EPiC Series in Computing

Volume 109, 2026, Pages 108–119

Proceedings of EUNIS 2026 Annual Congress

Developing and Maintaining Multi-Tenant, Multi-Cluster Kubernetes in a University Environment

Sven Haardiek, Markus Blank-Burian, Raimund Vogl

University of Münster, Germany
{shaardie,blankburian,rvogl}@uni-muenster.de

Abstract

The IT Department of the University of Münster (CIT) manages a diverse portfolio of infrastructure services. Its users include other IT departments, university management, researchers, and projects extending beyond the boundaries of the university. In response to the increasing adoption of containerized software [27], a Kubernetes cluster was established to provide a reliable and secure platform for software deployment. Our objective was to deliver a platform that meets or exceeds the reliability, security, and performance of legacy services. Accordingly, we implemented a setup combining multiple clusters distributed across several locations. By employing a single multi-tenant Kubernetes cluster rather than creating separate vanilla clusters for each tenant, we were able to maintain easy access, even for small projects. This approach allows tenants to focus on developing their applications using the provided tools and integrations, rather than managing the cluster itself.

1 Introduction

Aside from their other duties, the IT Department of the University of Münster (CIT) provides central IT infrastructure services to a diverse set of users, including other IT departments, administrative staff, internal teams, various researchers, as well as some external projects across North Rhine-Westphalia (NRW). Most of these workloads traditionally run on VMs, which means that teams need to manage not only their services but also the operating system and the integration of these services into the rest of the university's infrastructure. This includes, among others, certificate management, load balancing, firewall and whitelisting, DNS, security scans, and similar tasks. Some of these tasks are managed centrally but still require manual maintenance for all VMs, while others are continuously developed from scratch.

To centralize and automate some of these operations, the CIT created its first Kubernetes cluster, called ZIVKube (ZIV was the previous name of the CIT). The initial intention was to start with a relatively simple cluster, available only to CIT and ULB (University and State Library of Münster). Early users deployed their software on this cluster, which allowed both infrastructure providers and users to gain experience with Kubernetes and understand its benefits.

The first iteration was limited to a single data center, provided insufficient tenant isolation and proved to be difficult to maintain due to early architectural decisions. Because of these

limitations, we decided in 2020 to redesign the platform with stronger isolation guarantees, improved maintainability and multi-cluster resilience. The new Kubernetes platform was called **Uni Cloud Münster – Kubernetes**, or for short **the Kube**.

In this paper, we present the **Architecture and Design** (Section 3) of this service, explain the reasoning behind these decisions, and take a closer look at the implementation of key features, such as **Multi-Tenancy and Isolation** (Section 4) and **Multi-Cluster Management** (Section 5), followed by some **Operational Metrics** (Section 6) and insights from its ongoing production use.

2 Constraints and Requirements

Due to our experience with the previous iteration and the needs of the users, we decided on some requirements we wanted to fulfill with our Kubernetes platform. Additionally, we had some hard constraints given by the environment we were developing this solution in, which obviously influence the result. Understanding those constraints and requirements helps to understand the architecture and decisions we made during development.

Working in a university environment often means working with a limited amount of resources, in our case with a **small team** to operate the cluster, which also had other tasks to handle. So one of our requirements was to have as much **automation** as possible to be able to develop and maintain the setup without many manual interventions.

Next to Kubernetes, we also manage multiple OpenStack [19] clusters at different locations. Because of that, it only made sense to use this **existing cloud infrastructure** to host the new platform in virtual machines and use all benefits this cloud setup would offer. The OpenStack clusters were meant to be used by the same users as Kubernetes, so there was no real constraint on how many resources were used, since all resources assigned would benefit the same users, but with less overhead. Furthermore, the new platform targeted the same services and, just like OpenStack, was intended to be **no replacement for HPC or VDI environments**. As another benefit, we could spread our platform over multiple locations, aiming for **high availability** and even be able to offer our service if a complete data center at one of the locations went down.

As one of the lessons we learned from the previous iteration, we decided early on that Kubernetes needs to be **secure by design** and that the **isolation** between different users needs to be strict. Although it is much more likely that users interfere with each other unintentionally, we kept the scenario of a malicious user in mind during development.

