SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Development and Automation of Reliable Cloud Infrastructure for Scalable Microservices Deployment

Ana-Maria-Iulia Cornea

SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Development and Automation of Reliable Cloud Infrastructure for Scalable Microservices Deployment

Entwicklung und Automatisierung einer zuverlässigen Cloud-Infrastruktur für die skalierbare Bereitstellung von Microservices

Author:	Ana-Maria-Iulia Cornea
Supervisor:	Prof. Dr. Pramod Bhatotia
Advisor:	Evgeny Volynsky, M.Sc.
Submission Date:	16.10.2023

I confirm that this master's thesis in informatics is my own work and I have documented all sources and material used.

Munich, 16.10.2023 Ana-Maria-Iulia Cornea

Acknowledgments

I would like to thank Prof. Pramod Bhatotia for giving me the opportunity to write my Master's Thesis at the Chair of Distributed Systems & Operating Systems in the Informatics Department of Technical University of Munich.

I would also like to thank my advisor, Evgeny Volynsky, for his support and continuous feedback during the implementation and writing phases, which have been indispensable in achieving this manuscript.

Abstract

In the modern digital landscape, the demand for scalable and efficient systems is highly present, especially in domains managing vast volumes of sensitive data such as the Institution of German Human Genome-Phenome Archive (GHGA). The necessity to upgrade the existing infrastructure to ensure a reliable, scalable, and seamless deployment of microservices forms the core motivation of this thesis. At the heart of this project is the current OpenStack-based infrastructure, whose transformation is envisioned through the adoption of Infrastructure as Code (IaC), Kubernetes, and GitOps technologies. Central to this work is the primary question of how to design, implement, and automate a cloud infrastructure that encapsulates the dynamic essence of microservices while concurrently ensuring reliability, scalability, and cost-effectiveness. The initiated research journey uses a multi-tiered methodology with three main layers: building a specialized OpenStack infrastructure, orchestrating a Kubernetes cluster deployment atop OpenStack with the aid of Terraform, and integrating GitOps through ArgoCD for refined automation in the deployment processes.

The resulting prototype stands as a robust platform, primed for seamless scalability to align with production-ready infrastructure demands as resources allow. This architecture establishes a foundational benchmark for subsequent projects in this domain, spotlighting the novel integration of OpenStack, Kubernetes, and GitOps. Such a framework is particularly significant for scalable microservices deployment, especially when handling sensitive data is critical. The comprehensive source code, paired with installation guidelines and additional implementation details, are available at https://github.com/Evgeny-Volynsky/microservices-infrastructure. This resource serves as a vital reference for those seeking deeper insights and further exploration in the domain of microservices deployment in sensitive data contexts.

Kurzfassung

In einem modernen digitalen Umfeld besteht eine hohe Nachfrage nach skalierbaren und effizienten Systemen, insbesondere in Bereichen, die große Mengen sensibler Daten verwalten, wie das Institut für das Deutsche Humangenom-Phänom-Archiv (GHGA), das als Auftraggeber dieser der Arbeit zu nennen ist. Die Notwendigkeit, die bestehende Infrastruktur des Genomarchivs zu aktualisieren, um die sichere Speicherung, den Zugriff und die Analyse menschlicher Omics-Daten (z.B. Genome, Transkriptome) durch eine zuverlässige, skalierbare und nahtlose Bereitstellung von Mikrodiensten zu gewährleisten, bildet die Kernmotivation. Im Zentrum des Projekts in Kooperation mit dem GHGA steht die aktuelle auf OpenStack basierende Infrastruktur, deren Transformation durch die Adoption von Infrastructure as Code (IaC), Kubernetes und GitOps-Technologien angestrebt wird. Dabei konzentriert sich diese Arbeit auf die Beantwortung der primären Frage, wie eine Cloud-Infrastruktur so entworfen, implementiert und automatisiert werden kann, dass sie das dynamische Wesen von Mikrodiensten abbildet, während gleichzeitig Zuverlässigkeit, Skalierbarkeit und Wirtschaftlichkeit sichergestellt werden.

Die eingeleitete Forschungsreise verwendet eine mehrstufige Methodik mit drei Hauptebenen: Zunächst wird eine spezialisierte OpenStack-Infrastruktur aufgebaut. Danach folgt die Orchestrierung einer Kubernetes-Cluster-Bereitstellung auf OpenStack mit Hilfe von Terraform und zuletzt die Integration von GitOps durch ArgoCD für verfeinerte Automatisierung in den Bereitstellungsprozessen. Der resultierende Prototyp dient als robuste Plattform, ausgerichtet für nahtlose die nahtlose Skalierbarkeit, um sich an die Anforderungen einer produktionsreifen Infrastruktur anzupassen, sofern die Ressourcen dieses Vorgehen zulassen. Die vorgeschlagene Architektur etabliert ein grundlegendes Benchmark für nachfolgende Projekte in dem Bereich der Microservice-Bereitstellung und beleuchtet die neuartige Integration von OpenStack, Kubernetes und GitOps. Ein solches Framework ist insbesondere für die skalierbare Bereitstellung von Microservices von Bedeutung, wenn der Umgang mit sensiblen Daten kritisch ist. Der umfassende Quellcode sowie die Installationsskripte und die Implementierungsdetails sind verfügbar unter: https://github.com/Evgeny-Volynsky/microservices-infrastructure. Diese Ressource dient als Referenz für tiefere Einblicke und Erkundungen im Bereich der Bereitstellung von Mikrodiensten in Kontexten mit sensiblen Daten.

Contents

1. Introduction

In today's rapidly evolving technological environment, the need for scalable and efficient systems has become dominant, especially in domains with vast amounts of sensitive data. The German Human Genome-Phenome Archive (GHGA) stands as a significant example, handling sensitive data and requiring an infrastructure upgrade to ensure a seamless, reliable, and scalable deployment of microservices. While OpenStack has been at the heart of the current infrastructure, the requirement to make this infrastructure more dynamic and efficient necessitates shifting towards Infrastructure as Code (IaC) and integrating container orchestration tools like Kubernetes.

However, a review of the current state-of-the-art reveals a significant gap: there exists no fully automated, reliable, and seamless deployment pipeline tailored to microservices. While a myriad of Continuous Integration (CI) and Continuous Deployment (CD) tools exist, the challenge remains in achieving a fully automated continuous delivery pipeline that is focused primarily on the CD aspect. Using GitOps, this thesis aspires to bridge this gap by focusing on scalable deployment of microservices and leaving the out-of-scope requirements such as microservices testing and security considerations for the following stages of the GHGA's project.

The central research question that arises is: How can a cloud infrastructure, designed using OpenStack and Kubernetes, ensure scalable and reliable microservice deployment with a simplified setup process in the context of data-sensitive applications like human omics data? To tackle this question, a multi-tiered methodology has been adopted. Given that OpenStack is a non-negotiable requirement for this project, an in-depth understanding of its services is indispensable to discern the elements required for our infrastructure and the suitable installation methods. Following this, the research on the methods of deploying Kubernetes clusters on the OpenStack infrastructure is initiated. In this scenario, Terraform is a valuable tool, promoting a robust and highly scalable infrastructure. The ultimate layer includes GitOps technologies, offering a declarative command over infrastructure, thus improving traceability, visibility, and efficiency in the deployment process, especially within a microservices architecture. A detailed comparison between two leading GitOps platforms (presented in Chapter 5) leads to the choice of Argo CD due to its notable flexibility, user-friendly interface, and proficient user and permissions management, easing the development of an Access Control List (ACL) system.

The implications of successfully developing and automating this cloud infrastructure are numerous. Firstly, organizations stand to benefit from reduced deployment times, operational overhead and minimized human intervention. Concurrently, enhanced reliability in deployment strategies reduces downtimes and subsequently improves user satisfaction. The scalability inherent in the infrastructure ensures efficient resource utilization and reduced operational costs.

The thesis is structured into 7 primary chapters:

- **Chapter 1 - Introduction** provides the context and motivation of the thesis, articulating the problem statement, identifying the state-of-the-art components of this work, and offering an overview of the design and implementation.
- **Chapter 2 - Background** introduces all the theoretical concepts required for a comprehensive understanding of this thesis, laying the foundational knowledge upon which this project is built. Major Topics such as Cloud Computing, Microservices, DevOps, and GitOps were presented, along with an introduction to the technologies used in this project: OpenStack, Kubernetes, and Argo CD.
- **Chapter 3 - Design** formalizes the design goals and proceeds with a requirements analysis. Based on this analysis, several possible solutions are explored to improve the project's design and features. Moreover, this chapter concludes with an overview of system components and their interactions.
- **Chapter 4 - Implementation** describes in detail all three stages of development, concluding with a system overview that illustrates the project's overall architecture.
- **Chapter 5 - Evaluation** details how all technical and implementation-related decisions were made, providing evidence and comparisons between the researched technologies to justify our selections adequately.
- **Chapter 6 - Conclusion** provides a general overview of the achievements of this research and a presentation of the impact of this thesis on the main research project that is part of: "Reliable GHGA Infrastructure Using OpenStack: Safe, Secure, and Scalable Deployment of Microservices".
- **Chapter 7 - Future Work** situates the current work within the context of the main research project while presenting the next planned stages to be completed and the upcoming project goals to be fulfilled.

Delving further into this thesis reveals the innovative efforts initiated to automate and optimize microservices deployment, thereby establishing a benchmark for future projects in this domain and laying foundational work for addressing additional considerations and enhancements in microservices deployment in future stages of the GHGA's project.

2. Background

2.1. Cloud Computing Infrastructure

Cloud Computing has more than 20 definitions according to [1], but the one from the National Institute of Standards and Technology (NIST) encompasses all fundamental features of cloud computing:

"Cloud Computing is a model for enabling convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction." [2]

Cloud Computing Architecture consists mainly of 4 layers that can be visualized in Figure 2.1:

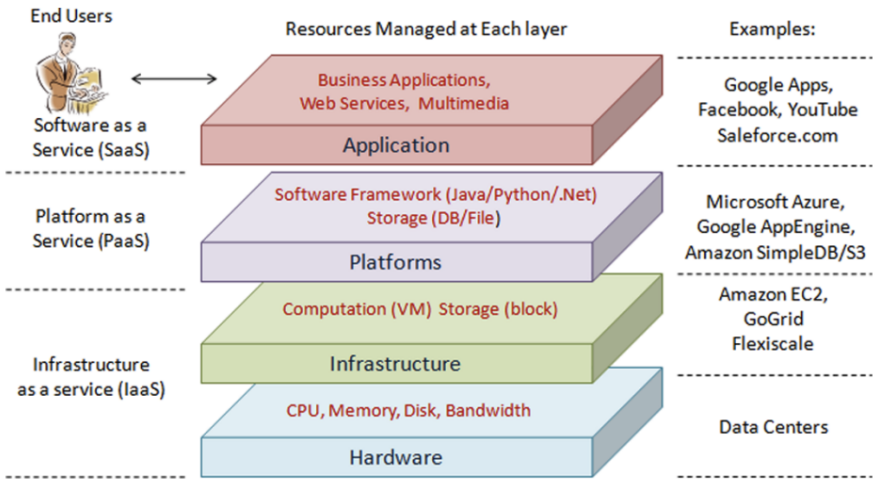

Figure 2.1.: Cloud Computing Architecture with examples for each type of service [3]

1. **The hardware layer** operates within data centers and regulates the physical cloud resources, which usually include servers, routers, switches, power and cooling systems. [3]
2. **The infrastructure layer**, also referred to as the virtualization layer, applies virtualization technologies like Xen [4], Kernel-based Virtual Machine (KVM)[5] and

VMWare[6] to partition physical resources, forming a pool of storage and computing resources. This layer is crucial in cloud computing as it enables significant functions like dynamic resource allocation that can only be achieved through virtualization techniques.

3. **The platform layer** runs on top of the infrastructure layer and encompasses operating systems and application frameworks. Its primary objective is to reduce the complexity of deploying applications directly onto Virtual Machine (VM) containers.

4. **The application layer** is the last layer of a cloud computing architecture and comprises tangible cloud applications. Diverging from conventional applications, cloud applications can utilize the automatic scaling functionality to attain improved performance, availability, and reduced operational expenses.

Overall, in contrast to conventional service hosting environments like dedicated server farms, the structure of cloud computing exhibits high modularity as "each layer is loosely coupled with the layers both above" [3] and below, enabling independent evolution of each layer.

Different application requirements have led to different types of cloud providers, each with its own set of advantages and disadvantages:

- **Public cloud** involves service providers offering their resources as services to the general public. This arrangement presents significant advantages to service providers, including the absence of upfront infrastructure investments and the transfer of risks to infrastructure providers. Nonetheless, the public cloud has limitations in controlling data, network, and security settings, which can hinder its effectiveness in various business cases. [3] Examples: Amazon Web Services (AWS) and Google Cloud Platform (GCP)

- **Private cloud** is intended to be solely used by single organizations. Also referred to as Internal Cloud, it can be established and run by the organization or external providers. While private cloud grants the utmost level of authority over performance, reliability, and security, it is also sometimes criticized for resembling conventional proprietary server farms and not delivering advantages like no initial infrastructure investments. [3] Examples: Hewlett Packard Enterprise and VMWare

- **Hybrid cloud** merges public and private cloud models to overcome their drawbacks. It combines the private and public cloud infrastructure, granting enhanced flexibility as compared to the other models. The hybrid cloud offers improved control and data security compared to the public cloud and the ability to scale services as needed. However, creating a hybrid cloud demands careful planning to determine the optimal balance between the public and private cloud components.[3] Examples: Red Hat and IBM

- **Virtual Private Cloud** is an alternative to hybrid cloud and tries to bridge the gaps between public and private clouds. It operates as a layer built on top of the public cloud, utilizing the Virtual Private Network (VPN) technology to enable customized network topology and security controls such as firewalls. Besides virtualizing servers and applications, Virtual Private Cloud (VPC) extends to encompass the underlying communication network while offering a smooth shift from traditional proprietary service setups to cloud-based systems due to its virtualized network layer.[3] Examples: Amazon VPC, Google Cloud VPC

- **Community cloud** is shared among several organizations with similar interests, requirements, or compliance considerations. It's a way for organizations within a specific industry or niche to pool resources while maintaining a level of isolation from the general public. [7] Example: Gaia-X.

Compared to the more traditional infrastructure models, Cloud Computing offers many advantages to business owners and developers. Some of them are [3]:

- **Small initial investments**: In the case of public cloud and VPC, business owners are not required to buy their hardware to start developing their products. Instead, they can rent only the necessary amount of resources based on their needs without worrying about possible later scalability needs.

- **Reduced operation costs**: Service providers can allocate and de-allocate cloud resources as needed due to cloud computing's "pay per use" model. This model, adopted by many cloud service providers, ensures customers pay only for the resources and services they use, rather than a constant fixed amount.

- **High scalability and accessibility**: As the infrastructure providers aggregate substantial resources from data centers and ensure convenient accessibility, the service provider can easily adapt its infrastructure needs according to their current demand, in most cases in real-time and directly on the internet.

- **Mitigating operational risks and maintenance costs**: Through the delegation of service infrastructure to cloud computing providers, the business owner transfers the operational risks (e.g., hardware failures or security concerns) to the cloud provider, who has greater expertise and usually more resources to handle these challenges.

- **Improved Software Development cycle**: As compared to traditional approaches, Cloud computing has shifted key responsibilities from developers to infrastructure providers, reducing time spent on setup and maintenance in the development cycle. This allows developers to focus more on product feature development and enhances cross-platform collaboration among teams. Investing more in IT capabilities, developers can thus contribute to business growth and increased revenue. [8]

Lastly, this chapter explores cloud computing, discussing its architecture, features, and advantages over traditional models, and analyzes various cloud types tailored to distinct industry needs. This foundation prepares for further examination of OpenStack - an open-source platform that facilitates the creation and management of private and public clouds, emphasizing flexibility and scalability.

2.1.1. OpenStack

OpenStack, as depicted in various scientific papers, is an open-source software collection intended for businesses and cloud service providers to construct and oversee their cloud systems, as per [9]. It encourages a cooperative atmosphere with a goal of creating an accessible, highly scalable, and enriched cloud infrastructure. Meanwhile, [10] describes OpenStack as a framework offering Infrastructure as a Service (IaaS) with a spectrum of resources like computing, storage, and networking. It also acts as a controller for numerous hypervisors and provides tools for managing cloud assets. From these descriptions, key characteristics can be distilled such as [11]:

- **Scalability**: OpenStack has been implemented globally in enterprises with massive data amounts reaching petabytes. It is designed for distributed systems and can easily scale up to accommodate as many as 1 million physical machines, up to 60 million virtual machines, and billions of stored objects.
- **Compatibility and Flexibility**: OpenStack is compatible with a wide range of virtualization options available in the market, such as ESX, Hyper-V, KVM, LXC, QEMU, UML, Xen, and XenServer.
- **Openness**: As an open-source technology, the complete code of OpenStack can be altered and customized according to specific requirements. The OpenStack project also offers a validation procedure for adopting and creating new standards.

Having these features, OpenStack focuses on enabling the computer industry to create an open-source hosting infrastructure with significant scalability, overcoming limitations associated with proprietary technologies. It aims to provide a platform for building extensive and scalable hosting architectures while promoting openness [11].

The following Figure 2.2 provides a comprehensive and up-to-date depiction of the various components and services constituting the OpenStack suite as of May 1, 2023. It groups these components based on their functional areas, offering an insightful overview of the rich landscape that forms the backbone of OpenStack's powerful capabilities.

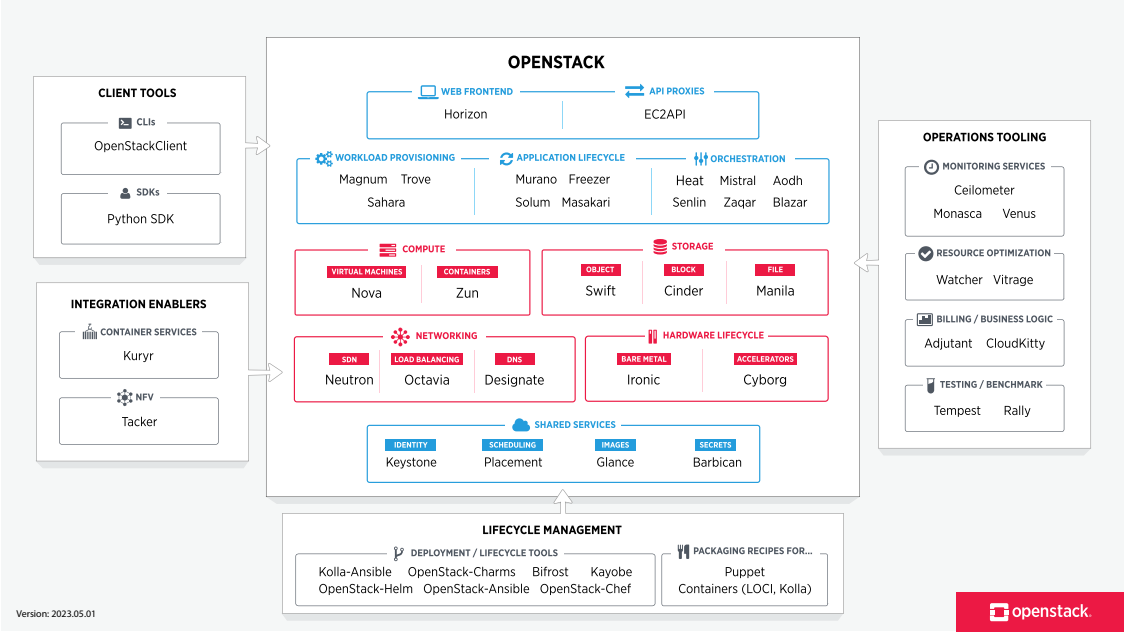

Figure 2.2.: Overview of OpenStack Services, grouped by responsibility and adjacent tools [12]

Based on the diagram above, we can identify five primary clusters comprising the OpenStack architecture, which we will delve into as outlined below:

- **Computing**: One of the main OpenStack services, *Nova*, also known as OpenStack Compute, is responsible for facilitating the provisioning of compute instances (e.g., virtual servers). This project handles virtual machine creation, facilitates bare-metal server deployment, and offers limited support for system containers. [13]
- **Networking**: *Neutron* manages all networking aspects within the OpenStack environment, including both virtual and physical layers. It enables the creation of complex virtual network setups, providing features like firewalls and VPNs. Neutron employs abstractions like networks, subnets, and routers to replicate the functions of their physical counterparts. [14]
- **Storing**: *Cinder* serves as OpenStack's Block Storage service, offering volumes to a variety of platforms such as Nova VMs and containers. Its design emphasizes a component-based architecture for smooth incorporation of new functionalities, along with high availability to handle substantial workloads, fault tolerance through isolated processes to prevent widespread failures, and a focus on easy recovery and debugging in case of failures. [15]
- **Shared Services**: *Keystone* serves as the central authentication point for OpenStack services, providing service discovery and distributed multi-tenant authorization

using its Application Programming Interface (API) [16]. *Glance* offers an Image service within OpenStack, allowing users to upload and find data assets intended for use with other services, encompassing images and metadata definitions. Glance's services involve locating, registering, and fetching VMs images, facilitated through its Representational State Transfer (REST) API and enabling metadata queries and image retrieval. [17]

In Figure 2.3, we can observe the OpenStack Conceptual Architecture illustrating the integration of key services within OpenStack and demonstrating their interaction via public APIs for end-user engagement, including dashboard, Command-Line Interface (CLI)s, and APIs, all authenticated through a shared Identity service.

