Technische Hochschule Ingolstadt

Specialist area Computer Science
Bachelor’s course Computer Science

Bachelor’s thesis

Subject: Conception, Implementation, and Evaluation of a Highly Scalable and Highly Available Kubernetes-Based SaaS Platform on Kubernetes Control Plane (KCP)

Name and Surname: David Linhardt

Matriculation number: 00122706

Issued on: 2025–04–09

Submitted on: 2025–09–09

First examiner: Prof. Dr. Bernd Hafenrichter

Second examiner: Prof. Dr. Ludwig Lausser

Abstract

Contents

List of Figures

List of Tables

1 Introduction

1.1 Problem Statement and Motivation

1.2 Objectives and Scope

1.3 Structure of the Thesis

2 Fundamentals

2.1 Kubernetes and Multi-Tenancy

2.1.1 Kubernetes as the Foundation for Cloud-Native Applications

As the de facto standard for deploying and managing *cloud-native applications*, Kubernetes, commonly referred to as Kubernetes () plays a pivotal role in modern cloud architecture (Poulton and Joglekar 2021, p. 7–8). Kubernetes works as an orchestrator for *containerized, cloud-native microservice* applications, meaning it can deploy apps and dynamically respond to changes (Poulton and Joglekar 2021, p. 3). It offers a platform for declarative configuration and automation for containerized workloads, enabling organizations to run distributed applications and services at scale (Kubernetes 2024; Red Hat 2024).

2.1.2 The Importance of Multi-Tenancy in Modern SaaS Platforms

Multi-tenancy plays a fundamental role in modern cloud computing. By allowing multiple tenants to share the same infrastructure through virtualization, it significantly increases resource utilization, reduces operational costs, and enables essential features such as VM mobility and dynamic resource allocation (AlJahdali et al. 2014, pp. 345–346). These benefits are crucial for cloud providers, as they make the cloud business model economically viable and scalable. In the context of modern Software as a Service () platforms, multi-tenancy goes even further by enabling unified management, frictionless onboarding, and simplified operational processes that allow providers to add new tenants without introducing incremental complexity or cost (AWS 2022, pp. 9–11).
However, while multi-tenancy is indispensable for achieving efficiency, scalability, and cost-effectiveness, it simultaneously introduces complex security challenges, especially in shared environments where resource isolation is limited. In particular, the potential for cross-tenant access and side-channel attacks makes security in multi-tenant environments a primary concern (AlJahdali et al. 2014, pp. 345–346). As such, understanding and addressing multi-tenancy from

both operational and security perspectives is essential when designing and securing modern cloud-native platforms (AWS 2022, pp. 9–11; *Information technology - Cloud computing - Part 2: Concepts* 2023, p. 4).

2.1.3 The Challenges of Multi-Tenancy and the Need for Solutions

Multi-tenancy introduces a spectrum of technical and security challenges that need to be addressed.

[1]: *Residual-data exposure*. Shared infrastructures may expose tenants to data leakage and hardware-layer attacks. Because hardware resources are only virtually partitioned, residual data left in reusable memory or storage blocks, known as *data remanence*, can be inadvertently leaked or deliberately harvested by co-resident tenants (Zissis and Lekkas 2012, p. 586; AlJahdali et al. 2014, pp. 344–345).

[2]: *Control and transparency*. By design, SaaS moves both data storage and security controls out of the enterprise's boundary and into the provider's multi-tenant cloud, depriving organizations of direct oversight and assurance and thereby heightening concern over how their critical information is protected, replicated and kept available (Subashini and Kavitha 2011, pp. 3–4). To complicate matters further, the customer might have no way to evaluate the SaaS vendors security measures, meaning the pricing and feature set will most likely determine which service is used in practice, often disregarding security concerns (Everett 2009, p. 6; Khorshed, Ali, and Wasimi 2012, p. 836).

[3]: *Scheduling*. In multi-tenant architectures multiple tenants utilize the same hardware, thus creating the need for fair scheduling to ensure cost-effectiveness and performance (Simić et al. 2024, p. 32597). Achieving fair and efficient resource allocation in scheduling first requires a quantitative assessment of the system's existing unfairness (Ebrahimi et al. 2012, p. 7; Beltre, Saha, and Govindaraju 2019, p. 14; Ghodsi et al. 2011, pp. 2–3). Various scheduling algorithms and policies can be employed in practice to achieve fairness (Beltre, Saha, and Govindaraju 2019, pp. 14–16; Ghodsi et al. 2011, p. 4). To fully leverage the advantages of multi-tenant architectures, resources must not only be shared fairly, but also efficiently, not hindering performance (Beltre, Saha, and Govindaraju 2019, p. 14). As stated by Beltre, Saha, and Govindaraju 2019, p. 14 “Balancing both cluster utilization and fairness is challenging”.

[4]: *Performance Isolation*. A single tenant is able to significantly degrade the performance of

other tenants working on the same hardware, if *performance isolation* is not given (Krebs and Mehta 2013, p. 195). The fundamental performance expectations of a system are commonly formalized in a Service Level Agreement (). As noted by Krebs and Mehta 2013, p. 195 “A system is said to be performance isolated, if for tenants working within their quotas the performance is within the (response time) SLA while other tenants exceed their quotas (e.g., request rate)”. As noted by Carrión 2022, p. 18 “Currently, it is difficult to achieve performance isolation for multi-tenancy on K8s clusters because this requires providing resource isolation and improving the abstraction of applications.”

