Technische Hochschule Ingolstadt

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:	2025–08–01
First examiner:	Prof. Dr. Bernd Hafenrichter
Second examiner:	Prof. Dr. Ludwig Lausser

Technische Hochschule Ingolstadt

Declaration in Accordance with § 30 Abs. 4 Nr. 7 APO THI

Declaration

I hereby declare that this thesis is my own work, that I have not presented it elsewhere for examination purposes and that I have not used any sources or aids other than those stated. I have marked verbatim and indirect quotations as such.

Ingolstadt, 2025–08–01

David Linhardt

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 (K8s) 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 (SaaS) 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 (SLA). 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) Service Level Agreement (SLA) while other tenants exceed their quotas (e.g., request rate)”. As noted by **carrion2022** “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 (KCP) offers three capabilities that map accurately onto today’s multi-tenancy pain points.

[1]: *Workspaces*. Kubernetes Control Plane (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). The name Kubernetes originates from the Greek word κυβερνήτης meaning helmsman or pilot (Kubernetes 2024). The abbreviation K8s results from counting the eight letters in between the "K" and the "s" (Kubernetes 2024).

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 **moravcik2022** almost 90% of organizations used Kubernetes as an orchestrator for managing containers and over 70% of organizations used it in production by 2021 (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 readiness for enterprise use 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 in 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 2025b; Docker 2025). The lightweight nature and isolation of containers can be leveraged by cloud-native software to enable both vertical and horizontal autoscaling, facilitated by fast startup times, as well as self-healing mechanisms and support for distributed, resilient infrastructures (Kubernetes 2025c; Kubernetes 2025e; AWS 2025b; 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 business 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). Additionally, they 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 (cgroups)* 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 within a single cluster and scoping resource names to prevent naming conflicts across different teams or projects (Kubernetes 2025f). However, these mechanisms may not always provide the sufficient isolation needed 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). Additionally, 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 2025d).
KCP implements the same Role-Based Access Control (RBAC)-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 (ABAC) (kcp Docs 2025c; Kubernetes 2025g). 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 allow 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, while `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

As defined in *Guide to SSL VPNs – Recommendations of the National Institute of Standards and Technology* 2008, p. 3-3 high availability (HA) “is a failover feature to ensure availability during device or component interruptions”.
KCP employs several mechanisms to ensure HA. Firstly to achieve this “failover feature” at the control-plane level, KCP relies heavily on its **cache-server layer** (kcp Docs 2025e). While individual shards can (and will) go offline, the cache server provides a logically-central, eventually-consistent replica of the small but critical objects that every shard must be able to see in order to keep tenant workspaces, API bindings or scheduling decisions functioning (kcp Docs 2025e). In essence, the cache server acts as a rendez-vous point that collapses the $n \times (n-1)$ mesh of direct shard-to-shard links into a single, well-known endpoint (kcp Docs 2025e). Instead of every shard having to maintain and re-establish dozens of peer connections after a failure, each shard needs only a single healthy path to any replica of the cache tier (kcp Docs 2025e). This single-hop topology is easier to debug, cheaper to secure, and—most importantly—continues to deliver the global metadata that controllers require even when one or several shards are offline (kcp Docs 2025e).
As described in kcp Docs 2025e, KCP uses a write-once / read-many replication model to populate and refresh that shared state:

[1]: *Write controllers* that run on every shard stream a selected set of objects, such as `APIExport`,

`APIResourceSchema`, `Shard`, certain RBAC rules and more into the cache server. Because the controller keeps its own authoritative copy, it can re-push data after a transient outage without the risk of split-brain. As noted by Levine 2021, split-brain describes “a phenomenon where the cluster is separated from communication but each part continues working as separate clusters, potentially writing to the same data and possibly causing corruption or loss.”.

[2]: *Read controllers* on all shards maintain informers that watch other shards’ objects from the cache server. These informers are isolated from a shard’s local etcd and can therefore start with a more tolerant back-off strategy: a shard may declare itself “ready” for tenant traffic even if cache connectivity has not yet been re-established, and will self-heal once the link returns.

Because objects in the cache server are stored per logical cluster but can be listed via wildcard paths, each controller needs only one LIST/WATCH stream per resource type rather than per cluster (kcp Docs 2025e). This dramatically reduces the number of long-lived Transmission Control Protocol (TCP) connections that must survive a fail-over and further increases control-plane availability.

Consequently, no individual shard becomes a single point of failure for global control-plane metadata. Only the cache tier must be deployed in a highly available configuration — something that can be achieved with standard Kubernetes Service + LoadBalancer constructs or by running two or more replicas behind a global Internet Protocol (IP) address. By funnelling cross-shard traffic through this purpose-built cache layer, KCP delivers the fail-over semantics described by *Guide to SSL VPNs – Recommendations of the National Institute of Standards and Technology* 2008: metadata remains available, and therefore the control plane remains operational, even when individual components fail.

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 for 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 as well as 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 promotes agility and efficiency, enabling rapid adoption.

Automation is the foundation for the utilization of the scaling effects that come along 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 (MSP) still leaves tenants running potentially different, separately-operated versions (AWS 2022, pp. 23–24). Furthermore it 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

At its core, zero-downtime deployment aims to upgrade all service instances while the application remains fully functional (Davis 2019, n. p., inside front matter). A broad consensus now centres on three primary techniques: rolling updates, blue-green deployments and canary releases, because each enables seamless updates without service interruptions (Rakshit and Banerjee 2024, p. 1).
Rolling updates incrementally replace small batches of instances, a process that integrates smoothly with continuous integration (CI)/continuous deployment (CD) pipelines and minimizes downtime (Rakshit and Banerjee 2024, p. 1). They are recognized as a standard zero-downtime mechanism that upgrades subsets of instances in sequence (Davis 2019, n. p., inside front matter). A noted drawback is that coexistence of old and new versions can create temporary latency spikes under heavy load (Rakshit and Banerjee 2024, p. 1).
Blue-green deployments run two identical environments in parallel so traffic can switch to the “green” version only after validation, thereby preserving high availability and cutting the blast radius of faults (Rakshit and Banerjee 2024, p. 1). This parallel setup allows thorough testing before traffic is rerouted, minimizing the introduction of new bugs (Rakshit and Banerjee 2024, p. 1). The principal trade-off is resource overhead because two full environments must be maintained during the transition (Rakshit and Banerjee 2024, p. 1).
Canary releases expose a new version to a small subset of users, gather real-time feedback and progressively expand the rollout when metrics look healthy (Rakshit and Banerjee 2024, pp. 1–2). Successful canaries rely on automated testing and continuous monitoring to catch regressions early (Rakshit and Banerjee 2024, p. 2). Yet managing multiple live versions can complicate performance tracking (Rakshit and Banerjee 2024, p. 2). Canaries are, according to AWS 2025a, pp. 33–34 “a type of blue/green deployment strategy that is more risk-averse”.
In-place deployments are generally regarded as the fourth deployment option, however they do not offer zero downtime (AWS 2025a, p. 34).
Together, these four patterns make up today’s deployment toolkit: the first three achieve zero-downtime rollouts, whereas in-place deployments trade reduced resource overhead for a short

service disruption window.