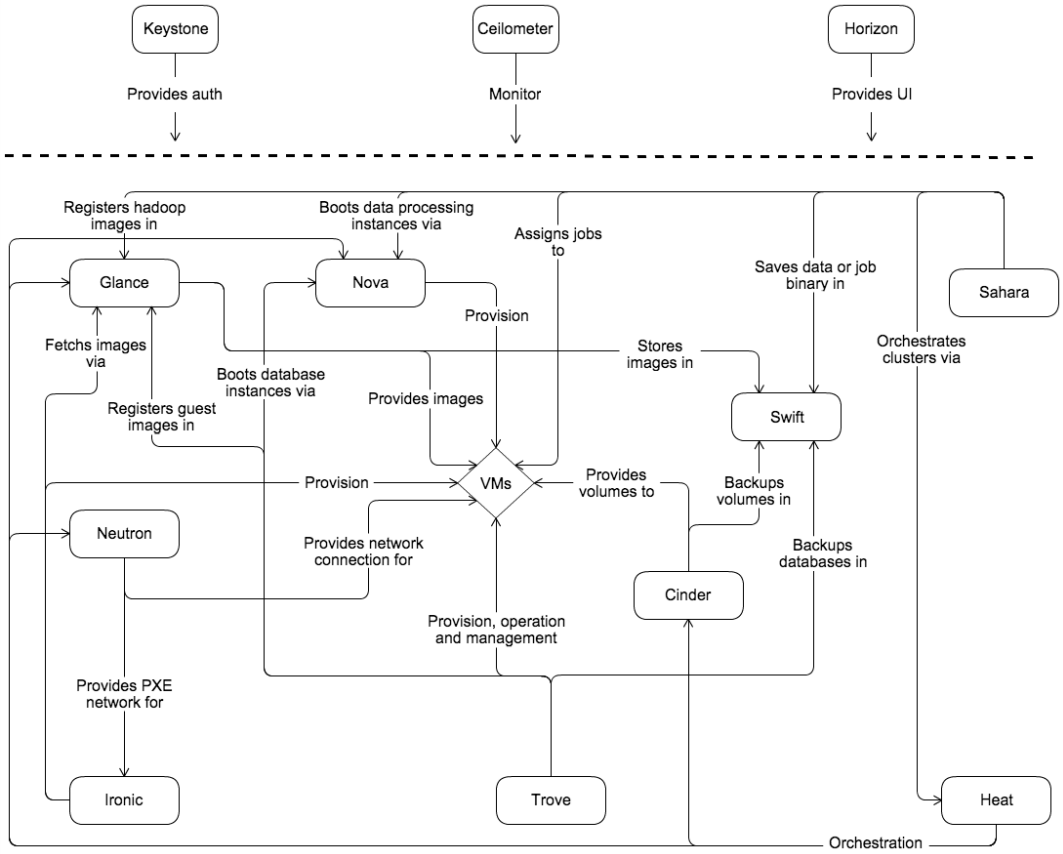

Figure 2.3.: OpenStack Conceptual Architecture presenting the interaction between all available OpenStack Services [18]

In our system design, OpenStack is a non-negotiable requirement. This decision was supported by evaluating its crucial elements, which delivered positive outcomes regarding performance, reliability, and end-user approval. The platform's modular architecture and straightforward implementation, apt for both large-scale data centers and smaller

hardware configurations, played a pivotal role in meeting our system's specifications. Such an approach allows projects like ours to develop a remarkably scalable cloud infrastructure. Furthermore, the open-source nature of OpenStack not only helps in efficient cost management but also ensures a prompt allocation of resources to end-users as needed. [19]

2.2. Microservices

In "Microservices: yesterday, today, and tomorrow" [20], the authors define the microservice as "a minimal independent process interacting via messages" and the microservice architecture as "a distributed application where all its modules are microservices". These two definitions seem simplistic. In 2014, Martin Fowler didn't give a hard definition of microservices at that time rather, he focused on the properties of this new architectural design, which he introduced as follows after several talks with different major players from the tech industry:

"In short, the microservice architectural style is an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an Hypertext Transfer Protocol (HTTP) resource API. These services are built around business capabilities and independently deployable by fully automated deployment machinery. There is a bare minimum of centralized management of these services, which may be written in different programming languages and use different data storage technologies." [21]

From this short history, we can conclude that microservices have first been adopted in the IT industry as individual solutions to the problems raised by the previously classical approaches, such as monoliths, lacking any predefined standards. However, nowadays, after the field of microservices has become a point of interest for the academic world, we can structure some of the key features of microservices as follows [20] [22]:

- Limited functionality results in a small code base, which restricts the potential scope of bugs. Microservices' independent nature allows developers to test and examine their functionalities in isolation from the rest of the system.
- Microservices exhibit resilience, an additional advantage where the failure of one component within the system doesn't impact the entire system.
- Gradual transitions to new versions are now possible. Microservices allow new versions to be deployed alongside older ones, enabling gradual adaptation of services that depend on them.
- Changes in a single module within a microservices architecture don't necessitate a complete system reboot. Only the affected microservice needs to be rebooted.

Moreover, the reduced size of microservices facilitates development, testing, and maintenance with minimal redeployment downtimes.

- Microservices are well-suited for containerization, providing developers flexibility in configuring deployment environments based on their specific needs, including factors like cost and service quality.
- Scaling a microservices architecture doesn't require duplicating all components or scaling all components equally. Developers can adapt each instance's size based on their load, promoting efficient resource utilization.
- Interoperating microservices are constrained only by communication technology (media, protocols, data encoding), empowering developers to freely select the best resources (languages, frameworks, etc.) for each microservice's implementation.

To better visualize the structure of microservices and see its benefits, we can have a look at the following example presented in Figure 2.4:

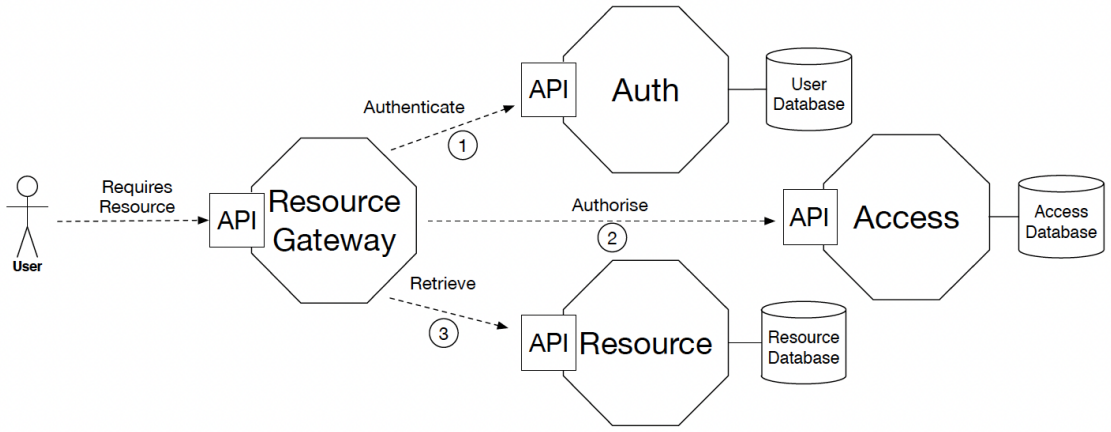

Figure 2.4.: Example of Microservice Architecture that handles the authorized access to resources [20]

The above-presented architecture outlines an application managing authorized access to resources, consisting of three core microservices: Auth (user authentication), Access (resource access authorization), and Resource (resource storage/retrieval). Developers focus on building these foundational functionalities while using a Resource Gateway to coordinate interactions: verifying identities via Auth, enabling access through Access, and retrieving resources via Resource upon authorization.[20]

While microservices offer notable benefits, improper migration from a monolithic architecture can result in a distributed monolith, combining the drawbacks of both distributed systems and single-process monoliths without reaping the benefits. Despite similarities to a Service-Oriented Architecture (SOA), distributed monoliths often result

from insufficient adherence to principles like information hiding and cohesive business functionality, causing tightly interconnected architectures and unintended disruptions across service boundaries due to minor local changes [23].

In conclusion, microservices are standalone components deployed independently with dedicated memory and persistence tools. Their architecture, which allows flexibility in programming paradigms, orchestrates components through messaging, guiding the partitioning of a distributed application into focused, easily managed entities, thereby benefiting development, testing, and project management. [20]

2.3. Deployment Automation and Management of Containerized Applications

Containers are pivotal in enhancing application agility and facilitating efficiency gains as cloud applications progress from bare metal to virtualized machines and beyond, enabling diverse new applications. Organizations leverage containers for various purposes such as long-running services, large-scale batch processing, control planes, Internet of Things, and artificial intelligence workloads, demonstrating their versatility. [24]

Asif Khan defines **containers** in his paper [24] as "a software encapsulation of an operating system process that allows the process to have its own private namespace and computational resources limits including memory and CPU". The standards formulated by the Open Container Initiative [25] with regard to operational paradigms with containers include principles such as independence and portability. A container operates independently of higher-level constructs, such as specific clients or orchestration frameworks. This autonomy allows containers to be effortlessly transferred across diverse operating systems, hardware configurations, Central Processing Unit (CPU) architectures, and public cloud environments, underscoring their exceptional portability.

In the realm of microservices, large applications with hundreds of containerized service instances must ensure fault tolerance, availability, scalability, and reliability for smooth operation. Container orchestration platforms were developed to simplify the management of these requirements by handling cluster state scheduling, ensuring fault tolerance/high availability, simplifying networking, enabling service discovery, facilitating continuous deployment, and providing monitoring/governance features [24]. Kubernetes, the platform utilized in our research, will be introduced in the subsequent section.

Numerous organizations are adopting DevOps and CI/CD techniques to address the slow service delivery seen in conventional IT practices for more efficient software development and customer service. DevOps bridges the divide between developers and operations by unifying them in a flexible and scalable cloud infrastructure. This ap-

proach automates the software delivery process through continuous integration, delivery, and deployment. [26]

In Henry van Merode's book [27], he introduces the topics of Continuous Integration (Continuous Integration), Continuous Delivery (Continuous Deployment), and Continuous Deployment (also CD) as follows.

Continuous integration relies on application code being housed in a source control management system (SCM). Any alterations to this code initiate an automated creation process, resulting in a build artifact saved in a primary repository. This creation process is consistent; thus, the same code should yield the same outcome every time. This entire operation occurs on a designated machine called the integration or build server, which is optimized for swift build execution. [27]

Continuous delivery ensures that there is always a stable primary code line from which deployment to production can occur at any moment. This primary code line remains ready for production by using automated testing and deployment procedures. A singular artifact is constructed and stored in a central repository, so both testing and production deployments utilize this same artifact, maintaining consistency across environments. Every build undergoes testing on a machine mirroring the real production setup, so successful tests on this machine suggest compatibility with the actual production environment. These comprehensive tests ensure the software meets all functional and non-functional criteria. The DevOps team stays informed about the continuous delivery progression through immediate feedback from the integration server, establishing efficient feedback cycles. [27]

While in some publications, the terms "Continuous Delivery" and "Continuous Deployment" are used as synonyms, in his book [27], Henry van Merode points out by giving a separate definition for Continuous Deployment as follows. **Continuous deployment** operates autonomously, bypassing manual dual-control stages. When a developer pushes code to the main branch, the pipeline executes all steps, included for production deployment, without manual intervention. Thus, this pipeline mirrors the Generic CI/CD Pipeline, excluding the dual-control phase.[27]

In conclusion, integrating container orchestration platforms with CI/CD pipelines offers radical improvements for modern software delivery. Such a combination of technologies enhances deployment reliability, ensuring consistent environments from development to production and accelerating the release cycles by automating container deployment and scaling processes. Moreover, it enhances resource utilization, optimizes operational costs, and ensures rapid recovery in case of failures. Organizations can achieve unparalleled agility and efficiency in their software delivery processes by fusing the reproducibility and scalability of containers with the automation and speed of CI/CD.

2.3.1. Kubernetes

Kubernetes is an open-source system for orchestrating containers, which oversees containerized applications on multiple hosts, handling their deployment, monitoring, and scaling. Initially developed by Google, it was given to the Cloud Native Computing Foundation (CNCF) in March 2016. [28]

Kubernetes offers a framework for operating distributed systems robustly by handling the scaling and failover of your software, delivering deployment strategies, and offering additional features such as: [29]

- **Service Discovery and Load Balancing**: Containers can be exposed via Kubernetes either with their Domain Name System (DNS) names or specific IP addresses. In the case of a container with high traffic, Kubernetes ensures even distribution of network traffic for stable deployments.
- **Storage Orchestration**: Kubernetes offers the flexibility to automatically integrate a preferred storage system, encompassing local storage options and various public cloud providers.
- **Automated Updates and Rollbacks**: Kubernetes allows you to define an optimal state for your container deployments. It then progressively modifies the current state to align with the desired one.
- **Optimized Resource Utilization**: By supplying Kubernetes with a node cluster, it manages containerized tasks efficiently. You specify each container's CPU and memory requirements, and Kubernetes efficiently allocates containers to maximize resource utilization.
- **Self-healing**: If containers fail, Kubernetes restarts them. It also replaces or discontinues non-responsive containers, ensuring they're only accessible to users once they're fully operational.
- **Secret and Configuration Handling**: Kubernetes offers a secure space for storing crucial data like passwords, Open Authorization (OAuth) tokens, and Secure Shell (SSH) credentials. This permits the deployment and modification of secrets and configurations without the need to reconfigure container images or unveil confidential stack details.

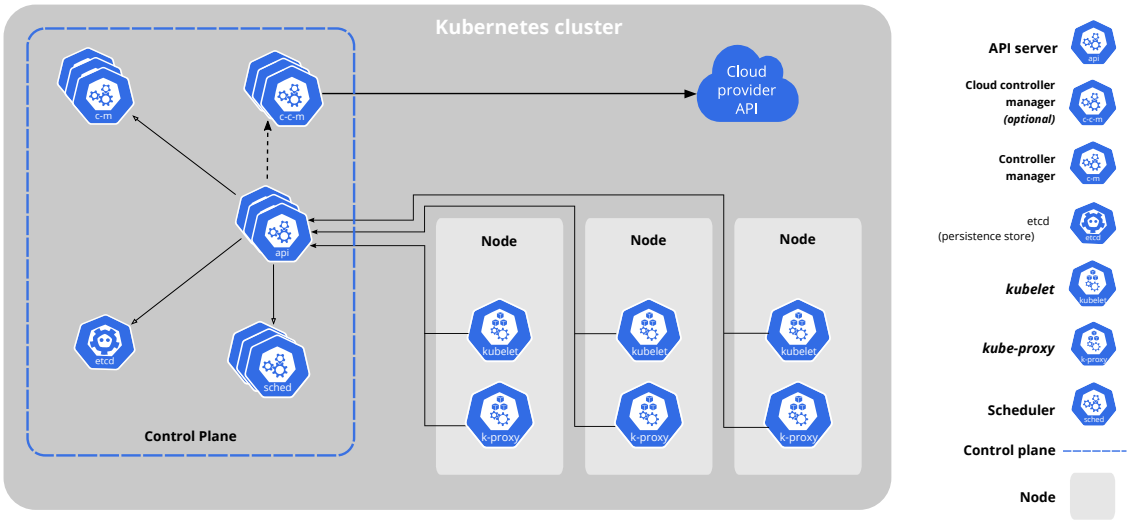

Figure 2.5.: Structure of a Kubernetes cluster, providing an in-depth look at the components within each node type that comprises the cluster [29]

Kubernetes is a sophisticated platform designed to automate the deployment, scale, and manage containerized applications across a distributed system, often called a cluster. Central to this cluster is its control plane - node architecture. While the control plane oversees and maintains the cluster's intended state, the nodes carry out the execution of containers and communicate regularly with the control plane. Each node is equipped with a crucial process known as Kubelet, which is responsible for executing containers on its designated node through Docker or its Containerization alternatives, performing routine health assessments, and updating the control plane about the containers' and node's conditions. [30]

At the heart of Kubernetes are pods — the most basic deployable units the system manages. A pod encapsulates one or more containers, granting them shared storage, network resources, and a unified IP and port environment. This setting is favorable for microservices, which are typically containerized and deployed in Kubernetes clusters as pods. Controllers are the guardians of these pods, ensuring their proper deployment and persistent management. These controllers use a pod template and consistently uphold the specified number of pod replicas. The system's resilience is evident when a pod's operation is compromised, so the overseeing controller quickly deploys a new pod to ensure seamless operation. [30]

Kubernetes introduces a "service" abstraction, identifying a logical group of pods and determining their access policy. Instead of utilizing IP addresses directly, services assemble pods based on their labels, hiding the pods' dynamic IP addresses from end-users. A service uses an inter-cluster virtual IP, routing traffic to its associated endpoints through random allocation or round-robin methods. [30]

As depicted from Figure 2.5, one of the key processes running on every node is Kube-Proxy, which monitors the control plane for changes in services and their corresponding endpoints. Kubernetes services vary in their typology, encompassing NodePort, Load Balancer, and Ingress. NodePort services, constructed over Cluster IP services, make a service accessible through the same port across all nodes. Meanwhile, Load Balancer services are only externally visible when operating in a public cloud environment. Complementing these, we have ingress, another mechanism for external access to cluster services. Ingress functions through a set of rules guiding inbound connections towards specific cluster services. However, the effectiveness of ingress is contingent upon the presence of an ingress controller, which is external to Kubernetes. This necessitates the deployment of available ingress controllers, such as Nginx or HAProxy, or the creation of custom implementations. [30]

