



INTERNATIONAL HELLENIC UNIVERSITY
SCHOOL OF ENGINEERING
DEPARTMENT OF INFORMATION AND ELECTRONIC ENGINEERING

DIPLOMA THESIS
**A Self-Service Orchestration Platform for Academic
Proxmox Clusters**

Student:
Thomas Soumpasis
Student ID: 2020251

Supervisor:
Antonis Sidiropoulos

06 September 2026

Title of Dissertation A Self-Service Orchestration Platform for Academic Proxmox Clusters

Code of Dissertation 25351

Student's full name Thomas Soumpasis

Supervisor's full name Antonis Sidiropoulos

Date of undertaking 03-11-2025

Date of completion 06-09-2026

We hereby affirm the authorship of this paper as well as the acknowledgement and credit of whichever assistance We received in its composition. We have, furthermore, noted the various sources from which We extracted data, ideas, visual or written material, in paraphrase or exact quotation. Moreover, we affirm the exclusive composition of this paper by myself only, for the purpose of it being a dissertation, in the Department of Information and Electronic Engineering of the I.H.U.

This paper constitutes the intellectual property of Thomas Soumpasis the student that composed it. According to the open-access policy, the author/composer offers the International Hellenic University authorisation to use the right to reproduce, borrow, publicly present and digitally distribute the paper globally, in electronic form and media of all kinds, for teaching or research purposes, voluntarily. Open access to the full text, by no means grants the right to trespass the intellectual property of the author/composer, nor does it authorise the reproduction, republication, duplication, selling, commercial use, distribution, publication, downloading, uploading, translation, modification of any kind, in part or summary of the paper, without the explicit written consent of the authors.

The approval of this dissertation by the Department of Information and Electronic Engineering of the International Hellenic University, does not necessarily entail the adoption of the author's views, on behalf of the Department.

Prolog

As the digital landscape of higher education continues to evolve, the demand for scalable, hands-on technical training is accelerating at an unprecedented pace. Academic institutions grapple with the challenge of bridging theoretical computer science curricula with practical, laboratory-driven experimentation, continuously redefining how computational resources are provisioned and governed. This thesis explores one such critical domain: the modernization of infrastructure virtualization and resource orchestration within academic computing environments.

This journey began as a quest to understand how virtualization technologies and cloud management paradigms can be reconciled with the specialized operational constraints of a university department. While enterprise cloud suites offer immense capabilities, they frequently introduce substantial architectural complexity and heavy operational footprints that render them impractical for lean departmental clusters. The motivation to delve into this topic was born out of observing the enduring friction in traditional academic laboratories—where students encounter rigid scheduling constraints and manual administrative provisioning workflows that delay active learning and experimentation.

Addressing this challenge required looking beyond conventional web development patterns and investigating how modern functional programming paradigms and concurrent runtimes can simplify hardware orchestration. By leveraging the actor-based concurrency of the Erlang Virtual Machine, the declarative modeling of the Ash Framework, and resilient asynchronous job queues, this work sought to design an orchestration gateway that provides self-service virtual machine lifecycles while enforcing strict multi-tenancy, dynamic quota boundaries, and institutional identity governance.

The following chapters embark on a comprehensive exploration of this subject, from the theoretical foundations of hypervisor virtualization and federated identity to the concrete software implementation and operational verification of the platform. Drawing on principles of concurrent systems design, declarative software engineering, and institutional access management, this thesis presents a focused, practical solution aimed at streamlining the educational computing experience. It is my hope that this work demonstrates the viability of lightweight orchestration in academia, stimulates further dialogue on educational cloud architectures, and serves as a meaningful foundation for future advancements in institutional infrastructure management.

Contents

1	Introduction	1
1.1	Context and Motivation	1
1.2	Platform Overview and Engineering Contributions	2
1.3	Structure of the Thesis	3
2	Theoretical Background and Related Work	4
2.1	Introduction	4
2.2	Hypervisor Virtualization and Infrastructure Abstraction	5
2.2.1	Virtualization as a Resource Abstraction Mechanism	5
2.2.2	Hypervisors and Virtual Machine Management	5
2.2.3	Virtualization Clusters and Resource Pools	6
2.2.4	Proxmox VE as the Target Infrastructure Environment	6
2.3	Enterprise Cloud Platforms and Lightweight Infrastructure Portals	7
2.3.1	The Cloud Management Model	7
2.3.2	OpenStack as an Enterprise Cloud Management Platform	8
2.3.3	Infrastructure Management Complexity	8
2.3.4	Lightweight Middleware Portals	9
2.4	Institutional Authentication and Federated Identity	10
2.4.1	Authentication and Authorization	10
2.4.2	Centralized and Federated Identity	11

2.4.3	OAuth 2.0	11
2.4.4	Separation of Identity and Application Authorization	12
2.5	Multi-Tenancy and Role-Based Access Control	12
2.5.1	Multi-Tenancy	12
2.5.2	Role-Based Access Control	13
2.5.3	Roles, Ownership, and Resource Context	14
2.5.4	Resource Quotas and Controlled Allocation	14
2.6	Related Work and Positioning of	15
2.6.1	Existing Infrastructure Management Approaches	15
2.6.2	The Gap Between Infrastructure Administration and Infrastructure Consumption	16
2.6.3	Positioning of the proposed solution	17
2.6.4	Research and Engineering Contribution	18
2.7	Chapter Summary	18
3	Technology Stack and Architectural Approach	20
3.1	Introduction to the Technology Chapter	20
3.1.1	Purpose and Scope of the Chapter	20
3.1.2	Platform Requirements, Evaluation Criteria, and the Selected Stack	20
3.2	Overall Architectural Approach	21
3.2.1	Core Architectural Principles	21
3.2.2	Comparative Stack Analysis and Architectural Justification	22
3.3	Backend Runtime and Concurrency Technologies	22
3.3.1	Actor-Based Concurrency and Preemptive Scheduling	22
3.3.2	Hierarchical Supervision and Fault Isolation	23
3.4	Declarative Domain Modeling and Authorization	23
3.4.1	Declarative Resource Engineering and Domain Actions	23
3.4.2	Centralized Multi-Tenant Policies and Access Control	24

3.5	Frontend and Reactive Interface Technologies	24
3.5.1	Phoenix LiveView and Stateful Process Architecture	24
3.5.2	Real-Time Event Propagation via Phoenix PubSub	24
3.6	Durable Background Processing with Oban	25
3.6.1	Transactional Job Orchestration and Retry Strategies	25
3.6.2	Queue Partitioning and Workload Isolation	25
3.7	Data Persistence and Infrastructure Integration	26
3.8	Chapter Summary	26
3.8.1	Main Reasons for the Chosen Stack	26
3.8.2	Transition to the Core Application Architecture	27
4	Core Platform Implementation & Domain Modeling	28
4.1	Introduction and Domain Architecture	28
4.1.1	Overview of the Implementation Scope	28
4.1.2	System Tiering and Control Flow Architecture	29
4.1.3	Relational Entity-Relationship Model	31
4.1.4	Separation of Application State and Hypervisor State	33
4.2	Identity, Affiliations, and Quota Enforcement	34
4.2.1	OAuth 2.0 Authorization Flow	34
4.2.2	First-User Bootstrap Hook	35
4.2.3	Dynamic Affiliation Resolution	37
4.2.4	Mathematical Model of Resource Quota Verification	37
4.3	Multi-Cluster Synchronization & Telemetry Architecture	38
4.3.1	The Cluster Discovery and State Synchronization Engine	38
4.3.2	In-Memory Telemetry Layer	40
4.3.3	Cluster Statistics Calculation	42
4.4	Catalog Domain & Distro Offering Abstraction	43