3.2. Kubernetes Scaling Methods

Kubernetes exposes several native autoscaling primitives that together enable both workload-centric and infrastructure-centric elasticity (Kubernetes 2025b). The Horizontal Pod Autoscaler (HPA) dynamically increases or decreases the replica count of scalable controllers such as *Deployments* or *StatefulSets* in response to metrics like central processing unit (CPU) utilization or custom application signals (Kubernetes 2025b). When right-sizing individual containers is preferable, the Vertical Pod Autoscaler (VPA) can recommend or automatically apply new CPU- and memory-request values for running Pods (Kubernetes 2025b). To ensure that sufficient capacity exists for these Pod-level changes, the Cluster Autoscaler grows or shrinks the underlying node pool by interacting with the cloud provider or on-premises infrastructure (Kubernetes 2025b). The documentation also highlights complementary techniques—such as event-driven autoscaling with Kubernetes Event Driven Autoscaler (KEDA) and scheduled scaling profiles, that extend Kubernetes beyond simple metric-based triggers (Kubernetes 2025b). A recent survey by Senjab et al. 2023, p. 1 categorizes “autoscaling-enabled scheduling” as one of the major research thrusts in Kubernetes scheduling literature. The authors note that such schedulers couple placement decisions with dynamic resource provisioning to reduce latency, boost utilization and cut operational costs (Senjab et al. 2023, p. 20). Many of the reviewed schemes embed forecasting or reinforcement-learning models that tune HPA or cluster-level loops in real time (Senjab et al. 2023, pp. 16–19). Open challenges identified include balancing multi-tenant fairness during bursts, preventing SLA violations while scaling and cutting energy consumption for greener operations (Senjab et al. 2023, pp. 20–23). The official project therefore provides HPA, VPA and the Cluster Autoscaler as modular building blocks for elasticity (Kubernetes 2025b). Contemporary research is now striving to orchestrate these primitives holistically inside the scheduler, paving the way for truly self-adaptive Kubernetes clusters (Senjab et al. 2023, pp. 6–7; Senjab et al. 2023, p. 22).

3.3. Multi-Tenancy Concepts in the Cloud

As already described in subsubsection 2.1.2, a multi-tenant solution is one that is used by multiple customers or tenants (Microsoft 2025, p. 57). As observed by Microsoft 2025, p. 137, architects can position themselves anywhere on a continuum that shares every resource among tenants at one extreme and deploys isolated resources for every tenant at the other. Amazon Web Services (AWS) names the same continuum with the terms *silo* (dedicated) and *pool* (shared) (AWS 2022, pp. 19–22). It emphasizes that these models are *not all-or-nothing concepts* and can be mixed per service or layer (AWS 2022, p. 19). To operate such mixtures efficiently, AWS separates a shared *control plane*, covering onboarding, identity, billing and other global services, from an *application plane* that hosts tenant workloads (AWS 2022, p. 10). Azure highlights the *Deployment Stamps* pattern, which deploys dedicated infrastructure for one or a small set of tenants to maximize isolation while reducing cost and operational overhead (Microsoft 2025, pp. 137–138).

Inside Kubernetes clusters, open-source systems such as **KCP**, **Capsule** and **Kiosk** partition a physical cluster into logical clusters via separate API servers or operators that enforce control-plane isolation (Nguyen and Y. Kim 2022, p. 651). As noted further by Nguyen and Y. Kim 2022, KCP has an API server to provide customers with an easy way to access their resources and supports advanced features such as inheritance and automated resource allocation, while Capsule and Kiosk use a Kubernetes extension to access their resources. The same study distinguishes *multi-tenant team* deployments that mainly share cluster namespaces from *multi-tenant customer* deployments that demand full isolation of data and control planes (Nguyen and Y. Kim 2022, p. 651). Capsule and Kiosk take a lighter “flat namespace” route. Capsule lets admins *replicate resources across a tenant’s namespaces* or copy them between tenants, easing day-to-day ops, but the authors caution that the pattern ““may not be fully scalable for extensive tenant settings”” and is not designed for edge scenarios (Şenel et al. 2023, p. 144581). Kiosk models each tenant as an *account* that may create a *space*: ““each space is strictly tied to only one namespace”” using templates (Şenel et al. 2023, p. 144581) Yet multi-cluster support is still on the roadmap and the codebase, however according to Şenel et al. 2023, it ““does not seem to be under active development””. At the *namespace only* end of the spectrum, the Hierarchical Namespace Controller (HNC) constructs multi-tenancy entirely from nested namespaces and policy inheritance, delivering near-zero control-plane overhead at the cost of weaker isolation (Şenel et al. 2023, p. 144581).

Stepping up the isolation ladder, *virtual-cluster frameworks*, such as **vcluster** run a full Kubernetes API server as a pod inside the host cluster, thus “each vcluster has a separate API server and data store” while its workloads are still scheduled onto the shared worker nodes (Şenel

et al. 2023, pp. 144580–144581). Other implementations of the same pattern (**VirtualCluster**, **k3v**, **Kamaji**) follow this design with varying degrees of data-plane isolation (Şenel et al. 2023, pp. 144580–144581). At the opposite extreme, **Arktos** represents a deep-modification approach: it injects tenant primitives directly into the Kubernetes API gateway and partitions, targeting “a single regional control plane to manage 300 000 nodes that multiple tenants will share” (Şenel et al. 2023, p. 144582). These additions complete the design space: hierarchical-namespace frameworks minimize overhead, virtual-cluster frameworks trade moderate cost for a dedicated control plane per tenant, and tenant-aware forks such as Arktos pursue extreme scale with built-in multi-tenancy primitives.

Ultimately, architects must weigh isolation, cost efficiency, performance, implementation complexity, and manageability to select the tenancy model that best fits their SaaS workload (Microsoft 2025, p. 137).

3.4. Related Work on KCP

Only two publicly available efforts examine KCP in depth: a peer-reviewed prototype by Nguyen and Y. Kim 2022 and a series of community workshops and conference presentations hosted at KubeCon and the platform engineering day European Union (EU) in Paris, all collected in the `kcp-dev/contrib` repository Vasek et al. 2025.

Nguyen and Y. Kim (2022) propose "a design of *dynamic resource allocation in [sic]Kubernetes multi-tenancy system* to address the missing dynamic resource allocation in the multi-tenant Kubernetes control plane". Their architecture adds "*scheduler/rescheduler*, Prometheus-driven autoscaler, and a *cloud provisioner* that can spin up new clusters based on workload changes" (Nguyen and Y. Kim 2022, p. 653). Central to the design is the KCP "*logical cluster*, a virtual cluster whose API resources are stored separately and can be created at *'nearly zero' cost for an empty cluster*" (Nguyen and Y. Kim 2022, p. 652). Preliminary tests report "*fast Pod creation time with the help of a policies-based scheduler*", but a comprehensive evaluation is deferred to future work (Nguyen and Y. Kim 2022, p. 651; Nguyen and Y. Kim 2022, p. 654).

Practical know-how is instead championed by the KubeCon 2025 London workshop, which frames KCP as a way to "*reimagine how we deliver true SaaS experiences for platform engineers*" on the official Cloud Native Computing Foundation (CNCF) YouTube channel (Cloud Native Computing Foundation 2025). All workshop material is collected in `kcp-dev/contrib`, "a repository containing *demo code, slides and other materials used in meetups and conferences*" (Vasek et al. 2025) underscoring that KCP expertise is presently practitioner-driven rather than research-driven. Together, the academic prototype validates KCP's elasticity potential, while the workshop series illustrates its API-centric platform model, highlighting both KCP's promise and the current research gap in systematic performance and security evaluation.