In the realm of container orchestration platforms, Kubernetes stands out as a revolutionary tool, offering unparalleled flexibility for the deployment and management of cloud-native applications while streamlining numerous manual processes associated with handling these types of applications. One of its most remarkable features is its "cloud agnostic" nature. Kubernetes is not only fault-tolerant and scalable but also possesses the unique ability to integrate with diverse infrastructures seamlessly, be it public clouds like Google Cloud or AWS, private or hybrid clouds, data centers, on-premises setups, or even intricate combinations of these environments. This cloud-agnostic capability ensures that users are not limited to a specific cloud provider, giving them greater freedom and flexibility in choosing and migrating between platforms. Its capacity for automatic placement and replication of containers across a vast array of hosts underscores its growing reputation as the predominant choice for managing microservices infrastructure in the cloud. [28]

2.4. DevOps

The modern IT industry highly emphasizes delivering quickly to the client. This urgency is evident in the growing adoption of agile and lean methods that integrate and combine the software development process with IT operations. This movement, which focuses on bridging the gap in terms of time, effort, and even organizational structure between software development and operations teams, is known as DevOps. [31]

In his paper [32], Yarlagadda gives a complete definition of DevOps as follows: *"DevOps entails a set of integrated activities or practices employed in automation and interlink software development processes with IT developers with the aim of building, testing, and releasing deliverables quickly and reliably. DevOps is an integrated term that refers to development and operations and culturally represents an interconnection between developers and operators, whose functionalities were initially based in silos."*

At its core, DevOps aims to bridge the longstanding gap that existed between software development (Dev) and operations (Ops) teams. Historically, these teams operated in "silos," leading to technical, socio-technical, and organizational barriers that hindered smooth communication, collaboration, and problem-solving. DevOps introduces practices and tools that methodically address these barriers, striving for a seamless and robust organization. A primary aspect of this approach is merging Dev and Ops teams, allowing them to address operational issues such as infrastructure design and cost-optimization collaboratively. Furthermore, embracing DevOps tools ensures a unified knowledge base for both teams, eliminating potential misunderstandings and ensuring workflow coherence. [31]

DevOps practices involve a continuous, iterative lifecycle emphasizing regular collaboration and refinement through all project stages: planning, coding, building, testing, releasing, deploying, operating, and monitoring. Although the lifecycle phases, visually illustrated in Figure 2.6, flow sequentially, consistent team communication is crucial across all to ensure speed, coherence, and quality in software development. Equipping teams with adaptable tools and guidelines allows for fast, high-quality software creation and issue resolution through the iterative process. [32]

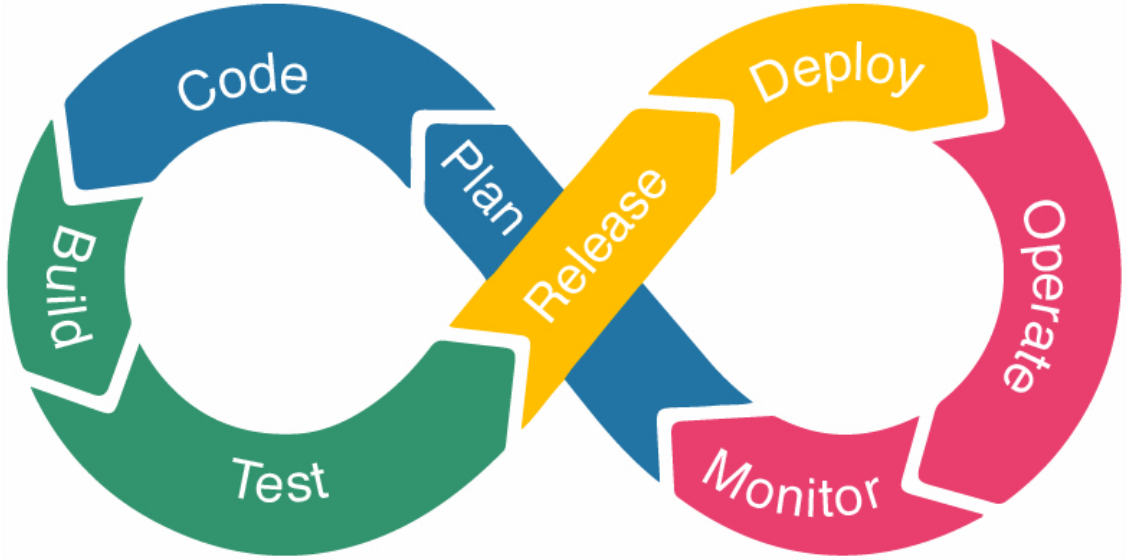

Figure 2.6.: Stages of DevOps development cycle [32]

Adopting DevOps practices profoundly impacts developers and the entire software development process, touching each phase of the life cycle in distinct ways. One of the first responsibilities for developers in a DevOps environment is to validate a system's origins, ensuring it has undergone the necessary quality checks and approvals. This initial step is vital in how smoothly the rest of the development process evolves later.

At the foundation of DevOps, we find the practice of continuous deployment, which allows developers to introduce code into the production environment without the need for cross-team coordination. This accelerates the development cycle and has substantial implications for design considerations and overall architectural choices. [33]

DevOps has revolutionized the industry landscape, with many businesses rapidly adopting its methodologies and tools to gain its numerous benefits. Organizations now experience a faster time-to-market, empowered by their capability to tackle real-time challenges. There is also a notable growth in their return on investment and an upswing in customer satisfaction due to the punctual roll-out of products and services. The operational facet, too, has witnessed a spike in efficiency, a boon attributed majorly to automation. Furthermore, the reinforced teamwork between developers and operations professionals has simplified problem-solving with coding approaches that accelerate error detection and resolution. [32]

Furthermore, adopting DevOps practices allows development teams to anticipate changes more effectively, reducing miscommunication or process misalignment risks. Through consistent communication and regular stages of integration, deployment, and testing, the efficiency and quality of products are improved while enabling quick error detection and resolution.

However, adopting DevOps practices, while promising enhanced collaboration and more streamlined operations, comes with its own set of challenges. A primary impediment to successfully implementing DevOps is the lack of effective communication. There's a noted lack of useful metrics that the operations teams monitor and share with developers. While operations focus on server uptime, developers prioritize release frequency. This divergence in priorities and over-reliance on electronic communication methods can lead to delayed responses to issues, emphasizing the irreplaceable value of in-person discussions.

Adopting DevOps introduces cultural challenges necessitating a shift in organizational mindsets. As roles merge and responsibilities change, individuals may need to step out of their comfort zones. Developers might resist operational roles, like being on-call, while operations teams may feel threatened by developers intruding on their domain. Long-term professionals may find this especially challenging. It's vital for all to embrace a cultural shift towards DevOps ideals. However, it's essential to understand that DevOps isn't always suitable. Industry constraints, complex scenarios, and legal restrictions can impede its adoption. Recognizing these challenges is key for organizations contemplating DevOps.

To conclude, DevOps is more than a set of tools or practices; it is a cultural shift aimed at breaking down silos between development and operations. By promoting

collaboration, automation, and integration, DevOps helps organizations deliver software faster and reliably, providing them with a competitive edge.

2.5. GitOps

The term "GitOps" was initially introduced by Weaveworks during KubeCon - CloudNativeCon 2019 during the panel discussion on “GitOps and Best Practices for Cloud Native CICD” [34]. This methodology revolutionizes the automation of the DevOps pipeline by leveraging a version control system for both infrastructure management and application source code. GitOps aims to streamline continuous delivery for container-based apps by utilizing tools that developers are already accustomed to, such as Git and Continuous Deployment instruments. This approach results in a more agile, developer-focused experience in overseeing the infrastructure.

GitOps operates on the principle of treating Git as the sole source of truth, explicitly defining the target infrastructure for the production environment. This approach automates aligning the production environment with this intended state thus, for deploying a new application or modifying an existing one, the developer simply updates the repository, and the automation takes care of the rest. [35]

With an active community around GitOps, the GitOps Working Group was born, and together with it also the official GitOps principle, currently at version 1.0.0 [36]:

- **Declarative**: In GitOps, the desired state of a system should be described in a declarative manner.
- **Versioned and Immutable**: The expressed desired state is preserved in an immutable format with version control, ensuring a comprehensive record of all versions.
- **Automatic Pull**: Automated software tools automatically fetch the declared desired states from their source.
- **Continuous Reconciliation**: Software agents consistently monitor the current system status and strive to align it with the desired state.

GitOps offers a streamlined set of best practices that enhance the user deployment experience through fully automated processes linked to developing, deploying, managing, and monitoring containerized apps and clusters. Embracing GitOps in development cycles introduces essential Cloud-Native advantages such as agility, reliability, and speed. This technique offers a holistic developer experience in app management, fusing end-to-end CI/CD pipelines and git workflows to both operations and development, yielding a more dependable deployment approach compared to traditional CI/CD pipelines. Thus, the benefits of GitOps include [35]:

- **Enhanced Deployment Speed**: With GitOps, all tools essential for application development are version-controlled, eliminating constant tool-switching.
- **Documented History of Environments**: Changes to the application environment occur via updates in the Git repository, capturing a comprehensive log of changes, the people involved, and the reasons.
- **Simplified Rollbacks to Previous States**: All modifications reside in the Git repository, making it straightforward to revert an environment to any previous operational state.
- **Secure Deployments**: Deployment management occurs entirely internally. The only necessities are access to the Git repository and the image registry, negating the need for developer access to the environment.
- **Tool-Agnostic Design**: GitOps is not bound to specific tools so that users can combine tools optimally. For instance, while a renowned Git server manages the Git aspects, tools like Flux or Argo CD synchronize the repository with the cluster.
- **Easy Environmental Comparisons**: The declarative configurations in Git repositories simplify the tracking of disparities across different environments in a development-testing-production chain.
- **Inherent Backups**: As environment states are chronicled in git repositories, any issues on platforms like Kubernetes will not result in data loss, positioning the git repository as a built-in backup.

Within the GitOps paradigm, two primary deployment methodologies exist Push-based and Pull-based deployments. These paradigms are characterized by unique features and methods of operation, making them feasible solutions for various tools and platforms in the industry.

The **Push-based deployment** methodology is commonly adopted by renowned CI/CD tools such as Jenkins, CircleCI, and TravisCI. In this approach, the application's source code coexists in the repository with the Yet Another Markup Language (YAML) scripts essential for deployment. When a change is committed to the Git repository, a CI/CD pipeline detects the change, processes it, and actively pushes the changes to the target environment. [35]

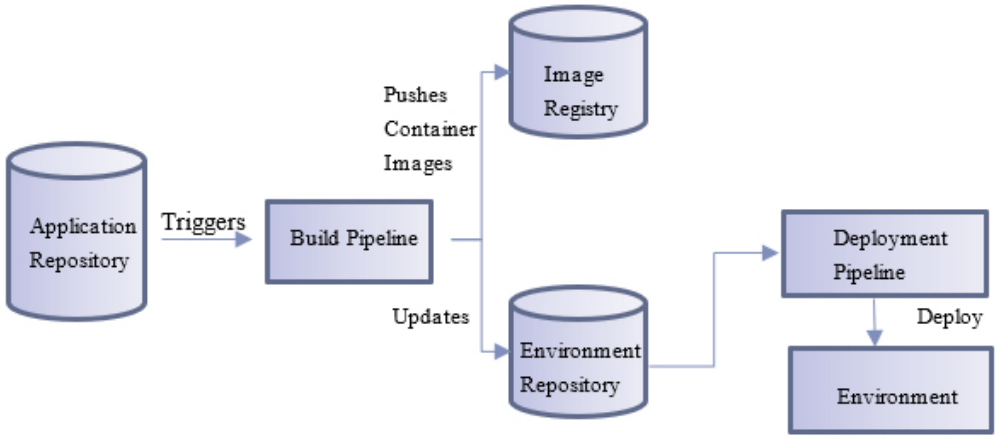

Figure 2.7.: Logical flow for the GitOps implementation of the Push-based deployments methodology [35]

On the other hand, **Pull-based deployments**, while sharing similarities to their Push-based counterparts, exhibit differences in the mechanism of the deployment pipeline. In traditional CI/CD settings, external events, such as code pushed to the repository, activate the pipeline. In pull-based deployment, agents in the target environment (like Kubernetes clusters) continuously monitor the Git repository. When they detect a change, they pull the changes and apply them to the environment. This approach retains the update-on-change mechanism found in Push-based deployments but with the added benefit of resolving their limitation—namely, that updates occur only when the environment repository itself is modified. [35].

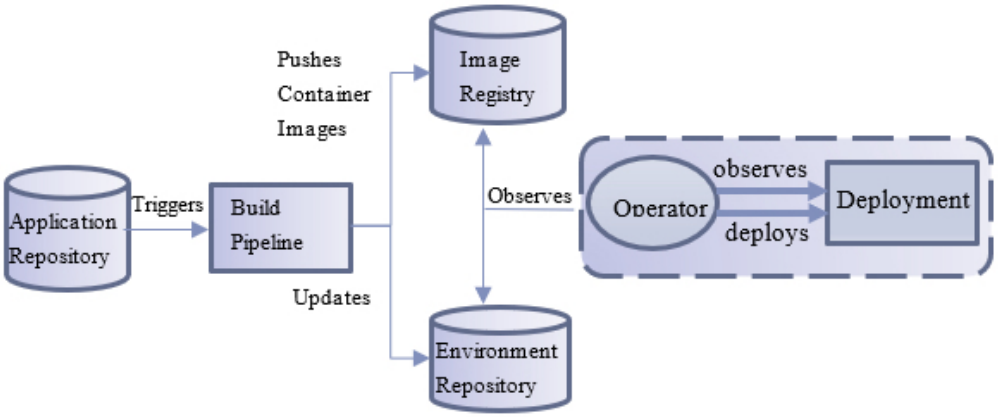

Figure 2.8.: Logical flow for the GitOps implementation of the Pull-based deployments methodology [35]

Managing a single repository and environment is not feasible for most applications, especially those using microservices architecture. GitOps addresses this limitation by establishing multiple build pipelines to update respective environment repositories. To handle various environments, GitOps configures distinct branches within these repositories, and the deployment pipelines can then respond to changes in their specific branch by deploying the changes to the production environment. [35]

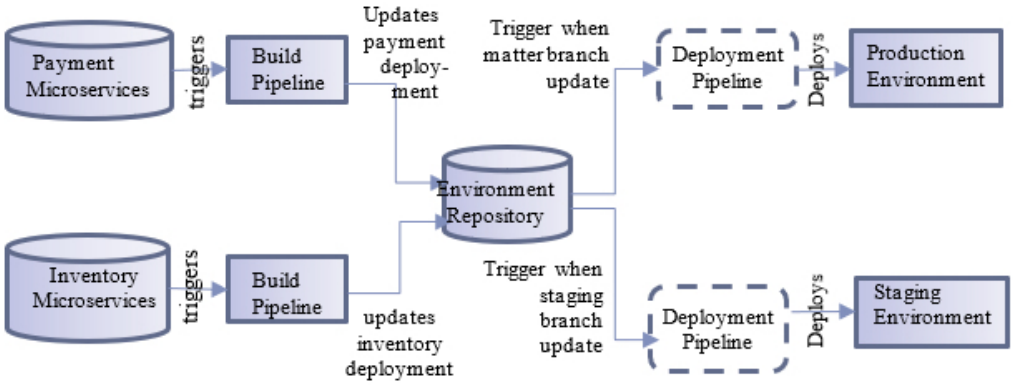

Figure 2.9.: Logical flow for the implementation of GitOps applied to an example microservices project [35]

To conclude, GitOps, an advanced form of DevOps, offers increased agility and efficiency in cloud-native development by using Git as the sole source of truth, streamlining CI/CD in Kubernetes. It emphasizes a developer-centric approach, linking code repositories to live environments and using version control for both code and environment. This standardizes workflows, improves productivity, and adds features like rollback. However, its pull-based deployment model limits tool choices and requires vigilance against issues like broken YAML manifests that could disrupt development.

2.5.1. Argo CD

Argo CD is an open-source, declarative GitOps continuous delivery tool for Kubernetes. It facilitates the automation of application deployments using Git repositories as the source of truth for the declarative infrastructure and application definitions. Developed under the CNCF, Argo CD is widely adopted in Kubernetes environments for its alignment with GitOps principles.

Argo CD employs the GitOps approach, treating Git repositories as the only source of truth for defining the intended state of applications. It supports multiple formats for Kubernetes manifests, including Kustomize applications, Helm charts, jsonnet files, plain YAML or JavaScript Object Notation (JSON) directories, or any custom tool set as a config management plugin. The tool automates the deployment of these predefined application states to target environments, offering flexibility in how it tracks changes in

the Git repository. It can be set to follow updates to specific branch tags or even lock it to a particular manifest version based on a Git commit, with various tracking strategies available for added customization. [37]

Argo CD ensures that the state of applications on Kubernetes matches the intended configurations specified in a Git repository, providing tools and processes to maintain and verify this consistency. Thus, this tool is built upon several core principles: [37]

1. **Application**: Defined as a group of Kubernetes resources based on a manifest, it serves as a Custom Resource Definition (CRD). The **Application source type** determines which tool, such as Kustomize, is utilized to construct the application.

2. **Target state** represents the desired state of an application, depicted by files in a Git repository, while the **Live state** reveals the actual current status of the application, indicating which pods or resources are actively deployed. The coherence between these states is determined by the **Sync status**, which ascertains if the deployed application aligns with the specification from the Git repository.

3. **Sync** describes the act of aligning an application with its target state, typically by implementing changes to a Kubernetes cluster. The outcome of this alignment is expressed by the **Sync operation status**. To stay updated, **Refresh** compares the latest code in Git against the live state, discerning any discrepancies.

4. **Health** indicates the application's operational efficiency, reflecting if it's running as intended and able to serve requests. For advanced customization, Argo CD supports **Configuration management plugins**, allowing the integration of unique tools.

Argo CD, a preferred GitOps tool over alternatives like Flux CD or Jenkins, offers a robust suite of features for effective application deployment. It enables automatic deployment to specified environments, supports multiple config management tools (e.g., Kustomize, Helm), and manages deployments across various clusters. Argo CD provides Single Sign-On (SSO) integration with OpenID Connect (OIDC) or OAuth2, along with multi-tenancy and robust Role-based Access Control (RBAC) policies. Users can easily roll back configurations, monitor application resource health, and identify configuration drifts, utilizing a real-time web User Interface (UI) and CLI, alongside webhook integrations with platforms like GitHub. The tool supports advanced deployment strategies using PreSync, Sync, and PostSync hooks and ensures traceability with audit trails and Prometheus metrics. [37]

Argo CD, acting as a Kubernetes controller, continually ensures applications' live states align with the targeted states defined in the Git repository, labeling discrepancies as "OutOfSync" and offering auto-correct or manual realignment options. Changes to the target state in the Git repository can be effortlessly reflected in respective environments.