Another key requirement was that the usage of the new platform is **easily accessible** and can be used in a **self-service manner** without frequent interventions by the cluster administrators, allowing users to quickly deploy and manage their applications, focus on development rather than infrastructure and experiment with new tools and services independently.

3 Architecture and Design

The architecture of our Kubernetes platform was designed to address the requirements and constraints described previously.

We aimed to create a flexible but robust setup with multiple environments, including development, staging, and production, while being resilient against failures at the infrastructure level, even at the data center level.

In this section, we describe the high-level cluster topology, the core components that enable the platform, and the integration with our underlying cloud infrastructure based on OpenStack [19]. We also explain how multi-tenancy and multi-cluster management are implemented.

3.1 High-Level Cluster Architecture

To ensure resilience against data center–like outages, we operate one cluster in each location, combined in a **multi-cluster** setup where traffic can easily flow between them. We deploy our platform in two completely separated OpenStack environments in two different locations, as already mentioned in Section 2. A more detailed description will follow in Section 5.

Despite that, our platform provides three environments: **development**, **staging**, and **production**. The **development** environment is only used by cluster administrators to test new features. This allows rapid development without affecting tenants, even if changes temporarily break the whole cluster. The **staging** environment serves both administrators and tenants. Administrators use it for meaningful tests with realistic workloads, while tenants can test their own services as well as new cluster features or changes before they are promoted to production. Finally, the **production** environment hosts stable services, with updates only after thorough testing in development and staging. Some tenants may also maintain their own staging workloads on staging or production, either to be close to development or to closely align staging and production setups.

For **replication**, we aim for five replicas for all important services so that we can easily survive an outage of two, as it is also recommended for etcd [12] in the Kubernetes Blog [28]. This way, even during an update, one of the replicas can fail without affecting availability.

To fulfill the different tasks, our Kubernetes cluster uses several **node groups**. Wherever possible, we run virtual machines in OpenStack for each node group with a *hard anti-affinity* rule to protect against hardware failures. Internal IP addresses are used whenever possible to prevent uncontrolled internet access and conserve the university's external IP resources. For stability and security, we run dedicated **control plane** nodes, hosting Kubernetes control plane pods, operators, and controllers for the critical services. **Regular nodes** host most cluster and tenant workloads, while **worker nodes** are slightly more powerful but do not require hard anti-affinity, as they are meant for processing jobs or holding unreplicated sessions. We also have nodes with special hardware. **GPU nodes** provide access to specialized vGPU hardware with different kinds of vGPUs. For access to services outside of the university network, we maintain **external nodes** with public IP addresses, and traffic for external services can be routed through these nodes, as described in Section 3.2. Figure 1 shows the number of nodes per group in production. We see that, as expected, there is little fluctuation and that, due to the partitioning, the fluctuation is largely limited to the worker and GPU nodes that are created when needed. The rest of the nodes are relatively stable. In practice, these groups are further subdivided by project or service, allowing finer-grained resource distribution and network segmentation.

Immutable and image-based deployments help prevent configuration drift by ensuring that infrastructure is always built from a known, versioned image rather than by modifying images in place, as described in [25]. Since we were planning to maintain Kubernetes over a long period of time, we decided to use **image-based deployment** for our virtual machines. This means that we do not manually maintain the virtual machines in our Kubernetes, but simply replace them with newly generated virtual machines based on pre-built operating system images in case of updates or errors.

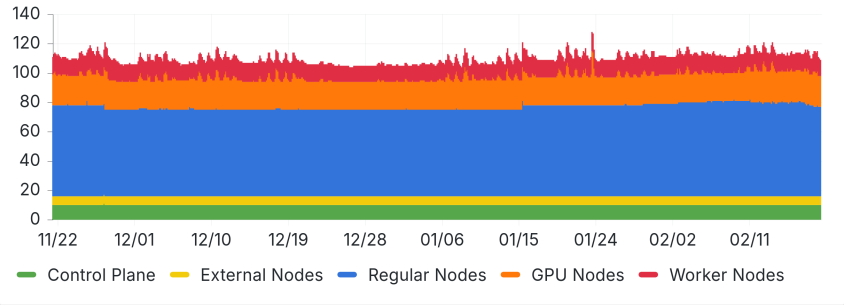

Figure 1: Number of Nodes by Group in production from November 2025 to February 2026 with dates in MM/DD format

3.2 Core Components

The Kubernetes platform is based on a set of core components that provide networking, traffic management, and operational lifecycle management services.