4. Conceptual Design

4.1. Proposed Scenario

To demonstrate the conception, implementation, and evaluation of a highly-scalable, highly available SaaS platform on KCP, the thesis adopts a deliberately light-weight yet realistic **case study**:
A **small and medium sized businesses (SMB) Web-Presence-as-a-Service**, whose sole purpose is to give SMBs an instantly available, single-page web presence. The service sits between do it yourself (DIY) website builders and full custom agency work. Tenants enter their company facts, choose a theme, and receive a live, secure page, without touching infrastructure. A data contract is used to allow the public frontend to be data-driven.

Actor	Responsibility	Interaction Pattern
SMB tenant	Enters or edits company information; chooses a theme	Write-heavy only at onboarding, then sporadic
End-User Visitor	Loads the generated page	Read-only; bursty traffic driven by marketing and search indexing
Platform Operator	Maintains themes, monitors capacity, rolls out new platform versions	Development and operations (DevOps)

Table 1: Actors, their responsibilities, and interaction patterns in the case case study

This scenario is attractive for evaluating KCP because it combines high multi-tenancy (potentially tens on thousands of logically isolated sites) with skewed workload characteristics, like very high read fan-out combined with low per-tenant write rate. The workload stresses horizontal scalability, without the confounding complexity of rich business logic or deep data lineage. The following subsection translates this narrative into concrete requirements.

4.1.1. Representative User Journey

To contextualize the conceptual design, this subsection describes a representative end-to-end journey from the perspective of the primary user: the SMB tenant.

[1]: *Onboarding*. The tenant accesses the dashboard and creates an account. They enter basic company information, select a theme, and upload optional assets like a logo or background image.

[2]: *Workspace Provisioning*. Upon submission, the platform automatically provisions a dedicated KCP workspace, generates a static `config.json` file, and triggers deployment of the tenant-specific frontend and API.

[3]: *Site Publication*. Once deployed, the tenant receives a public URL pointing to their live web presence. The frontend fetches dynamic content such as reviews from the tenant-local API.

[4]: *Ongoing Use*. The tenant can return to the dashboard at any time to update content. Any changes result in regeneration of the config and a controlled redeployment.

[5]: *End-User Interaction*. Visitors reach the tenant's public page, browse the content, and can optionally submit reviews. These reviews are stored in the tenant's isolated database.

This journey provides the foundation for the functional (see subsubsection 4.2.1: *Functional Requirements*) and non-functional (see subsubsection 4.2.2: *Non Functional Requirements*) requirements in the subsequent sections.

4.2. System Requirements

Building on the foundation of (see subsection 4.1: *Proposed Scenario*), this Chapter formalizes *what* the platform must do and *how well* it must do it before any architectural choices are justified. Clear, measurable requirements serve three purposes in this Thesis:

[1]: *Design driver*. They constrain the solution space explored in (see subsection 4.3: *Architecture Design with KCP for SaaS*), ensuring that every architectural element demonstrably supports a stated need rather than an implicit assumption.

[2]: *Benchmark for implementation*. During prototyping (see section 5: *Prototypical Implementation*) the tables in this chapter become acceptance criteria that guide configuration, automation scripts, and performance-test baselines.

[3]: *Reference for evaluation*. The metrics used in (see section 6: *Evaluation*) map one-to-one to the service-level objectives, latency budgets, and scalability targets enumerated here, allowing an objective pass/fail discussion of the prototype's behavior.

The derivation methodology of the requirements can be characterized as follows:

[1]: *Actor analysis*. Each functional capability traces back to an interaction Table in (see Table 1: *Actors, their responsibilities, and interaction patterns in the case case study*).

[2]: *Service-level expectations*. Non-functional figures reflect common SaaS SLAs for SMB-facing products and align with KCP documentation on acceptable control-plane latency.

[3]: *Platform constraints*. Limits on etcd input / output (I/O), shard utilization, and workspace-list latency bind scalability goals to realistic thresholds.

By separating *function* from *quality*, the chapter provides a traceable checklist that threads through design, implementation, and evaluation, ultimately demonstrating whether KCP can underpin a highly-scalable, highly-available SaaS offering for the target market.

4.2.1. Functional Requirements

The prototype offers only a handful of end-to-end user journeys (see subsubsection 4.1.1: *Representative User Journey*), yet each journey must be supported in a self-service and repeatable way across thousands of tenants. The purpose of this subsection is therefore to distill the actor interactions listed in Table 1 into a concise set of *functional* statements that can

later be traced unambiguously to implementation artifacts and test cases.
To stay aligned with the thesis goals, a requirement is admitted into the list, only if it is **visible to at least one external actor**, it can be **validated through an API call or user interface (UI) action**, and it is **independent of any particular architectural decision**.
The resulting catalogue of functional requirements is intentionally short. I focuses on Create Read Update Delete (CRUD) operations for tenant data, delivery of the public page, and the minimal platform workflow required to provision, store and cache tenant configuration.

F ID	Scope	Title	Description
F-01	*Tenant*	Company Profile CRUD	A tenant owner shall be able to perform CRUD operations on the company profile (name, address, contact, logo, about-text) through the dashboard API.
F-02	*Tenant*	Service Catalog CRUD	A tenant owner shall be able to manage a list of services/products.
F-03	*Tenant*	Ratings CRUD	Public visitors shall be able to post 1-to-5-star ratings with comments; the public API shall list ratings.
F-04	*Tenant*	Public Page Delivery	The per-tenant public frontend shall render the latest template and tenant data via the per-tenant API.
F-05	*Platform*	Tenant Provisioning	The dashboard shall create a new KCP workspace, deploy the tenant stack and return a public URL in $\leq$ 60 s.
F-06	*Platform*	Config Storage and Retrieval	Template configuration for every tenant shall be stored inside the tenants API as a JavaScript Object Notation (JSON) file.
F-07	*Tenant*	Config Caching	The per-tenant API shall cache configuration in-memory with a time to live (TTL) $\geq$ 24 h, falling back to the JSON file on cache miss.

Table 2: Functional Requirements

These requirements cover the complete life-cycle of a tenant environment from **creation** over **serving** to **evolution**. Furthermore they define a simple yet measurable consistency contract between object storage and the API cache.
By keeping the scope this narrow, the thesis can focus evaluation on control-plane scalability, HA, and page-delivery performance, without being distracted by edge functionality that would not exercise KCP in a meaningful way.

4.2.2. Non Functional Requirements

Where subsubsection 4.2.1 captured *what* the prototype must do, the present subsection specifies *how well* it must do it. Each requirement in Table 3 describes a service-level objective (SLO), that is:

[1]: *Observable*. It can be measured from outside the process boundary.

[2]: *Actionable*. It can be used as a pass/fail gate during architecture validation.

[3]: *Business aligned*. Its numeric target reflects viable SaaS expectations for SMB customers.

The SLOs can be summarized along the three quality dimensions **reliability and performance**, **scalability, consistency and security**, and **durability and operability**.
The following non-functional requirements provide the basis for the architecture design.