[5]: *Automation*. As noted by Nguyen and Y. Kim 2022, p. 651 “Presently, multi-tenant systems lack the facility of allowing clients to dynamic *[sic]* change their resources based on their business demands or create and allocate resources for new tenants. Multi-tenant system *[sic]* administrator manually does all the work of allocation or changing tenant’s *[sic]* resources.” However, to ensure efficiency and scalability, an Application Programming Interface (API) that allows automating deployments and dynamic changes in the application is needed.

A secure solution, keeping multi-tenancies advantages while also addressing security concerns is desperately needed (AlJahdali et al. 2014, p. 346; Şenel et al. 2023, pp. 14576–14577).

2.1.4 Kubernetes Control Plane (KCP) as a Promising Approach

Kubernetes Control Plane () offers three capabilities that map cleanly onto today’s multi-tenancy pain points.

[1]: *Workspaces*. KCP achieves strong resource isolation through the concept of *workspaces* (see subsubsection 2.2.1: *Workspaces*).

[2]: *API* KCP offers an Kubernetes-like API that enables the use of standard tools and automation to a degree (see subsubsection 2.2.2: *API*).

[3]: *Sharding* KCP offers sharding out of the box to manage high traffic and geo-distribution (see subsubsection 2.2.3: *Sharding*).

2.1.5 Background: The Evolution of Kubernetes

Kubernetes, an open-source container orchestration platform developed by Google, emerged from the need to manage the complexities of containerized applications effectively and to support large-scale deployments in a cloud-native environment (Google Cloud 2025; Kubernetes 2024). It was originally developed at Google and released as open source in 2014 (Google Cloud 2025). Kubernetes was conceived as a successor to Google's internal container management system called Borg, and designed to streamline the process of deploying, scaling, and managing applications composed of microservices running in containers (Verma et al. 2015, pp. 13–14; Bernstein 2014, p. 84). Since its inception, Kubernetes has gained traction among organizations because it provides robust features such as automated scaling, self-healing, and service discovery, which have made it the de facto standard for container orchestration in the tech industry (Damarapati 2025, pp. 855–858).

As noted by Moravcik et al. 2022, p. 457 by 2021 almost 90% of organizations used Kubernetes as an orchestrator for managing containers and over 70% of organizations used it in production (Shamim Choudhury 2025). The widespread adoption of Kubernetes is further underscored by Red Hat's latest (2024) report, which no longer asks survey respondents if they use Kubernetes for container orchestration, but rather **which** Kubernetes platform they use (Red Hat, Inc. 2024, p. 27). According to Damarapati 2025, pp. 855–856, Kubernetes has seen unprecedented industry adoption due to its vendor neutrality, strong community support, and flexible, extensible architecture in combination with enterprise readiness caused by high availability, disaster recovery and security.

Moreover Kubernetes enables faster time-to-market by providing a unified, declarative control plane that abstracts away infrastructure, guarantees consistent environments from development to production, and automates operational tasks such as scaling, rolling updates, and self-healing—advantages that translate directly into competitive delivery speed increasing its appeal to organizations of every size (Damarapati 2025, pp. 858–859).

Over the years, Kubernetes—and the many orchestration solutions inspired by or built on it—has evolved to handle an increasingly diverse range of workloads, supporting everything from conventional applications to emerging *edge-native* deployments (Biot et al. 2025, p. 21; Biot et al. 2025, pp. 1–4). Edge-native deployments are applications intended to run on computing resources located at or near the data source — the network *edge* — rather than in a central cloud (Satyanarayanan et al. 2019, p. 34). This adaptability reflects its fundamental design, which focuses on modularity and extensibility, allowing developers to customize their orchestration needs.

Overall, the history of Kubernetes showcases a transformative journey driven by the evolving demands of software architecture and the necessity for efficient application management in an increasingly complex technological landscape.

2.1.6 Background: Containerization as an Enabler of Kubernetes

Containerization is a way to bundle an application's code with all its dependencies to run on any infrastructure thus enhancing portability (AWS 2025; Docker 2025). The lightweight nature and isolation can be leveraged by cloud-native software by enabling vertical and horizontal autoscaling facilitated by quick container boot times, along with self-healing mechanisms and support for distributed, resilient infrastructures (Kubernetes 2025b; Kubernetes 2025d; AWS 2025; Davis 2019, pp. 58–59). Furthermore it complements the microservice architectural pattern by enabling isolated, low overhead deployments, ensuring consistent environments (Balalaie, Heydarnoori, and Jamshidi 2016, p. 209).