Cilium [5] is used as the **Container Network Interface (CNI)** for all clusters and provides the basis for pod-to-pod and pod-to-service networking. It uses eBPF for high-performance networking, as well as deep visibility into network flows inside the cluster. In addition, Cilium can also be used to enforce network policies as well as operate as an Egress Gateway [29] to route traffic from specific pods via our external nodes, as described in Section 3.1. Tenants can opt in to this by using a simple label and can apply network policies in the same way for external services as for internal ones.

Another core component of the clusters is **Istio** [14]. Its **service mesh** creates a new transparent infrastructure layer for communication within the cluster. Istio enables capabilities like observability, advanced traffic management, and security features like mTLS and is used as one of the components of our multi-cluster design, described in Section 5. Another primary use for Istio in our cluster is providing the **ingress gateway** to route traffic to the different services in our Kubernetes.

To operate the platform in a reproducible and auditable way, we follow a **GitOps-based deployment model** using **ArgoCD** [1]. In this model, also described in [24], Git repositories act as the single source of truth and describe the desired state. ArgoCD then continuously reconciles against the live state of the cluster. Compared to imperative or manually triggered deployment approaches, this model enables version-controlled and reviewable configuration changes, improves reproducibility across clusters and environments, and simplifies rollbacks in case of errors. For our multi-cluster setup, GitOps provides a consistent and scalable way to manage configuration drift and coordinate changes across environments [26].

A number of additional components handle the **integration with OpenStack**. Rather than building custom solutions for load balancing, storage, or secret management, we reuse existing OpenStack services through dedicated Kubernetes integration components, reducing operational effort and benefiting from the fact that the same team maintains both Kubernetes and the underlying OpenStack infrastructure. The **OpenStack Cloud Controller Manager** [8, 20] automatically provisions **Octavia** [18] load balancers for Kubernetes services of type *LoadBalancer*, through which all external traffic is routed. CSI plugins for **Cinder** [7, 6] (block storage) and **Manila** [17, 16] (shared file systems) allow tenants to create *PersistentVolumeClaims* that automatically provision and attach the corresponding OpenStack volumes. En-

cryption of sensitive Kubernetes resources at rest in etcd is handled by integrating OpenStack's **Barbican** [2] via its KMS plugin [3].

Cluster lifecycle management is handled using **Cluster API** [9]. Cluster API provides a declarative, Kubernetes-native approach to provision and manage nodes in our clusters by communicating with the OpenStack clouds. The needed configuration is done directly via Kubernetes resources from within the cluster so that the cluster manages its own virtual machines in OpenStack and is able to create and delete nodes for updates or scaling as needed. This declarative and Kubernetes-native integration reduces external orchestration complexity compared to traditional VM provisioning workflows and allows lifecycle operations such as scaling, rolling updates, or cluster upgrades to be handled in a consistent and automated manner. The **Cluster API Provider OpenStack** [10] implements the cloud-specific provisioning logic, while the **Cluster Autoscaler** [11] integrates with Cluster API to dynamically adjust node counts according to workload demand.

4 Multi-Tenancy and Isolation

Providing namespaces safely requires a careful isolation strategy [22], as outlined in Section 2.

4.1 Role-based Access Control (RBAC) and the Aggregated Kubernetes Admin Role

The basis of user isolation is the **Role-Based Access Control** [21] built into Kubernetes. It allows creating roles in the cluster that are granted access to different resources and operations, such as get, delete, create, and so on. These roles can then be bound to a user or a set of users to grant them the permissions defined within the role, either on a cluster-wide or a namespace-wide level. This way, it is possible to grant access to a very specific set of resources and operations to users within their namespace by binding a predefined role to them within their own namespace.

Fortunately, Kubernetes has a predefined *admin* role that is intended to be bound within a namespace to grant read and write access to most resources, excluding resources that could be used to tamper with the system.

