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

This thesis presents a practical, reproducible approach to building and operating a lightweight, Kubernetes-based multi-tenant web stack from local development to cluster networking and control-plane virtualization. Two cluster types (“dashboard” and “tenant”) are implemented to surface the integration points between containerized Next.js front ends, Node.js/Express back ends, and core Kubernetes primitives (deployments, services, ingress) on top of kind. Per-tenant clusters are created with deterministic host-to-node port mappings, while ingress-nginx and host entries provide name-based routing. KCP/TMC is incorporated conceptually for placement and synchronization by way of SyncTargets, Locations, Placements, and APIBinding of exported Kubernetes APIs into target workspaces. Evaluation follows scenario-driven criteria focused on functional correctness, reproducibility, isolation, and operability. The prototype delivers front ends and configuration APIs reachable under predictable hostnames and ports, with a clear separation of server- and client-side networking concerns: server code consumes an internal absolute endpoint, and client code targets a stable proxy implemented as a Next.js route handler with explicit cache control. Reproducibility is ensured by idempotent scripts, pinned tool and image versions, and consistent cluster templates. Shared front- and back-end templates and a central configuration schema enable uniform behavior across tenants and align with a CRD-oriented design.

The work contributes a concise troubleshooting playbook that addresses recurring issues in such stacks. Full utilization of KCP/TMC within the running prototype proved infeasible under the given constraints. The limitations and trade-offs are analyzed to inform future integration. Scope is intentionally limited to a developer-grade environment rather than production hardening or quantitative SLOs.

Overall, the thesis documents an architecture, reproducible workflow, and operational guidance that reduce friction when building multi-tenant web systems on Kubernetes, and outlines a path for future work: completing the KCP/TMC binding pipeline, standardizing ingress and TLS, adding telemetry for metric-based evaluation, and extending tests to multi-node and higher-scale scenarios.

Acronyms

ABAC Attribute-Based Access Control. 11

AKS Azure Kubernetes Service. 49

API Application Programming Interface. 1, 3, 7, 11–13, 19–21, 23, 25, 26, 28–38, 40, 42, 45, 47, 48, 56–58, 63, 64

AWS Amazon Web Services. 19

AZ Availability Zone. 2

B2B Business to Business. 15

B2C Business to Customer. 15

BE backend. 2, 3, 42–44, 47, 48, 50–54, 57

CD continuous deployment. 16, 27

CDN Content Delivery Network. 59

cgroups control groups. 9

CI continuous integration. 16, 27, 63, 64

CLI Command Line Interface. 43

CNCF Cloud Native Computing Foundation. 21

CORS Cross-Origin Resource Sharing. 3, 47–49, 51, 56, 59, 62, 64

CPU central processing unit. 18, 34–36, 61

CRD Custom Resource Definition. 2, 11, 12, 33, 35, 40, 42, 44–48, 50, 51, 54–64

CRUD Create Read Update Delete. 25, 35

DB database. 30–32, 34–38, 40

DevOps development and operations. 22

DIY do it yourself. 22

DNS Domain Name System. 37, 48, 60

DoD Definition of Done. 42

E2E End-to-End. 3, 43, 51–53, 58, 59, 61–64

EKS Elastic Kubernetes Service. 49

EU European Union. 21

FE frontend. 2, 3, 35, 37, 38, 40, 42–44, 47, 48, 51–54, 56, 57

FS filesystem. 32

GC Garbage Collection. 61

GKE Google Kubernetes Engine. 49

HA high availability. 3, 13, 26

HNC Hierarchical Namespace Controller. 19

HPA Horizontal Pod Autoscaler. 2, 18, 34–36, 51, 56, 59, 64

HTML hypertext markup language. 34, 45

HTTP Hypertext Transfer Protocol. 48, 56, 61

HTTPS Hypertext Transfer Protocol Secure. 48, 51

I/O input / output. 24, 32

IP Internet Protocol. 14, 46, 49, 52

ISR Incremental Static Regeneration. 45, 58, 59

JS JavaScript. 34

JSON JavaScript Object Notation. 25, 35, 40, 45, 46, 57, 58

K8s Kubernetes. 5–7, 11, 30, 59

KCP Kubernetes Control Plane. 1–3, 7, 10–14, 19, 21–26, 28, 32, 33, 35, 37, 40, 47, 48, 50, 52, 55, 58, 60, 62–64

KEDA Kubernetes Event Driven Autoscaler. 18, 59, 64

kind Kubernetes IN Docker. 48, 58, 59

KPI Key Performance Indicator. 2, 36

MCR multicluster-runtime. 63

MSP Managed Service Provider. 15

OS Operating System. 49

PDF Portable Document Format. 72

PV Persistent Volume. 30

PVC Persistent Volume Claim. 30

QPS queries per second. 34–36

RBAC Role-Based Access Control. 2, 3, 11–13, 27, 28, 36, 37, 42, 45, 50, 52, 56, 58, 62–64

RED Rate, Error, Duration. 64

REST Representational State Transfer. 12, 34

RPO recovery point objective. 27

RTO recovery time objective. 27

SaaS Software as a Service. 1–3, 5, 6, 10, 15, 16, 20, 21, 24, 26, 36, 41, 62

SHA Secure Hash Algorithm. 37

SLA Service Level Agreement. 6, 18, 24

SLI Service Level Indicator. 58, 59

SLO service-level objective. 2, 26, 39, 53, 54, 58, 62, 64

SMB small and medium sized businesses. 22–24, 26

SSH Secure Shell. 30

SSR server-side rendering. 34, 45

TCP Transmission Control Protocol. 14

TLS Transport Layer Security. 51

TMC Transparent Multi Cluster. 2, 3, 47, 48, 50, 52, 55, 58, 60, 62–64

TS TypeScript. 41

TTL time to live. 25, 32, 34, 58

UI user interface. 25, 35, 57

URL Uniform Resource Locator. 12, 23, 25, 37, 45, 48, 61

VCS Version Control System. 43

VPA Vertical Pod Autoscaler. 18, 59, 64

WAL write-ahead logging. 27

YAML YAML Ain't Markup Language, formerly Yet Another Markup Language. 37

Glossary

Admission Webhook external validator/mutator called by the Application Programming Interface (API) server before objects are stored. 12

ArgoCD A GitOps delivery controller that watches a Git repo and continuously syncs your Kubernetes cluster to the declarative state stored there.. 37

bit-rot The silent, gradual corruption of stored digital data whereby individual bits flip or decay over time due to media degradation, radiation, or other physical effects, rendering files partially or wholly unreadable.. 32

container A lightweight, portable, and isolated runtime environment that packages an application together with its dependencies and configuration. Containers share the host system's kernel but run in separate user spaces, enabling consistent execution across different environments.. 7, 8, 10

containerization packaging code and dependencies in Operating System (OS)-level containers. 8

data contract a *formal, version controlled agreement* between a data producer and its consumers, much like an API specification.. 22

Docker Docker is an open-source platform that enables developers to package applications and their dependencies into lightweight, portable containers that run consistently across different environments.. 29

etcd strongly-consistent key-value store that backs both Kubernetes and Kubernetes Control Plane (KCP) metadata. 11, 12, 14, 24, 35

Express Express is a minimal and flexible Node.js web application framework that provides robust features for building APIs and web servers.. 40

fan-out the pattern of sending one message, request, or event to many downstream consumers in parallel.. 22

high availability architectural and operational strategies employed to ensure that applications remain accessible and operational even in the face of failures, whether those are due to hardware malfunction, network issues, or other disruptions.. 8

key performance indicator A key performance indicator is a quantifiable metric chosen to measure progress toward a specific business or operational objective.. 36

Kubernetes Google-born container-orchestrator that underpins your whole platform. 1, 3, 5, 7–12, 19, 20, 28

Kubernetes Control Plane multi-tenant, horizontally-scalable control-plane project. 11

multi-tenancy many tenants sharing one platform while remaining logically isolated. 5–7, 9, 11, 22

Next.js Next.js is an open-source web development framework created by the private company Vercel providing React-based web applications with server-side rendering and static rendering.. 34, 40

Node.js Node.js is a free, open-source, cross-platform JavaScript runtime environment that lets developers create servers, web apps, command line tools and scripts.. 34, 40

performance isolation ensuring one noisy tenant can't violate another's Service Level Agreement (SLA). 6, 10

Redis Redis is an in-memory key-value data store for caching purposes.. 31

split-brain 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.. 13

Tekton A Kubernetes-native CI/CD framework that defines pipelines and tasks as CRDs, letting you run containerized build, test, and deploy steps entirely inside the cluster.. 37

Contents

List of Figures

List of Tables

1. Introduction

This thesis examines the conception, implementation, and evaluation of a highly scalable and highly available Software as a Service (SaaS) platform built on KCP. The prototype explores control-plane virtualization to provision and operate per-tenant application stacks while maintaining strong isolation and shared platform services. The focus is on translating architectural goals, such as elastic scale, fault tolerance, and tenant isolation into concrete design and code on top of KCP and Kubernetes.
Within this frame, the study evaluates what levels of scalability, availability, reproducibility, and change management are attainable today with KCP-based workflows, and where practical limits emerge in a realistic developer setup. The next subsections motivate the problem and context (subsection 1.1), define objectives and scope (**??**), and outline the thesis structure leading through design, implementation, evaluation, and conclusions (subsection 1.3).

1.1. Problem Statement and Motivation

Modern SaaS platforms depend on multi-tenancy to reach viable unit economics and elastic scale, yet shared infrastructure introduces nontrivial risks around security, fairness, and performance isolation. As outlined in subsubsection 2.1.2, multi-tenancy improves utilization and operational efficiency, but amplifies concerns such as residual-data exposure, limited tenant control and observability, scheduling fairness, and weak performance isolation, together with a pressing need for automation at scale (subsubsection 2.1.3). The practical question is not whether to adopt multi-tenancy, but how to do so safely, repeatably, and cost-effectively in contemporary cloud-native stacks.
This thesis addresses the problem of designing, implementing, and evaluating a Kubernetes-based SaaS platform that targets high scalability and high availability while preserving strong tenant isolation. The motivation is to explore whether control-plane virtualization — specifically KCP — can provide the isolation, extensibility, and automation hooks that modern SaaS demands, and to surface the operational realities and maturity gaps of this approach under realistic development conditions. KCP appears promising because it offers workspace-level isolation, a Kubernetes-compatible API surface, and built-in sharding (subsubsection 2.1.4), potentially aligning architectural goals with familiar tooling and workflows. The remainder of the thesis translates these drivers into concrete architecture and code, then evaluates what is feasible today and where additional engineering or ecosystem maturity is required.

1.2. Objectives and Scope

The work targets a *prototype* that demonstrates key concepts of a highly scalable and highly available SaaS platform on KCP. The objective is to translate the architectural design into running code, exercise the core tenant lifecycle paths, and surface constraints and failure modes observed in practice.

1.2.1. Objectives

[1]: *Minimal but coherent system*. Implement a minimal but coherent system consisting of dashboard frontend (FE) and backend (BE), and tenant FE and BE, including configuration flow from dashboard input to tenant rendering.

[2]: *Coherent schema*. Express tenant configuration as a shared, validated schema (TypeScript `zod`) and, where feasible, as a versioned Custom Resource Definition (CRD) intended for cross-workspace sharing via *APIBinding*.

[3]: *Reproducible setup*. Provide a reproducible local setup via scripts and manifests (clusters, deployments, ingress and networking substitutes, and debug tools).

[4]: *Workspace-oriented isolation*. Demonstrate workspace-oriented isolation using KCP concepts to the extent possible on a local developer machine.

[5]: *Evaluation*. Evaluate qualitatively against reframed criteria (functional correctness, reproducibility, isolation, operability, change management), documenting evidence and gaps.

1.2.2. Scope

The scope is explicitly confined to a prototype suitable for demonstrating concepts, not a production system. Engineering is limited to single-region, developer-workstation deployments. Observability is minimalistic. Security hardening, multi-Availability Zone (AZ) failover, automated Horizontal Pod Autoscaler (HPA) and service-level objective (SLO) enforcement, and comprehensive Role-Based Access Control (RBAC) policies are out of scope. Quantitative SLO and Key Performance Indicator (KPI) measurements with Prometheus and Grafana are deferred. Evaluation is scenario-based. Given KCP's alpha status and evolving documentation, the prototype accepts partial substitutions (e.g., KCP-Transparent Multi Cluster (TMC) sync path) where necessary to expose behavior and limitations rather than to deliver feature completeness.

1.2.3. Non-goals

End-to-End (E2E) tenant provisioning without operator intervention, multi-tenant billing, persistent data replication across clusters, advanced traffic management, and performance benchmarking at target loads are non-goals for this iteration.
Success is defined by a working demonstration of the core config → render path, reproducible setup from scripts, clear documentation of challenges (e.g., Cross-Origin Resource Sharing (CORS), ingress on local hosts, node port mapping, KCP-TMC, RBAC and APIBinding issues), and an evidence-based account of what would be required to progress toward production.

1.3. Structure of the Thesis

The thesis proceeds from foundations to design, implementation, and qualitative evaluation, closing with conclusions and supporting appendices.
Section 1 introduces the problem context, objectives, scope, and reading guide.
Section 2 consolidates fundamentals: Kubernetes and multi-tenancy, the specific capabilities of Kubernetes Control Plane (KCP) — workspaces, API, sharding, and high availability (HA) — and their relevance to SaaS.
Section 3 surveys related work on deployment strategies, scaling, cloud multi-tenancy, and prior art on KCP.
Section 4 develops the conceptual design: scenario and user journey, requirements, architecture on KCP (including config injection and workspace topology), deployment strategies, and technology choices.
Section 5 details the prototypical implementation: environment and reproducibility, templates and the shared `zod` configuration schema, the four core services (Tenant FE/BE, Dashboard FE/BE), cross-cutting concerns (configuration, security/RBAC, networking, resources), infrastructure and deployment scripts, and finally challenges, deviations, and limitations.
Section 6 presents the evaluation: a scenario-based method, reframed criteria (functional correctness, reproducibility, isolation, operability, change management), requirements coverage, lessons learned, threats to validity, and deferred metrics.
Section 7 concludes with a summary, a reflective conclusion, and an outlook.
The appendices provide implementation evidence and artifacts.
Appendix A maps repositories and where code resides.
Appendices B-G contain selected code and build/runtime excerpts for TenantFE, TenantBE, ConfigSchema, DashboardFE, DashboardBE, and Infra.
Appendix H includes documentation excerpts.
Appendix I captures Slack discussion snapshots.

Appendix J contains evaluation screenshots and logs referenced in section 6.

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 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) SLA while other tenants exceed their quotas (e.g., request rate)". As noted by Carrión 2022, p. 18 "Currently, it is difficult to achieve performance isolation for multi-tenancy on K8s clusters because this requires providing resource isolation and improving the abstraction of applications."

[5]: *Automation*. As noted by Nguyen and Y. Kim 2022, p. 651 “Presently, multi-tenant systems lack the facility of allowing clients to dynamic *[sic]*change their resources based on their business demands or create and allocate resources for new tenants. Multi-tenant system *[sic]*administrator manually does all the work of allocation or changing tenant’s *[sic]*resources.” However, to ensure efficiency and scalability, an 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*. 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 Moravcik et al. 2022, p. 457 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

The preceding concepts ground the requirements of modern SaaS: rapid change, frictionless onboarding, and a single, provider-owned control plane that scales efficiently across many tenants. Multi-tenancy increases utilization and reduces cost but raises hard problems around isolation, fair scheduling, and performance guarantees that any viable platform must address. KCP is relevant here because its workspace abstraction and control-plane virtualization promise isolation, extensibility, and shard-level scaling without abandoning Kubernetes semantics. Within this thesis, these fundamentals motivate a prototype that favors reproducibility and operability over completeness. The remainder of the document applies these ideas in design and implementation, then assesses what is feasible today and where limits appear in practice.

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 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 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 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 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 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-04	Tenant	API Latency	p95 latency for GET `/api/config` and GET `/api/reviews` shall be $\leq$ 150 ms under the same load.

NF ID	Scope	Title	Description
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*. Minimize external runtime dependencies by preferring K8s-native mechanisms. Concretely, configuration is **seeded at build time** to guarantee a sane bootstrap and then **propagated at runtime** via a versioned CRD that is exported from the provider workspace and *bound* into each tenant workspace using *APIBinding*. This avoids external stores, keeps edits scoped to workspaces, and still enables live updates.

[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. For readability the diagram abstracts away control-plane contracts. The CRD and its APIExport and APIBinding are not shown.

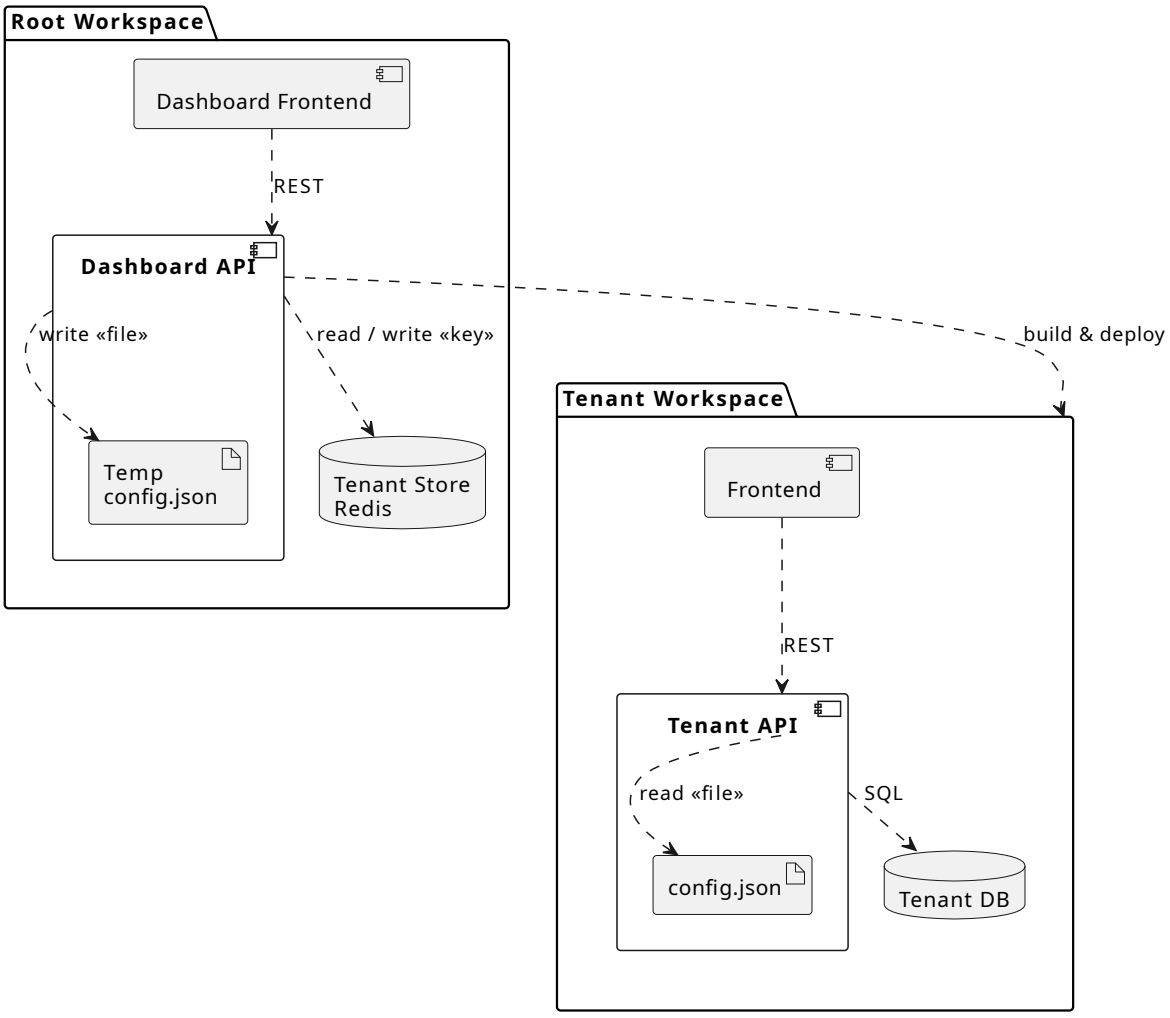

Figure 1: Overview of the system architecture

4.3.2. Rationale for Config Injection Strategy

To balance isolation, simplicity, and edit propagation, configuration delivery follows a **hybrid** approach. 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.
To propagate changes during runtime, a custom versioned CRD is used. The CRD is created inside the dashboard cluster and bound to the tenant cluster using KCP. This allows for runtime updates without redeploying the entire application while preserving strong workspace isolation.
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 persistence 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 in combination with the CRD at runtime strikes an effective balance between strong tenant isolation, operational simplicity, and runtime performance. It also provides a fallback mechanism for configuration changes without requiring a full image rebuild.

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 propagate changes via the CRD, 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-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 own 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 CRD is only bound to the specific tenant 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.

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 should deliberately rely on the built-in **CPU utilization metric** alone, contrary to the architectural best practices. 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 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.

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. 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.

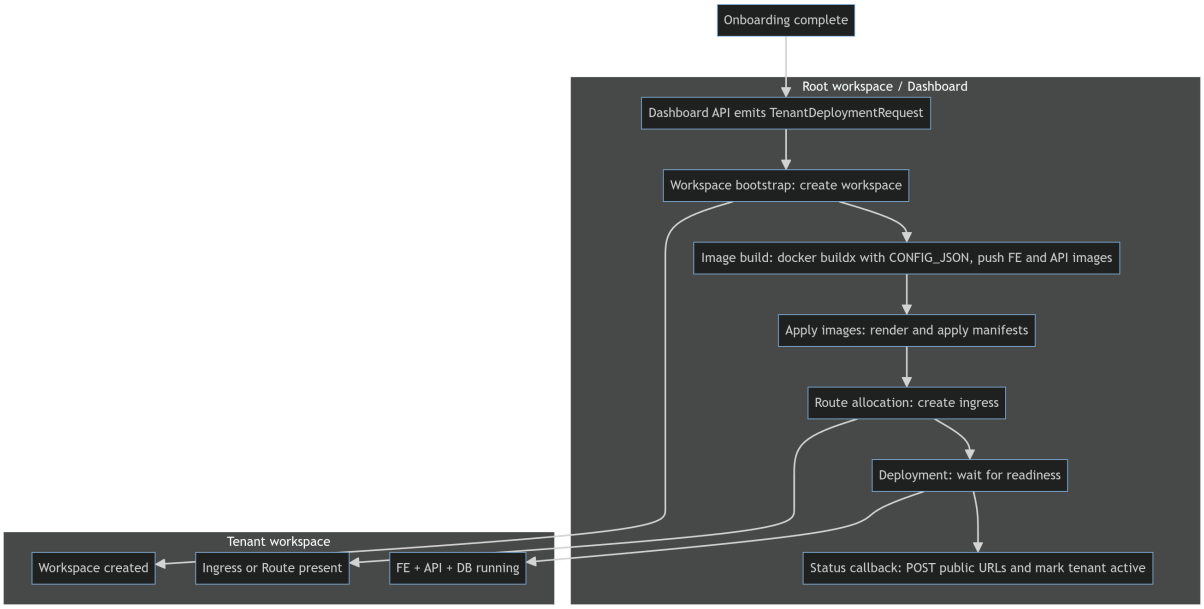

Figure 2: Initial tenant deployment pipeline

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 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. This triggers a pipeline that builds the updated image and rolls it out.

[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 creates the CRD carrying the JSON payload and binds it to the tenant workspace. 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

The continuous-deployment pipeline should deploy the updated version of the tenant code 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 pipeline 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 with the addition of the promising bleeding edge KCP project. 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 for initialization or injected into the tenant workspace at runtime via a CRD. This avoids cross-workspace communication and external storage dependencies. This enables runtime updates and a safe, clean way to provide the config to the tenant cluster for both initialization and runtime updates.

[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. Purpose and Reading Guide

This chapter explains how the architecture described in subsection 4.3 was turned into a working prototype. It documents what was implemented, how each piece maps back to the design, and why specific trade-offs were made. The Definition of Done (DoD) for the prototype was one end-to-end tenant lifecycle as described in the user journey subsubsection 4.1.1 with functional backends and frontends, container images, Kubernetes manifests and automation scripts.

How to read this chapter. First, the path from design to code is outlined, followed by a description of the environment to ensure a reproducible build. Subsequent sections cover templates and schemas, the four core services (tenant FE/BE, dashboard FE/BE), and cross-cutting concerns (configuration via CRD and APIBinding, security/RBAC, networking, resource management). The chapter then turns to infrastructure and deployment, challenges encountered, deviations from the design, and limitations. Where applicable, sections provide links to the appendix sections that contain the corresponding code snippets.

Where the code lives. All code artifacts are reproduced in the code appendices: B-G contain the directory structure, key files, and selected code snippets for the core applications, the configuration schema, and the infrastructure. Appendix A gives a quick map of repositories and their contents.

Conventions. File and directory names appear in `monospace`. Source snippets are shown as *listings* with line numbers. Long listings may span multiple pages. Figures are used for structure (e.g., directory trees).

5.2. From Design to Code

This subsection traces the concrete path from the architectural design to a working system. The implementation was structured as a sequence of vertical slices that each deliver E2E functionality, while continuously aligning with the multi-tenant architecture.

Repository and scaffold setup Separate repositories were created for each bounded component: tenant FE, tenant BE, dashboard FE, dashboard BE, the configuration schema, and the infrastructure. An overview and links to the repositories are listed in Appendix A. The concrete code is documented in Appendices B, C, E, F, D, and G. Each service was initialized with a minimal, runnable skeleton including a dockerfile.

5.3. Environment and Reproducibility

The prototype is packaged to be rebuilt and executed from a clean developer machine with minimal manual steps. The *Infra* repository (Appendix G) serves as the orchestrator: it vendors Kubernetes manifests, pins tool and image versions, and provides idempotent scripts for bootstrapping, building, and deploying all services.

Baseline toolchain A container-first setup is used to maximize reproducibility. Services are build into Docker images with explicit base-image tags. Clusters are provisioned using *kind*. Deployment is performed via checked in manifests. Required Command Line Interface (CLI) tools are installed and partly pinned by helper scripts.

Determinism safeguards Reproducibility is reinforced by:

[1]: *Tags and versions*. Explicit image tags and pinned tool versions.