4.5	Deterministic Static Network Allocation	44
4.5.1	Subnet Constraints and Allocation Pool	44
4.5.2	Concurrency Control via PostgreSQL Advisory Locking	44
4.6	Asynchronous Provisioning Pipeline	46
4.6.1	Pipeline Execution Flow	46
4.6.2	Algorithmic Node Placement Optimization	49
4.6.3	Post-Execution Database Credential Scrubbing	49
4.7	Lifecycle Power Management	50
4.8	Limitations and Directions Suggested by the Implementation	51
4.9	Chapter Summary	51
5	Platform Operations, Workflows, and System Verification	53
5.1	Introduction and Operational Overview	53
5.1.1	Operational Paradigm and Interface Tiers	53
5.1.2	Real-Time State Synchronization and Event Delivery	54
5.2	Institutional Onboarding and Authentication Flow	54
5.3	Student and Faculty Self-Service Operations	55
5.3.1	Overview Dashboard and Quota Visualization	55
5.3.2	Virtual Machine Provisioning Wizard and Quota Validation	56
5.3.3	Asynchronous State Progression and UI Updates	57
5.3.4	Remote Instance Connectivity Verification	58
5.4	Administrative Management Workflows	59
5.4.1	Cluster Registration and Hardware Node Telemetry	59
5.4.2	Template Discovery and Distro Offering Governance	61
5.4.3	Academic Role Hierarchy and Quota Ceilings	61
5.4.4	User Directory, Role Reassignment, and Custom Quotas	63
5.4.5	Global Virtual Machine Fleet Supervision and Error Recovery	63

5.5	Background Job Orchestration and Pipeline Execution	65
5.5.1	The 7-Step Provisioning Pipeline (VmProvisionWorker)	65
5.5.2	Power Lifecycle State Dispatch (VmLifecycleWorker)	66
5.6	Qualitative System Verification and Operational Reliability	66
5.6.1	Quota Boundary Enforcement and Real-Time Guarding	66
5.6.2	Deterministic Static Network Allocation	67
5.6.3	Cloud-Init Parameter Injection and Network Reachability	67
5.6.4	Hypervisor Fault Isolation and Administrative Error Recovery	67
5.6.5	Transient Credential Sanitization	67
5.7	Chapter Summary	68
6	Conclusion	69
6.1	Summary	69
6.2	Conclusions	70
6.3	Future Extensions	71
6.3.1	Embedded In-Browser Web Console Access (noVNC and SPICE)	71
6.3.2	Multi-Subnet IPAM Integration and VLAN Pool Management	71
6.3.3	Automated Idle Resource Reclamation and Power Scheduling	71
6.3.4	Multi-Service Cloud Expansion (DBaaS and Object Storage)	71
A	Modernization of Departmental Infrastructure	73
A.1	Context and Previous State	73
A.2	Infrastructure Overhaul and Modernization	73
A.3	Storage Area Network (SAN) Architecture	74
A.3.1	Multipath I/O Configuration	74
A.3.2	iSCSI Switch Configuration	76
A.4	Proxmox Host Network Interface Design	77
A.5	Cluster Quorum Considerations	79

A.6	Core Network Services and Security	79
A.7	Backup and Disaster Recovery	79
A.8	Documentation and Operational Continuity	79
A.9	Summary	80

List of Figures

4.1	End-to-end platform architecture, tracing control flow from client LiveView sockets through the Ash domain layer, PostgreSQL and Oban orchestration, to the external Proxmox VE REST API.	30
4.2	Entity-Relationship Diagram of the platform's domain architecture, illustrating cluster, catalog, user, role, quota, and virtual machine relationships.	32
4.3	Sequence diagram of the OAuth 2.0 authentication, claim extraction, dynamic user upsert, and session establishment lifecycle.	36
4.4	Execution flowchart of the periodic cluster discovery and synchronization pipeline. .	39
4.5	Execution flowchart of the automated virtual machine provisioning pipeline, illustrating step progression, error recovery paths, and database password scrubbing.	47
5.1	Public landing page and institutional OAuth 2.0 Single Sign-On (SSO) login portal of the Department of Information and Electronic Engineering.	54
5.2	Student Self-Service Overview Dashboard, displaying active resource quota consumption bars, compute summaries, terms of service rules, and getting started guides.	55
5.3	Interactive Virtual Machine Provisioning Wizard, illustrating dynamic client-side quota headroom calculation, offering selection, and automated subnet assignment. . .	56
5.4	Real-time virtual machine card state progression in the self-service dashboard. . . .	57
5.5	Terminal session establishing an authenticated SSH connection to <code>root@10.100.97.5</code> , confirming automated Cloud-Init network configuration, hostname injection, and guest OS initialization.	58
5.6	Two-step Administrative Cluster Registration wizard.	59
5.7	Cluster Telemetry and Node Management dashboard for the "Olympus" cluster, showing aggregate compute/memory capacity, live utilization bars, per-node workload metrics, and provisioning toggles.	60

5.8	Operating system template discovery and distribution offering governance console. .	61
5.9	Academic Role Management dashboard, displaying configured role tiers, baseline quotas, default flags, and mapped OAuth claim tags.	62
5.10	Modal dialogs for defining and adjusting academic roles and baseline quota envelopes.	62
5.11	User management console, role reassignment, and granular custom quota override dialogs.	64
5.12	Global virtual machine fleet supervision and administrative state unlock error recovery modal.	64
A.1	iSCSI cabling topology: dual-server, dual-switch connectivity to the HPE MSA 2060 array.	80

List of Tables

3.1	Architectural comparison of candidate technology stacks.	22
4.1	Data schema for Identity, Role, and Quota entities.	35
4.2	Data schema for Proxmox Infrastructure Entities.	41
4.3	Catalog and Distribution Offering Data Schema.	43
4.4	Action Dispatch and State Settling Mapping for Lifecycle Power Commands.	50
5.1	Summary of Virtual Machine Lifecycle Operations and Interface Behaviors.	66
A.1	Physical-to-logical interface mapping per Proxmox node	77

Chapter 1

Introduction

1.1 Context and Motivation

In modern computer science and engineering education, hands-on laboratory experimentation is an indispensable pillar of the academic curriculum. Students and faculty members require isolated computing environments to configure operating systems, deploy software stacks, run network experiments, and manage database infrastructure. Historically, educational institutions met these operational demands through physical computer laboratories. However, maintaining dedicated physical hardware imposes severe logistical and financial burdens on university IT departments: hardware suffers from underutilization during non-peak hours, physical lab spaces restrict access to fixed schedules, and manually re-imaging desktop machines between academic semesters requires extensive labor.

