined for a seamless user experience. It should also be noted that resistance to network partitions and bandwidth constraints as a design consideration and how Crane Cloud can practically implement this is an ongoing research area.

5. Experiments and results

To demonstrate the utility of the Crane Cloud platform, we deployed the rapid plant disease diagnosis (mcrops) microservice of APDD on the Crane Cloud platform. As introduced in Section 2, mcrops provides machine learning-based diagnostic tools to detect viral crop diseases in cassava plant using mobile and web-based technologies. In brief, the users upload images of suspected infected cassava root tubers through a mobile application or web browser and mcrops computes the Cassava Brown Streak Disease (CBSD) score to indicate the disease presence levels. The users can perform single uploads and/or multiple uploads of the image data. As shown in Section 2, mcrops is deployed as a monolith and for Crane Cloud deployment support, the application was containerized using Docker. The containerization process involved access to the mcrops source code, writing of the Dockerfile, iterative building and tagging of the Docker image and pushing it to the Crane Cloud image registry https://registry.cranecloud.io/. On the Crane Cloud portal, the deployment involved providing the application details such as name, container port and the docker image reference (as shown in Fig. 8) from which mcrops is run and ingress resources for external access are subsequently created and availed for use.

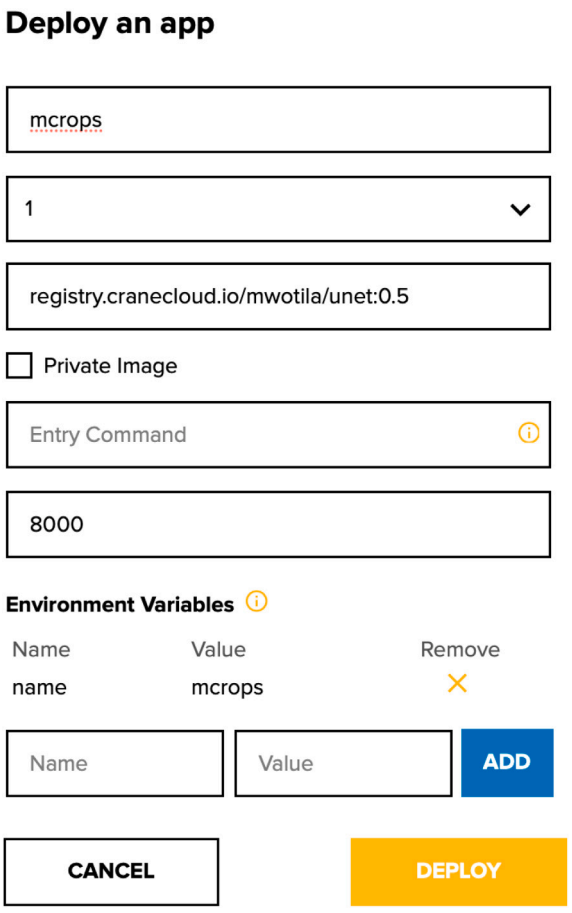

Fig. 8. Deployment of mcrops on Crane Cloud.

5.1. Experiment setup

The purpose of the experiment was to evaluate the performance and user experience of mcrops when deployed on a public cloud (AWS) compared to the deployment on the Crane Cloud platform. We consider the response time metric as an important metric for the measurement of the quality service and user experience. The response time has been pointed out as one of the key quality of service metrics for cloud providers (Xiong and Perros, 2009; Alhamad et al., 2010). Response time is particularly relevant to resource constrained environments that are characterized by frequent Internet partitions and bandwidth constraints and poor quality of service. We used Apache JMeter,[38] a

[37] https://prometheus.io

[38] https://jmeter.apache.org/

Table 4
JMeter Test Settings.

Code	Images	Users	Loops	Ramp-Up time
A1	1	1	10	10
B1	1	1	20	20
A2	10	1	10	10
B2	10	1	20	20

Table 5
Experimental setup for mcrops tests.

Connection/Sampler	unet.mcrops.org		mcrops.cranecloud.io	
2G (45 Kbps) 3G (4 Mbps) 4G (8 Mbps) WiFi (10 Mbps)	A1,B1	A2,B2	A1,B1	A2,B2

popular performance testing tool. Specifically we used JMeter tool to measure the response times of the two application setups against uploaded images over different mobile/wireless connections for a user situated in a bandwidth constrained setting. These testing settings and environments are a representative of the realities that developers and users working in the low-resource settings experience. On JMeter, four thread groups representing the image upload settings and two samplers (the two application setups: *unet.mcrops.org* and *mcrops.cranecloud.io*) were used as shown in Tables 4 and 5. In **A1** and **B1**, we considered the simplest scenario of a user uploading a single image but varied the number of times to 10 and 20 respectively as represented by the loops. The ramp-up time represents the seconds between successive user requests and we set this as same as the loops. In **A2** and **B2**, we randomly set the number of images to 10 to test a multi-upload use case as this feature is supported by the application. As the number of images and/or loops was increased, the response times were assumed to also rise hence the need to increase the ramp-up time. In all scenarios, 1 user was simulated because this is the typical field use case where a device has a non-shared connection to the next transmitting network device, for example an extension worker or rural smallholder farmer in a garden. The connection speeds for 2G, 3G, 4G and a local WiFi access point were determined by performing tests using the online Internet speed test tool *speedtest.net*[39] for upload and download speeds and the average computed and recorded in Table 5. The connections under use are motivated by real-world setups of resource constrained environments with no ideal and consistent network connectivity. Overall, we wanted to assess the utility of a platform like Crane Cloud for end-users situated in these environments. In this example, the students and researchers of the Computer Science Department at the University.

5.2. Results and discussion

The results of from the experiments are presented in Figs. 9, 10, 11 and 12 and Table 6. In the analysis, the median and mean (average) times were used to conclude on which experiment had a shorter response time using the different connections for both *unet.mcrops.org* and *mcrops.cranecloud.io*. The median was given priority in cases where its difference compared to the mean is large since it is not inflated by the existence of outliers in the data collected.