2.1.7 Background: The Role of Microservices in Cloud-Native Architectures

Microservices play a pivotal role in cloud-native architectures by promoting agility, scalability, and maintainability of applications. By decomposing applications into independent, granular services, microservices facilitate development, testing, and deployment using diverse technology stacks, enhancing interoperability across platforms (Waseem, Liang, and Shahin 2020, p. 1; Larrucea et al. 2018, p. 1) and help prevent failures in one component from propagating across the system, by isolating functionality into distinct, self-contained services (Davis 2019, p. 62). This architectural style aligns well with cloud environments, as it allows services to evolve independently, effectively addressing challenges associated with scaling and maintenance without being tied to a singular technological framework (Balalaie, Heydarnoori, and Jamshidi 2016, pp. 202–203). Furthermore, the integration of microservices with platforms like Kubernetes enhances deployment automation and orchestration, thus providing substantial elasticity to accommodate fluctuating workloads (Haugeland et al. 2021, p. 170). Additionally, migrating legacy applications to microservices can foster modernization and efficiency, thus positioning organizations favorably in competitive landscapes (Balalaie, Heydarnoori, and Jamshidi 2016, p. 214). Overall, the synergy between microservices and cloud-native architectures stems from their inherent capability to optimize resource utilization and streamline continuous integration and deployment processes.

2.1.8 Background: Kubernetes Resource Isolation Mechanisms

Kubernetes employs several resource isolation mechanisms, primarily through the use of *control groups ()* (control groups) and *namespaces* to limit resource allocation for containers. Cgroups are a Linux kernel feature that organizes processes into hierarchical groups for fine-grained resource limitation and monitoring via a pseudo-filesystem called *cgroupfs* (Kubernetes 2025a; Project 2024). *Namespaces* are a mechanism for isolating groups of resources withing a single cluster and scoping resource names to prevent naming conflicts across different teams or projects (Kubernetes 2025e). However, these mechanisms may not always provide sufficient isolation necessary for multi-tenant architectures, because the logical segregation offered by namespaces does not address the fundamental security concerns associated with multi-tenancy (Nguyen and Y. Kim 2022, p. 651) and research indicates that the native isolation strategies can lead to performance interference, where containers that share nodes can experience significant degradation in performance due to CPU contention (E. Kim, Lee, and Yoo 2021, p. 158). Specifically, critical services may be adversely affected when non-critical services monopolize available resources, which undermines the quality of service in multi-tenant environments (Li et al. 2019, p. 30410).

Moreover, while Kubernetes allows for container orchestration and resource scheduling, it can lead to resource fragmentation, further exacerbating the issue of performance isolation (Jian et al. 2023, p. 1). A common approach in multi-tenant scenarios is to deploy separate clusters for each tenant, which incurs substantial overhead—particularly in environments utilizing virtual machines for isolation (Şenel et al. 2023, pp. 144574–144575). In summary, although Kubernetes offers essential isolation mechanisms, the complexities of resource sharing and performance consistency in multi-tenant applications highlight the need for enhanced strategies to ensure robust resource management and performance isolation (Nguyen and Y. Kim 2022, p. 651; Jian et al. 2023, p. 2; E. Kim, Lee, and Yoo 2021, p. 158).

2.1.9 Relevance to SaaS and this Thesis

2.2 Kubernetes Control Plane (KCP)

KCP is “An open source horizontally scalable control plane for Kubernetes-like APIs” (The kcp Authors 2025).

2.2.1 Workspaces

KCP introduces the concept of *workspaces* to implement multi-tenancy. In KCP, a workspace is a Kubernetes-cluster-like HTTPS endpoint exposed under `/clusters/<parent>:<name>`, that regular tools such as *kubectl*, *Helm* or *client-go* treat exactly like a real Kubernetes cluster. Every workspace is backed by its own logical cluster stored in an isolated **etcd prefix**, so objects in one workspace (including cluster-scoped resources like **CRDs**) are completely invisible to others, delivering hard multi-tenancy without spinning up separate control planes (kcp Docs 2025l).
As per the definition by the etcd Authors 2025, “**etcd** is a strongly consistent, distributed key-value store that provides a reliable way to store data that needs to be accessed by a distributed system or cluster of machines. It gracefully handles leader elections during network partitions and can tolerate machine failure, even in the leader node.” etcd is the primary datastore used by KCP and K8s (kcp Docs 2025k; Sun et al. 2021, p. 214). **etcd prefixes** are a simple, inbuilt way to group keys using a prefix (etcd Docs 2025). This allows for resource isolation in KCP.
A Custom Resource Definition (CRD) is a declaratively specified schema that registers a new resource, defined by its group, version, kind, and OpenAPI schema, into the Kubernetes Control Plane so the native API server stores and serves the objects as first-class resources (Kubernetes 2025c).
KCP implements the same Role-Based Access Control ()-based authorization mechanism and cluster role and cluster role binding principles as Kubernetes inside workspaces (kcp Docs 2025c). However unlike Kubernetes KCP does currently not support Attribute-Based Access Control () (kcp Docs 2025c; Kubernetes 2025f). KCP likely supports only RBAC-based authorization because ABAC is considered overly complex, hard to audit, and increasingly deprecated in favor of RBAC, which offers a more structured and maintainable access control model (Cullen 2025). This allows for consistent access control and permission management across all workspaces, aligning with familiar Kubernetes patterns and simplifying multi-tenant environment administration.
Workspaces for a typed, parent-child tree, and each type can constrain which kind of workspaces it can contain or be contained by, giving platform teams a structured way to delegate environments while retaining policy control (kcp Docs 2025l).
This combination of strong isolation, familiar tooling, and hierarchical organization makes

workspaces offer an attractive solution to many of the problems commonly faced in multi-tenant environments: each tenant gets the freedom of a full dedicated cluster, yet operators manage only a single shared KCP control plane.

