FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Automated Backoffice for Managing Multi-Tenant Cloud Platforms

Luís Carlos Novais Alves

Mestrado em Engenharia Informática e Computação

Supervisor: Prof. Jorge Barbosa

July 28, 2026

Automated Backoffice for Managing Multi-Tenant Cloud Platforms

Luís Carlos Novais Alves

Mestrado em Engenharia Informática e Computação

Approved in oral examination by the committee:

President: Prof. Francisco Maia
Examiner: Miguel Areias
Supervisor: Prof. Jorge Barbosa

July 28, 2026

Resumo

As plataformas de software como serviço permitem que muitas organizações clientes partilhem uma mesma aplicação e infraestrutura, mantendo-se isoladas entre si. Operar estas plataformas de forma económica coloca em tensão duas exigências: concentrar muitos clientes, ou *tenants*, num único conjunto reduzido de recursos partilhados, e manter cada um isolado, de modo que nenhum perturbe ou aceda aos dados de outro. Esta dissertação estudou como automatizar a gestão de tenants e até onde um modelo partilhado, com um esquema de base de dados por tenant, pode ser levado antes de o isolamento ou a eficiência se degradarem, tomando como caso de estudo uma plataforma de suporte operacional em evolução para multi-tenancy.

Foi concebido e construído um módulo de backoffice que automatiza o ciclo de vida do tenant, da admissão e aprovação à configuração, às subscrições e à gestão de utilizadores, sobre uma arquitetura cloud-native em camadas, assente numa base de dados relacional e num serviço externo de identidade. Foi depois montado um ambiente experimental, combinando um cluster real, cargas sintéticas de tenants, escalonamento automático e *connection pooling*, para medir a densidade de tenants, a latência e o isolamento sob carga.

O estudo concluiu que o modelo partilhado cumpriu folgadamente o objetivo de densidade, que o fator limitante foi a camada de dados partilhada e não a camada aplicacional, e que o escalonamento horizontal não protegeu os tenants de um vizinho ruidoso, por a contenção estar no *connection pool* partilhado e não na computação. Estes resultados sugerem que, ao desenhar sistemas multi-tenant sobre bases de dados partilhadas, é a gestão do *connection pool*, e não o escalonamento de computação, a alavanca decisiva tanto para a eficiência como para o isolamento.

Palavras-chave: Multi-tenancy • Software as a Service (SaaS) • Gestão de tenants • Connection pooling • Kubernetes • Eficiência de recursos

Abstract

Software-as-a-service platforms let many customer organisations share a single application and infrastructure while staying isolated from one another. Running such platforms economically pulls in two competing directions: packing many customers, or tenants, onto one small set of shared resources to keep costs low, and keeping each tenant isolated so that none can disrupt or read another's data. This thesis studied how to automate tenant management and how far a shared model, with one database schema per tenant, can be pushed before isolation or efficiency breaks down, taking an operational support platform that was moving towards multi-tenancy as the case study.

A backoffice module was designed and built to automate the tenant lifecycle, from onboarding and approval to configuration, subscriptions, and user management, on a cloud-native, layered architecture backed by a relational database and an external identity service. An experimental setup was then assembled, combining a real cluster, synthetic tenant workloads, automatic scaling, and connection pooling, to measure tenant density, latency, and isolation under load.

The study found that the shared model comfortably met its density target, that the binding constraint was the shared data tier rather than the application tier, and that horizontal autoscaling did not protect tenants from a noisy neighbour, because the contention lay in the shared connection pool rather than in compute. These results suggest that, when designing multi-tenant systems over shared databases, it is connection-pool management, not compute autoscaling, that is the decisive lever for both efficiency and isolation.

Keywords: Multi-tenancy • Software as a Service (SaaS) • Tenant management • Connection pooling • Kubernetes • Resource efficiency

The United Nations Sustainable Development Goals

The United Nations Sustainable Development Goals (SDGs) provide a global framework for achieving a better and more sustainable future for all. It includes 17 goals (https://sdgs.un.org/goals) to address the world's most pressing challenges, including poverty, inequality, climate change, environmental degradation, peace, and justice.

This work contributes mainly to two goals. By consolidating tenant management into a single backoffice module and by characterising the pooled, schema-per-tenant model, it supports more reliable and resource-efficient cloud infrastructure (SDG 9). By increasing the number of tenants served per unit of compute, memory, and energy compared with a siloed deployment, it supports more responsible use of computing resources (SDG 12).

The specific Sustainable Development Goals mentioned have the following names:

SDG 9 Build resilient infrastructure, promote inclusive and sustainable industrialization and foster innovation

SDG 12 Ensure sustainable consumption and production patterns

SDG	Target	Contribution	Performance Indicators and Metrics
9	9.1	A unified tenant-manager module automates onboarding and lifecycle operations, making the platform's management layer more reliable and less error-prone.	Number of manual steps removed; share of lifecycle operations automated.
	9.4	The pooled, schema-per-tenant model raises resource-use efficiency, serving many tenants from shared infrastructure rather than a private stack per tenant.	Tenants served per footprint; CPU and memory consumed per tenant.
12	12.2	Connection pooling and server density reduce the compute and energy required to serve each tenant, lowering idle and overprovisioned capacity.	Tenant density on a shared instance; reduction in idle capacity.

Acknowledgements

I would like to thank my supervisor, Prof. Jorge Barbosa, for the guidance, availability, and constructive feedback that shaped this work from start to finish.

A special thanks to Ricardo Santos Ferreira, my supervisor at Altice Labs, for his patience, his availability, and everything I learned from him. I am grateful to Altice Labs for hosting this dissertation, and to all the colleagues I met there, for the everyday company and for the coffee breaks that made the long days lighter.

To the Tuna de Engenharia da Universidade do Porto, thank you for these five years: for the rehearsals, the performances, the trips, and the friendships that came with them. You became my second family.

Finally, and above all, I thank my family, who made all of this possible. They gave me the opportunity to be here and stood by me at every step.

Luís Carlos Novais Alves

Declaração de Honra

Declaro que a presente dissertação é de minha autoria e não foi previamente apresentada e avaliada nesta ou em outra instituição. As referências a outros autores (afirmações, ideias, pensamentos), assim como a utilização de trabalhos publicados respeitam escrupulosamente as regras da atribuição, e encontram-se devidamente indicadas no texto e nas referências bibliográficas, de acordo com as normas de referenciação. Tenho consciência de que a prática de plágio ou de autoplágio constituem ilícitos académicos, conforme estabelecido no Código de Ética da U.Porto.

Declaro, ainda, que utilizei ferramentas de Inteligência Artificial para a realização de parte(s) da presente dissertação, e essa utilização e os seus resultados estão devidamente assinalados e foram objeto de cuidada verificação para deteção de erros, imprecisões, atribuições infundadas ou outras formas de enviesamento de conteúdos e argumentos, bem como de determinação de autoria.

Luís Carlos Novais Alves

Contents

List of Figures

List of Tables

List of Acronyms

AI	Artificial Intelligence
API	Application Programming Interface
BFF	Backend For Frontend
CPU	Central Processing Unit
CSV	Comma-Separated Values
DDD	Domain-Driven Design
DevOps	Development and Operations
DTO	Data Transfer Object
ECS	Elastic Cloud Services
HPA	Horizontal Pod Autoscaler
HTTP	HyperText Transfer Protocol
IAM	Identity and Access Management
JPA	Java Persistence API
JSON	JavaScript Object Notation
JVM	Java Virtual Machine
K8s	Kubernetes
OIDC	OpenID Connect
OSS	Operations Support System
PaaS	Platform as a Service
PRISMA	Preferred Reporting Items for Systematic Reviews and Meta-Analyses
QoS	Quality of Service
RBAC	Role-Based Access Control
RPS	Requests Per Second
RAM	Random Access Memory
SaaS	Software as a Service
SCS	Synchro Cloud Services
SDG	Sustainable Development Goal
SPL	Software Product Line
SQL	Structured Query Language
SLA	Service Level Agreement

SLO Service Level Objective
TTK Trouble Ticket
TPS Transactions Per Second
UI User Interface
VM Virtual Machine
VU Virtual User
WSL Windows Subsystem for Linux

Chapter 1

Introduction

1.1 Context and Motivation

Cloud computing [1] has transformed the software industry, with Software as a Service (SaaS) becoming widely adopted due to its scalability, cost-effectiveness, and simplified delivery model [6]. In SaaS systems, multiple clients share the same application and infrastructure while maintaining logical isolation between tenants [6, 14].

Managing multi-tenant SaaS platforms introduces challenges in resource management, performance monitoring, and tenant-specific customization [6, 17, 18]. As the number of tenants and services increases, maintaining availability, efficiency, and compliance becomes more complex [6].

The NOSSIS One platform developed by Altice Labs [7] is a comprehensive Operations Support System (OSS) suite with a cloud-native, open, and modular architecture, covering the main operational processes across inventory, fulfilment, and assurance domains. As the platform evolves towards a multi-tenant SaaS model, the absence of a dedicated and unified tenant management layer makes onboarding, configuration, and lifecycle operations time-consuming and error-prone [3].

This motivates the development of an intelligent backoffice system to enhance automation, improve operational efficiency, and simplify administrative tasks within NOSSIS One [14]. Integrating intelligent management features into multi-tenant SaaS platforms can lead to more autonomous, efficient, and adaptive cloud operations [14].

1.2 Problem Identification

Despite the advantages of multi-tenant SaaS platforms, many organizations still rely on manual or semi-automated processes for managing tenants, monitoring resources, and ensuring service quality. These processes are time-consuming, error-prone, and difficult to scale as the number of tenants increases [6].

In the context of the NOSSIS One platform, current management workflows require significant human intervention to configure tenants, assign features, and monitor system status [3]. This limits the platform's efficiency and hinders the ability to respond quickly to operational issues [6].

The core problem addressed by this research is the absence of an intelligent backoffice system that can automate and optimize the management of multi-tenant cloud platforms [14]. Solving this problem is relevant for software providers and enterprise users, as it can reduce administrative workload, improve service reliability, and enable more consistent adherence to Service Level Agreements (SLAs). By integrating intelligent management and automation features, the proposed solution aims to enhance operational performance and scalability [6, 14], filling a gap in current SaaS backoffice capabilities.

1.3 Research Questions

Based on the problem identified in the management of multi-tenant SaaS platforms, the following research questions guide this thesis:

- **Q1:** How can the tenant lifecycle (onboarding, approval, configuration, and subscription management) be automated to reduce manual workload in a multi-tenant SaaS platform?
- **Q2:** Which architectural approaches best support scalable, secure, and modular tenant-management backoffice systems?
- **Q3:** How can resource optimization and autoscaling be designed for such a platform, and what benefits would they be expected to provide?

1.4 Objectives and Scope

The main objective of this thesis is to design and develop an intelligent backoffice for the control and management of multi-tenant cloud platforms, using NOSSIS One as the primary case study. The scope covers the design and implementation of the *tenant-manager* module within SIGO, the part of the NOSSIS One platform that encompasses the Trouble Ticketing, Maintenance, and OneCare modules. The objectives of this work are:

- Develop an algorithm to efficiently manage multi-tenant platforms.
- Implement functionality for the automatic instantiation of new tenants, including resource and permission configuration.
- Create interactive dashboards for platform monitoring and management.
- Integrate licensing mechanisms and per-tenant feature management.
- Ensure platform scalability by applying Development and Operations (DevOps) and cloud-native principles.

Chapter 2

Literature Review and State of the Art

This chapter presents a systematic literature review on multi-tenant SaaS automation and provisioning. It follows the Preferred Reporting Items for Systematic Reviews and Meta-Analyses (PRISMA) framework and analyzes the selected studies to describe the current state of the art in the domain.

2.1 Literature Review

This section describes how the relevant literature was identified and collected. The process was structured and reproducible, and it targeted scientific works on multi-tenant SaaS automation and provisioning.

2.1.1 Methodology

A systematic literature review is conducted following the PRISMA framework [13]. Figure 2.1 illustrates the overall process, including identification, screening, eligibility, and inclusion stages.

Paper Collection The paper collection process comprised three stages: selection of data sources, definition of search strings, and organization of the collected references.

Data Sources Three reputable databases were used: **IEEE Xplore**, **Scopus**, and **ACM Digital Library**. They were selected for their broad coverage in computer science and engineering, and all three index peer-reviewed journals and conferences in cloud computing and software engineering.

Search Strings Relevant keywords were identified from the main research themes: multi-tenancy, SaaS automation, resource management, cloud-native orchestration, and architecture migration. Boolean operators (`AND`, `OR`) were applied to refine the scope. The search strings were iteratively adjusted to achieve a manageable number of relevant papers. Only journal articles, conference papers, and books published from 2020 onwards were included. The final search strings are shown in Table 2.1, and the corresponding results per database in Table 2.2.

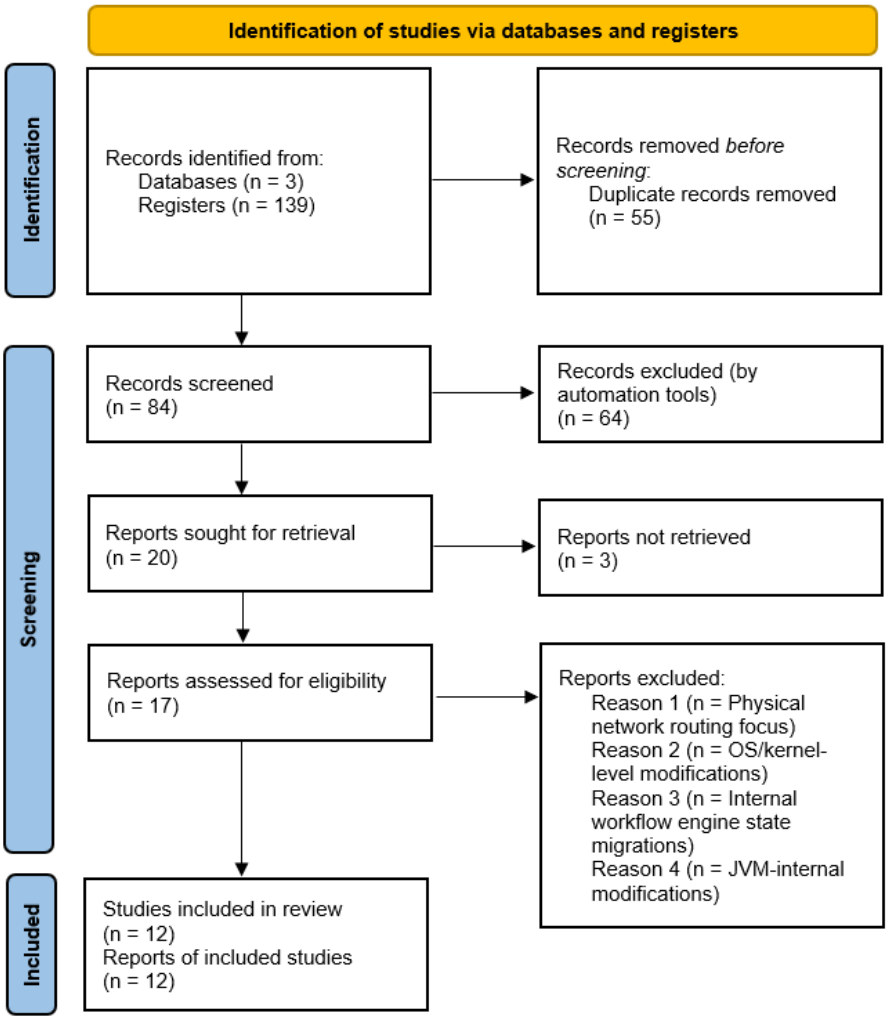


Figure 2.1: PRISMA Flow Diagram

Data Collection and Organization A total of 139 documents were retrieved across the three selected digital libraries using the defined search strings. All retrieved records were exported and imported into **Zotero** [5], an open-source reference management tool. Zotero was used to detect duplicates, manage bibliographic metadata, and organize papers across collections.

Screening A total of 55 duplicate records were removed, resulting in 84 unique papers.

From these 84 records, 64 were excluded after title and abstract screening, because they covered unrelated topics or did not match the focus on SaaS, multi-tenancy, cloud-native architectures, and automation. This left 20 papers.

The Zotero "Find Full Text" functionality was then used to retrieve the full documents automatically. Some files were added manually when they were not accessible. In the end, 3 papers without accessible full text were discarded, which left 17 documents for detailed assessment.

Table 2.1: Search strings

QS1	("multi-tenant" OR "multi-tenancy") AND "saas" AND ("automation" OR "automated" OR "orchestration")
QS2	("multi-tenant" OR "multi-tenancy") AND "saas" AND ("resource sharing" OR "resource management" OR "resource allocation")
QS3	("multi-tenant" OR "multi-tenancy") AND ("tenant provisioning" OR "tenant management" OR "automated provisioning" OR "subscription management" OR "tenant onboarding" OR "tenant lifecycle")
QS4	("multi-tenant" OR "multi-tenancy") AND "saas" AND ("kubernetes" OR "cloud-native")
QS5	("saas" OR "multi-tenancy") AND ("migration" OR "transformation") AND ("single-tenant" OR "multi-tenant")

Selection After the screening phase, the remaining 17 papers underwent an in-depth full-text assessment to refine the selection.

Based on this review, 5 papers were excluded for being out of the architectural context of the project or triggering specific exclusion criteria:

- Focus on physical network routing layers rather than application-level cloud-native orchestration;
- Modifying operating system or Java Virtual Machine (JVM) kernels to achieve isolation, which conflicts with the required containerized approach;
- Exclusive focus on internal workflow engine state migrations rather than cloud-native tenant provisioning.

The final set comprises 12 papers. These are the most relevant contributions to SaaS, multi-tenant, and cloud-native architecture.

Table 2.3 summarizes these studies, presenting for each the research purpose and corresponding approach/domain.

Table 2.3: Research purpose description for selected studies

References	Research purpose	Approach/domain
[2]	To report an industrial experience of migrating a legacy monolith to a multi-tenant microservice architecture, demonstrating improvements in scalability, management, and third-party integration.	Monolith-to-Microservices Migration; Domain-Driven Design (DDD); Database Isolation; Industrial Case Study.

Table 2.3 Research purpose description for selected studies (cont.)

References	Research purpose	Approach/domain
[4]	To suggest an approach for migrating exclusive monolithic enterprise applications to a Cloud-native SaaS model that supports tenant-specific deep customizations while preserving tenant isolation.	Monolith Migration; Multi-tenant Customization; Application Programming Interface (API) Gateways; Tenant Isolation; DDD.
[8]	To design a comprehensive cloud-native monitoring and alerting system based on Prometheus to manage distributed large-scale clusters, multi-tenant access control, and dynamic anomaly detection.	Cloud-Native Monitoring; Prometheus/Grafana; Multi-tenant Cluster Management; Dynamic Anomaly Detection.
[9]	To present a Kubernetes-based Platform as a Service (PaaS) featuring a custom priority-based allocation algorithm for dynamic tenant creation, quota updates, and real-time resource monitoring.	Kubernetes-based PaaS; Priority-based Resource Allocation; Namespaces and ResourceQuotas; Tenant Lifecycle Management.
[10]	To describe the design and operation of Elastic Cloud Services (ECS), Snowflake's control plane, focusing on automated software rollouts, autoscaling, throttling, and cross-AZ load balancing at scale.	Cloud Control Plane ECS; Autoscaling and Throttling; Multi-Cloud Orchestration; Availability Zone Balancing.
[11]	To propose an event-based, non-intrusive deep customization approach for migrating monoliths to microservices, enabling synchronous and asynchronous business logic customizations for tenants.	Event-Based Customization; Microservices Migration; Message Broker/Event Bus; Multi-tenant SaaS.
[12]	To propose a hybrid architecture integrating Software Product Lines (SPLs) with microservices to reduce development complexity and time-to-market while supporting customizability and cloud-native scalability.	Software Product Line SPL; Hybrid Architecture; Multi-tenant SaaS; DDD; Healthcare Domain.

Table 2.3 Research purpose description for selected studies (cont.)

References	Research purpose	Approach/domain
[15]	To outline a reference architecture and principles for both intrusive and non-intrusive deep customization of multi-tenant SaaS using microservices, balancing isolation, assimilation, and economies of scale.	Non-intrusive Customization; API Gateway Pattern; SaaS Architecture; Docker Isolation.
[16]	To evaluate the feasibility of Kubernetes container orchestration for achieving adaptive performance isolation, Quality of Service (QoS) differentiation, and admission control without experiencing Service Level Objective (SLO) instability.	Container Orchestration (Kubernetes); Adaptive Performance Isolation; QoS Differentiation; SLO Stability.
[19]	To construct a Cross-Tenant Role and Attribute-Based Access Control (CT-RABAC) model ensuring data isolation and secure cross-domain collaboration in distributed multi-tenant environments.	Access Control (CT-RABAC); Cross-tenant Collaboration; Semi-distributed Deployment; SaaS Security.
[20]	To propose VirtualCluster, a framework extending Kubernetes to provide dedicated control planes and shared data planes for tenants, improving multi-tenant isolation while maintaining API compatibility.	Kubernetes Multi-tenancy (VirtualCluster); Control/Data Plane Isolation; Resource Syncer; Cloud Container Services.
[21]	To propose a feature tree and dynamic QoS-based model that translates fuzzy tenant requirements into service integration rules, allowing flexible functional and non-functional SaaS customization.	Feature Tree Model; Dynamic QoS Customization; SaaS Service Integration; Resource Consumption Modeling.