Wifi connection test results. In the WiFi connection setup, the response times for experiment **B1** and **A2** were lower in *unet.mcrops.org* compared to *mcrops.cranecloud.io* using the median and mean times as shown in Table 6. For experiment **A1**, the results indicate that *unet.mcrops.org* has its lowest response time at 11.23 s with an average of 22.9 s compared to *mcrops.cranecloud.io* at 6.91 s with an average of 11.66 s. The mean times also indicate that responses are better (lower) in *mcrops.cranecloud.io* compared to *unet.mcrops.org*. The experiment **B2** response times are lower in *mcrops.cranecloud.io* compared to *unet.mcrops.org* using both median and mean. The erratic behavior of the graphs for *unet.mcrops.org* in Fig. 9 is partly attributed to the high number of network hops, packet losses (as shown by the completion rates in Table 7 and connection variations from the packet sources. It should also be noted that the completion rates for *mcrops.cranecloud.io* under WiFi was at 100% compared to *unet.mcrops.org* at 95%.

4G connection test results. Using the 4G connection, the response times for experiment **B1** and **B2** was lower in *unet.mcrops.org* compared to *mcrops.cranecloud.io* using both the median and mean as shown in Fig. 10 and Table 6. In experiment **B1**, *unet.mcrops.org* had 28.01 s and 29.69 s while *mcrops.cranecloud.io* had 47.29 s and *49.06 s for the median and mean times respectively*. For experiment **B2**, the results followed a similar pattern. However, experiments **A1** and **A2** performed significantly better under *mcrops.cranecloud.io* for example; in **A2**, *unet.mcrops.org* had 24.43 s and 33.29 s while *mcrops.cranecloud.io* had 10.81 s and 10.999 s for the median and mean times respectively. The completion rates for *mcrops.cranecloud.io* under WiFi was at 100% compared to *unet.mcrops.org* at 88.75%.

3G connection test results. Under the 3G connection, the average response time for experiments **A1**, **A2** and **B2** is lower in *mcrops.cranecloud.io* compared to *unet.mcrops.org* as shown in Fig. 11 and Table 6. In experiment **A2**, for example, *unet.mcrops.org* had *64.37* s and *60.05* s while *mcrops.cranecloud.io* had 27.18 s and 30.48 s for the median and mean times respectively. This was quite significant as the response times for *mcrops.cranecloud.io* were half and very similar patterns for **A1** and **B2**. Despite the presence of an outlier in experiment **A1**, *mcrops.cranecloud.io* still performed better. However, for experiment **B1**, the response time was much lower at an average of 30.97 s using *unet.mcrops.org* compared to *107.46* seconds for *mcrops.cranecloud.io*. This could be attributed to a network congestion or server load at execution time especially given the positive results from **A1**, **A2** and **B2**. In general, the completion rate for *mcrops.cranecloud.io* under 3G was at 100% compared to *unet.mcrops.org* at 85%.

2G connection test results. Using the 2G connection, only two experiments (**A1** and **B1** were completed successfully as shown in Fig. 12 and Table 6. These experiments involved single image data with the loops varied. *mcrops.cranecloud.io* in both experiments had lower response times compared to *unet.mcrops.org* using both the median and mean. In **A1**, *unet.mcrops.org* had 57.33 s and 56.61 s while *mcrops.cranecloud.io* had 38.81 s and 48.48 s for the median and mean times respectively. The spikes in the graphs are attributed to network unreliability especially under constrained capacities supported by 2G.

The computation of the CBSD score is a resource-intensive task as shown by the response times recorded in all the scenarios. The increasing response times are attributed to the ramp-up time where new requests are generated at specific intervals before some computations are concluded. From the results, it is also clear the *mcrops.cranecloud.io* is more consistent in the increasing response times and this is attributed to the completion rates of the execution as shown in Table 7 and Fig. 13. In all instances, *mcrops.cranecloud.io* has a 100% completion rate compared to *unet.mcrops.org* at 89.64%. In cases where the public cloud hosted instance performs better, server errors such as 502 (Bad Gateway) and 504 (Gateway Timeout) were recorded. As expected and shown by the Internet speed results, the WiFi connection performs much better compared to the rest of the connectivity options.

The behavior of the trends as observed in the graphs is due to the Internet speed variations during the course of the experiments. For example, we noticed that experiments done in the morning provide better response times compared to the later hours. This is explained by typical network usage patterns over a 24-hour period that characterize these settings (Alliance, 2021).