2.2.2 API

As previously noted, most Kubernetes based multi-tenant systems currently require manual intervention by an administrator to deploy new tenants or modify resource allocation. However KCP, other than similar frameworks, like Capsule or Kiosk, provides an API server to the customer, that provides an easy way to access their resources (Nguyen and Y. Kim 2022, p. 651; The kcp Authors 2025). This in turn enables a degree of automation (Nguyen and Y. Kim 2022, p. 651). Every workspace has its own API endpoint (kcp Docs 2025l). This ensures, that more control can be shifted back to the customer. KCP ships a curated set of built-in Kubernetes APIs, such as `Namespaces`, `ConfigMaps`, `Secrets` and RBAC objects, so tenants can start working with familiar primitives immediately (kcp Docs 2025d). Additional functionality can be added per workspace simply by installing a CRD, and KCP permits multiple independent versions of the same CRD to coexist across workspaces (kcp Docs 2025b).
To share an API with other tenants, a provider declares an `APIResourceSchema` and then exports it through an `APIExport`, while consumers attach that API to their workspace with an `APIBinding` (kcp Docs 2025f). This export/bind workflow lets platform teams evolve APIs centrally without touching each consumer workspace, reinforcing the system's self-service goal (kcp Docs 2025f).
KCP supports admission webhooks only through Uniform Resource Locator (URL)-based client configurations, `service`-based webhooks and conversion webhooks are currently unsupported, so operators must host their hooks externally (kcp Docs 2025a).
Because admission requests include the logical-cluster name, webhook back-ends can enforce policies per workspace and thus maintain strong tenant isolation (kcp Docs 2025c). Every Representational State Transfer (REST) call is scoped under the path pattern /clusters/<workspace>, ensuring that automation never leaks objects across workspaces (kcp Docs 2025h). API providers can also access consumer data through a virtual-workspace URL rooted at `/services/apiexport/` enabling safe cross-workspace reconciliation loops without cluster-wide privileges (kcp Docs 2025h). Together, built-in APIs, CRDs, the APIExport / APIBinding model, admission controls and workspace-prefixed REST routing give tenants a rich yet safe surface for automation while keeping operational responsibility with the platform team.

2.2.3 Sharding

KCP employs sharding to **horizontally scale** the control-plane, letting an installation grow far beyond the limits of a single API-server/etcd pair (kcp Docs 2025i; kcp Docs 2025j).
Each shard hosts a set of logical clusters, so every workspace (and therefore every tenant) gets its own Kubernetes-compatible consistency domain on that shard (kcp Docs 2025j). Because “a set of known shards comprises a KCP installation”, operators can add or remove shards at will, expanding or contracting capacity with no downtime for existing workspaces (kcp Docs 2025j).
Cross-shard traffic is funneled through a dedicated **cache-server**, avoiding the $n \times (n-1)$ explosion of of direct links that would otherwise appear in large deployments (kcp Docs 2025e).
This cache-server also underpins **workspace migration and object replication**, so tenants remain oblivious to topology changes while the platform evolves underneath them (kcp Docs 2025e).
Administrators can define **Partitions** that group shards, for example by region or load profile, giving schedulers a topology-aware API for controller placement (kcp Docs 2025g). Partitions therefore deliver **geo-proximity, load distribution, and fault isolation** for multi-tenant control-plane components (kcp Docs 2025g). Taken together, sharding provides the scalability, noisy-neighbor isolation, and topology flexibility required to run **large numbers of independent workspaces in a single multi-tenant KCP deployment**.

2.2.4 High Availability

2.3 SaaS Architecture and Automation

SaaS is, above all else, a **business and software-delivery model** in which a provider offers its solution through a low-friction, service-centric model that maximizes value for customers and providers surrounding all tenant environments with a single, unified experience (AWS 2022, pp. 3–4; AWS 2022, p. 11). According to AWS 2022, pp. 3-4, SaaS is associated with six major objectives:

[1]: *Agility*. SaaS companies prosper by designing for continuous adaptation to evolving markets, customer demands, competitive pressures, pricing models, and target segments.

[2]: *Operational efficiency* SaaS companies grow and scale by fostering a culture of **operational efficiency** that unifies tooling, enables rapid collective deployment across all customer environments, and eliminates one-off customizations.

[3]: *Frictionless onboarding* SaaS providers must minimize friction in onboarding every Business to Business (B2B) and Business to Customer (B2C) tenant by creating repeatable, efficient processes that accelerate time-to-value.

[4]: *Innovation* SaaS providers build a flexible foundation that lets them respond to current customer needs while using that same agility to innovate and unlock new markets, opportunities, and efficiencies.

[5]: *Market response* SaaS replaces long-cycle releases with near-real-time agility, enabling organizations to pivot strategy in response to emerging market dynamics.