The screening and selection stages progressively refined the corpus, keeping studies with sound methodology and direct relevance to the research objectives. The initial 139 records were reduced to 12 primary studies, each covering aspects of SaaS, multi-tenancy, and cloud-native system design.

Together these studies cover container orchestration frameworks, monolith-to-microservices migration, dynamic resource allocation and quota management, multi-tenant monitoring, and au-

Table 2.2: Search results by database and query

Data Source	QS1	QS2	QS3	QS4	QS5	Total
IEEE Xplore	7	4	9	2	11	33
Scopus	10	19	18	7	19	73
ACM DL	17	5	4	3	4	33
Total	34	28	31	12	34	139

tomated provisioning. They give a representative basis for the current research trends and open challenges in building and managing modern SaaS systems and intelligent backoffices.

The next section examines these studies in more depth to identify common patterns, research gaps, and emerging directions.

2.2 State of the Art

This section reviews the state of the art in multi-tenant SaaS across three areas: architectural approaches, automated provisioning and tenant management, and resource optimization and observability. It analyzes the limitations of current systems to identify the main gaps and set the technical basis for the proposed solution.
Note: All figures in this section are reproduced from the original publications and may contain text in languages other than English.

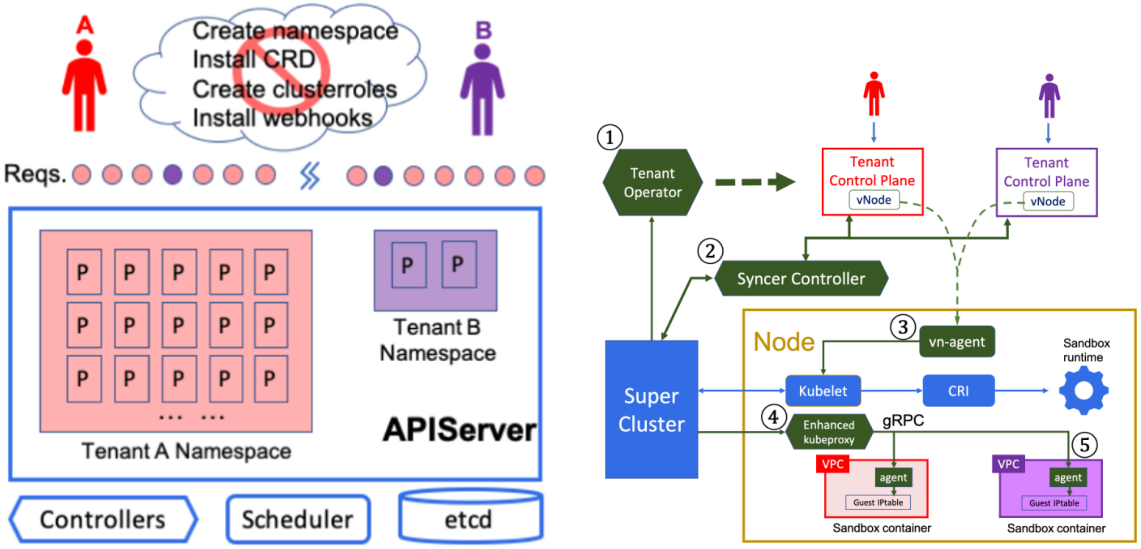
2.2.1 Architectural Approaches

Research in this area addresses the structural isolation and logical division of resources needed to serve many users on shared infrastructure. The main challenge is balancing the cost benefits of sharing resources against the security needs of isolation.

Native container orchestrators provide weak multi-tenant isolation. The *VirtualCluster* framework [20] answers this with control plane separation: each tenant gets a dedicated control plane, while the underlying compute nodes of a super cluster are shared. This gives strict isolation and keeps API compatibility. For an intelligent backoffice, it shows how to orchestrate isolated tenant environments on shared infrastructure without sacrificing resource utilization. The isolation impact and the overall structure are shown in Figure 2.2.

The left subfigure shows the isolation risks when all tenants share a single control plane: a malicious or faulty tenant can affect others by consuming control plane resources or triggering cascading failures. The right subfigure presents the VirtualCluster solution: each tenant has an isolated virtual control plane while sharing the underlying compute infrastructure, eliminating cross-tenant blast radius while maintaining efficient resource utilization.

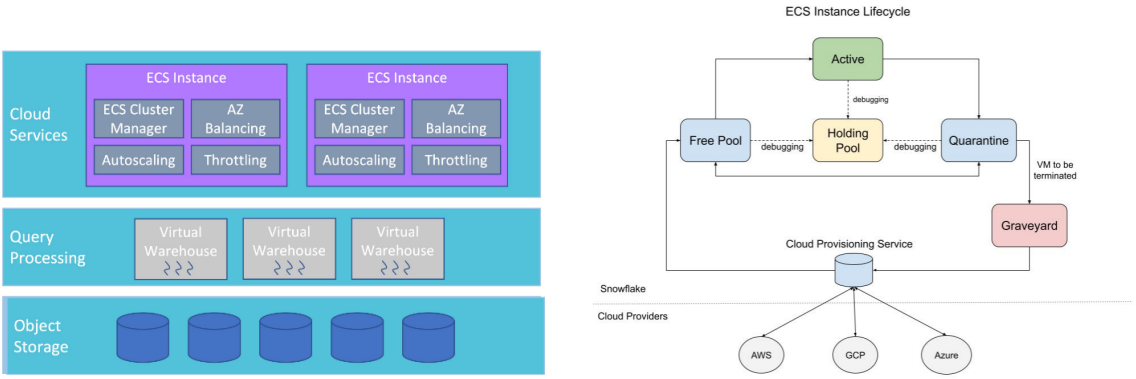
Snowflake's ECS layer [10] is an industrial example of a large-scale control plane. ECS manages virtual machine lifecycles, runs automated software rollouts, and keeps high availability through continuous zone balancing and health monitoring. Its main trait is the separation of the control plane from the data processing layer, which shows how a centralized backoffice can



(a) Impact of sharing a single control plane (b) VirtualCluster Architecture Overview

Figure 2.2: Visualization of the VirtualCluster Isolation Impact and Architecture [20].

manage the lifecycle, scaling, and recovery of distributed multi-tenant workloads on its own. The control plane architecture and lifecycle states are shown in Figure 2.3.



(a) Snowflake Architecture Presentation (b) ECS Cluster State Machine

Figure 2.3: Overview of the Snowflake ECS Control Plane and Lifecycle Management [10].

The left subfigure illustrates Snowflake's overall architecture with the ECS control plane managing a distributed fleet of virtual machines and data processing nodes across multiple availability zones. The right subfigure shows the lifecycle state machine for ECS clusters: provisioning, running, scaling, and decommissioning states with automated transitions driven by health checks and resource demands. This design separates management concerns from data processing, allowing the control plane to independently manage cluster lifecycles, rolling updates, and zone balancing.

Moving legacy systems to cloud-native environments needs a structured migration strategy. One such approach migrates monoliths into customizable microservices [4]. It uses a tenant manager together with an API gateway to separate the core product logic from tenant-specific customizations, so that individual changes do not affect the shared codebase. For a multi-tenant

backoffice, this gives a clear pattern for isolating tenant data and serving customizable services while still gaining the cloud's economies of scale. The target architecture is shown in Figure 2.4.

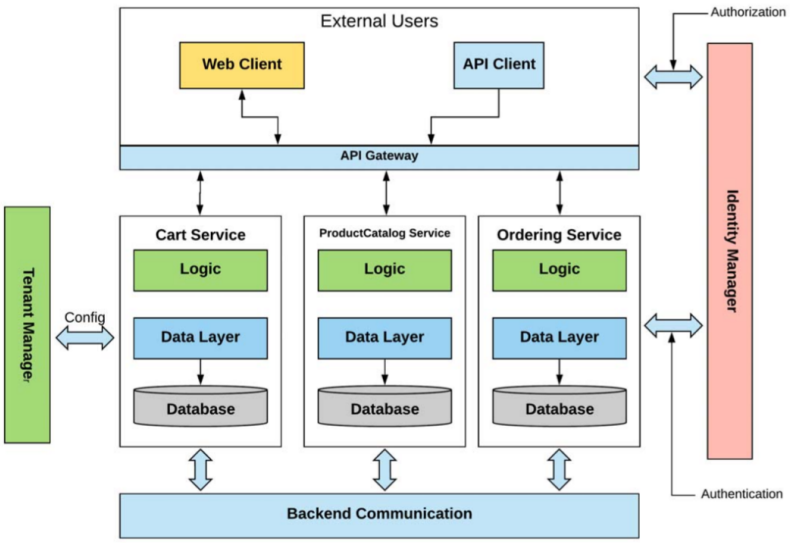
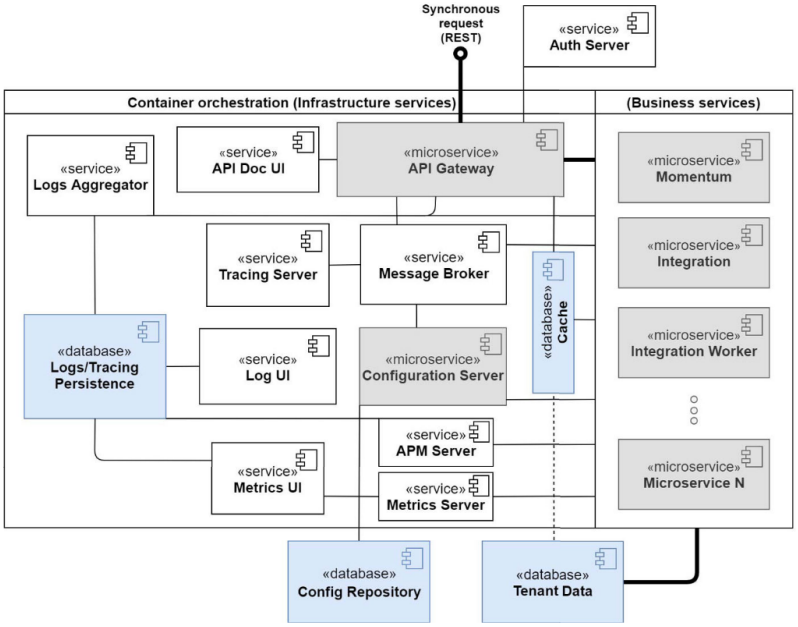


Figure 2.4: Target Architecture for Customizable Multi-Tenant SaaS [4].

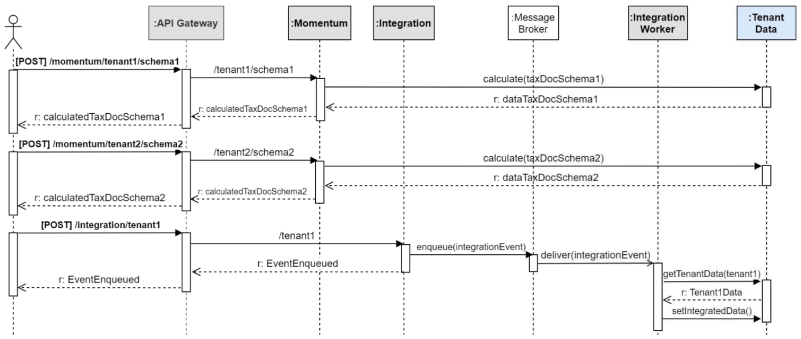
A multi-tenant microservice architecture improves scalability and maintenance but needs careful orchestration of shared resources. The Synchro Cloud Services (SCS) architecture [2], drawn from industry, combines an API Gateway, an authentication server, and a message broker to manage tenant requests. It isolates each tenant in its own database and uses an asynchronous worker pattern for heavy workloads, so that one tenant's load does not block the rest of the system. This pattern fits platforms that already rely on event streaming. The structural layout and request lifecycle are shown in Figure 2.5.

The first subfigure shows the core components of the SCS architecture: an API Gateway as the single entry point, an authentication service for tenant isolation, a message broker for asynchronous communication, and isolated tenant databases. This separation ensures that tenant requests are independently routed, authenticated, and processed without cross-tenant interference. The second subfigure presents a sequence diagram of a typical request lifecycle: a tenant's request flows through the gateway, authentication, and either to the main service or asynchronously to worker pools via the message broker. This pattern prevents one tenant's heavy workload from blocking other tenants' requests.

Microservice flexibility can conflict with the need to keep development complexity and time-to-market low. A hybrid architecture [12] addresses this by combining backend APIs for the core functionality with dedicated microservices only for tenant-specific variabilities. Its gateway layer is built as per-interface Backend For Frontends (BFFs), which route traffic from the different user interfaces to the right backend services and microservices. A separate asynchronous reporting microservice aggregates data without blocking the core operations. The abstract design is shown in Figure 2.6.



(a) SCS Microservice Architecture



(b) SCS Sequence Diagram

Figure 2.5: SCS Architecture and Interactions [2].

This figure depicts a hybrid approach combining a central backend API with per-tenant variabilities handled by dedicated microservices. The architecture uses per-interface Backend-for-Frontend (BFF) components that intelligently route requests from multiple user interfaces (web, mobile, third-party) to either the shared core services or tenant-specific microservices. An asynchronous reporting microservice collects data independently, avoiding blocking the main application flow. This design balances economies of scale through shared core functionality with customization flexibility through isolated microservices.

2.2.2 Automated Provisioning and Tenant Management

Beyond structural isolation, multi-tenant environments are dynamic and need mechanisms to onboard new tenants, assign resources, and enforce SLAs.

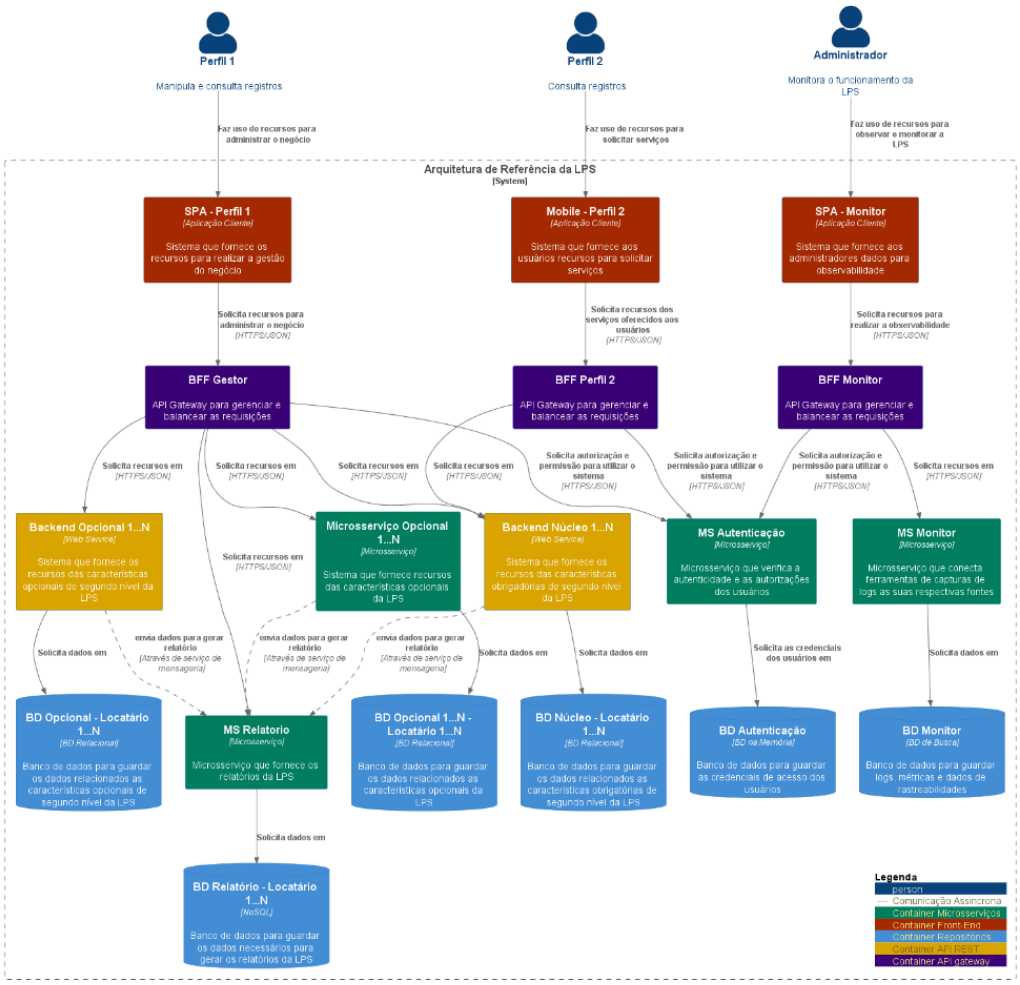


Figure 2.6: Abstract Architecture [12].

Automated tenant onboarding and lifecycle management can be handled by a custom PaaS layer built directly on the container orchestrator [9]. It uses a priority-based scheduling algorithm that allocates resources to new tenants by high, medium, and low priority, and holds lower-priority requests in a pending state when the cluster runs short of resources. This matches the goals of an intelligent backoffice: it shows how native cloud primitives can be automated for self-service provisioning and fair resource distribution. The proposed backend and frontend components are shown in Figure 2.7.

The figure shows the proposed PaaS system with two layers: a backend management service that implements the priority-based allocation algorithm, and a frontend for tenant self-service provisioning. The system manages Kubernetes namespaces with resource quotas for each tenant, implements scheduling logic to allocate high, medium, and low priority requests, and queues lower-priority requests when resources are scarce. A monitoring component tracks real-time resource availability, allowing the system to make dynamic allocation decisions and inform tenants of their provision status.

A further challenge is allowing tenant-specific customizations without changing the shared

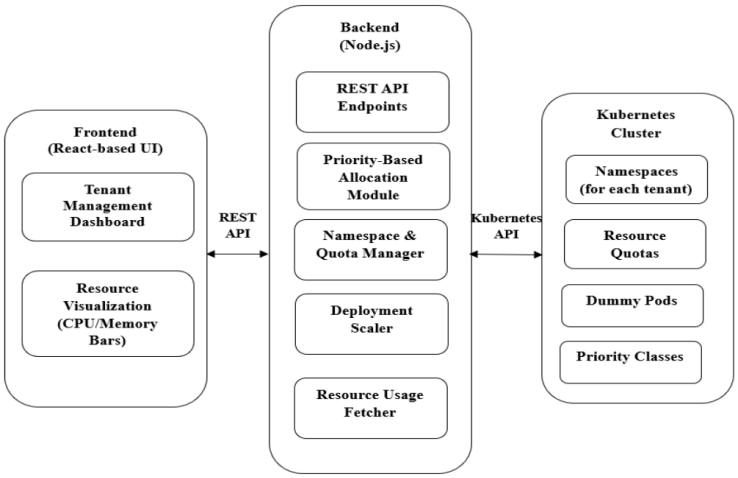
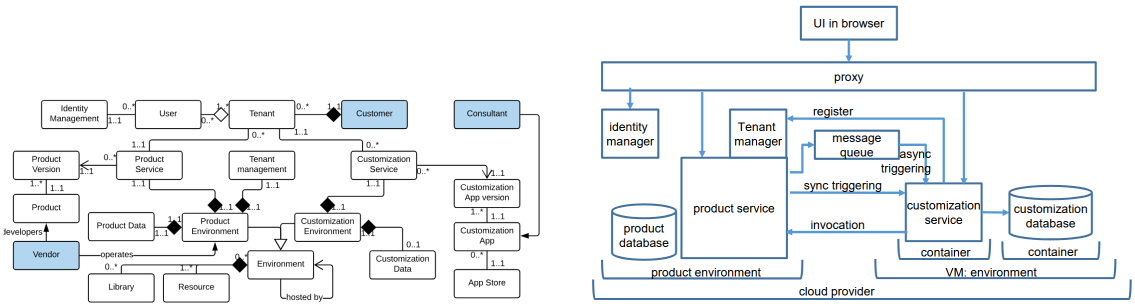


Figure 2.7: Proposed System Architecture for Container-based PaaS [9].

core codebase. A non-intrusive approach [15] handles this through an API Gateway that gives isolated, tenant-specific microservices authorized access to the main product's business logic. Custom code then cannot harm the security or performance of the main application, which lets a backoffice support deep tenant customization while keeping strict isolation. The conceptual model and reference architecture are shown in Figure 2.8.



(a) Conceptual Customization Model

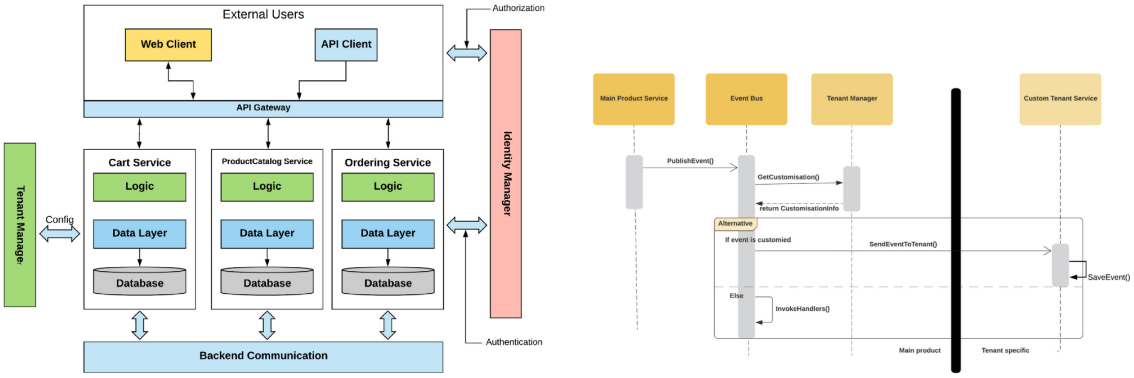
(b) Reference Architecture

Figure 2.8: Models for Multi-tenant SaaS Customization [15].