[39] http://speedtest.net/

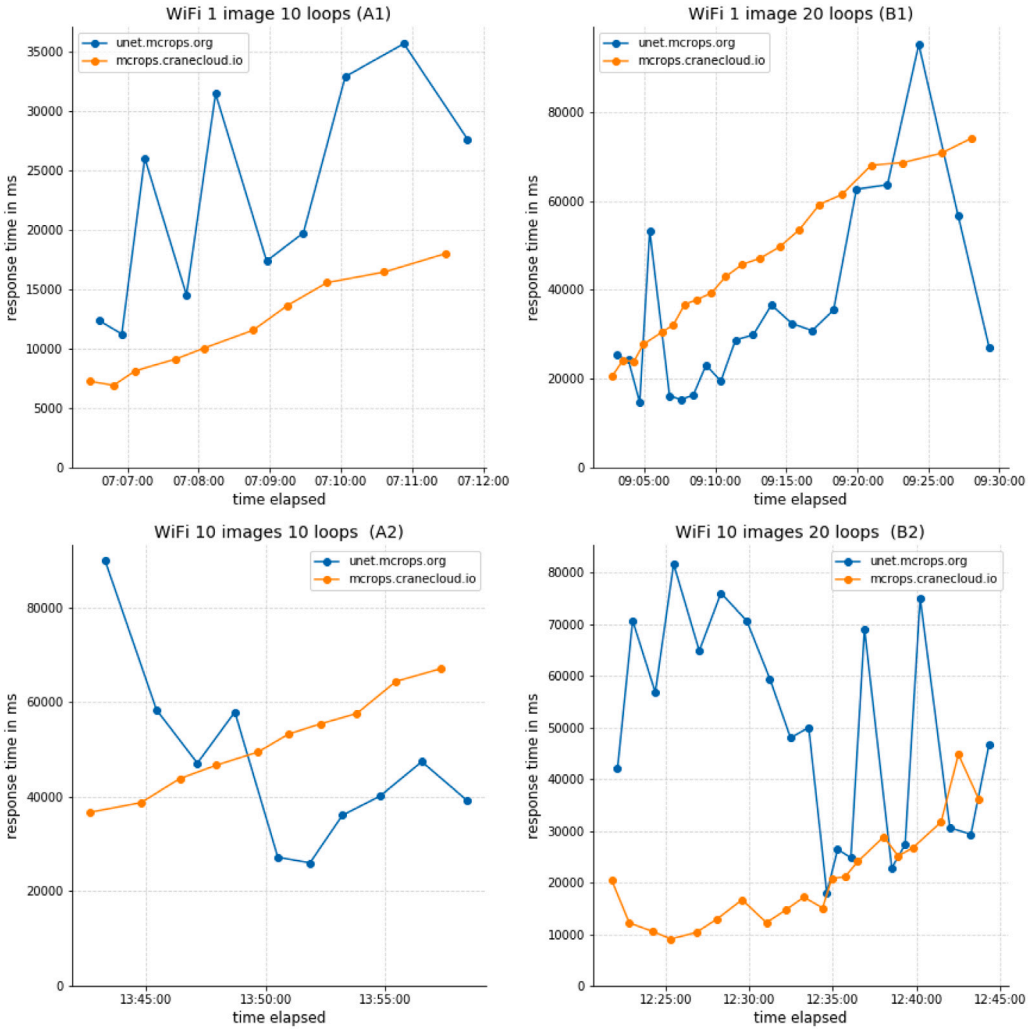

Fig. 9. WiFi connection test results.

Table 6
Comparison of the Execution Response Times *(exp = Experiment, min = Minimum, med= Median, max = Maximum, avg = Average)*.

Sampler ->		unet.mcrops.org (seconds)				mcrops.cranecloud.io (seconds)			
Connection	exp	min	med	max	avg	min	med	max	avg
2G	A1	31.09	57.33	**69.10**	56.61	**28.38**	**38.81**	157.48	**48.48**
	B1	67.34	106.29	**303.91**	143.39	**47.70**	**100.76**	426.60	**141.19**
	A2	–		–	–	–		–	–
	B2	–		–	–	–		–	–
3G	A1	13.31	22.94	**33.91**	22.03	**7.88**	**11.11**	78.03	**17.41**
	B1	**13.51**	**30.97**	**59.10**	**35.26**	69.03	107.46	163.94	111.52
	A2	17.69	64.37	89.80	60.05	**17.52**	**27.18**	**51.97**	**30.48**
	B2	19.04	47.56	75.28	46.19	**11.16**	**26.06**	**55.82**	**30.86**
4G	A1	9.32	28.10	37.37	26.03	**7.95**	**12.50**	**20.48**	**13.19**
	B1	**10.65**	**28.01**	**62.56**	**29.69**	23.16	47.29	80.31	49.06
	A2	12.72	24.43	64.39	33.29	**7.06**	**10.81**	**15.68**	**10.99**
	B2	**16.5**	**27.53**	69.18	**37.75**	21.69	41.05	**66.18**	41.37
WiFi	A1	11.23	22.90	35.70	22.90	**6.91**	**10.78**	**18.03**	**11.66**
	B1	**14.62**	**29.24**	95.23	**35.31**	20.42	44.30	**74.13**	45.67
	A2	**26.01**	**43.61**	89.97	**46.93**	36.74	51.36	**67.07**	51.31
	B2	18.04	49.04	81.70	49.55	**9.09**	**18.80**	**44.81**	**20.55**

The results from this experiment show the response time patterns for applications and services hosted at cloud providers situated in rich-resource settings and accessed from varying resource constrained environments. The results are thus not specific a cloud provider, in this case AWS and the choice of the application but rather a general pattern of applications with similar characteristics.