Hardware-level virtualization has transformed institutional computing by enabling physical servers to host multiple isolated Virtual Machines (VMs). Type-1 hypervisors, such as open-source Proxmox Virtual Environment (PVE), allow computing departments to consolidate hardware, optimize power consumption, and provide remote infrastructure access. Despite these architectural advantages, an operational bottleneck remains: the resource provisioning pipeline. In typical university setups, students or professors requesting virtual resources must submit manual helpdesk tickets to IT administrators. The administrator must manually assess cluster capacity, create user accounts, assign network bridges, allocate CPU and RAM limits, deploy operating system images, and securely communicate credentials back to the user. This manual loop delays academic progress and scales poorly across large numbers of students.

Conversely, granting students direct access to an enterprise hypervisor dashboard like Proxmox VE introduces severe operational and security risks.

- **Absence of Academic Multi-Tenancy:** Native hypervisor panels are designed for system administrators rather than end-users, lacking fine-grained isolation mechanisms to safely separate hundreds of students.
- **Lack of Role-Based Quota Structures:** Proxmox VE does not natively enforce dynamic,

multi-tier user quotas aligned with academic standings (e.g., distinguishing between Bachelor students, Master students, and Professors)

- **Expanded Security Attack Surface:** Exposing native hypervisor management interfaces increases the risk of misconfiguration, unauthorized network packet sniffing, resource hoarding, or accidental disruption of shared compute nodes.

While enterprise cloud suites (such as OpenStack or VMware vCloud Director) solve some multi-tenancy challenges, they are resource-heavy, require dedicated administrative expertise, and present prohibitive operational complexity for small-to-medium university departments operating lean hypervisor clusters. Therefore, a clear practical and academic need exists for a custom, lightweight, secure, and automated web portal that acts as a middleware abstraction layer over university Proxmox VE clusters.

1.2 Platform Overview and Engineering Contributions

To solve these provisioning bottlenecks, this thesis presents the complete design, architectural modeling, and verification of a lightweight, reactive, multi-tenant web platform engineered specifically for academic Proxmox VE cluster orchestration.

The platform will function as an automated broker between institutional users and underlying hypervisor clusters. It will provide a streamlined self-service portal where students and professors could provision, monitor, power-cycle, and delete virtual compute instances within strict administrative boundaries, while providing IT administrators with full oversight over nodes, roles, quotas and offerings.

The core contributions of the platform include:

- **Institutional OAuth Integration:** Direct integration with the university's central OAuth 2.0 single sign-on system, restricting access to verified academic personnel and automatically extracting user identity metadata.
- **Dynamic Affiliation & Quota Engine:** An automated mapping engine parses institutional affiliation claims upon user login, assigning users to predefined academic roles (e.g., Bachelor Student, Master Student, Professor) and binding them to granular hardware quotas (max vCPUs, max RAM, max storage, and max number of VMs).
- **Multi-Cluster Hardware Orchestration:** Administrative mechanisms to connect multiple Proxmox VE clusters, dynamically discover cluster nodes and templates, selectively enable or disable individual nodes for provisioning, and cache node telemetry in memory.
- **Resilient Asynchronous Provisioning Pipeline:** A multi-step background provisioning pipeline managed by Oban worker queues that handles load-balanced node placement, template cloning, disk expansion, Cloud-Init configuration, power state transitions, and in-memory credential scrubbing from database logs.

- **Reactive User Interface:** A real-time web portal built with Phoenix LiveView that updates VM power states, cluster telemetry, and quota consumption bars over WebSockets without full browser reloads or client-side JavaScript Single Page Application (SPA) complexity.

1.3 Structure of the Thesis

The remainder of this diploma thesis is organized into five subsequent chapters, structured as follows:

Chapter 2: Theoretical Background & Related Work establishes the theoretical foundation for the project. It examines hypervisor virtualization, compares enterprise cloud platforms against lightweight middleware portals, reviews institutional authentication protocols, and explores multi-tenant Role-Based Access Control (RBAC).

Chapter 3: Technology Stack and Architectural Approach justifies the technology selections. It details the Erlang Virtual Machine (BEAM) concurrency model, functional programming with Elixir, declarative domain modeling via the Ash Framework, real-time web interfaces with Phoenix LiveView, and background job processing using Oban.

Chapter 4: Core platform architecture & domain modeling provides a technical dissection of the platform's data structures and domain logic. It models the core Ash resources (User, Role, Quota, Cluster, Node, Template, Offering, VirtualMachine), details the OAuth onboarding flow, and explains the multi-cluster synchronization engine

Chapter 5: Platform Operations, Workflows, and System Verification presents the operational workflows for administrators and regular users. It walks through the 7-step VM provisioning pipeline, explains job queue isolation and fail-safe state machines, and provides qualitative verification of system safety and reliability

Chapter 6: Summary, Conclusions, and Future Extensions summarizes the engineering outcomes, reflects on the architectural trade-offs of the Elixir/Ash stack, and outlines prospective future extensions, such as embedded web console access (noVNC), dynamic volume expansion, and software-defined virtual networking.

Chapter 2

Theoretical Background and Related Work

2.1 Introduction

The development of a web-based infrastructure management platform requires an understanding of several related areas of computing. At its foundation, the platform interacts with virtualized infrastructure and must therefore operate within the architectural model established by hypervisor-based virtualization. At a higher level, the platform addresses a management problem that is also addressed by large-scale cloud computing systems, although the requirements of a university environment differ significantly from those of a commercial cloud provider.

In addition to infrastructure management, the platform must integrate institutional users into a controlled environment in which authentication, authorization, and resource ownership are clearly defined. Students and professors may require different levels of access, while the resources allocated to individual users must remain isolated from those belonging to other users.

This chapter establishes the theoretical and technical background necessary to position the proposed platform within this problem space. Section 2.2 introduces hypervisor-based virtualization and the abstraction of physical computing resources. Section 2.3 examines enterprise cloud platforms and contrasts them with lightweight middleware portals. Section 2.4 discusses institutional authentication and federated identity protocols. Section 2.5 examines multi-tenancy, role-based access control, and resource authorization. Finally, Section 2.6 discusses related approaches and positions the proposed platform within the existing landscape.

The purpose of this chapter is not to provide a detailed implementation description of the platform. Instead, it establishes the problem context and identifies the architectural requirements that motivate the technology and design decisions discussed in the following chapters.

2.2 Hypervisor Virtualization and Infrastructure Abstraction

2.2.1 Virtualization as a Resource Abstraction Mechanism

Virtualization is a technique that allows physical computing resources to be abstracted and divided into multiple isolated execution environments. Rather than assigning an entire physical server to a single operating system or application, a virtualization layer enables multiple virtual machines to share the underlying physical resources. [1]

A virtual machine provides an isolated software-defined representation of a computing system. Each virtual machine can run its own operating system and applications while operating independently from other virtual machines hosted on the same physical infrastructure. The virtualization layer is responsible for allocating physical resources and maintaining the separation between guest environments.