NF ID	Scope	Title	Description
NF-01	Tenant	Public Availability	The public page shall achieve $\geq$ 99.95 % monthly uptime; workspace moves or API pod restarts shall cause 0 failed GET requests.
NF-02	Platform	Dashboard Availability	The owner dashboard shall achieve $\geq$ 99.5 % monthly uptime (so brief maintenance windows are acceptable).
NF-03	Tenant	Page Performance	p95 full-page load time shall be $\leq$ 600 ms at 200 req/s from one region.

NF ID	Scope	Title	Description
NF-04	Tenant	API Latency	p95 latency for GET `/api/config` and GET `/api/reviews` shall be $\leq$ 150 ms under the same load.
NF-05	Platform	Horizontal Scalability	The control plane shall handle at least 2 000 tenant workspaces with etcd IO $\leq$ 70 % and < 100 ms workspace-list latency.
NF-06	Platform	Cache Consistency	Config cache shall reflect config changes within 15 min after a tenant triggers “publish” in the dashboard.
NF-07	Platform	Tenant Isolation and Security	Data and traffic from one tenant shall not be accessible to another (RBAC, row-level security, network policies).
NF-08	Platform	Data Durability	Reviews and config data shall have an recovery point objective (RPO) = 0 h (write-ahead logging (WAL)-based backups) and an recovery time objective (RTO) $\leq$ 2 h via cross-region restore.
NF-10	Operator	Maintainability / CI/CD	A full platform deployment (dashboard + initializer + tenant chart) shall complete via GitOps pipeline in $\leq$ 15 min.

Table 3: Non-Functional Requirements

4.3. Architecture Design with KCP for SaaS

4.3.1. Overview and Design Principles

The platform architecture is designed to support secure, scalable, and isolated multi-tenant Software-as-a-Service (SaaS) deployments within a cloud-native environment. It leverages KCP to provide logical Kubernetes-like workspaces per tenant, while maintaining centralized control through a parent workspace.
The architecture follows a clear separation of concerns: the parent workspace hosts the control plane responsible for provisioning and lifecycle management, while tenant workspaces encapsulate the execution and data layers of the application in a fully isolated environment.
The design adheres to the following core principles:

[1]: *Isolation by Workspace*. A dedicated vanilla Kubernetes cluster per tenant is operationally inhibitive. Conversely, a flat namespace in a single cluster risks cross-tenant resource contention, complicated RBAC, and a noisy neighbor problem. KCP's *workspace* abstraction offers the middle ground, providing strong boundaries at the API layer while keeping a single point to manage tenants.

[2]: *Dependencies*. The system minimizes runtime dependencies in favor of static or build-time mechanisms, e.g. injecting the config into the API at build time (see subsubsection 4.3.2: *Rationale for Config Injection Strategy*) to reduce operational complexity and deployment effort.

[3]: *Scalability*. The architecture supports horizontal scaling of both the control plane and tenant workloads. New tenants can be provisioned on demand without impacting existing tenants.

[4]: *Portability and Compliance*. Since all tenant data and services reside within a dedicated workspace, regulatory and compliance requirements (such as data locality or retention) can be more easily fulfilled or verified.

[5]: *Declarative Lifecycle Management*. All tenant resources are provisioned declaratively and idempotently via Kubernetes-native APIs, making the system predictable and suitable for GitOps-style workflows.

[6]: *Push state to the edge, pull control to the centre*. All *public* traffic terminates inside the tenant workspace (close to the cache and database), whereas *administrative* traffic aggregates in a central place, that can be rate-limited and protected.

This architectural foundation enables a modular and future-proof SaaS platform that favors operational autonomy, extensibility, and secure multi-tenancy.
Summed up these principles translate into the logical layout in Figure 1.

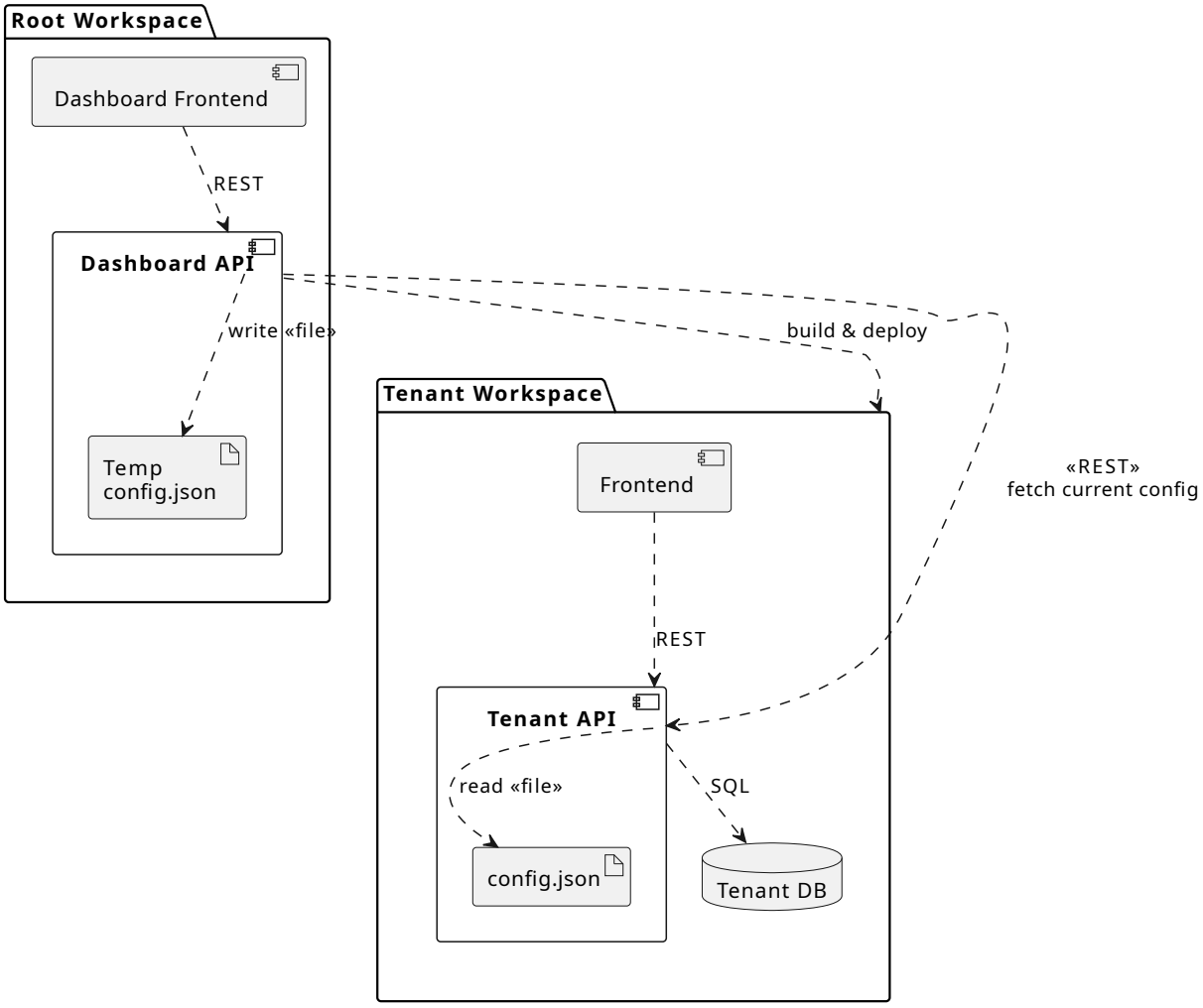

Figure 1: Overview of the system architecture

4.3.2. Rationale for Config Injection Strategy

To enable tenant-specific customization in a secure and maintainable manner, the configuration is injected directly into the API container during image build time using the Docker `-build-arg` mechanism. This approach ensures that the configuration resides fully within the corresponding tenant's workspace post-deployment, thereby preserving strong workspace isolation. Moreover,

it avoids the operational complexity associated with runtime config injection mechanisms and simplifies the tenant lifecycle by keeping all relevant data embedded within the container image. Several alternative approaches were considered during the design process. Each of them was evaluated based on criteria such as workspace boundary isolation, persistence, scalability, operational complexity, and dependency footprint. The following list summarizes these alternatives and the primary reasons for their exclusion:

[1]: *Persistent Volume (PV)*. PVs offer cluster-wide storage, but they are bound to the underlying infrastructure and not scoped to individual workspaces. Since tenant data must remain fully isolated within its respective workspace, using a shared PV would have violated this design constraint. Furthermore PVs would introduce significant operational complexity.