[2]: *Manifests*. Declarative manifests committed to Version Control System (VCS).

[3]: *Scripting*. Shell scripts that automate the build and deployment process. Most are idempotent and can be run multiple times without side effects.

5.4. Templates and Schemas

This section introduces the code templates used to standardize the FE and BE repos in terms of project configuration. The goal is to minimize boilerplate decisions. Furthermore it introduces the configuration schema used to validate the tenant configuration and provide the data type across services.

5.4.1. Frontend Template

The frontend template provides a Next.js/TypeScript boilerplate including an `.editorconfig`, a `tsconfig.json`. Furthermore it includes code quality tooling configuration for ESLint and Prettier as well as a boilerplate multi-stage dockerfile. Both frontends, the tenant FE and the dashboard FE, are based on this template repository.
Due to its minor relevance for the thesis, the template repository contents are not reproduced in the appendix.

5.4.2. Backend Template

The backend template offers a TypeScript/Node.js (Express) baseline with the same tooling and conventions as the frontend to reduce cognitive switching and friction. It includes a `tsconfig.json`, a `.editorconfig`, ESLint and Prettier, and a boilerplate multi-stage dockerfile.
Due to the again minor relevance for the thesis, the template repository contents are not reproduced in the appendix.

5.4.3. Configuration Schema

A central `zod` schema is used to validate and version the tenant configuration. It can be imported by all services to ensure consistent data types and validation. The schema enables runtime validation with distinct error messages at service boundaries as well as a TypeScript type definition for type safety. It was published as a `npm` package to allow for easy imports and maintainability across all core services.
The same structure underlies the versioned CRD used to make the configuration available in the tenant workspace. The schema and its unit tests are shown in Appendix D.

5.5. Core Components

This section describes the four runtime services at a high level and their responsibilities within the architecture. Each of the sections below states the component's role and boundaries, its key dependencies and the contracts it exposes. Cross-cutting topics such as configuration, propagation with CRDs and APIBindings, RBAC, networking and port mappings are treated separately in 5.6 and 5.8.

5.5.1. Tenant Frontend

The tenant frontend is a small Next.js (App Router) site that renders the tenant's public page from the shared configuration and exposes a deliberate failure trigger for the tenant backend for chaos testing.
The `layout.tsx` file fetches the tenant configuration from the tenant backend's `/config` endpoint and passes it to the page component, revalidating every ten minutes. It is depicted in Listing 5. The result is stored in a react context so the root page can access it without fetching — enabling SSR. The context provider is depicted in Listing 4.
The root page uses this context and renders company details, proposition, products etc.as shown in Listing 3.
Using Incremental Static Regeneration (ISR) ensures that changes to the tenant's config become visible within at most ten minutes without a full rebuild, while still serving static HTML between refreshes.
A "crash" button issues a request to the tenant API's `/crash` endpoint to intentionally terminate that service. This allows testing the backends recovery behavior.
The dockerfile accepts the endpoint URLs as build arguments. These are baked into the frontend as environment variables at build time. It is shown under Listing 6.

5.5.2. Tenant API

A minimal Node.js/Express service that exposes exactly two endpoints and serves as the tenant's backend.
`/config` returns the tenant's configuration as a JSON object.
`/crash` deliberately terminates the service using `process.exit(1)` to support failure-injection experiments for evaluation.
The service reads a single JSON file baked into the image at build time and serves it via `/config`. The intended design was to watch and fetch the latest config from the CRD during runtime, so updates would flow through without a rebuild. That integration did not work and is discussed

in subsection 5.8. Until then, the backend operates on the immutable file included during the image build. The `server.ts` is shown under Listing 7.
The dockerfile accepts a `CONFIG_PATH` as build argument and copies the referenced file into the image to make it available for the service. This keeps the container self-contained. The dockerfile is shown under Listing 8.

5.5.3. Dashboard Frontend

A small Next.js application that provides a minimal login and a tenant-configuration form.
The login page accepts a username (no password for simplicity during prototyping) and uses a session cookie to keep track of the username. This is a placeholder for actual authentication and just implemented to allow associating a username with a tenant cluster later on. The authentication logic is provided and handled by the login page, depicted at Listing 16, the login form, depicted at Listing 20, the login action, depicted at Listing 17 and the auth lib, depicted at Listing 21.
The root page routes to the login page if no session cookie is present. Otherwise it renders the form to create a new tenant or modify an existing one (not part of the prototype due to issues discussed in subsection 5.8). The form is provided by the `ConfigForm` component, depicted at Listing 18. It uses the zod schema to safely parse and validate the input and dynamically adds input fields to add more products. On success, the form data is posted to the dashboard backend as JSON.
The dockerfile accepts the dashboard backend's `CONFIG_ENDPOINT` as build argument and sets it as an environment variable in the container. The dockerfile is shown under Listing 22.

5.5.4. Dashboard API

A lightweight Node.js/Express service that accepts tenant configurations, validates them against the zod schema, and was supposed to coordinate lifecycle events for the tenant cluster (creation and CRD updates).
The service exposes a single `/config` endpoint that accepts a JSON payload in a `POST` request and validates it against the zod schema. It reads the tenant username from the session cookie in the request header. Furthermore it persists the validated configuration as a file under `data/${user}/config.json` to allow for later retrieval and usage in the tenant lifecycle. The `server.ts` is shown under Listing 23.
The service also maintains a Redis cache to associate usernames with tenant cluster IPs. The Redis logic is shown under Listing 24.
The dockerfile does not need any build arguments in this case. It is shown under Listing 25.

The integration with the tenant lifecycle could not be implemented due to issues discussed in subsection 5.8.
The service is stateless aside from the ephemeral file drop and the Redis lookup table. Authentication is deliberately minimal to avoid out-of-scope complexity. Production hardening is discussed in subsection 5.10.

5.6. Cross-cutting Concerns

A single configuration contract (Appendix D) governs both build-time creation and runtime use. In the prototype, the dashboard API validates configs and writes a versioned file used as build context for tenant images (Appendix F). The intended runtime path, publishing the same structure as a versioned CRD and importing it via *APIBinding* during runtime, is defined under Listing 39. Limitations and the resulting fallback possibilities are discussed in subsection 5.8.
External access terminates at the dashboard ingress (Listing 41) however the ingress controller is not fully implemented in the prototype due to complexities discussed in subsection 5.8. Tenant services are discovered via the dashboard's Redis lookup. CORS is restricted to the dashboard origin due to limitations discussed in subsection 5.8. Traffic and dependencies between tenant and dashboard clusters are avoided by using file-based build injection and CRDs rather than direct calls across clusters to achieve isolation.
Validation failures are provided by zod. In the case of an CRD update being unavailable the system degrades to the last valid configuration.
All services accept their dependencies via env as build args.

5.7. Infrastructure and Deployment

The whole infrastructure is scripted to produce a repeatable demo environment following the `README` file in the *Infra* repository. Most util scripts are idempotent and can be run multiple times without side effects. The entry point is the `create-dashboard-workspace.sh` script (depicted under Listing 37) creating a KCP-TMC workspace, the dashboard cluster and the deployments for the dashboard FE and BE. The rationale for using TMC is discussed in subsubsection 5.8.5. All scripts and manifests are listed in Appendix G.
Tooling and cluster prerequisites are prepared by multiple scripts in the *Infra* repository ensuring a reproducible environment.
The layout is described in subsection G.2 and the simplified deployment workflow is shown in subsection G.3.
Service images are built using parametrized scripts as well to ease the process of building and

deploying the services.
On success the dashboard cluster is ready and the frontend is available at `dashboard.local:8080`.

5.8. Challenges, Bugs and Fixes

The prototype spans several layers (Next.js/Express services, container builds, Kubernetes on Kubernetes IN Docker (kind), KCP-TMC workspaces, and configuration via CRD and APIBinding). Along this path a mix of non-obvious defaults, complexities, version drift and underspecified behaviors surfaced. This section documents the most consequential issues, why they occurred, how they were detected, and the remedies or workaround applied. Items are grouped by theme and point to the relevant code in the appendix where applicable. Where problems remain partially resolved, the residual risks and alternatives are stated to inform the evaluation and future work.

5.8.1. CORS

Browser backend calls failed with CORS errors (blocked preflight `OPTIONS` and mismatched `Origin`). The underlying cause was a divergence of hosts and ports between frontend URLs and backend services. Attempts to “open up” CORS on the backends and to normalize hosts via ingress (as described in subsubsection 5.8.2) still left inconsistent origins during local kind testing (multiple ports, per service hostnames), so the browser refused the cross-origin requests. The effective workaround was to route traffic through server-side Next.js handlers under `/api`, so the browser sees a same-origin request and the FE service then proxies to the BE. This avoids the browsers CORS checks entirely. The proxy preserves method, headers, and body, forwards response status and headers, and reads the backend base URL from an environment variable set at build and run time.
The proxy handlers can be found in the appendix under Listing 1, Listing 2 and Listing 12.
This pattern centralizes auth/cookies and simplifies local development, but it hides the browser’s true origin from the backend, so any future direct browser to backend path must reintroduce strict CORS and a stable host schema via a gateway or ingress and possibly DNS. Once the cluster ingress is finalized, the CORS proxy can be removed or kept as a fallback for local development.

5.8.2. Ingress

Declaring an nginx ingress in Kubernetes is straightforward (cf. Listing 41 and Listing 42). The difficulty arose only on a Linux developer machine. A typical ingress controller expects to serve on the standard Hypertext Transfer Protocol (HTTP) and Hypertext Transfer Protocol Secure (HTTPS) ports `80` and `443`, which competes with existing services and requires elevated

privileges and system tweaks. In practice, this turned local setup into OS plumbing rather than application work, straying from the thesis scope.
Introducing MetalLB to gain `LoadBalancer` services on bare metal was discarded due to additional network configuration, IP range management and operational overhead not central to the thesis. Given these trade-offs, an ingress on standard ports was deferred for the local development. The system instead relies on the server-side proxy pattern described in subsubsection 5.8.1 for direct component to component calls. The browser still uses the ingress to access the frontends combined with custom hostnames in `/etc/hosts`. This keeps the development path simple and reproducible. In managed layouts (Elastic Kubernetes Service (EKS), Google Kubernetes Engine (GKE), Azure Kubernetes Service (AKS)), the intended production layout becomes trivial to enable without host-level changes.

5.8.3. End-to-End Port Mapping

Accessing services via `NodePort` surfaced several mismatches across the chain *application* → *container* → *service* → *cluster-exposed port*. The application listened on one port, the container image and deployment declared another (`containerPort`), while the `Service` used a conflicting `targetPort`/`port`, and the public `nodePort` finally exposed yet another value. The result was traffic black-holing that looked like CORS or network errors on the surface. Relevant manifests are listed in Appendix G.
A dedicated *netshoot* pod (Listing 36) was introduced to probe connectivity from inside the cluster. This quickly distinguished in-cluster routing issues from host access problems and revealed which hop in the mapping was incorrect.
The mapping was normalized to the working version in Appendix G. The debug container manifest was deliberately kept in the thesis, because it outlines a easy to deploy best practice for debugging network issues in Kubernetes.

5.8.4. Hostnames

Consistent browser access to cluster services is achieved by mapping friendly hostnames (e.g., `dashboard.local`) to a chosen IP via `/etc/hosts`. Two small idempotent helpers implement this (Listing 43 and Listing 44). This approach was aimed at easing local development and testing without having to look up IPs or ports permanently.
Therefore it is deliberately left in the thesis as a reproducible approach, that turned out to be useful for local development.

5.8.5. KCP-TMC

The initial plan relied on *glskcp* to provision clusters in workspaces. However a breaking change between KCP `v0.19` and `v0.20` invalidated that approach. The change was made in 2023, however the breaking change was documented only in the changelogs and the KCP slack channel. The stack was therefore pivoted to *KCP-glstmc* mid development. KCP-TMC itself is a plugin meant to provide syncing workspaces with *workloads* — meaning Kubernetes clusters that provide actual computation tied to an older KCP version of `v0.20` and is not actively maintained by the KCP team as of now. The `start-kcp.sh` script in Appendix G that replaced the earlier `start-kcp.sh` script reflects this pinning.
Bringing up the TMC syncer required explicit RBAC permissions and role bindings beyond the minimal examples available. The TMC documentation is confined to a single quick start guide in the `README` file of the `contrib-tmc` repository that does not cover steps required in the prototype. Several iterations of try and error debugging were necessary before finding the right syntax to apply the syncer and bind the cluster to the workspace. However, even with a clean bootstrap, attempts to bind compute via `kubectl tmc bind compute root:dashboard` as documented in Listing 37 failed due to missing KCP CRDs. Trying to apply these CRDs manually failed due to the resources being protected as documented in Appendix H.
Because a compute binding could not be established under the pinned KCP-TMC version, the design that depended on cross-workspace and more importantly cross-cluster configuration via CRD could not be completed. As an operational workaround, the configuration could be injected into the tenant's BE image at build time, however this is out of scope for this thesis, which explicitly aims to evaluate such an architecture using KCP.
The implementation of the CRD that would have been used to inject the configuration into the tenant cluster is shown in Listing 39.
A possible workaround would include custom controllers implemented in Go to manage the CRD, however this would have invalidated most of the implementation and therefore required a complete redesign of the prototype.

5.9. Deviations from Design

The prototype in multiple targeted areas:

[1]: *Control plane*. The control plane approach pivoted from *glskcp* to *KCP-glstmc* due to constraints discussed in subsubsection 5.8.5.
As a result, cross-workspace CRD-based configuration sharing was not implemented.

[2]: *Proxy*. The intended ingress controller was not implemented due to the complexities discussed in subsubsection 5.8.2. Instead, a server-side proxy was used to bridge CORS and cross-origin calls. This preserves semantics while changing the hop pattern.

[3]: *Deferred features*. Ancillary capabilities like CitusDB review storage or autoscaling with HPA were deferred because core control-plane functionality did not reach a stable baseline to build on. Where relevant fixed replicas were used as a placeholder as seen in Appendix G.

These changes are reflected in the implementation and evaluation, and are explicitly accounted for in subsection 5.10.

5.10. Limitations and Risks

The prototype intentionally narrows the scope to demonstrate E2E flow. Several constraints and risks remain:

[1]: *Control plane immaturity*. Stable `APIExport`/`APIBinding` across workspaces could not be realized with *glskcp-glstmc*. Configuration propagation via CRD is therefore no part of the running system (see subsubsection 5.8.5).

[2]: *Local exposure model*. Services are exposed via `NodePort` `/etc/hosts` entries instead of a proper ingress/Transport Layer Security (TLS) setup (see subsubsection 5.8.2). The result is a brittle developer setup without HTTPS.

[3]: *CORS workaround*. Browser $\rightarrow$ BE calls are proxied through Next.js `/api` routes to avoid CORS (see subsubsection 5.8.1). This way the FE server becomes a choke point and the additional hop adds latency and operational coupling.

[4]: *Configuration freshness*. The tenant BE reads a file baked at build time and runtime updates are not pulled from a CRD. FE revalidation is periodic. This risks stale configuration and a rebuild and redeploy are required to propagate changes.

[5]: *Security*. The prototype login omits passwords and tokens to avoid additional complexity. Inter-service traffic is therefore unauthenticated inside the cluster and there is no TLS termination. This setup is not suitable for production use due to attack surface via `NodePort`.

[6]: *Availability and scaling*. The prototype uses single-replica deployments and has no scaling mechanism or readiness/liveness hardening beyond defaults. This limited fault tolerance and unpredictable throughput make it unsuitable for production.

[7]: *Observability*. The prototype does not implement centralized logging, metrics, or tracing. Debugging relies on manual inspection and local logs. This limits operational insights and complicates incident response.

[8]: *State and persistence*. The Redis lookup for tenant cluster IPs is a simple, non-hardened cache. This risks data loss across restarts.

[9]: *Testing*. Automated tests are limited to basic unit tests for the zod schema. Integration tests and E2E tests are not implemented. This may cause regressions across service boundaries to go undetected.

[10]: *Version pinning and ecosystem*. The stack depends on specific tool versions and KCP-TMC is not actively maintained. This would inadvertently lead to compatibility issues with future updates.

These limitations inform the evaluation scope and the suggested future work.

5.11. Summary and Link to Evaluation

This chapter translated the architecture into a (partly) runnable prototype. Four services (tenant FE, tenant BE, dashboard FE and dashboard BE), a shared schema, and infrastructure that provisions workspaces and clusters to deploy the system were implemented. Cross-cutting concerns like the configuration flow, RBAC, networking, expose, and resource settings were addressed with pragmatic choices suited to a local, reproducible setup. Known deviations and limitations delimit what the prototype can demonstrate.
The next chapter (section 6) evaluates the prototype along the axes implied by this implementation.

6. Evaluation

6.1. Introduction and Scope

Planned evaluation targeted SLOs and metrics (latency, availability, error rate, resource use) via Prometheus/Grafana. Due to prototype constraints, the approach was reframed to a qualitative, scenario-based evaluation focused on correctness, reproducibility, isolation, and operability. The goal is to assess whether the implemented paths work as specified and to extract lessons that inform future iterations.

6.2. Method (Scenario-based)

Given incomplete E2E automation, the evaluation adopts a concise, scenario-based method that exercises the working paths and documents known gaps. Each scenario defines preconditions, steps, expected outcomes, and a pass/partial/fail verdict, with minimal artifacts (screenshots, command logs, and code references) captured in the appendices. For each scenario, the preconditions are set, the steps are defined and executed, expected outcomes are stated and a pass/partial/fail verdict is given with minimal artifacts in Appendix J. The artifacts are cited in the evaluation below, but the full details are deferred to the appendix to keep the main text concise.

[1]: *Dashboard submission*. *Preconditions:* dashboard cluster is running, dashboard FE and BE are deployed, and the dashboard FE is accessible at `dashboard.local:8080`.
Steps: open the dashboard FE, authenticate with a username, submit a tenant configuration via the form.
Expected: `2xx` response code and a config file written to the dashboard BE containers `data` directory.
Evidence: Evidence for the successful submission is found in subsection J.1 and subsection J.2.
Verdict: Pass.

[2]: *Tenant rendering (manual cluster)*. *Preconditions:* A tenant cluster created manually via scripts. The config file needs to be baked into the tenant BE image. The tenant FE needs to be deployed with the backend endpoint configured.
Steps: Open the tenant FE at `tenant.local:8001`. Verify that the shown details match the backend config.
Expected: The tenant FE renders the company details, products, and proposition from the config file.

Evidence: Evidence for the successful rendering is found under subsection J.3 and subsection J.4.
Verdict: Pass.

[3]: *CRD creation*. *Preconditions:* See *Dashboard submission*.
Steps: See *Dashboard submission*.
Expected: CRD is created and bound to the tenant cluster.
Evidence: As the CRD creation was not successful, the CRD is not automatically created.
Verdict: Failed.

This method aligns the evaluation with the implemented surface and provides screenshots as evidence for each scenario while acknowledging non-functional instrumentation and full automation as deferred work.

6.3. Evaluation Criteria (Reframed)

This section defines the qualitative criteria used to judge the prototype in lieu of full SLO/metric instrumentation. Each criterion maps to the scenarios defined in subsection 6.2.

6.3.1. Functional Correctness

Functional behavior is judged against the scenario set n subsection 6.2 without explicitly reciting the preconditions, steps and expected outcomes.

[1]: *Dashboard submission*. The dashboard path behaves as specified. Well formed configurations are accepted with `2xx` responses and written to the dashboard BE data directory. Malformed configurations are rejected by the frontend (see Figure 27) as well as by the backend (see Figure 29).

[2]: *Tenant rendering (manual cluster)*. The tenant is correctly rendered by the tenant FE based on the configuration baked into the tenant BE. The tenant FE correctly fetches the configuration from the backend and renders the company details, products and proposition (see subsection J.3). Updates to the tenant configuration can not be reflected due to the missing CRD integration.

[3]: *CRD creation*. The CRD creation is not successful due to the issues discussed in subsection 5.8. The tenant BE does not read the configuration from the CRD and therefore does not reflect updates to the configuration. The tenant FE does not reflect updates to the configuration either.

6.3.2. Reproducibility

Reproducibility is judged by whether a third party can recreate the working paths from clean sources and obtain the same outcomes as in item Challenge 1 and item Challenge 2. The infrastructure repository provides shell scripts that install dependencies, set up KCP-TMC, create workspaces/clusters, deploy services, and standardize local hostnames via `add-to-hosts.sh` and `cleanup-hosts.sh` (Appendix G). Most scripts are idempotent and can be rerun safely. Dockerfiles pin build-time arguments and embed the tenant config to minimize drift (cf. Sections subsection 5.3, subsection 5.4). Under these conditions, two outcomes are reproducible. First a dashboard submission persists a validated configuration in the backend's `data/` directory and second a manually created tenant stack renders that configuration via `GET /config`. Note that due to lack of automation the tenant stack might need some manual tinkering outside the provided `create-tenant-x-cluster.sh` script found in Listing 31.
The remaining gaps are mostly due to the problems discussed in subsection 5.8. The local ingress on the developer machine also reduces portability due to the `/etc/hosts` entries as discussed in subsubsection 5.8.4. The flaky setup is mitigated by node ports and static host mappings but still requires manual attention. Overall the prototype is *mostly reproducible*. The provided scripts reliably produce the dashboard cluster and and most of the tenant stack, while CRD-based propagation and automated tenant provisioning remain dysfunctional.

6.3.3. Isolation

Isolation is considered along three axes: failure, data, and (rough) performance isolation. The intended design allocates a dedicated workspace and cluster per tenant. In the prototype this is approximated by manual per-tenant clusters plus a separate dashboard cluster (Appendix G) and dedicated workspaces. This topology already yields strong blast-radius reduction: crashing the tenant backend via `/crash` affects only that tenant's pods and does not impact the dashboard stack or other clusters by definition.
Data isolation is simplified by avoiding shared state across tenants: the tenant backend serves a baked config file, and the dashboard backend writes configs to its own container-local `data/` directory (subsection J.2). No shared database is present, so there is no cross-tenant read path. Planned CRD / APIBinding–based distribution would reintroduce a shared control-plane surface. Correct scoping and read-only access would be required to preserve isolation.
Network isolation is acceptable for a local prototype: clusters are distinct, node ports plus static host mappings expose only the necessary services, and there is a dedicated ingress for every cluster. However, within a cluster no `NetworkPolicy` is enforced, and pod security or admission hardening is not configured. Thus, isolation relies primarily on "separate clusters" rather than

defense-in-depth controls in this prototype.
Performance isolation is only partial. Without `requests` or `limits`, quotas, or HPA, pods can contend for node resources. Multiple local clusters still share the same host. In a managed environment, per-tenant clusters would be scheduled to separate nodes or pools to improve this dimension.
Failure and data isolation meet prototype goals. Network and performance isolation are incomplete and would require `NetworkPolicy`, pod security and admission, resource quotas or limits, and autoscaling to reach production expectations.

6.3.4. Operability

Operability is assessed along day-2 tasks: deploy, observe, troubleshoot, and change safely. Deployment is largely script-driven and idempotent (Appendix G), which reduces setup friction, but several steps remain manual (e.g., tenant cluster creation and image rebuilds). In particular, configuration changes require rebuilding the tenant backend image because runtime distribution via CRD/APIBinding is not functional. This in turn limits agility.
Observability is minimal. Services log to stdout/stderr and are inspected via `kubectl logs`. There is no central log aggregation, metrics, or tracing, and no liveness or readiness probes are defined for the HTTP endpoints (`/config`, `/crash`). As a result, failure detection relies on manual checks rather than automated health signals.
Troubleshooting workflows are workable but manual. The `netshoot` debug pod is effective for diagnosing node port and service wiring issues (see subsubsection 5.8.3). Host modification scripts standardize access from the browser. CORS issues were mitigated by proxying through Next.js server routes, which simplifies local operations but couples the FE to BE routing (subsubsection 5.8.1).
Security practices and RBAC are permissive for development: components run under broad privileges, and there is no secret management beyond environment variables. This is acceptable for a prototype but increases operational risk in production settings.
Basic operability is achieved (repeatable deploys, workable debugging), but automated health, centralized observability, robust RBAC and secrets, and runtime config rollout are missing. These gaps elevate operational toil and slow incident response.

6.3.5. Change Management

Change management is evaluated in terms of validation, versioning, propagation, safety, and rollback. Validation is strong: tenant configurations are checked at submission in the dashboard FE and re-validated in the dashboard API against the shared `zod` schema (subsubsection 5.4.3,

subsubsection 5.5.3, subsubsection 5.5.4). However, runtime propagation is limited because CRD/APIBinding-based distribution is non-functional in the prototype, the Tenant BE reads a file baked at image build time. Changes therefore require rebuilding and redeploying the tenant BE image. The tenant FE's 10-minute revalidation only reflects new data once the BE serves it, so update latency is dominated by rebuild and redeploy time rather than cache expiry.
Versioning exists for the schema (TypeScript type tied to `ConfigSchema`), but there is no persisted history of applied configs, no semantic version carried with each config instance, and no automated downgrade or rollback beyond manually redeploying a previous image. Safety mechanisms are basic. Schema validation prevents malformed updates, and tenant isolation bounds blast radius to a single tenant's components, but there is no staged rollout (e.g., canary), no admission control, and no audit trail.
Overall the prototype partially meets change-management goals. Validation and isolation are in place, but runtime updates, controlled propagation, auditability, and rollbacks are manual or missing. Enabling CRD-backed distribution with a reconciler, attaching explicit config versions, and adding staged deploys and history would close the gap in future iterations.

6.4. Requirements

This section assesses the prototype against the requirements defined in subsection 4.2. Each requirement is judged qualitatively (e.g., met, partially met, not met) with brief justification and references to implementation evidence.

6.4.1. Functional Requirements

[1]: *F-01*. This requirement is **partially met**. Creation of configurations is implemented via the dashboard form. The dashboard API validates the payload against the shared `zod` schema and persists it as a JSON file. However updating via CRD or deletion are not implemented. Furthermore logo upload was omitted in the prototype.

[2]: *F-02*. This requirement is **partially met**. The list of services and products is carried as the `products` array inside the same config object. As above, updates to the configuration are not reflected in the tenant FE or BE.

[3]: *F-03*. This requirement is **not met**. No ratings domain (API, storage, UI) was implemented. This was deferred to focus on core provisioning and configuration paths.