[6]: *Growth* SaaS is a growth-oriented model that aligns organizational agility and efficiency to welcome rapid adoption of the offering.

Automation is the foundation for utilizing the scaling effects that come with SaaS architectures. The onboarding service automatically orchestrates other services to create users, tenant, isolation policies, provision and per-tenant resources (AWS 2022, p. 14). Once live, automated pipelines let new features roll to every tenant through a single, shared process, giving operators a single pane of glass for the whole estate (AWS 2022, p. 10). However merely automating the provisioning of each customer environment and offloading its management to an Managed Service Provider () still leaves tenants running potentially different, separately-operated versions and distances the software provider from unified onboarding, operations, and customer insight—so automation alone creates an MSP setup, whereas true SaaS requires one shared version and a single, provider-owned control plane for every tenant (AWS 2022, pp. 23–24). Ultimately, SaaS depends on automated, repeatable workflows that remove internal and external friction and ensure stability, efficiency and repeatability for this process (AWS 2022, p. 14).

3 State of the Art and Related Work

3.1 Zero-Downtime Deployment Strategies

3.2 Kubernetes Scaling Methods

3.3 Multi-Tenancy Concepts in the Cloud

3.4 KCP

4 Conceptual Design

4.1 Proposed Scenario

4.2 System Requirements

4.2.1 Functional Requirements

FR ID	Title	Description
FR1	Test	TestDescription

Table 1: Functional Requirements

4.2.2 Non Functional Requirements

NFR ID	Title	Description
NFR1	Test	TestDescription

Table 2: Non Functional Requirements

4.3 Architecture Design with KCP for SaaS

4.4 Automated Deployment Strategies

5 Prototypical Implementation

5.1 Infrastructure with KCP

5.2 Tenant Provisioning (Automation, Multi-Tenancy)

5.3 Scaling Mechanisms (Horizontal Pod Autoscaler)

5.4 Monitoring and Logging (Prometheus, Grafana)

6 Evaluation

6.1 Performance Measurements (Downtime, Latency, Scaling)

6.2 Scaling Scenarios and Optimizations

6.3 Discussion of Results

7 Conclusion and Outlook

7.1 Summary

7.2 Personal Conclusion

7.3 Future Outlook

References

AlJahdali, Hussain, Abdulaziz Albatli, Peter Garraghan, Paul Townend, Lydia Lau, and Jie Xu (2014). “Multi-tenancy in Cloud Computing”. In: *2014 IEEE 8th International Symposium on Service Oriented System Engineering*, pp. 344–351. DOI: 10.1109/SOSE.2014.50.

AWS (2022). *AWS Whitepaper - SaaS Architecture Fundamentals*. AWS. URL: https://docs.aws.amazon.com/whitepapers/latest/saas-architecture-fundamentals/re-defining-multi-tenancy.html (visited on 05/01/2025).

AWS (2025). *What is Containerization?* URL: https://aws.amazon.com/what-is/containerization/ (visited on 05/01/2025). Archived at https://web.archive.org/web/20250519121803/https://aws.amazon.com/what-is/containerization/.

Balalaie, A., A. Heydarnoori, and P. Jamshidi (2016). “Migrating to cloud-native architectures using microservices: an experience report”. In: pp. 201–215. DOI: 10.1007/978-3-319-33313-7_15.

Beltre, Angel, Pankaj Saha, and Madhusudhan Govindaraju (Aug. 2019). “KubeSphere: An Approach to Multi-Tenant Fair Scheduling for Kubernetes Clusters”. In: *2019 IEEE Cloud Summit*. IEEE, pp. 14–20. DOI: 10.1109/cloudsummit47114.2019.00009. URL: http://dx.doi.org/10.1109/cloudsummit47114.2019.00009.

Bernstein, David (2014). “Containers and Cloud: From LXC to Docker to Kubernetes”. In: *IEEE Cloud Computing* 1.3, pp. 81–84. DOI: 10.1109/MCC.2014.51.

Biot, F., A. Fornés-Leal, R. Vaño, R. Simon, I. Lacalle, C. Guardiola, and C. Palau (2025). “A novel orchestrator architecture for deploying virtualized services in next-generation iot computing ecosystems”. In: *Sensors* 25 (3), p. 718. DOI: 10.3390/s25030718.

Carrión, Carmen (Dec. 2022). “Kubernetes Scheduling: Taxonomy, Ongoing Issues and Challenges”. In: *ACM Comput. Surv.* 55.7. ISSN: 0360-0300. DOI: 10.1145/3539606. URL: https://doi.org/10.1145/3539606.

Cullen, Jacob Simpson; Greg Castle; CJ (2025). *RBAC Support in Kubernetes*. URL: https://kubernetes.io/blog/2017/04/rbac-support-in-kubernetes/ (visited on 05/28/2025). Archived at https://web.archive.org/web/20250528173123/https://kubernetes.io/blog/2017/04/rbac-support-in-kubernetes/.