[2]: *Persistent Volume Claim (PVC)*. PVCs are workspace-scoped and technically suitable for storing tenant configuration. However, managing their lifecycle dynamically per tenant (including updates and clean-up) would introduce significant operational complexity.

[3]: *ConfigMap*. A ConfigMap is a lightweight and K8s native way to inject configuration, but it has a very constraining size limit (typically 1 MiB) and is not designed for cross-workspace usage. Since KCP workspaces enforce strict isolation, injecting a ConfigMap from the parent workspace into the tenant workspace would violate boundary constraints, or require custom controllers. It was primarily not considered a viable long-term option due to its limited support for binary data and poor scalability with growing or media-rich configuration payloads.

[4]: *Git Repository*. Polling or pulling tenant configuration from a Git repository would offer central control and versioning, but would couple each tenant's runtime to an external dependency. It would also require embedding Git credentials or Secure Shell (SSH) keys within the workspace, raising security concerns and operational burden. Furthermore the data does not live inside the tenant violating isolation principles.

[5]: *initContainer + emptyDir*. This approach involves using an `initContainer` to write the config into an `emptyDir` shared volume before the main API container starts. While this ensures data locality, the config is ephemeral and lost on pod restart or rescheduling. Additionally, updates would require a full pod redeployment including controlled init re-execution, adding complexity.

[6]: *Tenant database (DB)*. Storing the configuration in the tenant database would offer persis-

tence and locality. However access control for the write operations and reaching out to the DB from the parent workspace are critical pain points.

[7]: *Redis*. A Redis store within the tenant workspace was considered for fast config access. However, this introduces a full additional service dependency per tenant, which contradicts the goals of lightweight and cost-efficient tenant deployments. Redis also requires persistence management if configuration must survive restarts, therefore adding complexity.

[8]: *Central Document Store* Maintaining a central document store in the root workspace (e.g. MongoDB or MinIO) was ruled out due to isolation concerns. This would require the tenant API to reach out beyond its workspace boundary, which is explicitly avoided in the current architecture to enforce strict data sovereignty per tenant.

Ultimately, the chosen build-time config injection strategy strikes an effective balance between strong tenant isolation, operational simplicity, and runtime performance. While it requires an image rebuild for every configuration change, this trade-off is acceptable given the infrequent update pattern of tenant configuration and the clear benefits it provides in terms of system architecture and security boundaries.

4.3.3. Data Synchronization and Caching Strategy

As the tenant-API is built with its own **static** `config.json` as shown above (see subsubsection 4.3.2: *Rationale for Config Injection Strategy*), the resulting caching strategy is straightforward. At startup the tenant-API reads the immutable `config.json` that is baked into the container images filesystem (FS). Because the file never changes, while the image is running, the API loads its content once, materializes the contents **in memory**, and attaches a single long-lived cache entry with a TTL of 24 hours. A daily TTL offers two advantages without introducing the complexity of an explicit cache invalidation.

[1]: *Performance*. Subsequent requests bypass FS I/O entirely and are served from memory, massively increasing speed and reducing disk pressure under burst traffic.

[2]: *Resilience and self-healing*. Although the file is static, a bounded TTL guarantees that each pod refreshes its configuration once per day, so *bit-rot* or a silently corrupted memory page cannot persist indefinitely.

Because the only legitimate way to alter tenant configuration is to build and deploy a new image, a shorter TTL or a manual cache-flush API would add operational complexity without practical benefit. Caching the configuration with a long TTL therefore strikes the best balance between maximal runtime throughput and the minimal housekeeping needed to keep every pod's view of configuration fresh and fault-tolerant over time.

4.3.4. Workspace Topology and Multi-Tenancy model

Layer	KCP construct	Example resources	Provisioned Amount
Parent (control)	Workspace (`root`)	`dashboard`, `dashboard-api` Deployments	1 (static)
Tenant (data)	Workspace (`root:<tenant-id>`)	`frontend`, `tenant-api` Deployments, `DB` StatefulSet	1 per tenant

Table 4: Overview of architectural layers

The system architecture is logically divided into two primary layers, a static parent workspace and a dynamically provisioned set of tenant workspaces. Each layer corresponds to a dedicated KCP workspace and fulfills distinct responsibilities.
The parent workspace acts as the control plane of the system. It hosts globally accessible resources such as the Dashboard frontend and the API responsible for tenant lifecycle management. It exists as a single static `root` workspace and is not scaled horizontally.
Each tenant is isolated in its own dedicated workspace (`root:<tenant-id>`) created via KCP's Workspace CRD API. These tenant workspaces are dynamically provisioned on demand through the `dashboard-API`. They encapsulate all data and execution logic associated with the tenant, including the tenant-specific API, frontend, and database state.
Tenant workspaces are strictly isolated from each other and from the parent workspace. All tenant-related resources, including configuration and persistent state, reside exclusively within their respective workspaces, maintaining full logical and operational isolation.
Deleting a workspace recursively tears down all associated Kubernetes resources, ensuring complete and automated cleanup without additional control-plane logic.

4.3.5. Tenant Workspace: Data Plane Components

Component	Scaling	Purpose
Frontend (Next.js)	HPA based on CPU and queries per second (QPS)	Serves static hypertext markup language (HTML) / JavaScript (JS), does server-side rendering (SSR) for dynamic contents
API (Node.js)	HPA	Auth-less REST endpoints with cache consumed by the Next.js app.
DB	StatefulSet	Stores ratings

Table 5: Per Tenant Components

The request flow of the visitor path can be summarized as follows:

[1]: *Frontend*. The Next.js frontend is called.

[2]: *Tenant API*. The Next.js frontend calls the tenant API to get dynamic data. The tenant API reads from the cache.

[3]: *Data sources*. The tenant API periodically renews its cache based of a TTL. The dynamic data (reviews) comes from the tenant DB while the static data (config) comes from the tenant API.

This ensures, that for all customer interactions on the public website the workload remains local to the tenant workspace, thus improving the traceability.

4.3.6. Parent Workspace: Control Plane Components

Component	Scaling	Purpose
Dashboard	HPA	Authenticated UI for tenant owners; CRUD operations on tenant data
Parent API	HPA	validates input and writes JSON

Table 6: Control Plane Components

4.3.7. Scalability, Isolation and Security

The described architecture should be able to provide a high level of scalability, isolation and security for the stated use case. It should achieve this as follows:

[1]: *Horizontal Scalability*. Inside the tenant workspace the tenant-frontend (FE) and the tenant-API run as stateless deployments. A HPA driven by CPU utilization and QPS metrics adds or removes replicas linearly, so throughput increases proportionally with the number of pods (see **??**: **??**). Because the API keeps no local state, new replicas can be scheduled on any node without coordination, ensuring that burst traffic on popular sites never impacts neighboring tenants.