The entire deployment process, visible in Figure 2.10, spans from the developer and Git repository, through CI tools, to Argo CD, functioning with three main components:

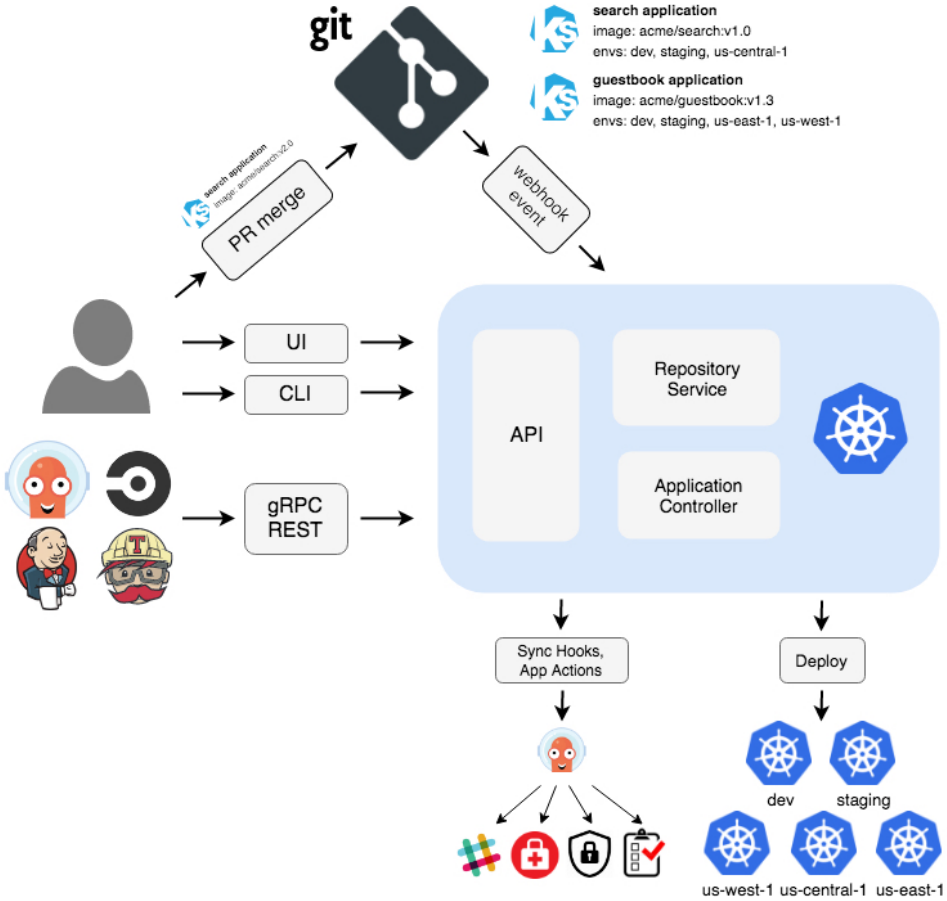

Figure 2.10.: Architectural Overview of Argo CD implemented as a Kubernetes Controller and adjacent tools [37]

1. **API Server**, functioning as a Google Remote Procedure Call (gRPC)/REST server, provides an API used by the Web UI, CLI, and CI/CD systems, handling application management, status updates, and operations like syncing and rollback. Additionally, it manages repository and cluster credentials (as Kubernetes secrets), oversees authentication, delegates authority, enforces RBAC, and receives Git webhook events.

2. **Repository Server** acts as an internal service that maintains a localized cache of the Git repository containing application manifests. Its main function is to generate and deliver the Kubernetes manifests, actions that require details such as the repository URL, revision (like commit, tag, or branch), application path, and template-specific settings, including parameters and helm values.yaml.

3. **Application Controller** functions as a Kubernetes controller and persistently observes applications in action. It evaluates if the real-time state of an application matches its intended state as stated in the repository. If discrepancies arise, signifying an OutOfSync state, the controller identifies them and can optionally make rectifications. Additionally, it triggers any user-specified hooks during lifecycle events, including PreSync, Sync, and PostSync.

To conclude, Argo CD represents a significant shift in how deployments are handled in Kubernetes environments. By aligning with GitOps principles, it ensures consistency, traceability, and automation in the deployment process, ensuring rapid and safe application rollouts. As organizations continue to adopt cloud-native solutions, tools like Argo CD will become foundational to their continuous delivery pipelines.

3. Design

3.1. Design goals

Before diving into the implementation, gaining a comprehensive view of the design objectives was essential. The primary aim of the GHGA Munich data hub, situated at the Leibniz Supercomputing Center (LRZ), is to build an infrastructure prototype. This is envisioned to be secure, resilient, and immensely scalable to meet the growing demands of Bioinformatics research.

At the base of this prototype lies the idea of providing IaC. IaC is an approach where the infrastructure of an IT environment is codified. This means that these resources are defined and managed through code instead of manually setting up servers, networks, and other infrastructure components. Adopting IaC ensures that infrastructure is repeatable, scalable, and can be version-controlled like any other software product.

In our case, the chosen platform to implement IaC is an OpenStack cluster. Being an open-source cloud computing platform that allows the creation and management of large computing, storage, and networking resources, OpenStack provides the flexibility to design custom infrastructure tailored to the specific needs of projects, making it an ideal choice for the data hub.

On top of the OpenStack infrastructure, a Kubernetes cluster is expected to be deployed. With Kubernetes, microservices can be efficiently managed, scaled up or down as required, and easily updated or replaced without disrupting the entire system.

As the research project of GHGA Munich data hub is still in its early stages, the primary focus has been ensuring the microservice's deployment is fully functional. However, it is acknowledged that security is of high importance, especially for a data hub that stores and processes sensitive information like Genome-Phenome data. The plan is to thoroughly address security enhancements and fortifications in the advanced stages of the research. This might involve implementing advanced security protocols, encryption techniques and possibly even integrating third-party security solutions to ensure data protection and integrity.

3.2. Requirements Analysis

When it comes to scaling infrastructure, planning and resource allocation, represent a step in the process. In our research proposal, we envisioned a robust setup that would meet the standards of a High-Availability (HA) production system. However, due to constraints, a scaled-down version was implemented. In the following, our choices in terms of technologies used and resource management are presented and reasoned about:

- **Choosing K3s over K8s**: Kubernetes, or K8s, is the industry standard for orchestrating large-scale containerized applications. However, its vast set of features and comprehensive architecture demands a more resource-intensive environment. Given our resource constraints, K3s emerged as a viable alternative. K3s is a lightweight version of Kubernetes, leaving out many non-essential features and configurations. This simpler approach allowed us to run Kubernetes with lower overhead, making it a perfect fit for our limited resource scenario.
- **Cluster Configuration**: A typical HA Kubernetes cluster would have at least three master nodes to ensure failover capabilities and a number of agent nodes proportional to the workload [38]. However, given our constraints, we opted for a smaller setup containing just one master and one agent. This configuration is not ideal for production as it doesn't offer the redundancy and resilience of a true HA setup. But, given our limited infrastructure, it allowed us to run containerized applications while understanding the fundamental mechanics of Kubernetes.
- **Hardware Limitations**: Our available infrastructure was significantly limited. Our options were limited with only three servers, each having 16GB Random Access Memory (RAM) on which OpenStack was running, and one 8GB RAM server dedicated to Ceph. It's important to note that an OpenStack installation demands 8GB RAM. This left us just an additional 8GB for the Kubernetes cluster to be deployed and the microservices applications running on top of it.

Thus, the current implementation is designed to include a single Controller node, complemented by a variable number of Compute Nodes. In our testing, we evaluated configurations with one and two Compute Nodes. This design is inherently scalable, allowing for the addition of more Compute Nodes to the OpenStack cluster as needed. Similarly, for the Kubernetes layer, we anticipate having one Master Node and currently just one Worker Node. However, this can be easily scaled up when additional hardware resources become available. For a comprehensive view of the system's structure, Figure 3.1 illustrates the current architectural design.

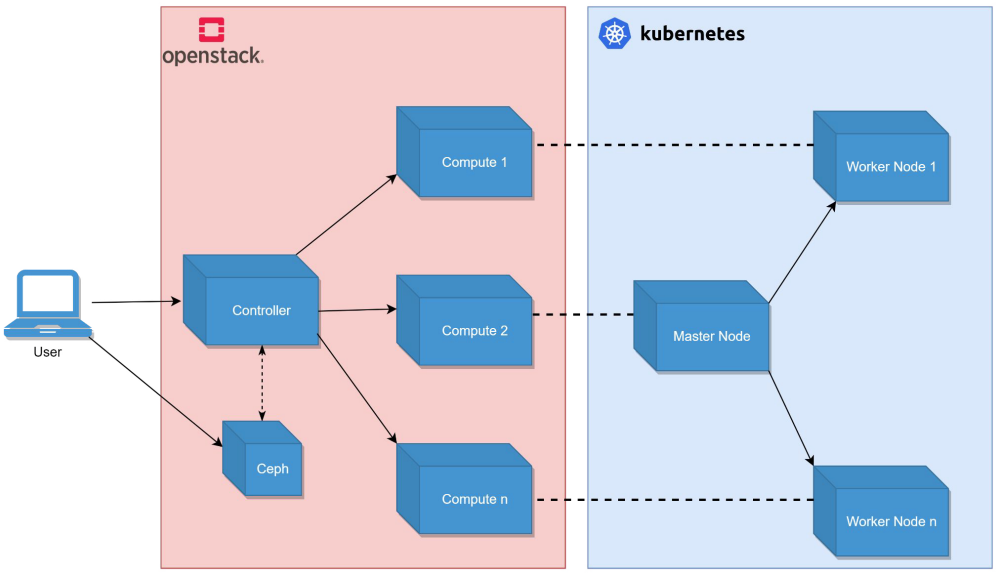

Figure 3.1.: Proposed system architecture and the minimum hardware requirements (own illustration)

3.3. Possible Solutions

While developing a robust and automated cloud infrastructure for microservice deployment, we explored various market solutions that significantly influenced the current design approach. This section delves into potential alternative solutions and provides a comprehensive analysis of the currently implemented infrastructure in this project.

The current IT market offers several platforms that show a range of functionalities designed to simplify the deployment process of scalable microservices within a cloud environment. One notable platform is Railway [39], which allows users to customize application deployment through an intuitive user interface. Although it is a paid product, Railway represents a class of solutions that could significantly elevate the deployment process by making it more user-friendly and customizable. Exploring a model similar to the one of Railway could have been a potential solution if more time and resources were available.

In addition to Railway, other platforms like Heroku [40], Vercel [41], and Netlify [42] present viable alternatives. Heroku is a cloud platform that enables developers to build, run, and operate applications entirely in the cloud. It abstracts and hides infrastructure complexities, allowing developers to focus solely on application development. Vercel is another platform optimized for front-end projects, providing features such as continuous deployment from git, serverless functions, and much more. On the other hand, Netlify is a robust platform offering an automation-filled developer experience, allowing

continuous deployment from git across a global application delivery network. All these platforms are paid products, and analyzing them has enabled us to plan our prototype's main features while also considering the stipulated timeline and human and hardware resources.

In the current approach, the absence of a web application capable of conducting basic unit testing and obtaining precise deployment information for CI pipeline integration places these tasks squarely on the shoulders of our users. This is largely because these tasks are highly specific to each project, and automating them would have made the process overly complex. To address this, our current solution uses Argo CD for the (CD) pipeline, which is based on a "Pull-Based Deployment" paradigm. Thus, Argo CD automatically pulls and deploys the changes, simplifying the deployment process and alleviating the need for manual intervention.

A significant issue resolved through the implementation of Argo CD is the matter of ACL. Argo CD offers a variety of authentication methodologies, either through "local" users (created directly through Argo CD) or via SSO. The prototype has utilized both "local" users and SSO through the Bundled Dex OIDC provider, facilitating the mapping of GitHub organizations and teams to OIDC group claims. The testing phase involved the creation of a GitHub organization with varied roles to replicate and assess the application in a real-world scenario, aligning efficiently with the project's use case.

The pathway towards a more simplified deployment process may involve the creation of an integrated CI/CD pipeline. A dedicated web application could be developed to perform preliminary unit testing and collect deployment-specific information, fostering a more seamless integration between the CI and CD pipelines. Furthermore, a user-centric approach could be considered when designing an intuitive interface similar to what Railway offers. Such an interface would allow users to effortlessly customize their application deployment, potentially accelerating the deployment process while upholding the standards of reliability and scalability.

The similarities between market-based solutions and the current implementation accentuate the potential for achieving an enhanced deployment process. While the current solution meets this project's objectives, more exploration could be done in this direction to further refine the microservices deployment infrastructure under more favorable conditions of time and resources.

3.4. System Components and Workflow

In order to design a robust and scalable system, it is critical to choose the right set of components that work seamlessly together to deliver the desired functionalities. The proposed system incorporates a combination of well-established technologies to build a resilient and scalable infrastructure. The system's primary components include OpenStack Infrastructure with integrated Ceph Storage, a Kubernetes Cluster set up with Terraform and Terragrunt, and Argo CD implemented on top of the Kubernetes cluster as a GitOps solution. This section demonstrates the role of each component and defines the workflows that sustain the interactions among these components, as illustrated in Figure 3.2.

- **OpenStack Infrastructure:** OpenStack is employed as the core infrastructure platform that supports the entire system. Using Ansible via Kolla-Ansible enables a simplified setup of the OpenStack infrastructure through several automated scripts. The resulting infrastructure provides a range of services essential for building and managing the cloud environment where the Kubernetes cluster will be orchestrated. Its modular architecture allows for seamless integration with the other components of the system, like Ceph Storage, providing a cohesive and manageable infrastructure.
- **Ceph Storage:** Ceph Storage delivers a reliable and scalable storage solution, smoothly integrated with OpenStack via Ansible and bash scripts. Once provisioned, it ensures persistent data storage for the Kubernetes cluster, thus facilitating stateful applications and services within the system. Ceph's capability to self-heal and self-manage significantly reduces administrative overhead while ensuring data durability and availability.
- **Kubernetes Cluster:** The Kubernetes Cluster is a fundamental component orchestrating containerized applications. The setup and management of the Kubernetes Cluster are automated using Terraform and Terragrunt. Terraform facilitates the provisioning of the infrastructure resources required, while Terragrunt wraps around Terraform to provide extra layers of configuration and manageability. This automation ensures a repeatable and reliable deployment process, which is crucial for maintaining the desired system state and facilitating scaling.
- **Argo CD:** Argo CD is used as a GitOps technology, enabling a declarative and version-controlled method for deploying and managing applications within the Kubernetes Cluster. Argo CD ensures that the real-time state of these applications matches their desired state defined in Git. Any deviation between the two triggers Argo CD to bring the application back to its intended state. Moreover, it provides the required ACL capabilities, enabling users' control over various deployed projects with specific privileagesß.

The interaction between these components is shown in Figure 3.2. The workflow starts with the setup and configuration of the OpenStack Infrastructure via Ansible. Following

this, the Ceph Storage is integrated, and the Kubernetes Cluster is set up using Terraform and Terragrunt. Post-setup, Argo CD is installed on top of the Kubernetes cluster to facilitate GitOps workflows. Application deployments are then performed through Argo CD, conforming to GitOps principles. These ensure synchronization between the desired system state defined in Git and the actual system state in the Kubernetes Cluster. This setup promotes a structured and automated workflow, reducing manual intervention and enhancing system resilience and scalability.

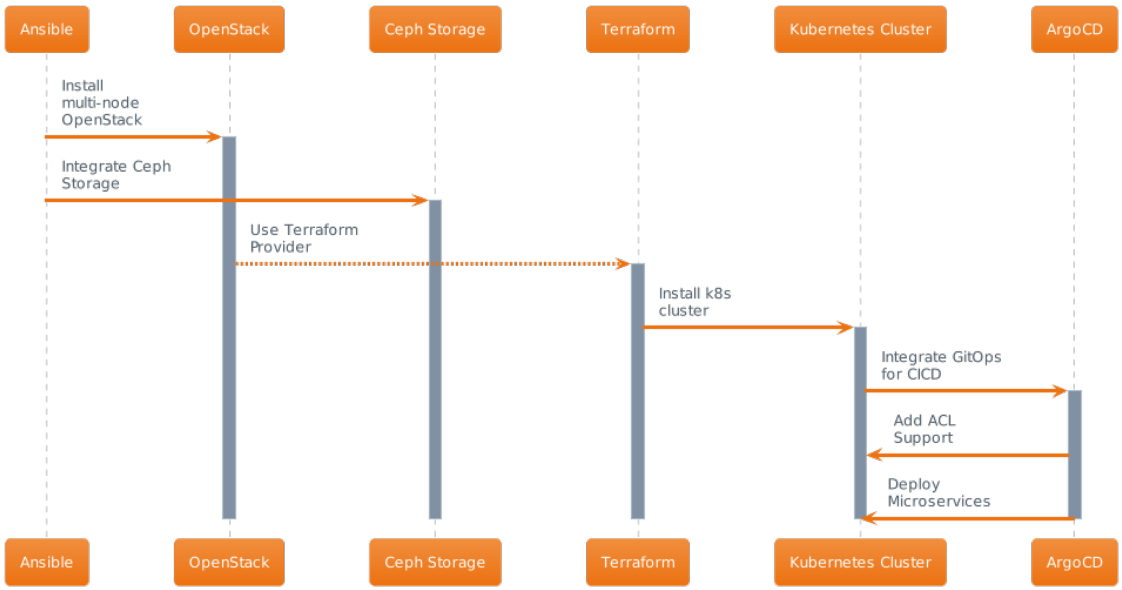

Figure 3.2.: Proposed system overview and interaction between the main system components (own illustration)

4. Implementation

4.1. Implementation Stages

This section aims to provide a comprehensive look into the series of methodologies, technologies, and procedures utilized to accomplish the overarching goal: to develop and automate a reliable cloud infrastructure for the scalable deployment of microservices. Every phase of the implementation serves a strategic purpose and incorporates a set of specialized tools chosen for their proficiency in delivering specific objectives. The following subsections will unpack each stage, offering insight into their purpose in fulfilling the project's overall vision. The source code can be found here[43].

1. **Setting up an OpenStack Multi-node Cluster: Script Development and Execution**
   The project's initiation was marked by creating a suite of scripts that could automatically set up an OpenStack multi-node cluster. This foundation was crucial for the project as the stability, scalability, and reliability of all subsequent stages depended on it. These scripts were stored in a directory named multinode-kolla-ceph-install-deploy.

2. **Storage Backend Integration: Ceph Storage**
   After the multi-node OpenStack cluster was successfully established, Ceph Storage was integrated into the system. The Ceph storage backend provides persistent volumes and ensures that the stored data remains consistently available and reliable. Its architecture enables data replication and fault tolerance, making it ideal for stateful microservices that require persistent storage.

3. **Kubernetes Cluster Provisioning: Terraform Automation**
   Upon successfully establishing the OpenStack and Ceph storage, the next phase involved provisioning the Kubernetes cluster. Terraform, an open-source infrastructure-as-code software tool, was used for this task. Scripts found in the 'terraform' directory specify the configurations for the Kubernetes nodes, services, and other resources, ensuring that they are provisioned in an idempotent manner.

4. **GitOps for Continuous Deployment: Argo CD Implementation**
   Argo CD was introduced as a GitOps tool for Continuous Deployment. It was integrated directly into the Kubernetes cluster, facilitating a GitOps workflow. In practical terms, this meant that any push to the project's git repository triggered an automatic update across the Kubernetes services, leading to near-instantaneous

deployment of new features, updates, or fixes without requiring manual intervention.

5. **Deployment of Microservices: Operationalization**
   The final step involved deploying microservices onto the secured and highly available Kubernetes cluster. These microservices serve as the functional units of the entire infrastructure. Their deployment onto a cluster, which was meticulously designed for scalability and security, indicates that the system is now ready for real-world challenges.

In summary, the implementation phase encapsulated an intricate combination of steps aimed at building a secure, scalable, and reliable infrastructure for microservices. Each stage used specialized tools and procedures to achieve specific objectives, from setting up the OpenStack cluster via IaC methodologies to microservice deployment. The final result is an operational cloud infrastructure capable of scalable microservices deployment, fulfilling the project's ultimate goals.

4.2. Layer 1: Openstack

Implementing the OpenStack phase proved especially challenging due to the multiple methods available for configuring OpenStack, each presenting its unique characteristics. Initially, more straightforward solutions like DevStack [44] were examined, which offer a rapid establishment of a full-fledged OpenStack environment. However, DevStack's drawback is its suitability solely for testing or brief prototyping, making it unsuitable for environments ready for production. Given this study's objective to develop a dependable cloud infrastructure for scalable microservices deployment, our search extended to more resilient and adaptable alternatives, such as Kolla-Ansible [45], which aligns more fittingly with production-grade requirements.