While the *admin* role has proper permissions for the base Kubernetes resources, for each component with custom resource definitions, the permissions potentially need to be extended to include access to the new custom resources. For this purpose, **aggregated roles** can be used to extend an already defined role with additional permissions. The process of deciding which permissions for custom resources should be granted to users is manual and must be done with great care and consideration, because every granted permission can represent a potential security risk.

Due to strict namespace-scoped RBAC, tenants cannot install cluster-wide resources such as operators or custom controllers. While this limits some advanced use cases, it ensures that all tenants remain fully isolated and cannot affect cluster-level operations, preserving security and stability across the platform.

4.2 Resource Quotas

Another aspect that needs to be considered during isolation is limiting the resources each user can use in their namespace. This includes resources such as CPU, memory, storage, and specialized hardware like vGPUs, which are either not granted at all or only in a limited amount, but also limits on Kubernetes resources to prevent flooding the central etcd server.

Kubernetes already includes a tool to achieve this restriction called **Resource Quotas**, which can be defined at the namespace level to restrict the number of the resources mentioned above. The only caveat is that, for the restriction to work, each container needs to define resource requests and limits so that Kubernetes can check whether the sum of these resources is below the quota. To force users to define this, we use customized **Gatekeeper** rules described in Section 4.3.

4.3 Gatekeeper Mutation and Restriction

The restrictions described in Section 4.1 and Section 4.2 are a good basis for isolating users within their namespace, but there are still ways to circumvent them. For example, if we want to grant access for users to *Certificate* resources from **Cert-Manager** [4], so that they can generate TLS certificates for their application from a central Certificate Authority, we need a way to also restrict the values for some keys in the resource, like *commonName* or *dnsNames*. Otherwise, they could create certificates not only for their own services but also for other services.

To address these limitations, we use **Gatekeeper** [13]. Gatekeeper extends Kubernetes with a validating and mutating admission controller that allows the definition and enforcement of custom policies written in Rego, a declarative language especially designed to write policies. These policies, written in **Rego**, are defined declaratively as *ConstraintTemplates* and *Constraints* and can validate or modify resources at creation time.

By using Gatekeeper, we can restrict specific keys within custom resources such as *Certificate*. For example, we enforce that the *commonName* and *dnsNames* fields must match predefined patterns associated with the respective namespace or tenant. This ensures that users can only request certificates for domains assigned to them and prevents misuse of the central Certificate Authority.

The policies are parameterizable, allowing us to define reusable templates and create namespace-specific constraints. This enables fine-grained control, as individual rules can be tailored per tenant while still following a common security model across the cluster.

Gatekeeper also provides a library with many predefined admission and mutation policies. These include a re-creation of the deprecated **Pod Security Policies** and we deploy them with reasonable defaults to control access to host filesystems, privilege escalation, capabilities, and many more. We maintain two different profiles for these policies: **default** and **hardened**, with different configurations. Both are secure by design, but the hardened profile also contains additional hardening features, like restricting the user UID or denying privilege escalation. These profiles can be set on namespaces, and each policy can be optionally disabled per label.

In addition, we implemented several tenant-related **custom Gatekeeper policies** restricting domain usage, load balancer exposure, node group access, and service mesh configuration. These policies complement default security profiles and enforce namespace-scoped isolation guarantees, preventing tenants from interfering with each other or with critical cluster resources.

4.4 Network Isolation with Cilium Network Policies

We are also using Cilium to enforce network policies for all tenants where by default almost all traffic is denied, with some small exceptions, like DNS, to ensure maximal security. This way, users need to explicitly allow ingress as well as egress traffic and are not able to reach workloads of other tenants without consent.

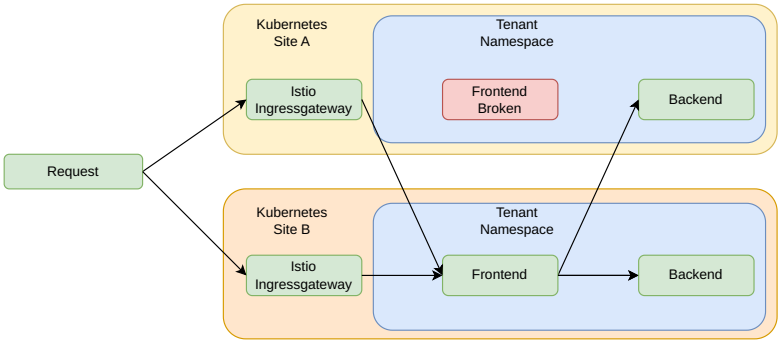

Figure 2: Request Flow

5 Multi-Cluster Management

To be able to host services across multiple clusters for higher availability and resilience, we leverage the multi-cluster features of Cilium and Istio, along with a custom DNS solution deployed across different locations. This setup allows us to seamlessly span services, including web applications and databases, across clusters, while also shaping traffic between them to prevent service interruptions.