[2]: *Tenant Isolation*. Every customer receives its ows dedicated KCP workspace at onboarding time. A workspace maps to a private logical cluster stored in an independent etcd prefix, so objects created by one tenant are completely invisible to others. All runtime artifacts — FE pods, API pods, DB and even the `config.json` injected into the API — live exclusively inside that workspace. The platform never mounts cross-workspace volumes or reaches out to shared data services. Because there is **no shared state across workspaces** the design should eliminate noisy-neighbor effects, simplify compliance and guarantee data sovereignty for every tenant.

[3]: *Security*. Multi-tenancy is only acceptable if confidentiality and integrity are preserved across tenant boundaries. subsubsection 2.1.3 highlighted, that residual-data exposure, cross-tenant access and side-channel attacks are the primary threats in shared environments. Because every tenant runs inside its **own** KCP workspace, backed by a private etcd prefix, objects created in one workspace are completely invisible to others, including non-namespaced objects like CRDs (kcp Docs 2025l). This hard logical boundary, combined

with the absence of any shared state outside the workspace allows for:

Data confidentiality, as data resides inside the tenant, eliminating residual-data exposure vectors.

Privilege containment, as workspace-local RBAC rules grant only verbs needed to operate intra-workspace resources and there are no cluster-wide roles, blocking lateral movement and privilege-escalation attempts.

Secure storage, as only the tenant-API has write access on the tenant-DB.

By confining a potential attacker to the compromised workspace, the architecture should neutralize the multi-tenant security risks identified earlier while still reaping the economic benefits of sharing the underlying platform.

Collectively the security measures have to fulfill nf7.

4.4. Automated Deployment Strategies

Platform success depends on a deployment workflow that is fully automated, repeatable, and invisible to tenants. The design therefore distinguishes between the **first-time (initial) deployment** of a tenant workspace and the **continuous update policy** that keeps thousands of workspaces on a single, uniform release without interrupting production traffic.

4.4.1. Initial Tenant Deployment Pipeline

When a new customer completes the on-boarding flow F-05, the Dashboard-API running in the root workspace emits a custom resource (`TenantDeploymentRequest`) that seeds a GitOps pipeline (TODO: insert figure). The pipeline is implemented as a set of *Tekton* tasks that run directly under the control of the Dashboard-API, therefore no separate *ArgoCD* controller is required.

[1]: *Workspace bootstrap*. A controller watching for `TenantDeploymentRequest` objects calls the KCP API to create a dedicated workspace `root:<tenant-id>` and assigns only the minimal RBAC roles required by the tenant components.

[2]: *Image build*. A build task invokes `docker buildx` with `–build-arg CONFIG_JSON` to inject the tenant-specific configuration into the API image layer (cf. subsubsection 4.3.2). The resulting API and FE images are pushed to the internal registry and tagged `:<tenant-id>-<git-sha>`.

[3]: *Manifest rendering*. Helm/Kustomize templates are parameterized with the tenant ID, image tags and a computed `ROUTE_BASE` and rendered to plain YAML Ain't Markup Language, formerly Yet Another Markup Language (YAML).

[4]: *Route allocation*. The pipeline creates an `Ingress` (NGINX) or `Route` (OpenShift) inside the tenant workspace. A wildcard Domain Name System (DNS) entry (`*.example.com`) avoids central DNS updates and keeps certificates simple.

[5]: *Deployment*. A Helm `upgrade –install` task applies the rendered manifests in the tenant workspace, rolling out the FE, API and backing DB. Readiness probes ensure the pipeline blocks until all Pods are ready.

[6]: *Status callback*. Once every component is healthy, a final task POSTs the public URLs and the image Secure Hash Algorithm (SHA) back to the Dashboard-API, which in turn marks the tenant as "active" for subsequent logins.

By chaining these Tekton tasks the platform turns the single `TenantDeploymentRequest` event into a fully-isolated, routable tenant environment in one pass. Every artefact is generated from code that lives in Git, every cluster mutation is scoped to the tenant workspace, and every build is automatically signed by Tekton Chains. Consequently the pipeline satisfies all functional requirements (TODO: which) while remaining short enough to version-control next to the product source and easy to extend with additional steps (e.g. quota enforcement or cost metering) as the service matures.

4.4.2. Continuous Deployment

After the initial hand-off the platform must react to two kinds of change without operator intervention:

[1]: *Source-code updates*. A merge on the main branch of either the tenant FE or the tenant API Git repository emits a signed GitHub Webhook TODO: finish this when more info is available

[2]: *Content-driven updates (single tenant rebuild)*. When business users change dynamic catalogue data or feature flags, the dashboard API issues a `TenantReconfigureRequest` object that carries the new `config.json` payload. A dedicated Tekton pipeline in that tenant's workspace performs a rebuild of the tenant API image with a new `config.json`. Following this a local helm upgrade of the tenant API is performed, while the FE and DB remain untouched.

4.4.3. Zero-Downtime Rollout

Both continuous-deployment pipelines should deploy the updated version of the tenant's application with zero downtime for customers while ensuring stability. This can be achieved through a *canary deployment*. A canary deployment is a type of blue-green deployment strategy (AWS 2025a, pp. 33–34). Blue-green deployments are a common deployment strategy to ensure zero downtime. In a blue-green deployment the currently running application (blue) runs alongside the newly deployed, updated application (green) (AWS 2025a, pp. 32–33)[pp. 176–178]davis2019. This allows for testing the green version in production, while still having the blue version live to handle production traffic (Davis 2019, pp. 176–178). With a canary deployment, the cut-over from blue to green is performed gradually (AWS 2025a, pp. 33–34). A small percentage of traffic is first routed to the green application, the *canary-group*, so that its real-world behavior can be observed under production load (AWS 2025a, pp. 33–34). If no regressions are detected, the remaining traffic is shifted either in one further step or through a series of linear increments until

100 % of users are served by the green version (AWS 2025a, pp. 33–34). Because the blue environment remains intact throughout the process, any anomaly can be mitigated instantly by redirecting requests back to it, thereby minimizing blast radius while still achieving zero downtime for customers (AWS 2025a, pp. 33–34).
Broken down step by step, the pipelines should act as follows:

[1]: *Readiness probe*. Verifies readiness probes and functional smoke-tests on the green Pods.

[2]: . *Shift traffic*. Shifts traffic incrementally by patching the weight on the tenant's ingress.

[3]: *Canary window*. Aborts and rewinds the weight if any SLO breach or error-budget drain is detected during the canary window.

[4]: *Deletion*. Deletes the blue deployment only after 100 % of requests have successfully moved to green.

4.5. Choice of Technologies

The platform architecture leverages modern, well-supported technologies that align with the core design principles of modularity, performance, and isolation. Each major component is built using tools selected for their suitability in a cloud-native, tenant-isolated SaaS environment.

[1]: *Dashboard FE*. The central Dashboard is implemented using `Next.js`. Its file-based routing, support for both static and server-side rendering, and seamless integration with TypeScript make it a suitable choice for building a modular and responsive frontend. Reusability and developer ergonomics were key factors in this decision.