The left subfigure presents the conceptual customization model: tenants are isolated in dedicated microservices that connect to the main application through an API Gateway with strict authorization rules. The right subfigure shows the reference architecture implementing this model with tenant-specific containers, each exposing a custom interface, while the gateway enforces access policies and isolation boundaries. This approach allows deep customization of tenant business logic without exposing the core application to custom code, preventing security breaches or performance degradation from affecting other tenants.

Deep customization can also be done asynchronously with an event-driven architecture [11]. A message broker lets the main system publish events that trigger a tenant's custom microservices, which cuts direct API calls and avoids bottlenecks at the gateway. The tenant microservices intercept these events, run their custom logic, and return the flow to the main application. This scales

well on event-driven cloud platforms. The target architecture and event-based flow are shown in Figure 2.9.



(a) Target Architecture (b) Event-based Customization Flow

Figure 2.9: Target Architecture and Flow for Event-Based Customization [11].

The left subfigure illustrates the target event-driven architecture: the core application publishes domain events to a message broker, and tenant-specific microservices subscribe to these events, execute custom logic asynchronously, and optionally return results. The right subfigure shows a concrete customization flow where a business operation in the core system triggers events that flow through the message broker to tenant microservices, which intercept and customize the behavior before returning control. This asynchronous pattern eliminates tight coupling and gateway bottlenecks, allowing each tenant's customizations to scale independently.

A multi-tenant backoffice needs more than structural isolation. It also needs fine-grained, dynamic control over who can access tenant data, especially when cross-tenant collaboration is allowed. The Cross-Tenant Role and Attribute-Based Access Control (CT-RABAC) model [19] addresses this by combining role-based and attribute-based access control within each tenant, where attribute constraints refine coarse roles to give multi-level, fine-grained permissions. Collaboration goes through a central access control center that defines cross-tenant roles, to which each tenant publishes only the permissions it chooses to expose. A semi-distributed deployment keeps intra-tenant access decisions local to each tenant and reserves the central center for cross-tenant authorization, which lowers load and preserves each tenant's internal access policy. For an intelligent backoffice, this shows how one set of credentials can span multiple tenants while still guaranteeing data isolation and auditable permission control. The semi-distributed deployment mechanism is shown in Figure 2.10.

2.2.3 Resource Optimization and Observability

On shared infrastructure, high availability and performance depend on continuous monitoring and adaptive resource allocation, which keep aggressive tenants from degrading the experience of others.

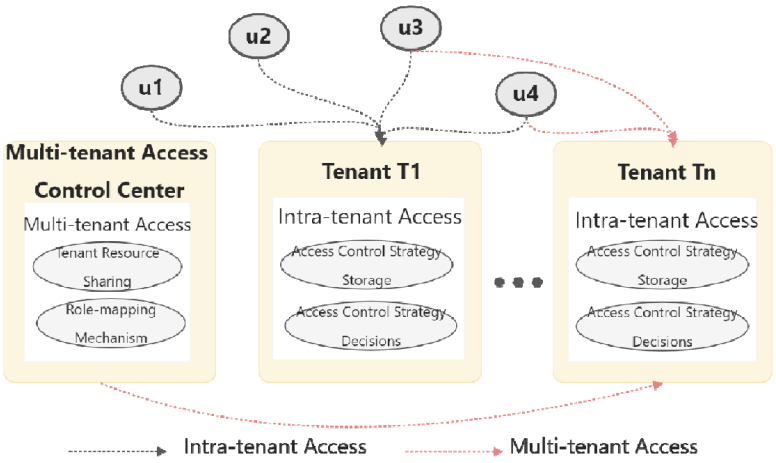


Figure 2.10: Semi-distributed deployment mechanism of the CT-RABAC model [19].

Adaptive performance isolation under strict SLOs needs precise resource management, which maps onto container orchestration concepts [16]. Tenants are grouped into separate containers by SLA class, and cloud-native horizontal scaling and resource limits take over part of the work, so the request scheduler can be reduced to admission control while the orchestrator handles QoS differentiation and adaptive resource allocation. The study confirms that container orchestration can enforce QoS differentiation and stop aggressive tenants from consuming shared cluster resources. It also cautions that SLO stability breaks under linear scaling when the number of subscribed tenants changes, because resource demand scales non-linearly. For an intelligent backoffice, this shows both the strengths and the calibration limits of using native orchestration primitives for performance isolation. The impact of this architecture on multi-tenant SaaS is illustrated in Figure 2.11.

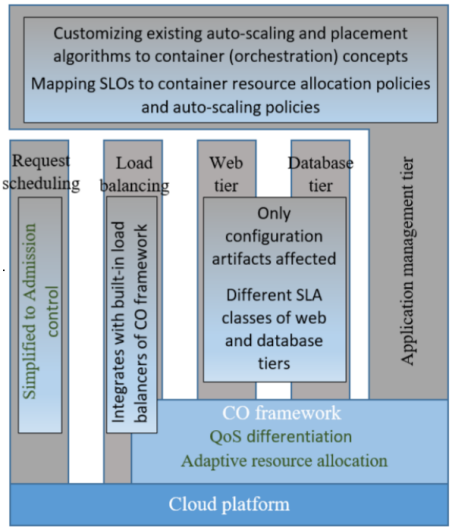
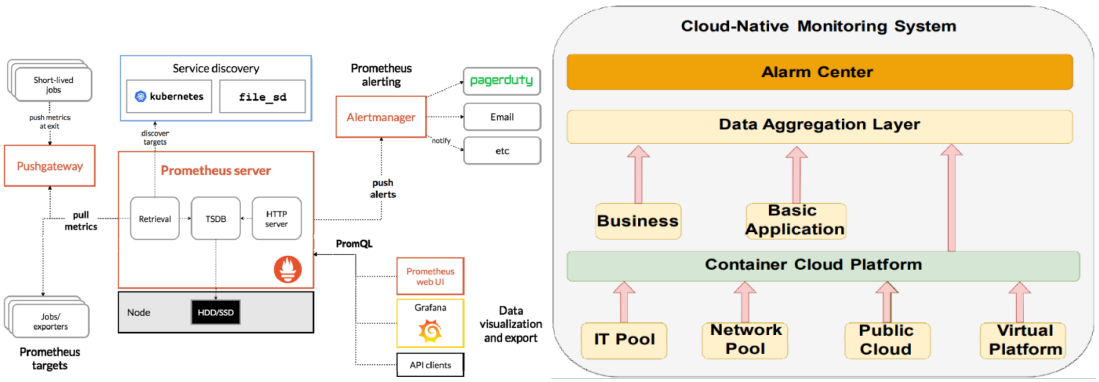


Figure 2.11: Impact on existing multi-tenant SaaS architecture when using a container orchestration framework [16].

Stability in dynamic microservice architectures needs solid observability. A cloud-native monitoring system built on the Prometheus ecosystem [8] aggregates metrics, logs, and invocation chains for dynamic anomaly detection and alert convergence, and uses role-based access control to isolate tenant data. This gives the real-time monitoring and early fault detection that an intelligent backoffice needs for continuous availability and performance. The Prometheus integration and the overall monitoring system are shown in Figure 2.12.



(a) Prometheus Architecture (b) Cloud-Native Monitoring System

Figure 2.12: Overview of the Cloud-Native Monitoring and Prometheus Architecture [8].

Tailoring service quality to individual tenants on a shared resource pool is a further optimization concern, addressed by a feature tree and dynamic QoS model [21]. An ontology-based feature tree breaks ambiguous tenant requirements into service-integration rules and maps each functional and non-functional feature to concrete components. A dynamic QoS model then describes each component's quality under different load patterns and links per-tenant QoS requirements to a resource-consumption model that drives component placement across virtual machines. For an intelligent backoffice, this turns individual tenant functional and quality demands into cost-aware, resource-efficient provisioning decisions, and it complements the runtime monitoring and scaling described above. The extended feature model is shown in Figure 2.13.

2.2.4 Conclusion

The reviewed work shows a clear shift in multi-tenant SaaS, from monolithic legacy infrastructure to fine-grained, customizable, cloud-native microservices [2, 4].

- **Architectural foundations:** The field has moved from traditional Virtual Machines (VMs) to containerization and container orchestration frameworks. These are widely adopted for their isolation and cost-efficiency: they use lightweight namespaces and dedicated control planes instead of relying only on physical hardware separation [16, 20].
- **Automated provisioning and customization:** Tenant management has moved from manual updates to automated, policy-driven lifecycles [9]. To serve many customers on a shared

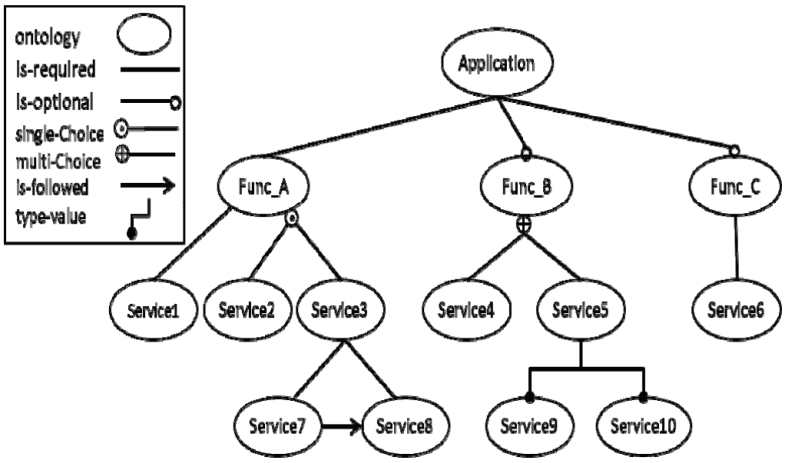


Figure 2.13: Extended multi-tenant SaaS application feature model [21].

codebase, current architectures use API gateways and event-driven patterns for deep, non-intrusive tenant customizations that need no change to the core product and no downtime [11, 15]. Secure tenant management also depends on fine-grained, role- and attribute-based access control that keeps data isolated while allowing controlled cross-tenant collaboration [19].

- **Resource optimization and observability:** Resource management is shifting toward dynamic, auto-scaling control planes and stronger observability [10]. Recent approaches combine native metric monitoring with scheduling algorithms to adjust quotas, enforce SLOs, and detect anomalies, which improves resource allocation and limits interference between tenants [8, 16]. Other models map per-tenant functional and QoS requirements to resource-aware placement, so service quality can differ per tenant on shared infrastructure [21].

In short, modern SaaS platforms favor modular architecture, strict but efficient isolation, and data-driven observability. This thesis builds on these findings and on the requirements of the NOSSIS One platform to develop an intelligent backoffice that automates multi-tenant instantiation, centralizes feature and license management, and keeps continuous reliability through cloud-native orchestration and real-time monitoring.

Chapter 3

Proposed Solution

This chapter presents the proposed solution for managing tenants in a multi-tenant cloud platform. The focus is the *tenant-manager* subsystem developed within SIGO, part of the NOSSIS One platform at Altice Labs, which automates the commercial and operational lifecycle of tenants: onboarding, approval, configuration, product subscriptions, usage visibility, and identity integration.

The chapter is organized as follows. Section 3.1 defines the functional and non-functional requirements. Section 3.2 describes the system architecture, technology stack, and authentication model. Section 3.3 presents the domain model and persistence schema. Section 3.4 details the tenant lifecycle and its state transitions. Section 3.5 describes the integration with the NOSSIS Identity and Access Management (IAM) service for identity provisioning and tenant ownership enforcement. Section 3.6 describes the main system flows. Section 3.7 covers the API design. Section 3.8 presents the frontend interface. Section 3.9 summarises the chapter.

3.1 System Requirements

The system supports three main roles: public users, platform administrators, and tenant-scoped users. Public users can request tenant creation and consult the product catalog. Administrators manage the full tenant lifecycle, inspect tenant details, and operate subscriptions and settings on behalf of any tenant. Tenant-scoped users access only the information associated with their own tenant.

The functional requirements are organized by role.

Public

- Submit a tenant creation request with a name and an email address.
- Consult the global product catalog, including available plans for each product.

Administrator

- List tenants with optional filters by status, name, or email, with pagination support.

- Consult aggregate tenant counts by status.
- Retrieve the full details of a tenant, including its active subscriptions.
- Approve or reject a pending tenant request.
- Disable and enable tenants, controlling their access to the platform.
- Soft-delete a tenant while preserving the record for auditability.
- Manage settings and subscriptions on behalf of any tenant.

Tenant-Scoped

- Retrieve and update the authenticated tenant's own information and settings.
- Subscribe to products, change the active subscription plan, and cancel active subscriptions.
- Consult aggregated subscription usage for active subscriptions.
- Manage tenant users: list, create, update, disable, enable, and delete users within the plan limits.

The non-functional requirements are:

- Expose a REST API with JavaScript Object Notation (JSON) payloads and an OpenAPI contract.
- Persist data in a relational database.
- Support OpenID Connect (OIDC)-based authentication and integrate with NOSSIS IAM for authorization and identity provisioning.
- Enforce tenant isolation through the authenticated tenant claim, restricting each user to their own tenant's data.
- Follow a cloud-native, modular architecture to ensure scalability and portability.

3.2 Architecture

The *tenant-manager* is a module within SIGO, the part of the NOSSIS One platform at Altice Labs that groups its operational products. The module is deployed as an independent Quarkus service with its own dedicated PostgreSQL database, alongside the other SIGO modules. It consumes shared infrastructure components, NOSSIS IAM and the product engines, while exposing its own REST API to the portal frontend and to other platform services.

The target architecture for SIGO, including the tenant-manager and its full integration with all platform modules, is illustrated in Figure 3.1. The scope of this work is limited to the tenant-manager module. Integration with other platform modules, including the Email Service and the

Trouble Ticket (TTK) Engine, is part of the intended target architecture and is treated as future work in Chapter 7. The architecture as implemented, without those integrations, is shown in Figure 3.2.

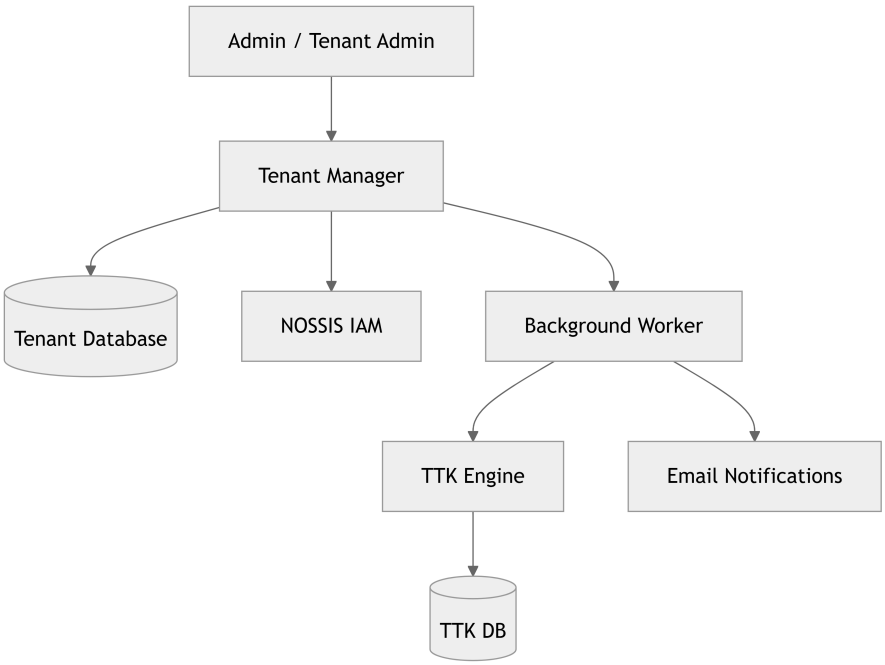


Figure 3.1: Target architecture of SIGO with tenant-manager

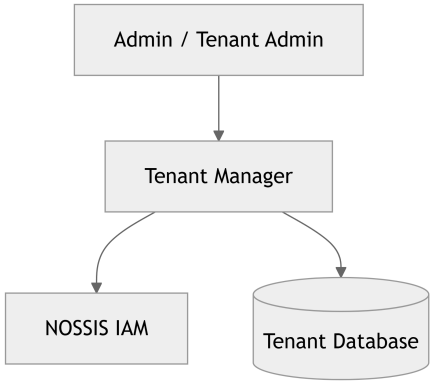


Figure 3.2: Implemented system context of the tenant-manager within SIGO

The backend follows a strict layered architecture, with each layer having a single well-defined responsibility. The layers, from the outermost to the innermost, are:

- **API layer**: REST resources that map HyperText Transfer Protocol (HTTP) verbs and paths to service operations, with no business logic.
- **Service layer**: Contains all business rules, lifecycle validations, and orchestration of operations across repositories and external integrations.

- **Mapper layer**: Converts between HTTP Data Transfer Objects (DTOs), domain models, and Java Persistence API (JPA) entities, preventing persistence details from leaking into the service layer and API contracts.
- **Repository layer**: Data access objects responsible for all database interactions.
- **Entity layer**: JPA entities representing the database tables.

The typical request flow is:

1. The REST resource in the API layer receives the HTTP request.
2. The DTO is converted into a domain model by the mapper.
3. The service layer validates business rules and allowed state transitions.
4. The repository layer loads or persists the corresponding JPA entities in PostgreSQL.
5. The response is mapped back into a JSON DTO and returned.

The layered architecture of the module is illustrated in Figure 3.3.

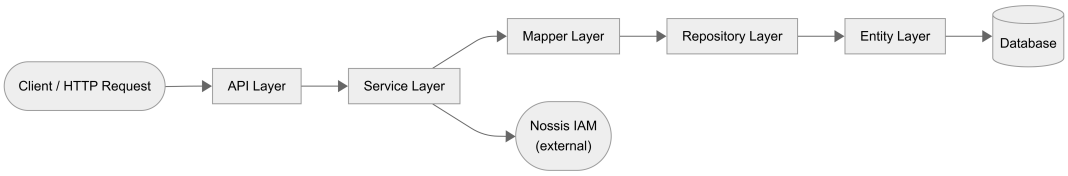


Figure 3.3: Layered architecture of the tenant-manager backend

The technology stack is the same one used across SIGO, which keeps the module consistent with the rest of the system. The backend is implemented in Java 21 with Quarkus, using Hibernate ORM Panache for persistence, PostgreSQL as the relational database, and Flyway for schema migrations. The frontend is implemented in Angular, composed into the SIGO portal shell.

3.2.1 Authentication and Authorization

Authentication is based on OIDC. Protected endpoints require a Bearer token issued by the NOSSIS IAM service, which acts as the OIDC provider. The token is validated by Quarkus OIDC on every request using the issuer's public keys.

To avoid ambiguity, the word *role* is used for four distinct concepts, kept separate throughout this work:

- **Access profile**: a high-level user category introduced in Section 3.1: public, administrator, or tenant-scoped.
- **IAM role**: a concrete authorization grant managed by the NOSSIS IAM service and carried in the access token, such as SIGO :: TENANT MANAGER :: Admin.

- **User role**: an attribute of an individual tenant user, with value ADMIN or MEMBER, that determines which tenant-manager IAM roles the user receives.
- **Product role**: a per-product authorization grant declared by a product (for example, an Admin or Operator role for the Trouble Ticketing product) and mapped to an IAM role that governs the user's permissions inside that product. Product roles are gated by the tenant's active subscriptions and are described in Section 3.3.2.

An access profile is realised by one or more IAM roles, and the user role attribute decides whether a tenant user acts as a tenant administrator or as a plain member.

Authorization to the tenant-manager itself is driven by the IAM roles carried in the token. Three roles govern access to the module:

- **SIGO :: TENANT MANAGER :: Admin**: assigned manually to system administrators, and grants full `XCRUD` permissions on all tenant-manager resources.
- **SIGO :: TENANT MANAGER :: Tenant**: assigned to the tenant admin account at approval time and to tenant users with the ADMIN role at creation time, and grants read-only access to the `sigo.tenant-manager.tenants` resource and full `XCRUD` on settings, subscriptions, and users.
- **SIGO :: TENANT MANAGER :: Oper**: assigned to tenant users with the MEMBER role at creation time, granting no access to any tenant-manager management endpoint. These users are consumers of the subscribed products, not of the tenant-manager.

These three roles govern access to the tenant-manager module and are distinct from product roles, which grant permissions inside the subscribed products rather than in the tenant-manager itself.

The identity provisioning operations performed through the NOSSIS IAM service are described separately in Section 3.5.

The full authentication and authorization flow, including OIDC token validation and IAM resource authorization, is illustrated in Figure 3.4.

3.3 Domain Model

3.3.1 Entity Relationships

The domain model is centered on the tenant entity. A tenant represents a customer organization and is identified by its name. It stores the tenant email, lifecycle status, and audit timestamps. Each tenant has exactly one settings record containing configuration such as the preferred language and logo URL.