Damarapati, Abhinav (Jan. 2025). “Containers vs. Virtual machines: Understanding the shift to Kubernetes”. In: *World Journal of Advanced Engineering Technology and Sciences* 15.1, pp. 852–861. ISSN: 2582-8266. DOI: 10.30574/wjaets.2025.15.1.0305. URL: http://dx.doi.org/10.30574/wjaets.2025.15.1.0305.

Davis, Cornelia (2019). *Cloud Native Patterns - Designing change-tolerant software*. Shelter Island, NY: Manning. ISBN: 9781617294297.

Docker (2025). *Use containers to Build, Share and Run your applications*. URL: https://www.docker.com/resources/what-container/ (visited on 05/01/2025). Archived at https://web.archive.org/web/20250519121103/https://www.docker.com/resources/what-container/.

Ebrahimi, Eiman, Chang Joo Lee, Onur Mutlu, and Yale N. Patt (Apr. 2012). "Fairness via Source Throttling: A Configurable and High-Performance Fairness Substrate for Multicore Memory Systems". In: *ACM Transactions on Computer Systems* 30.2, pp. 1–35. ISSN: 1557-7333. DOI: 10.1145/2166879.2166881. URL: http://dx.doi.org/10.1145/2166879.2166881.

etcd Authors (2025). *etcd - A distributed, reliable key-value store for the most critical data of a distributed system*. URL: https://etcd.io/ (visited on 05/27/2025). Archived at https://web.archive.org/web/20250527201500/https://etcd.io/.

etcd Docs (2025). *How to get keys by prefix*. URL: https://etcd.io/docs/v3.5/tutorials/how-to-get-key-by-prefix/ (visited on 05/27/2025). Archived at https://web.archive.org/web/20250527200935/https://etcd.io/docs/v3.5/tutorials/how-to-get-key-by-prefix/.

Everett, Catherine (June 2009). "Cloud computing - A question of trust". In: *Computer Fraud & Security* 2009.6, pp. 5–7. ISSN: 1361-3723. DOI: 10.1016/s1361-3723(09)70071-5. URL: http://dx.doi.org/10.1016/s1361-3723(09)70071-5.

Ghodsi, Ali, Matei Zaharia, Benjamin Hindman, Andy Konwinski, Scott Shenker, and Ion Stoica (2011). "Dominant resource fairness: fair allocation of multiple resource types". In: *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*. NSDI'11. Boston, MA: USENIX Association, pp. 323–336.

Google Cloud (2025). *What is Kubernetes?* URL: https://cloud.google.com/learn/what-is-kubernetes (visited on 05/01/2025). Archived at https://web.archive.org/web/20250519121940/https://cloud.google.com/learn/what-is-kubernetes.

Haugeland, S., P. Nguyen, H. Song, and F. Chauvel (2021). "Migrating monoliths to microservices-based customizable multi-tenant cloud-native apps". In: pp. 170–177. DOI: 10.1109/seaa53835.2021.00030.

Information technology - Cloud computing - Part 2: Concepts (2023). ISO/IEC 22123-2:2023(E). Standard. ISO, p. 42.

Jian, Z., X. Xie, Y. Fang, Y. Jiang, T. Li, and Y. Lu (2023). "Drs: a deep reinforcement learning enhanced kubernetes scheduler for microservice-based system". In: DOI: 10.22541/au.167285897.72278925/v1.

kcp Docs (2025a). *Admission Webhooks*. URL: https://docs.kcp.io/kcp/main/concepts/apis/admission-webhooks/ (visited on 06/01/2025). Archived at https://web.a rchive.org/web/20250601195914/https://docs.kcp.io/kcp/main/concepts/apis/admission-webhooks/.

kcp Docs (2025b). *APIs in kcp*. URL: https://docs.kcp.io/kcp/main/concepts/apis/ (visited on 06/01/2025). Archived at https://web.archive.org/web/20250601195711/https://docs.kcp.io/kcp/main/concepts/apis/.

kcp Docs (2025c). *Authorization*. URL: https://docs.kcp.io/kcp/main/concepts/authorization/ (visited on 05/28/2025). Archived at https://web.archive.org/web/20250528171015/https://docs.kcp.io/kcp/main/concepts/authorization/.

kcp Docs (2025d). *Built-in APIs*. URL: https://docs.kcp.io/kcp/main/concepts/apis/built-in/ (visited on 06/01/2025). Archived at https://web.archive.org/web/20250528061940/https://docs.kcp.io/kcp/main/concepts/apis/built-in/.

kcp Docs (2025e). *Cache Server*. URL: https://docs.kcp.io/kcp/main/concepts/sharding/cache-server/ (visited on 06/01/2025). Archived at https://web.archive.org/web/20250601192839/https://docs.kcp.io/kcp/main/concepts/sharding/cache-server/.

kcp Docs (2025f). *Exporting and Binding APIs*. URL: https://docs.kcp.io/kcp/main/concepts/apis/exporting-apis/ (visited on 06/01/2025). Archived at https://web.archive.org/web/20250601200220/https://docs.kcp.io/kcp/main/concepts/apis/exporting-apis/.

kcp Docs (2025g). *Partitions*. URL: https://docs.kcp.io/kcp/main/concepts/sharding/partitions/ (visited on 06/01/2025). Archived at https://web.archive.org/web/20250601192707/https://docs.kcp.io/kcp/main/concepts/sharding/partitions/.