[4]: *F-04*. This requirement is **met** on a manual path. The tenant frontend fetches `/config` from the tenant backend and renders the company profile, products, and proposition. ISR revalidates roughly every ten minutes. A deliberate failure path is exposed via the `/crash` endpoint for chaos testing. This holds when the tenant cluster is created manually.

[5]: *F-05*. This requirement is **not met**. Automated E2E provisioning from the dashboard could not be realized due to the pivot from KCP to KCP-TMC and ensuing syncer, RBAC and APIBinding issues as discussed in subsubsection 5.8.5. Only manual scripts and clusters were used during evaluation.

[6]: *F-06*. This requirement is **met**. The dashboard API validates the config, writes it to disk, and the tenant backend image is built with that file baked in. The backend serves it via `/config`. The planned CRD-based shared store was not operational, so runtime edits require rebuild and redeploy.

[7]: *F-07*. This requirement is **not met**. The tenant backend currently reads the baked JSON file and does not maintain an in-memory cache with TTL nor a file-fallback strategy. In short: the “happy path” for submitting a config and rendering it in a manually provisioned tenant is demonstrated. Automated provisioning, ratings, and runtime config management and caching remain open.

6.4.2. Non-functional Requirements

[1]: *NF-01*. This requirement was **not evaluated**. The prototype runs on a local kind cluster without Prometheus, Grafana or external uptime probes, so monthly availability and “0 failed GET during workspace moves and pod restarts” cannot be measured. Ad-hoc tests show the tenant page serves correctly when the tenant cluster is up. However, the absence of automated rollout and health checks mean the stated SLO cannot be asserted. Meeting this target would require basic liveness and readiness gates, rolling updates, and external probing plus time-window aggregation (see subsection 6.8).

[2]: *NF-02*. This requirement was **not evaluated**. As with NF-01, no continuous probing or log-based Service Level Indicator (SLI) exists. The dashboard is a single instance without redundancy or auto-healing, so even short local outages would count against the SLO. To substantiate this requirement, the system would need at least replicated dashboard pods, persistent state for submitted configs, and an availability SLI derived from probe success ratio over the month (see subsection 6.8).

[3]: *NF-03*. This requirement was **not evaluated**. No load testing was performed. Only spot checks via browser dev tools are feasible within the time window. The target performance is ambitious for a single-node kind cluster. The ISR and revalidation model reduces backend reads, which helps latency under load, but without Content Delivery Network (CDN) or edge caching, autoscaling, and measured SLIs, the requirement cannot be claimed. Future validation would need a load generator, request and latency histograms, and resource scaling with HPA, VPA or KEDA to sustain 200 req/s while keeping p95 under the budget.

6.5. Results and Evidence

This section aggregates the outcomes of the scenario-based evaluation without reprinting artifacts. Detailed logs and screenshots are indexed in Appendix J.

6.5.1. Summary

The dashboard accepts and validates tenant configurations and persists them (Scenario Challenge 1, *Pass*). A manually provisioned tenant renders the submitted data E2E (Scenario Challenge 2, *Pass*). Automated CRD-based propagation and binding to tenant workspaces remain non-functional (Scenario Challenge 3, *Fail*), which blocks the fully automated path from submission to deployed tenant.

Evidence map.

6.6. Lessons Learned

Several design and implementation insights emerged during the prototype and are listed below.

[1]: *Solve browser-to-backend early*. Direct browser calls repeatedly failed due to CORS. Moving requests behind server-side proxy routes in Next.js (`/api/...`) eliminated policy mismatches and simplified endpoint management (see subsubsection 5.8.1).

[2]: *Local ingress is fragile on developer machines*. Nginx ingress is trivial to declare in K8s but awkward locally due to privileged ports and host networking. For developer workflows, `NodePort` + host mapping proved as a workaround (see **??**, **??**).

[3]: *Make port paths explicit*. Mismatches across app → container → service → cluster ports were the root cause of several days of debugging. A dedicated debug pod (`netshoot`) should be part of the baseline toolchain (see **??**).

[4]: *Schema-first helps across services*. The shared `zod` schema provided one contract for validation and types, reducing drift and clarifying failure modes at service boundaries.

[5]: *Decouple "config as data" from control plane coupling*. Version-and-share via a CRD remains attractive, but binding across workspaces with KCP-TMC was brittle on the available versions. Keeping build-time injection and runtime contracts independent avoided a hard blocker for the base functionality (see subsubsection 5.8.5).

[6]: *Idempotent scripts pay off*. Infrastructure scripts written to be repeatable and environment-aware shortened recovery time and supported the evaluation's reproducibility goals.

[7]: *Observability first is not optional*. Lack of metrics and traces slowed diagnosis. Basic Prometheus counters and request tracing hooks should be part of the initial scaffolding, not deferred work.

[8]: *Documentation is crucial*. KCP provides a intriguing approach, but its complexity necessitates thorough documentation, that is not provided in all necessary forms. Because of its alpha status breaking changes are introduced and easily available resources often did not reflect the current state of the project. The combination of hard-to-grasp concepts and missing reference code and architectures made the project harder to implement than it needed to be. As an example for a pain point in the TMC documentation, the official quick start guide is provided in Appendix H.

6.7. Threats to Validity

The following factors limit generalizability:

[1]: *Environment bias*. Evaluation was conducted on a single-machine with local clusters. results may not transfer to managed clouds with load balancers, DNS and storage classes.

[2]: *Manual steps*. Several scenarios required manual provisioning. Hidden assumptions or side effects may affect outcomes despite the scripts' intent to be idempotent.

[3]: *Incomplete instrumentation*. Absence of runtime metrics and tracing reduces confidence in negative findings (e.g., "no error" vs. "no observed error").

[4]: *Control plane drifts*. The prototype relies on specific KCP-TMC versions and plugins. Behavior will inevitably change with newer releases.

[5]: *Single implementation path*. Templates and services reflect one tech stack (TypeScript + Next.js + Express). Alternative stacks may surface different constraints.

6.8. Planned Metrics (Deferred)

The following metrics were originally planned, but deferred due to time and tooling gaps:

[1]: *Availability and error budget*. Uptime for public tenant pages (NF-01) and dashboard (NF-02) as well as error budget burn based on `5xx`/`4xx` rates from synthetic probes.

[2]: *Latency and throughput*. p50/p95/p99 for key endpoints (`/config`, tenant page render) under load. Target p95 $\leq$ 600ms at 200req/s (NF-03).

[3]: *Provisioning time*. E2E time from submission to tenant URL readiness (F-05), including CRD propagation lag and image build and pull time.

[4]: *Resource efficiency*. CPU and memory per service at idle and under burst and cache hit ratio for configuration (F-07).

[5]: *Intended tooling*. Service-level Prometheus metrics (HTTP, Garbage Collection (GC), cache), Grafana dashboards, Gatling load tests, blackbox-exporter probes, and basic OpenTelemetry traces.

6.9. Summary and Link to Conclusion

The evaluation confirms that configuration submission, validation, persistence, and tenant rendering work as intended when the tenant stack is provisioned manually. The automated path via CRD creation and cross-workspace binding remained non-functional, and observability was insufficient for quantitative assessment. These findings frame the priorities for the next iteration, stabilizing the control-plane integration, reinstating automated provisioning, and adding first-class telemetry, which are discussed in the Conclusion and Outlook.

7. Conclusion and Outlook

This chapter synthesizes the implementation and evaluation results, distills the main findings relative to the project goals, and identifies the practical implications of the prototype. The summary recaps what was built and what worked. The personal conclusion reflects on the significance and limitations of those results. And finally, the outlook outlines concrete next steps to mature the system toward a production-ready state.

7.1. Summary

The project set out to demonstrate a tenant-aware SaaS foundation on Kubernetes with workspace-scoped isolation, a shared configuration model, and “push-button” tenant provisioning. A partly working prototype was delivered comprising four services (tenant frontend and backend, and dashboard frontend and backend), a shared TypeScript plus `zod` configuration schema, and a set of (mostly idempotent) scripts and manifests to stand up the local environment. The dashboard accepts and validates tenant configurations, persists them as files, and the tenant stack, when built and deployed manually, renders the same configuration in the public page. Practical issues around browser CORS were resolved by proxying through server-side Next.js routes, and reproducibility was improved via templates, dockerfiles, and scripted setup. Key design promises remain only partially realized. Automatic workspace and cluster creation from the dashboard is not wired E2E. The intended CRD-based distribution and versioning of tenant configuration was not made to work reliably, so the tenant backend reads a file baked at image build time, and production-grade observability with metrics and SLOs was deferred. Local networking constraints (node ports, ingress on privileged ports) further limited an E2E path on a developer machine.

A major factor in these outcomes was substrate volatility: **KCP is still an alpha-stage project** and underwent breaking shifts (e.g., from v0.19 to v0.20) with sparse, fast-moving documentation. The switch to `KCP-TMC` (itself based on an older KCP and lightly maintained) introduced additional friction, RBAC for the syncer, compute binding hurdles, and protected core CRDs that complicated APIBinding plus APIExport usage. These ecosystem constraints, rather than fundamental flaws in the service design, were the primary blockers.

Overall, the prototype validates the feasibility of the configuration model and the tenant surface but stops short of automated provisioning and runtime configuration flows. The evaluation and lessons learned identify a clear path to completion once the control-plane substrate is stabilized and the CRD-based pipeline is made reliable.

7.2. Personal Conclusion

The experience with KCP and KCP-TMC was shaped less by formal documentation and more by community support: the Slack workspace routinely provided the clearest, most actionable guidance, especially around the core concepts behind KCP and TMC and why it was moved to its own project, where official documentation was fragmentary in terms of comprehensiveness and clear implementation details as well as breaking changes. As a platform idea, the project looks promising: a shared control plane that offers workspaces, APIBinding, APIExport, and syncers could, in theory, standardize multi-tenancy concerns and reduce bespoke glue.
In practice, however, the stack tended to introduce rather than hide complexity. The concepts and interaction between workspaces, exports, bindings, syncers, the KCP core, TMC and the newer multicluster-runtime (MCR) in combination with custom RBAC and CRDs created a steep learning curve. It was often especially unclear when to apply which concept, and where the limitations were. Combined with fast-moving releases and no E2E examples, the system quickly becomes a black box for anyone who is not a senior Kubernetes practitioner or active contributor. Much of the engineering effort went into deciphering substrate behavior rather than delivering tenant features.
The underlying concept, decoupling APIs from clusters and composing them per workspace, remains compelling. But given the current maturity, scarce examples, and operational sharp edges, it should be observed rather than adopted as the foundation described in this thesis. A pragmatic near-term path would favor better-known patterns (namespace-level isolation with strict RBAC, NetworkPolicy, ResourceQuota, GitOps-driven rollout, or established provisioning controllers) while tracking KCP's trajectory. If the ecosystem stabilizes and documentation catches up with the design, the approach may become a viable core for multi-tenant platforms. Until then, it is best treated as an experimental avenue to monitor or a concept to apply in edge-case scenarios.

7.3. Future Outlook

The short-term priority is to harden the prototype, reconnect the missing paths, and add observability so that a metric-based evaluation becomes feasible. The immediate focus should be to close the loop from dashboard to tenant: replace the broken CRD path with a pragmatic delivery channel (for example, a Git-backed configuration repository with a CI trigger, or object storage with a webhook) so that a submitted configuration deterministically builds a tenant image and deploys to a target cluster. The external interface should remain stable so that a later swap to CRD plus APIBinding is possible without impacting clients. Networking should be standardized for local development by adopting node port with nginx ingress or, if required,

adding MetalLB with a fixed address pool. A single source of truth for ports should be defined and encoded in manifests and scripts. Observability should be introduced via Prometheus, Grafana and synthetic probes. Frontends and backends should expose Rate, Error, Duration (RED) metrics, with logs shipped to Loki and traces emitted through OpenTelemetry to enable the deferred SLO evaluation. Given past issues, the server-side proxy from frontend to backend should remain the default while CORS policies for a future ingress-only model are documented. Reproducibility should be strengthened by adding smoke tests to CI.

Mid-term, configuration should be treated explicitly “as an API”. A renewed attempt at “config as CRD” should be made only behind a narrow adapter so that the rest of the system remains decoupled. Where KCP is not required, the platform can emulate “workspaces” with mature primitives (namespaces, ResourceQuota, NetworkPolicy, and hierarchical namespace controller), provision clusters via Cluster API or Crossplane, and manage rollout using Argo CD or Flux. Multi-tenant hardening should introduce pod security admission, per-tenant quotas and limits, and network isolation, together with tenancy E2E tests. Operational readiness should be improved with concise runbooks for build failures, rollout rollback, configuration drift, and cluster replacement.

Longer-term, KCP and KCP-TMC should be re-evaluated periodically. Adoption gates should include stable and documented APIExports for the required resources, clear RBAC patterns for syncers, and a credible migration plan from the baseline adapter. With instrumentation in place, scalability can be addressed by introducing HPA, VPA or KEDA and queue-based autoscaling where applicable, validating behavior under controlled load and revisiting SLOs accordingly. As functionality expands (for example, ratings or service catalogs requiring persistence), the data plane should incorporate a managed database with backup/restore and multi-tenant isolation patterns.

Across all horizons, the migration posture should keep KCP-specific logic behind adapters and maintain two modes: a baseline mode (namespaces + GitOps) that works today and a KCP mode that can be enabled as risks decrease. This limits lock-in, preserves near-term progress, and keeps the architecture aligned with an evolving ecosystem.

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.

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

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).

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

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

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

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

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.
Furthermore all unicode characters in code listings have been replaced with `<unicode-meaning>` to avoid encoding issues in the Portable Document Format (PDF).

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)

B.1. 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 3: TenantFE directory tree (trimmed)

B.2. Key Files

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

```ts
'use server'

import { 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() {
    try {
        const response = await fetch(endpoint, {
            method: 'GET'
        });
        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 });
    }
}
```

Why Work around browser CORS by calling the service server-side.

What Forwards a `GET` to `CONFIG_ENDPOINT` read from environment variable.

Role CORS-bridging proxy route.

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 });
    }
}
```

Why Work around browser CORS by calling the service server-side.

What Forwards a `GET` to `CRASH_ENDPOINT` read from environment variable.

Role CORS-bridging proxy route.

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}>
```

```
40                  <h2 className={styles.addressHeader}>Address</h2>
41                  <address>
42                      {cfg.address?.street} {cfg.address?.streetNumber} <br/>
43                      {cfg.address?.zipCode} {cfg.address?.city} <br/>
44                      {countryName()}
45                  </address>
46              </div>
47              <div className={styles.aboutDiv}>
48                  <h2 className={styles.aboutHeader}>About us</h2>
49                  <p>{cfg.about}</p>
50              </div>
51              <footer>
52                  <button className='crash' onClick={() => {
53                      fetch(`/crash`)
54                          .then(() => {alert("Crash triggered")})
55                          .catch(() => alert("Backend not reachable"))
56                  }}>
57                      Crash
58                  </button>
59              </footer>
60          </main>
61      )
62  }
```

Why Tenant landing page driven by runtime config. Includes a safe crash trigger for testing.

What Client component reads config via `useConfig` hook, formats the country with `Intl.DisplayNames`, conditionally lists products, renders address/about, and adds a footer button that calls `/crash` and alerts on success/failure.

Role Presentation layer for the tenant frontend.

Listing 4.: Config provider: `src/lib/ConfigProvider.tsx`

```tsx
'use client'

import React, { createContext, useContext } from 'react';
import  type { Config } from '@mystiker123/config-schema';

const ConfigCtx = createContext<Config | null>(null);

export function useConfig(): Config {
    const ctx = useContext(ConfigCtx);
    if (!ctx) throw new Error('useConfig must be inside <ConfigProvider />');
    return ctx;
}

export default function ConfigProvider(
    { cfg, children }: { cfg: Config; children: React.ReactNode },
) {
    return <ConfigCtx.Provider value={cfg}>{children}</ConfigCtx.Provider>
}
```

Why Single source of truth for tenant runtime config. Consumed by root page.

What Defines a React context for `Config`, exposes `useConfig()` that throws if used outside `<ConfigProvider>`, and provides `cfg` to descendants.

Role Configuration layer for the tenant frontend. Bridges the schema package to the UI tree via React Context.

B. TenantFE (Selected Code)

Listing 5.: Layout: `src/app/layout.tsx`

```tsx
import './globals.css';
import type { Metadata } from 'next';
import { ConfigSchema } from '@mystiker123/config-schema';
import ConfigProvider from '@/lib/ConfigProvider';

export const dynamic = 'force-dynamic'; // Force dynamic rendering to avoid fetching config at build time

async function loadConfig() {
    const base = process.env.INTERNAL_BASE_URL || 'http://127.0.0.1:3000';
    const url = new URL('/api/config', base).toString();  // <- absolute to proxy (server side component)
    const res  = await fetch(url, { next: { revalidate: 60*10 } });
    if (!res.ok) throw new Error('Failed to load config');
    const json = await res.json();
    const parsed = ConfigSchema.parse(json);
    return parsed;
}

export const metadata: Metadata = { title: 'Tenant FE' };

export default async function RootLayout(
    { children }: { children: React.ReactNode },
) {
    const cfg = await loadConfig();

    return (
        <html lang="en">
            <body>
                <ConfigProvider cfg={cfg}>
                    {children}
                </ConfigProvider>
            </body>
        </html>
    );
}
```

Why Ensure every page uses validated, fresh tenant config at runtime — no stale build-time values.

What Sets `dynamic='force-dynamic'`, fetches `/config` (revalidating every 10 minutes), validates via `ConfigSchema`, wraps children in `<ConfigProvider>`, and defines page `metadata`.

Role App composition root that bridges the config proxy to the UI via React Context.

B.3. Build and Runtime

Listing 6.: Dockerfile

```
# syntax=docker.io/docker/dockerfile:1

FROM node:18-alpine AS base

# Install dependencies only when needed
FROM base AS deps
# Check https://github.com/nodejs/docker-node/tree/b4117f9333da4138b03a546ec926ef50a31506c3#nodealpine to understand why libc6-compat might be needed.
RUN apk add --no-cache libc6-compat
WORKDIR /app

# Install dependencies based on the preferred package manager
COPY package.json yarn.lock* package-lock.json* pnpm-lock.yaml* .npmrc* ./
RUN \
  if [ -f yarn.lock ]; then yarn --frozen-lockfile; \
  elif [ -f package-lock.json ]; then npm ci; \
  elif [ -f pnpm-lock.yaml ]; then corepack enable pnpm && pnpm i --frozen-lockfile; \
  else echo "Lockfile not found." && exit 1; \
  fi


# Rebuild the source code only when needed
FROM base AS builder
WORKDIR /app
COPY --from=deps /app/node_modules ./node_modules
COPY . .

# Build ARG for CONFIG_ENDPOINT and CRASH_ENDPOINT
ARG CONFIG_ENDPOINT
ARG CRASH_ENDPOINT
ENV CONFIG_ENDPOINT=$CONFIG_ENDPOINT
ENV CRASH_ENDPOINT=$CRASH_ENDPOINT

# Next.js collects completely anonymous telemetry data about general usage.
# Learn more here: https://nextjs.org/telemetry
# Uncomment the following line in case you want to disable telemetry during the build.
# ENV NEXT_TELEMETRY_DISABLED=1
```

```
37
38  RUN \
39    if [ -f yarn.lock ]; then yarn run build; \
40    elif [ -f package-lock.json ]; then npm run build; \
41    elif [ -f pnpm-lock.yaml ]; then corepack enable pnpm && pnpm run build; \
42    else echo "Lockfile not found." && exit 1; \
43    fi
44
45  # Production image, copy all the files and run next
46  FROM base AS runner
47  WORKDIR /app
48
49  ENV NODE_ENV=production
50  # Uncomment the following line in case you want to disable telemetry during runtime.
51  # ENV NEXT_TELEMETRY_DISABLED=1
52
53  RUN addgroup --system --gid 1001 nodejs
54  RUN adduser --system --uid 1001 nextjs
55
56  COPY --from=builder /app/public ./public
57
58  # Automatically leverage output traces to reduce image size
59  # https://nextjs.org/docs/advanced-features/output-file-tracing
60  COPY --from=builder --chown=nextjs:nodejs /app/.next/standalone ./
61  COPY --from=builder --chown=nextjs:nodejs /app/.next/static ./.next/static
62
63  USER nextjs
64
65  EXPOSE 3000
66
67  ENV PORT=3000
68
69  # server.js is created by next build from the standalone output
70  # https://nextjs.org/docs/pages/api-reference/config/next-config-js/output
71  ENV HOSTNAME="0.0.0.0"
72  CMD ["node", "server.js"]
```

Why Reproducible, minimal image for the tenant UI. Reduced attack surface and env-specific endpoints baked at build time.

What Multi-stage build (`deps`, `builder`, `runner`). Installs deps from a lockfile, sets `CONFIG_ENDPOINT`/`CRASH_ENDPOINT` via build args, builds Next.js standalone, copies `.next/standalone` and `.next/static`, runs as non-root `nextjs` on port 3000 via `server.js`.

Role Container artifact of the Tenant FE. Deployed in the tenant Cluster.

C. TenantBE (Selected Code)

C.1. Directory Structure

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

Figure 4: TenantBE directory tree (trimmed)

C.2. Key Files

Listing 7.: Server: `src/server.ts`

```ts
import express, { Request, Response } from 'express';
import { readFileSync } from 'fs';
import NodeCache from 'node-cache';
import { Handler } from 'express-serve-static-core';

/* ---------------------------------------------------------------- */
/* cache setup                                                      */
/* ---------------------------------------------------------------- */

const cache = new NodeCache({
    stdTTL: 0,
    checkperiod: 300,
});

const CONFIG_TTL = 60 * 60 * 24;

/* ---------------------------------------------------------------- */
/* express app                                                      */
/* ---------------------------------------------------------------- */

const app = express();
app.use(express.json());

const getConfig: Handler = async (req: Request, res: Response): Promise<void> => {
    const hit = cache.get('config');
    if (hit) {
        res.json(hit);
        return;
    }

    try {
        const rawConfig = readFileSync('config.json', 'utf-8');
        const parsed = JSON.parse(rawConfig);

        cache.set('config', parsed, CONFIG_TTL);
        res.json(parsed);
    } catch (e) {
```

```
38          console.error('[config] load failed:', e);
39          res.status(500).json({ error: 'Failed to load configuration' });
40      }
41  }
42
43  const crash: Handler = async (req: Request, res: Response): Promise<void> => {
44      res.status(200);
45      process.exit(1);
46  }
47
48  app.get('/config', getConfig);
49  app.get('/crash', crash);
50
51  /* ------------------------------------------------------------------ */
52  /* start server                                                       */
53  /* ------------------------------------------------------------------ */
54
55  const PORT = Number(process.env.PORT ?? '8080');
56  const HOST = process.env.HOST ?? '0.0.0.0';
57  app.listen(PORT, HOST, () => {
58      /* eslint-disable no-console */
59      console.log(`<unicode-rocket> API ready on http://${HOST}:${PORT}`);
60  });
```

Why Stable config source for the Tenant UI plus a controlled crash hook to test failure handling and orchestration.

What Express app with JSON parsing.`GET /config` reads `data/config.json`, parses and caches it via `node-cache` (TTL 24h).`GET /crash` exits the process. Listens on `PORT` (default 3000).

Role Tenant backend microservice exposing a config API and a chaos hook. Consumed by the FE through its server-side proxy. Managed by the platform for restarts.

C.3. Build and Runtime

Listing 8.: Dockerfile

```
# ---------- build stage ----------
FROM node:20-alpine AS build

ARG CONFIG_PATH=./config/config.json

WORKDIR /app
COPY package*.json ./
RUN npm ci --omit=dev \
    && npm install typescript --no-save

# copy source & compile TS → JS
COPY tsconfig.json ./
COPY src ./src
COPY ${CONFIG_PATH} ./data/config.json
RUN npx tsc -p tsconfig.json

# ---------- production stage ----------
FROM node:20-alpine

WORKDIR /app
COPY --from=build /app/package*.json ./
COPY --from=build /app/node_modules ./node_modules
COPY --from=build /app/dist ./dist
COPY --from=build /app/data/config.json ./config.json

ENV NODE_ENV=production
EXPOSE 3000
CMD ["node", "dist/server.js"]
```

Why Reproducible, minimal container build for the tenant BE with configurable config injection.

What Two-stage Node 20 Alpine image: installs prod deps + TypeScript, compiles TS to JS, copies artifacts. Embeds config via `ARG CONFIG_PATH` (ends up at `/app/config.json`), runs `node dist/server.js` on port 3000.

Role Deployable image for the Tenant backend API. Enables tenant-specific configuration at build time and clean separation of build/runtime layers.

D. ConfigSchema

D.1. Directory Structure

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

Figure 5: ConfigSchema directory tree (trimmed)

D.2. Key Files

Listing 9.: Index: `src/index.ts`

```ts
import { z } from 'zod';

/* Address sub-schema  */
const AddressSchema = z.object({
    country: z
        .string()
        .regex(/^[a-z]{2}-[A-Z]{2}$/, { message: 'must look like “de-DE” or “en-US”' }),

    zipCode: z.string().regex(/^\d{4,10}$/, { message: 'digits, 4-10 chars' }),

    city: z.string().min(1),

    street: z.string().min(1),

    streetNumber: z.string().regex(/^\d+[a-zA-Z]?$/, {
        message: 'e.g. “12”, “12a”, “99B”',
    }),
});
export type Address = z.infer<typeof AddressSchema>;

/* Contact sub-schema */
const ContactSchema = z.object({
    areaCode: z.string().regex(/^\d{2,5}$/, {
        message: 'area code must be 2-5 digits (e.g. “212”, “030”, “089”)',
    }),

    phoneNumber: z.number().int(),

    email: z.string().email(),
});
export type Contact = z.infer<typeof ContactSchema>;

/*  Root config schema  */
export const ConfigSchema = z.object({
    address: AddressSchema,

    companyName: z.string().min(1),
```

```
38
39      proposition: z.string().min(1),
40
41      products: z.array(z.string()).nonempty({
42          message: 'products must contain at least one item',
43      }),
44
45      about: z.string().min(1),
46  });
47  export type Config = z.infer<typeof ConfigSchema>;
```

Why Single source of truth for tenant config. Prevents malformed data from reaching UI or API.