Kolla-Ansible is an integral component of the OpenStack ecosystem, specifically designed to simplify the deployment, maintenance, and upgrading of OpenStack services using container technology. Leveraging the robustness of Ansible for orchestration and Kolla's container templates, Kolla-Ansible encapsulates OpenStack services within Docker containers. This encapsulation allows for consistent deployments and isolated environments, reducing conflicts and inconsistencies. In practice, a developer initiates a series of Ansible playbooks that, in turn, pull the required container images and deploy them according to the configuration specified. By combining the modularity of containers with the automation capabilities of Ansible, Kolla-Ansible presents a powerful and streamlined approach to establishing and managing OpenStack environments, making it an optimal choice for those seeking production-grade OpenStack deployments.

Kolla-Ansible offers two primary methods of deploying OpenStack, suitable for developmental experimentation and larger-scale production needs: the All-In-One and

Multinode installations.

While the All-In-One (AIO) method involves setting up all the necessary OpenStack services on a single machine, it is ideal for developers or those wanting to experiment with OpenStack without the overhead of multiple nodes. In an AIO setup using Kolla-Ansible, all OpenStack services are encapsulated within Docker containers on a single host. This provides an easy and rapid means to deploy a functional OpenStack environment, although it's typically unsuitable for production due to potential performance constraints.

Thus, we discovered that OpenStack services are distributed across multiple servers or nodes in a multinode setup. This method aligns more with production scenarios where scalability, redundancy, and performance are highly important. With Kolla-Ansible, each node is provisioned with specific OpenStack services as Docker containers. For instance, one node might host compute services while another manages storage. This distribution allows for horizontal scaling and optimizes resource utilization. Using Kolla-Ansible's automation capabilities, orchestrating such a distributed deployment becomes more streamlined and manageable.

We have meticulously documented every installation step within bash scripts throughout the development journey. To execute these, one simply needs to adhere to the guidelines in the README.md located in each directory. For the purpose of this discussion, the emphasis will primarily be on the Multinode installation, as it's the methodology integrated into the final prototype. The relevant scripts can be accessed in their dedicated directories here: All-In-One Installation and Multinode Installation.

Our multinode implementation contains the following scripts:

- **init-communication.sh**: This script establishes secure communication between the Control node and Compute nodes using SSH. By generating a SSH key pair on the Control node, the public key can be distributed to all Compute nodes, ensuring uninterrupted communication. To successfully share the public key, the public IP addresses of all Compute nodes and the one running Ceph are required. Although our current setup features only a single Control node due to resource constraints, the script's design is scalable and can be extended for production environments with multiple Control nodes. Its flexibility is further highlighted as it can adjust to any number of Compute nodes, given that it prompts the user for the necessary variables:
  - Control Node's Public Key: To be distributed across the cluster nodes.
  - Number of Compute nodes needed for communication setup
  - Compute Nodes' Public IP Addresses
  - Ceph Node's Public IP Address

- **init-control.sh**: This script serves as a comprehensive guide for setting up the Control node in preparation for a Kolla-Ansible installation, which aims to deploy OpenStack in a more straightforward and manageable way. The installation procedure is broken down into carefully planned stages to ensure a seamless experience and robust security measures.
    1. **User Creation for Administrative Access:** The first step involves creating a new user named "kolla" on the machine. This user is granted sudo privileges, and notably, the password authentication is disabled. The advantage of this approach is that it enables the script to execute commands directly, making the entire process more automated and less prone to human error.
    2. **Enhanced Developer Access for Diagnostics:** In an effort to allow for smooth debugging and diagnosis, the developer's public key is added to the "kolla" user's authorized_keys file. This procedure grants the developer secure and password-less SSH access, which is particularly useful for troubleshooting potential issues in the OpenStack infrastructure.
    3. **Installation of Prerequisite Packages:** Before proceeding with the Kolla-Ansible installation, specific software packages must be present on the system. These include git, python3-dev, libffi-dev, gcc, libssl-dev, python3-selinux, python3-setuptools, python3-venv, net-tools, gum, and iproute2. The script automates the installation of these required packages to eliminate any dependencies that could interrupt the Kolla-Ansible installation process.
    4. **Customizable Input Parameters:** Next, the script seeks the developer's input for certain variables integral to Kolla's configuration. These variables include the IP address assigned for kolla_internal_vip_address, the network interface designated for internal Kolla communication (which remains consistent across all nodes), and the specific number of compute nodes to be set up. Collecting this information allows the script to customize the installation according to the unique specifications of the environment.
    5. **Preparation of Compute Nodes:** Once the necessary inputs are obtained, the script takes the initiative to prepare all designated compute nodes for the Kolla-Ansible installation. It accomplishes this by copying the "init-compute.sh" script to each compute node, ensuring each is prepared and ready for the subsequent installation steps.

Code Snippet 4.1: Part of init-control.sh script that copies and executes the init-compute.sh script on the compute nodes

```
copy_and_execute_script() {
    local ip_addresses=("$@")
    for ip in "${ip_addresses[@]}"; do
        echo "Copying $script_path to $ip..."
        scp "$script_path" root@"$ip":/root/init-compute.sh
        echo "Executing script on $ip..."
        ssh root@"$ip" "bash /root/init-compute.sh"
        ssh root@"$ip" "echo '$public_key' >> /home/kolla/.ssh/authorized_keys"
    done
}
copy_and_execute_script "${ip_addresses_compute_nodes[@]}"
```

6. **Networking Configurations and Persistence:** The final stage of the script focuses on networking protocols. It creates a dummy network interface during the system boot and saves the related settings in "/opt/network.sh" and "/etc/systemd/system/tap-interface.service" files. After the "kolla-control.sh" script is executed under the "kolla" user, the networking rules are appended back to "/opt/network.sh." This ensures the OpenStack infrastructure remains intact and operational, even during a system reboot or failure.

Code Snippet 4.2: Content of /opt/network.sh

```
sudo bash -c 'cat << EOF > /opt/network.sh
#!/bin/bash
set -x -o errexit -o nounset -o pipefail
sudo ip tuntap add mode tap br_ex_port
sudo ip link set dev br_ex_port up
export EXT_NET_CIDR='10.0.2.0/24'
export EXT_NET_RANGE='start=10.0.2.150,end=10.0.2.199'
export EXT_NET_GATEWAY='10.0.2.1'
EOF'
```

Code Snippet 4.3: Content of /etc/systemd/system/tap-interface.service and starting command for the tap-interface

```
sudo bash -c 'cat << EOF > /etc/systemd/system/tap-interface.service
[Unit]
Description=Create persistent tap interface
[Service]
ExecStart=/opt/network.sh
ExecStop=/sbin/ip link set dev br_ex_port down
RemainAfterExit=yes
Type=oneshot
[Install]
WantedBy=multi-user.target
EOF'
systemctl daemon-reload
systemctl start tap-interface
```

By following these structured and detailed steps, the script ensures that the Control node is optimally configured and ready for a successful Kolla-Ansible installation.

- **init-compute.sh**: The provided script is designed to run on all Compute nodes in preparation for the Kolla-Ansible installation. While it mirrors the first three steps found in the init-control.sh script, there are minor variations, particularly in step 3, where gum and iproute2 packages are excluded from installation for the Compute Nodes. Subsequently, the script proceeds to step 6 of init-control.sh, implementing the same network configurations as detailed in Code Snippet 4.2 and Code Snippet 4.3 on the Compute nodes. These network setups facilitate seamless communication across all nodes within the OpenStack cluster, ensuring that the infrastructure remains robust and functional, even during system disruptions or reboots.

- **init-ceph.sh**: This script is designed to install Ceph [46] and configure it for integration with OpenStack. Given the constraints of our resources and to maintain our prototype's simplicity, our OpenStack cluster will contain only a single node dedicated exclusively to running Ceph. Consequently, we utilize cephadm [47], a utility for managing Ceph clusters, to deploy Ceph in a single-host mode, as indicated by the flag "–single-host-defaults".

Code Snippet 4.4: Content of init-ceph.sh script

```
#!/bin/bash
set -x -o errexit -o nounset -o pipefail

apt install -y cephadm
CEPH_NODE_IP=$1

cephadm bootstrap --single-host-defaults --mon-ip $CEPH_NODE_IP
cephadm shell ceph orch apply osd --all-available-devices
cephadm shell ceph osd pool create volumes 64 64 replicated
cephadm shell ceph osd pool application enable volumes rbd
```

The code snippet above Code Snippet 4.4 contains the complete script dedicated to the configuration of the Ceph node. The command "**cephadm shell**" initiates a bash shell within a container equipped with all necessary Ceph packages. This environment facilitates the execution of Ceph-related commands. For example, it allocates any available and unused storage devices to Ceph, establishes a new storage pool named "volumes," and then tags this pool with "rbd," signifying its intended use for storing Rados Block Devices (RBD)s.

In summary, the script accomplishes four key tasks: it installs cephadm, initializes a Ceph cluster on a single node, allocates all available storage devices to this cluster, and configures a replicated storage pool specifically for RBD utilization.

- **kolla-control.sh**: This script orchestrates the installation and configuration of the OpenStack cluster by using the Kolla-Ansible deployment approach. The process is broken down into several essential steps:

  1. **Virtual Environment Preparation for Kolla-Ansible:** This initial step involves creating the virtual environment, updating to the most recent pip version, and installing the following packages: 'ansible>=4,<6', python3-docker, and wheel. Subsequently, Ansible is configured to pave the way for the Kolla-Ansible installation. The specifics of this configuration can be referred to in Code Snippet 4.5.

     Code Snippet 4.5: The configuration of Ansible

     ```
     sudo mkdir -p /etc/ansible
     sudo bash -c 'cat << EOF > /etc/ansible/ansible.cfg
     [defaults]
     host_key_checking=False
     pipelining=True
     forks=100
     EOF'
     ```

2. **Kolla-Ansible Installation:** For our prototype, we have selected the Zed release, one of the latest stable versions of Kolla-Ansible.
3. **OpenStack Deployment Preparation:** This crucial step lays down the foundational configurations for the OpenStack infrastructure through two primary files: the "multinode" inventory file and "globals.yaml". The process is efficiently automated, sourcing variables directly from the init-control.sh script. This automation is depicted in Code Snippet A.2. The "globals.yaml" which also be examined in Code Snippet A.3 outlines the OpenStack services integrated into our infrastructure, including Horizon, Heat, Nova, Cinder, Neutron, Keystone, and Glance.
4. **Initiation of the "init-ceph.sh" Script:** This entails copying the "init-ceph.sh" script to the designated Ceph node and initiating its execution. Once completed, configurations from the Ceph node are transferred to the control node. It's imperative to ensure that the previously established 'kolla' user has access rights to the 'ceph.conf' and 'ceph.client.admin.keyring' configuration files. One can refer to Code Snippet 4.6 for an in-depth understanding of this stage.

Code Snippet 4.6: The configuration of Ceph for Kolla-Ansible

```
sudo scp root@"$CEPH_NODE_IP":/etc/ceph/ceph.conf
/etc/kolla/config/cinder/ceph.conf
sudo scp root@"$CEPH_NODE_IP":/etc/ceph/ceph.client.admin.keyring
/etc/kolla/config/cinder/cinder-volume/ceph.client.admin.keyring
sudo scp root@"$CEPH_NODE_IP":/etc/ceph/ceph.client.admin.keyring
/etc/kolla/config/nova/ceph.client.admin.keyring

sudo chown kolla:kolla /etc/kolla/config/cinder/ceph.conf
sudo chown kolla:kolla
/etc/kolla/config/cinder/cinder-volume/ceph.client.admin.keyring
sudo chown kolla:kolla
/etc/kolla/config/nova/ceph.client.admin.keyring
sed -i 's/\t//g' /etc/kolla/config/cinder/ceph.conf
sed -i 's/\t//g'
/etc/kolla/config/cinder/cinder-volume/ceph.client.admin.keyring
sed -i 's/\t//g' /etc/kolla/config/nova/ceph.client.admin.keyring
```

5. **OpenStack Cluster Deployment:** The final and most crucial step encompasses a sequence of commands that initiate the OpenStack cluster deployment. The servers are first bootstrapped using the 'multinode' inventory file. This is followed by a series of pre-checks and the ultimate deployment of the cluster. Upon successful completion of the deployment, the OpenStack client is installed, enabling direct interaction with the OpenStack cluster from the control node. These steps are documented in Code Snippet 4.7.

Code Snippet 4.7: The configuration of Ansible

```
kolla-ansible -i ./multinode bootstrap-servers
kolla-ansible -i ./multinode prechecks
kolla-ansible -i ./multinode deploy
pip install python-openstackclient -c
https://releases.openstack.org/constraints/upper/master
```

In summary, this script systematically guides the installation of the Kolla-Ansible-based OpenStack cluster, from environment preparation and component installation to configuration and deployment, with each step geared towards achieving a streamlined and fully operational OpenStack cluster.

Upon a detailed examination of the commands required to set up an OpenStack cluster with Kolla-Ansible, Figure 4.1 offers a clearer visualization of the deployment flow for the OpenStack infrastructure. The process begins with the execution of init-communication.sh, the initial script developers use to establish access to the Control node of the cluster. This culminates with the OpenStack Dashboard - Horizon launch, granting comprehensive control over the OpenStack cluster. This control spans various aspects, including security configurations, the selection of images for launching new instances, and numerous other customizable features tailored to meet specific development needs.

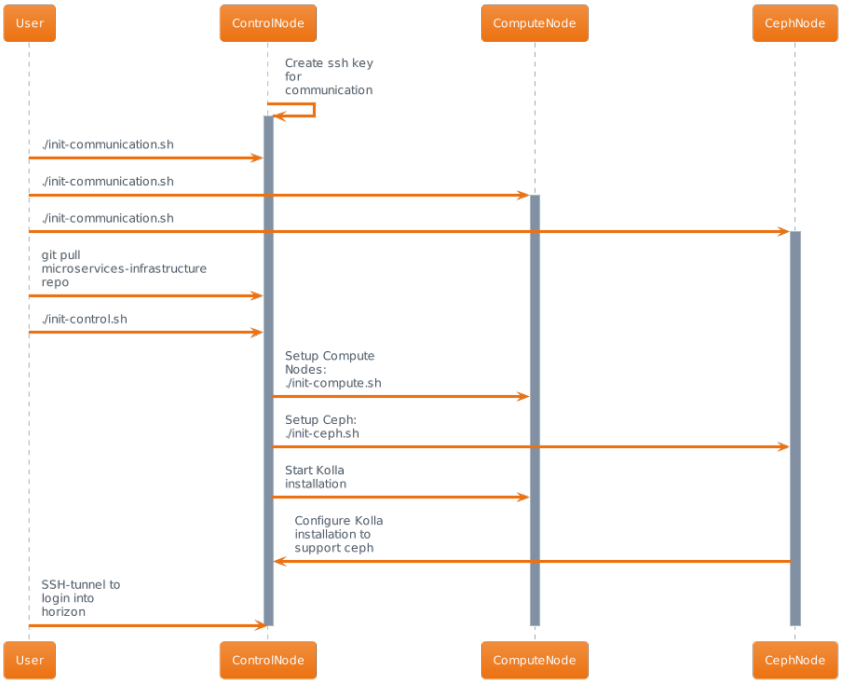

Figure 4.1.: Proposed OpenStack Installation and setup workflow automated with Bash and Kolla-Ansible (own illustration)

4.3. Layer 2: Kubernetes

For the deployment of the Kubernetes cluster, we have decided to use Terraform together with Terragrunt to automate the deployment process. Created by HashiCorp, Terraform is a leading IaC software. It allows developers and operations teams to define and provision infrastructure using a declarative configuration language, ensuring consistent and reproducible infrastructure provisioning. With its cloud-neutral stance, Terraform supports various providers, encompassing AWS, Google Cloud, OpenStack, and many others. Beyond its broad cloud support, Terraform has also shown other benefits, such as :

- **Modularity and Reusability**: Terraform modules can be used to create reusable infrastructure components, which is particularly beneficial for standardizing Kubernetes cluster configurations.
- **State Management**: Terraform maintains the state of your infrastructure, making incremental changes predictable and efficient.
- **Plan & Apply Mechanism**: Before applying changes, Terraform displays a plan, ensuring transparency and avoiding unintended changes.

Terragrunt, designed as a complementary tool to Terraform, acts as a thin wrapper, adding additional functionalities to simplify and enrich the Terraform user experience. One of its standout features is the efficient remote state management, allowing teams to have refined control over remote state configuration, thus promoting state consistency across various teams. Further emphasizing the "Don't Repeat Yourself" (DRY) principle, Terragrunt aids in averting repeated configurations, making extensive Terraform configurations considerably more manageable. Additionally, when orchestrating the intricate components of a Kubernetes cluster, Terragrunt's capability to handle dependencies between modules becomes indispensable. This functionality ensures that interrelated components are set up coherently and without hassle while also being one of the main decision factors in our adoption of Terragrunt for the Kubernetes deployment.

While Kubernetes is undeniably a powerful platform for container orchestration, its setup can be quite a complex task. This is where tools like Terraform and Terragrunt come into play. Firstly, they promote consistency. By using Terraform, organizations can ensure that every Kubernetes cluster is provisioned uniformly for development, staging, or production. This drastically reduces the occurrence of environment-specific anomalies. Secondly, as businesses evolve and the demands on Kubernetes increase, Terraform and Terragrunt templates can be easily modified and redeployed, providing a scalable infrastructure solution without the heavy manual overhead. Lastly, regarding team collaboration, Terraform's state management system and Terragrunt's configuration enhancements ensure that multiple team members can work together without interference, leading to safe and coordinated infrastructure alterations. In conclusion,

Terraform and Terragrunt form together a superior toolkit for Kubernetes deployment. Their capabilities in codifying infrastructure, guaranteeing consistency, and facilitating scalability make them indispensable for any organization eager to leverage Kubernetes efficiently and seamlessly.

Deploying a Kubernetes cluster within an OpenStack infrastructure using Terraform requires several configuration files that, together, sculpt the desired infrastructure.

- **'variables.tf'**: This file is conventionally used to declare and describe the variables that will be used within your Terraform configuration. This intentional separation of variable definitions from the main configuration amplifies modularity and readability. This separation of variables from the main configuration achieves two significant goals: configurations can be easily adapted every time Terraform is executed, and if a value isn't supplied during execution, the defaults supplied in this file take over. For our Kubernetes deployment, variables such as "availability_zone", "server_flavor_id" (which corresponds to the Control Plane), "agent_flavor_id" (pointing to the Nodes), "image_id", and "external_net_name" play pivotal roles.

- **prerequisites.tf**: In our specific implementation, this file serves two main purposes: it defines shared resources essential for deploying other primary resources and performs data lookups. Using the data block, this file fetches existing resources or configurations that the rest of the Terraform code relies on. As illustrated in (**Code Snippet A.4**), several OpenStack networking configurations are set up as prerequisites. Notable resources defined in this context include "openstack_compute_keypair_v2", "openstack_networking_network_v2", "openstack_networking_subnet_v2" and several others. These resources must be in place before it proceeds with the primary deployment tasks encapsulated in main.tf.

- **main.tf**: The main.tf file is often considered the cornerstone of Terraform deployments. It addresses multiple areas, starting with Provider Configurations, which are pivotal for aligning the Kubernetes cluster within the OpenStack infrastructure.