This abstraction provides several advantages. Physical resources can be consolidated, reducing the number of dedicated servers required for individual workloads. Virtual machines can also be created, configured, started, stopped, migrated, and removed through software-defined operations. As a result, computing infrastructure becomes more flexible than an environment in which each workload is permanently associated with a specific physical machine.

For institutional environments, virtualization is particularly useful because infrastructure resources can be allocated dynamically to different users and workloads. A university may operate a shared pool of physical servers while providing students, researchers, and professors with independent virtual environments for coursework, experimentation, research, or software development.

2.2.2 Hypervisors and Virtual Machine Management

The software responsible for creating and managing virtual machines is commonly referred to as a hypervisor. The hypervisor mediates access between virtual machines and the underlying physical hardware.

Hypervisors are commonly categorized into two general types.

Type 1 Hypervisors. A Type 1 hypervisor executes directly on the underlying hardware or forms part of a dedicated virtualization environment. Examples include systems based on KVM, VMware ESXi, and Microsoft Hyper-V. [1] This approach is commonly used in server and data center environments because the virtualization layer operates independently of a conventional desktop operating system.

Type 2 Hypervisors. A Type 2 hypervisor executes as an application on top of a host operating system. Examples include Oracle VirtualBox and VMware Workstation. These systems are commonly used for development, testing, and desktop virtualization.

The distinction between these categories is important because the proposed platform operates within an infrastructure environment based on server virtualization. The objective is not to provide local desktop virtualization but to manage shared computing resources hosted across one or more infrastructure nodes.

Modern hypervisor environments also extend beyond the management of individual virtual machines. A cluster may consist of multiple physical nodes that collectively provide computing, networking, and storage resources. Management software can coordinate workloads across these nodes, provide high-availability mechanisms, and expose administrative interfaces through web dashboards, command-line tools, or programmatic APIs.

2.2.3 Virtualization Clusters and Resource Pools

A virtualization cluster consists of multiple physical nodes managed as a coordinated infrastructure environment. Individual nodes contribute resources such as processor capacity, memory, storage, and network connectivity. Virtual machines may be distributed across these nodes according to administrative policies, resource availability, or workload requirements.

This model introduces an important distinction between the physical infrastructure and the management interface exposed to users. A user requesting a virtual machine does not necessarily need to know which physical node will host that machine. Similarly, users generally should not require direct access to cluster-level administrative functionality in order to perform routine operations.

The management layer can therefore provide an abstraction above the underlying infrastructure. Users interact with logical concepts such as virtual machines, templates, available resource offerings, and quotas, while the platform is responsible for translating these requests into operations performed against the underlying virtualization environment.

This separation is particularly relevant to the proposed solution. The platform does not replace the underlying hypervisor or attempt to implement virtualization functionality independently. Instead, it operates as an intermediary management layer that simplifies selected infrastructure operations and exposes them through a controlled user-facing portal.

2.2.4 Proxmox VE as the Target Infrastructure Environment

Proxmox Virtual Environment is an open-source virtualization management platform that supports virtual machines and containers within a clustered infrastructure environment. [2] It provides web-based administration, command-line tools, clustering functionality, and a REST-based API that allows external applications to interact programmatically with the virtualization environment. [3]

The availability of a programmatic API is particularly important for external management platforms. Operations such as creating, configuring, starting, stopping, and monitoring virtual machines can be initiated through API requests rather than requiring direct interaction with the native administrative interface.

However, access to a hypervisor API alone does not solve the problem of user-oriented infrastructure management. A virtualization platform is generally designed to expose a broad range of administrative functionality. In a university environment, providing every user with direct access to the complete management interface may introduce unnecessary complexity and create authorization challenges.

The role of the proposed platform is therefore positioned above the hypervisor layer. Proxmox VE remains responsible for the execution and management of the underlying infrastructure, while the platform provides an additional layer responsible for user interaction, institutional authentication, authorization, quota enforcement, and simplified infrastructure workflows.

2.3 Enterprise Cloud Platforms and Lightweight Infrastructure Portals

2.3.1 The Cloud Management Model

Cloud computing platforms extend the principles of virtualization by providing mechanisms for managing infrastructure resources through standardized services and interfaces. Rather than interacting directly with individual hypervisor nodes, users interact with higher-level abstractions representing compute, storage, networking, identity, and resource allocation.

Enterprise cloud platforms frequently provide extensive functionality, including:

- virtual machine provisioning;
- image and template management;
- software-defined networking;
- block and object storage;
- identity and access management;
- multi-tenant resource isolation;
- quotas and usage limits;
- orchestration and automation;
- monitoring and administrative interfaces.

These capabilities are appropriate for large and complex infrastructure environments. However, the operational requirements associated with such platforms may exceed the needs of smaller institutional deployments.

The distinction between a full cloud platform and a lightweight infrastructure portal is therefore important. The objective of the proposed solution is not to reproduce every component of a cloud computing environment. Instead, the objective is to provide a focused management layer for a specific set of infrastructure operations.

2.3.2 OpenStack as an Enterprise Cloud Management Platform

OpenStack represents a prominent example of a modular cloud computing platform.[4] Its architecture consists of multiple services responsible for different infrastructure concerns, including compute, networking, storage, image management, and identity.

The OpenStack Horizon project provides a web-based dashboard through which users can interact with cloud services.[5] Depending on the deployed services, users can manage instances, images, storage, networking, users, roles, and quotas.

OpenStack also includes a dedicated identity component called Keystone, which manages authentication, authorization-related assignments, users, groups, projects, domains, and tokens.[6]

This architecture provides substantial flexibility and supports large-scale multi-tenant environments. However, this flexibility is achieved through a collection of specialized services that must be deployed, configured, integrated, and maintained.

For a university environment that already operates a functioning virtualization cluster, deploying a complete cloud platform may not always be necessary. If the primary requirement is to provide institutional users with controlled access to a limited set of virtual machine operations, a lighter management layer may provide a more focused solution.

This does not imply that enterprise cloud platforms are unsuitable. Rather, the appropriate architecture depends on the scope of the problem being addressed.

2.3.3 Infrastructure Management Complexity

The complexity of an infrastructure management solution generally increases with the number of services and responsibilities that it attempts to provide.

A complete cloud management platform may include separate components for:

- identity management;
- compute scheduling;
- networking;
- storage;

- image management;
- orchestration;
- monitoring;
- message brokering;
- service discovery;
- web-based administration.

These components can provide significant flexibility and scalability. However, they also introduce operational dependencies and configuration requirements.

A smaller institution may already possess several of these capabilities through existing systems. For example, the institution may already operate an identity provider, a virtualization cluster, network infrastructure, and centralized administrative processes.

In such an environment, the requirement may not be to replace the existing infrastructure with a new cloud platform. Instead, the requirement may be to integrate existing components into a more accessible workflow.

This creates a role for lightweight middleware.

2.3.4 Lightweight Middleware Portals

A middleware portal operates between users and an existing infrastructure system. Instead of implementing the underlying virtualization, storage, or networking functionality, the portal communicates with the infrastructure through its available interfaces.