[2]: *Tenant FE*. Each tenant is provisioned with a separate frontend instance based on a shared `Next.js` template. This enables customization at the workspace level while maintaining consistent structure and behavior. Next.js also allows efficient static generation of tenant-specific content during build time. To further streamline provisioning, a pre-defined set of layout and component *skeletons* is embedded in the template repository. These skeletons provide a consistent structure for common views, while still allowing per-tenant extension and branding.

[3]: *Dashboard API*. The control plane's API is built using `Node.js` and `Express`, providing a lightweight, fast, and JSON-native web service environment. It integrates a volatile `node-cache` layer for storing configuration artifacts during tenant provisioning, minimizing external dependencies.

[4]: *Tenant API*. Each tenant workspace contains its own `Node.js` + `Express` API instance. This ensures strict separation of runtime logic and supports injection of static configuration at image build time. The framework's flexibility and ecosystem make it well-suited for containerized multi-tenant deployment.

[5]: *Temporary Config Store*. During tenant creation, configuration files are temporarily stored on the Dashboard API's local filesystem. These files are injected into the tenant image using Docker's `-build-arg` and `COPY` mechanisms. This avoids cross-workspace communication and external storage dependencies.

[6]: *Tenant DB*. The tenants database layer is provided by `Citus`, a horizontally scalable extension of PostgreSQL. Citus allows sharded, isolated storage of tenant data and supports scaling out as load increases. Its compatibility with PostgreSQL clients simplifies development and integration.

This technology selection provides a robust foundation for implementing the desired multi-tenant SaaS architecture while keeping the operational footprint minimal and the development process maintainable. Furthermore it aims to live inside the TypeScript (TS) ecosystem, simplifying the development process.

5. Prototypical Implementation

5.1. Infrastructure with KCP

5.2. Tenant Provisioning (Automation, Multi-Tenancy)

5.3. Scaling Mechanisms (Horizontal Pod Autoscaler)

HPA in Kubernetes can react to any metric that the control plane exposes through resource or custom-metrics APIs. For a request-driven *express* service, such as the tenant-API the most accurate signal would be **QPS** at the ingress of each pod, because it tracks the real work performed and is immune to the noise that CPU-bound metrics introduce when background jobs or garbage collection dominate a sample window. Nevertheless, the implementation in its prototypical nature deliberately relies on the built-in **CPU utilization metric** alone, contrary to the architectural design. The choice is pragmatic rather than idealistic. Instrumenting per-pod QPS would require an extra metrics pipeline or at minimum a side-car that counts requests, a Prometheus deployment to scrape the counter, and a Prometheus adapter (or custom-metrics server) so the HPA controller can consume the data. Each of those moving parts introduces significant configuration surface, security considerations, and potential failure modes that distract from the core objective of this thesis: to validate multi-tenant isolation and deployment strategies, not to engineer a production-grade observability stack. CPU metrics, by contrast, are available out of the box from the kubelet and require only two declarative lines in the HPA manifest.
Using CPU as the scaling trigger therefore keeps the infrastructure lightweight, repeatable, and easy to grasp while still demonstrating that the platform can elastically adjust capacity under load. In a real-world SaaS environment the HPA spec could be refined to a compound metric using QPS latency percentiles, or custom business Key Performance Indicators (KPIs) , but for a prototype aimed at architectural proof of concept, the built-in metric is “good enough” and avoids adding too much complexity out of the thesis scope.

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 (2025a). *AWS Whitepaper - Overview of Deployment Options on AWS*. AWS. URL: https://docs.aws.amazon.com/whitepapers/latest/overview-deployment-options/welcome.html (visited on 07/02/2025).

AWS (2025b). *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.

Cloud Native Computing Foundation (2025). *Tutorial: Exploring Multi-Tenant Kubernetes APIs and Controllers With Kcp*. Video description consulted. URL: https://youtu.be/Fb_3dWJdY9I?si=OgAgV0wyyONBWjma (visited on 07/02/2025).

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.

Guide to SSL VPNs – Recommendations of the National Institute of Standards and Technology (2008). NIST Special Publication 800-113. Standard. NIST, p. 87.

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.archive.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). *Autoscaling Workloads*. URL: https://kubernetes.io/docs/concepts/workloads/autoscaling/ (visited on 07/02/2025). Archived at https://web.archive.org/web/20250702112540/https://kubernetes.io/docs/concepts/workloads/autoscaling/.

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

Kubernetes (2025f). *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 (2025g). *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.

Levine, Steven (June 2021). *Red Hat Enterprise Linux 7 High Availability Add-On Overview*. Version 7.1-1 — last updated 2021-06-29. Red Hat. URL: https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/7/html/high_availability_add-on_overview/index.

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.

Microsoft (2025). *SaaS and multitenant solution architecture*. Microsoft. URL: https://learn.microsoft.com/en-us/azure/architecture/guide/saas-multitenant-solution-architecture/ (visited on 07/02/2025).

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.

Rakshit, Haranath and Subhasis Banerjee (2024). "Scalability Evaluation on Zero Downtime Deployment in Kubernetes Cluster". In: *2024 IEEE Calcutta Conference (CALCON)*, pp. 1–5. DOI: 10.1109/CALCON63337.2024.10914046.

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.

Senjab, Khaldoun, Sohail Abbas, Naveed Ahmed, and Atta ur Rehman Khan (June 2023). "A survey of Kubernetes scheduling algorithms". In: *Journal of Cloud Computing* 12.1. ISSN: 2192-113X. DOI: 10.1186/s13677-023-00471-1. URL: http://dx.doi.org/10.1186/s13677-023-00471-1.

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/.

Vasek, Robert, Mangirdas Judeikis, Marvin Beckers, Stefan Schimanski, Mirza Kopic, and Nelo-T˙ Wallus (2025). *KCP contrib repository*. URL: https://github.com/kcp-dev/contrib (visited on 07/02/2025). Forked at https://github.com/MysterionAutotronic/kcpContribFork.

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://doi.org/10.1016/j.future.2010.12.006. URL: https://www.sciencedirect.com/science/article/pii/S0167739X10002554.

A. Repository Overview

Note. The following template and artifacts repositories are included for completeness. They have no dedicated appendix sections.