A basic requirement for a multi-cluster deployment is that the pods need to be able to communicate with each other across different clusters. **Cilium's ClusterMesh** enables this communication and also expands the usage of network policies over all clusters. This way Pods in one cluster can still use all features like label-based allow and deny rules to secure their traffic even if the destination or source of this traffic is in another cluster. So we can ensure seamless communication while still being able to enforce our security.

After enabling pod-to-pod communication, **Istio's Service Mesh** can also be spread over all clusters, providing transparent mTLS for secure communication between Pods, as well as observability, traffic shaping, and routing. With these features, traffic can easily flow between the clusters and, for example, enter the cluster via the ingress gateway at one location to flow to a service at the other, increasing availability. Figure 2 shows how this can, for example, be used to circumvent a request being redirected to a broken frontend on the Kubernetes cluster on site A and instead be redirected only to the frontend on site B.

To facilitate service discovery, we also deploy a **custom DNS** solution capable of resolving multiple subdomains corresponding to different clusters. Each DNS service has access to all other clusters' information via their API servers and resolves services and pods. This allows Pods and Services to be addressed uniformly across clusters.

6 Operational Metrics

The presented Kubernetes platform has been operated in production since late 2021, initially serving internal services before onboarding external tenants. Since then, it has evolved into a stable infrastructure component within the university environment, now supporting a growing number of projects. This section quantitatively highlights selected aspects of the platform using representative metrics.

Figure 3 shows the growth of tenants and namespaces over time, increasing from 7 tenants

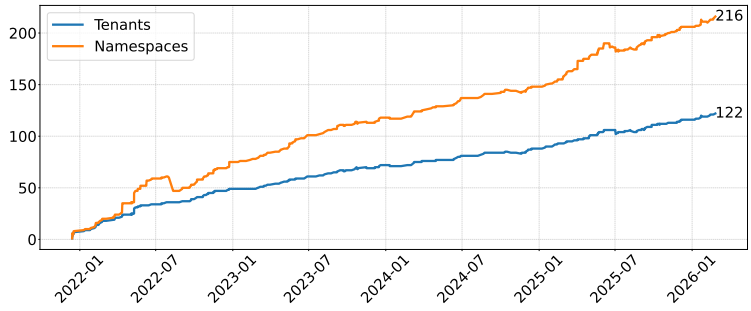

Figure 3: Growth of Tenants and Namespaces since 2021

in 2021 to 122 tenants in early 2026. This steady growth reflects both the demand for container-based infrastructure and the acceptance of the self-service namespace model.

Table 1 shows an overview of the current platform size, including some of the most important Kubernetes objects and the most frequently requested resources. Of course, due to the cluster's autoscaling capabilities, this is only a snapshot of the current size.

Table 1: Cluster Size and Resource Overview

Node	115
Pods	3421
PVCs	1024
Requested CPUs	1276
Requested GPUs	19
Requested Memory in GB	7330
Requested Ephemeral Storage in GB	2467
Requested Storage in TB	7250

Figure 4 shows the request throughput handled by the Istio mesh. This is also a good approximation for the complete inter-cluster traffic, since most of the traffic between the clusters goes through the service mesh, in contrast to cluster-internal traffic, where a large proportion, especially in the tenant namespaces, does not. Most of the traffic between the clusters is not failovers, but from real multi-cluster setups with load balancing capabilities and database replications between clusters.

For traffic in the clusters, we can take a look at Figure 5. We can see that the cluster has a mean traffic input of approximately 250 MB per second, distributed over the two locations via a simple DNS load balancer mechanism.