What Defines `zod` schema and enforces formats. Exports inferred TS types.

Role Shared contract used for runtime validation of tenant configuration. Ensures all components agree on the structure and types of `config.json`.

Listing 10.: Index test: `src/index.test.ts`

```ts
import { describe, it, expect } from 'vitest';
import { ConfigSchema } from "../src";

const validConfig = {
    address: {
        country:     'de-DE',
        zipCode:     '85049',
        city:        'Ingolstadt',
        street:      'Esplanade',
        streetNumber:'10',
    },
    companyName:  'Test Inc.',
    proposition:  'We do nothing!',
    products:     ['nothing', 'still nothing'],
    about:        'we\'re not even a real business',
};

describe('ConfigSchema', () => {
    it('accepts a well-formed config object', () => {
        expect(() => ConfigSchema.parse(validConfig)).not.toThrow();
    });

    it('rejects an invalid zipCode', () => {
        const bad = { ...validConfig, address: { ...validConfig.address, zipCode: 'XYZ' } };
        const res = ConfigSchema.safeParse(bad);
        expect(res.success).toBe(false);
        if(!res.success){
            expect(res.error.issues[0].path).toEqual(['address', 'zipCode']);
        }
    });

    it('rejects an empty products array', () => {
        const bad = { ...validConfig, products: [] };

        const res = ConfigSchema.safeParse(bad);
        expect(res.success).toBe(false);

```

```
38          if (!res.success) {
39              expect(res.error.issues[0].path).toEqual(['products']);
40          }
41      });
42
43      it('rejects when companyName is missing', () => {
44          const { companyName, ...noName } = validConfig;
45          expect(() => ConfigSchema.parse(noName as any)).toThrow();
46      });
47  });
```

Why Test to verify the schema is valid and matches the expected structure.

What Vitest suite for `ConfigSchema`: accepts a valid sample, rejects bad ZIP code, empty `products`, and missing `companyName`. Asserts zod error paths.

Role CI guardrail for FE/BE compatibility, keeps the schema stable across services.

D.3. Build and Runtime

Listing 11.: Build: `tsconfig.build.json`

```
{
    "compilerOptions": {
        "moduleResolution": "node",
        "allowSyntheticDefaultImports": true,
        "declaration": true,
        "emitDeclarationOnly": true
    },
    "include": ["src"]
}
```

Why Publishable package includes usable types for consumers.

What TS build config that emits only `.d.ts` from `src` (Node module resolution, synthetic default imports).

Role Enables FE/BE to import the shared schema's types via NPM without compiling TS.

E. DashboardFE (Selected Code)

E.1. 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
└── Dockerfile
```

Figure 6: DashboardFE directory tree (trimmed)

E.2. Key Files

Listing 12.: API route: `src/app/api/config`

```
'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 POST(req: NextRequest) {
    const body = await req.json();
    const user = req.headers.get('user') || '';

    try {
        const backendRes = await fetch(endpoint, {
            method: 'POST',
            headers: {
                'Content-Type': 'application/json',
                'user': user,
            },
            body: JSON.stringify(body),
        });
        const contentType = backendRes.headers.get('content-type') || '';
        const raw = await backendRes.text();
        const isJson = contentType.includes('application/json');
        const responseBody = isJson ? JSON.parse(raw) : raw;


        return new NextResponse(responseBody, {
            status: backendRes.status,
            headers: {
                'Content-Type': contentType || 'text/plain',
                'Cache-Control': 'no-store',
            }
        });
    } catch (error) {
        console.error('Proxy request failed:', error);
```

```
36          return new NextResponse('Internal Server Error trying to reach ' + endpoint, { status:
            ↪  500 });
37      }
38  }
```

Why Server-side CORS bypass and security boundary for config changes.

What Proxies a POST to `CONFIG_ENDPOINT` with body and `user` header. Preserves status/`content-type`, parses JSON when applicable, and sets `Cache-Control: no-store`.

Role CORS-bridging proxy route.

Listing 13.: Root page: `src/app/page.tsx`

```tsx
import { getSession } from '@/lib/auth';
import { redirect } from 'next/navigation';
import DashboardButton from '@/components/DashboardButton';

export default async function Home() {
    const session = await getSession();
    if (!session) redirect('/login');

    return (
        <main className="p-8">
            <h1 className="text-3xl font-bold">Welcome, {session.user}!</h1>
            <DashboardButton />
        </main>
    );
}
```

Why Redirects to the login page if no user is authenticated.

What Fetches the session server-side; if absent calls `redirect('/login')`, otherwise renders a welcome and `DashboardButton`.

Role Entry controller for the dashboard FE, gating access and routing authenticated users.

Listing 14.: Layout: `src/app/layout.tsx`

```tsx
import type { Metadata } from "next";
import { Geist, Geist_Mono } from "next/font/google";
import "./globals.css";

const geistSans = Geist({
  variable: "--font-geist-sans",
  subsets: ["latin"],
});

const geistMono = Geist_Mono({
  variable: "--font-geist-mono",
  subsets: ["latin"],
});

export const metadata: Metadata = {
  title: "Create Next App",
  description: "Generated by create next app",
};

export default function RootLayout({
  children,
}: Readonly<{
  children: React.ReactNode;
}>) {
  return (
    <html lang="en">
      <body className={`${geistSans.variable} ${geistMono.variable}`}>
        {children}
      </body>
    </html>
  );
}
```

Why Serves as root DOM container for the application.

What Sets metadata and renders children.

Role Framework-level layout.

Listing 15.: Dashboard page: `src/app/dashboard/page.tsx`

```tsx
import { getSession } from '@/lib/auth';
import { redirect } from 'next/navigation';
import ConfigForm from '@/components/ConfigForm';

export default async function DashboardPage() {

    const session = await getSession();
    if (!session) redirect('/login');

    return (
        <main>
            <h1>Tenant Configuration for <b>{session.user}</b></h1>

            <ConfigForm user={session.user}/>
        </main>
    );
}
```

Why Renders the `ConfigForm` for authenticated users.

What Checks session on the server, redirects unauthenticated users to `/login` and renders `ConfigForm` for the current `session.user`.

Role Auth wrapper component for the `ConfigForm`.

Listing 16.: Login page: `src/app/login/page.tsx`

```tsx
import LoginForm from "@/components/LoginForm"

export default function LoginPage() {
    return (
        <main className="mx-auto max-w-sm py-20">
            <h1 className="text-2xl font-bold mb-6">Log in</h1>
            <LoginForm />
        </main>
    );
}
```

Why Provides the entry point for authentication in the dashboard FE.

What Renders the `LoginForm` component inside a centered layout.

Role UI endpoint for the auth flow. Separates page shell from the actual login logic handled by `LoginForm` and the auth layer.

Listing 17.: Login action: `src/app/login/actions.ts`

```ts
'use server';
import { cookies } from 'next/headers';
import { redirect } from 'next/navigation';

export async function login(_: unknown, formData: FormData) {
    const username = formData.get('username')?.toString().trim();

    if (!username) {
        return { error: 'Username required' };
    }

    const store = await cookies();
    store.set('user', username, {
        path: '/',
        httpOnly: true,
        sameSite: 'lax',
        maxAge: 60 * 60 * 24, // 24h
    });

    redirect('/');
}
```

Why Implements the actual login step by creating a session cookie.

What Reads `username` from form data. If missing, returns an error. Otherwise sets an HTTP-only `user` cookie (24h) and redirects to `/`.

Role Auth server action invoked by `LoginForm`. Bootstraps the session state used across the dashboard FE.

Listing 18.: Config form component: `src/components/ConfigForm.tsx`

```tsx
'use client';

import { useState, FormEvent } from 'react';
import {
    ConfigSchema,
    type Config,
} from '@mystiker123/config-schema';
import styles from './ConfigForm.module.css';

type ConfigDraft = Omit<Config, 'products'> & {
    products: string[];
};

interface ConfigFormProps {
    user: string
}

export default function ConfigForm({ user }: ConfigFormProps) {
    const [draft, setDraft] = useState<ConfigDraft>({
        address: {
            country: '',
            zipCode: '',
            city: '',
            street: '',
            streetNumber: '',
        },
        companyName: '',
        proposition: '',
        products: [''],
        about: '',
    });

    const [status, setStatus] = useState<
        'idle' | 'saving' | 'success' | 'error'
    >('idle');

    function patch<K extends keyof ConfigDraft>(
```

```
        key: K,
        value: ConfigDraft[K],
    ) {
        setDraft(prev => ({ ...prev, [key]: value }));
    }

    function updateProduct(index: number, value: string) {
        setDraft(prev => {
            const next = [...prev.products];
            next[index] = value;
            if (index === next.length - 1 && value.trim() !== '') next.push('');
            return { ...prev, products: next };
        });
    }

    async function handleSubmit(e: FormEvent<HTMLFormElement>) {
        e.preventDefault();

        const cleaned: Config = {
            ...draft,
            products: draft.products
                .filter(p => p.trim() !== '') as [string, ...string[]],
        };

        const parsed = ConfigSchema.safeParse(cleaned);
        if (!parsed.success) {
            console.warn(parsed.error);
            setStatus('error');
            return;
        }

        setStatus('saving');

        const res = await fetch('/api/config', {
            method: 'POST',
            headers: {
                'Content-Type': 'application/json',
                'user': user,
            },
            body: JSON.stringify(parsed.data),
```

```
78          });
79
80          setStatus(res.ok ? 'success' : 'error');
81      }
82
83      return (
84          <form onSubmit={handleSubmit} className='space-y-6'>
85              <div className={styles.row}>
86                  <label htmlFor='companyName' className={styles.label}>Company Name</label>
87                  <input
88                      id='companyName'
89                      value={draft.companyName}
90                      onChange={(e) => patch('companyName', e.target.value)}
91                      placeholder='Company name'
92                      className={styles.input}
93                  />
94              </div>
95              <div className={styles.row}>
96                  <label htmlFor='proposition' className={styles.label}>Proposition</label>
97                  <input
98                      id='proposition'
99                      value={draft.proposition}
100                     onChange={(e) => patch('proposition', e.target.value)}
101                     placeholder='Proposition'
102                     className={styles.input}
103                 />
104             </div>
105             <div className={styles.row}>
106                 <label htmlFor='about' className={styles.label}>About</label>
107                 <input
108                     id='about'
109                     value={draft.about}
110                     onChange={(e) => patch('about', e.target.value)}
111                     placeholder='About'
112                     className={styles.input}
113                 />
114             </div>
115             <div className={styles.row}>
116                 <label id='country' className={styles.label}>Country</label>
117                 <input
```

```
                id='country'
                value={draft.address?.country}
                onChange={(e) => {
                    patch('address', {
                        ...draft.address,
                        country: e.target.value
                    })
                }}
                placeholder='Country'
                className={styles.input}
            />
        </div>
        <div className={styles.row}>
            <label htmlFor='zipCode' className={styles.label}>Zip Code</label>
            <input
                id='zipCode'
                value={draft.address?.zipCode}
                onChange={(e) => {
                    patch('address', {
                        ...draft.address,
                        zipCode: e.target.value
                    })
                }}
                placeholder='Zip Code'
                className={styles.input}
            />
        </div>
        <div className={styles.row}>
            <label htmlFor='city' className={styles.label}>City</label>
            <input
                id='city'
                value={draft.address?.city}
                onChange={(e) => {
                    patch('address', {
                        ...draft.address,
                        city: e.target.value
                    })
                }}
                placeholder='City'
                className={styles.input}
```

```
158                 />
159             </div>
160             <div className={styles.row}>
161                 <label htmlFor='street' className={styles.label}>Street</label>
162                 <input
163                     id='street'
164                     value={draft.address?.street}
165                     onChange={(e) => {
166                         patch('address', {
167                             ...draft.address,
168                             street: e.target.value
169                         })
170                     }}
171                     placeholder='Street'
172                     className={styles.input}
173                 />
174             </div>
175             <div className={styles.row}>
176                 <label htmlFor='streetNumber' className={styles.label}>Street Number</label>
177                 <input
178                     id='streetNumber'
179                     value={draft.address?.streetNumber}
180                     onChange={(e) => {
181                         patch('address', {
182                             ...draft.address,
183                             streetNumber: e.target.value
184                         })
185                     }}
186                     placeholder='Street Number'
187                     className={styles.input}
188                 />
189             </div>
190             <div className={styles.row}>
191                 <label className={styles.label}>Products</label>
192                 <div className={styles.productsRow}>
193 
194                     {draft.products.map((prod, i) => (
195                         <input
196                         key={i}
197                         value={prod}
```

```
198                         placeholder={`Product ${i + 1}`}
199                         onChange={e => updateProduct(i, e.target.value)}
200                         className={styles.input}
201                         />
202                     ))}
203                 </div>
204             </div>
205             <div className={styles.center}>
206                 <button
207                 type='submit'
208                 disabled={status === 'saving'}
209                 className={styles.button}
210             >
211                 {status === 'saving' ? 'Saving. . . ' : 'Save'}
212             </button>
213             </div>
214
215             {status === 'success' && (
216                 <p className='text-green-600'>Saved!</p>
217             )}
218             {status === 'error' && (
219                 <p className='text-red-600'>
220                 Validation failed or server error.
221                 </p>
222             )}
223         </form>
224     );
225 }
```

Why Core UI component for authoring tenant configuration. Validates on the client side to prevent bad data and reduce backend failures.

What Manages form state, adds product fields dynamically, cleans + validates with `ConfigSchema`, then POSTs the config (with `user` header) to `/api/config`. Shows saving/-success/error status.

Role Dashboard FE form component. Feeds validated tenant config to the backend through the proxy route, bridging admin input to persisted configuration.

Listing 19.: Dashboard button component: `src/components/DashboardButton.tsx`

```tsx
'use client';

import { useRouter } from 'next/navigation';

export default function DashboardButton() {
    const router = useRouter();

    return (
        <button
        onClick={() => router.push('/dashboard')}
        className="rounded bg-blue-600 px-4 py-2 text-white hover:bg-blue-700 focus:outline-none
        ↪  focus:ring"
        >
        Go to Dashboard
        </button>
    );
}
```

Why Button navigating to the dashboard page.

What Navigates to `/dashboard` via Next.js`useRouter()`.

Role Entry point for the form.

Listing 20.: Login form component: `src/components/LoginForm.tsx`

```tsx
'use client';
import { useState, useTransition } from 'react';
import { login } from '@/app/login/actions';

export default function LoginForm() {
    const [error, setError] = useState<string | null>(null);
    const [pending, start]  = useTransition();

    function handleSubmit(e: React.FormEvent<HTMLFormElement>) {
        e.preventDefault();
        const fd = new FormData(e.currentTarget);

        start(async () => {
        const res = await login(null, fd); // call action
        if (res?.error) setError(res.error);
        });
    }

    return (
        <form onSubmit={handleSubmit} className="space-y-4">
            <input
                name="username"
                placeholder="username"
                required
                disabled={pending}
                className="border p-2 w-full"
            />
            {error && <p className="text-red-500 text-sm">{error}</p>}
            <button
                type="submit"
                disabled={pending}
                className="bg-black text-white px-4 py-2 disabled:opacity-50"
            >
                {pending ? '. . . ' : 'Sign in'}
            </button>
        </form>
    );
}
```

Why Entry point for signing in.

What Client form builds `FormData` and calls server action `login` via `useTransition`

Role Authentication UI.

Listing 21.: Auth lib: `src/lib/auth.ts`

```ts
import { cookies } from 'next/headers';

const COOKIE = 'user';

// read username from cookie
export async function getSession(): Promise<{ user: string } | null> {
    const store = await cookies();
    const user  = store.get(COOKIE)?.value;
    return user ? { user } : null;
}

// clear cookie
export async function logout() {
    const store = await cookies();
    store.delete(COOKIE);
}
```

Why Single source of truth for auth state on the server.

What `getSession()` reads the `user` cookie via `cookies()` and returns a session or `null`.`logout()` deletes the cookie.

Role Authentication helper in the dashboard FE. Enables server components to enforce login.

E.3. Build and Runtime

Listing 22.: Dockerfile

```
# syntax=docker.io/docker/dockerfile:1

FROM node:18-alpine AS base

# Install dependencies only when needed
FROM base AS deps
# Check https://github.com/nodejs/docker-node/tree/b4117f9333da4138b03a546ec926ef50a31506c3#nodealpine to understand why libc6-compat might be needed.
RUN apk add --no-cache libc6-compat
WORKDIR /app

# Install dependencies based on the preferred package manager
COPY dashboard_fe/package.json dashboard_fe/yarn.lock* dashboard_fe/package-lock.json* dashboard_fe/pnpm-lock.yaml* dashboard_fe/.npmrc* ./
RUN \
  if [ -f yarn.lock ]; then yarn --frozen-lockfile; \
  elif [ -f package-lock.json ]; then npm ci; \
  elif [ -f pnpm-lock.yaml ]; then corepack enable pnpm && pnpm i --frozen-lockfile; \
  else echo "Lockfile not found." && exit 1; \
  fi


# Rebuild the source code only when needed
FROM base AS builder
WORKDIR /app
COPY --from=deps /app/node_modules ./node_modules
COPY dashboard_fe/. .

# Build ARG for CONFIG_ENDPOINT
ARG CONFIG_ENDPOINT
ENV CONFIG_ENDPOINT=$CONFIG_ENDPOINT

# Next.js collects completely anonymous telemetry data about general usage.
# Learn more here: https://nextjs.org/telemetry
# Uncomment the following line in case you want to disable telemetry during the build.
# ENV NEXT_TELEMETRY_DISABLED=1

```

```
RUN \
  if [ -f yarn.lock ]; then yarn run build; \
  elif [ -f package-lock.json ]; then npm run build; \
  elif [ -f pnpm-lock.yaml ]; then corepack enable pnpm && pnpm run build; \
  else echo "Lockfile not found." && exit 1; \
  fi

# Production image, copy all the files and run next
FROM base AS runner
WORKDIR /app

ENV NODE_ENV=production
# Uncomment the following line in case you want to disable telemetry during runtime.
# ENV NEXT_TELEMETRY_DISABLED=1

RUN addgroup --system --gid 1001 nodejs
RUN adduser --system --uid 1001 nextjs

COPY --from=builder /app/public ./public

# Automatically leverage output traces to reduce image size
# https://nextjs.org/docs/advanced-features/output-file-tracing
COPY --from=builder --chown=nextjs:nodejs /app/.next/standalone ./
COPY --from=builder --chown=nextjs:nodejs /app/.next/static ./.next/static

USER nextjs

EXPOSE 3000

ENV PORT=3000

# server.js is created by next build from the standalone output
# https://nextjs.org/docs/pages/api-reference/config/next-config-js/output
ENV HOSTNAME="0.0.0.0"
CMD ["node", "server.js"]
```

Why Containerized build for the dashboard FE with minimal attack surface and environment-specific endpoints baked at build time.

What Multi-stage build: installs deps, builds Next.js with `CONFIG_ENDPOINT` via `ARG`/`ENV`, copies `.next/standalone` and static assets, runs as non-root on port 3000.

Role Deployment artifact for the Dashboard Frontend.

F. DashboardBE (Selected Code)

F.1. Directory Structure

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

Figure 7: DashboardBE directory tree (trimmed)

F.2. Key Files

Listing 23.: Server: `src/server.ts`

```ts
import express, { Handler, Request, Response } from 'express';
import { mkdirSync, writeFileSync, existsSync } from 'fs';
import { setUserIp } from './redis';
import { ConfigSchema } from '@mystiker123/config-schema';
import cors from 'cors';
import { dirname } from 'path';

/* ------------------------------------------------------------------ */
/* express app                                                        */
/* ------------------------------------------------------------------ */

const app = express();
app.use(cors({
    origin: 'http://dashboard.local:8080',
    methods: ['GET', 'POST', 'PUT', 'DELETE', 'OPTIONS'],
    allowedHeaders: ['Content-Type', 'user'],
    exposedHeaders: ['user'],
    credentials: true,
}));

app.use(express.json());

const createConfig: Handler = async (req: Request, res: Response): Promise<void> => {
    const user: string | undefined = req.get('user');
    if (!user) {
        console.error('user not provided');
        res.status(400).send('user not provided');
        return;
    }
    const parsed = ConfigSchema.safeParse(req.body);
    if (!parsed.success) {
        console.error('JSON does not fit schema')
        res.status(400).send('JSON does not fit schema');
        return;
    }
    try {
        const filePath = `data/${user}/config.json`;
```

```
38          console.log(`Attempting to create config directory for user: ${user}`);
39          if (!existsSync(dirname(filePath))) {
40              console.log(`Creating directory for user: ${user}`);
41              mkdirSync(dirname(filePath), { recursive: true });
42          }
43          console.log(`Attempting to write config for user: ${user}`);
44          writeFileSync(filePath, JSON.stringify(parsed.data),'utf-8');
45          console.log(`Config written successfully for user: ${user}`);
46          res.status(200).send('Config written successfully');
47      } catch (e) {
48          console.error('[config] write failed for user: ', user, e);
49          res.status(500).send('Internal Server Error');
50      }
51  }
52
53  app.post('/config', createConfig);
54
55  /* ------------------------------------------------------------------ */
56  /* start server                                                       */
57  /* ------------------------------------------------------------------ */
58
59  const PORT = Number(process.env.PORT) || 3000;
60  app.listen(PORT, '0.0.0.0', () => {
61      /* eslint-disable no-console */
62      console.log(`<unicode-rocket>  API ready on http://0.0.0.0:${PORT}`);
63  });
```

Why Central write path for tenant config. Validates and persists admin edits safely.

What Express app with CORS policy for the dashboard FE.`POST /config` reads `user` header, validates body via `ConfigSchema`, ensures `data/<user>/` exists, writes `config.json`, returns 200/4xx/5xx, listens on `PORT`.

Role Dashboard backend service. Persists tenant configuration to make it available to the CRD.

Listing 24.: Redis: `src/lib/redis.ts`

```ts
import Redis from 'ioredis';

const redis = new Redis({
    host: process.env.REDIS_HOST || 'redis',
    port: Number(process.env.REDIS_PORT || 6379),
});

export async function setUserIp(username: string, ip: string): Promise<void> {
    await redis.hset('users', username, JSON.stringify({ ip }));
}

export async function getUserIp(username: string): Promise<string | undefined> {
    const raw = await redis.hget('users', username);
    return raw ? JSON.parse(raw).ip : undefined;
}
```

Why Fast, central lookup of tenant -> tenant cluster IP for routing and ops.

What Initializes an `ioredis` client (env `REDIS_HOST`/`REDIS_PORT`, defaults to `redis:6379`). Stores per-user IPs in hash `users` as JSON via `setUserIp`/`getUserIp`.

Role Infrastructure state cache used by frontend to provide the IP.

F.3. Build and Runtime

Listing 25.: Dockerfile

```dockerfile
# ---------- build stage ----------
FROM node:20-alpine AS build

WORKDIR /app
COPY package*.json ./
RUN npm ci

# copy source & compile TS → JS
COPY tsconfig.json ./
COPY src ./src
RUN npx tsc -p tsconfig.json

# ---------- production stage ----------
FROM node:20-alpine

WORKDIR /app
COPY --from=build /app/package*.json ./
COPY --from=build /app/node_modules ./node_modules
COPY --from=build /app/dist ./dist

ENV NODE_ENV=production
ENV PORT=8080
EXPOSE 8080
CMD ["node", "dist/server.js"]
```

Why Reproducible build and slim runtime image for the Dashboard backend.

What Two-stage Docker build on `node:20-alpine`: runs `npm ci`, compiles TS via `npx tsc`, copies `dist` and `node_modules` into a minimal runtime, sets `NODE_ENV=production`, `PORT=8080`, and starts `node dist/server.js`.

Role Deployment artifact for the Dashboard backend.

G. Infra (Scripts and Manifests)

G.1. Directory Structure

```
./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 8: Infra directory tree (trimmed)

G.2. Layout

Path	Purpose	Highlights
`config/`	Configuration sample	`config.example.json`
`k8s/`	All Kubernetes and KCP artifacts (CRDs, RBAC, workspaces, deployments, ingress)	Subfolders by concern: `cluster/`, `deployments/`, `rbac/`, `apibindings/`, `kcp-crds/`, `workspaces/`, `ingress/`
`scripts/`	Automation: setup, build, cluster+workspace creation, TMC bootstrap, utilities	Split into `build/` and task-specific scripts

G.3. Simplified Deployment Workflow

1. **Bootstrap tools** (kubectl, krew, plugins, kind, etc.):
   `scripts/install-deps.sh`,
   `scripts/clone-repos.sh`,
   `scripts/setup-kcp.sh`,
   `scripts/update-krew.sh`,
   `scripts/setup-krew-plugins.sh`,
   `scripts/path.sh`,
   `scripts/setup-tmc.sh`,
   `scripts/path.sh`.

2. **Start TMC-glskcp (KCP)**:
   `scripts/start-tmc-kcp.sh`,
   (`scripts/start-kcp.sh`).

3. **Create workspaces (creates clusters in the process)**:
   `scripts/create-dashboard-workspace.sh`,
   `scripts/create-tenant-x-workspace.sh`.

4. **Create CRDs**:
   `scripts/CRDs/create-tenant-x-CRD.sh`.

G.4. Build and Images

Listing 26.: Build dashboardBE: `/scripts/build/dashboardBE.sh`

```
#!/bin/bash
set -e

SERVICE_NAME="dashboard-be"
IMAGE_NAME="${SERVICE_NAME}:latest"
DOCKERFILE_PATH="./BachelorThesis_DashboardBE/Dockerfile"
BUILD_CONTEXT="./BachelorThesis_DashboardBE"

# <unicode-magnifying-glass> Check for Dockerfile
if [ ! -f "$DOCKERFILE_PATH" ]; then
    echo "<unicode-cancel> Dockerfile not found: $DOCKERFILE_PATH"
    exit 1
fi

# <unicode-building> Build Docker image

echo "<unicode-construction> Building Docker image: ${IMAGE_NAME}"

docker build \
    -t "$IMAGE_NAME" \
    -f "$DOCKERFILE_PATH" \
    "$BUILD_CONTEXT"

echo "<unicode-check> Build completed: $IMAGE_NAME"
```

Why One-command, reproducible container build for the dashboard BE

What Bash script building the dashboard BE image.

Role Build artifact producer. Generates the image consumed by the Kubernetes deployment.