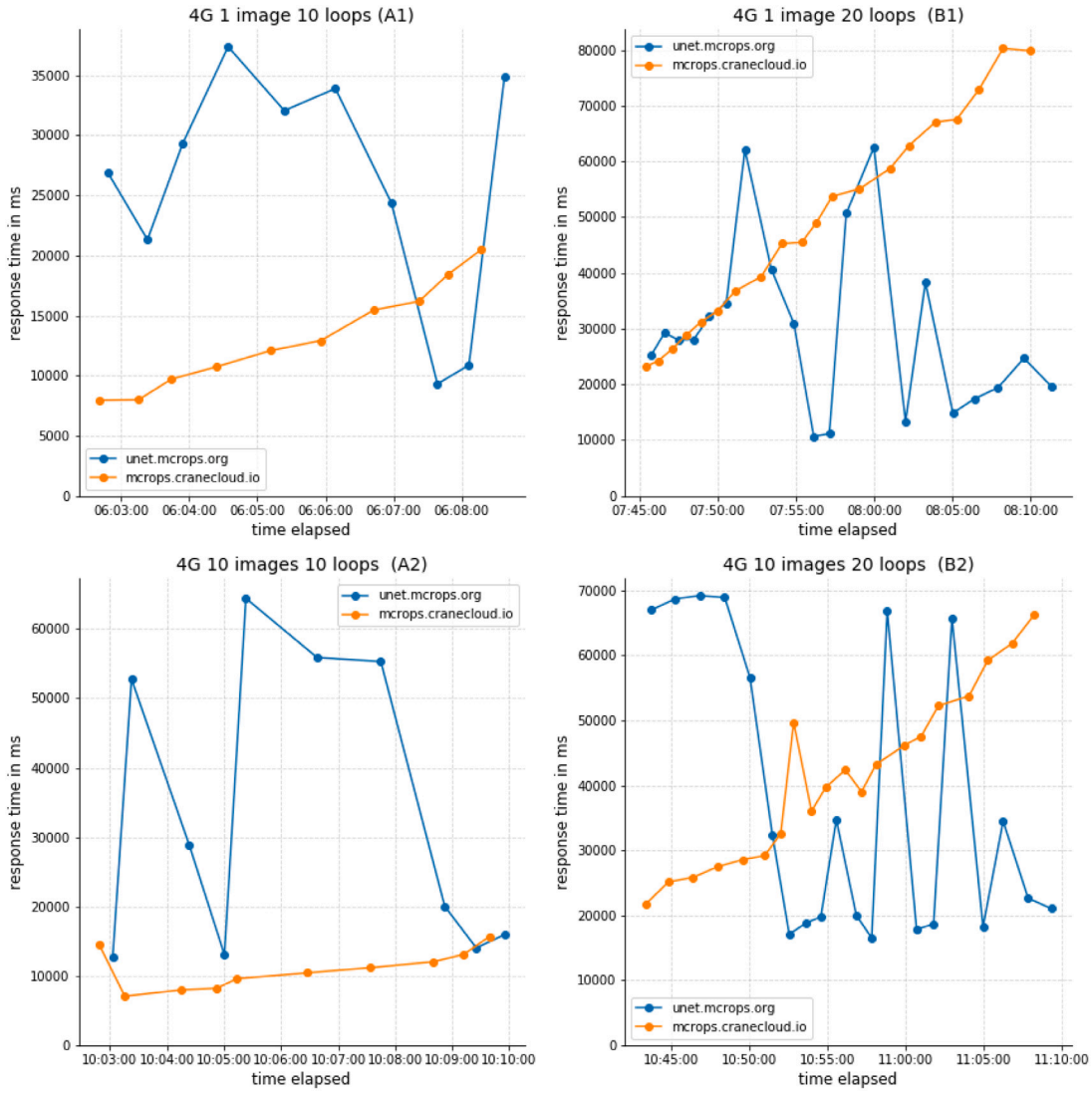

Fig. 10. 4G connection test results.

Table 7
Execution Completion Rates (%).

Connection/Sampler	Experiment	unet.mcrops.org (%)	mcrops.cranecloud.io (%)
2G	A1	90	100
	B1	90	100
	A2	–	–
	B2	–	–
3G	A1	100	100
	B1	90	100
	A2	60	100
	B2	90	100
4G	A1	100	100
	B1	90	100
	A2	90	100
	B2	75	100
WiFi	A1	100	100
	B1	90	100
	A2	100	100
	B2	90	100

6. Related work

Van den Bossche et al. (2011) addressed the challenge of cost-efficiently scheduling deadline constrained batch type applications on IaaS (virtual machines) hybrid clouds using custom heuristics. Properties such as high availability, scalability, fault-tolerance and monitoring are not discussed and the use of virtual machines may not be the most cost-effective approach to running application workloads. Filip et al. (2018) proposed a solution that considers a finite catalog of primitive microservices and designs a hybrid scheduling algorithm that matches tasks to resources based on task history and availability of resources. In addition to benefits of using a microservice architecture, the paper asserted that costs can further be reduced by placing data closer to processing points based on user density. Müssig et al. (2017) describes the concept of a high scalable microservice infrastructure using custom metrics in addition to commonly used ones such as CPU and RAM . In this paper, custom metrics such as service utilization for scaling, load balancing and load prediction often results in better business-alignment of the scaling behavior as well as cost reduction. Guerrero et al. (2018) presented an optimization approach to reduce service cost, microservices repair time, and microservices network latency overhead in the orchestration process of containers in multi-cloud environments using the scale level of the microservices and their allocation in the virtual machines, the provider and virtual machine type selection and the number of virtual machines. Sousa et al. (2016) developed a framework for automated deployment of microservice applications in multi-cloud environments with containers. The application's multi-cloud requirements are defined and a systematic method for obtaining