Products and plans are global platform concepts. A product represents a commercial service offered on the platform and additionally declares the set of product roles it exposes, stored as a

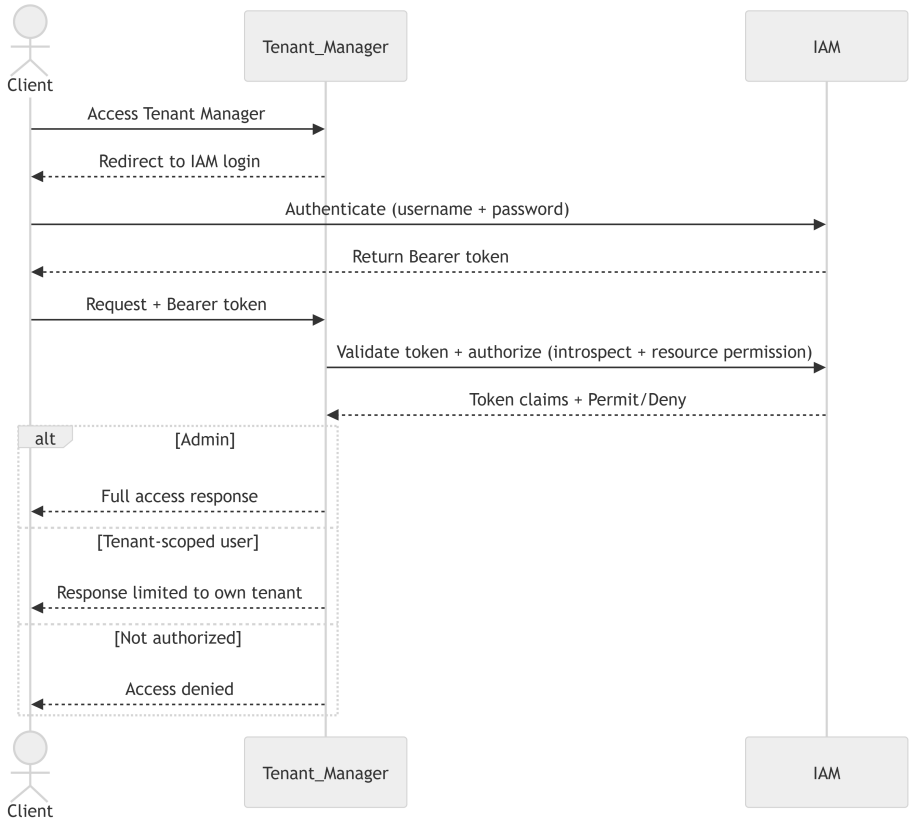


Figure 3.4: Authentication, authorization, and IAM provisioning flow

JSONB list on the product record (detailed in Section 3.3.2). A plan defines the pricing, billing period, and feature limits for a product. Feature limits are stored as a JSON object and include mandatory constraints such as `maxUsers` and `maxApiCalls`, which are enforced at subscription and user creation time. The `maxUsers` limit is applied per product, capping the number of distinct users that may hold roles for that product.

Tenant subscriptions link tenants to product plans. A subscription records the selected product and plan, its lifecycle status, and the billing period boundaries (periodStart, periodEnd). Subscription history is preserved through status transitions (ACTIVE, CANCELLED, COMPLETED) rather than physical deletion.

Usage information is stored at two granularities. `SubscriptionUsage` holds an aggregated snapshot of total users and API calls for an active subscription period. `SubscriptionUsageDaily` stores per-day feature usage data, populated by the daily aggregation cron job. The two granularities let the platform report current consumption and still retain historical usage across subscription periods and plan changes.

The main entity relationships are:

- one tenant has one tenant settings record;
- one tenant can have multiple subscription records across different periods;

- one tenant manages multiple IAM users, stored externally in NOSSIS IAM and scoped to that tenant;
- one product can have multiple plans and declares one or more product roles;
- one active subscription generates one aggregated usage snapshot and multiple daily usage records.

The database schema is managed through Flyway versioned migrations, with one table created per migration file. The migrations create the schema and its seven main tables: tenant, tenant_setting, product, plan, tenant_subscription, subscription_usage, and subscription_usage_daily. The product roles offered by each product are not a separate table. They are stored inline as a JSONB `roles` column on the product table.

The entity-relationship diagram, including the external IAM_USERS entity managed by the NOSSIS IAM service, is shown in Figure 3.5.

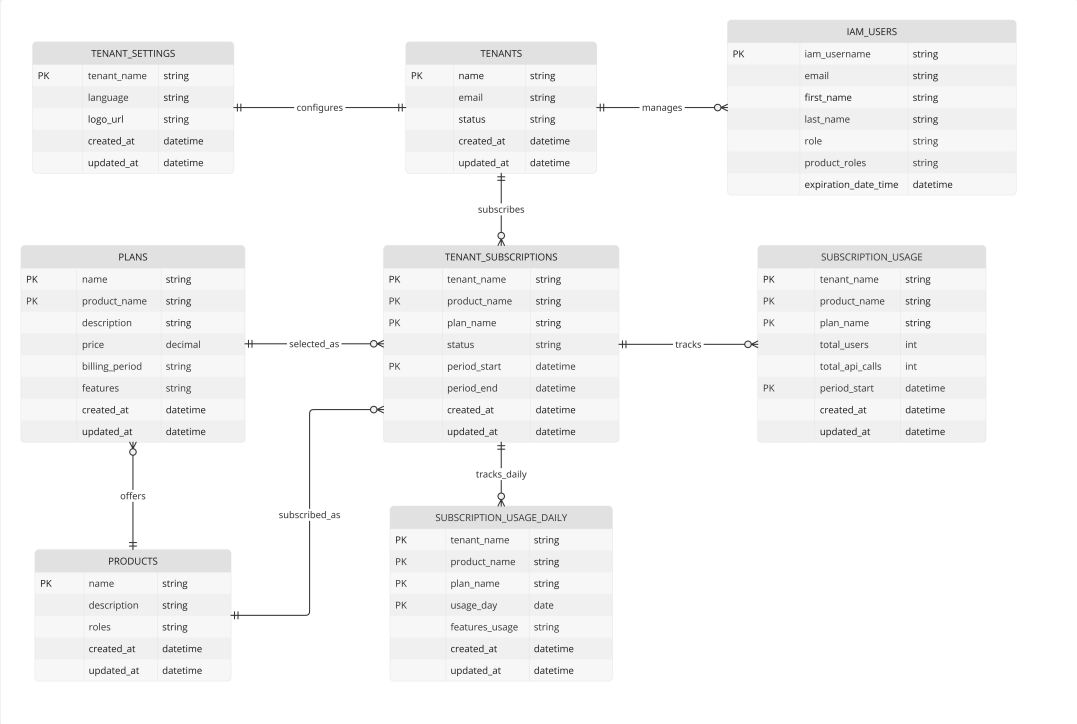


Figure 3.5: Tenant-manager entity-relationship and persistence schema

Listing 3.1 shows the tenant table definition. The status column uses a `CHECK` constraint to enforce valid state values at the database level, providing a secondary guard in addition to the service-layer validation. The unique index on the normalised name backs the case-insensitive duplicate-name check performed at tenant creation.

```
1 CREATE TABLE tenant (
2     name        TEXT PRIMARY KEY,
```

```
    email       TEXT NOT NULL UNIQUE,
    status      TEXT NOT NULL CHECK (
        status IN (
            'PENDING', 'PROCESSING', 'ACTIVE',
            'DISABLING', 'DISABLED',
            'DELETING', 'DELETED',
            'FAILED', 'REJECTED'
        )
    ),
    created_at TIMESTAMPTZ NOT NULL DEFAULT NOW(),
    updated_at TIMESTAMPTZ NOT NULL DEFAULT NOW()
);

CREATE UNIQUE INDEX ux_tenant_lower_name ON tenant (LOWER(name));
```

Listing 3.1: Tenant table migration

3.3.2 Product Roles

Beyond the three IAM roles that govern access to the tenant-manager itself (Section 3.2.1), each product declares its own set of product roles that determine what a user can do inside that product once the tenant has subscribed to it. Each product role has two attributes, each aimed at a different audience:

- `displayName`: a tenant-facing label exposed through the API (for example, Admin or Operator). Consumers select product roles by this name.
- `qualifiedRole`: the fully qualified IAM role name actually granted in the NOSSIS IAM service (for example, SIGO :: TTK :: Admin).

Two vocabularies therefore cross the API boundary. Externally, a product-role assignment is expressed as a reference pair `{productName, role}`, where `role` is the display name. This is the form carried in user creation, update, and read payloads, and the product catalog exposes display names only. Internally, the service layer resolves each reference to its qualified IAM role name before provisioning, and reverses the mapping when reading users back from IAM. This indirection keeps the public contract stable and human-readable while the concrete IAM role names remain an implementation detail.

Product roles are governed by two rules enforced in the service layer:

- A product role can only be assigned to a user if the tenant holds an active subscription to that product. Requests referencing a product without an active subscription are rejected before any IAM call is made.

- The per-product `maxUsers` limit, taken from the subscribed plan's feature limits, caps how many distinct users may hold roles for that product. A user's first role for a product counts against this limit and increments the product's usage counter. Removing a user's last role for a product decrements it.

The product administrator role, identified as the product role whose qualified name ends in Admin, is assigned automatically: when a tenant subscribes to a product, every existing tenant ADMIN user receives that product's admin role, and the assignment is reversed when the subscription is cancelled. This behaviour is described with the subscription flow in Section 3.6.2.

3.4 Tenant Lifecycle

Managing the tenant lifecycle is a central function of the module. The lifecycle is modelled as an explicit state machine with guarded transitions. Only transitions permitted by the business rules are allowed, and any attempt to perform an invalid transition returns a domain-oriented error response.

A new tenant request starts in the PENDING state. An administrator reviews the request and either approves or rejects it. On approval, the system transitions the tenant to the intermediate PROCESSING state, creates default settings (language EN, empty logo URL), provisions a NOSSIS IAM admin user, and transitions the tenant to ACTIVE. The approval response includes the provisioned IAM username and a temporary password to allow the tenant to log in immediately. On rejection, the tenant record is physically deleted, as no data needs to be retained for a request that was not accepted.

From the ACTIVE state, an administrator can disable the tenant. The current implementation transitions the tenant synchronously through the intermediate DISABLING state and settles at DISABLED, disabling the tenant's IAM admin account in the same request. A disabled tenant can be re-enabled, which restores ACTIVE status and re-enables the IAM account. A tenant can also be soft-deleted, which is allowed from either the ACTIVE or the DISABLED state. The soft-delete operation transitions the tenant synchronously through DELETING and settles at DELETED, disabling every IAM account scoped to the tenant while preserving all records for auditability.

The DISABLING, DELETING, FAILED, and REJECTED states also support the intended asynchronous provisioning architecture, described further in Section 3.6.1, in which lifecycle operations would be completed by background workers rather than inline in the request.

The tenant lifecycle state machine is shown in Figure 3.6.

The soft-delete flow is shown in Figure 3.7. Unlike disable, which acts only on the tenant admin account, soft-delete lists every user scoped to the tenant in the IAM service and disables each one before settling at DELETED, while preserving all database records.

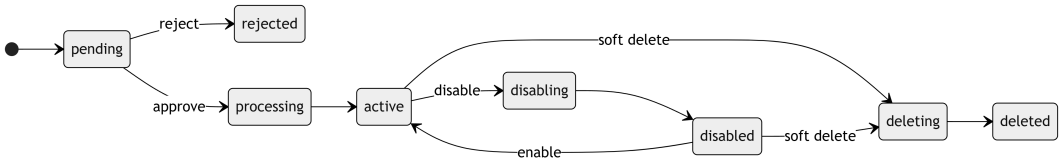


Figure 3.6: Tenant lifecycle state machine

3.5 IAM Integration

The tenant-manager integrates with the NOSSIS IAM service for two purposes: OIDC-based authentication and authorization (Section 3.2.1), and identity provisioning, described here. All tenant user accounts live in IAM rather than in the local database, so the module never stores credentials.

When an administrator approves a pending tenant, the service provisions a dedicated IAM admin account for that tenant with a temporary password and the tenant role, and returns the username and password in the approval response so that the administrator can hand them to the tenant. Disabling the tenant disables this account and re-enabling it restores access. The account is platform-managed and cannot be changed through the user-management endpoints.

Tenant users are created in IAM with a temporary password and a tenant-manager role determined by their ADMIN or MEMBER attribute: ADMIN users can manage their own tenant, while MEMBER users are plain product consumers. Every account is scoped to its tenant, and users can be listed, updated, disabled, re-enabled, and deleted within the plan limits.

A user may additionally hold product roles, which grant permissions inside the products the tenant has subscribed to. Each requested product role is validated against an active subscription and the per-product `maxUsers` limit before being granted, and the per-product user count is updated as roles are added or removed (Section 3.3.2).

Tenant isolation is enforced by `TenantOwnershipService` on every protected endpoint scoped to a tenant. The service derives the caller's tenant from its authenticated IAM identity and rejects the request with `403 SIGO_FORBIDDEN` when it does not match the tenant addressed in the path. System and super-admin accounts, which are not tenant-scoped, bypass the check, as do anonymous requests on public endpoints.

3.6 Main Flows

The sequence diagrams presented in this section represent the intended target architecture for each flow. The current implementation covers the core synchronous paths. Asynchronous provisioning through background tasks and integration with other platform engines such as the TTK Engine are part of the planned evolution and are treated as future work.

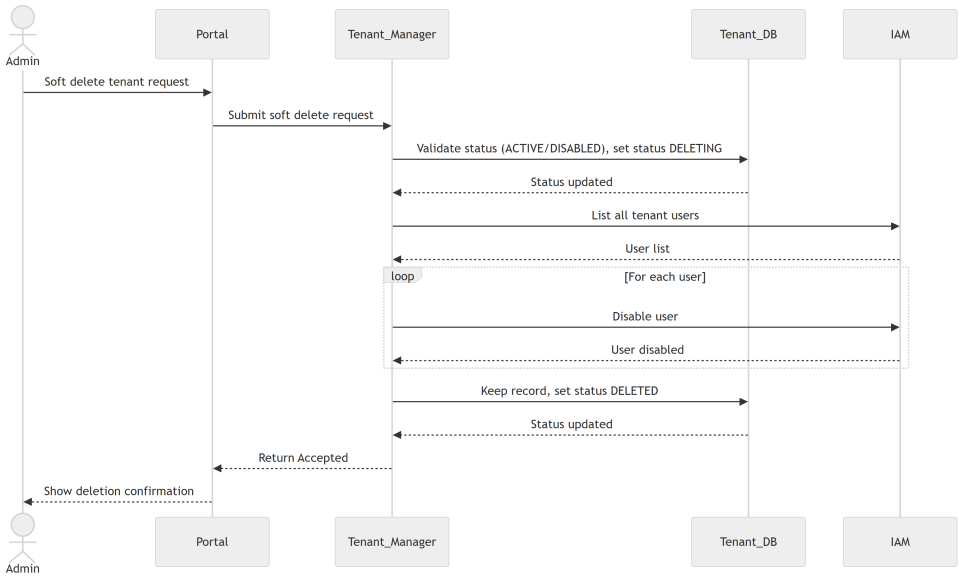


Figure 3.7: Tenant soft-delete flow

3.6.1 Tenant Onboarding and Approval

The tenant onboarding flow begins when a public user submits a creation request through the portal. The tenant-manager stores the record with status PENDING and returns the created tenant. An administrator then reviews the request through the admin dashboard.

The request flow is shown in Figure 3.8.

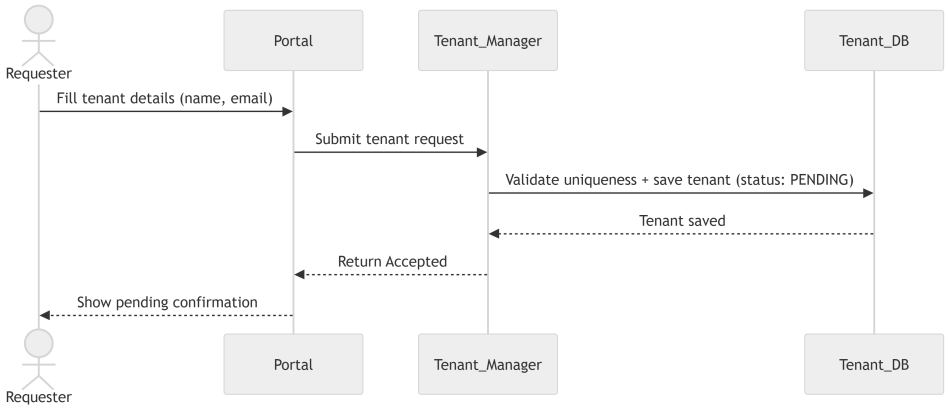


Figure 3.8: Tenant creation request flow

Approval and rejection are shown in Figure 3.9. The notable design point is that the current implementation provisions the tenant inline within a single transaction, so any failure rolls the whole approval back. Rejection instead deletes the tenant record outright.

The intended target architecture models the provisioning as an asynchronous background task. In this design, the approval endpoint returns immediately after setting the tenant to PROCESSING, and a background worker completes the provisioning and updates the status to ACTIVE or FAILED. This design improves responsiveness and supports more complex provisioning flows,

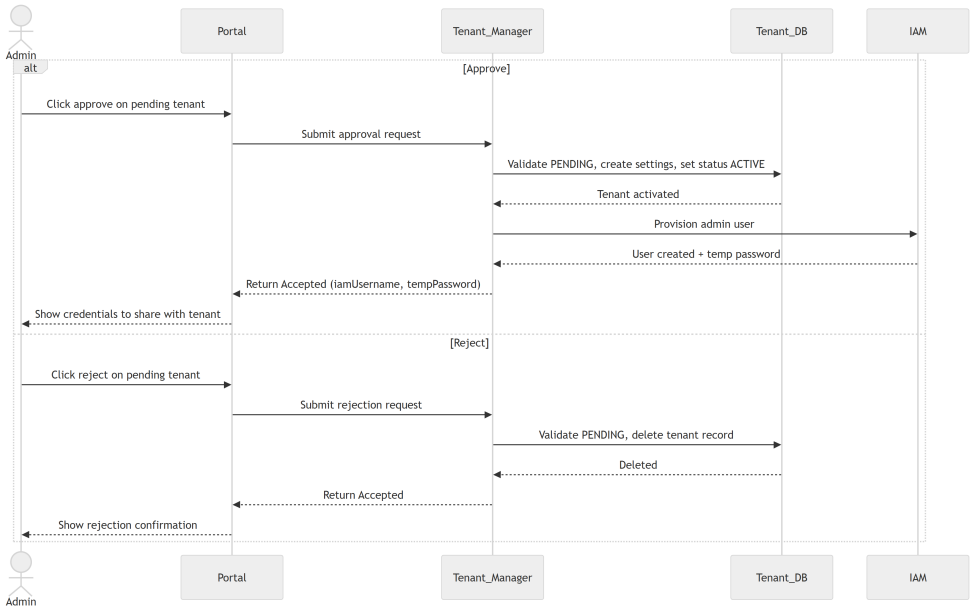


Figure 3.9: Tenant approval and rejection flow (target architecture)

including integration with other platform engines. The current synchronous implementation is a deliberate simplification of this target.

3.6.2 Subscription Management

Subscriptions connect active tenants to product plans. Creating a subscription requires the tenant to be in ACTIVE state and the selected plan to exist and contain the mandatory feature limits `maxUsers` and `maxApiCalls`. Duplicate active subscriptions for the same tenant and product are rejected.

Creating a subscription also propagates product roles. The product's admin role is auto-assigned in IAM to every existing tenant ADMIN user, so they gain access to the newly subscribed product. The initial usage snapshot is seeded accordingly, with totalUsers set to the number of tenant ADMIN users that received the auto-assigned role. Cancelling a subscription reverses this: all of the product's roles are removed from every tenant user in IAM before the subscription is set to CANCELLED.

Plan changes are validated before execution. A downgrade, detected by a lower price or lower limits in the target plan, is rejected if the number of active users exceeds the `maxUsers` limit of the target plan. On a valid plan change, the current subscription is cancelled and a new subscription is created with a fresh period. The usage snapshot for the new subscription inherits the current user count from the previous period to maintain accurate tracking.

The subscription expiration cron job runs daily and transitions subscriptions whose periodEnd has passed to COMPLETED. A separate daily aggregation job creates a `SubscriptionUsageDaily` row for each active subscription and the current day, initializing featuresUsage as an empty JSON object if no row yet exists.

The creation flow, including period calculation and the initial usage snapshot, is illustrated in Figure 3.10.