For the stability of the Kubernetes platform itself, we can look at Figure 6, which shows the rate of erroneous responses of the API Server, as well as the latency in Figure 7. Although API Server error rates and latency are not ideal, particularly as the number of tenants and workloads has increased, the platform continues to provide reliable, highly available services for

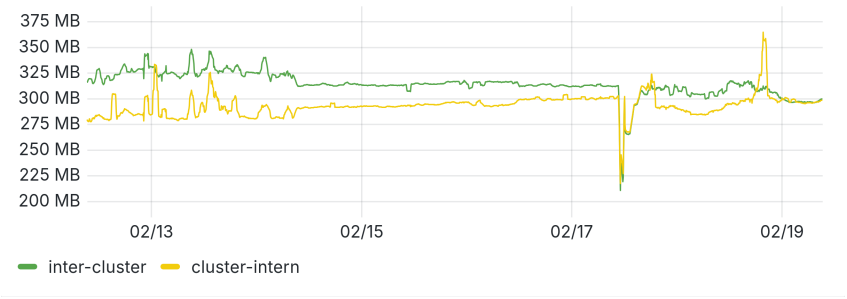

Figure 4: Service Mesh Traffic per Second over one week in February 2026 with dates in MM/DD format

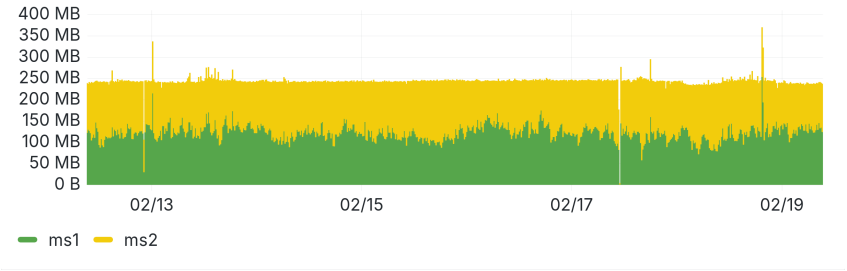

Figure 5: Istio Ingress Gateway Traffic per Second over one week in February 2026 with dates in MM/DD format (ms1 and ms2 refer to the two cluster locations)

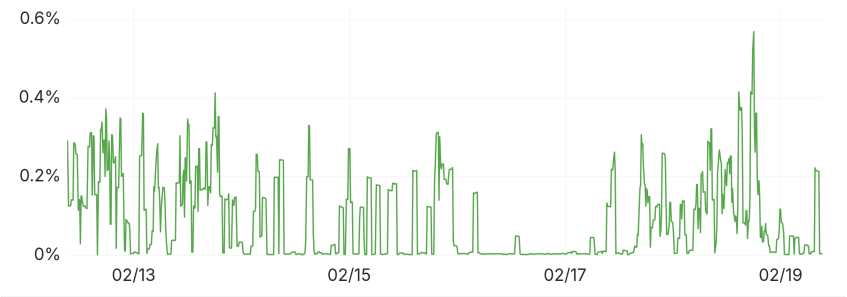

Figure 6: Kubernetes API Server Error Rate (5xx) over one week in February 2026 with dates in MM/DD format

most tenants. We are actively investigating optimizations in API Server scaling and request handling to improve these metrics. Despite these challenges, tenants are able to operate their services with stable performance, demonstrating that the multi-tenant platform remains robust in practice.

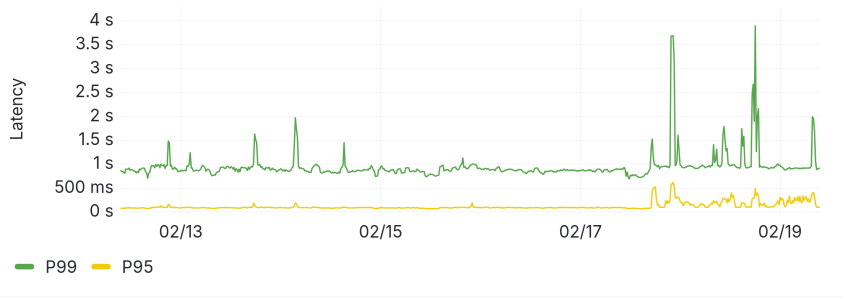

Figure 7: Kubernetes API Server Latency (P99 and P95) over one week in February 2026 with dates in MM/DD format

7 Conclusion and Outlook

The namespace-based multi-tenant model, combined with strict RBAC policies, default-deny networking, and admission control via Gatekeeper, has proven to be a viable alternative to per-tenant clusters in our context. While certain limitations exist, such as the inability for tenants to install cluster-wide operators, the model enables rapid onboarding and self-service for most workloads while maintaining strong security guarantees.