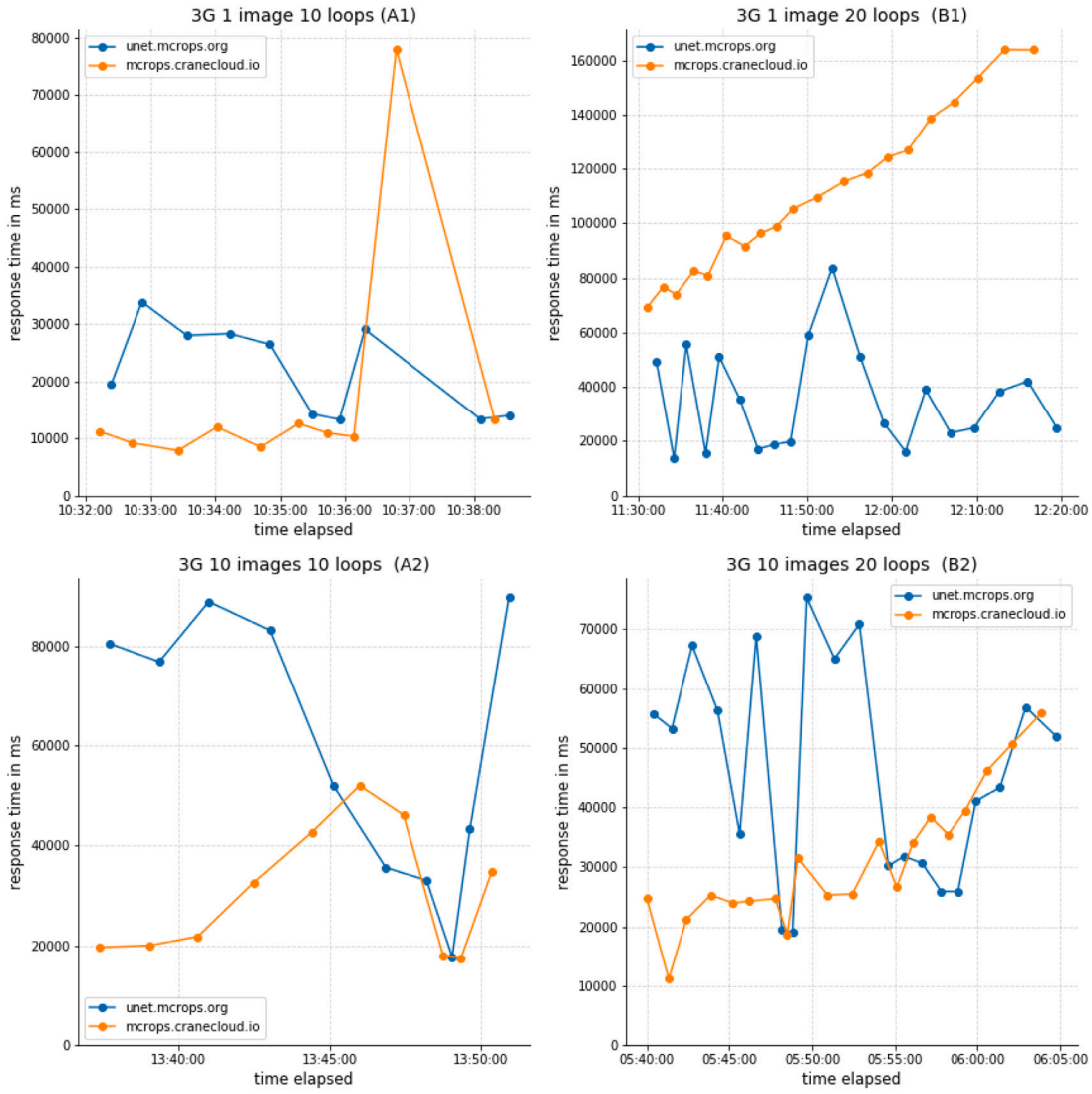

Fig. 11. 3G connection test results.

proper configurations that comply with the application's requirements and the cloud providers' constraints is adopted. Rancher[40] and D2C[41] are two examples of container management platforms that simplify the process of operating container clusters on any cloud or infrastructure platform. The downside with these platforms is in addressing application requirements such as data processing and storage restrictions and the distinctive requirements for resource constrained settings. As more established cloud providers such as Microsoft, Google, Oracle and Amazon move towards hosted cloud-native platforms such as Kubernetes for easier configuration and management, the vendor-lockin issues are expected to exacerbate especially with no plans of integration tools or APIs. In summary, there is no standardized solution for implementation and operation of a multi-cloud service layer but rather blocks that independently address the design considerations in Section 3.

7. Conclusion and future work

In this paper, we presented Crane Cloud - a resilient multi-cloud service layer for resource constrained environments using Kubernetes and assorted management tools. We highlighted the characteristics of a resource constrained environment that includes poor Internet connectivity, frequent Internet partitions and data center power cuts ultimately resulting in poor user experience or even service unavailability. Based on these challenges, we enumerated a number of design considerations and properties for a resilient multi-cloud service layer that would form the foundation for Crane Cloud. From easing terminal complexities of operating a cloud service, desirable scaling, availability, migration and loadbalancing to platform monitoring, Crane Cloud tries to provide an all-inclusive solution that best fits the resource constrained compute environment. As much as Crane Cloud directs implementations for the subject environment, it should be noted there are many moving parts some of which are under active development and research. The bandwidth constraints, for example, may require consensus algorithms for better handling of network splits but there are always trade-offs that should be considered. Management of persistent storage, replication and fault tolerance across geographically distant clusters accessible via Wide Area Networks (WANs) is still an open research area. There are always penalties introduced on the network especially with application data that has to traverse bottleneck links to maintain real-time application consistency. Further future work includes analyzing and optimizing the scheduling processes for applications in production Crane Cloud clusters. In bare metal clusters, service load balancing usually requires more investment in the network