Listing 27.: Build dashboardFE:`/scripts/build/dashboardFE.sh`

```
#!/bin/bash

set -e

# <unicode-robot> Configurable values

IMAGE_NAME="dashboard-fe"
TAG="latest"
API_URL="${1}"
DOCKERFILE_PATH="./BachelorThesis_DashboardFE/Dockerfile"
BUILD_CONTEXT="./BachelorThesis_DashboardFE"

# Check if API URL is provided
if [ -z "$API_URL" ]; then
    echo "<unicode-cancel> API URL is required as the first argument. Usage:
    ↪  ./scripts/build/dashboardFE.sh <API_URL>"
    exit 1
fi

# Check if Dockerfile exists
if [ ! -f "$DOCKERFILE_PATH" ]; then
    echo "<unicode-cancel> Dockerfile not found: $DOCKERFILE_PATH"
    exit 1
fi

echo "<unicode-construction> Building Docker image: $IMAGE_NAME:$TAG"
echo "Using API URL: $API_URL"

# Build Docker image
docker build \
    -t "$IMAGE_NAME:$TAG" \
    --build-arg CONFIG_ENDPOINT="${API_URL}/config" \
    -f "$DOCKERFILE_PATH" \
    "$BUILD_CONTEXT"
```

Why Parameterized, reproducible build for the Dashboard Frontend. Binds the FE to the correct backend at build time.

What Bash script that requires and API URL as build argument. Builds the FE.

Role Produces the frontend build artifact for the dashboard application.

Listing 28.: Build tenantBE:/scripts/build/tenantBE.sh

```
#!/bin/bash

set -e

# <unicode-robot> Configurable values
IMAGE_NAME="tenant-be"
TAG="latest"
CONFIG_SRC_PATH="./config/config.example.json"
CONFIG_TMP_PATH="./BachelorThesis_TenantBE/temp.config.json"
CONFIG_ARG_NAME="CONFIG_PATH"
DOCKERFILE_PATH="./BachelorThesis_TenantBE/Dockerfile"
BUILD_CONTEXT="./BachelorThesis_TenantBE"

echo "<unicode-construction> Building Docker image: $IMAGE_NAME:$TAG"
echo "<unicode-page> Using config: $CONFIG_SRC_PATH"

# <unicode-magnifying-glass> Check for Dockerfile
if [ ! -f "$DOCKERFILE_PATH" ]; then
    echo "<unicode-cancel> Dockerfile not found: $DOCKERFILE_PATH"
    exit 1
fi

# <unicode-magnifying-glass> Check for config
if [ ! -f "$CONFIG_SRC_PATH" ]; then
    echo "<unicode-cancel> Config file not found: $CONFIG_SRC_PATH"
    exit 1
fi

# <unicode-test-tube> Prepare temporary config inside build context
cp "$CONFIG_SRC_PATH" "$CONFIG_TMP_PATH"

# <unicode-building> Build Docker image
docker build \
    -t "$IMAGE_NAME:$TAG" \
    --build-arg "$CONFIG_ARG_NAME=temp.config.json" \
    -f "$DOCKERFILE_PATH" \
    "$BUILD_CONTEXT"
```

```
38
39 # <unicode-broom> Clean up temp file
40 rm "$CONFIG_TMP_PATH"
41
42 echo "<unicode-check> Build completed: $IMAGE_NAME:$TAG"
```

Why Reproducible tenant backend build with a known config.

What Copies `config.example.json` into the build context as `temp.config.json`, builds the image, then deletes the temp file. Builds the tenant BE image.

Role Infra build script for the tenant backend. Takes `TENANT_ID` as argument. Produces the container consumed by the cluster with the config baked in.

Listing 29.: Build tenantFE:`/scripts/build/tenantFE.sh`

```
#!/bin/bash

set -e

# Tenant ID
TENANT_ID="$1"
TENANT_BE_URL="$2"

if [ -z "$TENANT_ID" ]; then
    echo "<unicode-cancel> Tenant ID is required as the first argument. Usage:
    ↪  ./scripts/build/tenantFE.sh <TENANT_ID> <TENANT_BE_URL>"
    exit 1
fi

if [ -z "$TENANT_BE_URL" ]; then
    echo "<unicode-cancel> Tenant BE URL is required as the second argument. Usage:
    ↪  ./scripts/build/tenantFE.sh <TENANT_ID> <TENANT_BE_URL>"
    exit 1
fi

# read .env.production file
source ./.env.production

# <unicode-brain> Resolve dynamic API URL
UPPER_TENANT_ID=$(echo "$TENANT_ID" | tr '[:lower:]' '[:upper:]')
URL_VAR_NAME="TENANT_${UPPER_TENANT_ID}_BE_URL"

if [ -z "$TENANT_BE_URL" ]; then
    echo "<unicode-cancel> No backend URL found for tenant ID '$TENANT_ID' in .env.production"
    exit 1
fi

# <unicode-robot> Configurable values
IMAGE_NAME="tenant-fe-$TENANT_ID"
TAG="latest"
DOCKERFILE_PATH="./BachelorThesis_TenantFE/Dockerfile"
BUILD_CONTEXT="./BachelorThesis_TenantFE/tenant_fe"
```

```
CONFIG_ENDPOINT="${TENANT_BE_URL}/config"
CRASH_ENDPOINT="${TENANT_BE_URL}/crash"

echo "<unicode-construction> Building Docker image: $IMAGE_NAME:$TAG"

# <unicode-magnifying-glass> Check for Dockerfile
if [ ! -f "$DOCKERFILE_PATH" ]; then
    echo "<unicode-cancel> Dockerfile not found: $DOCKERFILE_PATH"
    exit 1
fi

# <unicode-building> Build Docker image
docker build \
    -t "$IMAGE_NAME:$TAG" \
    --build-arg CONFIG_ENDPOINT="$CONFIG_ENDPOINT" \
    --build-arg CRASH_ENDPOINT="$CRASH_ENDPOINT" \
    -f "$DOCKERFILE_PATH" \
    "$BUILD_CONTEXT"

echo "<unicode-check> Build completed: $IMAGE_NAME:$TAG"
```

Why Produces a per-tenant Frontend image with the correct backend endpoints baked in.

What Takes `TENANT_ID` and `TENANT_BE_URL` as arguments, builds the tenant FE image with the correct backend endpoint.

Role Creates the deployable FE artifact bound to its tenant's backend service.

G.5. Clusters

Listing 30.: Dashboard Cluster Manifest: `/k8s/cluster/kind-config.yaml`

```yaml
kind: Cluster
apiVersion: kind.x-k8s.io/v1alpha4
name: dashboard
nodes:
  - role: control-plane
    extraPortMappings:
      - containerPort: 80
        hostPort: 8080 # 80 might be in use by another service
        protocol: TCP
      - containerPort: 443
        hostPort: 8443
        protocol: TCP
```

Why Manifest for the dashboard cluster.

What Maps container ports.

Role Runtime for the dashboard stack.

Listing 31.: Create Tenant Cluster Script: `/scripts/create-tenant-x-cluster.sh`

```
#!/usr/bin/env bash

set -euo pipefail

# TENANT ID
TENANT_ID=$1

if [[ -z "$TENANT_ID" ]]; then
    echo "Usage: $0 <TENANT_ID>"
    exit 1
fi

CLUSTER_NAME="tenant-${TENANT_ID}"
HTTP_PORT=$((8000 + TENANT_ID)) # collision on 8080 -> we don't need 80 tenants for the prototype
HTTPS_PORT=$((8400 + TENANT_ID))

echo "Creating cluster '$CLUSTER_NAME' with ports:"
echo "  → HTTP  : $HTTP_PORT → containerPort 80"
echo "  → HTTPS : $HTTPS_PORT → containerPort 443"

kind create cluster --name "$CLUSTER_NAME" --config=- <<EOF
kind: Cluster
apiVersion: kind.x-k8s.io/v1alpha4
nodes:
  - role: control-plane
    extraPortMappings:
      - containerPort: 80
        hostPort: $HTTP_PORT
        protocol: TCP
      - containerPort: 443
        hostPort: $HTTPS_PORT
        protocol: TCP
EOF
```

Why Create an isolated per-tenant cluster with the provided manifest and the `TENANT_ID` as argument.

What Maps container ports. Makes the manifest dynamic.

Role Runtime for the tenant stack.

G.6. Deployment Manifests

Listing 32.: dashboardBE Manifest: `/k8s/deployments/dashboard-be.yaml`

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: dashboard-be
spec:
  replicas: 1
  selector:
    matchLabels:
      app: dashboard-be
  template:
    metadata:
      labels:
        app: dashboard-be
    spec:
      containers:
        - name: dashboard-be
          image: dashboard-be:latest
          imagePullPolicy: Never
          ports:
            - containerPort: 8080
---
apiVersion: v1
kind: Service
metadata:
  name: dashboard-be
spec:
  selector:
    app: dashboard-be
  ports:
    - protocol: TCP
      port: 80
      targetPort: 8080
  type: NodePort
```

Why Deploys and exposes the dashboard backend.

What Creates a `Deployment` running `dashboard-be:latest` with `imagePullPolicy: Never` and a `Service` (NodePort) exposing port 80 -> targetPort 8080.

Role Integrates the dashboard backend into the cluster.

Listing 33.: dashboardFE Manifest: `/k8s/deployments/dashboard-fe.yaml`

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: dashboard-fe
  labels:
    app: dashboard-fe
spec:
  replicas: 1
  selector:
    matchLabels:
      app: dashboard-fe
  template:
    metadata:
      labels:
        app: dashboard-fe
    spec:
      containers:
        - name: dashboard-fe
          image: dashboard-fe:latest
          imagePullPolicy: Never
          ports:
            - containerPort: 3000
          env:
            - name: CONFIG_ENDPOINT
              value: http://dashboard-be.default.svc.cluster.local/config

---
apiVersion: v1
kind: Service
metadata:
  name: dashboard-fe
spec:
  selector:
    app: dashboard-fe
  ports:
    - protocol: TCP
      port: 80
```

```
38       targetPort: 3000
39   type: NodePort
```

Why Deploys and exposes the dashboard frontend.

What Creates a `Deployment` of the Next.js app on port 3000 and a NodePort `Service` (80 -> 3000); sets `CONFIG_ENDPOINT` to `http://dashboard-be.default.svc.cluster.local/config`.

Role Integrates the dashboard frontend into the cluster.

Listing 34.: tenantBE Manifest: `/k8s/deployments/tenant-be.yaml`

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: tenant-be
spec:
  replicas: 1
  selector:
    matchLabels:
      app: tenant-be
  template:
    metadata:
      labels:
        app: tenant-be
    spec:
      containers:
        - name: tenant-be
          image: tenant-be:latest
          imagePullPolicy: Never
          ports:
            - containerPort: 8080
---
apiVersion: v1
kind: Service
metadata:
  name: tenant-be
spec:
  selector:
    app: tenant-be
  ports:
    - protocol: TCP
      port: 80
      targetPort: 8080
  type: NodePort
```

Why Deploys and exposes the tenant backend.

What Deploys `tenant-be` on port 8080 and publishes it via a NodePort `Service` (80 -> 8080).

Role Integrates the tenant backend into the cluster.

Listing 35.: tenantFE Manifest: `/k8s/deployments/tenant-fe.yaml`

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: tenant-fe
  labels:
    app: tenant-fe
spec:
  replicas: 1
  selector:
    matchLabels:
      app: tenant-fe
  template:
    metadata:
      labels:
        app: tenant-fe
    spec:
      containers:
        - name: tenant-fe
          image: tenant-fe:latest
          imagePullPolicy: Never
          ports:
            - containerPort: 3000
          env:
            - name: NEXT_PUBLIC_CONFIG_ENDPOINT
              value: http://tenant-be.default.svc.cluster.local/config
            - name: NEXT_PUBLIC_CRASH_ENDPOINT
              value: http://tenant-be.default.svc.cluster.local/crash
---
apiVersion: v1
kind: Service
metadata:
  name: tenant-fe
spec:
  selector:
    app: tenant-fe
  ports:
    - protocol: TCP
```

```
38       port: 80
39       targetPort: 3000
40   type: NodePort
```

Why Deploys and exposes the tenant frontend.

What Deploys `tenant-fe` on port 3000 and publishes it via a NodePort `Service` (80 -> 3000). Injects backend URLs via `NEXT_PUBLIC_CONFIG_ENDPOINT` and `NEXT_PUBLIC_CRASH_ENDPOINT` at runtime.

Role Integrates the tenant frontend into the cluster.

Listing 36.: Debug Pod Manifest: `/k8s/deployments/debug.yaml`

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: debug
spec:
  containers:
    - name: debug
      image: nicolaka/netshoot
      command: [ "sleep", "infinity" ]
      stdin: true
      tty: true
  restartPolicy: Never
```

Why On-cluster troubleshooting without bundling tools into app images.

What Starts a throwaway Netshoot Pod that sleeps forever with TTY+stdin.

Role Operational helper to debug network/DNS/service reachability inside the cluster.

G.7. Workspaces

Listing 37.: Create Dashboard Workspace: `/scripts/create-dashboard-workspace.sh`

```
#!/usr/bin/env bash

set -euo pipefail

# Make sure kubectl + plugins are in PATH
export PATH="$HOME/Dokumente/BachelorThesis_Infra/bin:$PATH"
export PATH="${KREW_ROOT:-$HOME/.krew}/bin:$PATH"

# Set KUBECONFIG
export PATH=$PATH:/home/mysterion/Dokumente/BachelorThesis_Infra/contrib-tmc/bin

# Create the workspaces and enter dashboard workspace
kubectl ws use :root
kubectl tmc workspace create dashboard --type tmc
kubectl create-workspace tenants --type universal
kubectl ws use :root:dashboard

# Create the kind cluster for the dashboard
kind create cluster --config ./k8s/cluster/kind-config.yaml

# Build dashboardBE docker image
./scripts/build/dashboardBE.sh

# Load image to kind cluster
kind load docker-image dashboard-be:latest --name dashboard

# Apply deployment
kubectl config use-context kind-dashboard
kubectl apply -f ./k8s/deployments/dashboard-be.yaml

# Build dashboardFE docker image
./scripts/build/dashboardFE.sh http://dashboard-be.default.svc.cluster.local/config

# Load image to kind cluster
kind load docker-image dashboard-fe:latest --name dashboard

# Apply deployment
```

```
kubectl apply -f ./k8s/deployments/dashboard-fe.yaml

# Ingress (not needed for development, but useful for production)
# kubectl apply -f https://raw.githubusercontent.com/kubernetes/ingress-nginx/controller-v1.10
↪  .1/deploy/static/provider/kind/deploy.yaml
# kubectl label node dashboard-control-plane ingress-ready=true
# kubectl wait --namespace ingress-nginx \
    # --for=condition=Ready pod \
    # --selector=app.kubernetes.io/component=controller \
    # --timeout=90s
# sudo bash ./scripts/add-to-hosts.sh 127.0.0.1 dashboard.local api.dashboard.local
# kubectl apply -f ./k8s/ingress/dashboard-ingress.yaml
# sudo iptables -t nat -A OUTPUT -p tcp --dport 80 -d 127.0.0.1 -j REDIRECT --to-port 30863

# Create role binding and service account for dashboard
kubectl apply -f ./k8s/roleBinding/tmc-role-binding.yaml

# Register sync target
kubectl config use-context root
kubectl tmc ws use root:dashboard
kubectl tmc workload sync kind-dashboard \
    --syncer-image ghcr.io/kcp-dev/contrib-tmc/syncer:latest \
    --output-file kind-dashboard.yaml
kubectl config use-context kind-dashboard
kubectl apply -f kind-dashboard.yaml
kubectl config use-context root
kubectl tmc bind compute root:dashboard
kubectl ws use :root:dashboard
kubectl apply -f ./k8s/apibindings/standard-bindings.yaml
```

Why One-shot bootstrap for the dashboard control plane, so anyone can reproduce the full setup quickly and consistently.

What Exports plugin paths, creates KCP workspaces (:root, dashboard, tenants), spins up a kind cluster, builds/loads FE + BE images, applies deployments / services, grants RBAC, generates+applies the TMC syncer, binds compute, and applies standard APIBindings.

Role Provisioning glue between control plane (KCP workspaces) and data plane (kind). It installs and wires the dashboard stack that operators use to manage tenant configs.

Listing 38.: Create Tenant Workspace: `/scripts/create-tenant-x-workspace.sh`

```
#!/usr/bin/env bash

set -euo pipefail

# TENANT ID
TENANT_ID="${1}"

# Check if TENANT_ID is provided
if [ -z "$TENANT_ID" ]; then
    echo "<unicode-cancel> Tenant ID is required as the first argument. Usage:
    ↪  ./scripts/create-tenant-x-workspace.sh <TENANT_ID>"
    exit 1
fi

# Make sure kubectl + plugins are in PATH
export PATH="$HOME/Dokumente/BachelorThesis_Infra/bin:$PATH"
export PATH="${KREW_ROOT:-$HOME/.krew}/bin:$PATH"

# Set KUBECONFIG
export PATH=$PATH:/home/mysterion/Dokumente/BachelorThesis_Infra/contrib-tmc/bin

kubectl config use-context root

# Navigate to the root workspace
kubectl ws use :root:tenants

# Create the tenant workspace
kubectl tmc workspace create "tenant-${TENANT_ID}" --type tmc
kubectl ws use :root:tenants:tenant-${TENANT_ID}

# Create the kind cluster for the tenant
./scripts/create-tenant-x-cluster.sh "${TENANT_ID}"

# Build tenantBE docker image
./scripts/build/tenantBE.sh

# Load image to kind cluster
```

```
kind load docker-image tenant-be:latest --name tenant-${TENANT_ID}

# Apply deployment
kubectl config use-context kind-tenant-${TENANT_ID}
kubectl apply -f ./k8s/deployments/tenant-be.yaml

# Build tenantFE docker image
./scripts/build/tenantFE.sh ${TENANT_ID} http://tenant-be.default.svc.cluster.local

# Load image to kind cluster
kind load docker-image tenant-fe:latest --name tenant-${TENANT_ID}

# Apply deployment
kubectl apply -f ./k8s/deployments/tenant-fe.yaml

# Create CRD for the tenant
./scripts/CRDs/create-tenant-x-CRD.sh "${TENANT_ID}"

# Create role binding and service account for tenant
kubectl apply -f ./k8s/roleBinding/tmc-role-binding.yaml
```

Why One-command, repeatable provisioning of a new tenant so onboarding is fast and consistent.

What Validates the tenant id, creates a KCP workspace under:root:tenants, spins up a per-tenant kind cluster, builds and loads FE / BE images, applies deployments, creates the tenant-specific CRD, and assigns RBAC.

Role Tenant bootstrap orchestrator. Wires the control plane to the tenant's runtime cluster and installs the per-tenant app stack in isolation.

G.8. CRDs

Listing 39.: Create Tenant CRD Script: `/scripts/CRDs/create-tenant-x-CRD.sh`

```
#!/usr/bin/env bash
set -euo pipefail

# <unicode-robot> Configurable values
ROOT_WS=":root"
TENANT_ID="$1"
TENANT_WS=":root:tenants:tenant-${TENANT_ID}"
CRD_FILE="k8s/CRDs/tenant-${TENANT_ID}-crd.yaml"
EXPORT_FILE="k8s/CRDs/tenant-${TENANT_ID}-export.yaml"

# Check if TENANT_ID is provided
if [ -z "$TENANT_ID" ]; then
    echo "<unicode-cancel> Tenant ID is required as the first argument. Usage: $0 <TENANT_ID>"
    exit 1
fi

# CRD
cat > $CRD_FILE <<EOF
apiVersion: apiextensions.k8s.io/v1
kind: CustomResourceDefinition
metadata:
    name: tenantconfigs-${TENANT_ID}.example.com
spec:
    group: example.com
    scope: Namespaced
    names:
        plural: tenantconfigs
        singular: tenantconfig
        kind: TenantConfig
    versions:
        - name: v1alpha1
            served: true
            storage: true
            schema:
                openAPIV3Schema:
                    type: object
                    properties:
```

```
38                      spec:
39                          type: object
40                          required:
41                              - address
42                              - companyName
43                              - proposition
44                              - products
45                              - about
46                          properties:
47                              address:
48                                  type: object
49                                  required:
50                                      - country
51                                      - zipCode
52                                      - city
53                                      - street
54                                      - streetNumber
55                                  properties:
56                                      country:
57                                          type: string
58                                          pattern: '^[a-z]{2}-[A-Z]{2}$'
59                                      zipCode:
60                                          type: string
61                                          pattern: '^[0-9]{4,10}$'
62                                      city:
63                                          type: string
64                                          minLength: 1
65                                      street:
66                                          type: string
67                                          minLength: 1
68                                      streetNumber:
69                                          type: string
70                                          pattern: '^[0-9]+[a-zA-Z]?$'
71                              companyName:
72                                  type: string
73                                  minLength: 1
74                              proposition:
75                                  type: string
76                                  minLength: 1
77                              products:
```

```
                                    type: array
                                    minItems: 1
                                    items:
                                        type: string
                                about:
                                    type: string
                                    minLength: 1
EOF

# Export
cat > "$EXPORT_FILE" <<EOF
apiVersion: apis.kcp.io/v1alpha1
kind: APIExport
metadata:
    name: tenantconfigs-${TENANT_ID}.example.com
spec:
    latestResourceSchemas:
        - example.com_tenantconfigs
EOF
```

Why Enforce a typed, cluster-native contract for tenant configuration and make it consumable across workspaces in KCP.

What Generates a namespaced CRD `TenantConfig` (schema mirrors the NPM `ConfigSchema`) and an `APIExport`. Writes both YAML files under `k8s/CRDs/tenant-ID-*.yaml`.

Role Defines the source of truth for per-tenant config and publishes it via KCP so other workspaces can bind and validate it.

G.9. Role Binding

Listing 40.: TMC Role Binding: `/k8s/roleBinding/tmc-role-binding`

```yaml
apiVersion: v1
kind: Namespace
metadata:
  name: tmc-system
---
apiVersion: v1
kind: ServiceAccount
metadata:
  name: kcp-syncer
  namespace: tmc-system
---
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: kcp-syncer-role
rules:
  - apiGroups: ["*"]
    resources: ["*"]
    verbs: ["*"]
---
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: kcp-syncer-binding
subjects:
  - kind: ServiceAccount
    name: kcp-syncer
    namespace: tmc-system
roleRef:
  kind: ClusterRole
  name: kcp-syncer-role
  apiGroup: rbac.authorization.k8s.io
```

Why Authorize the KCP syncer to operate in the runtime cluster so compute binding can actually apply resources.

What Creates namespace `tmc-system`, ServiceAccount `kcp-syncer`, a broad `ClusterRole` over all resources, and binds that role.

Role Gives the sync target RBAC to mirror K8s objects between workspace and cluster (wildcard perms for prototype).

G.10. Ingress

Listing 41.: Dashboard Ingress: `/k8s/ingress/dashboard-ingress.yaml`

```yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: dashboard-fe-ingress
  annotations:
    nginx.ingress.kubernetes.io/rewrite-target: /
spec:
  ingressClassName: nginx
  rules:
    - host: dashboard.local
      http:
        paths:
          - path: /
            pathType: Prefix
            backend:
              service:
                name: dashboard-fe
                port:
                  number: 80
---
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: dashboard-be-ingress
spec:
  ingressClassName: nginx
  rules:
    - host: api.dashboard.local
      http:
        paths:
          - path: /
            pathType: Prefix
            backend:
              service:
                name: dashboard-be
                port:
                  number: 80
```

38

Why Expose the dashboard UI and API via friendly local hostnames, mirroring production routing.

What Defines two K8s Ingresses: `dashboard-fe-ingress` routes `dashboard.local` to Service `dashboard-fe`:80, `dashboard-be-ingress` routes `api.dashboard.local` to Service `dashboard-be`:80. Requires `ingress-nginx`.

Role Provides stable edge entrypoints and decouples external DNS from in-cluster Service details.

Listing 42.: Tenant Ingress: `/k8s/ingress/tenant-ingress.yaml`

```yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: tenant-ingress
  annotations:
    nginx.ingress.kubernetes.io/rewrite-target: /
spec:
  ingressClassName: nginx
  rules:
    - host: tenant.local
      http:
        paths:
          - path: /
            pathType: Prefix
            backend:
              service:
                name: tenant-fe
                port:
                  number: 80
---
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: tenant-be-ingress
spec:
  ingressClassName: nginx
  rules:
    - host: api.tenant.local
      http:
        paths:
          - path: /
            pathType: Prefix
            backend:
              service:
                name: tenant-be
                port:
                  number: 80
```

Why Expose the tenant UI and API via friendly local hostnames, mirroring production routing.

What Defines two K8s Ingresses: `tenant-fe-ingress` routes `tenant-ID.local` to Service `tenant-fe`:80, `tenant-be-ingress` routes `api.tenant-ID.local` to Service `tenant-be`:80. Requires `ingress-nginx`.

Role Provides stable edge entrypoints and decouples external DNS from in-cluster Service details.

G.11. Helpers

Listing 43.: Add to Hosts Script: `/scripts/add-to-hosts.sh`

```
#!/bin/bash

IP="$1"
shift

if [ -z "$IP" ] || [ "$#" -lt 1 ]; then
    echo "<unicode-cancel> Usage: $0 <IP_ADDRESS> <HOSTNAME_1> [HOSTNAME_2] ..."
    exit 1
fi

for HOST in "$@"; do
    if ! grep -qE "^[^#]*\s+$HOST(\s|$)" /etc/hosts; then
        echo "<unicode-plus> Adding $HOST → $IP to /etc/hosts"
        echo "$IP $HOST # added-by-script" | sudo tee -a /etc/hosts > /dev/null
    else
        echo "<unicode-check> $HOST already present in /etc/hosts"
    fi
done
```

Why Enable local name resolution for vanity domains during development without setting up external DNS.

What Appends `IP HOSTNAME` pairs to `/etc/hosts` if missing, handling multiple hosts and skipping duplicates (idempotent).

Role Local networking shim so dashboard and API hostnames resolve to the kind node / ingress IP on the developer machine.

Listing 44.: Cleanup Hosts Script: `/scripts/cleanup-hosts.sh`