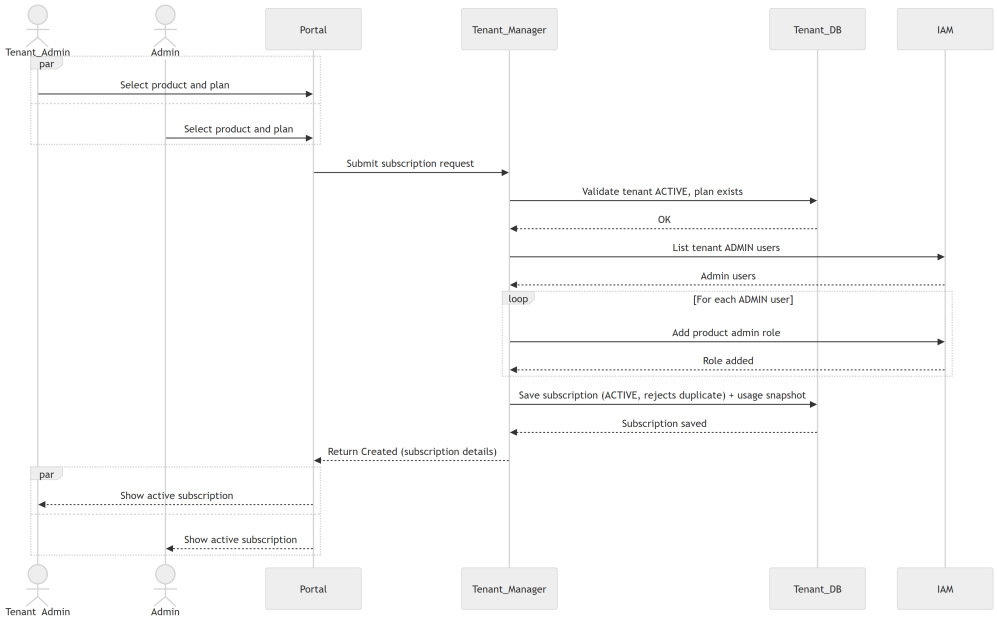


Figure 3.10: Subscription creation flow

3.6.3 User Management

All user operations are scoped to the authenticated tenant and subject to the subscription plan limits. Before creating a user, the system checks the active subscription usage against the plan's `maxUsers` limit and rejects the request if it is reached. The new user receives a tenant-manager role according to its ADMIN or MEMBER attribute, plus any requested product roles, each validated against an active subscription and the per-product `maxUsers` limit (Section 3.3.2).

The user creation flow is shown in Figure 3.11.

3.7 API Design

The API follows an API-first approach, with the contract generated from the Quarkus code using SmallRye OpenAPI. The full contract is available as an OpenAPI 3.1.1 specification. Endpoints are organized under the base path `/api/v1` and grouped into five resource areas.

The endpoints are grouped into five resource areas, summarised in Table 3.1. Two endpoints are public and the remaining twenty-one are protected by the authorization model described in Section 3.2.1. The full endpoint list, with the HTTP verb and purpose of each operation, is given in Appendix A.

The two list endpoints, `GET /tenants` and `GET /tenants/{tenantName}/users`, return a paged envelope `PagedResponseDto { data, total, page, size }` rather than

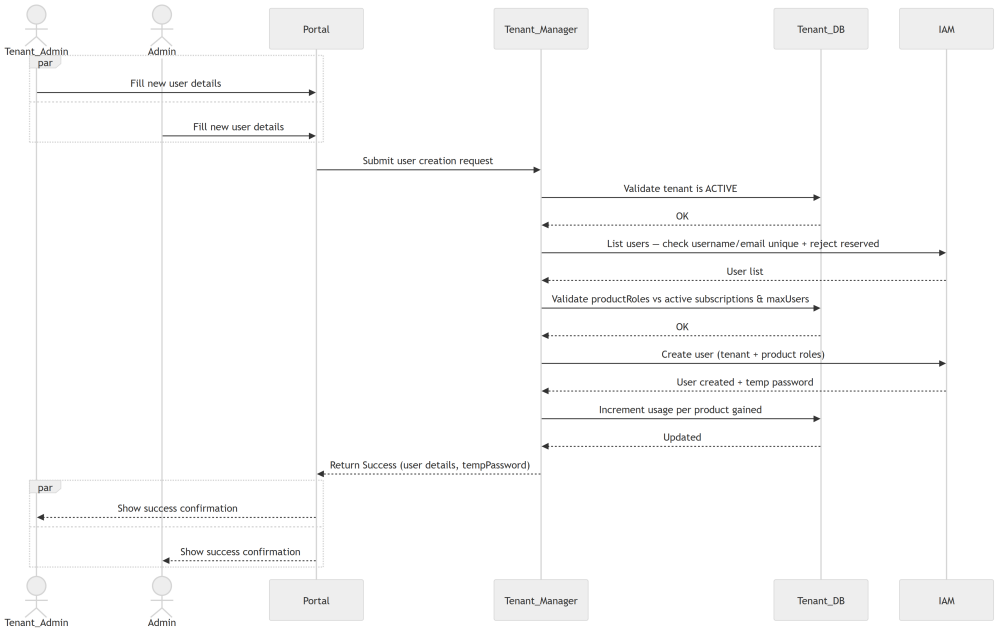


Figure 3.11: User creation flow

a bare array, where `total` is the full filtered count. This lets the portal paginate server-side, requesting one page at a time and reporting the total number of pages, instead of loading the entire collection into the browser. Both endpoints accept `page` and `size` query parameters and fall back to `page = 0` and `size = 20` when these are missing or invalid.

Table 3.1: REST resource groups under `/api/v1`

Resource group	Endpoints	Access
Tenant management	9	1 public, 8 protected
Products	1	public
Settings	2	protected
Subscriptions	4	protected
Users	7	protected

Listing 3.2 shows the public tenant creation request, and Listing 3.3 shows the corresponding response with the tenant in PENDING status.

```
POST /api/v1/tenants

{
  "name": "acme",
  "email": "admin@acme.com"
}
```

Listing 3.2: Create tenant request

```
{
  "name": "acme",
  "email": "admin@acme.com",
  "status": "PENDING",
  "subscriptions": [],
  "createdAt": "2026-05-01T10:00:00Z",
  "updatedAt": "2026-05-01T10:00:00Z"
}
```

Listing 3.3: Create tenant response

Listing 3.4 shows the approve response. It includes the provisioned IAM credentials alongside the updated tenant record.

```
{
  "name": "acme",
  "email": "admin@acme.com",
  "status": "ACTIVE",
  "subscriptions": [],
  "createdAt": "2026-05-01T10:00:00Z",
  "updatedAt": "2026-05-01T10:00:01Z",
  "iamUsername": "sigo-acme",
  "tempPassword": "Tz9xQ2vR..."
}
```

Listing 3.4: Approve tenant response

Listing 3.5 shows a subscription creation request, and Listing 3.6 shows the subscription usage response.

```
POST /api/v1/tenants/acme/subscriptions

{
  "product": { "name": "Trouble Ticketing" },
  "plan":    { "name": "pro" }
}
```

Listing 3.5: Subscribe to product request

```
{
  "plan":          { "name": "pro" },
```

```
  "periodStart":   "2026-05-01T00:00:00Z",
  "totalUsers":    12,
  "totalApiCalls": 18750,
  "featuresUsage": {
    "workOrders":    420,
    "notifications": 1500
  },
  "createdAt": "2026-05-01T00:00:00Z",
  "updatedAt": "2026-05-08T00:05:00Z"
}
```

Listing 3.6: Subscription usage response

Listing 3.7 shows a user creation request that assigns a product role, and Listing 3.8 shows the response, which echoes the assigned product roles as `{productName, role}` references and returns the one-time temporary password.

```
POST /api/v1/tenants/acme/users

{
  "username":  "jdoe",
  "email":     "jdoe@acme.com",
  "firstName": "John",
  "lastName":  "Doe",
  "role":      "MEMBER",
  "productRoles": [
    { "productName": "Trouble Ticketing", "role": "Operator" }
  ]
}
```

Listing 3.7: Create user request with a product role

```
{
  "username":  "jdoe",
  "email":     "jdoe@acme.com",
  "firstName": "John",
  "lastName":  "Doe",
  "role":      "MEMBER",
  "status":    "ACTIVE",
  "productRoles": [
    { "productName": "Trouble Ticketing", "role": "Operator" }
  ],
```

```
  "tempPassword": "Kp4mWq8t..."
}
```

Listing 3.8: Create user response

3.8 User Interface

The frontend is an Angular standalone-component application exposed through Webpack Module Federation for integration into the SIGO portal shell. Its dashboards exercise the full tenant-management lifecycle against the live backend: tenant registration, lifecycle operations, settings, subscriptions, and user management.

The interface provides three role-based dashboards: a public dashboard for tenant registration and product browsing, an admin dashboard for tenant lifecycle management and filtering, and a tenant-scoped dashboard for settings, subscriptions, and user management. The role is selected at runtime, and an optional Bearer token can be supplied for protected operations.

The tenant and user listings are rendered as paginated tables whose filter controls share the same panel as the results. Pagination is performed server-side, fetching one page at a time against the paged API endpoints rather than loading the full collection into the browser, and the search filters are applied only when the search action is triggered. A user's product roles are shown compactly in the user table, with a hover tooltip revealing any assignments beyond the first.

Figure 3.12 shows the administrative dashboard with the tenant list and status filters.

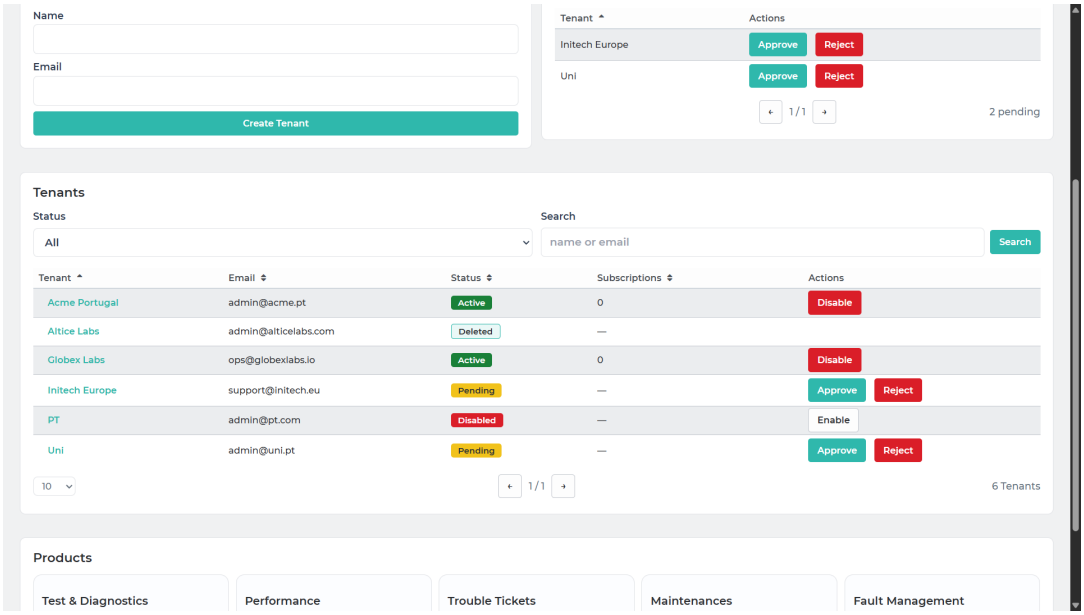


Figure 3.12: Admin dashboard: tenant list with status filters

The remaining views, namely the tenant detail page, the approve-tenant modal showing the provisioned IAM credentials, subscription management, and user management, are collected in Appendix B.

3.9 Summary

This chapter described the proposed tenant-management solution for SIGO. The solution provides a Quarkus backend organized in strict layers, PostgreSQL persistence with application-layer tenant isolation enforced through the authenticated tenant claim, OIDC-based authentication through NOSSIS IAM, NOSSIS IAM integration for identity provisioning and user lifecycle management, and an Angular frontend with role-based dashboards. The tenant lifecycle is modelled as an explicit state machine with guarded transitions and integrated identity operations. Subscription management enforces plan limits and preserves history through status transitions. The IAM integration spans tenant admin provisioning at approval time, tenant user management, and token-based ownership enforcement. The sequence diagrams presented describe the intended target architecture. The current implementation covers the synchronous core flows, with asynchronous provisioning and cross-engine integration planned as future work. The next chapter defines the experimental process used to evaluate the implemented flows.

Chapter 4

Implementation

4.1 Development Approach

The tenant-manager module was built in a deliberate sequence designed to validate each layer before building the next. The process began with research across four areas relevant to the design decisions ahead: multi-tenant database organisation strategies, subscription and billing plan modelling, resource usage limits as applied in cloud platforms such as Kubernetes, and portal integration patterns. This research phase informed the data model and lifecycle design before any code was written.

The first concrete output was a set of sequence diagrams covering the main operational flows (tenant creation, approval, disable, delete, and user provisioning), alongside portal mockups for both the admin and tenant-scoped views. These artefacts were a shared reference between the backend design and the frontend prototype, and shaped the API contract before implementation began.

The API contract was defined next. Endpoint paths, request and response shapes, status codes, and error codes were specified and validated against the use cases derived from the requirements. The data model was designed in parallel, establishing the PostgreSQL schema, table structure, and entity relationships, including the placement of feature limits as a JSON column on the plan entity.

Backend implementation followed the layered architecture described in Section 3.2: entities and repositories first, then service layer business logic, then mapper and DTO definitions, and finally the REST resources. Each service method was tested incrementally against a live database as it was written, with integration tests added as soon as an endpoint reached a stable state.

The frontend was implemented last, built against the already-stable API contract. Because the contract was defined before the frontend existed, no API shape changes were required to accommodate the User Interface (UI), and the frontend served to exercise the contract visually and validate flows against the live backend.

4.2 Key Technical Decisions

Framework: Quarkus The backend is implemented in Quarkus because it is already the established framework across SIGO. Beyond consistency, Quarkus provides native support for OIDC token validation, Hibernate ORM Panache, SmallRye OpenAPI contract generation, SmallRye Health, and Micrometer metrics, all of which are required by this module, without additional integration work. The alternative of introducing a different framework for a single module would create operational inconsistency with no functional benefit.

IAM-backed users instead of a local user table A deliberate decision was made to store all tenant user accounts in the NOSSIS IAM service rather than in a local user table in PostgreSQL. This keeps the platform's identity store consolidated: Role-Based Access Control (RBAC) assignments, credential management, and user lifecycle are all handled by IAM, and the tenant-manager does not need to implement password storage, hashing, or authentication flows for users. The tradeoff is that IAM becomes a hard runtime dependency (any IAM outage makes user management operations unavailable), and the Comma-Separated Values (CSV) bulk import mechanism introduces coupling to a specific IAM integration contract that may require adaptation if the IAM service evolves its import interface.

CSV bulk import for IAM operations The `IamUserManagerService` interacts with the NOSSIS IAM service through a CSV bulk import endpoint (`POST /api/import`) authenticated via the `client_credentials` OAuth 2.0 grant. This interface was already established as the integration pattern for SIGO, so reusing it avoids introducing a separate Keycloak Admin REST API integration path. The tradeoff is that CSV-encoded payloads are structurally more fragile than JSON: field ordering and escaping must be precise, and validation errors from the IAM service are harder to map to specific fields in the response.

Explicit tenant lifecycle state machine Tenant status is modelled as an explicit state machine with a fixed set of named states (PENDING, PROCESSING, ACTIVE, DISABLING, DISABLED, DELETING, DELETED, FAILED, REJECTED) and guarded transitions enforced in the service layer. The alternative, using boolean flags such as *active* and *deleted*, would be simpler to implement initially but would produce ambiguous combinations and require scattered conditional checks throughout the codebase. The explicit state machine makes illegal transitions impossible to express, localises transition logic in a single place, and produces domain-oriented error responses that clients can handle without parsing message strings. The tradeoff is a more complex validation layer and the need to handle intermediate states (PROCESSING, DISABLING, DELETING) that represent asynchronous work not yet fully implemented.

Soft delete via DELETED status Tenant records are never physically removed after approval. Instead, a soft-delete operation transitions the tenant to the DELETED status, preserving the

record and all associated subscription history. This decision directly satisfies the auditability non-functional requirement: historical billing periods, usage snapshots, and status transitions remain queryable after a tenant is removed from active use. Physical deletion was rejected because it would cascade to subscription records, destroying data that may be required for billing reconciliation or support investigations. The tradeoff is that the tenant table grows indefinitely and list queries must filter by status to exclude deleted records.

4.3 API Validation

The module is validated through integration tests rather than unit tests. This choice reflects the nature of the business logic: the service layer depends on both PostgreSQL for state persistence and the NOSSIS IAM service for identity operations. Mocking either dependency would validate the mocks rather than the actual system behaviour. A test that sends a real HTTP request, triggers a real state transition in PostgreSQL, and asserts the resulting response body and status code provides more confidence than an isolated unit test of a method whose correctness depends on database constraints and IAM responses.

The test harness uses JUnit 5 with Playwright's `APIRequest` context for HTTP execution. Playwright was chosen because it provides a built-in HTTP client without requiring a dedicated REST testing library, fits the JUnit 5 lifecycle hooks, and produces readable request construction and assertion code. Tests run against a dedicated database seeded with the product catalog only, with no pre-existing tenants, which ensures a deterministic starting state for every run. Each test case creates the tenants it needs and relies on test-scoped cleanup to avoid ordering dependencies between tests.

All 23 implemented endpoints are covered by more than 190 test scenarios across sixteen test classes, spanning happy paths, negative cases, and pagination and not-found edge cases. Every documented business rule has at least one corresponding test, including the lifecycle transition guards, the subscription downgrade and duplicate guards, reserved-username enforcement, usage-counter tracking, the product-role rules, and the tenant-ownership checks that restrict a tenant-scoped token to its own data.

4.4 Summary

This chapter described how the tenant-manager module was built and validated. The development followed an API-first, backend-before-frontend sequence grounded in prior research and contract definition, ensuring that each layer was stable before the next was built upon it. Key technical decisions (IAM-backed user storage, explicit lifecycle state machine, soft delete, and composite subscription keys) were driven by the auditability, isolation, and consistency requirements established in Chapter 3. The integration test suite covers all 23 endpoints and every documented business rule. The next chapter evaluates the resource efficiency of the implemented solution against the platform's operational requirements.

Chapter 5

Experimental Process

This chapter describes the experimental process used to evaluate the proposed solution. Its main subject is *resource efficiency*: whether the pooled, schema-per-tenant model that the platform adopts can serve many small tenants from a single small infrastructure footprint while holding a latency objective, and where that model's scaling limit actually lies. This answers research question Q3 and delivers the resource-efficiency evaluation that Chapter 4 anticipates.

5.1 Resource-Efficiency Study

This study evaluates the resource efficiency of the pooled, schema-per-tenant model: how many small tenants it can serve from a single footprint, where it saturates, and what the missing elasticity layer adds on top of it. The subsections below set out the goals and hypotheses, the workload, the tenancy models compared, the method and environment, and the scenarios and metrics.

5.1.1 Goals and Hypotheses

The platform runs a *pooled* application tier (one shared deployment set serving all tenants) over a *schema-per-tenant* data tier (isolated PostgreSQL schemas on one shared instance). This is the midpoint of the silo-to-pool isolation spectrum: a shared instance with per-schema isolation. The study evaluates the efficiency of this model and the effect of adding the missing elasticity layer (autoscaling, request right-sizing, admission guardrails, and connection pooling) on top of it. It is framed as four hypotheses:

- **H1 (density):** the pooled, schema-per-tenant model serves on the order of 20–30 small tenants within a single small footprint at the target objective.
- **H2 (elasticity):** a Horizontal Pod Autoscaler (HPA) holds the objective as offered load grows.
- **H3 (bottleneck):** the study identifies where the model's scaling bottleneck lies, locating the tier and resource at which it saturates.

- **H4 (isolation):** request right-sizing, limits, and quotas contain a noisy-neighbour tenant.

The target objective is a proposed SLO, justified by the norms of interactive operational tooling and the light expected workload: **p95 API latency $\leq$ 500 ms, p99 $\leq$ 1 s, and error rate $<$ 1%** under expected load, with 99.9% availability as a design goal that is not testable in short runs.

5.1.2 Workload Model

The workload is synthetic, derived from the host's real client data with assumptions stated explicitly so that the validity discussion can reference them. The base unit is a *small tenant*: approximately 20 users generating about 5 issues per day over an eight-hour business window. The request mix is read-heavy, dominated by ticket list and search operations and ticket-detail reads, with writes (create, update, comment) a small minority. A peak factor of three to five is applied to the average to obtain the peak rate.

From these figures, one small tenant is estimated at roughly 30 requests per user per active hour, about 600 requests per hour, or **0.17 Requests Per Second (RPS) on average** with a peak near **0.5–0.85 RPS**, and about 5 writes per day. This deliberately tiny per-tenant load is the source of the density opportunity: most of a dedicated installation's capacity sits idle. An *aggressive* tenant is modelled as a sustained create burst to saturation, used to probe the noisy-neighbour case (H4).

5.1.3 Tenancy Models Compared

The baseline comparison is *siloed* (one full application stack and a private database per tenant) against the chosen *pooled, schema-per-tenant* model (one shared stack, isolated schemas on one shared instance). Both keep the same per-tenant database sizing from the client ground truth. One small footprint is defined, from the client data, as approximately 4 Central Processing Unit (CPU) cores and 4 GiB of application memory plus a 4-CPU/8-GiB database instance, claimed to serve 20–30 such tenants. That density figure is the central claim the study validates or corrects. The alternative isolation models (full silo, and a fully pooled single schema with a tenant discriminator column) are discussed as design alternatives in Chapter 3 and are not separately implemented. The full silo is rejected for defeating the efficiency goal and the single-schema pool for its weaker data separation and blast radius.