[40] http://rancher.com/
[41] http://d2c.io

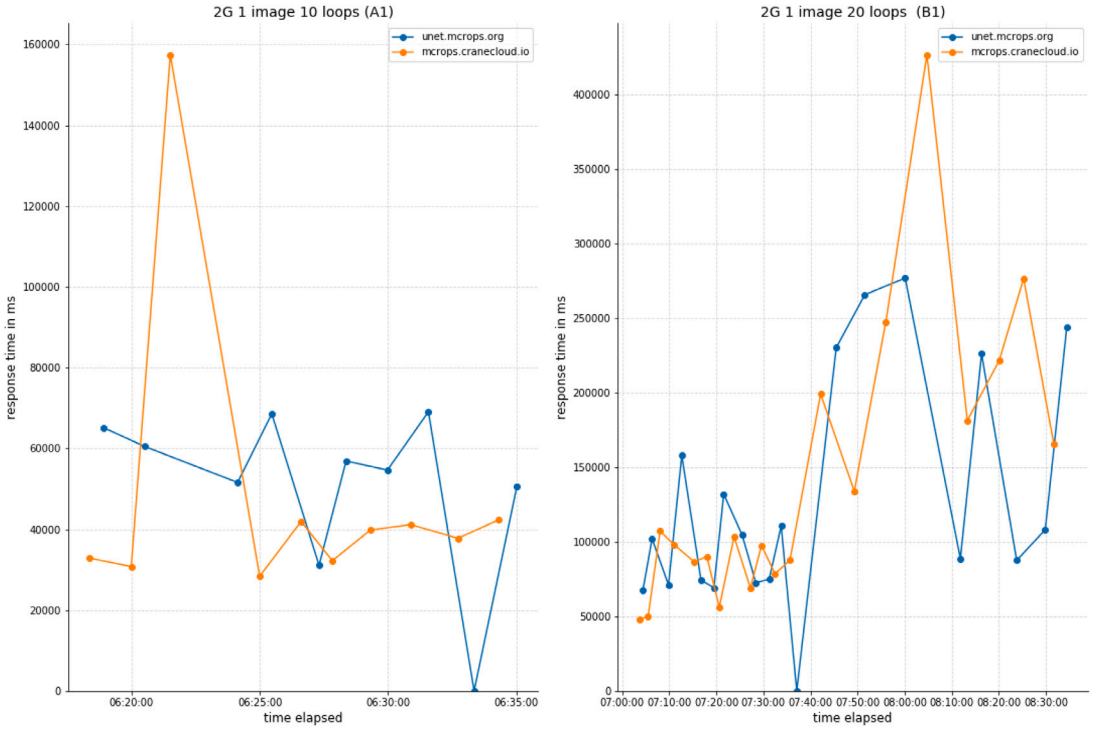

Fig. 12. 2G connection test results.

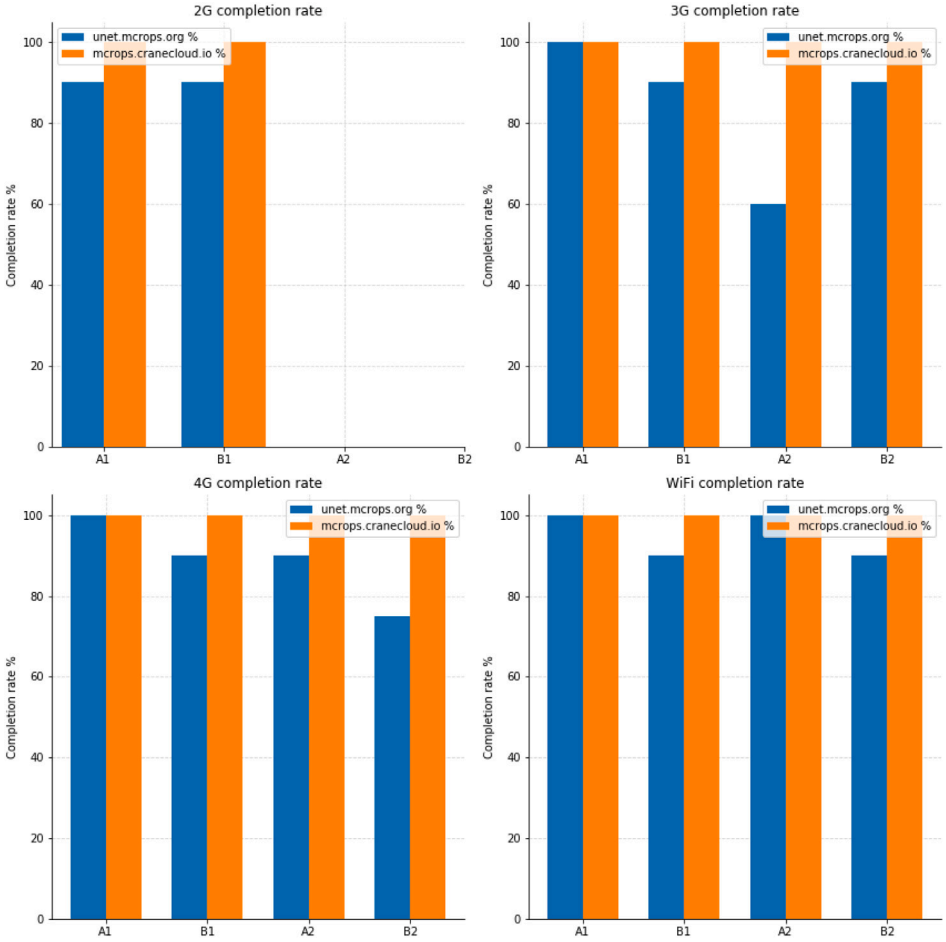

Fig. 13. Instance completion rates.

infrastructure which is not an option in a low resource setting and should be further explored.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

Data will be made available on request.

Acknowledgments

The authors would like to acknowledge all persons who have contributed to the thought process of Crane Cloud and resources in funding or in-kind to bring the project to life. We thank Paul Maritz for the support and advice in the conceptualization of the Crane Cloud project. We thank the Crane Cloud team for feedback and input. The authors would also like to acknowledge support from the Government of Uganda through the Makerere University Research Innovation Fund (RIF).

References

Alabbadi, M.M., 2011. Cloud computing for education and learning: Education and learning as a service (ELaaS). In: 2011 14th International Conference on Interactive Collaborative Learning. IEEE, pp. 589–594.

Alhamad, M., Dillon, T., Wu, C., Chang, E., 2010. Response time for cloud computing providers. In: Proceedings of the 12th International Conference on Information Integration and Web-Based Applications and Services. iiWAS '10, Association for Computing Machinery, New York, NY, USA, pp. 603–606. http://dx.doi.org/10.1145/1967486.1967579.

Alliance, U., 2021. UbuntuNet monitor. https://monitor.ubuntunet.net/cacti/. (Accessed 21 July 2021).

Alquraan, A., Takruri, H., Alfatafta, M., Al-Kiswany, S., 2018. An analysis of network-partitioning failures in cloud systems. In: 13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18). pp. 51–68.

Balalaie, A., Heydarnoori, A., Jamshidi, P., 2016. Microservices architecture enables DevOps: Migration to a cloud-native architecture. IEEE Softw. 33 (3), 42–52. http://dx.doi.org/10.1109/MS.2016.64.