```
#!/bin/bash

sudo sed -i '/# added-by-script$/d' /etc/hosts
```

Why Revert local DNS overrides to avoid stale / incorrect mappings after development.

What Deletes any `/etc/hosts` lines tagged with `# added-by-script` using `sed`.

Role Cleanup utility to restore the machine's default name resolution outside the dev environment.

Listing 45.: Restore kind context script: `/scripts/restore-kind-context`

```
#!/bin/bash
kind export kubeconfig --name dashboard
kind export kubeconfig --name tenant-1
```

Why Restore `kubectl` access to the dev clusters after context loss.

What Runs `kind export kubeconfig` for `dashboard` and `tenant-1`, merging credentials into the active kubeconfig.

Role Local ops helper to rehydrate control-plane and tenant cluster contexts.

H. Documentation Excerpts

H.1. KCP-TMC Quick Start Guide

Listing 46.: TMC Quick Start Guide

```
# Build CLI binaries

  make build

# Copy binaries to PATH

  # Either copy the binaries to a directory in your PATH
  cp bin/{kubectl-tmc,kubectl-workloads} /usr/local/bin/kubectl-tmc

  # Or add the bin directory to your PATH
  export PATH=$PATH:$(pwd)/bin

# Start TMC-KCP

  go run ./cmd/tmc start

# Create TMC workspace

  kubectl tmc workspace create tmc-ws --type tmc --enter

# Create SyncTarget for remote cluster

  kubectl tmc workload sync cluster-1 --syncer-image ghcr.io/kcp-dev/contrib-tmc/syncer:latest
  ↪  --output-file cluster-1.yaml

# Login into child cluster

  KUBECONFIG=<pcluster-config> kubectl apply -f "cluster-1.yaml"