kcp Docs (2025h). *REST Access Patterns*. URL: https://docs.kcp.io/kcp/main/concepts/apis/rest-access-patterns/ (visited on 06/01/2025). Archived at https://web.archive.org/web/20250601200521/https://docs.kcp.io/kcp/main/concepts/apis/rest-access-patterns/.

kcp Docs (2025i). *Sharding*. URL: https://docs.kcp.io/kcp/main/concepts/sharding/ (visited on 06/01/2025). Archived at https://web.archive.org/web/20250601192318/https://docs.kcp.io/kcp/main/concepts/sharding/.

kcp Docs (2025j). *Shards*. URL: https://docs.kcp.io/kcp/main/concepts/sharding/shards/ (visited on 06/01/2025). Archived at https://web.archive.org/web/20250601192002/https://docs.kcp.io/kcp/main/concepts/sharding/shards/.

kcp Docs (2025k). *Storage to rest patterns*. URL: https://docs.kcp.io/kcp/main/developers/storage-to-rest-patterns/ (visited on 05/27/2025). Archived at https://web.archive.org/web/20250527203146/https://docs.kcp.io/kcp/main/developers/storage-to-rest-patterns/.

kcp Docs (2025l). *Workspaces*. URL: https://docs.kcp.io/kcp/v0.27/concepts/workspaces/ (visited on 05/27/2025). Archived at https://web.archive.org/web/20250527195516/https://docs.kcp.io/kcp/v0.27/concepts/workspaces/.

Khorshed, Md. Tanzim, A.B.M. Shawkat Ali, and Saleh A. Wasimi (June 2012). “A survey on gaps, threat remediation challenges and some thoughts for proactive attack detection in cloud computing”. In: *Future Generation Computer Systems* 28.6, pp. 833–851. ISSN: 0167-739X. DOI: 10.1016/j.future.2012.01.006. URL: http://dx.doi.org/10.1016/j.future.2012.01.006.

Kim, Eunsook, Kyungwoon Lee, and Chuck Yoo (Jan. 2021). “On the Resource Management of Kubernetes”. In: *2021 International Conference on Information Networking (ICOIN)*. IEEE, pp. 154–158. DOI: 10.1109/icoin50884.2021.9333977. URL: http://dx.doi.org/10.1109/icoin50884.2021.9333977.

Krebs, Rouven and Arpit Mehta (Sept. 2013). “A Feedback Controlled Scheduler for Performance Isolation in Multi-Tenant Applications”. In: *2013 International Conference on Cloud and Green Computing*. IEEE, pp. 195–196. DOI: 10.1109/cgc.2013.36. URL: http://dx.doi.org/10.1109/cgc.2013.36.

Kubernetes (2024). *Concepts / Overview*. URL: https://kubernetes.io/docs/concepts/overview/ (visited on 05/01/2025). Archived at https://web.archive.org/web/20250519122217/https://kubernetes.io/docs/concepts/overview/.

Kubernetes (2025a). *About cgroup v2*. URL: https://kubernetes.io/docs/concepts/architecture/cgroups/ (visited on 05/02/2025). Archived at https://web.archive.org/web/20250519120201/https://kubernetes.io/docs/concepts/architecture/cgroups/.

Kubernetes (2025b). *Concepts / Workloads / Autoscaling Workloads*. URL: https://kubernetes.io/docs/concepts/workloads/autoscaling/ (visited on 05/01/2025). Archived at https://web.archive.org/web/20250519121534/https://kubernetes.io/docs/concepts/workloads/autoscaling/.

Kubernetes (2025c). *Custom Resources*. URL: https://kubernetes.io/docs/concepts/extend-kubernetes/api-extension/custom-resources/ (visited on 05/27/2025). Archived at https://web.archive.org/web/20250527205333/https://kubernetes.io/docs/concepts/extend-kubernetes/api-extension/custom-resources/.

Kubernetes (2025d). *Kubernetes Self-Healing*. URL: https://kubernetes.io/docs/concepts/architecture/self-healing/ (visited on 05/01/2025). Archived at https://we b.archive.org/web/20250519121258/https://kubernetes.io/docs/concepts/architecture/self-healing/.

Kubernetes (2025e). *Namespaces*. URL: https://kubernetes.io/docs/concepts/overview/working-with-objects/namespaces/ (visited on 05/02/2025). Archived at https://web.archive.org/web/20250519115910/https://kubernetes.io/docs/concepts/overview/working-with-objects/namespaces/.

Kubernetes (2025f). *Using ABAC Authorization*. URL: https://kubernetes.io/docs/reference/access-authn-authz/abac/ (visited on 05/28/2025). Archived at https://web.archive.org/web/20250528172622/https://kubernetes.io/docs/reference/access-authn-authz/abac/.

Larrucea, Xabier, Izaskun Santamaria, Ricardo Colomo-Palacios, and Christof Ebert (2018). "Microservices". In: *IEEE Software* 35.3, pp. 96–100. DOI: 10.1109/MS.2018.2141030.