The middleware layer can perform several functions:

- authenticate users through institutional identity systems;
- translate user requests into infrastructure API operations;
- restrict the operations available to different users;
- enforce resource quotas and ownership boundaries;
- simplify infrastructure terminology and workflows;
- provide a unified interface across multiple infrastructure clusters;
- present real-time feedback regarding asynchronous operations.

This model is particularly relevant when the underlying infrastructure already provides reliable virtualization functionality but is not intended to be exposed directly to a broad population of non-administrative users.

The middleware approach therefore creates a separation between *infrastructure administration* and *infrastructure consumption*. Hypervisor administrators retain control over the physical and cluster-level environment, while regular users interact with a restricted interface designed around their actual requirements.

The proposed solution follows this architectural direction. It does not attempt to replace Proxmox VE or reproduce the complete functionality of a cloud computing platform. Instead, it provides an application-specific control layer focused on institutional users and controlled virtual machine provisioning.

2.4 Institutional Authentication and Federated Identity

2.4.1 Authentication and Authorization

Authentication and authorization represent separate security concerns.

Authentication answers the question:

Who is the user?

Authorization answers the question:

What is the authenticated user permitted to do?

This distinction is particularly important in institutional environments. A university may already maintain an authoritative identity system containing student and employee accounts. Creating an additional set of local credentials within every institutional application introduces duplicated identity data and additional administrative responsibilities.

A more suitable approach is often to delegate authentication to an existing identity provider while allowing the application itself to define authorization rules that are specific to its domain.

For example, a university identity provider may confirm that a user is an authenticated member of the institution. The platform can subsequently determine whether that user is permitted to provision virtual machines, which clusters they may access, and which resource limits apply to their account.

2.4.2 Centralized and Federated Identity

Centralized identity management allows multiple applications to rely on a common source of identity. Users authenticate through an identity provider rather than maintaining separate credentials for every individual application.

Federated identity extends this concept by allowing an application to trust authentication performed by another system. This approach supports single sign-on, reduces password duplication, and allows institutions to maintain identity information independently from individual services.

For a university environment, this model is useful because users may already authenticate through institutional systems. The infrastructure portal can therefore integrate with the existing authentication environment rather than requiring students and professors to create a separate account.

However, authentication alone is insufficient for infrastructure management. The fact that a user is an authenticated university member does not necessarily imply unrestricted access to all virtual machines or infrastructure resources. Application-level authorization is therefore required after the user's identity has been established.

2.4.3 OAuth 2.0

OAuth 2.0 is an authorization framework that enables applications to obtain controlled access to protected resources without requiring users to provide their credentials directly to the application. Instead of handling user credentials, the application interacts with an authorization server to obtain an access token that can be used to access authorized resources. [7]

In a typical OAuth 2.0 flow, the user is redirected to an authorization server, where they authenticate and grant the requested permissions. Following successful authorization, the application receives an authorization response and can exchange it for an access token. This token is subsequently used to access the resources for which the application has been granted permission.

This model is suitable for institutional applications because authentication and credential management can remain under the responsibility of the institution's existing authorization infrastructure. The application does not need to directly handle or store the user's institutional password, while access to protected resources can be limited according to the permissions associated with the issued token.

In the proposed solution, OAuth 2.0 provides the mechanism through which institutional users can authorize the application to access permitted resources. The resulting authorization context is then used by the platform to associate the external user with an internal user representation and apply locally defined roles, quotas, and infrastructure access rules.

2.4.4 Separation of Identity and Application Authorization

A central principle of the proposed architecture is that authentication and authorization should not be treated as identical concerns.

The institutional identity provider establishes that a user has successfully authenticated. However, the application may require additional information that is specific to the infrastructure domain.

For example, the platform may define whether a user:

- may provision a virtual machine;
- may manage only resources that they own;
- may access a particular cluster;
- may exceed a standard resource allocation;
- has administrative infrastructure privileges.

These decisions are application-specific and should therefore be managed within the platform's authorization model rather than being entirely delegated to the external identity provider.

This separation also allows the institution to modify the internal roles or quotas associated with an authenticated user without changing the underlying institutional identity.

2.5 Multi-Tenancy and Role-Based Access Control

2.5.1 Multi-Tenancy

Multi-tenancy refers to an architectural model in which multiple users, groups, or organizations share an application or infrastructure environment while maintaining logical separation between their resources and data.

The concept of a tenant depends on the system being modeled. In a commercial cloud environment, a tenant may represent an organization or customer. Within a university environment, the concept may instead represent individual users, departments, courses, research groups, or other institutional units.

Regardless of the exact definition, multi-tenancy introduces an isolation requirement. A user should not automatically gain access to resources belonging to another user simply because both resources are managed by the same application or hosted on the same physical infrastructure.

For the proposed solution, the primary ownership relationship exists between users and the virtual machines or resources allocated to them. The application must therefore maintain clear associations between authenticated users and the infrastructure resources that they are permitted to access.

Multi-tenancy must be considered across several layers:

- **Identity isolation:** distinguishing authenticated users;
- **Data isolation:** restricting access to application records;
- **Resource isolation:** restricting access to virtual machines and infrastructure operations;
- **Administrative isolation:** limiting privileged operations to authorized administrators;
- **Quota isolation:** ensuring that one user's resource consumption does not exceed the allocation assigned to that user or tenant.

A secure multi-tenant architecture therefore requires more than simply displaying different user interfaces. Authorization must be enforced wherever protected resources or actions are accessed.

2.5.2 Role-Based Access Control

Role-Based Access Control (RBAC) is an authorization model in which permissions are associated with roles and users obtain permissions through their assigned roles.

The basic RBAC model can be represented through three relationships:

1. Users are assigned one or more roles.
2. Roles are associated with permissions.
3. Permissions determine which operations may be performed on protected resources.

For example, an infrastructure portal may define roles such as:

- **Student:** may provision and manage virtual machines assigned to their account within defined quotas;
- **Professor:** may receive a different set of resource limits or permissions;
- **Administrator:** may manage users, infrastructure configuration, clusters, templates, and resource offerings.

The main advantage of RBAC is that permissions can be assigned through roles rather than being individually configured for every user. This reduces administrative complexity and provides a structured representation of authorization responsibilities.

RBAC is particularly useful in institutional environments because users can be grouped according to their functional responsibilities rather than requiring a unique permission set for each individual.

2.5.3 Roles, Ownership, and Resource Context

Although RBAC provides an effective foundation for authorization, a role alone is often insufficient to determine whether an operation should be permitted.

Consider two users who both possess the same student role. Both may be authorized to start or stop virtual machines. However, this does not imply that either user should be allowed to manage every virtual machine belonging to another student.

Authorization therefore requires contextual information about the resource being accessed.

A simplified authorization rule can be expressed as:

$$\text{Permit}(u, a, r) = \text{RoleAllowed}(u, a) \wedge \text{ResourceAccessible}(u, r) \quad (2.1)$$

where u represents the user, a represents the requested action, and r represents the target resource.