Code Snippet 4.8: The Provider Configuration in main.tf

```
provider "kubernetes" {
  host = module.server.k3s_external_url
  token = local.token
  cluster_ca_certificate = data.k8sbootstrap_auth.auth.ca_crt
}
provider "openstack" {
  user_name = "admin"
  tenant_name = "admin"
  region = "RegionOne"
}
```

Module invocations represent another essential feature, and for the current setup, the deployment is based on a specialized fork of the tf-k3s module. This fork was necessary to meet Debian-specific requirements for the instances that are part of the Kubernetes cluster, leading to a change in the method of k3s installation from curl to get. The forked module, originating from the repository https://github.com/iuliacornea99/tf-k3s/tree/main, facilitates the provisioning of K3s nodes, allowing the setting up of a multinode cluster. In Code Snippet A.5, the "server" module facilitates the deployment of a Control Plane node on an OpenStack infrastructure. Conversely, the "agent" module is responsible for deploying worker nodes within the same infrastructure. Given the constraints of limited resources — characterized by a scarcity of machines and restricted memory — the Terraform scripts configure a Kubernetes cluster with a single Control Plane and one Agent node. Nevertheless, scaling the number of nodes in the Kubernetes cluster can be trivial. This can be achieved by adjusting the 'count' property in the module and modifying the flavors available from OpenStack.

Additionally, the main.tf file houses the definitions of core resources. These are critical tokens required by Kubernetes when using kubeadm to bootstrap the cluster on an OpenStack setup. Variables like cluster_token for cluster authentication, and bootstrap tokens bootstrap_token_id and bootstrap_token_secret for worker nodes are all defined in Code Snippet 4.9.

Code Snippet 4.9: The Resource Definitions in 'main.tf'

```
resource "random_password" "cluster_token" {
  length = 64
  special = false
}
resource "random_password" "bootstrap_token_id" {
  length = 6
  upper = false
  special = false
}
resource "random_password" "bootstrap_token_secret" {
  length = 16
  upper = false
  special = false
}
```

Finally, the outputs section, situated at the end of the main.tf file, provides indispensable information about the resources that have been deployed, such as the kubeconfig file and the IPs of the Control Plane.

Given this setup, we have a flexible way to deploy a Kubernetes cluster within OpenStack. The structured separation of concerns using the files ensures modularity, easy troubleshooting, and scaling.

The deployment process is split into prerequisite steps and core cluster deployment steps. Within the main Terraform directory, you'll navigate to a subdirectory named prerequisites. Here, you'll initialize Terragrunt and proceed with planning and applying your configurations, which set up the necessary conditions for the core cluster deployment. Once the prerequisites are met, you'll return to the main terraform/cluster directory to initialize, plan, and apply the core configurations. These steps are executed using Terragrunt commands in a sequential manner, as can be seen in Code Snippet 4.10.

Code Snippet 4.10: Commands for deploying the Kubernetes cluster on top of OpenStack infrastructure

```
cd microservices-infrastructure/terraform/cluster/prerequisites
terragrunt init
terragrunt plan -out prerequisites
terragrunt apply prerequisites
cd ..
terragrunt init
terragrunt plan -out cluster
terragrunt apply cluster
```

After the core cluster is deployed, you'll need to retrieve the kubeconfig file that allows Kubectl to interact with the cluster. Terragrunt can output this file; the content can be stored in your local kubeconfig file. Now, one is set to deploy test applications to the cluster using Kubectl. Executing a few more Kubectl commands, which can be found in Code Snippet 4.11, will allow you to ascertain the deployment and pods' status, as well as the specific Kubernetes service's details.

Code Snippet 4.11: Commands for deploying a Kubernetes test app (provided in microservices-infrastructure/test-deployment.yaml)

```
kubectl apply -f test-deployment.yaml
kubectl get deployment
kubectl get pods
kubectl get service kube-test-container
```

In the current prototype, the Kubernetes cluster may employ port forwarding for external access to applications, necessitating additional steps to retrieve the IP of the control plane and the service's port number. These can be fetched using Kubectl, facilitating your interaction with the deployed application. Finally, to access your deployed applications, the developer needs to establish an SSH tunnel to the cluster. This tunnel is directed to the specific IP and port that the service within the Kubernetes cluster is using.

Thus, by following these well-defined steps, which can also be observed in Figure 4.2 and making use of powerful tools like Terraform, Terragrunt, and Kubectl, deploying a Kubernetes cluster on an OpenStack environment becomes a well-structured and automated task.

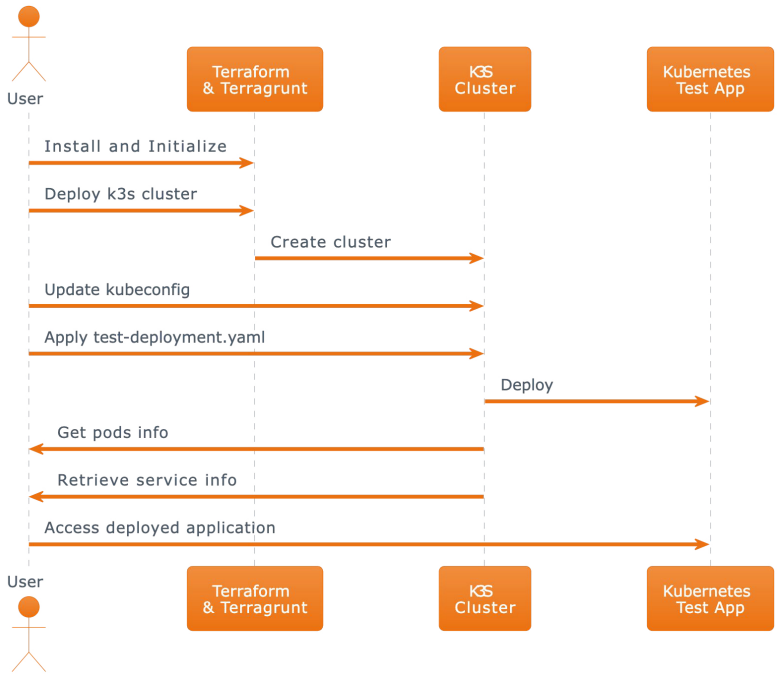

Figure 4.2.: Proposed Kubernetes Cluster Setup Workflow using Terraform (own illustration)

4.4. Layer 3: GitOps (Argo CD)

The final layer in our microservices architecture is orchestrated by Argo CD. Following the establishment of the Kubernetes cluster in the preceding phase, we now focus on deploying Argo CD atop this cluster using Helm. As a package manager for Kubernetes, Helm functions like 'apt' for Linux or 'brew' for macOS. It empowers developers and operators to define, install, and upgrade intricate Kubernetes applications by using "charts". In the context of Kubernetes, charts are bundled configurations of pre-defined resources, ranging from individual applications and services to a collection of related Kubernetes components. These charts facilitate the streamlined deployment of applications onto a Kubernetes cluster.

Opting for Helm as our installation and setup methodology offers several advantages over traditional approaches. Not only does it allow for consistent deployment of Argo CD across various environments and clusters, but it also enhances version management and control over these environments. Furthermore, when paired with Terraform scripts designed for the Kubernetes cluster deployment, Helm charts contribute to a fully automated deployment process, covering both the cluster and the required packages.

The existing automation framework for deploying Argo CD is built around a single Bash script and multiple YAML templates. These templates are dynamically configured

based on user input parameters collected during the execution of the Bash script. This setup significantly enhances the automation and customization capabilities of the Argo CD deployment process. With this contextual overview in place, we shall now delve into the individual functionalities of each file:

- **argo.sh**: The primary function of this file is to gather essential data to customize the three associated template files. It requires details like the domain name, which will be utilized to expose the Kubernetes cluster, and the email address needed for Let's Encrypt certificates.

  Let's Encrypt is a free, automated, and open Certificate Authority(CA) that aims to make the process of obtaining and managing Secure Sockets Layer (SSL)/Transport Layer Security (TLS) certificates as simple and accessible as possible. Its primary objective is to promote a more secure web by ensuring that websites adopt encrypted connections. Through its Automatic Certificate Management Environment (ACME) protocol, Let's Encrypt provides domain-validated certificates to over a million websites globally, making it one of the most popular CAs. These certificates not only enhance the security of web traffic by encrypting it but also boost user trust by verifying the website's authenticity.

  Upon securing the necessary data inputs, the script substitutes the placeholders in the YAML templates with the corresponding user-provided values.

- **helm-argo.yaml.tmpl**: This YAML template file is structured to facilitate the installation of Argo CD using Helm charts.

  The initial section of the file establishes a Namespace resource named argocd. This dedicated namespace ensures that all resources associated with Argo CD are systematically grouped together, offering both organized management and resource isolation.

  Subsequent to the namespace definition, a HelmChart resource, named argo, is specified to reside within the kube-system namespace. This section indicates that the Helm chart for Argo CD should be retrieved from the repository located at https://argoproj.github.io/argo-helm and explicitly points to the argo-cd chart. The target namespace for the deployment is identified as the previously created argocd namespace.

  The valuesContent segment of the current YAML template provides custom configurations tailored for the Argo CD installation. This section outlines the domain name previously provided in the argo.sh script and integrates the Dex configuration for GitHub OAuth authentication, which includes the clientID, clientSecret, and GitHub organization name.

Lastly, the YAML delineates RBAC configurations for the platform. Explicit policies detail the privileges and restrictions for a role called "role:org-admin". These policies provide granular control over actions, such as get and create, on Argo CD resources, including applications, clusters, repositories, and logs. To cement the hierarchical structure, the team "Admins" defined in the "thesis-openstack" organization has been assigned the privileges of a previously defined role: "org-admin".

In essence, this YAML file (which can also be seen at Code Snippet A.6) orchestrates a tailored deployment of Argo CD using a Helm chart, cooperating with GitHub for user authentication and configuring distinct roles and permissions tailored for administrative oversight within the organization.

- **ingress.yaml.tmpl**: This YAML file outlines a configuration designed for establishing ingress for Argo CD, utilizing Traefik as its primary ingress controller. This file defines an IngressRoute resource named argocd-server under the argocd namespace. This resource defines the routing instructions and criteria for incoming traffic. Specifically, there are two distinct route rules articulated.

  The first rule is focused on standard traffic routing. This rule is set to match traffic directed at a host specified by the provided hostname variable in argo.sh script. Given a priority level of 10, traffic that aligns with this rule will be forwarded to the service named argo-argocd-server, utilizing port 80 for communication.

  The second rule uniquely accommodates gRPC traffic. Not only does it match the same host indicated by the provided hostname, but it also requires that the incoming traffic possesses a Content-Type header set to application/grpc. With a priority of 11, higher than the standard rule, this ensures a proper distinction between general traffic and gRPC traffic. Traffic that adheres to this rule is then directed to the same service, argo-argocd-server, on port 80. However, this rule has a notable distinction as it leverages the h2c scheme, which is specifically designed to manage HTTP/2 traffic without TLS, a common protocol for gRPC exchanges.

  The concluding section, labeled tls, signifies the security aspect of the configuration. It points out that the TLS certificates, crucial for encrypting and securing the traffic, will be sourced using a resolver tagged as le, which refers to Let's Encrypt, the certificate authority that secures incoming and outgoing traffic.

  To conclude, this YAML template (which can be found in Code Snippet A.7) crafts an ingress setup for Argo CD that adeptly segregates standard traffic from gRPC traffic, routing each appropriately. This is achieved through Traefik's capabilities, and the secure layer is ensured by integrating Let's Encrypt.

- **traefik.yaml.tmpl**: This YAML file outlines the configuration for creating the Traefik IngressRoute in the deployed Kubernetes cluster, focusing on exposing the Argo CD service. The resource type is specified as a HelmChart, named traefik, and resides within the kube-system namespace. Helm, being a package manager for Kubernetes, will handle the deployment using the specified Traefik chart fetched from an external URL.

  In the configuration, the providers section specifies that Traefik will use Kubernetes Ingress resources for backend service discovery. This is particularly useful for the auto-discovery of services within the Kubernetes cluster. The "publishedService" option is enabled, which allows Traefik to auto-detect the correct external IP for routing traffic. The image used for deploying Traefik is specifically tagged as 2.9.4 and sourced from Rancher's mirrored library. This ensures that a specific, tested version of Traefik is being used. The tolerations specified allow Traefik to be scheduled on critical nodes, including the control-plane and master nodes, ensuring its high availability and fault tolerance.

  A notable feature is the support for dual IP stack, both IPv4 and IPv6, as indicated by the ipFamilyPolicy set to "PreferDualStack". This enhances the networking capabilities of the deployed Traefik service. When it comes to SSL/TLS certificate management, the "additionalArguments" section is crucial. It configures Traefik to use Let's Encrypt by specifying the TLS challenge method for domain verification, an email address for administrative notifications, and a storage path for the acquired certificates. Lastly, persistence is enabled, ensuring that important data, like certificates, are not lost upon pod restarts.

  Overall, this YAML file is meticulously designed to deploy the Traefik IngressRoute with optimal settings for high availability, security through Let's Encrypt, and compatibility with Argo CD.

In conclusion, integrating Argo CD with a Kubernetes cluster offers a streamlined solution for continuous deployment. This GitOps approach ensures automated synchronization, promoting transparency and consistency. Combined with Kubernetes' scalability and resilience, a robust, efficient, and modern deployment strategy is achieved.

4.5. System Overview

The proposed architecture intertwines the power of OpenStack, Ceph, Kubernetes, and Argo CD to create a robust, efficient, and scalable infrastructure.

At the foundation of our system, we have OpenStack, a comprehensive open-source platform for cloud computing. OpenStack manages vast computing resources with a suite of interrelated services. Essential to this setup is the integration of Ceph, a unified

storage solution. Ceph seamlessly integrates with OpenStack, providing highly scalable and redundant storage. Its distributed nature ensures there are no single points of failure, and data is evenly distributed across the cluster. This symbiotic relationship enhances performance, fault tolerance, and system resiliency.

Atop this foundation, we deploy Kubernetes, a leading container orchestration system. Running Kubernetes on OpenStack harnesses the elasticity of cloud resources to manage containerized applications. With OpenStack's flexible compute offerings, Kubernetes nodes can be rapidly provisioned, scaled, or migrated, catering to dynamic workloads. Additionally, the native networking features of OpenStack augment Kubernetes' networking capabilities, ensuring secure and seamless communication between pods.

Argo CD is integrated into the Kubernetes cluster to elevate our system's efficiency. Argo CD, a declarative GitOps continuous delivery tool, ensures that the deployed resources inside Kubernetes match the configurations stored in Git repositories. This setup promotes consistency, version control, and transparency. It also simplifies deployment processes and accelerates development cycles.

This multi-layered architecture is not just a stacking of technologies but an interdependent interaction. With Ceph backing OpenStack, we ensure high availability and scalable storage, while Kubernetes offers application scalability and management. Argo CD's inclusion creates an automated, transparent, and reliable deployment process. Together, they forge an infrastructure that is adaptable to varying workloads, resilient against failures, and primed for continuous development and deployment.

In essence, this system architecture, which can also be seen in Figure 4.3 encapsulates a future-ready infrastructure model that harnesses the strengths of leading open-source technologies to deliver unparalleled performance and agility.

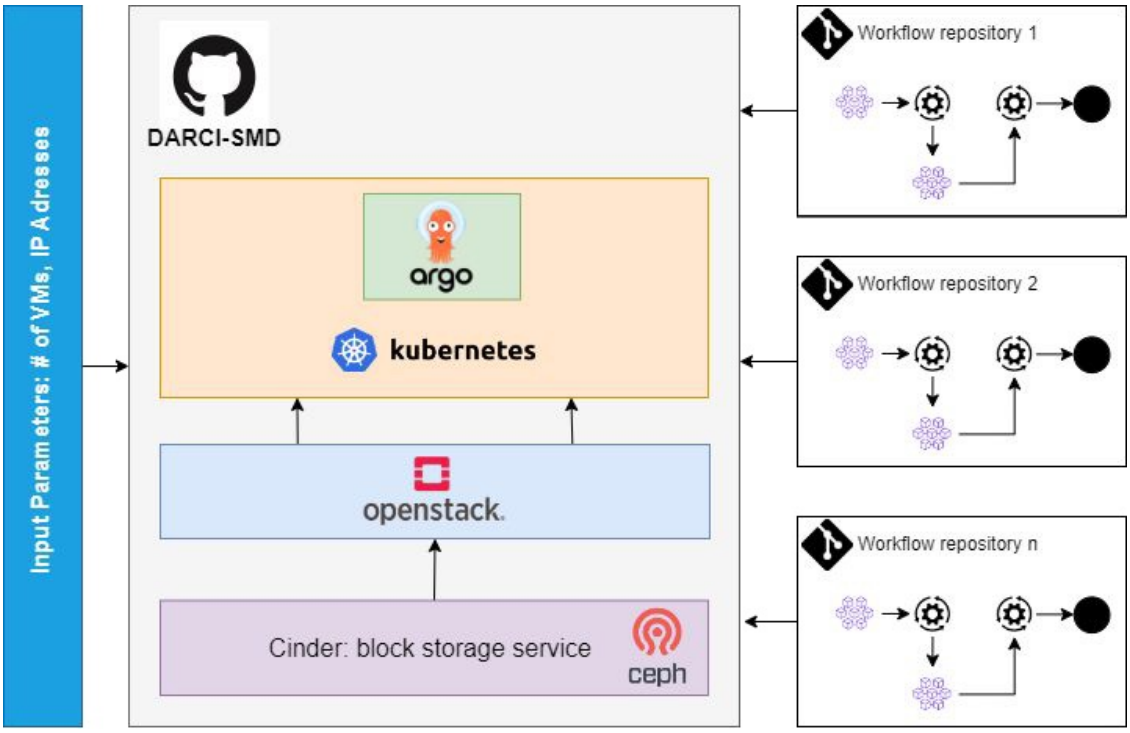

Figure 4.3.: Proposed system overview containing Ceph Block Storage, OpenStack Infrastructure layer, Kubernetes cluster deployed with Terraform and Argo CD running as a Kubernetes Controller (own illustration)

5. Evaluation

5.1. OpenStack Layer

Given the complex nature of OpenStack installation, extensive research on various installation methodologies to discern the most suited one to our use case was vital. In our study, we encountered the OpenStack Survey Report Analytics [48], a collection of yearly user-submitted reports. The contained data is categorized into four sections: Demographics (which traces organization-specific information), Deployments (exploring the popularity and operational environments of OpenStack services), Deployment Decisions (mainly addressing OpenStack installation methodologies, drivers, and packages used for each service), and Cloud Size (analyzing user pool sizes, instance numbers, provisioned storage, and more). Among these, the Deployment Decisions section was the most relevant to our research.

The optimal installation method is crucial for establishing a stable and scalable OpenStack infrastructure. Thus, we have analyzed the reports spanning from 2018 to 2022 to identify the most favored installation methods, such as Ansible [49], Kolla-Ansible [45], DevStack [44], Puppet [50], Juju [51], and OpenStack-Ansible [52]. As depicted in Figure 5.1, Kolla-Ansible appeared as one of the methods showing a significant rising trend, establishing it as a current best practice for installing OpenStack. Moreover, Kolla-Ansible is highly recommended for production environments, unlike DevStack, which is more suited for crafting small prototypes and gaining a basic understanding of primary OpenStack services.

While Ansible is a widely used installation method, it demands substantial pre-setup and high expertise in the domain beforehand. On the other hand, Kolla-Ansible offers a ready-to-use solution with extensive customization options, thus cementing it as our chosen method for OpenStack installation. Post installation, however, we uncovered some insights: the OpenStack documentation lacked in covering several crucial aspects, which we discovered in developers' repositories or various blog posts, making the Kolla-Ansible installation process quite inconvenient and challenging. The absence of instructions for a "basic setup" further increased the complexity of the OpenStack installation, especially for individuals unfamiliar with OpenStack services.