Users can deploy and manage applications in an isolated environment without worrying about infrastructure, while operators benefit from reduced overhead through centralized maintenance and consistent policy enforcement.

A key advantage of the shared-cluster model is that platform-level services are deployed once and benefit all tenants without individual configuration effort. This includes a scalable ingress gateway with load balancing, integration into the university's DNS infrastructure, automated TLS certificate provisioning, network-level firewalling through Cilium, automated security scanning, and a shared monitoring and observability stack. Tenants receive these integrations as part of their namespace, replacing tasks that would otherwise need to be implemented and maintained individually per VM or per cluster.

The steady growth to over 120 tenants is further reflected in several large-scale production deployments running on the platform. These include a central container registry hosting more than 120,000 artifacts (container images, Helm charts, and SBOMs), a monitoring stack processing approximately 1.2 million metrics, 100,000 log lines, and 20,000 traces per second, the university's on-premise LLM service **UniGPT** [23], as well as the state-wide JupyterHub platform **JupyterHub NRW** [15]. These deployments demonstrate that the platform reliably supports high-throughput, data-intensive, and user-facing services in a multi-tenant setup.

The multi-cluster approach increases architectural complexity but provides real benefits in terms of resilience, scalability, and distribution. By combining Cilium's ClusterMesh for cross-cluster networking with Istio's service mesh for traffic shaping and transparent mTLS, services can seamlessly span multiple locations and survive data center-level outages. With the ongoing integration of an additional site, these advantages are expected to become even more significant.

In retrospect, the most impactful design decision was adopting a fully declarative and image-based lifecycle using GitOps and Cluster API. This approach effectively prevents configuration drift and has been a key factor in the longevity and the long-term stability of the platform. Core components such as Cilium and Istio have proven robust under production load and have allowed us to implement strict isolation while still offering advanced networking and traffic management features.

Future work focuses on expanding the platform by onboarding additional tenants, increasing adoption across the university and beyond, and integrating a third site to further enhance resilience and distribution. Further development of platform features and automation will hopefully support this growth while maintaining security and stability.

Overall, the platform demonstrates that Namespace-as-a-Service, combined with a multi-cluster architecture, strong policy enforcement, and declarative operations, can successfully replace many traditional VM-based and per-cluster deployment models in a university-scale environment.

References

[1] Argo CD. URL: https://argo-cd.readthedocs.io/en/stable/.

[2] Barbican Documentation (Victoria). URL: https://docs.openstack.org/barbican/victoria/.

[3] Barbican KMS Plugin. URL: https://github.com/kubernetes/cloud-provider-openstack/blob/master/docs/barbican-kms-plugin/using-barbican-kms-plugin.md.

[4] cert-manager. URL: https://cert-manager.io/.

[5] Cilium. URL: https://cilium.io/.

[6] Cinder CSI Plugin. URL: https://github.com/kubernetes/cloud-provider-openstack/blob/master/docs/cinder-csi-plugin/using-cinder-csi-plugin.md.

[7] Cinder Documentation (Victoria). URL: https://docs.openstack.org/cinder/victoria/.

[8] Cloud Controller Manager. URL: https://kubernetes.io/docs/concepts/architecture/cloud-controller/.

[9] Cluster API. URL: https://cluster-api.sigs.k8s.io/.

[10] Cluster API Provider OpenStack. URL: https://cluster-api-openstack.sigs.k8s.io/.

[11] Cluster Autoscaler. URL: https://github.com/kubernetes/autoscaler/tree/master/cluster-autoscaler.

[12] etcd. URL: https://etcd.io/.

[13] Gatekeeper. URL: https://open-policy-agent.github.io/gatekeeper/website/.

[14] Istio. URL: https://istio.io/.

[15] JupyterHub.NRW. URL: https://www.jupyterhub.nrw/.