The first condition determines whether the user's role permits the requested action. The second condition determines whether the specific resource is accessible to that user.

This distinction is important for the platform. A student may be authorized to perform a lifecycle operation such as starting a virtual machine, but only when the target virtual machine belongs to that student or has otherwise been explicitly assigned to them.

Consequently, the authorization model combines role-based permissions with ownership and resource-level constraints.

2.5.4 Resource Quotas and Controlled Allocation

Authorization determines whether a user may perform an operation. Resource quotas determine the amount of infrastructure capacity that a user is permitted to consume.

A quota may define limits on resources such as:

- the number of virtual machines;
- processor cores;
- memory;
- storage capacity;
- access to particular infrastructure offerings.

Quotas are particularly important in a shared university environment because the underlying infrastructure represents a finite resource pool.

Without controlled allocation, a small number of users could consume a disproportionate amount of available computing capacity. Quotas provide a mechanism for applying institutional policies to infrastructure consumption.

The quota model can be represented conceptually as:

$$\text{Usage}_{u,r} + \text{Requested}_{u,r} \leq \text{Quota}_{u,r} \quad (2.2)$$

where $\text{Usage}_{u,r}$ represents the current consumption of resource r by user u , $\text{Requested}_{u,r}$ represents the additional requested allocation, and $\text{Quota}_{u,r}$ represents the maximum allocation permitted for that user.

A provisioning request should only proceed when the requested allocation remains within the applicable resource limits.

The combination of authentication, RBAC, ownership constraints, and quotas therefore provides several layers of control. Authentication establishes the user's identity. RBAC determines which categories of actions the user may perform. Ownership and resource policies determine which specific resources may be accessed. Quotas limit the amount of infrastructure capacity that may be allocated.

2.6 Related Work and Positioning of

2.6.1 Existing Infrastructure Management Approaches

Existing approaches to infrastructure management generally occupy different positions on a spectrum.

At one end are direct hypervisor management interfaces. These provide administrators with extensive control over infrastructure resources and expose functionality related to virtual machines, storage, networking, clustering, and system administration.

At the other end are complete cloud management platforms. These provide broader abstractions and may include identity management, tenant isolation, networking, storage, orchestration, quotas, and self-service provisioning.

Between these approaches exists an intermediate category of application-specific infrastructure portals. These systems do not replace the underlying hypervisor or attempt to implement a complete cloud computing environment. Instead, they expose a selected set of infrastructure capabilities through a simplified application interface.

The choice between these approaches depends on the requirements of the organization.

A full cloud platform may be appropriate when an institution requires extensive infrastructure services, complex tenant structures, software-defined networking, distributed storage, or large-scale self-service infrastructure.

Direct hypervisor access may be appropriate for experienced system administrators who require detailed control over the infrastructure.

However, neither approach necessarily provides the optimal user experience for a broad university population consisting of students, professors and researchers whose primary requirement is to request and manage a limited number of virtual machines.

2.6.2 The Gap Between Infrastructure Administration and Infrastructure Consumption

A central observation motivating the development of the proposed solution is the difference between the requirements of infrastructure administrators and infrastructure consumers.

An administrator may require access to:

- cluster configuration;
- physical node management;
- storage configuration;
- network configuration;
- template management;
- backup systems;
- high-availability configuration;
- infrastructure-wide monitoring.

A student requesting a development environment generally requires a substantially smaller set of capabilities. Their primary workflow may consist of:

1. authenticating through their institutional identity;
2. selecting an available virtual machine configuration;
3. requesting a virtual machine;
4. monitoring the provisioning process;
5. starting, stopping, or accessing the resulting machine;
6. managing only the resources assigned to their account.

Providing a simplified workflow does not eliminate the complexity of the underlying infrastructure. Instead, the complexity is managed within an intermediary application that applies institutional policies before interacting with the hypervisor environment.

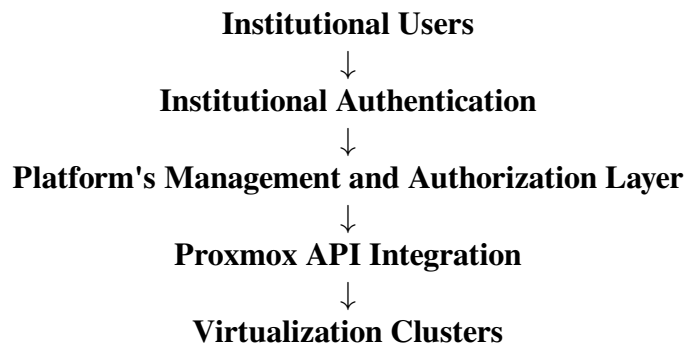
This distinction forms the primary conceptual basis for the proposed solution.

2.6.3 Positioning of the proposed solution

The proposed solution is positioned as a lightweight infrastructure management portal located between institutional users and existing Proxmox VE clusters.

The platform does not replace the underlying hypervisor and does not attempt to provide a complete cloud computing environment. Instead, it combines selected concepts from cloud management systems with the existing capabilities of the underlying virtualization infrastructure.

Its conceptual architecture can be summarized as:



The proposed solution acts as an intermediary layer responsible for translating user-oriented requests into controlled infrastructure operations. It introduces an application-specific domain model representing users, roles, quotas, infrastructure clusters, templates, offerings, and virtual machines.

This positioning differentiates the proposed platform from both a direct hypervisor administration interface and a comprehensive cloud management platform.

Compared with direct infrastructure administration, the proposed solution restricts the set of available operations while introducing institution-specific mechanisms for authentication, authorization, resource ownership, and quota enforcement.

In contrast to a complete cloud platform, the proposed solution deliberately maintains a limited scope. It builds upon an existing virtualization environment rather than attempting to replace it with an independent ecosystem for compute, networking, storage, and identity management.

The resulting architecture is intended to provide a practical balance between infrastructure capability and operational complexity.

2.6.4 Research and Engineering Contribution

The primary contribution of the project is therefore not the creation of a new hypervisor or a replacement for existing cloud platforms. The contribution lies in the design and implementation of a focused management layer that integrates existing virtualization infrastructure with institutional identity and application-specific multi-tenant authorization.

The project investigates how a modern web platform can provide:

- simplified self-service virtual machine provisioning;
- integration with existing Proxmox VE clusters;
- institutional authentication;
- role-based and resource-level authorization;
- controlled infrastructure quotas;
- real-time feedback for asynchronous infrastructure operations;
- centralized management of multiple infrastructure resources.

These requirements establish the problem context for the architectural decisions presented in the following chapter.

2.7 Chapter Summary

This chapter established the theoretical and technical background for the development of the proposed solution.

Hypervisor-based virtualization provides the foundation for abstracting physical computing resources into independently managed virtual machines. Clustered virtualization environments extend this model by combining multiple physical nodes into a coordinated infrastructure platform. Proxmox VE provides the underlying virtualization environment and exposes programmatic interfaces that allow external applications to interact with infrastructure resources.