5.1.4 Method and Environment

The evaluation is *hybrid*: an empirical layer establishes measured per-request costs on a real but small cluster, and a scale-out layer extrapolates those costs to tenant counts beyond a laptop's capacity. Everything runs locally. The production cloud infrastructure is used only as a read-only source of sizing numbers and is never deployed to, so no cloud cost is incurred.

Empirical layer. A local Kubernetes cluster (kind) runs the *real* tenant-manager service, a PostgreSQL instance, and the kube-prometheus-stack with metrics-server, on a developer laptop with Docker inside Windows Subsystem for Linux (WSL), with the cluster confined to 4 vCPUs and 8 GB of Random Access Memory (RAM) through a WSL2 resource cap so the capacity ceilings are stable and known. Application load is generated with k6 against the service's endpoints, and replica counts and per-pod CPU are sampled over each run. This layer measures the application tier directly.

TTK-engine layer. To exercise a genuinely CPU-bound, stateful data-plane tier, the real TTK engine was deployed and load-tested. It runs as a Quarkus fast-jar against a dedicated PostgreSQL instance (schema-per-tenant), with the same 250m request / 500m limit per pod used for the control plane so that density and autoscaling regimes are comparable across tiers. Because the engine's external dependencies (the IAM service, Kafka, and object storage) are irrelevant to its compute and database cost, they are stubbed: a build-time authentication bypass admits requests without a token, the messaging channels are disabled, and the object-storage client is pointed at an unused endpoint, so the measured cost is the real create/list path (validation, mapping, rule evaluation, and the Structured Query Language (SQL) write or query) without external noise. This layer is what makes the elasticity claim (H2) testable on a tier that actually saturates on CPU, and it hosts the realistic per-tenant capacity and noisy-neighbour experiments.

Data-tier load generator. To measure the schema-per-tenant data tier without running the full TTK engine, a synthetic generator creates K tenant schemas in one PostgreSQL instance, seeds each with a small tenant's ticket history, and drives a TTK-like query mix (list 50%, detail 25%, search 20%, write 5%) with pgbench executing inside the database pod, so the measurement is pure database cost with no network noise. For this data-tier density measurement the full TTK engine is deliberately bypassed: it would add Kafka, IAM, and object-storage dependencies for negligible fidelity gain at five issues per day per tenant (the engine itself is exercised separately, with those dependencies stubbed, in the TTK-engine layer above). This mirrors the method of Truyen et al. [16], who used a synthetic application and excluded shared-component dependencies to measure isolation cost cleanly. The omission of daily table partitioning is recorded as a conservative simplification (partitioning only helps, through partition pruning).

Scale-out layer. To reach tenant counts beyond the laptop's real CPU, KWOK schedules many phantom nodes and pods on a real Kubernetes control plane (no real workload runs on them), exposing scheduling, packing, and `ResourceQuota` behaviour at 30 or more tenants. The phantom nodes are sized as one client small footprint, and the per-request costs measured in the empirical layer calibrate the interpretation.

Table 5.1 summarises the tools used across all of these layers, their roles, and where each runs.

Table 5.1: Simulation environment and tools

Tool	Role	Layer
Docker (in WSL)	container runtime	all
kind	local Kubernetes (K8s) cluster (nodes as containers)	empirical
kube-prometheus-stack + metrics-server	metrics, HPA input	empirical
k6	application load generator	empirical
PostgreSQL + pgbench	schema-per-tenant data-tier load	empirical
PgBouncer	connection pooling (transaction / session)	empirical
KWOK	phantom-node scale-out to 30+ tenants	scale-out

5.1.5 Scenarios, Ablation, and Metrics

The core experiment is an ablation between configurations of the same pooled stack:

- **A0 (floor):** the platform as it ships for application workloads, pinned to a single replica with no autoscaler.
- **A1 (reactive):** the same stack with right-sized requests and limits, a CPU-target HPA scaling out from a single warm replica, and `ResourceQuota` and `LimitRange` guardrails. This is the proposed elasticity layer measured as it behaves in production: the base replica is warm, but scale-up is reactive, so each replica the HPA adds cold-starts inside the measured load.
- **A2 (warm ceiling):** a pre-provisioned warm fleet pinned at the HPA's maximum replica count, with no autoscaler. Applied only on the CPU-bound TTK engine, it isolates the steady-state scale-out benefit (A2 versus A0) from the reactive scale-up tax (A1 versus A2) that an instant, perfect scaler would avoid.

The ablation is applied by sweeping the offered load upward until the SLO breaks, on both the control-plane tenant-manager and the TTK engine, and the pooled and siloed footprints are compared on the packing axis (H1). A noisy-neighbour run then adds an aggressive tenant alongside a quiet baseline to test whether the guardrails protect the others (H4). The data-tier density sweep raises the pgbench client count against the seeded schemas to find where the shared instance saturates (H3), and a pooler-mode study compares transaction-mode against session-mode connection pooling on the schema-per-tenant access pattern.

The metrics collected are tenant density (tenants per core and per GiB), p95 and p99 latency, throughput, error rate, HPA replica count over time, active database connections and schemas, and pod schedulability. The reported latency and throughput figures are taken from the k6 and pgbench summaries, and the pooling figures from PostgreSQL and PgBouncer runtime statistics.

5.1.6 Load-Generation Methodology

Two methodological controls proved necessary to obtain steady-state figures rather than start-up artefacts, and both are reported here because they materially affect the measured SLO.

Open versus closed load model. A *closed* model runs a fixed pool of virtual users looping request–think–request, so the arrival rate follows the server's response time. An *open* model (k6's constant-arrival-rate executor) injects requests at a fixed rate, independent of how fast the server replies. The closed model's fixed concurrency is the natural way to expose the throughput headroom a scaling tier gains, so the TTK-engine elasticity runs use it, with a gradual virtual-user ramp and a warm-up to avoid a synchronised start-up burst against the bounded connection pool. The open model's offered-rate semantics fit the saturation-knee question and match the realistic per-tenant workload, so the control-plane capacity figures use it.

Warm-up under a tight CPU limit. Under the 500m per-pod CPU limit, a freshly started JVM pays a substantial just-in-time-compilation and garbage-collection tax during its first minute, and if that work falls inside the measurement window it inflates p95 from milliseconds to seconds. Each measured run is therefore preceded by a 60-second warm-up at the target rate on the same code path, so the reported figures reflect the steady state. The magnitude of this tax under a constrained CPU allowance is reported as a finding in its own right (Section 6.1.3).

5.2 Summary

This chapter defined the experimental process for the resource-efficiency study of the pooled, schema-per-tenant model, framed by four hypotheses and a proposed latency SLO. The study uses a synthetic small-tenant workload, a hybrid empirical-plus-scale-out method on a local cluster, and an A0/A1/A2 ablation that isolates the effect of the proposed elasticity layer and separates its steady-state benefit from its reactive scale-up cost. The next chapter reports and discusses its results.

Chapter 6

Results and Discussion

This chapter presents the results of the resource-efficiency study defined in Chapter 5 and discusses their implications. Section 6.1 reports the study: control-plane autoscaling on the tenant-manager, data-plane elasticity and realistic per-tenant capacity on the TTK engine, noisy-neighbour isolation, scale-out packing, data-tier density, and the connection-pooling lever with its isolation limit. Section 6.2 then discusses the findings against the four hypotheses and the threats to validity. The conclusions and future work that follow from these results are presented in Chapter 7.

6.1 Resource-Efficiency Results

This section reports the resource-efficiency study, moving from the application tier inward to the shared data tier and finally to connection pooling. Each result is presented as measured. The discussion (Section 6.2) then draws them together against the four hypotheses, including where the binding constraint lies.

6.1.1 Application-Tier Autoscaling (H2)

The autoscaling ablation was run on the real tenant-manager, comparing A0 (one pinned replica) against A1 (HPA on a 70% CPU target, one to five replicas), with each arm capped at 500m CPU. Load was generated with the same open-model constant-arrival-rate method used throughout this work (Section 5.1.6): for each of the read, write, and mixed paths the offered rate was swept upward until the SLO broke, locating the saturation knee, and A0 was then compared against A1 at that knee over three warmed repetitions on an otherwise idle single node. On one 500m pod the read, write, and mixed paths all saturate at about the same offered rate, near 300 RPS. Below that rate p95 stays under the 500 ms objective.

The HPA gives no benefit, and on reads it harms. Table 6.1 reports the result at the 300-RPS knee. A single pod already meets the SLO on every path (warm-run p95 in the tens to low hundreds of milliseconds across the three paths, at 0% error), and turning on the autoscaler does not raise throughput on any of them: all arms sustain about 290 RPS, and the HPA adds replicas

that carry no extra load. The autoscaler climbs to three replicas on the brief CPU spike of the opening burst and then stops there, leaving two of the three idle: once requests begin blocking on the shared database the per-pod CPU falls back below the 70% target, so no further pod is requested and the short run's stabilisation window holds the idle pods in place rather than scaling back down. That the extra replicas sustain no additional throughput is itself the evidence that the binding constraint is the shared database, not pod CPU. On the read path the autoscaler is actively counterproductive: A1's p95 degrades from about 75 ms to roughly 1.6 s, because the reactive scale-up spins up cold replicas whose just-in-time warm-up falls inside the measurement window, the same transient tax later measured on the TTK engine (Section 6.1.2). This cold-start tail is not unique to reads: Table 6.1 lists all three repetitions, and on the warmed write and mixed paths one repetition of three also spikes to 1.1–1.6 s while the other two stay in the low hundreds of milliseconds, whereas on the read path all three repetitions spike. Reads are the most consistently affected because each request is the cheapest, so the cheap reads cycle connections onto the not-yet-warm replicas fastest, placing a steady fraction of every run on a cold pod throughout its warm-up window. Reporting every repetition rather than a single central value makes this intermittent reactive-scaling tax explicit instead of hiding it. The reproducible signal is this A1-versus-A0 contrast. The absolute rate is bounded by the single-node cluster.

Table 6.1: Control-plane autoscaling ablation

Path	**Arm**	**Throughput (RPS)**	**p95**	**Errors (%)**
read	A0	285	72 / 81 / 73 ms	0.0
	A1 (1→3)	275	1.68 / 1.60 / 1.46 s	0.0
write	A0	291	177 / 150 ms / 1.07 s	0.0
	A1 (1→3)	291	293 / 186 / 196 ms	0.0
mixed	A0	291	1.43 s / 110 / 85 ms	0.0
	A1 (1→3)	291	1.62 s / 94 / 103 ms	0.0

The verdict for H2 on the control plane is therefore unambiguous: horizontal autoscaling is the wrong lever for the tenant-manager. A single right-sized pod meets the objective up to the saturation knee, the autoscaler adds replicas that the shared data tier renders inert, and on the read path the reactive scale-up makes latency worse rather than better. The binding constraint is consistently the shared data tier, so the effective levers are request right-sizing, admission guardrails, and connection pooling, not horizontal autoscaling. Whether application elasticity instead becomes the right lever on a genuinely CPU-bound stateless tier is the open question this control-plane result raises. Rather than settle it for the TTK engine analytically, the next subsection measures it directly (Section 6.1.2). This matches Truyen et al. [16], whose method locates the cut-off point and shows that elasticity raises the cut-off rate rather than lowering low-load latency.

6.1.2 Data-Plane Elasticity on a CPU-Bound Tier (H2)

To test the prediction that autoscaling pays off precisely where a tier is genuinely CPU-bound, the real TTK engine was deployed and load-tested with a 50% create / 50% list mix. Unlike the control plane, the engine performs substantial per-request compute before touching the database (priority and team rule evaluation, SLA computation, mandatory-field and catalogue validation, object mapping, and JSON serialisation). At only 10 virtual users an uncapped pod already shows a p95 of about 226 ms against the control plane's 4 ms, confirming the tier is compute-heavy. The elasticity ablation (HPA on a 70% CPU target, one to six 500m pods) was run here decomposed three ways at a fixed load, to separate the steady-state benefit of scaling out from the reactive scale-up tax, with three repetitions per arm. The two load points, 30 and 40 virtual users, each a closed-loop client issuing requests back-to-back at roughly 2–3 per second, sit at the single pod's CPU knee, where A0 is saturated and any scaling benefit is visible. Reporting both confirms the effect is not specific to one load level.

The result is a large but qualified benefit, and the three-way decomposition (Table 6.2) shows exactly where it comes from. Three arms are run at a fixed load. A0, one static pod, is the floor. A1, the reactive HPA starting from one warm pod and cold-starting each replica it adds as it scales to six under load, is the realistic production case. A2, six pinned warm pods, is the ceiling an instant, perfect scaler would give. At 30 virtual users A0 sustains 11.6 RPS. The reactive A1 lifts this only to 15.2 RPS (about 1.3×), because the scale-up tax falls inside the short run. The warm A2, by contrast, reaches 36.6 RPS, roughly three times the floor, and restores the latency SLO (its residual errors are data-tier, treated below). The 40-virtual-user point behaves identically. This is the exact opposite of the database-bound control plane and the empirical confirmation of H2 on the tier the model predicted, with one sharpening: on a genuinely CPU-bound stateless tier horizontal scale-out works, but the win is realised by a warm fleet, not by reactive scaling alone.

Table 6.2: TTK-engine scale-up decomposition (p95 over successful requests)

Load	Arm	Throughput (RPS)	p95	Errors (%)
30 Virtual Users (VUs)	A0 (floor)	11.6	804 ms	72.6
	A1 (reactive 1→6)	15.2	2.30 s	37.1
	A2 (warm ceiling)	36.6	89 ms	22.1
40 VUs	A0 (floor)	13.4	1.22 s	80.8
	A1 (reactive 1→6)	18.4	2.40 s	50.5
	A2 (warm ceiling)	38.2	134 ms	31.4

The warm ceiling meets the latency SLO comfortably: six warm pods (A2) hold a p95 of 90–135 ms, a 4–5× margin under the 500 ms objective, so at steady state the engine fleet is not the constraint and horizontal scale-out fully works. The reactive HPA (A1), however, pays a heavy transient tax: it captures only about 40–48% of the ceiling throughput and its mean p95 blows out to 2.3–2.4 s, because the cold one-to-six scale-up (pod scheduling plus a roughly 12-second JVM start, repeated per new pod) dominates a short run. A1 is also by far the noisiest arm, precisely

because the outcome depends on how fast the autoscaler happened to react in each run: at 40 VUs its per-repetition p95 ranged from about 0.9 s (a fast-reacting run) to 3.2 s, so the tabulated mean understates the spread of this reactive-scaling tax. A separate run in which the warm-up was allowed to pre-scale the fleet, so the measured window began with roughly three already-warm pods rather than one cold pod, landed between the two arms at 24.3 RPS: direct evidence that the gap between A1 and A2 is the cold-start tax rather than the scale-out itself, and a measurement of the warm-reserve mitigation discussed below. The residual errors at the warm ceiling (22–31%) come from the single shared PostgreSQL instance saturating: the H3 "shared data tier is the ceiling" result, reappearing one tier down.

The lesson generalises the H2 verdict: horizontal autoscaling is the correct lever for a CPU-bound stateless tier, and at steady state it restores the latency SLO with a wide margin, but a purely reactive CPU-target HPA mitigates collapse rather than fully restoring latency under an aggressive step load, because the scale-up lag and JVM warm-up fall inside the demand spike. The actionable refinement is to keep a small warm pod reserve (a higher floor, slower scale-down) so the fleet is closer to the ceiling than to the floor when load arrives.

6.1.3 Realistic Per-Tenant Capacity (H1)

The elasticity experiments above drive the tier to saturation to find where it breaks. This subsection instead measures the SLO headroom under the *realistic* workload of Section 5.1.2, to answer directly how many real tenants one 500m TTK-engine pod serves. The 30-tenant read target is modelled as 600 users each refreshing their ticket list once every 30 seconds, an offered 20 searches/s, or 0.667 searches/s per tenant. This figure is the read component on its own, not the full per-tenant request mix estimated in Section 5.1.2, and it is deliberately read-heavy, so the pod is exercised on its dominant path. Writes are modelled separately and are negligible (about 0.005 creates/s across the 30 tenants, from 5 issues per day per tenant), so the create path is swept on its own to find its ceiling.

Reads are not the constraint (Table 6.3). At the 20-RPS demand point a single warm pod answers with a p95 of about 11 ms at 0% error (median of three repetitions), and the read rate sweeps to 100 RPS, roughly 150 tenants, still at 0% error with a p95 in the tens of milliseconds. The realistic read load of the entire 20–30-tenant target uses a small fraction of one pod.

Table 6.3: TTK-engine read sweep

Offered (RPS)	Approx. tenants	p95	Errors (%)
20	30	11 ms	0
40	60	17 ms	0
60	90	31 ms	0
80	120	22 ms	0
100	150	9 ms	0

The write path has a sharp, pool-bound ceiling (Table 6.4). A single pod sustains 7 creates/s at 0% error and tens of milliseconds p95. At 8 creates/s it hits a hard cliff where the p95 pins to exactly 3.00 s, the connection pool's acquisition timeout, and about a third of creates fail. The *successful* creates at 8/s stay fast (median 24 ms): the engine and database are idle-fast, and the failures are purely connection-pool exhaustion as the pool saturates and the marginal request waits out its timeout. The write ceiling is therefore bounded by the connection pool, not by CPU, reinforcing the H3 finding at the per-pod level. Even so, one pod's 7 creates/s sits far above demand: at 5 issues per day per tenant concentrated in the eight-hour business window (about 1.7×10^{-4} creates/s each), it absorbs the active-hours write demand of roughly 4×10^{4} tenants, so for the stated 20–30 tenants the only realistic write risk is a synchronised burst, which the elasticity layer above absorbs.

Table 6.4: TTK-engine create-path write knee

Offered (creates/s)	p95	Errors (%)
6	28–73 ms	0
7	31–78 ms	0
8	3.00 s	32

The JVM warm-up tax is a measurable SLO artefact. These figures are reported only after the 60-second warm-up of Section 5.1.6, and the size of that correction is itself a finding. Without it, the same 20-RPS read load on a freshly restarted pod gives a p95 of about 2 s rather than 11 ms, a roughly 180-fold inflation, and the cold write knee appears one step lower (6 rather than 7 creates/s). Under a tight per-pod CPU limit the JVM warm-up is a first-order tax on tail latency, which a production deployment should hide behind a warm pod reserve and readiness gating.

6.1.4 Noisy-Neighbour Isolation (H4)

The density argument rests on tenants sharing one engine and one database. H4 asks whether the platform protects a well-behaved tenant when one tenant misbehaves. Two concurrent load streams hit the same engine: a *baseline* of the quiet 30 tenants at the realistic 20-searches-per-second read load (the reported SLO), and an *aggressor*, a single tenant flooding creates far past its share (5 issues per day) and past the 7-per-second pool knee.

Autoscaling does not isolate. Table 6.5 reports the quiet tenants' SLO with the aggressor at 30 creates/s. Alone, the baseline sees a 41 ms p95 at 0% error. With the aggressor and no HPA, it collapses to a 4.4 s p95 at 96% error. With the CPU-target HPA enabled it is no better (6.1 s, 96%). The autoscaler scaled only one to three replicas and never rescued the quiet tenant, for two compounding reasons. For the first roughly two minutes per-pod CPU sat at 3–6% while every request blocked on *connection-pool* acquisition, so the CPU signal never crossed the 70% target to trigger scale-up. Later, when CPU did eventually peg the cap and pods were added, the extra replicas

merely opened more connections against the same shared PostgreSQL instance (a 100-connection ceiling), while the flood kept the pool saturated and scale-up lagged the burst. The quiet tenants' errors are the 3-second pool-acquisition timeout, not application errors. `ResourceQuota` and per-pod limits cap the namespace's blast radius but give *no* intra-pool isolation between tenants sharing a pod.

Table 6.5: Noisy-neighbour isolation (quiet-tenant SLO)

Arm	Configuration	p95	Errors (%)
c0	baseline alone, 1 pod, no HPA	41 ms	0
np	baseline + aggressor, 1 pod, no HPA	4.39 s	96
pr	baseline + aggressor, HPA 1→6	6.12 s	96

The harm threshold is the pool knee. Sweeping the aggressor's rate at a fixed single pod (Figure 6.1) shows the collateral damage is sharply gated by the connection pool. While the aggressor stays below its own create knee (5/s) the quiet tenants are untouched (50 ms, 0% error). The instant it crosses the knee (8/s) the quiet SLO collapses (3.0 s, 24% error), worsening monotonically to 93% at 30/s. The same 7–8/s knee recurs across the warm write-knee sweep and both isolation runs, an internally consistent signature that the shared connection pool is the unprotected resource and its capacity is the exact collateral-damage boundary.