[16] Manila CSI Plugin. URL: https://github.com/kubernetes/cloud-provider-openstack/blob/master/docs/manila-csi-plugin/using-manila-csi-plugin.md.

[17] Manila Documentation (Victoria). URL: https://docs.openstack.org/manila/victoria/.

[18] Octavia Documentation (Victoria). URL: https://docs.openstack.org/octavia/victoria/.

[19] OpenStack. URL: https://www.openstack.org/.

[20] OpenStack Cloud Controller Manager. URL: https://github.com/kubernetes/cloud-provider-openstack/blob/master/docs/openstack-cloud-controller-manager/using-openstack-cloud-controller-manager.md.

[21] Using RBAC Authorization. URL: https://kubernetes.io/docs/reference/access-authn-authz/rbac/.

[22] Sharan Babu Paramasivam Murugesan . Deep Multi-Layer Isolation for Secure Kubernetes Multi-Tenancy in a Single Shared Cluster . *International Journal of Computer Applications* , 187(62):50–55, 2025. doi:10.5120/ijca2025926026.

[23] Jonathan Radas and Benjamin Risse and Raimund Vogl. Building UniGPT: A Customizable On-Premise LLM-Solution for Universities. In Raimund Vogl and Laurence Desnos and Jean-François Desnos and Spiros Bolis and Lazaros Merakos and Gill Ferrell and Effie Tsili and Manos Roumeliotis, editor, *Proceedings of EUNIS 2024 annual congress in Athens*, volume 105 of *EPiC*

Series in Computing, pages 108–116. EasyChair, 2025. URL: /publications/paper/CDHx, doi:{10.29007/jv1l}.

[24] Limoncelli, Thomas A. GitOps: a path to more self-service IT. *Communications of the ACM*, 61(9):38–42, 2018.

[25] Mikkelsen, Anders and Grønli, Tor-Morten and Kazman, Rick. Immutable Infrastructure Calls for Immutable Architecture: Deploying a Changeless Architecture in the Cloud. In *Proceedings of HICSS*, 2019.

[26] Raju Shrestha and Abdi Nur Ali Ali. Configuration Management in Kubernetes Environments: A GitOps Approach. In *Proceedings for the 2024 IEEE/ACM 17th International Conference on Utility and Cloud Computing UCC 2024*. IEEE, 2024.

[27] Shazibul Islam Shamim and Jonathan Alexander Gibson and Patrick Morrison and Akond Rahman. Benefits, Challenges, and Research Topics: A Multi-vocal Literature Review of Kubernetes, 2022. URL: https://arxiv.org/abs/2211.07032, arXiv:2211.07032.

[28] Steven Wong, Michael Gasch. Out of the Clouds onto the Ground: How to Make Kubernetes Production Grade Anywhere. URL: https://kubernetes.io/blog/2018/08/03/out-of-the-clouds-onto-the-ground-how-to-make-kubernetes-production-grade-anywhere/.

[29] The Cilium Authors. Egress Gateway. URL: https://cilium.io/use-cases/egress-gateway/.

8 Biographies

S. Haardiek is a research assistant at the IT Department of the University of Münster (CIT) and cloud architect of the Uni Cloud. He graduated from University of Osnabrück (Germany) in 2015 with degrees in both mathematics and computer sciences. His research focuses on private clouds and cloud native technology.

M. Blank-Burian is a research assistant at the IT Department of the University of Münster (CIT) and lead cloud architect of the Uni Cloud. He graduated from University of Münster (Germany) in 2013 with degrees in both physics and computer sciences. In 2018, he received his Ph.D. in theoretical physics. His research focuses on private clouds and cloud native technology. More info: https://www.uni-muenster.de/forschungaz/person/17330

R. Vogl holds a Ph.D. in elementary particle physics from the University of Innsbruck (Austria). After completing his Ph.D. studies in 1995, he joined Innsbruck University Hospital as IT manager for medical image data solutions and moved on to be deputy head of IT. He served as a lecturer in medical informatics at UMIT (Hall, Austria) and as managing director for a medical image data management software company (icoserve, Innsbruck) and for a center of excellence in medical informatics (HITT, Innsbruck). Since 2007 he has been director of the IT Department of the University of Münster (CIT, Germany). His research interests focus on management of complex information systems and information infrastructures.