Repository	Appendix	Scope
`BachelorThesis_TenantFE`	Appendix B	Next.js tenant frontend (per-workspace)
`BachelorThesis_TenantBE`	Appendix C	Express tenant API (+ in-memory cache)
`BachelorThesis_ConfigSchema`	Appendix D	JSON schema / validation for `config.json`
`BachelorThesis_DashboardFE`	Appendix E	Central operator/tenant dashboard (Next.js)
`BachelorThesis_DashboardBE`	Appendix F	Onboarding + pipeline triggers (Express)
`BachelorThesis_Infra`	Appendix G	KCP / cluster manifests and automation
Templates:		
`BachelorThesis_TemplateFE`	—	Template for frontends (Next.js)
`BachelorThesis_TemplateBE`	—	Template for backends (Express)
Artifacts:		
`BachelorThesis`	—	LaTeX thesis source code

B. TenantFE (Selected Code)

Directory Structure

```
./BachelorThesis_TenantFE//
├── Dockerfile
└── tenant_fe/
    ├── public/
    ├── src/
    │   ├── app/
    │   │   ├── api/
    │   │   │   ├── config/
    │   │   │   │   └── route.ts
    │   │   │   └── crash/
    │   │   │       └── route.ts
    │   │   ├── layout.tsx
    │   │   └── page.tsx
    │   └── lib/
    │       └── ConfigProvider.tsx
    └── tsconfig.json
```

Figure 2: TenantFE directory tree (trimmed)

Key Files

Listing 1.: API route: `src/app/api/config/route.ts`

```ts
'use server'

import { NextRequest, NextResponse } from 'next/server';

if (!process.env.CONFIG_ENDPOINT) console.error('environment variable CONFIG_ENDPOINT not defined');
const endpoint = process.env.CONFIG_ENDPOINT!;

export async function GET(req: NextRequest) {
    const body = await req.json();

    try {
        const response = await fetch(endpoint, {
            method: 'GET',
            headers: {
                'Content-Type': 'application/json',
                'user': body.user
            }
        });
        const data = await response.json();
        return new NextResponse(JSON.stringify(data), { status: response.status });
    } catch (error) {
        console.error('Proxy request failed:', error);
        return new NextResponse('Internal Server Error trying to reach ' + endpoint, { status: 500 });
    }
}
```

Listing 2.: API route: `src/app/api/crash/route.ts`

```ts
'use server'

import { NextRequest, NextResponse } from 'next/server';

if (!process.env.CRASH_ENDPOINT) console.error('environment variable CRASH_ENDPOINT not defined');
const endpoint = process.env.CRASH_ENDPOINT!;

export async function GET(req: NextRequest) {
    try {
        const response = await fetch(endpoint, {
            method: 'GET',
            headers: {
                'Content-Type': 'application/json',
                'user': req.headers.get('user') || ''
            }
        });
        return new NextResponse(response.body, {
            status: response.status,
            headers: {
                'Content-Type': response.headers.get('content-type') || 'text/plain',
                'Cache-Control': 'no-store',
            }
        });
    } catch (error) {
        console.error('Crash request failed:', error);
        return new NextResponse('Internal Server Error trying to reach ' + endpoint, { status: 500 });
    }
}
```

B. TenantFE (Selected Code)

Listing 3.: Root page: `src/app/page.tsx`

```tsx
'use client';

import { useConfig } from '@/lib/ConfigProvider';
import styles from './page.module.css';

export default function Home() {
    const cfg = useConfig();

    let loc: Intl.Locale | undefined = undefined;
    if(cfg.address?.country) {
        loc = new Intl.Locale(cfg.address.country);
    }

    function countryName(): string | null {
        if (!loc) return null;
        const countryEn = new Intl.DisplayNames([loc.language], { type: 'region'});
        const res = countryEn.of(loc.region!);
        if (!res) return null;
        return res;
    }

    return(
        <main>
            <div className={styles.center}>
                <h1 className={styles.companyName}>{cfg.companyName}</h1>
                <p>{cfg.proposition}</p>
            </div>
                {
                    cfg.products ?
                        <div className={styles.productsDiv}>
                            <h2>Products</h2>
                            <ul>
                                {cfg.products.map(p => <li key={p}>{p}</li>)}
                            </ul>
                        </div>
                    :
                        null
                }
            <div className={styles.addressDiv}>
                <h2 className={styles.addressHeader}>Address</h2>
                <address>
                    {cfg.address?.street} {cfg.address?.streetNumber} <br/>
                    {cfg.address?.zipCode} {cfg.address?.city} <br/>
                    {countryName()}
                </address>
            </div>
            <div className={styles.aboutDiv}>
                <h2 className={styles.aboutHeader}>About us</h2>
                <p>{cfg.about}</p>
```

C. TenantBE (Selected Code)

Directory Structure

```
./code/BachelorThesis_TenantBE//
├── Dockerfile
├── src/
│   └── server.ts
└── tsconfig.json
```

Figure 3: TenantBE directory tree (trimmed)

Key Files

Build and Runtime

D. ConfigSchema

Directory Structure

```
./code/BachelorThesis_ConfigSchema//
├── dist/
├── src/
│   └── index.ts
├── tests/
│   └── index.test.ts
└── tsconfig.build.json
```

Figure 4: ConfigSchema directory tree (trimmed)

E. DashboardFE (Selected Code)

Directory Structure

```
./code/BachelorThesis_DashboardFE//
├── dashboard_fe/
│   ├── public/
│   ├── src/
│   │   ├── app/
│   │   │   ├── api/
│   │   │   │   └── config/
│   │   │   │       └── route.ts
│   │   │   ├── dashboard/
│   │   │   │   └── page.tsx
│   │   │   ├── layout.tsx
│   │   │   ├── login/
│   │   │   │   ├── actions.ts
│   │   │   │   └── page.tsx
│   │   │   └── page.tsx
│   │   ├── components/
│   │   │   ├── ConfigForm.tsx
│   │   │   ├── DashboardButton.tsx
│   │   │   └── LoginForm.tsx
│   │   └── lib/
│   │       └── auth.ts
│   └── tsconfig.json
└── Dockerfile
```

Figure 5: DashboardFE directory tree (trimmed)

F. DashboardBE (Selected Code)

Directory Structure

```
./code/BachelorThesis_DashboardBE//
├── Dockerfile
├── src/
│   ├── redis.ts
│   └── server.ts
└── tsconfig.json
```

Figure 6: DashboardBE directory tree (trimmed)

G. Infra (Scripts and Manifests)

```
./code/BachelorThesis_Infra//
├── config/
│   └── config.example.json
├── k8s/
│   ├── apibindings/
│   │   └── standard-bindings.yaml
│   ├── cluster/
│   │   └── kind-config.yaml
│   ├── deployments/
│   │   ├── dashboard-be.yaml
│   │   ├── dashboard-fe.yaml
│   │   ├── debug.yaml
│   │   ├── tenant-be.yaml
│   │   └── tenant-fe.yaml
│   ├── ingress/
│   │   └── dashboard-ingress.yaml
│   ├── kcp-crds/
│   │   ├── apis.kcp.io_apibindings.yaml
│   │   └── apis.kcp.io_apiexports.yaml
│   ├── rbac/
│   │   └── root.yaml
│   ├── roleBinding/
│   │   └── tmc-role-binding.yaml
│   └── workspaces/
│       └── root-dashboard-cluster.yaml
└── scripts/
    ├── add-to-hosts.sh*
    ├── build/
    │   ├── dashboardBE.sh*
    │   ├── dashboardFE.sh*
    │   ├── tenantBE.sh*
    │   └── tenantFE.sh*
    ├── cleanup-hosts.sh*
    ├── clone-repos.sh*
    ├── CRDs/
    │   └── create-tenant-x-CRD.sh*
    ├── create-dashboard-workspace.sh*
    ├── create-tenant-x-cluster.sh*
```

```
├── create-tenant-x-workspace.sh*
├── install-deps.sh*
├── path.sh*
├── restore-kind-context.sh*
├── setup-kcp.sh*
├── setup-krew-plugins.sh*
├── setup-tmc.sh*
├── start-kcp.sh*
├── start-tmc-kcp.sh*
└── update-krew.sh*
```

Figure 7: Infra directory tree (trimmed)

H. Documentation Excerpts

I. Slack Channel Screenshots

Note: User names and sensitive data are redacted to comply with privacy requirements.