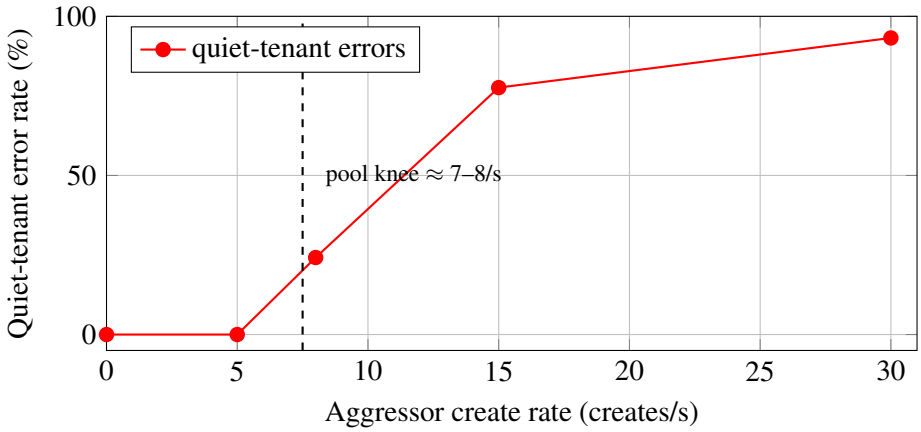


Figure 6.1: Noisy-neighbour harm threshold vs aggressor rate

The verdict on H4 is therefore negative for the autoscaling-and-quota guardrails and pointed in its remedy: `ResourceQuota`, per-pod CPU limits, and a CPU-target HPA protect the node and namespace, but not tenants from each other inside a shared pod. Isolation on the pooled data plane requires *connection-pool governance* (per-tenant pool quotas, a fairness-aware pooler, or write rate-limiting), which is the connection-pooling lever the proposal already centres on (Section 6.1.7). The stronger reading is that autoscaling is the wrong lever for pool-bound multi-tenant contention, fully consistent with the elasticity and capacity results above.

6.1.5 Scale-Out Packing and the Admission Guardrail (H1)

This layer checks one thing: whether the Kubernetes orchestration layer (scheduling, bin-packing, admission) ever limits tenant density. It is a paper-capacity check, not a performance test: KWOK schedules *phantom* pods that carry the real resource *requests* but run no workload, so the scheduler and `ResourceQuota` do the same arithmetic as for real pods at counts a laptop could never run. Sweeping 1 to 100 tenants on footprint-sized phantom nodes (Figure 6.2), the siloed model scales $O(N)$ (one full stack per tenant, memory-bound at one tenant per 4-GiB footprint) while the pooled model is flat at $O(1)$ *by construction*, since a new tenant adds only a schema and a pool slot, not a pod. At $N = 100$ all 100 siloed pods scheduled with zero pending while the pooled stack stayed at one footprint, a 100× packing ratio.

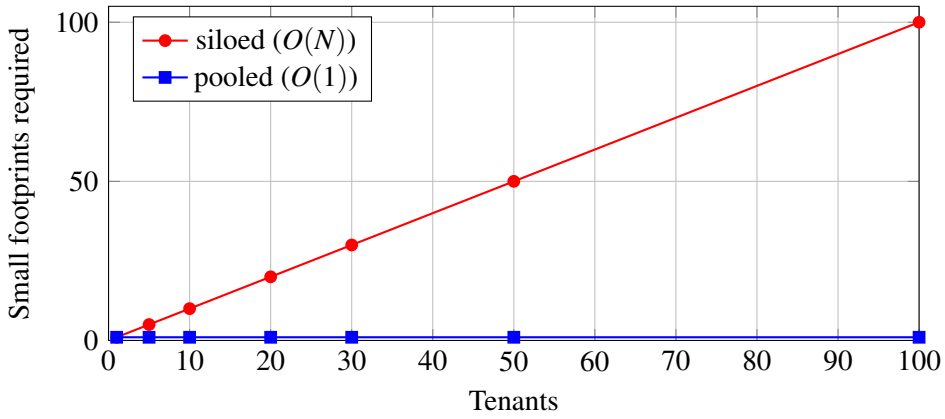


Figure 6.2: Scale-out packing: siloed $O(N)$ vs pooled $O(1)$ (KWOK phantom pods, packing axis)

The run's contribution is bounded but real: it rules out the orchestration layer as the density limit through 100 tenants, and it validates the admission guardrail: under a one-footprint `ResourceQuota` the first siloed tenant was admitted and the second rejected with an explicit quota-exceeded error, turning "too many tenants" into a hard boundary instead of silent overcommit. Two caveats keep it honest: "fits" means the declared *requests* fit, not that the system serves that tenant within the SLO (the question answered by Sections 6.1.3, 6.1.4, and 6.1.6). The second is that since the pooled curve is flat by construction it extends past 100 tenants, so the real ceiling on tenants-per-footprint is the shared instance's connection and catalogue budget, not packing.

6.1.6 Data-Tier Density (H1 and H3)

The data-density sweep drove the TTK-like query mix against 30 seeded tenant schemas on one PostgreSQL instance, with pgbench running inside the database pod. Table 6.6 shows the result. Throughput plateaus near 5000 Transactions Per Second (TPS) from about 20 concurrent clients onward, then declines slightly toward the 100-connection cap. Past the knee, added clients only queue and latency rises. The instance is CPU-bound, using about three and a half of the machine's four cores across the plateau, while connections only approach the 100-connection cap at the highest client count. The per-request database cost calibrated from the unsaturated point is on the

order of 0.7 CPU-milliseconds per transaction, with writes roughly three times the cost of reads. This per-transaction cost calibrates the interpretation of tenant counts beyond the laptop's capacity.

Table 6.6: Schema-per-tenant data-tier density

Clients	Throughput (TPS)	Avg latency (ms)	DB CPU (m)	Active conns
20	5243	3.8	3615	21
50	5034	9.9	3502	51
80	4851	16.5	3552	81
95	4022	23.6	3578	96

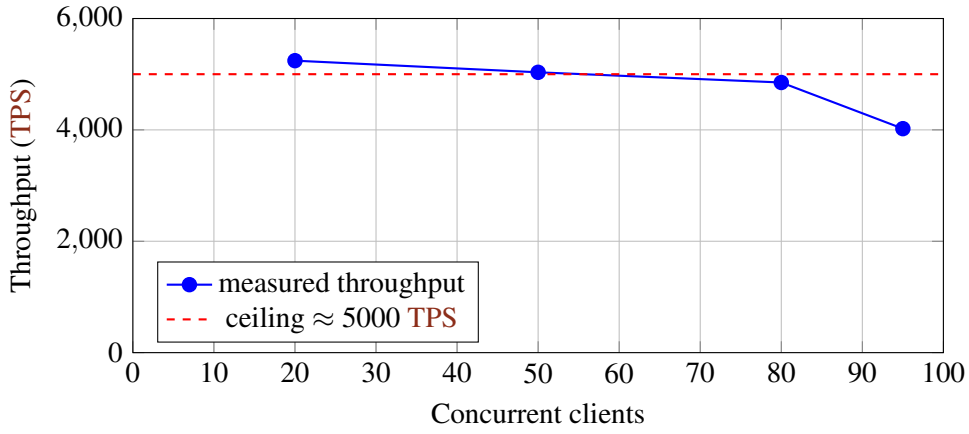


Figure 6.3: Data-tier throughput vs concurrent clients

The density margin (H1) is large. Against a small tenant's roughly 0.5 queries/s of peak demand, a 5000-TPS capacity corresponds to thousands of small tenants of pure throughput, so 30 small tenants sits well within the data tier's capacity. The binding constraint (H3) is therefore not throughput but the connection count and per-schema catalogue overhead at much higher tenant counts, which is precisely the argument for connection pooling (Section 6.1.7).

6.1.7 The Connection-Pooling Lever and Its Isolation Limit

Because the binding constraint is the shared instance's connection budget, connection pooling is the proposal's main data-tier recommendation. Its effect, and a critical correctness constraint on how it may be applied, were both measured.

Pooling decouples server connections from client fan-out. Sweeping client connections directly against PostgreSQL versus through PgBouncer in transaction mode (pool size 20) shows the failure the pooler prevents. Direct connections grow one-to-one with the client count and, past the instance's 100-connection ceiling, fail outright (connection errors at 150 clients). Through the pooler, server connections stay flat at the pool size of 20 with zero failures as clients rise to 300, multiplexing the fan-out onto a small, bounded server-connection set. This is the lever that turns the roughly 100-connection ceiling into far more tenants per instance.

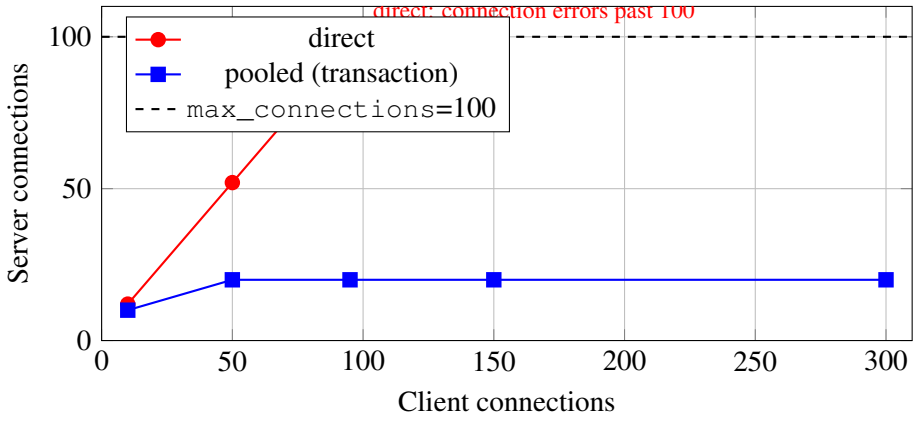


Figure 6.4: Server connections vs client fan-out: direct vs pooled

Transaction-mode pooling is unsafe for the schema-per-tenant data plane. Transaction mode rotates the server connection per transaction and does not reset session state between clients. The schema-per-tenant engines select a tenant with a session-level `SET search_path`, so under transaction mode that setting persists on the rotated server connection and is visible to the next client, which never issued it. This was demonstrated on the cluster by forcing connection reuse: a second client, on a fresh pooled connection with no `SET` of its own, read the first client's tenant data. Table 6.7 contrasts the two modes. Session mode is safe because the pooler runs `DISCARD ALL` on connection release, resetting `search_path` so the next client starts clean. The single-schema control plane (tenant-manager) is unaffected either way, because it never switches schema per tenant.

Table 6.7: Pooler mode and tenant isolation

Mode	**Second client, fresh connection, no `SET`**	**Verdict**
transaction	sees the first client's `search_path` and data	leak
session	default `search_path`, tenant table not visible	isolated

The safe mode multiplexes less, and the cost was quantified. Session mode is correct but pins a server connection to a client for the whole session, so it cannot multiplex persistent connections beyond the pool size. This was measured with persistent pgbench clients running the real data-plane query (`SET search_path TO tenant_N` followed by the ticket list) through each pooler at pool size 20, averaged over ten repetitions. Table 6.8 reports the result. In transaction mode, 80 clients multiplex onto 20 server connections with zero failures (a 4:1 ratio, bounded only by the pool size). The 39 clients waiting in any `SHOW POOLS` snapshot are transient, clearing as each short transaction returns its connection. In session mode, only the pool size of 20 clients run. Every excess client is parked waiting for a server connection, deterministically 20 waiting at 40 clients and 60 waiting at 80 clients across all repetitions. Throughput is comparable in both modes (about 2100–2250 TPS) because the 20 active server connections saturate the cheap query equally.

The difference is not throughput but *client admission*. When a blocked client's wait exceeds the pooler's query-wait timeout, it is rejected outright: a longer-hold run realised these as failures, rejecting 20 of 40 and 60 of 80 clients. Session mode therefore sheds the load it cannot pool, while transaction mode absorbs it.

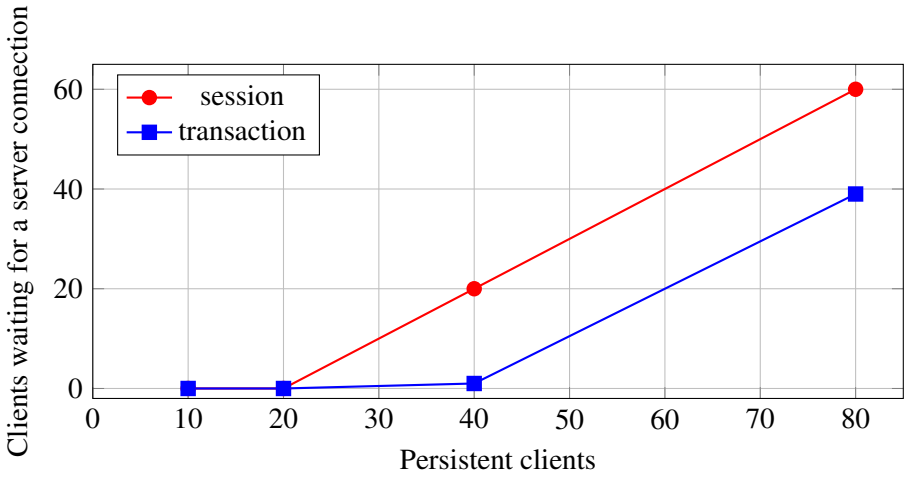


Figure 6.5: Multiplexing: session vs transaction mode

Table 6.8: Pooler multiplexing

Clients	Mode	Server conns	Waiting clients	Throughput (TPS)
40	transaction	20	1	2180
80	transaction	20	39 (transient)	2115
40	session	20	20	2253
80	session	20	60	2192

The design rule that follows bounds where the density lever applies. Transaction-mode pooling (or a managed equivalent such as AWS RDS Proxy) is safe and high-multiplexing for the *single-schema control plane*. For the *schema-per-tenant data plane*, the safe choices are session-mode pooling, which is correct but multiplexes only up to the pool size, or changing the tenant resolver to set the schema per transaction (`SET LOCAL`) or use schema-qualified access, which makes the high-density transaction mode safe there too. The last option is the stronger choice and is recommended for the data plane.

6.2 Discussion

Hypotheses. H1 (density) is confirmed and shown to be conservative on two axes: the data tier sustains thousands of small tenants of throughput, the packing axis is flat to 100 tenants, and at the application tier a single 500m pod clears the realistic 20–30-tenant read load by orders of magnitude (an 11 ms p95) with the write path bounded at 7 creates/s by the connection pool. H2 (elasticity) is confirmed but qualified, and the qualification is now measured rather than asserted:

horizontal autoscaling gives no benefit on the database-bound control plane, but on the genuinely CPU-bound TTK engine a warm fleet roughly triples throughput and restores the latency SLO with a wide margin, whereas reactive scaling alone captures only part of that gain. A purely reactive CPU-target HPA mitigates rather than fully restores latency under a step load, owing to scale-up and JVM warm-up lag. H3 (bottleneck) is confirmed four independent ways (the database-bound write path, the commit-throughput ceiling, the data-density plateau, and the per-pod connection-pool knee that bounds both the write ceiling and tenant isolation), all pointing to the shared data tier and specifically to its connection budget rather than its throughput. H4 (isolation) is *refuted* for the autoscaling-and-quota guardrails: a tenant crossing the connection-pool knee collapses the quiet tenants' SLO (41 ms to 6 s, 0% to 96% error) and a CPU-target HPA does not recover it, because the contention is pool-bound and invisible to a CPU signal. `ResourceQuota` still contains a tenant's footprint at admission, but intra-pool isolation requires connection-pool governance, not autoscaling.

Contribution. The study delivers a validated resource-sharing characterisation of the pooled, schema-per-tenant model and identifies its true scaling lever (connection pooling on the shared instance) together with a correctness constraint on applying that lever under schema-per-tenant isolation. A concrete, actionable finding emerged for the platform: the pooler-mode design rule that prevents a cross-tenant data leak under schema-per-tenant isolation.

Threats to validity. The workload is synthetic and derived from client estimates rather than production traces. The laptop scale forces the use of phantom-node extrapolation for high tenant counts, mitigated by calibrating against measured per-request costs. The shared production database is not load-tested. A local PostgreSQL instance stands in for it, so absolute throughput numbers are indicative rather than production figures, and the per-pod connection-pool knee (and therefore the absolute isolation threshold) scales with the configured pool size and the instance's connection budget rather than being a universal constant. The evaluation is single-region and single-cluster, and absolute throughput is bounded by the single-node laptop cluster, so the reproducible signals are the contrasts (A0 versus A1, cold versus warm, below versus above the pool knee) rather than the absolute rates. Cost and monetary efficiency, although correlated with the technical efficiency measured here, are deliberately out of scope.

6.3 Summary

This chapter reported and discussed the resource-efficiency study against its four hypotheses. The unifying result is that the pooled, schema-per-tenant model meets its tenant-density target with room to spare, and that what bounds its scaling is the shared data tier's connection budget rather than application CPU, so connection pooling, applied under a schema-per-tenant isolation rule, is the decisive lever. Chapter 7 draws these results together with the conclusions and the future work that follows from them.

Chapter 7

Conclusions and Future Work

This chapter revisits the problem and what was built, answers the research questions, states the main contributions, and outlines the future work that follows from the results.

7.1 Conclusions

This dissertation addressed the absence of a dedicated, unified tenant-management layer in the NOSSIS One platform as it evolves towards a multi-tenant SaaS model. The work designed and implemented the *tenant-manager*, a backoffice module within the SIGO part of NOSSIS One that automates the management of tenants and their subscriptions, users, and lifecycle.

The module is a cloud-native Quarkus service with a strict layered architecture and a PostgreSQL database, integrated with the NOSSIS IAM service for OIDC authentication and identity provisioning. It exposes a REST API that models the tenant lifecycle as a guarded state machine, manages per-product subscriptions and feature limits, provisions and manages users and product roles in the IAM service, and enforces tenant isolation through the authenticated tenant identity. The functional evaluation, an integration suite covering all 23 endpoints and every documented business rule, confirmed that the implemented flows behave as specified and that the isolation and rule-enforcement guarantees hold.

Beyond the implementation, the dissertation characterised the resource efficiency of the pooled, schema-per-tenant model that the platform adopts. The study showed that the model comfortably meets the target tenant density, that the binding constraint is the shared data tier rather than the application tier, and that connection pooling, not horizontal autoscaling, is the control plane's effective scaling lever, subject to an isolation rule on the schema-per-tenant data plane.

7.2 Research Questions Revisited

The work was guided by three research questions, restated and answered here.

Q1, on lifecycle automation. The tenant lifecycle is automated through a guarded state machine exposed over a REST API, in which onboarding, approval and rejection, configuration,

subscription management, and user provisioning are driven by single endpoint calls that orchestrate the database and the IAM service. Manual, multi-step administrative work is replaced by validated transitions, and the integration suite confirms that invalid transitions are rejected rather than silently applied.

Q2, on architectural approaches. A modular, cloud-native architecture proved well suited to a tenant-management backoffice. A strict layered backend keeps business rules, mapping, and persistence separate; delegating authentication and identity provisioning to the IAM service keeps credentials out of the module; and the pooled, schema-per-tenant model balances density against per-tenant data separation. Tenant isolation enforced through the authenticated identity provides security without a per-tenant code path.

Q3, on resource optimization and autoscaling. The resource-efficiency study answered this question empirically. Horizontal autoscaling helps only a genuinely CPU-bound tier and gives no benefit to the data-bound control plane, so it is not the platform's main lever. The effective levers are request right-sizing, admission guardrails, and connection pooling, the last of which turns the shared instance's connection ceiling into far higher tenant density, provided the schema-per-tenant data plane uses session-mode pooling or a per-transaction schema reset.

7.3 Contributions

The main contributions of this dissertation are:

- the design and implementation of the tenant-manager module, a complete tenant-management backoffice for SIGO, with an automated lifecycle, subscription and user management, and IAM integration;
- an integration test suite covering all 23 endpoints and every documented business rule, validating the functional correctness and isolation guarantees of the implemented flows;
- a validated resource-sharing characterisation of the pooled, schema-per-tenant model, identifying the shared data tier as the binding constraint and connection pooling as its true scaling lever;
- a concrete, actionable finding for the platform: a pooler-mode design rule that prevents a cross-tenant data leak under schema-per-tenant isolation.

7.4 Future Work

Several directions follow directly from the results. The asynchronous tenant-provisioning architecture described in Chapter 3 should be implemented so that long-running lifecycle operations

move off the request path. On the CPU-bound TTK engine the study found that a purely reactive CPU-target HPA pays a heavy scale-up tax, so pre-emptive scaling, such as a warm pod reserve, scheduled minimum replicas, or predictive autoscaling, should be evaluated to realise the warm-ceiling throughput without the transient latency penalty. The recommended tenant-resolver change (`SET LOCAL` per transaction or schema-qualified access) should be implemented so that the data plane can use high-density transaction-mode pooling safely, and a managed pooler such as RDS Proxy evaluated on the production database. Because the isolation experiment showed that autoscaling and quotas do not protect a tenant from a pool-bound noisy neighbour, the connection-pool governance levers it points to, namely per-tenant pool quotas, a fairness-aware pooler, or write rate-limiting, should be implemented and evaluated. A cost model would translate the technical density into monetary terms. Finally, integration with the remaining platform modules, including the Email Service and the TTK Engine, would complete the target architecture beyond the current tenant-manager scope.

References

[1] Michael Armbrust et al. “A View of Cloud Computing”. In: *Communications of the ACM* 53.4 (2010), pp. 50–58. DOI: 10.1145/1721654.1721672. URL: https://doi.org/10.1145/1721654.1721672.

[2] C. Batista et al. “Towards a Multi-Tenant Microservice Architecture: An Industrial Experience”. In: *Proc. - IEEE Annu. Comput., Softw., Appl. Conf., COMPSAC*. Proceedings - 2022 IEEE 46th Annual Computers, Software, and Applications Conference, COMPSAC 2022. Ed. by Va Leong H. et al. Journal Abbreviation: Proc. - IEEE Annu. Comput., Softw., Appl. Conf., COMPSAC. Institute of Electrical and Electronics Engineers Inc., 2022, pp. 553–562. ISBN: 978-166548810-5 (ISBN). DOI: 10.1109/COMPSAC54236.2022.00100. URL: https://www.scopus.com/inward/record.uri?eid=2-s2.0-85136983518&doi=10.1109%2FCOMPSAC54236.2022.00100&partnerID=40&md5=1d45bc7ebbbd81d0bf5301517f221bd3.