The chapter also examined the distinction between complete cloud management platforms and lightweight infrastructure portals. Enterprise cloud systems provide extensive functionality but may introduce a level of operational complexity that is unnecessary when an organization already operates an established virtualization environment and requires only a focused self-service interface.

Institutional authentication provides a mechanism for integrating existing user identities into the platform, while the separation between authentication and authorization allows infrastructure-specific access policies to remain under application control. Multi-tenancy requires users and their resources to

remain logically separated, while RBAC, ownership constraints, and resource quotas provide complementary mechanisms for controlling access and infrastructure consumption.

These concepts position the proposed solution as a lightweight management layer between institutional users and existing Proxmox VE clusters. The platform simplifies selected infrastructure workflows without attempting to replace either the institutional identity system or the underlying virtualization environment.

Having established the problem domain and theoretical foundation, Chapter 3 examines the technology stack and architectural approach selected to implement the platform. The following chapter explains why Elixir, the BEAM runtime, the Ash Framework, Phoenix LiveView, Oban, and PostgreSQL were selected to address the requirements identified throughout this chapter.

Chapter 3

Technology Stack and Architectural Approach

3.1 Introduction to the Technology Chapter

3.1.1 Purpose and Scope of the Chapter

This chapter establishes the technological and architectural foundation of the proposed multi-tenant infrastructure management system. Its purpose is to detail the technical reasoning behind the selected technology stack and to demonstrate how the chosen tools directly address the functional and non-functional requirements introduced by the system's role as an intermediary between institutional users and external Proxmox VE hypervisor clusters.

The scope of this chapter encompasses the technologies responsible for application execution, declarative domain modeling, access control and policy enforcement, reactive user interfaces, durable background orchestration, data persistence, and external infrastructure integration.

3.1.2 Platform Requirements, Evaluation Criteria, and the Selected Stack

The proposed platform functions as an intermediary management portal between institutional users (students, educators, and administrators) and external Proxmox VE clusters. Rather than exposing underlying hypervisor APIs directly, the system provides a controlled portal through which users request and manage virtual machines within strict quotas and authorization policies.

To evaluate candidate technologies, five core criteria were established based on system requirements:

1. **Non-Blocking Asynchronous Concurrency:** Infrastructure lifecycle tasks (cloning, disk allocation, provisioning) require extended execution times. The architecture must support high concurrency without blocking incoming HTTP requests.

2. **Fault Isolation and Resilience:** External network disruptions or hypervisor API timeouts must be isolated to the affected worker process without degrading the broader application.
3. **Declarative Authorization and Policy Enforcement:** Multi-tenant boundaries and role-based access rules must be centrally defined and enforced consistently across interfaces, APIs, and background workers.
4. **Low-Latency Client Interactivity:** State transitions of long-running operations must be reflected in real time without manual page reloads or excessive client-side state overhead.
5. **Minimal Operational Footprint:** The system must remain practical to deploy and maintain on institutional hardware, favoring consolidated deployment units over complex distributed service meshes.

Guided by these criteria, the platform is developed as a unified full-stack system built on the Erlang VM (BEAM) ecosystem using **Elixir** and the **Phoenix Framework**. Real-time, reactive user interfaces are delivered via **Phoenix LiveView**, eliminating the architectural duplication of a standalone client-side Single-Page Application (SPA). Declarative domain logic, access policies, and data mappings are managed through the **Ash Framework** alongside **AshPostgres** and **Ecto**. Durable background orchestration and infrastructure job queues are powered by **Oban**, backed by a transactional **PostgreSQL** relational database. Direct communication with hypervisor nodes is handled via the **Proxmox VE REST API**.

3.2 Overall Architectural Approach

3.2.1 Core Architectural Principles

The platform follows a unified full-stack application model where a single codebase handles domain modeling, authorization, background processing, routing, and interface rendering. This avoids the fragmentation and maintenance overhead of maintaining decoupled frontend and backend services while providing a centralized location for cross-cutting security and tenant isolation rules.

Rather than relying on a traditional client-side SPA that consumes an external REST or GraphQL API, the platform utilizes a server-driven reactive architecture. Stateful server processes maintain authoritative client state and stream minimal visual diffs over persistent WebSockets. This paradigm combines modern interface responsiveness with server-authoritative security.

A further architectural principle is the strict separation between synchronous user request-response lifecycles and asynchronous infrastructure execution. Ephemeral user interactions create durable job records, which are processed independently by dedicated workers, ensuring that slow hypervisor operations never block client workflows.

3.2.2 Comparative Stack Analysis and Architectural Justification

During the architectural design phase, three distinct technology stacks were evaluated against the project's criteria. Table 3.1 summarizes the comparative findings.

Table 3.1: Architectural comparison of candidate technology stacks.

Evaluation Criterion	Django / Celery / React	Node.js / BullMQ / Vue.js	Elixir / Phoenix / Ash / Oban
Concurrency Approach	Worker and OS-process concurrency	Event-loop-based asynchronous I/O	Lightweight preemptive BEAM processes
Fault Isolation	Application and worker-dependent	Application and process-dependent	Granular process isolation and supervision
Client Architecture	Decoupled SPA	Decoupled SPA	Server-driven reactive interface
Background Processing	External broker and worker infrastructure	External Redis-backed queue system	PostgreSQL backed transactional jobs
Authorization Approach	Framework and view-level enforcement	Route middleware and application logic	Declarative resource policies via Ash
Operational Footprint	High (multiple services and brokers)	High (separate frontend, backend, Redis)	Minimal (consolidated app and PostgreSQL)

While alternative Python and Node.js based stacks offer mature ecosystems, the Elixir/Phoenix/Ash stack was selected because it natively integrates lightweight concurrency, fault supervision, declarative multi-tenant authorization, and real-time state delivery within a significantly smaller operational footprint.

3.3 Backend Runtime and Concurrency Technologies

The backend tier is implemented in Elixir [8], a functional language built on the Erlang Virtual Machine (BEAM). [9] Engineered originally for telecommunications systems requiring five-nines availability, the BEAM runtime provides preemptive scheduling, fault tolerance, and lightweight concurrency primitives that align naturally with the requirements of orchestrating external virtualization infrastructure.

3.3.1 Actor-Based Concurrency and Preemptive Scheduling

At the core of the BEAM concurrency model is the actor abstraction, implemented as an isolated lightweight process. Unlike operating system threads which incur significant memory overhead and

context switching penalties, BEAM processes initialize with an operational footprint of only a few hundred bytes. The BEAM scheduler distributes execution slices across available CPU cores using reduction-based preemptive scheduling. In the context of the management platform, this ensures that CPU intensive routines (such as cryptographic token verification or heavy JSON payload deserialization) cannot monopolize compute resources or induce latency spikes in concurrent user sessions.

Inter-process communication relies exclusively on asynchronous message passing, eliminating shared mutable state across threads. Each process maintains an independent execution heap and garbage collector, which prevents long-tail garbage collection pauses from impacting the broader system. Within the platform, ephemeral hypervisor polling tasks, user session listeners, and administrative audit streams execute in decoupled process loops.