An intriguing observation was the notable decrease in responses to the OpenStack Survey Report, as shown in Table 5.1. This could be attributed to the complexity of OpenStack services and the evolving trends within the IT industry. Although OpenStack

offers a high degree of customization, the increased complexity in setup and maintenance may discourage smaller companies, particularly startups, from adopting OpenStack and push them towards more user-friendly, ready-to-use cloud infrastructures like AWS or Google Cloud.

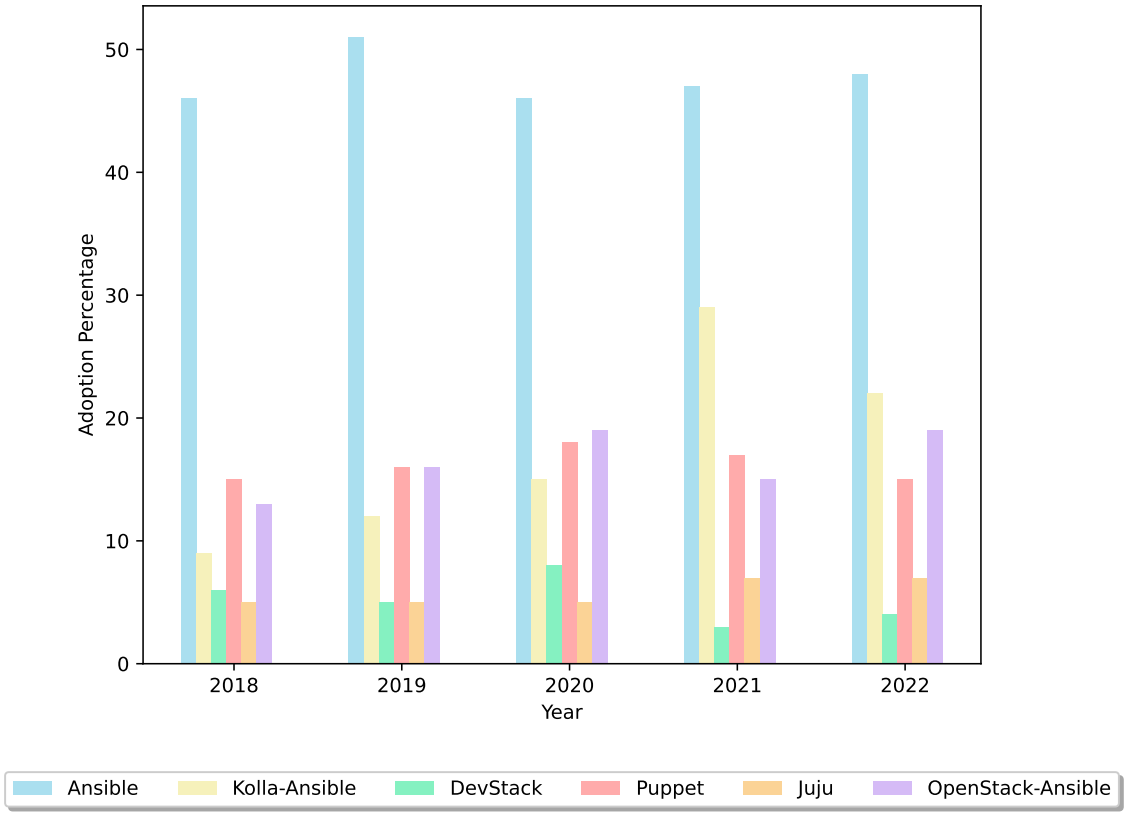

Figure 5.1.: Statistics for the most used tools for installing OpenStack over the period 2018-2022 based on [48] (own illustration)

Year	Number of Responses
2018	704
2019	383
2020	195
2021	251
2022	246

Table 5.1.: Statistics for the number of OpenStack survey responses per year over the period 2018-2022 based on [48]

5.2. Kubernetes Layer

In the evaluation of IaC tools for deploying a Kubernetes cluster on top of an OpenStack infrastructure, several tools were analyzed, and Terraform appeared to be the most fitting, outperforming others like Chef[53], Puppet[50], Ansible[49], and CloudFormation[54] in several aspects relevant to the project's goals. The following analysis was structured based on Table 5.2, which compiles the main features of each IaC tool analyzed [55]:

- **Configuration Management vs. Orchestration**: Configuration management tools like Chef, Puppet, and Ansible primarily handle the installation and management of software on existing servers, contrasting to the orchestration nature of Terraform and CloudFormation that provisions the servers themselves. Given the project's emphasis on deploying scalable microservices, an orchestration tool like Terraform proved more pertinent for provisioning the required server infrastructure.

- **Mutable vs. Immutable Infrastructure Paradigms**: One significant challenge with mutable configuration management tools is the possibility of configuration drift over time, leading to inconsistency across servers. This is not only challenging to diagnose but also almost impossible to replicate. On the other hand, Terraform, paired with other image creation technologies like Docker or Packer, follows an immutable infrastructure paradigm where every change is a deployment of a new server. This diminishes the tendency for configuration drift, providing a more predictable and manageable infrastructure crucial for scalable microservices deployment.

- **Declarative vs. Procedural Coding Style**: The procedural style encouraged by Chef and Ansible requires a step-by-step specification to achieve the desired state. On the other side, Terraform, similar to CloudFormation and Puppet, adopts a declarative style where the end state is specified, and the tool orchestrates the necessary steps independently. This abstraction facilitates easier scaling and management, as observed when deploying additional servers or altering configurations, making Terraform an ideal fit for managing the evolving infrastructure requisite for scalable microservices deployment. The declarative style of Terraform not only eases the declaration of the desired infrastructure state but also retains awareness of the created state. This state-awareness simplifies scaling operations, enhancing reusability and maintainability of the code, which are crucial for agile and iterative infrastructure management in microservices deployments.

- **Client/Server vs. Client-only Architecture**: The client/server architecture of Chef and Puppet requires additional server deployments for configuration management and agent installations on every server to be managed. This contrasts with the client-only architecture of Terraform, which liaises directly with cloud provider APIs, removing the need for additional software installations and extra hardware, hence simplifying the deployment and reducing the potential points of failure.

This architecture proved more favorable for maintaining the scalable and reliable infrastructure required by this project.

- **Cloud-Agnostic Nature**: Terraform's cloud-agnostic nature was also a crucial factor in its selection. Unlike CloudFormation, which is limited to AWS, Terraform's ability to interact with multiple cloud providers, including OpenStack, provided the desired flexibility and future-proofing necessary for a heterogeneous cloud environment, aligning with the project's objective of developing a reliable and scalable cloud infrastructure.
- **Community Levels and Maturity**: Terraform was chosen for its functional benefits and substantial, active community despite being a relatively young IaC tool with medium maturity compared to peers like Chef and Puppet. Its extensive resources, continuous enhancements, and vast contributor base have made it attractive, even as it transitions from Mozilla Public License (MLP) to Business Source License (BSL), potentially affecting its enterprise use. The OpenTofu initiative plans to sustain its open-source appeal by forking and maintaining previous versions, ensuring it remains a valuable tool amidst licensing changes and fostering an environment ripe for innovation and reliable infrastructure management vital for the project.

In conclusion, Terraform is this project's most suitable IaC tool due to its orchestration capabilities, immutable infrastructure paradigm, declarative coding style, state awareness, client-only architecture, and cloud-agnostic nature. The attributes not only aligned with the project's goals but also provided a robust foundation for the reliable and scalable deployment of microservices on a cloud infrastructure.

IaC Tool	Terraform	Chef	Puppet	Ansible	CloudFormation
Code	Open source	Open source	Open source	Open source	Closed source
Cloud	All	All	All	All	AWS only
Type	Orchestration	Configuration Management	Configuration Management	Configuration Management	Orchestration
Infrastructure	Immutable	Mutable	Mutable	Mutable	Mutable
Language	Declarative	Procedural	Declarative	Procedural	Declarative
Architecture	Client-Only	Client/Server	Client/Server	Client-Only	Client-Only
Community	Huge	Large	Large	Huge	Small
Maturity	Medium	High	High	Medium	Medium

Table 5.2.: Comparison of IaC Tools based on code access, tool type, coding style, base architecture, community size and maturity [55]

5.3. GitOps Layer

Our research identified the adoption of GitOps principles as a crucial point in building robust, automated, and scalable infrastructure. GitOps is a powerful paradigm where Git repositories act as the single source of truth for both infrastructure and application code. This approach offers declarative control over infrastructure, augmenting traceability, visibility, and efficiency in the deployment pipeline, especially within a microservices architecture.

Two dominant players in the GitOps field are Flux and Argo CD. Both tools provide solid platforms to make use of the GitOps advantages; however, after comprehensive analysis (detailed in Table 5.3) combined with the requirements specific to this project, Argo CD was chosen as the preferred GitOps tool over Flux based on the following aspects: ease of use, community backing, feature completeness, and integration capabilities with current systems.

Argo CD stands out with its mature and user-centric interface, offering clear visualization of deployment activities and improving the experience of managing and monitoring the system's state for developers. This feature was indispensable in troubleshooting and optimizing the deployment workflows. Complementing its interface, Argo CD boasts dedicated user and permission systems, representing the base of the ACL implementation. Furthermore, it accommodates SSO integrations with several providers, adaptable for both Graphical User Interface (GUI) and CLI, depending on team needs. These features were vital for this project, and the alternatives, such as Flux CD, did not fulfill all requirements.

Moreover, Argo CD's CRDs are more extensible, providing a more versatile mechanism for defining and managing applications. In the context of deploying a diverse set of microservices with varying configurations and dependencies, this level of flexibility became crucial. This flexibility extends to Argo CD's capacity to integrate with several CI/CD tools, ensuring a fluid and adaptable deployment pipeline.

Argo CD's well-established community support facilitated a proper environment for resolving technical problems easily, a less pronounced feature in Flux CD. The extensive documentation and active community engagement surrounding Argo CD significantly reduced the learning curve and sped the implementation stage related to integrating Argo CD on top of the deployed Kubernetes cluster. In addition, using Argo CD's capacity to manage multiple namespaces and clusters was crucial in deploying and managing the project's microservices on a larger scale. Despite being available on Flux CD, this functionality was found to be more advanced and user-friendly on Argo CD.

In conclusion, while both Flux and Argo CD are competent GitOps tools, the choice of Argo CD for this project was influenced by its superior user interface, configuration

flexibility, integration capabilities, and user-related permissions system. It catered more coherently to the specific requirements of building a reliable cloud infrastructure tailored for scalable microservices deployment with ACL capabilities.

GitOps Tool	Argo CD	Flux CD
Detect changes in Git	Pull-based (default: every 3 min.), push-based (Webhook)	Pull-based (default: every 5 min.), push-based (Webhook)
Application definitions	Application CRD	HelmRelease CRD, Kustomization CRD
Kubernetes resource definitions	Helm charts, Kustomize, YAML, Ksonnet, Jsonnet, JSON	Helm charts, Kustomize, YAML
Multi-cluster support	Yes	Yes
Multi-tenancy with multiple clusters	Yes	No
Automated sync	Yes	Yes
Manual sync	Yes	Yes
Cluster drift reconciliation (Self heal)	Yes	Not for all GitOps resource kinds (Yes for Kustomize, No for Helm Charts)
Garbage collection (Pruning)	Yes	Yes
Sync windows	Yes	No
Selective reconciliation	Yes	No
Sync hooks	Yes	No
Inline configuration in the GitOps resource	No	Yes
Graphical user interface	Yes	No
SSO integrations for GUI and CLI	Yes	No
Own user management system	Yes	No
Own permission system	Yes	No

Table 5.3.: Comparison of GitOps Tools [56]

6. Conclusion

In this research, we have conceptualized, developed, and automated a robust cloud infrastructure to ensure the scalable deployment of microservices, aligning with the project's aims defined in Chapter 3. The current infrastructure of the GHGA Munich data hub is built with OpenStack, which significantly guided the technology selection for this project based on the desire to continue with OpenStack while also seeking the required improvements in security and testing.

The prototype, elaborated in Chapter 4, was developed through a three-layered approach. The initial phase focused extensively on setting up a distinct OpenStack infrastructure, demanding significant time due to its learning curve and limited official documentation. Subsequently, we deployed a Kubernetes cluster on this OpenStack foundation, with Terraform easing the transition. Using Terraform modules, specifically designed for Kubernetes deployments on OpenStack, sped the development timeline and ensured a consistent and easily replicable infrastructure provisioning process.

The final phase of building our prototype was marked by the integration of state-of-the-art technology, GitOps, utilizing Argo CD — a choice based on its flexibility in defining and managing applications and the built-in users and permissions system, both of which are indispensable for deploying microservices and incorporating the ACL feature in our prototype.

We crafted the resulting prototype as an example of a robust platform capable of effortless scalability to production-ready infrastructure dimensions as resources permit. Moreover, it lays a substantial groundwork for advancing the core research project, "Reliable GHGA Infrastructure Using OpenStack: Safe, Secure, and Scalable Deployment of Microservices." The future steps in this project will contain augmented security measures and rigorous microservices testing.

The complete source code, together with installation steps and additional relevant implementation details, is made accessible for a more profound understanding at https://github.com/Evgeny-Volynsky/microservices-infrastructure/. This project not only significantly contributes to the GHGA's broader objective of establishing a resilient and scalable infrastructure but also pioneers a vital pathway towards achieving scalable microservices deployment specifically using sensitive data.

7. Future Work

German Human Genome-Phenome Archive is a robust initiative to centralize human omics data for research while ensuring robust protection against misuse. Bridging research and medical care, GHGA harnesses data and technology to pioneer new treatments and diagnostics. Committed to rigorous data protection, it integrates secure software with GDPR-compliant ethical-legal frameworks, focusing on patient needs. By centralizing scattered omics data, GHGA streamlines genomic research, enabling data-driven discoveries and their clinical integration. As part of the European Genome-Phenome Archive (EGA), GHGA boosts German research's international standing, positioning German scholars at the forefront of global genomic data exchange initiatives.

While navigating the evolving landscape of cloud computing and microservices, ensuring reliability, security, and scalability remains a cornerstone of effective deployment. This work serves as an instrumental first step in the significant research initiative entitled "Reliable GHGA Infrastructure Using OpenStack: Safe, Secure, and Scalable Deployment of Microservices". This research endeavor aims to resolve some of the most pressing challenges in microservices architecture, particularly in environments where security and data integrity are paramount. Through automating cloud infrastructure and ensuring scalable microservices deployment, this work lays the foundational groundwork, setting the stage for subsequent research phases that delve into various facets of reliability, safety, and security within the GHGA infrastructure. As we delve deeper into the intricate aspects of this domain, it is imperative to acknowledge how this initial phase has set the tone for subsequent explorations and advancements.

The intricate nature of our work with microservices, especially when dealing with confidential datasets, demands a sophisticated security framework. This is even more pressing when considering deployments in environments potentially lacking trust, such as specific third-party data centers. The potential for adversaries to breach confidentiality and integrity in such settings cannot be overlooked. As a countermeasure, we explore the possibilities presented by cutting-edge, hardware-driven, trusted computing platforms such as Intel SGX, AMD-SEV, and ARM Realms/Trustzone. Thus, we have two primary goals. Firstly, we aim to craft a secure execution environment supported by this trusted computing hardware, ensuring that the broader infrastructure cannot undermine the app's security. Secondly, we seek to design a distributed protocol for persistently managing the state of microservices, even on platforms that might not inherently guarantee security.

Reliability is another pivotal consideration in microservice application development, especially given the inherently decentralized nature of such undertakings. We recognize the inherent challenges and propose a holistic, dual-layered testing methodology to ensure optimal dependability. At its core, this strategy involves Fuzz testing to uncover and address semantic bugs. Complementing this is Crash testing, deployed on an expansive scale to gain a deeper understanding of the potential repercussions of both software and hardware anomalies on the larger application infrastructure.

Taking a step back and assessing the broader architectural framework, we acknowledge the formative design philosophy of our GHGA microservice architecture. While the initial focus was undeniably on functionality, it possibly came at the expense of a comprehensive security assessment. Moving forward, we recognize the need to rectify this potential oversight. The overall aim is a meticulous, in-depth security review of the GHGA microservice framework, ensuring no potential vulnerability remains unaddressed.

Given the identified concerns and proposed solutions, the research project's comprehensive plan can be segmented into four pivotal stages. Stage 1 focuses on Infrastructure Setup, encompassing the establishment of the Kubernetes cluster, integration of Ceph Storage, CI/CD workflows, and the implementation of basic ACL. Stage 2 emphasizes Microservices Security, integrating a confidential computing framework for microservices, including secure storage, networking, and a Secure Key Management Service (KMS). Stage 3 shifts the spotlight to Reliability Assurance, aiming to devise a framework designed to rigorously test the reliability of microservices. Lastly, Stage 4 culminates in a Comprehensive Security Review, evaluating the entire architecture, notably the OpenStack infrastructure operating the Kubernetes cluster. These stages are also graphically depicted in Figure 7.1.