Boru, D., Kliazovich, D., Granelli, F., Bouvry, P., Zomaya, A.Y., 2015. Models for efficient data replication in cloud computing datacenters. In: 2015 IEEE International Conference on Communications. ICC, IEEE, pp. 6056–6061.

Bozman, J., Chen, G., 2010. Cloud computing: The need for portability and interoperability. IDC Exec. Insights.

Brewer, E.A., 2000. Towards robust distributed systems. In: PODC, vol. 7. Portland, OR.

Burns, B., Grant, B., Oppenheimer, D., Brewer, E., Wilkes, J., 2016. Borg, omega, and kubernetes. Queue 14 (1), 70–93.

Calandro, E., Chavula, J., Phokeer, A., 2018. Internet development in africa: a content use, hosting and distribution perspective. In: International Conference on E-Infrastructure and E-Services for Developing Countries. Springer, pp. 131–141.

Corneo, L., Eder, M., Mohan, N., Zavodovski, A., Bayhan, S., Wong, W., Gunningberg, P., Kangasharju, J., Ott, J., 2021. Surrounded by the clouds: A comprehensive cloud reachability study. In: Proceedings of the Web Conference 2021. pp. 295–304.

Daigle, B., 2021. Data protection laws in Africa: A pan-African survey and noted trends. J. Int'L Com. Econ. 1.

DigitBin, 2019. WhatsApp lite APK download for Android. https://www.digitbin.com/whatsapp-lite/. (Accessed 30 June 2021).

Dragoni, N., Dustdar, S., Larsen, S.T., Mazzara, M., 2017. Microservices: Migration of a mission critical system. ArXiv preprint arXiv:1704.04173.

Ecobank Research, 2018. The high cost of mobile data in Sub-Saharan Africa. High data costs are constraining Africa's digital revolution. URL https://www.ecobank.com/upload/publication/20180910054643018QJEBKEVZKD/20180910054635730h.pdf.

Esteves, S., Silva, J., Veiga, L., 2012. Quality-of-service for consistency of data geo-replication in cloud computing. In: European Conference on Parallel Processing. Springer, pp. 285–297.

Filip, I.-D., Pop, F., Serbanescu, C., Choi, C., 2018. Microservices scheduling model over heterogeneous cloud-edge environments as support for iot applications. IEEE Internet Things J. 5 (4), 2672–2681.

Fowler, M., Lewis, J., 2014. Microservices a definition of this new architectural term. p. 22, URL: http://martinfowler.com/articles/microservices.html.

Gao, A., Diao, L., 2010. Lazy update propagation for data replication in cloud computing. In: 5th International Conference on Pervasive Computing and Applications. IEEE, pp. 250–254.

Gillwald, A., Mothobi, O., 2019. After Access 2018: A Demand-Side View of Mobile Internet from 10 African Countries. Research ICT Africa.

Gonidis, F., Paraskakis, I., Kourtesis, D., 2012. Addressing the challenge of application portability in cloud platforms. In: 7th South-East European Doctoral Student Conference. pp. 565–576.

Google LLC, 2019. Lite but packs a punch: Google go comes to Android everywhere. https://www.blog.google/products/search/lite-packs-punch-google-go-comes-android-everywhere. (Accessed 30 June 2021).

Google LLC, 2021. See Gmail in standard or basic HTML version - Gmail Help. https://support.google.com/mail/answer/15049?hl=en. (Accessed 30 June 2021).

Guerrero, C., Lera, I., Juiz, C., 2018. Resource optimization of container orchestration: a case study in multi-cloud microservices-based applications. J. Supercomput. 74 (7), 2956–2983.

Haselböck, S., Weinreich, R., 2017. Decision guidance models for microservice monitoring. In: 2017 IEEE International Conference on Software Architecture Workshops. ICSAW, IEEE, pp. 54–61.

Hashem, I.A.T., Yaqoob, I., Anuar, N.B., Mokhtar, S., Gani, A., Khan, S.U., 2015. The rise of "big data" on cloud computing: Review and open research issues. Inf. Syst. 47, 98–115.

Hasselbring, W., Steinacker, G., 2017. Microservice architectures for scalability, agility and reliability in e-commerce. In: 2017 IEEE International Conference on Software Architecture Workshops. ICSAW, IEEE, pp. 243–246.

Hohpe, G., 2019. Don't get locked up into avoiding lock-in. URL https://martinfowler.com/articles/oss-lockin.html.

Hope, P., 2002. Using jails in freebsd for fun and profit. Login: The Magazine of USENIX & SAGE 27 (3).

Hüttermann, M., 2012. DevOps for Developers. A Press.

Jaramillo, D., Nguyen, D.V., Smart, R., 2016. Leveraging microservices architecture by using Docker technology. In: SoutheastCon 2016. IEEE, pp. 1–5.

Kamp, P.-H., Watson, R.N., 2000. Jails: Confining the omnipotent root. In: Proceedings of the 2nd International SANE Conference, vol. 43. p. 116.

Knoche, H., Hasselbring, W., 2019. Drivers and barriers for microservice adoption-a survey among professionals in germany. Enterprise Modelling and Information Systems Architectures (EMISAJ)-Int. J. Concept. Model. 14 (1), 1–35.

Kratzke, N., et al., 2014. Lightweight virtualization cluster how to overcome cloud vendor lock-in. J. Comput. Commun. 2 (12), 1.

Kshetri, N., 2010. Cloud computing in developing economies. Computer 43 (10), 47–55.

Levijarvi, E.S., Mitzev, O.S., 2015. Private Cloud Replication and Recovery. Google Patents, US Patent 8, 930, 747.

Li, W., Yang, Y., Chen, J., Yuan, D., 2012. A cost-effective mechanism for cloud data reliability management based on proactive replica checking. In: 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing. Ccgrid 2012, IEEE, pp. 564–571.