Li, Y., J. Zhang, C. Jiang, J. Wan, and Z. Ren (2019). "Pine: optimizing performance isolation in container environments". In: *Ieee Access* 7, pp. 30410–30422. DOI: 10.1109/access.2019.2900451.

Moravcik, Marek, Martin Kontsek, Pavel Segec, and David Cymbalak (2022). "Kubernetes - evolution of virtualization". In: *2022 20th International Conference on Emerging eLearning Technologies and Applications (ICETA)*, pp. 454–459. DOI: 10.1109/ICETA57911.2022.9974681.

Nguyen, Nguyen Thanh and Younghan Kim (Oct. 2022). "A Design of Resource Allocation Structure for Multi-Tenant Services in Kubernetes Cluster". In: *2022 27th Asia Pacific Conference on Communications (APCC)*. IEEE, pp. 651–654. DOI: 10.1109/apcc55198.2022.9943782.

Poulton, Nigel and Pushkar Joglekar (2021). *The Kubernetes Book*. 2021 Edition. No ISBN provided. Independently published, p. 243.

Project, The Linux Documentation (2024). *cgroups(7): Linux control groups*. 6.10. Online; accessed 2025-05-02. Linux man-pages project. URL: https://man7.org/linux/man-pages/man7/cgroups.7.html.

Red Hat (2024). *What is Kubernetes?* URL: https://www.redhat.com/en/topics/containers/what-is-kubernetes (visited on 05/01/2025). Archived at https://web.archive.org/web/20250519122615/https://www.redhat.com/en/topics/containers/what-is-kubernetes.

Red Hat, Inc. (2024). *The State of Kubernetes Security Report: 2024 Edition*. Red Hat. URL: https://www.redhat.com/en/resources/kubernetes-adoption-security-market-trends-overview (visited on 05/19/2025).

Satyanarayanan, Mahadev, Guenter Klas, Marco Silva, and Simone Mangiante (July 2019). "The Seminal Role of Edge-Native Applications". In: *2019 IEEE International Conference* *on Edge Computing (EDGE)*. IEEE, pp. 33–40. DOI: 10.1109/edge.2019.00022. URL: http://dx.doi.org/10.1109/edge.2019.00022.

Şenel, B., M. Mouchet, J. Cappos, T. Friedman, O. Fourmaux, and R. McGeer (2023). "Multitenant containers as a service (caas) for clouds and edge clouds". In: *Ieee Access* 11, pp. 144574–144601. DOI: 10.1109/access.2023.3344486.

Shamim Choudhury (2025). *Kubernetes adoption, security, and market trends report 2021 - by RedHat*. URL: https://www.javelynn.com/cloud/kubernetes-adoption-security-and-market-trends-report-2021 (visited on 05/19/2025). Archived at https://web.archive.org/web/20250519115027/https://www.javelynn.com/cloud/kubernetes-adoption-security-and-market-trends-report-2021.

Simić, Miloš, Jovana Dedeić, Milan Stojkov, and Ivan Prokić (2024). "A Hierarchical Namespace Approach for Multi-Tenancy in Distributed Clouds". In: *IEEE Access* 12, pp. 32597–32617. ISSN: 2169-3536. DOI: 10.1109/access.2024.3369031. URL: http://dx.doi.org/10.1109/access.2024.3369031.

Subashini, S. and V. Kavitha (Jan. 2011). "A survey on security issues in service delivery models of cloud computing". In: *Journal of Network and Computer Applications* 34.1, pp. 1–11. ISSN: 1084-8045. DOI: 10.1016/j.jnca.2010.07.006. URL: http://dx.doi.org/10.1016/j.jnca.2010.07.006.

Sun, X., L. Suresh, A. Ganesan, R. Alagappan, M. Gasch, L. Tang, and T. Xu (2021). "Reasoning about modern datacenter infrastructures using partial histories". In: *Proceedings of the Workshop on Hot Topics in Operating Systems*, pp. 213–220. DOI: 10.1145/3458336.3465276.

The kcp Authors (2025). *Building Massively Multi-Tenant Platforms. - An open source horizontally scalable control plane for Kubernetes-like APIs.* URL: https://www.kcp.io/ (visited on 05/29/2025). Archived at https://web.archive.org/web/20250526195351/https://www.kcp.io/.

Verma, Abhishek, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes (2015). "Large-scale cluster management at Google with Borg". In: *Proceedings of the Tenth European Conference on Computer Systems*. EuroSys '15. Bordeaux, France: Association for Computing Machinery. ISBN: 9781450332385. DOI: 10.1145/2741948.2741964. URL: https://doi.org/10.1145/2741948.2741964.

Waseem, M., P. Liang, and M. Shahin (2020). "A systematic mapping study on microservices architecture in devops". In: *Journal of Systems and Software* 170, p. 110798. DOI: 10.1016/j.jss.2020.110798.

Zissis, Dimitrios and Dimitrios Lekkas (2012). "Addressing cloud computing security issues". In: *Future Generation Computer Systems* 28.3, pp. 583–592. ISSN: 0167-739X. DOI: https://d oi.org/10.1016/j.future.2010.12.006. URL: https://www.sciencedirect.com/science/article/pii/S0167739X10002554.