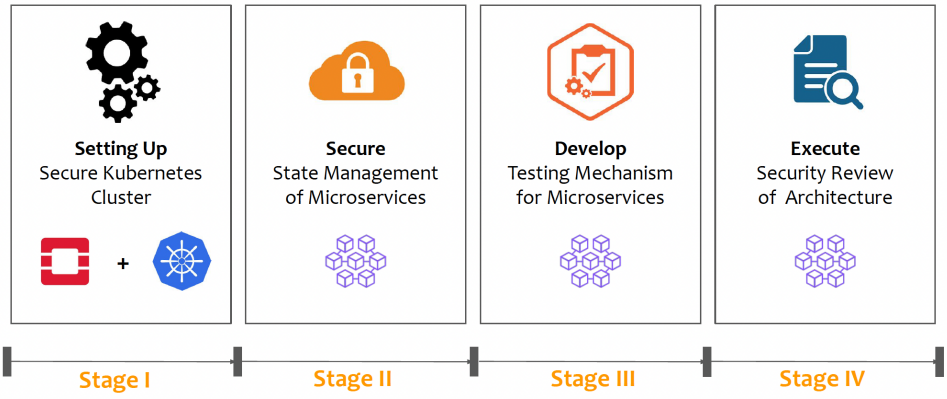

Figure 7.1.: Proposed stages of the main research project "Reliable GHGA Infrastructure Using OpenStack: Safe, Secure, and Scalable Deployment of Microservices" (own illustration)

A. Appendix

A.1. Code Snippets

Code Snippet A.1: Content of init-communication.sh script

```
#!/bin/bash
set -x -o errexit -o nounset -o pipefail

echo "Add the public key of the controller node that will be added to
the other nodes in the cluster"
PUBLIC_KEY_CONTROL=$(gum input --prompt "Enter the public key of the control node:")

echo "How many compute nodes you would like to setup?"
NO_COMPUTE_NODES=$(gum input --prompt "Enter the number of compute nodes:")

re='^[0-9]+$'
if ! [[ $NO_COMPUTE_NODES =~ $re ]]; then
  echo "Invalid input. Please enter a valid number."
  exit 1
fi

ip_addresses_compute_nodes=()

# Loop to prompt for IP addresses
for ((i = 0; i < $NO_COMPUTE_NODES; i++)); do
  ip=$(gum input --prompt "Enter IP address of compute node $(($i + 1)):")
  ip_addresses_compute_nodes+=("$ip")
done

# Display entered IP addresses
echo -e "Entered IP addresses:\n${ip_addresses_compute_nodes[*]}"

# Get the IP address for the node running ceph
export CEPH_NODE_IP=$(gum input --prompt "Please enter the IP address for
the node running ceph: ")
```

```
copy_and_execute_script() {
  local ip_addresses=("$@")

  for ip in "${ip_addresses[@]}"; do
    echo "Append public key to $ip..."
    ssh root@"$ip" "echo '$PUBLIC_KEY_CONTROL' >> /root/.ssh/authorized_keys"
  done
}

copy_and_execute_script "${ip_addresses_compute_nodes[@]}"

echo "Append public key to $CEPH_NODE_IP..."
ssh root@"$CEPH_NODE_IP" "echo '$PUBLIC_KEY_CONTROL' >> /root/.ssh/authorized_keys"
```

Code Snippet A.2: Configuration of multinode inventory deployment file

```
cat << EOF > multinode
[control]
control ansible_ssh_user=kolla ansible_become=True
ansible_private_key_file=/home/kolla/.ssh/id_ed25519
[network]
control ansible_connection=local
[compute]
EOF
# Add the ip addresses of the compute nodes to the multinode inventory file
for ip in "${ip_addresses_compute_nodes[@]}"; do
    echo "$ip ansible_ssh_user=kolla ansible_become=True
    ansible_private_key_file=/home/kolla/.ssh/id_ed25519" >> multinode
done
cat << EOF >> multinode
[monitoring]
control ansible_connection=local
[storage]
control ansible_connection=local
[deployment]
localhost ansible_connection=local
EOF
```

Code Snippet A.3: Configuration of globals.yaml file

```
cat << EOF > /etc/kolla/globals.yml
workaround_ansible_issue_8743: yes
kolla_base_distro: "ubuntu"
kolla_internal_vip_address: "$IP_ADDRESS"
enable_haproxy: "no"
neutron_external_interface: "br_ex_port"
network_interface: "$INTERFACE"
neutron_plugin_agent: "openvswitch"
enable_neutron_provider_networks: "yes"
enable_openstack_core: "yes"
nova_console: "novnc"
nova_compute_virt_type: "qemu"
enable_cinder: "yes"
enable_cinder_backup: "no"
enable_cinder_backend_lvm: "no"
enable_cinder_backend_iscsi: "no"
enable_openvswitch: "{{ enable_neutron | bool and neutron_plugin_agent != 'linuxbridge' }}"
fernet_token_expiry: 86400
glance_backend_file: "yes"
cinder_volume_availability_zone: internal
cinder_volume_group: "cinder-volumes"
cinder_backend_ceph: "yes"
ceph_cinder_user: "admin"
ceph_cinder_keyring: "ceph.client.admin.keyring"
EOF
```

Code Snippet A.4: The prerequisites of deploying Kubernetes cluster saved in prerequisites.tf

```
resource "openstack_compute_keypair_v2" "k3s" {
    name = "k3s"
    }

resource "openstack_networking_network_v2" "kubernetes" {
  name = "kubernetes"
  admin_state_up = "true"
    }

resource "openstack_networking_subnet_v2" "kubernetes" {
  network_id = "${openstack_networking_network_v2.kubernetes.id}"
  cidr = "192.168.199.0/24"
    }

data "openstack_networking_network_v2" "external_net"{
  name=var.external_net_name
    }

resource "openstack_networking_router_v2" "router" {
  name = "router"
  admin_state_up = "true"
  external_network_id = data.openstack_networking_network_v2.external_net.id
    }

resource "openstack_networking_router_interface_v2" "router_internal_interface" {
  router_id = openstack_networking_router_v2.router.id
  subnet_id = openstack_networking_subnet_v2.kubernetes.id
    }
```

Code Snippet A.5: The templates for the cloudinit files used for creating different types of nodes in a K3S clsuter

```
module "server" {
  source = "git::https://github.com/iuliacornea99/tf-k3s.git//k3s-openstack"
  name = "k3s-server"
  image_id = var.image_id
  flavor_id = var.server_flavor_id
  availability_zone = var.availability_zone
  keypair_name = openstack_compute_keypair_v2.k3s.name
  network_id = openstack_networking_network_v2.kubernetes.id
  subnet_id = openstack_networking_subnet_v2.kubernetes.id
  security_group_ids = [module.secgroup.id]
  data_volume_size = 1
  floating_ip_pool = var.external_net_name
  cluster_token = random_password.cluster_token.result
  k3s_args = concat(["server", "--cluster-init"], local.common_k3s_args)
  bootstrap_token_id = random_password.bootstrap_token_id.result
  bootstrap_token_secret = random_password.bootstrap_token_secret.result
}

module "agents" {
  source = "git::https://github.com/iuliacornea99/tf-k3s.git//k3s-openstack"
  count=1
  name = "k3s-agent-${count.index + 1}"
  image_id = var.image_id
  flavor_id = var.agent_flavor_id
  availability_zone = var.availability_zone
  keypair_name = openstack_compute_keypair_v2.k3s.name
  network_id = openstack_networking_network_v2.kubernetes.id
  subnet_id = openstack_networking_subnet_v2.kubernetes.id
  security_group_ids = [module.secgroup.id]
  data_volume_size = 5
  floating_ip_pool = var.external_net_name
  k3s_join_existing = true
  k3s_url = module.server.k3s_url
  cluster_token = random_password.cluster_token.result
  k3s_args = ["agent", "--node-label", "az=${var.availability_zone}"]
  depends_on = [
    openstack_networking_subnet_v2.kubernetes
  ]
}
```

Code Snippet A.6: Content of helm-argo.yaml.tmpl

```
apiVersion: v1
kind: Namespace
metadata:
  name: argocd
---
apiVersion: helm.cattle.io/v1
kind: HelmChart
metadata:
  name: argo
  namespace: kube-system
spec:
  repo: https://argoproj.github.io/argo-helm
  chart: argo-cd
  targetNamespace: argocd
  valuesContent: |-
    configs:
      params:
        server.insecure: true
      cm:
        url: https://$HOST_NAME
        dex.config: |
          connectors:
            # GitHub example
            -type: github
              id: github
              name: GitHub
              config:
                clientID: 7d1f1fedf59d92a14690
                clientSecret: b92bb8af06dc7937527640c37fd350ddbc9a0667
                orgs:
                -name: thesis-openstack
      rbac:
        policy.csv: |
          p, role:org-admin, applications, *, */*, allow
          p, role:org-admin, clusters, get, *, allow
          p, role:org-admin, repositories, *, *, allow
          p, role:org-admin, logs, get, *, allow
          p, role:org-admin, exec, create, */*, allow
          g, thesis-openstack:Admins, role:org-admin
```

Code Snippet A.7: Content of ingress.yaml.tmpl

```
apiVersion: traefik.containo.us/v1alpha1
kind: IngressRoute
metadata:
  name: argocd-server
  namespace: argocd
spec:
  routes:
    -kind: Rule
      match: Host(‘$HOST_NAME‘)
      priority: 10
      services:
        -name: argo-argocd-server
          port: 80
    -kind: Rule
      match: Host(‘$HOST_NAME‘) && Headers(‘Content-Type‘, ‘application/grpc‘)
      priority: 11
      services:
        -name: argo-argocd-server
          port: 80
          scheme: h2c
  tls:
    certResolver: le
```

Code Snippet A.8: Content of traefik.yaml.tmpl

```
apiVersion: helm.cattle.io/v1
kind: HelmChart
metadata:
  name: traefik
  namespace: kube-system
status:
  jobName: helm-install-traefik
spec:
  chart: https://traefik.github.io/charts/traefik/traefik-20.3.1.tgz
  set:
    global.systemDefaultRegistry: ''
  valuesContent: |-
    providers:
      kubernetesIngress:
        publishedService:
          enabled: true
    priorityClassName: "system-cluster-critical"
    image:
      name: "rancher/mirrored-library-traefik"
      tag: "2.9.4"
    tolerations:
    -key: "CriticalAddonsOnly"
      operator: "Exists"
    -key: "node-role.kubernetes.io/control-plane"
      operator: "Exists"
      effect: "NoSchedule"
    -key: "node-role.kubernetes.io/master"
      operator: "Exists"
      effect: "NoSchedule"
    service:
      ipFamilyPolicy: "PreferDualStack"
    additionalArguments:
      ---certificatesresolvers.le.acme.tlschallenge
      ---certificatesresolvers.le.acme.email=$SSL_EMAIL
      ---certificatesresolvers.le.acme.storage=/data/acme.json
    persistence:
     enabled: true
```

Abbreviations

ACL Access Control List

AIO All-In-One

API Application Programming Interface

AWS Amazon Web Services

CD Continuous Deployment

CI Continuous Integration

CLI Command-Line Interface

CNCF Cloud Native Computing Foundation

CPU Central Processing Unit

CRD Custom Resource Definition

GCP Google Cloud Platform

GHGA German Human Genome-Phenome Archive

gRPC Google Remote Procedure Call

GUI Graphical User Interface

HA High-Availability

HTTP Hypertext Transfer Protocol

IaaS Infrastructure as a Service

IaC Infrastructure as Code

JSON JavaScript Object Notation

KVM Kernel-based Virtual Machine

NIST National Institute of Standards and Technology

OAuth Open Authorization

OIDC OpenID Connect

RAM Random Access Memory

RBD Rados Block Devices

RBAC Role-based Access Control

REST Representational State Transfer

SSH Secure Shell

SSL Secure Sockets Layer

SSO Single Sign-On

TLS Transport Layer Security

UI User Interface

VM Virtual Machine

VPN Virtual Private Network

VPC Virtual Private Cloud

YAML Yet Another Markup Language

List of Figures

List of Tables

List of Code Snippets

Bibliography

[1] L. M. Vaquero, L. Rodero-Merino, J. Caceres, and M. Lindner. "A Break in the Clouds: Towards a Cloud Definition". In: *SIGCOMM Comput. Commun. Rev.* 39.1 (2009), pp. 50–55. ISSN: 0146-4833. DOI: 10.1145/1496091.1496100. URL: https://doi.org/10.1145/1496091.1496100.

[2] P. Mell and T. Grance. *The NIST definition of cloud computing*. Tech. rep. Jan. 2011. DOI: 10.6028/nist.sp.800-145. URL: https://doi.org/10.6028/nist.sp.800-145.

[3] Q. Zhang, L. Cheng, and R. Boutaba. "Cloud computing: state-of-the-art and research challenges". In: *Journal of Internet Services and Applications* 1.1 (Apr. 2010), pp. 7–18. DOI: 10.1007/s13174-010-0007-6. URL: https://doi.org/10.1007/s13174-010-0007-6.

[4] *Xen Project*. https://xenproject.org/. [Accessed 13-08-2023].

[5] *KVM*. https://www.linux-kvm.org/page/Main_Page. [Accessed 13-08-2023].

[6] *VMWare ESXi*. https://www.vmware.com/products/esxi-and-esx.html. [Accessed 13-08-2023].

[7] J. Surbiryala and C. Rong. "Cloud Computing: History and Overview". In: *2019 IEEE Cloud Summit*. 2019, pp. 1–7. DOI: 10.1109/CloudSummit47114.2019.00007.

[8] B. Furht. "Cloud Computing Fundamentals". In: *Handbook of Cloud Computing*. Springer US, 2010, pp. 3–19. DOI: 10.1007/978-1-4419-6524-0_1. URL: https://doi.org/10.1007/978-1-4419-6524-0_1.

[9] L. Kurup, C. Chandawalla, Z. Parekh, and K. Sampat. "Comparative Study of Eucalyptus , Open Stack and Nimbus". In: 2014. URL: https://api.semanticscholar.org/CorpusID:212586892.

[10] S. Sahasrabudhe and S. Sonawani. "Comparing openstack and VMware". In: Oct. 2014, pp. 1–4. DOI: 10.1109/ICAECC.2014.7002392.

[11] O. Sefraoui, M. Aissaoui, and M. Eleuldj. "OpenStack: Toward an Open-source Solution for Cloud Computing". In: *International Journal of Computer Applications* 55.3 (Oct. 2012), pp. 38–42. DOI: 10.5120/8738-2991. URL: https://doi.org/10.5120/8738-2991.

[12] *OpenStack - Open Source Cloud Computing Platform Software Map*. https://www.openstack.org/software/. [Accessed 23-08-2023].

[13] *OpenStack Compute Nova documentation*. https://docs.openstack.org/nova/latest/. [Accessed 21-08-2023].

[14] *OpenStack Neutron documentation*. https://docs.openstack.org/neutron/latest/. [Accessed 21-08-2023].

[15] *OpenStack Block Storage Cinder documentation*. https://docs.openstack.org/cinder/latest/. [Accessed 21-08-2023].

[16] *OpenStack Identity Service Keystone documentation*. https://docs.openstack.org/keystone/latest/. [Accessed 21-08-2023].

[17] *OpenStack Glance documentation*. https://docs.openstack.org/glance/latest/. [Accessed 21-08-2023].

[18] *OpenStack Conceptual architecture Installation Guide documentation*. https://docs.openstack.org/install-guide/get-started-conceptual-architecture.html#get-started-conceptual-architecture. [Accessed 23-08-2023].

[19] T. Rosado and J. Bernardino. "An Overview of Openstack Architecture". In: *Proceedings of the 18th International Database Engineering & Applications Symposium*. IDEAS '14. Porto, Portugal: Association for Computing Machinery, 2014, pp. 366–367. ISBN: 9781450326278. DOI: 10.1145/2628194.2628195. URL: https://doi.org/10.1145/2628194.2628195.

[20] N. Dragoni, S. Giallorenzo, A. Lluch-Lafuente, M. Mazzara, F. Montesi, R. Mustafin, and L. Safina. "Microservices: yesterday, today, and tomorrow". In: (June 2016).

[21] *Microservices — martinfowler.com*. https://martinfowler.com/articles/microservices.html. [Accessed 24-08-2023].

[22] O. Al-Debagy and P. Martinek. "A Comparative Review of Microservices and Monolithic Architectures". In: *2018 IEEE 18th International Symposium on Computational Intelligence and Informatics (CINTI)*. 2018, pp. 000149–000154. DOI: 10.1109/CINTI.2018.8928192.

[23] S. Newman. *Monolith to Microservices: Evolutionary Patterns to Transform Your Monolith*. O'Reilly Media, Incorporated, 2019. ISBN: 9781492047841. URL: https://books.google.ro/books?id=iul3wQEACAAJ.

[24] A. Khan. "Key Characteristics of a Container Orchestration Platform to Enable a Modern Application". In: *IEEE Cloud Computing* 4.5 (2017), pp. 42–48. DOI: 10.1109/MCC.2017.4250933.

[25] *Open Container Initiative*. https://opencontainers.org/. [Accessed 25-08-2023].

[26] A. Cepuc, R. Botez, O. Craciun, I.-A. Ivanciu, and V. Dobrota. "Implementation of a Continuous Integration and Deployment Pipeline for Containerized Applications in Amazon Web Services Using Jenkins, Ansible and Kubernetes". In: *2020 19th RoEduNet Conference: Networking in Education and Research (RoEduNet)*. IEEE, Dec. 2020. DOI: 10.1109/roedunet51892.2020.9324857. URL: https://doi.org/10.1109/roedunet51892.2020.9324857.

[27] H. van Merode. *Continuous Integration (CI) and Continuous Delivery (CD)*. Apress, 2023. DOI: 10.1007/978-1-4842-9228-0. URL: https://doi.org/10.1007/978-1-4842-9228-0.

[28] DZone. "Getting started with Kubernetes". In: *Kubernetes. Retrieved May* (2019).

[29] *Kubernetes: Production-Grade Container Orchestration — kubernetes.io*. https://kubernetes.io/. [Accessed 27-08-2023].

[30] L. A. Vayghan, M. A. Saied, M. Toeroe, and F. Khendek. "Kubernetes as an Availability Manager for Microservice Applications". In: *CoRR* abs/1901.04946 (2019). URL: http://arxiv.org/abs/1901.04946.

[31] M. Artac, T. Borovssak, E. D. Nitto, M. Guerriero, and D. A. Tamburri. "DevOps: Introducing Infrastructure-as-Code". In: *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*. IEEE, May 2017. DOI: 10.1109/icse-c.2017.162. URL: https://doi.org/10.1109/icse-c.2017.162.

[32] R. T. Yarlagadda. "DevOps and its practices". In: *International Journal of Creative Research Thoughts (IJCRT), ISSN* (2021), pp. 2320–2882.

[33] L. Zhu, L. Bass, and G. Champlin-Scharff. "DevOps and its practices". In: *IEEE software* 33.3 (2016), pp. 32–34.

[34] *What DevOps is to the Cloud, GitOps is to Cloud Native — weave.works*. https://www.weave.works/blog/gitops-is-cloud-native. [Accessed 30-08-2023].

[35] S. Gupta, M. Bhatia, M. Memoria, and P. Manani. "Prevalence of GitOps, DevOps in Fast CI/CD Cycles". In: *2022 International Conference on Machine Learning, Big Data, Cloud and Parallel Computing (COM-IT-CON)*. IEEE, May 2022. DOI: 10.1109/com-it-con54601.2022.9850786. URL: https://doi.org/10.1109/com-it-con54601.2022.9850786.

[36] A. S. B. Natale Vinto. *GitOps Cookbook - Kubernetes Automation in Practice*. 1st ed. O'Reilly Media, Inc., 2022. ISBN: 9781492097471.

[37] *Argo CD - Declarative GitOps CD for Kubernetes*. https://argo-cd.readthedocs.io/en/stable/. [Accessed 31-08-2023].

[38] *Options for Highly Available Topology — kubernetes.io*. https://kubernetes.io/docs/setup/production-environment/tools/kubeadm/ha-topology/. [Accessed 09-10-2023].

[39] *Railway*. https://railway.app/. [Accessed 23-09-2023].

[40] *Heroku - Cloud Application Platform*. https://www.heroku.com/home. [Accessed 23-09-2023].

[41] *Vercel*. https://vercel.com/. [Accessed 23-09-2023].

[42] *Netlify - Develop and deploy websites and apps in record time*. https://www.netlify.com/. [Accessed 23-09-2023].

[43] *Development and Automation of Reliable Cloud Infrastructure for Scalable Microservices Deployment — github.com*. https://github.com/Evgeny-Volynsky/microservices-infrastructure. [Accessed 05-09-2023].

[44] *DevStack documentation*. https://docs.openstack.org/devstack/latest/. [Accessed 07-09-2023].

[45] *Kolla Ansible documentation*. https://docs.openstack.org/kolla-ansible/latest/. [Accessed 07-09-2023].

[46] *Ceph Documentation*. https://docs.ceph.com/en/quincy/. [Accessed 10-09-2023].

[47] *Cephadm Ceph Documentation*. https://docs.ceph.com/en/quincy/cephadm/. [Accessed 10-09-2023].

[48] *OpenStack User Survey Analytics and Data*. https://www.openstack.org/analytics/. [Accessed 24-09-2023].

[49] *Ansible - Red Hat*. https://www.ansible.com/. [Accessed 24-09-2023].

[50] *Puppet*. https://www.puppet.com/. [Accessed 24-09-2023].

[51] *Juju*. https://juju.is/. [Accessed 24-09-2023].

[52] *OpenStack-Ansible documentation*. https://docs.openstack.org/openstack-ansible/latest/. [Accessed 24-09-2023].

[53] *Chef - Configuration Management System Software*. https://www.chef.io/products/chef-infra. [Accessed 25-09-2023].

[54] *AWS CloudFormation - Provision Infrastructure as Code*. https://aws.amazon.com/cloudformation/. [Accessed 25-09-2023].

[55] Y. Brikman. *Why we use Terraform and not Chef, Puppet, Ansible, Pulumi, or CloudFormation — blog.gruntwork.io*. https://blog.gruntwork.io/why-we-use-terraform-and-not-chef-puppet-ansible-saltstack-or-cloudformation-7989dad2865c. [Accessed 25-09-2023].

[56] D. Szakallas. *Comparison: Flux vs Argo CD — earthly.dev*. https://earthly.dev/blog/flux-vs-argo-cd/. [Accessed 26-09-2023].