[3] Hong Cai, Ning Wang, and Ming Jun Zhou. “A Transparent Approach of Enabling SaaS Multi-Tenancy in the Cloud”. In: *2010 6th World Congress on Services*. 2010, pp. 40–47. DOI: 10.1109/SERVICES.2010.48.

[4] S.G. Haugeland et al. “Migrating Monoliths to Microservices-based Customizable Multi-tenant Cloud-native Apps”. In: *Proc. - Euromicro Conf. Softw. Eng. Adv. Appl., SEAA*. Proceedings - 2021 47th Euromicro Conference on Software Engineering and Advanced Applications, SEAA 2021. Ed. by Baldassarre M.T., Scanniello G., and Skavhaug A. Journal Abbreviation: Proc. - Euromicro Conf. Softw. Eng. Adv. Appl., SEAA. Institute of Electrical and Electronics Engineers Inc., 2021, pp. 170–177. ISBN: 978-166542705-0 (ISBN). DOI: 10.1109/SEAA53835.2021.00030. URL: https://www.scopus.com/inward/record.uri?eid=2-s2.0-85119200078&doi=10.1109%2FSEAA53835.2021.00030&partnerID=40&md5=35ad65147a4e668e3c5f74395b99318e.

[5] Roy Rosenzweig Center for History and New Media. *Zotero: Your Personal Research Assistant*. 2025. URL: https://www.zotero.org/.

[6] Ritesh Kumar. “Multi-Tenant SaaS Architectures: Design Principles and Security Considerations”. In: 5 (2020), pp. 49–62.

[7] Altice Labs. *NOSSIS One Platform*. Accessed: 2025-10-06. 2025. URL: https://www.alticelabs.com/products/nossis-one/.

[8] Y. Liu et al. “Research on Cloud-Native Monitoring System Based on Prometheus”. In: *Proc SPIE Int Soc Opt Eng*. Proceedings of SPIE - The International Society for Optical Engineering. Ed. by Wu L. and Qiu Z. Vol. 13107. Journal Abbreviation: Proc SPIE Int Soc Opt Eng. SPIE, 2024. ISBN: 0277786X (ISSN); 978-151067868-2 (ISBN). DOI: 10.1117/12.3029320. URL: https://www.scopus.com/inward/record.uri?eid=2-s2.0-85193469828&doi=10.1117%2F12.3029320&partnerID=40&md5=453812b367a1e839c701f6be75c96d35.

[9] N. Manasa and N.V. Pujari. "Kubernetes-based multitenant platform as a service". In: *Proc SPIE Int Soc Opt Eng*. Proceedings of SPIE - The International Society for Optical Engineering. Ed. by Zhang C. Vol. 13977. Journal Abbreviation: Proc SPIE Int Soc Opt Eng. SPIE, 2025. ISBN: 0277786X (ISSN); 978-151069900-7 (ISBN). DOI: 10.1117/12.3092362. URL: https://www.scopus.com/inward/record.uri?eid=2-s2.0-105026339471&doi=10.1117%2F12.3092362&partnerID=40&md5=c9b58d747b9ee15b674404295a5db61d.

[10] Themis Melissaris et al. "Elastic cloud services: scaling snowflake's control plane". In: *Proceedings of the 13th Symposium on Cloud Computing*. SoCC '22. New York, NY, USA: Association for Computing Machinery, Nov. 7, 2022, pp. 142–157. ISBN: 978-1-4503-9414-7. DOI: 10.1145/3542929.3563483. URL: https://dl.acm.org/doi/10.1145/3542929.3563483.

[11] E.T. Nordli et al. "Migrating monoliths to cloud-native microservices for customizable SaaS". In: *Information and Software Technology* 160 (2023). ISSN: 09505849 (ISSN). DOI: 10.1016/j.infsof.2023.107230. URL: https://www.scopus.com/inward/record.uri?eid=2-s2.0-85154031347&doi=10.1016%2Fj.infsof.2023.107230&partnerID=40&md5=28761a58bf7d98ec7ed4366e9858ba30.

[12] Manoel Marisergio Alves de Oliveira et al. "SPL integrated with Microservices: a hybrid architectural proposal for multitenant SaaS". In: *Proceedings of the 17th Brazilian Symposium on Software Components, Architectures, and Reuse*. SBCARS '23. New York, NY, USA: Association for Computing Machinery, 2023, pp. 1–10. ISBN: 979-8-4007-0952-4. DOI: 10.1145/3622748.3622749. URL: https://dl.acm.org/doi/10.1145/3622748.3622749.

[13] Matthew J Page et al. "The PRISMA 2020 statement: an updated guideline for reporting systematic reviews". In: *BMJ* 372 (2021). Publisher: BMJ Publishing Group Ltd _eprint: https://www.bmj.com/content/372/bmj.n71.full.pdf. DOI: 10.1136/bmj.n71. URL: https://www.bmj.com/content/372/bmj.n71.

[14] Rishi Kumar Sharma. "Multi-Tenant Architectures in Modern Cloud Computing: A Technical Deep Dive". In: *International Journal of Scientific Research in Computer Science Engineering and Information Technology* 11 (2025), pp. 307–317. DOI: 10.32628/CSEIT25111236.

[15] H. Song, P.H. Nguyen, and F. Chauvel. "Using microservices to customize multi-tenant Saas: From intrusive to non-intrusive". In: *OpenAccess Ser. Informatics*. OpenAccess Series in Informatics. Ed. by Cruz-Filipe L. et al. Vol. 78. Journal Abbreviation: OpenAccess Ser. Informatics. Schloss Dagstuhl- Leibniz-Zentrum fur Informatik GmbH, Dagstuhl Publishing, 2020. ISBN: 21906807 (ISSN); 978-395977137-5 (ISBN). DOI: 10.4230/OASIcs.Microservices.2017-2019.1. URL: https://www.scopus.com/inward/record.uri?eid=2-s2.0-85081548293&doi=10.4230%2FOASIcs.Microservices.2017-2019.1&partnerID=40&md5=288a4d5dbccfd6453641b2be6105b640

[16] E. Truyen et al. "Feasibility of container orchestration for adaptive performance isolation in multi-tenant SaaS applications". In: *Proc ACM Symp Appl Computing*. Proceedings of the ACM Symposium on Applied Computing. Journal Abbreviation: Proc ACM Symp Appl Computing. Association for Computing Machinery, 2020, pp. 162–169. ISBN: 978-145036866-7 (ISBN). DOI: 10.1145/3341105.3374034. URL: https://www.scopus.com/inward/record.uri?eid=2-s2.0-85083036112&doi=10.1145%2F3341105.3374034&partnerID=40&md5=8d89c48301702706acca037502e619f6.

[17] Eddy Truyen et al. “Towards a Container-Based Architecture for Multi-Tenant SaaS Applications”. In: *Proceedings of the 15th International Workshop on Adaptive and Reflective Middleware*. Trento, Italy: Association for Computing Machinery, 2016, pp. 1–6. DOI: 10.1145/3008167.3008173. URL: https://doi.org/10.1145/3008167.3008173.

[18] Wei-Tek Tsai et al. “Towards a Scalable and Robust Multi-Tenancy SaaS”. In: *Proceedings of the Second Asia-Pacific Symposium on Internetware*. Suzhou, China: Association for Computing Machinery, 2010, pp. 1–15. DOI: 10.1145/2020723.2020731. URL: https://doi.org/10.1145/2020723.2020731.

[19] C. Zhao, C. Yang, and R. Zhao. “The Model of Cross-Tenant Information Access Control in SAAS Cloud”. In: *Proc. - Int. Conf. Comput., Commun., Percept. Quantum Technol., CCPQT*. Proceedings - 2022 International Conference on Computing, Communication, Perception and Quantum Technology, CCPQT 2022. Journal Abbreviation: Proc. - Int. Conf. Comput., Commun., Percept. Quantum Technol., CCPQT. Institute of Electrical and Electronics Engineers Inc., 2022, pp. 174–180. ISBN: 978-166547020-9 (ISBN). DOI: 10.1109/CCPQT56151.2022.00038. URL: https://www.scopus.com/inward/record.uri?eid=2-s2.0-85143059000&doi=10.1109%2FCCPQT56151.2022.00038&partnerID=40&md5=5b8f9b0ffb1bb883de08ddf998c1bfd0.

[20] C. Zheng, Q. Zhuang, and F. Guo. “A Multi-Tenant Framework for Cloud Container Services”. In: *Proc Int Conf Distrib Comput Syst*. Proceedings - International Conference on Distributed Computing Systems. Vol. 2021-July. Journal Abbreviation: Proc Int Conf Distrib Comput Syst. Institute of Electrical and Electronics Engineers Inc., 2021, pp. 359–369. ISBN: 978-166544513-9 (ISBN). DOI: 10.1109/ICDCS51616.2021.00042. URL: https://www.scopus.com/inward/record.uri?eid=2-s2.0-85117109527&doi=10.1109%2FICDCS51616.2021.00042&partnerID=40&md5=61b628795b6756e4755b79f6a075c824.

[21] X. Zhou et al. “A Feature Tree and Dynamic QoS based Service Integration and Customization Model for Multi-tenant SaaS Application”. In: *Proc. Int. Conf. Serv. Sci., ICSS*. Proceedings of International Conference on Service Science, ICSS. Vol. 2020-August. Journal Abbreviation: Proc. Int. Conf. Serv. Sci., ICSS. Institute of Electrical and Electronics Engineers Inc., 2020, pp. 107–114. ISBN: 21653836 (ISSN); 978-172818531-6 (ISBN). DOI: 10.1109/ICSS50103.2020.00025. URL: https://www.scopus.com/inward/record.uri?eid=2-s2.0-85099433084&doi=10.1109%2FICSS50103.2020.00025&partnerID=40&md5=fe55c82ea72a4af07417784860642c05.

Appendix A

Full Endpoint Catalog

This appendix lists the complete REST API surface of the tenant-manager. All endpoints are served under the base path `/api/v1`. The authoritative contract is the OpenAPI 3.1.1 specification generated from the source.

Public Endpoints

- `POST /tenants`: submit a tenant creation request.
- `GET /products`: retrieve the global product catalog with all plans.

Tenant Management (Protected)

- `GET /tenants`: list tenants with optional `status`, `search`, `page`, and `size` filters.
- `GET /tenants/stats`: retrieve aggregate tenant counts by status.
- `GET /tenants/{tenantName}`: retrieve a tenant with its subscriptions.
- `DELETE /tenants/{tenantName}`: soft-delete the tenant.
- `POST /tenants/{tenantName}/approve`: approve a pending tenant.
- `POST /tenants/{tenantName}/reject`: reject and physically delete a pending tenant.
- `POST /tenants/{tenantName}/disable`: disable an active tenant.
- `POST /tenants/{tenantName}/enable`: enable a disabled tenant.

Settings (Protected)

- `GET /tenants/{tenantName}/settings`: retrieve tenant settings.
- `PATCH /tenants/{tenantName}/settings`: update `language` and `logoUrl`.

Subscriptions (Protected)

- `POST /tenants/{tenantName}/subscriptions`: subscribe to a product plan.
- `PATCH /tenants/{tenantName}/subscriptions/{productName}`: change the active subscription plan.
- `DELETE /tenants/{tenantName}/subscriptions/{productName}`: cancel the active subscription.
- `GET /tenants/{tenantName}/subscriptions/{productName}/usage`: retrieve aggregated usage.

Users (Protected)

- `GET /tenants/{tenantName}/users`: list users for the tenant.
- `POST /tenants/{tenantName}/users`: create a user within subscription limits.
- `GET /tenants/{tenantName}/users/{username}`: retrieve user details.
- `PATCH /tenants/{tenantName}/users/{username}`: update user attributes.
- `DELETE /tenants/{tenantName}/users/{username}`: delete a user.
- `POST /tenants/{tenantName}/users/{username}/disable`: disable a user.
- `POST /tenants/{tenantName}/users/{username}/enable`: re-enable a disabled user.

Appendix B

User Interface Screenshots

This appendix collects the tenant-manager dashboard views referenced in Section 3.8. The administrative dashboard itself is shown there (Figure 3.12).

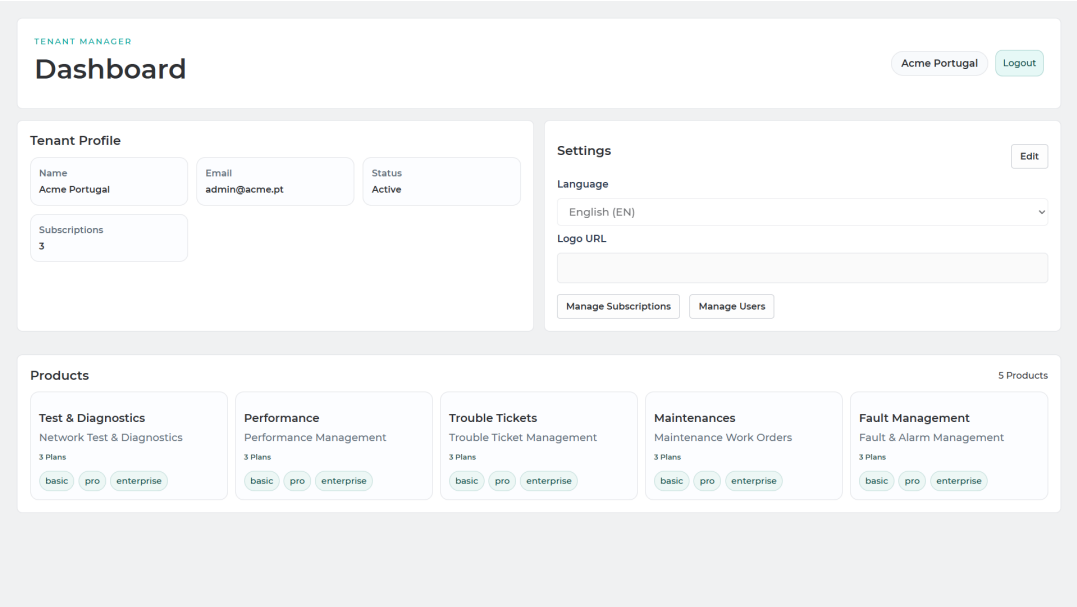


Figure B.1: Tenant detail view with subscriptions

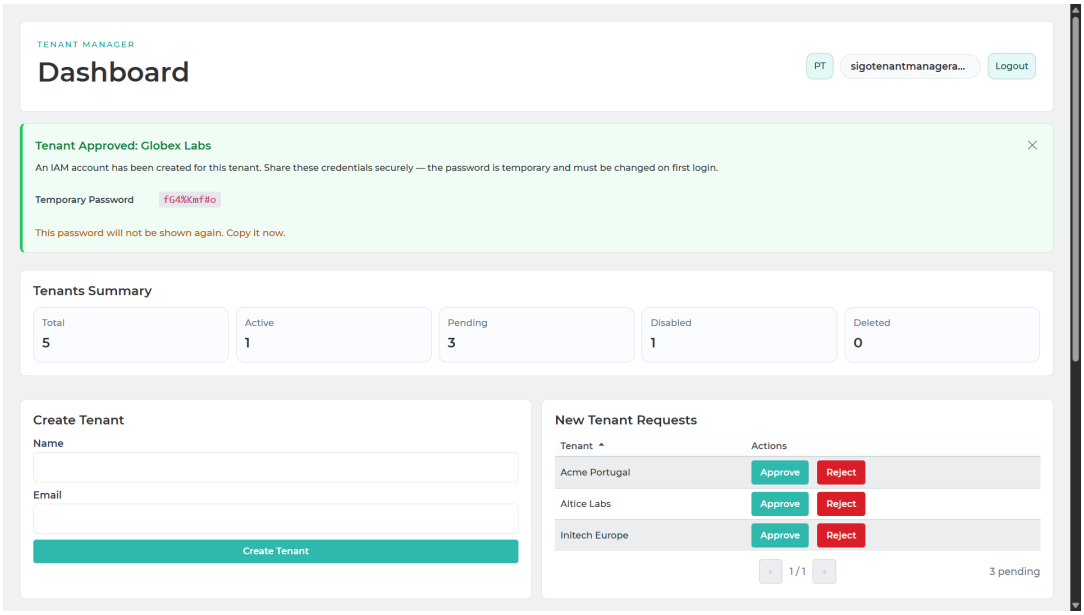


Figure B.2: Approve tenant modal with IAM credentials

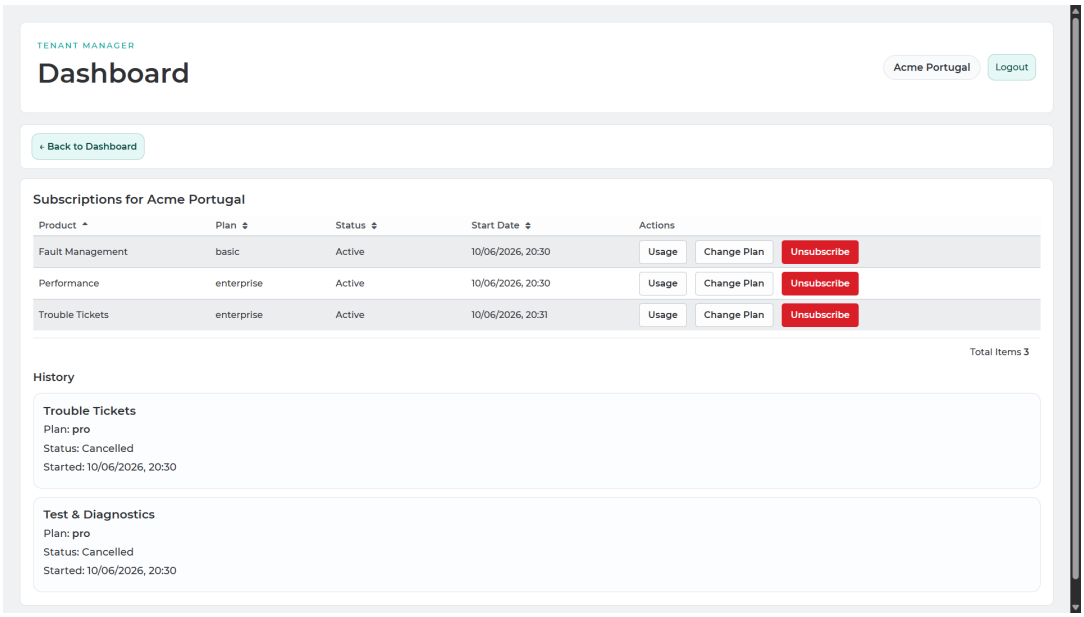


Figure B.3: Subscription management view

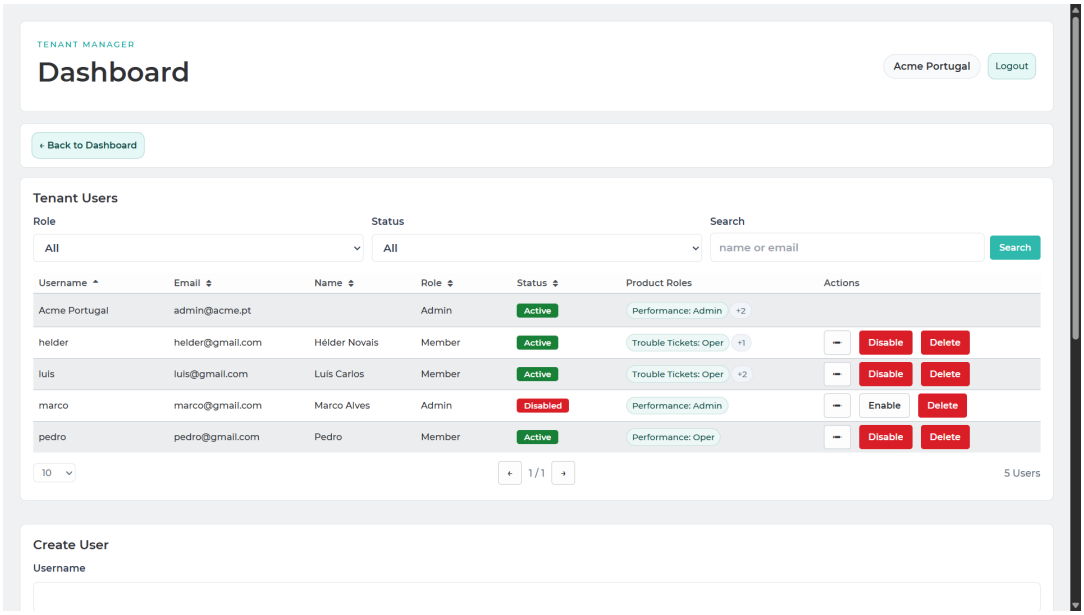


Figure B.4: User management panel

Appendix C

Use of Artificial Intelligence Tools

During the preparation of this dissertation I used Artificial Intelligence (AI)-assisted tools for language editing and proofreading, for improving the clarity and structure of the text, for formatting support (LaTeX tables and figures), and for assistance with the simulation and analysis scripts. All experimental data, measurements, and technical conclusions are my own. Every AI-assisted output was reviewed, verified, and edited by me, and I take full responsibility for the final content.