# Bind compute resources

  kubectl tmc bind compute root:tmc-ws

# Create a workload on TMC-KCP cluster

```

```
35  kubectl create deployment kuard --image gcr.io/kuar-demo/kuard-amd64:blue
```

Why The quick start guide uses a KUARD demo workload rather than a local cluster, which makes the process of binding a workload to the TMC workspace opaque.

What A minimal path that builds the TMC/KCP CLI binaries, starts TMC-KCP, creates a TMC workspace, generates and installs a syncer on a child cluster, binds compute, and deploys the KUARD demo — thereby "making things work" without exposing the full lifecycle of a local cluster.

Role Documents the hidden assumptions and shortcuts that produced misleading "green" signals, explains why later steps (RBAC, APIBinding, CRDs, ingress) surfaced as hard-to-diagnose issues, and frames why the approach is not prescriptive for production.

I. Slack Channel Screenshots

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

I.1. Thread on TMC (2024–09–13)

This thread was the rationale for switching to TMC.

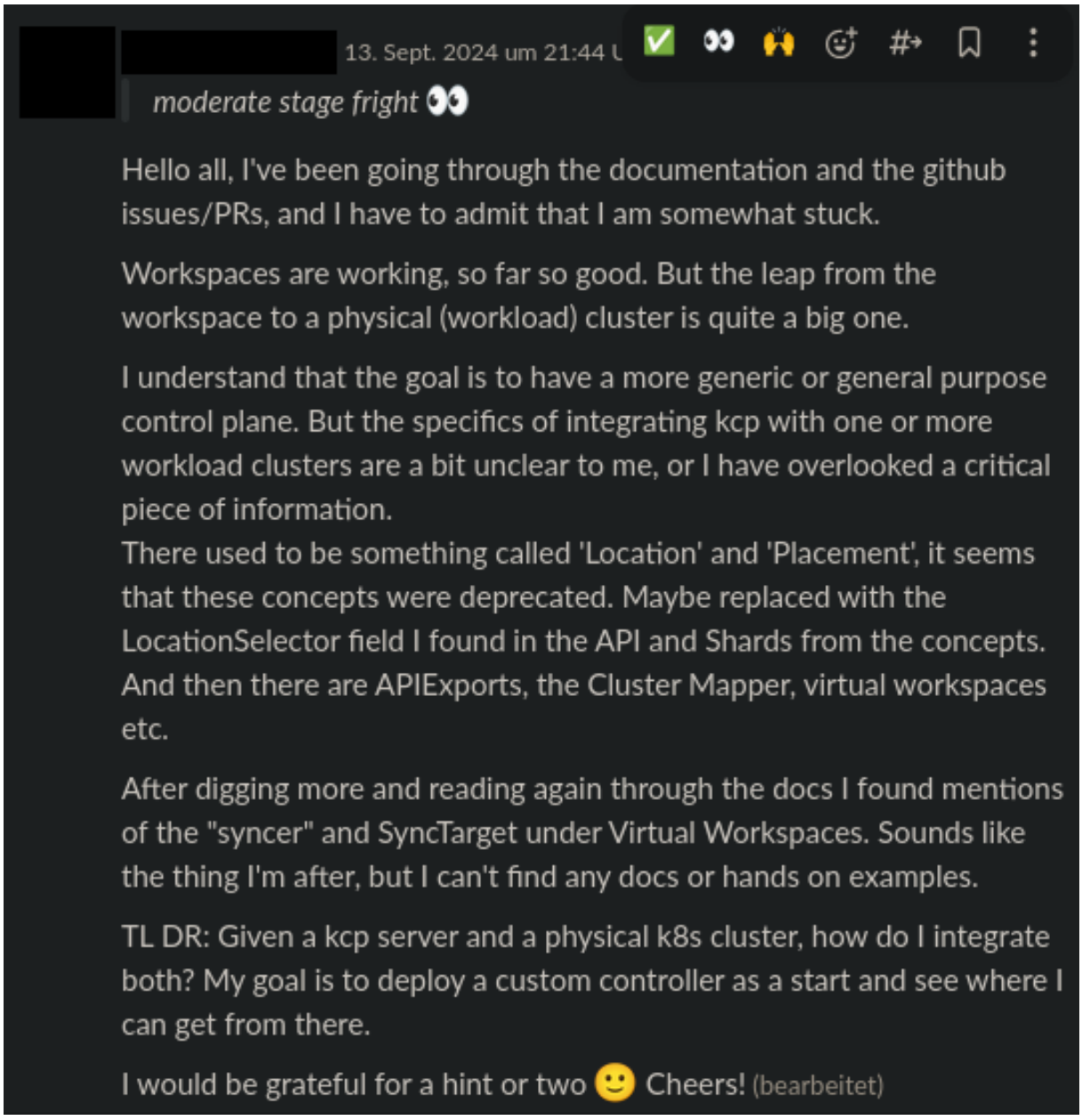

Figure 9: Thread on TMC (1)

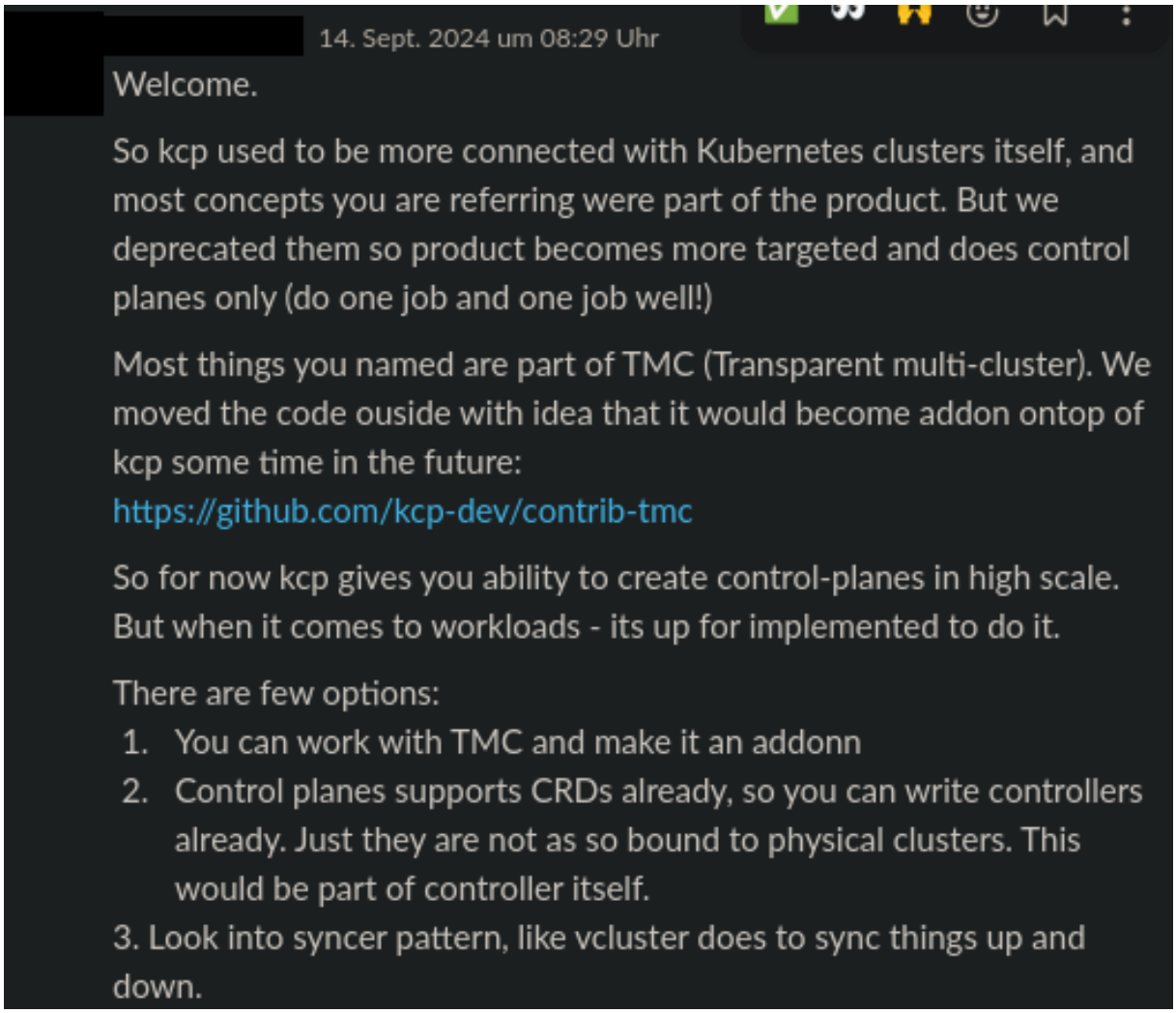

Figure 10: Thread on TMC (2)

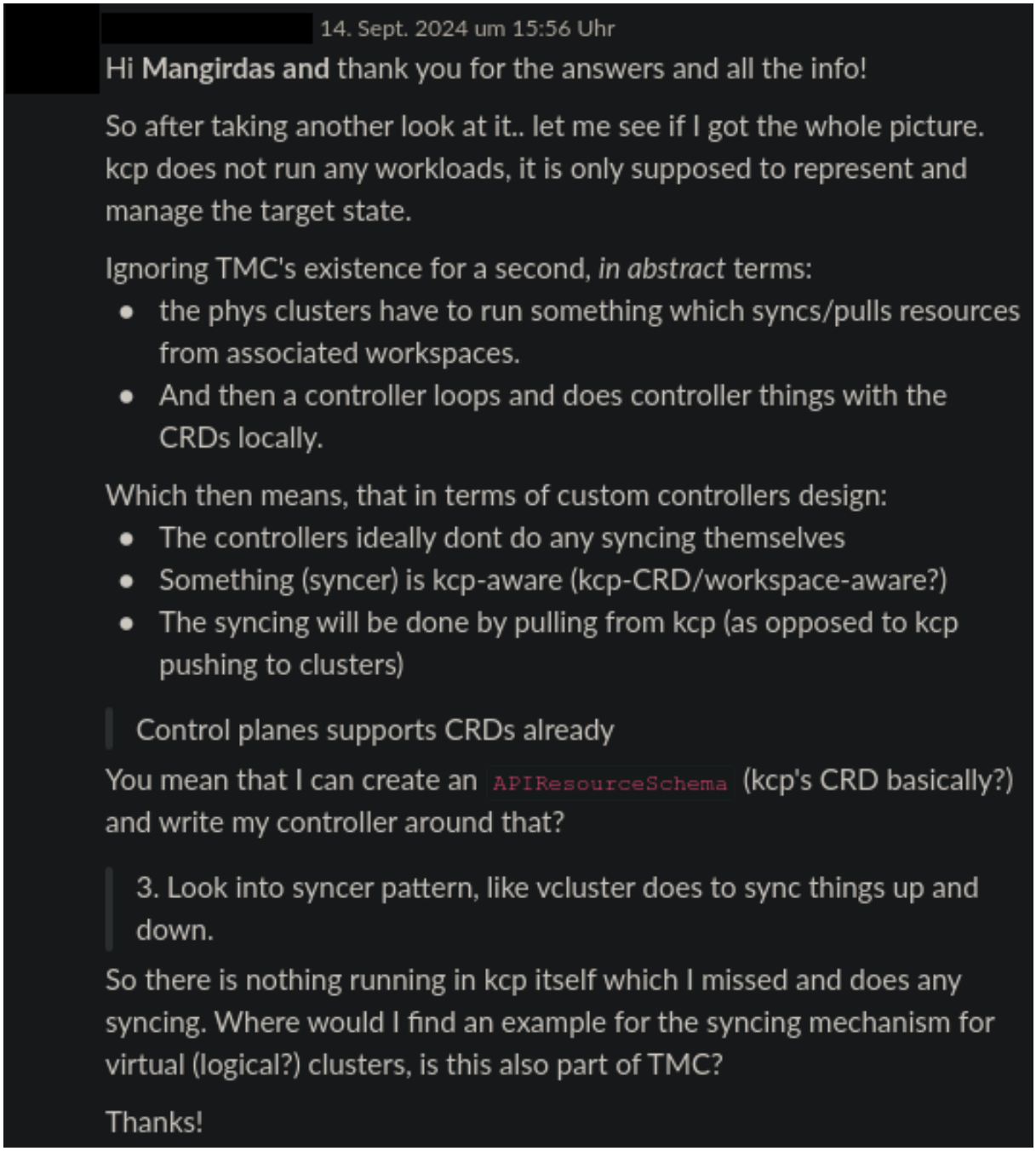

Figure 11: Thread on TMC (3)

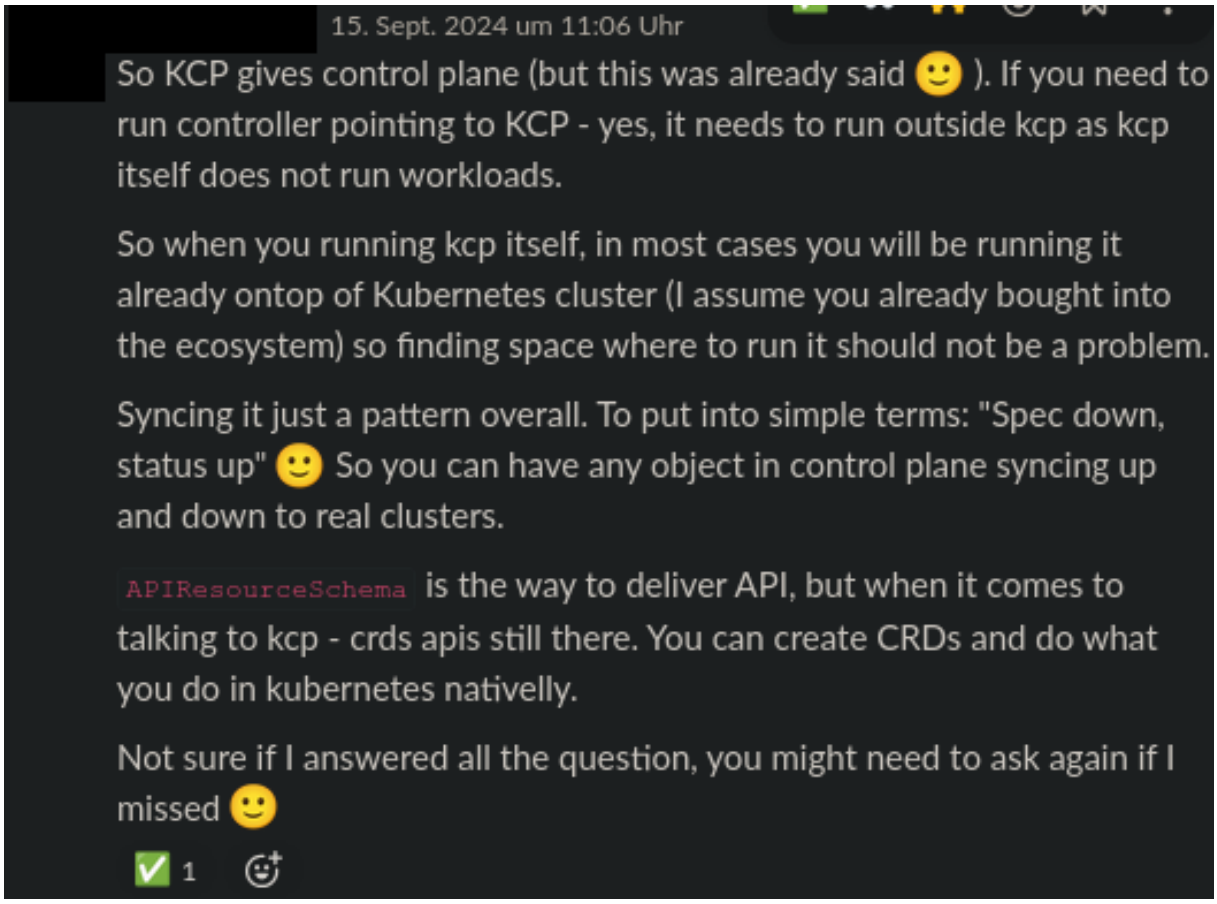

Figure 12: Thread on TMC (4)

I.2. Thread on Cross-workspace Resource Reconciliation (2025–08–13)

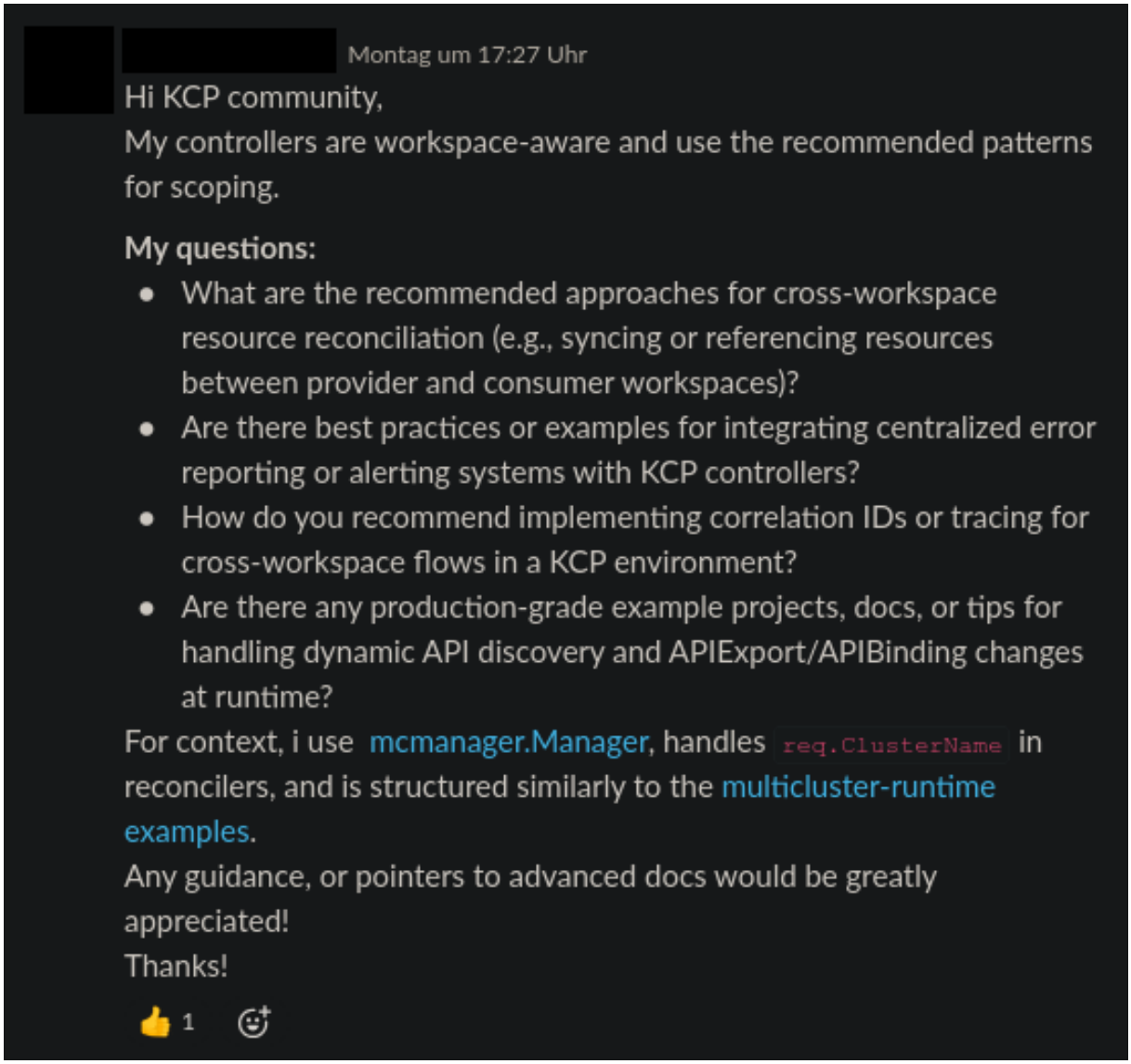

Figure 13: MCR related question

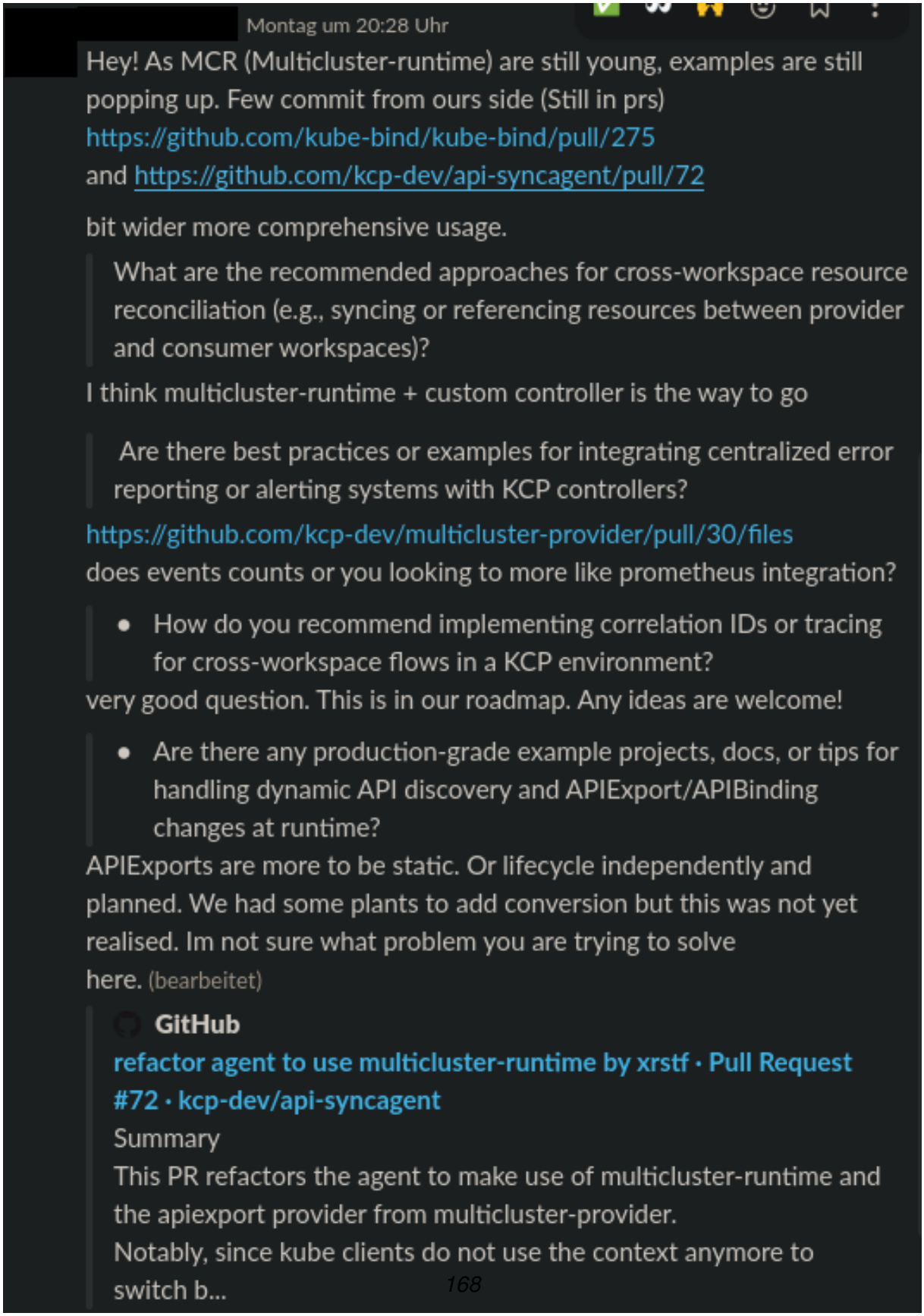

Figure 14: MCR related response

I.3. Thread on Workloads and TMC (2025–04–30)

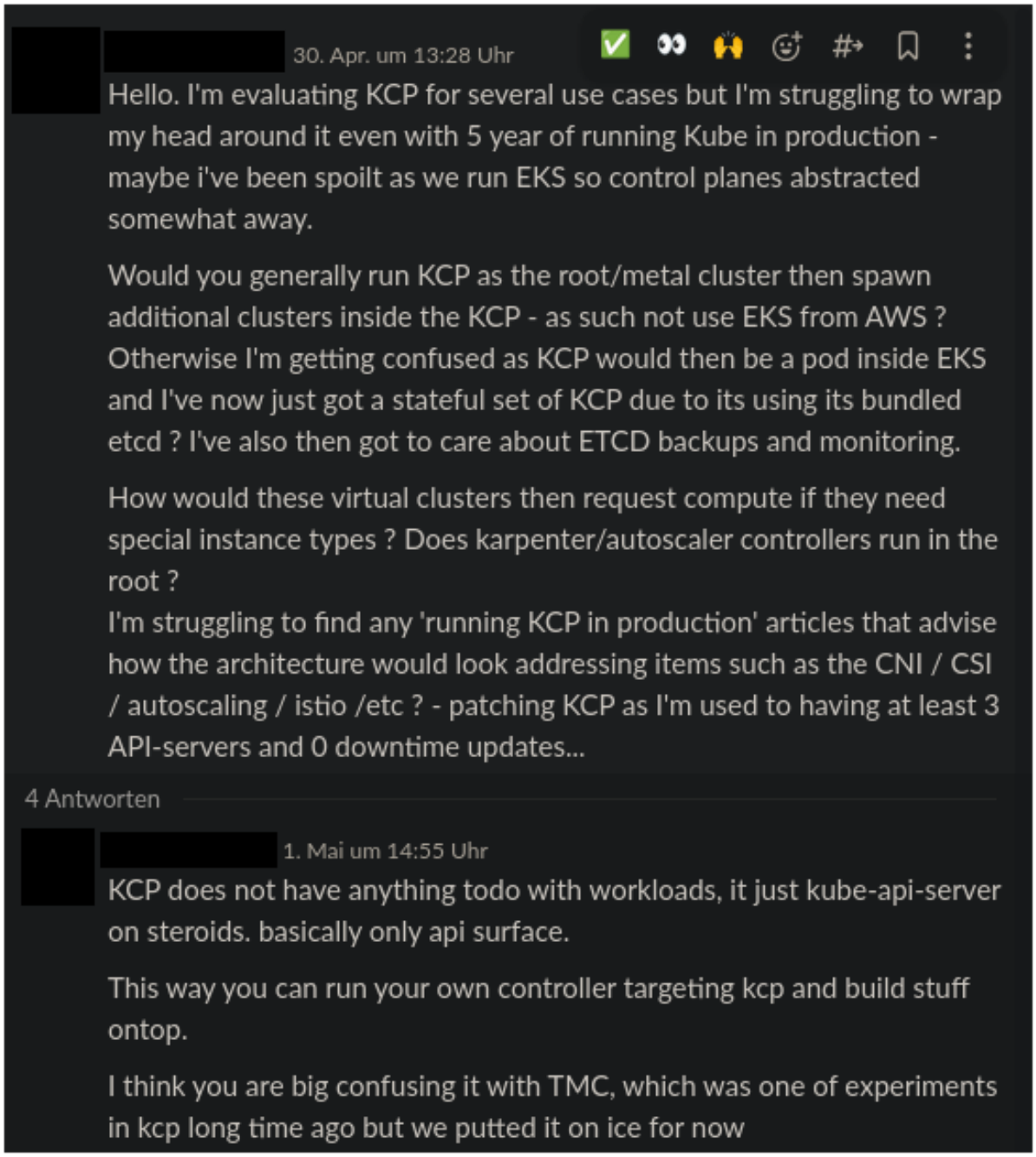

Figure 15: Thread on workloads and TMC

I.4. Thread on api-syncagent (2025–02–13)

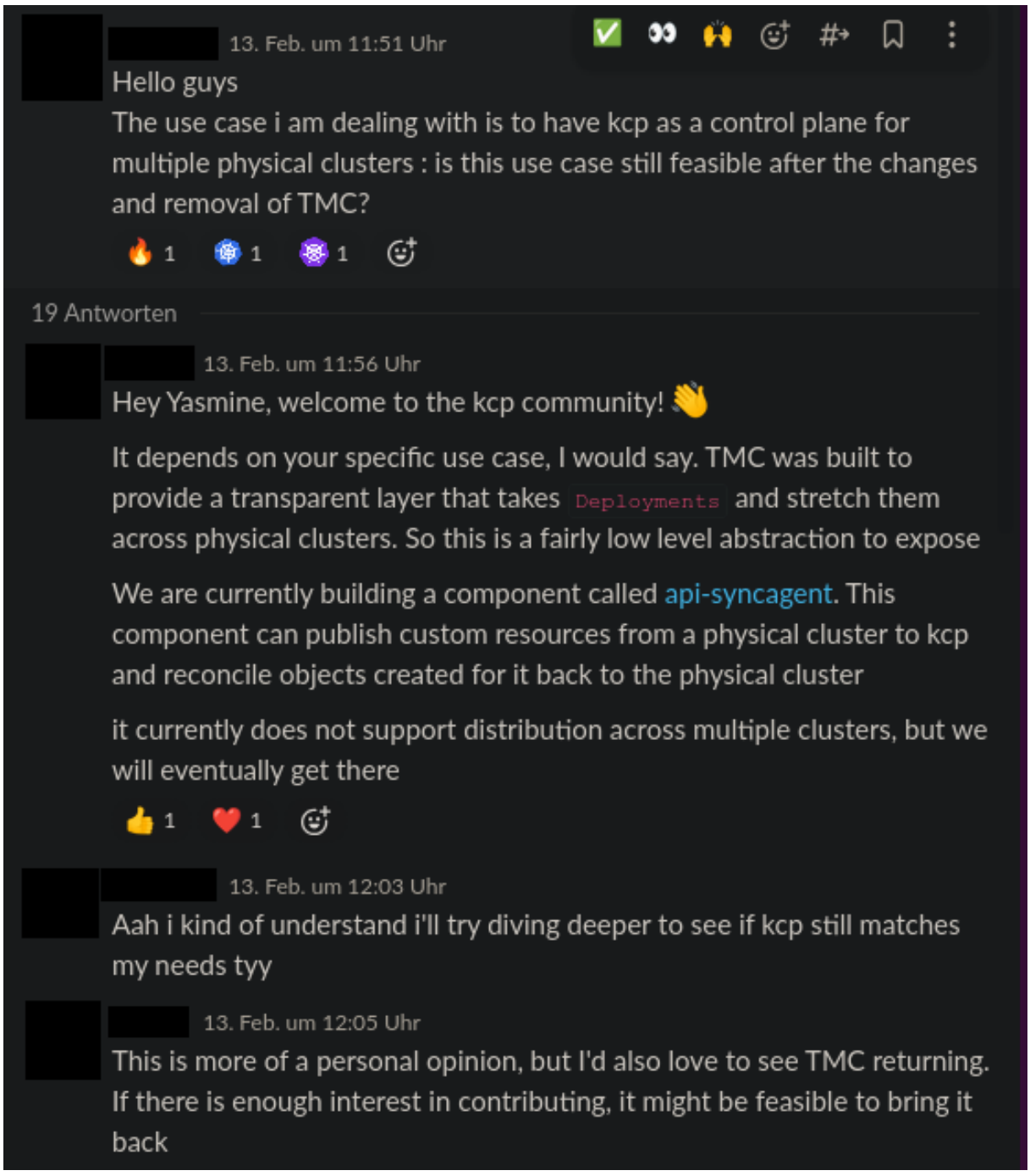

Figure 16: Thread on api-syncagent (1)

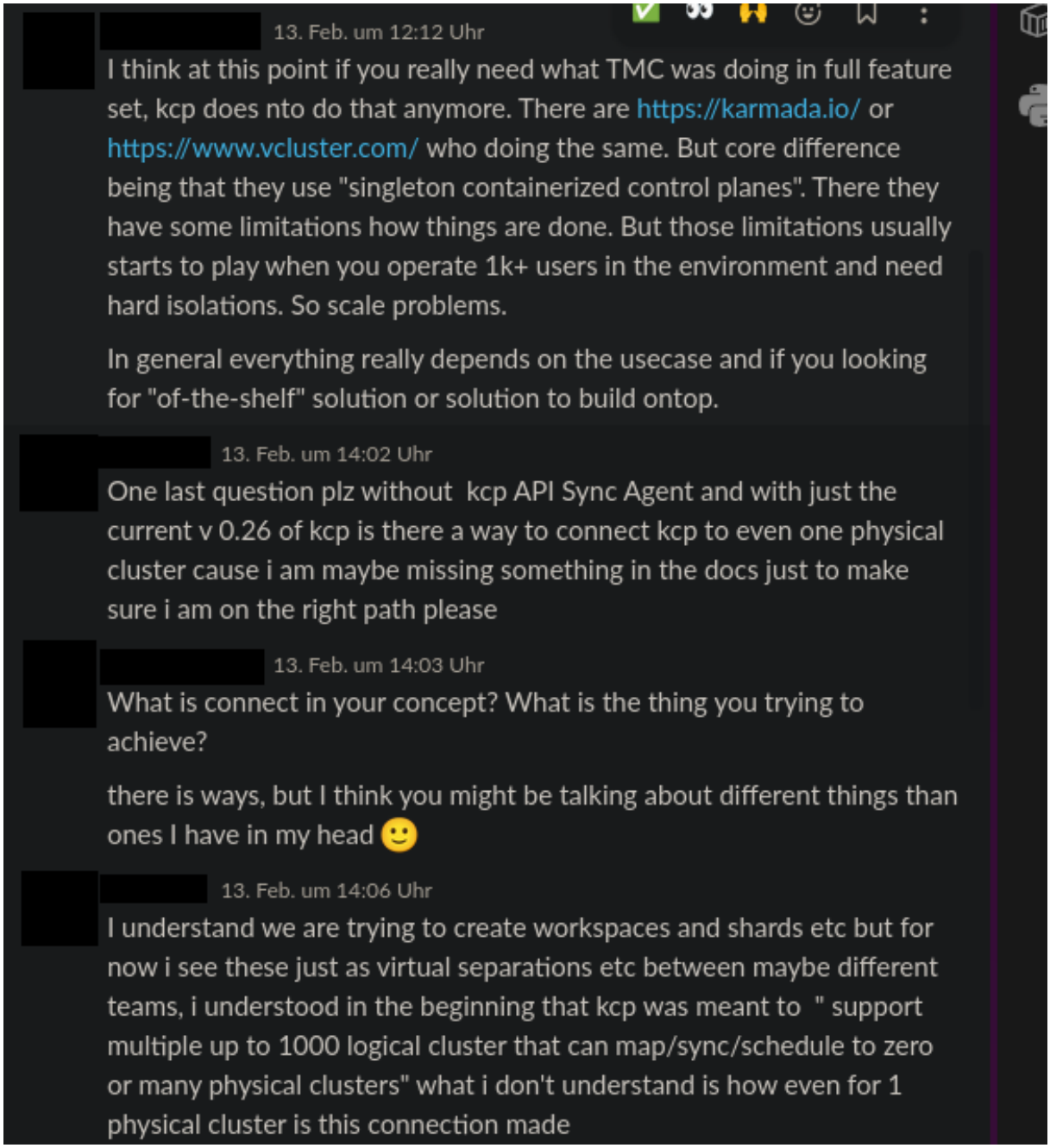

Figure 17: Thread on api-syncagent (2)

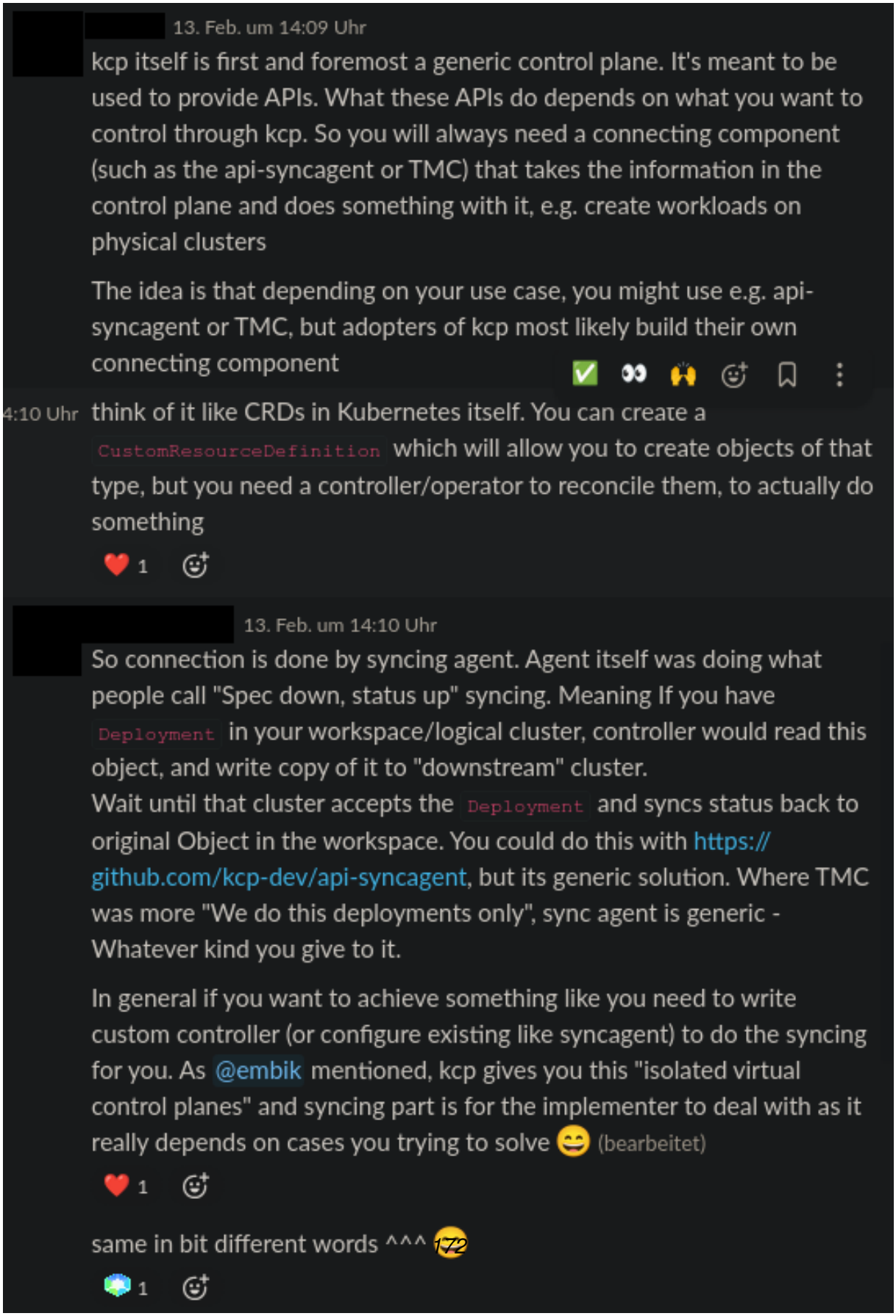

Figure 18: Thread on api-syncagent (3)

I.5. Thread on Removal of TMC (2023–07–21)

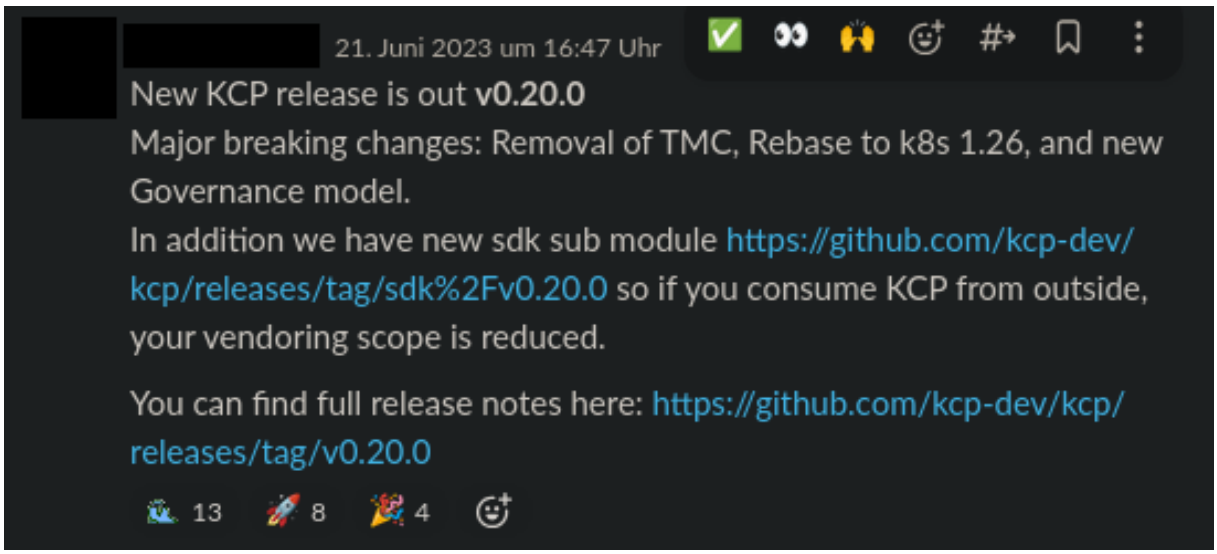

Figure 19: Removal of TMC Announcement

J. Evaluation

J.1. Dashboard FE

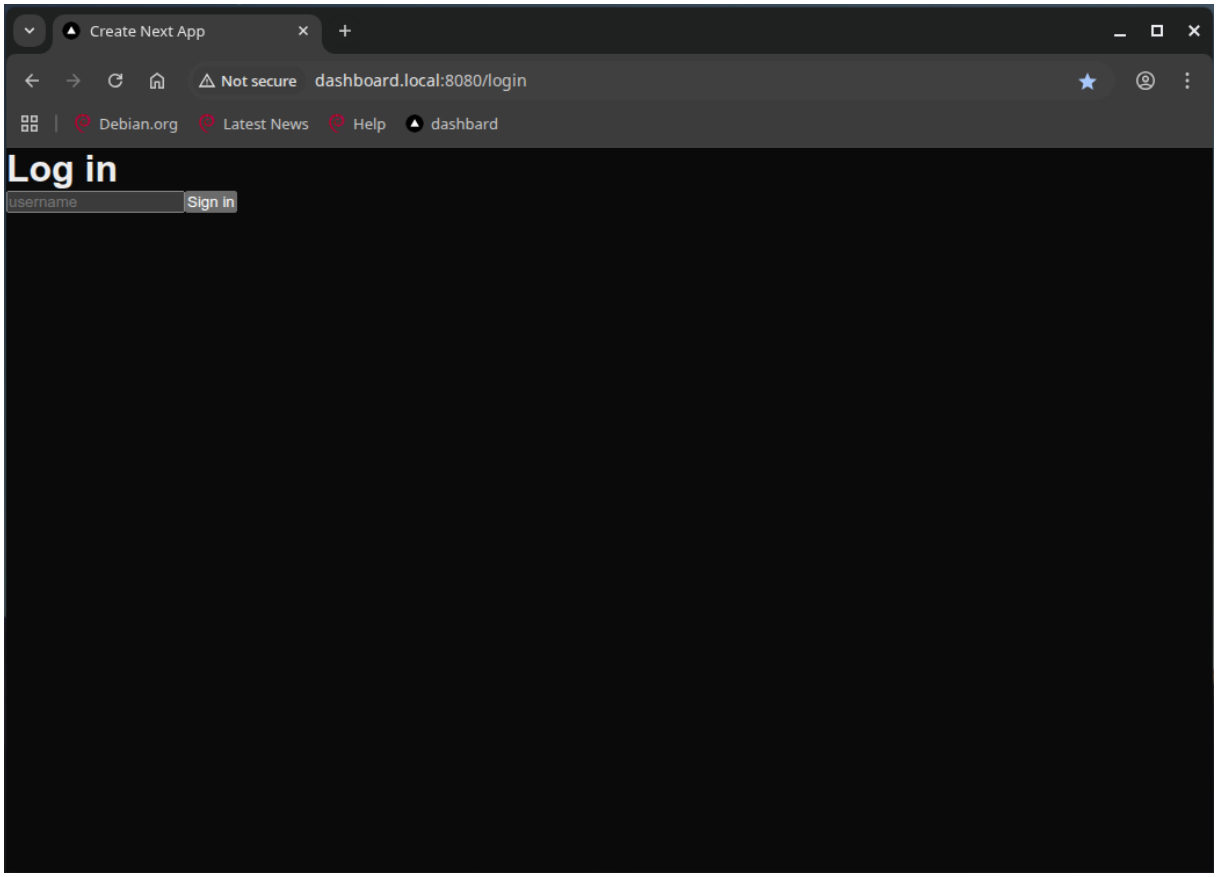

Figure 20: Login page

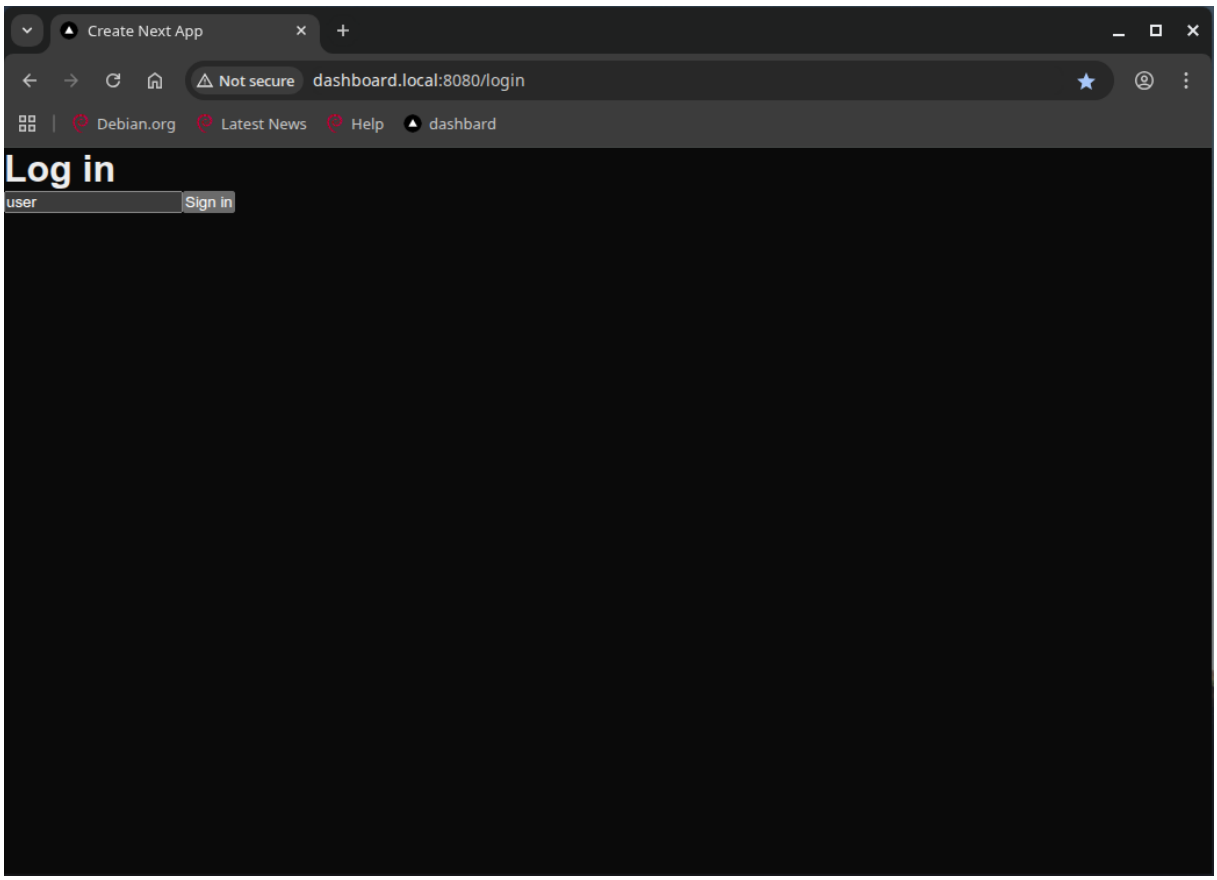

Figure 21: Login page with username

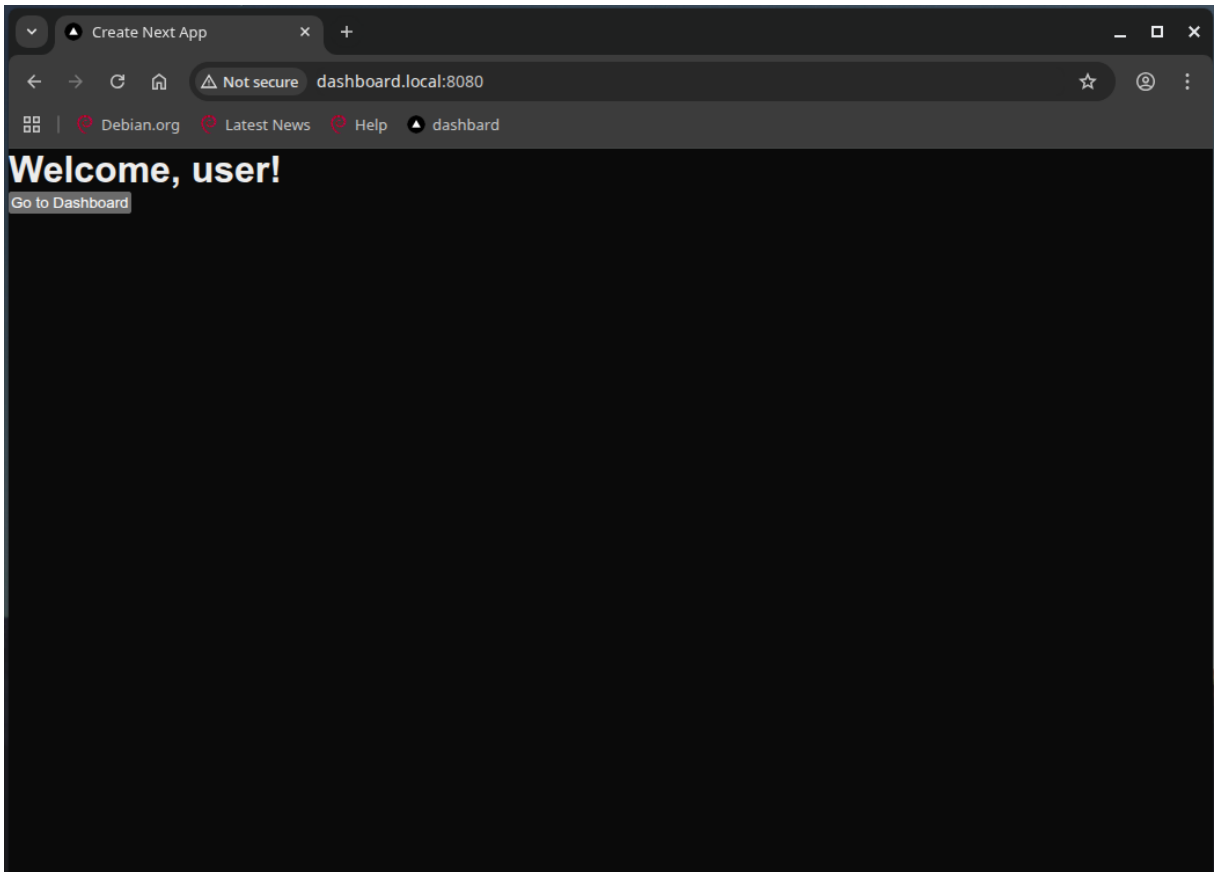

Figure 22: Welcome page

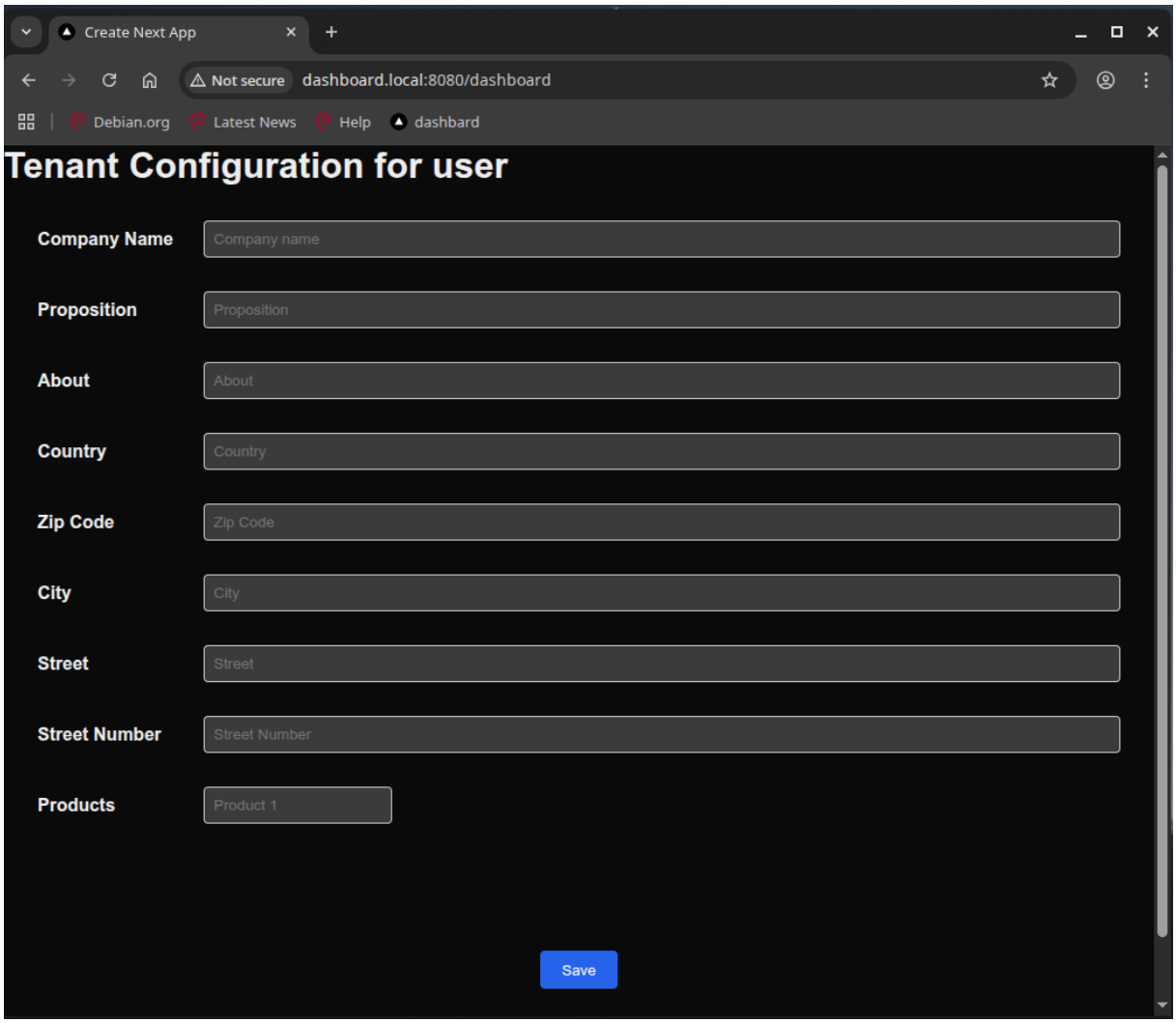

Figure 23: Form

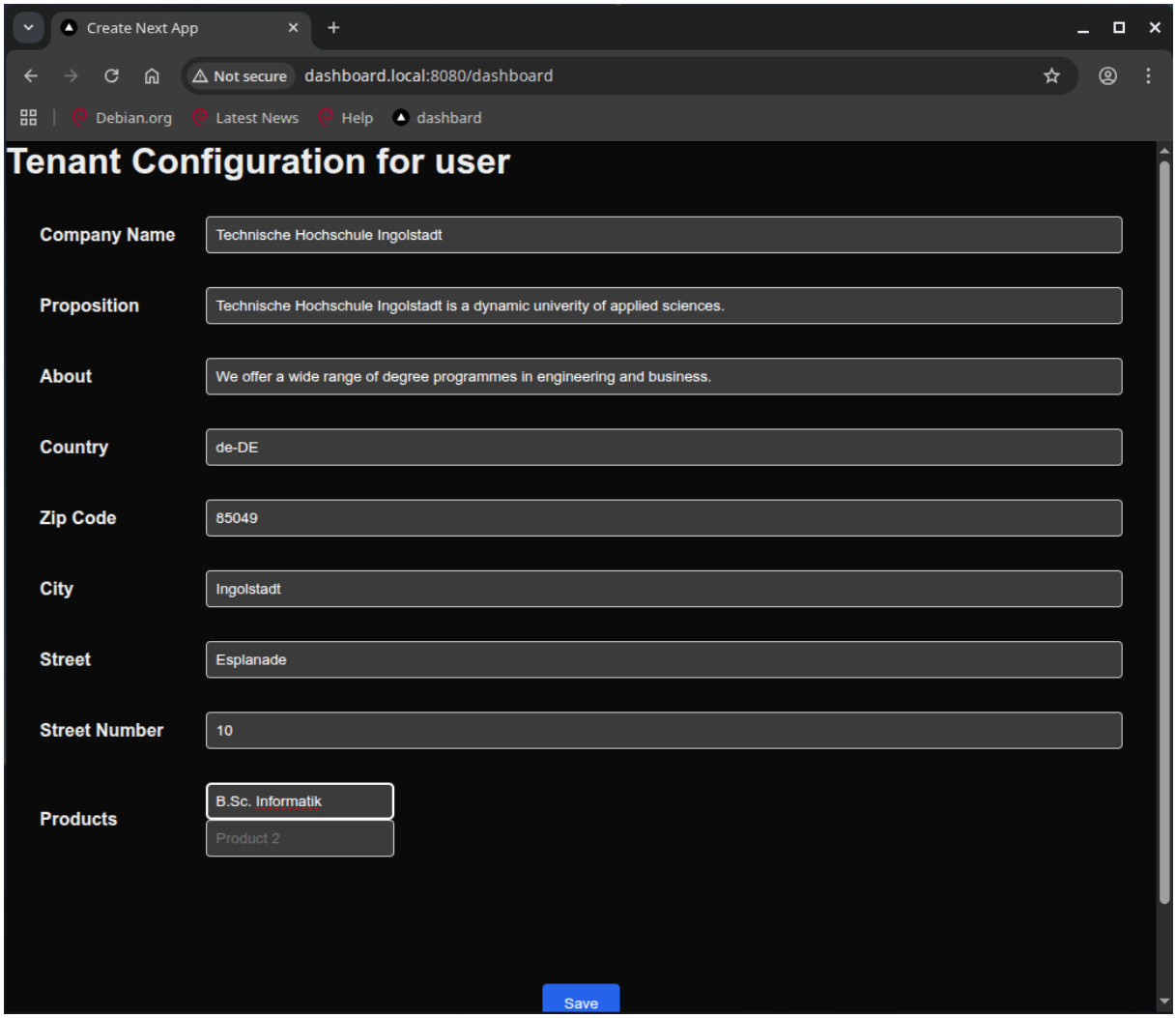

Figure 24: Form with inputs

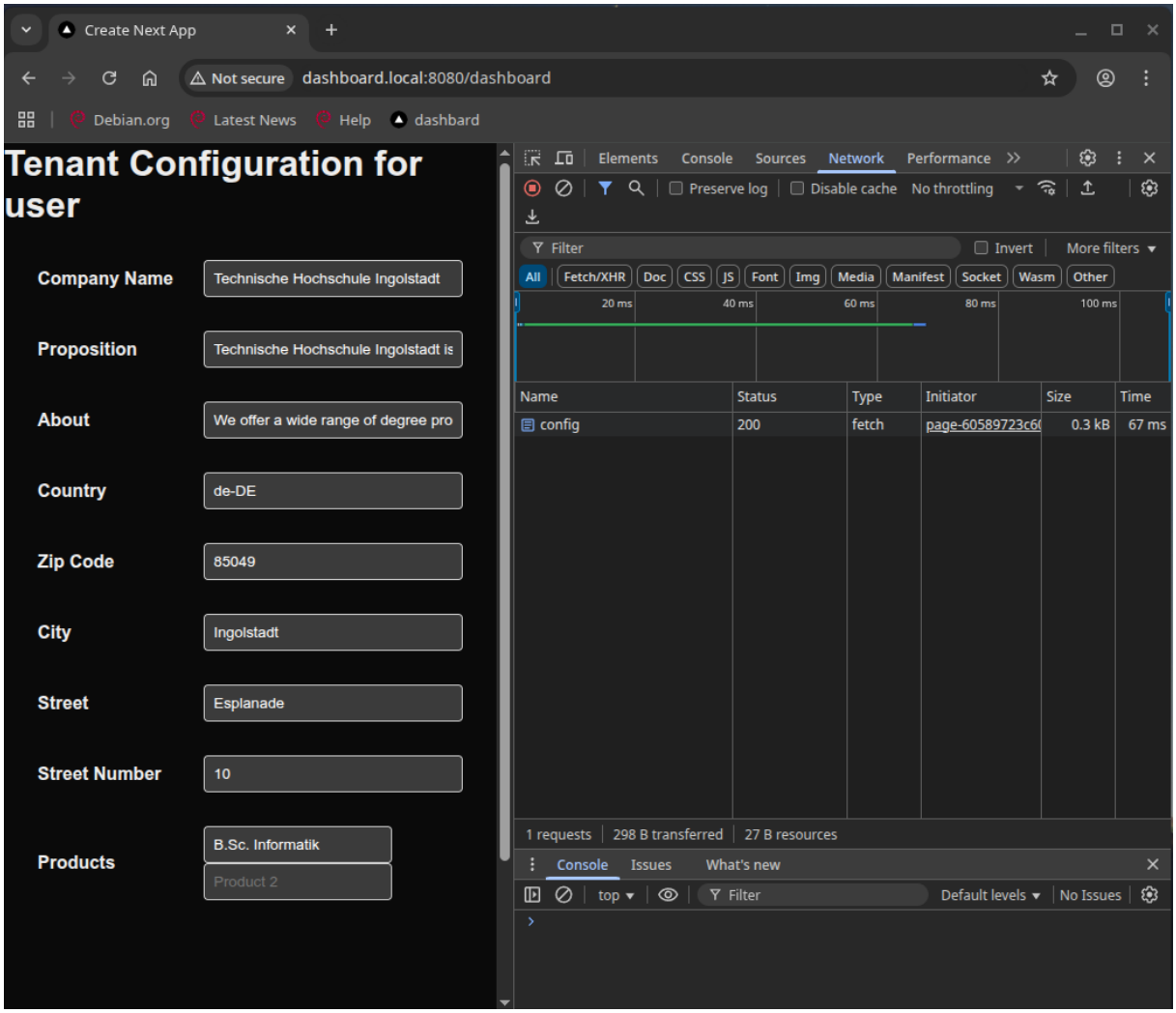

Figure 25: Form-success

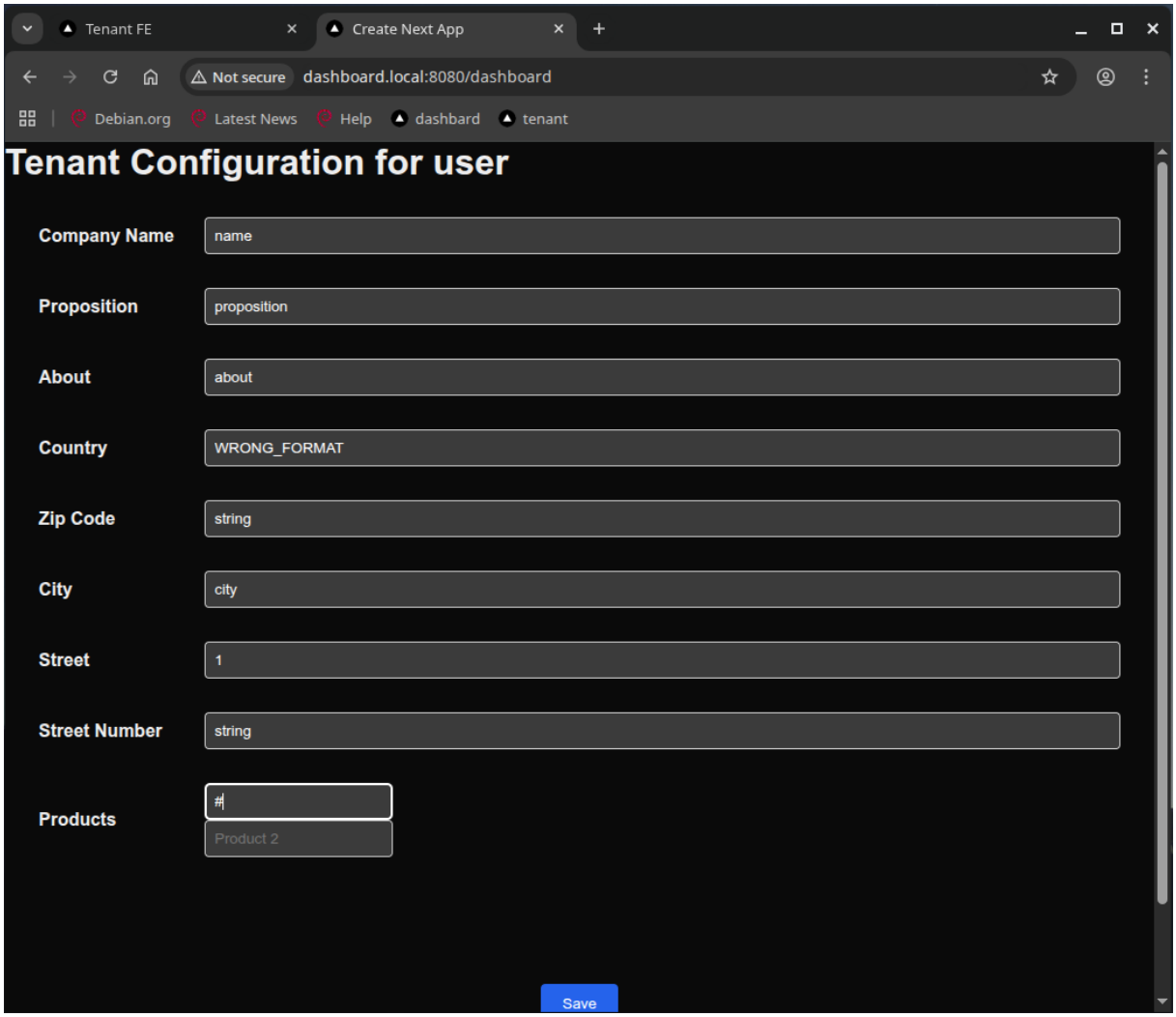

Figure 26: Form with malformed inputs

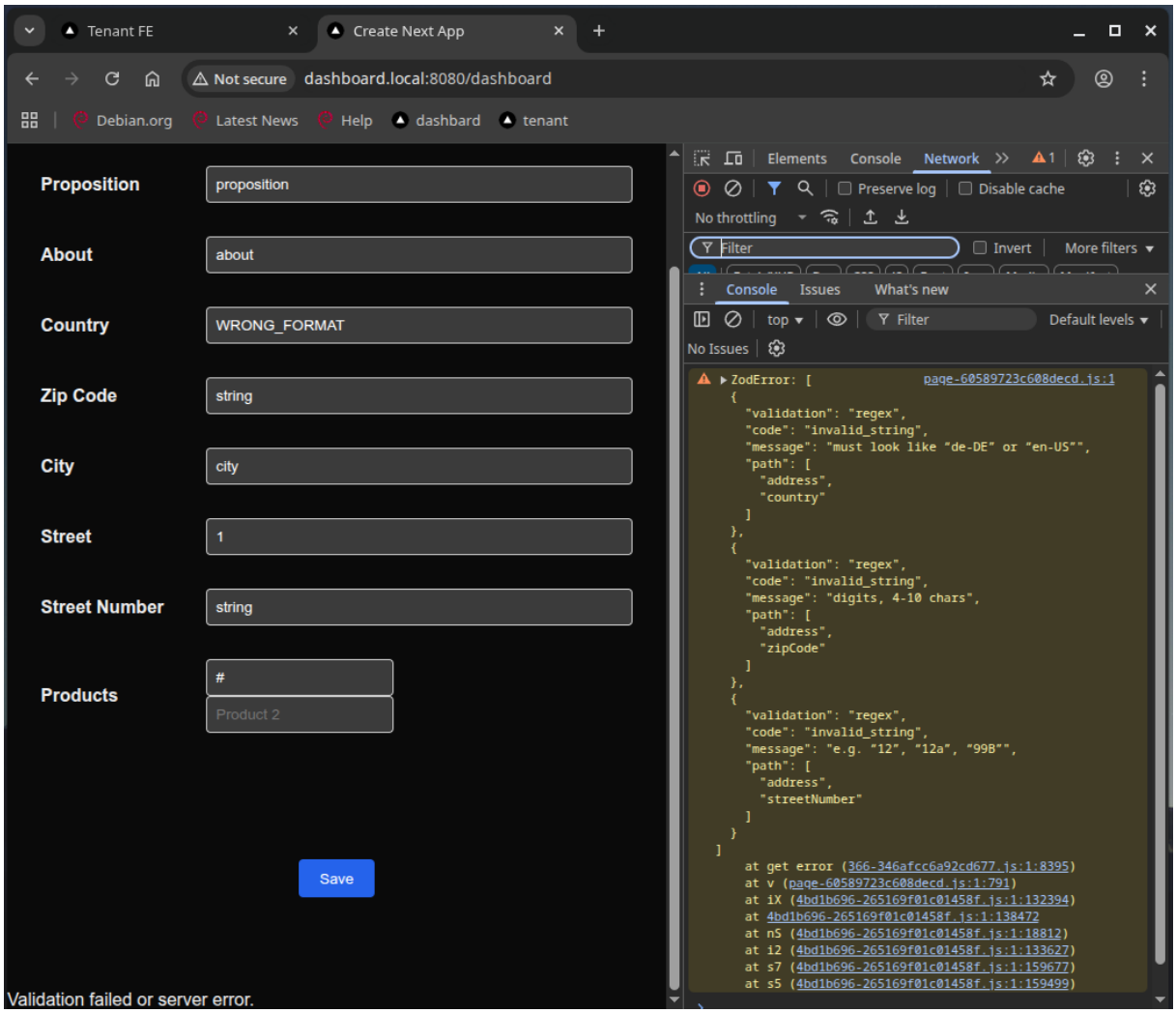

Figure 27: Form-validation failure

J.2. Dashboard BE