Li, W., Yang, Y., Yuan, D., 2011. A novel cost-effective dynamic data replication strategy for reliability in cloud data centres. In: 2011 IEEE Ninth International Conference on Dependable, Autonomic and Secure Computing. IEEE, pp. 496–502.

Liu, F., Tong, J., Mao, J., Bohn, R., Messina, J., Badger, L., Leaf, D., 2011. NIST cloud computing reference architecture. NIST Spec. Publ. 500 (2011), 292.

Maaref, S., 2012. Cloud computing in africa situation and perspectives. Telecommun. Dev. Sect.-ITU 70.

Modak, A., Chaudhary, S., Paygude, P., Ldate, S., 2018. Techniques to secure data on cloud: Docker swarm or kubernetes? In: 2018 Second International Conference on Inventive Communication and Computational Technologies. ICICCT, IEEE, pp. 7–12.

Müssig, D., Stricker, R., Lässig, J., Heider, J., 2017. Highly scalable microservice-based enterprise architecture for smart ecosystems in hybrid cloud environments. In: ICEIS (3). pp. 454–459.

Mwebaze, E., Biehl, M., 2016. Prototype-based classification for image analysis and its application to crop disease diagnosis. In: Advances in Self-Organizing Maps and Learning Vector Quantization. Springer International Publishing, pp. 329–339.

Mwotil, A., Bainomugisha, E., Araka, S.G., 2022. Mira: an application containerisation pipeline for small software development teams in low resource settings. In: Proceedings of the Federated Africa and Middle East Conference on Software Engineering. pp. 31–38.

Nadareishvili, I., Mitra, R., McLarty, M., Amundsen, M., 2016. Microservice Architecture: Aligning Principles, Practices, and Culture. O'Reilly Media, Inc.

Nkosi, M., Mekuria, F., 2010. Cloud computing for enhanced mobile health applications. In: 2010 IEEE Second International Conference on Cloud Computing Technology and Science. IEEE, pp. 629–633.

Noor, A., Jha, D.N., Mitra, K., Jayaraman, P.P., Souza, A., Ranjan, R., Dustdar, S., 2019. A framework for monitoring microservice-oriented cloud applications in heterogeneous virtualization environments. In: 2019 IEEE 12th International Conference on Cloud Computing. CLOUD, IEEE, pp. 156–163.

Ongaro, D., Ousterhout, J., 2014. In search of an understandable consensus algorithm. In: 2014 {USENIX} Annual Technical Conference ({USENIX}{ATC} 14). pp. 305–319.

Opara-Martins, J., Sahandi, R., Tian, F., 2016. Critical analysis of vendor lock-in and its impact on cloud computing migration: a business perspective. J. Cloud Comput. 5 (1), 4.

Ranjan, R., Zhao, L., Wu, X., Liu, A., Quiroz, A., Parashar, M., 2010. Peer-to-peer cloud provisioning: Service discovery and load-balancing. In: Cloud Computing. Springer, pp. 195–217.

Rolim, C.O., Koch, F.L., Westphall, C.B., Werner, J., Fracalossi, A., Salvador, G.S., 2010. A cloud computing solution for patient's data collection in health care institutions. In: 2010 Second International Conference on EHealth, Telemedicine, and Social Medicine. IEEE, pp. 95–99.

Sabharwal, N., Pandey, S., Pandey, P., 2021. Getting started with nomad. In: Infrastructure-As-Code Automation using Terraform, Packer, Vault, Nomad and Consul. Springer, pp. 201–236.

Sahandi, R., Alkhalil, A., Opara-Martins, J., 2013. Cloud computing from SMEs perspective: a survey based investigation. J. Inf. Technol. Manage. 24 (1), 1–12.

Shankar, V., 2015. Announcing facebook lite. Facebook Newsroom.

Shi, T., Ma, H., Chen, G., Hartmann, S., 2020. Location-aware and budget-constrained service deployment for composite applications in multi-cloud environment. IEEE Trans. Parallel Distrib. Syst. 31 (8), 1954–1969.

Sousa, G., Rudametkin, W., Duchien, L., 2016. Automated setup of multi-cloud environments for microservices applications. In: 2016 IEEE 9th International Conference on Cloud Computing. CLOUD, IEEE, pp. 327–334.

Sultan, N., 2010. Cloud computing for education: A new dawn? Int. J. Inf. Manage. 30 (2), 109–116.

Uber, 2021. Uber lite - fast, reliable, and just 5MB. https://www.uber.com/ug/en/u/uber-lite-app/. (Accessed 30 June 2021).

Van den Bossche, R., Vanmechelen, K., Broeckhove, J., 2011. Cost-efficient scheduling heuristics for deadline constrained workloads on hybrid clouds. In: 2011 IEEE Third International Conference on Cloud Computing Technology and Science. IEEE, pp. 320–327.

von Wielligh, R.J., Grobler, M.J., Marais, H.-J., 2018. Cellular IoT capacity estimation for african smart cities. In: 2018 IEEE Global Conference on Internet of Things. GCIoT, IEEE, pp. 1–6.

Xiong, K., Perros, H., 2009. Service performance and analysis in cloud computing. In: 2009 Congress on Services - I. pp. 693–700. http://dx.doi.org/10.1109/SERVICES-I.2009.121.

Zhang, W., Chen, Q., 2010. From E-government to C-government via cloud computing. In: 2010 International Conference on E-Business and E-Government. IEEE, pp. 679–682.

Zhang, D., Yan, B., Feng, Z., Zhang, C., Wang, Y., 2017. Container oriented job scheduling using linear programming model. In: 2017 3rd International Conference on Information Management. ICIM, pp. 174–180.

Zhou, J., Shi, Z., 2010. Unstructured P2P-enabled service discovery in the cloud environment. In: International Conference on Intelligent Information Processing. Springer, pp. 173–182.