```
mysterion@debian-desktop:~/Dokumente/BachelorThesis_Infra$ kubectl config use-context kind-da
shboard
Switched to context "kind-dashboard".
mysterion@debian-desktop:~/Dokumente/BachelorThesis_Infra$ kubectl get pods
NAME                              READY   STATUS    RESTARTS       AGE
dashboard-be-75857d8d8-6k2t5      1/1     Running   6 (100m ago)   14d
dashboard-fe-58cd4f9746-vr8fs     1/1     Running   6 (100m ago)   14d
debug                             0/1     Unknown   0              20d
mysterion@debian-desktop:~/Dokumente/BachelorThesis_Infra$ kubectl exec -it dashboard-be-7585
7d8d8-6k2t5 -- cat ./data/user/config.json
{"address":{"country":"de-DE","zipCode":"85049","city":"Ingolstadt","street":"Esplanade","str
eetNumber":"10"},"companyName":"Technische Hochschule Ingolstadt","proposition":"Technische H
ochschule Ingolstadt is a dynamic univerity of applied sciences.","products":["B.Sc. Informat
ik"],"about":"We offer a wide range of degree programmes in engineering and business."}myster
ion@debian-desktop:~/Dokumente/BachelorThesis_Infra$
```

Figure 28: Config

```
mysterion@debian-desktop:~/Dokumente/BachelorThesis_Infra$ kubectl -n default exec -it debug
-- sh -lc \
'curl -i -sS -X POST \
  -H "Content-Type: application/json" \
  --data-raw "{\"companyName\":123,\"address\":null}" \
  http://dashboard-be.default.svc.cluster.local/config'
HTTP/1.1 400 Bad Request
X-Powered-By: Express
Access-Control-Allow-Origin: http://dashboard.local:8080
Vary: Origin
Access-Control-Allow-Credentials: true
Access-Control-Expose-Headers: user
Content-Type: text/html; charset=utf-8
Content-Length: 17
ETag: W/"11-bpDS431vEEMMm8bjEois79qj8bQ"
Date: Tue, 12 Aug 2025 15:07:06 GMT
Connection: keep-alive
Keep-Alive: timeout=5
```

Figure 29: Dashboard BE Bad Request (curl)

J.3. Tenant FE

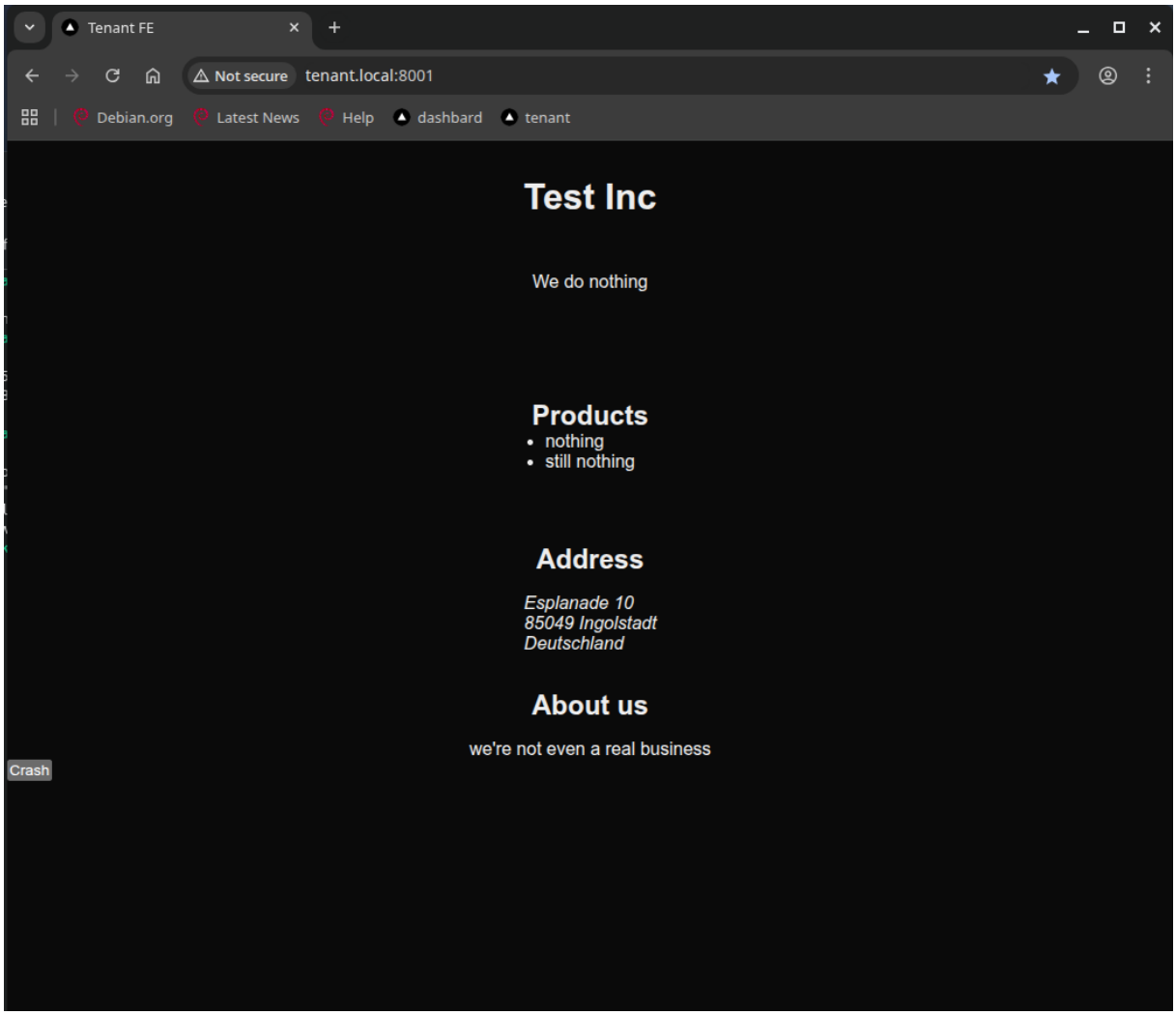

Figure 30: Tenant FE page

J.4. Tenant BE

```
mysterion@debian-desktop:~/Dokumente/BachelorThesis_Infra$ kubectl exec -it debug -- curl
 -sS http://tenant-be.default.svc.cluster.local/config
{"address":{"country":"de-DE","zipCode":"85049","city":"Ingolstadt","street":"Esplanade",
"streetNumber":"10"},"companyName":"Test Inc","proposition":"We do nothing","products":["
nothing","still nothing"],"about":"we're not even a real business"}mysterion@debian-deskt
op:~/Dokumente/BachelorThesis_Infra$
```

Figure 31: Tenant BE (curl)