3.3.2 Hierarchical Supervision and Fault Isolation

System resilience is enforced through deterministic supervision trees, adhering to the standard Erlang “let it crash” engineering philosophy. When an unexpected exception occurs, such as a network socket drop or an unhandled HTTP response from a Proxmox node, the affected worker process terminates immediately without corrupting shared runtime state.

A dedicated supervisor detects the exit signal and enforces an appropriate restart strategy (such as `:one_for_one` for independent workers, or `:rest_for_one` for sequential pipeline stages). While supervision cannot remediate an external hypervisor outage, it acts as a circuit breaker, confining transient faults to the operational boundary of the failing request and guaranteeing uninterrupted service for other institutional tenants.

3.4 Declarative Domain Modeling and Authorization

3.4.1 Declarative Resource Engineering and Domain Actions

Conventional web architectures frequently scatter domain logic across controllers, ORM models, background jobs, and middleware layers. The Ash Framework addresses this by providing a declarative resource-oriented approach. [10] Resources define their attributes, relationships, calculations, validations, and lifecycle actions in a single centralized module.

Within this model, business operations are modeled as explicit domain actions (such as `:provision`, `:start`, `:stop`, or `:delete`) rather than generic database mutations. Actions encapsulate all input validation, multi-step state transformations, and external side-effects, ensuring uniform execution regardless of how the action is triggered.

3.4.2 Centralized Multi-Tenant Policies and Access Control

Multi-tenancy and institutional access control are enforced through Ash's declarative policy engine. Policies evaluate actor identity, tenant affiliation, resource state, and organizational quotas before allowing an action to proceed. Because policies are embedded directly within the domain layer, authorization rules are automatically enforced across LiveView interfaces, external API routes, and asynchronous Oban workers alike.

Policies are composed from ordered sets of checks, each of which either authorizes, forbids, or defers a request based on the evaluated condition. Typical checks within the platform include verifying that the acting user belongs to the tenant that owns the targeted resource, confirming that the actor's assigned role (student, educator, or administrator) permits the requested action, and validating that a provisioning request does not exceed the tenant's remaining quota. Because these checks are declared alongside the resource definition rather than duplicated in individual controllers or workers, a single policy change propagates consistently to every entry point that touches the resource.

In addition to action-level policies, field-level policies restrict which attributes and calculated values are visible to a given actor, preventing, for example, a student from viewing internal cluster identifiers even if the underlying resource is otherwise readable. This layered approach allows coarse-grained tenant isolation and fine-grained data visibility to be expressed within the same declarative framework, reducing the risk of inconsistent enforcement across the application's many interfaces.

3.5 Frontend and Reactive Interface Technologies

3.5.1 Phoenix LiveView and Stateful Process Architecture

Phoenix LiveView powers the platform's user interface. [11] LiveView enables rich, real-time client interactivity while executing application and rendering logic on the server. This model eliminates the need for separate client-side state management frameworks, REST serializer layers, and duplicate validation code.

When a user connects, the server initiates an isolated LiveView process that renders the initial static HTML and establishes a stateful WebSocket connection. User actions (button clicks, form inputs, pagination) are transmitted as lightweight binary events to the server process, which updates its state, re-renders the affected component diff, and pushes the minimal HTML patch back to the browser.

3.5.2 Real-Time Event Propagation via Phoenix PubSub

To notify users of background infrastructure changes without polling, the application integrates Phoenix PubSub. When an asynchronous provisioning job completes or a cluster node reports a status change, the event is broadcast across the relevant PubSub topic. Connected LiveView processes

subscribed to that topic receive the event, mutate their internal state, and instantly update the user's interface.

Topics within the platform are scoped hierarchically, typically combining a tenant identifier with a resource type or individual resource identifier (for example, a topic dedicated to a specific virtual machine's lifecycle events, and a broader topic covering all activity within a tenant). This scoping ensures that a LiveView process only receives the subset of events relevant to the resources it is currently displaying, avoiding unnecessary message traffic and re-render cycles for unrelated tenants or resources.

Because Oban workers and LiveView processes both run within the same BEAM runtime, a worker can broadcast a PubSub event as its final step immediately after committing a database transaction, without requiring an external message broker or polling interval. This tight coupling between background job completion and interface updates allows administrators and students to observe infrastructure state transitions such as a virtual machine moving from `queued` to `provisioning` to `running` within a sub second latency window of the underlying hypervisor operation actually completing.

3.6 Durable Background Processing with Oban

3.6.1 Transactional Job Orchestration and Retry Strategies

Hypervisor management operations (such as virtual machine cloning and disk reallocations) are long-running and cannot execute safely within ephemeral request processes. Oban provides transactional background job execution backed by PostgreSQL.[12] [13] Job insertions can be wrapped in the same database transactions as domain resource mutations, preventing state inconsistencies.

Oban also provides robust mechanisms for scheduling, execution timeouts, and exponential backoff retries. When a Proxmox cluster returns a transient HTTP error or an API timeout, the job engine records the execution failure, preserves the error stack trace, and schedules automated retries according to configured backoff policies.

3.6.2 Queue Partitioning and Workload Isolation

Workloads have distinct latency profiles. The platform uses Oban's queue partitioning to segregate tasks into specialized execution pipelines (e.g., `default`, `provisioning`, `synchronization`, and `metrics`). Independent concurrency limits per queue ensure that long-running provisioning tasks cannot monopolize system resources or delay lightweight lifecycle operations.

Each queue is configured with its own concurrency ceiling, allowing the platform to reflect the operational cost of the underlying work rather than applying a single global limit. The `provisioning` queue, which handles virtual machine cloning and disk allocation against external Proxmox nodes, is

deliberately constrained to a small number of concurrent workers to avoid overwhelming a hypervisor cluster with simultaneous long-running requests. In contrast, the `metrics` and `synchronization` queues, which perform lightweight polling and status reconciliation, are permitted a higher degree of concurrency since their individual jobs complete quickly and impose minimal load on both the database and the Proxmox API.

This partitioning also serves as a form of workload prioritization: because each queue is drained independently by its own pool of worker processes, a backlog in one queue (for example, a surge of synchronization jobs following a cluster-wide status refresh) does not delay jobs waiting in another queue. Combined with Oban's built-in job uniqueness and scheduling constraints, this allows the platform to guarantee that a student's request to start a virtual machine is not starved by unrelated background maintenance tasks running elsewhere in the system.

3.7 Data Persistence and Infrastructure Integration

PostgreSQL serves as the primary relational persistence tier, providing ACID transactional guarantees, foreign key constraints, and JSONB document support for cluster configurations. [13] Data access is facilitated by **Ecto** [14] and **AshPostgres**, bridging declarative Ash resources directly into optimized SQL queries and schema migrations. AshPostgres translates resource-level relationships, calculations, and aggregates into efficient joins and subqueries at the database layer, avoiding the N+1 query patterns that commonly arise when domain logic is layered on top of a generic ORM. Schema evolution is managed through Ecto migrations generated directly from resource definitions, keeping the database schema, domain model, and authorization rules in a single, version-controlled source of truth.

The platform interfaces with external Proxmox VE clusters via its HTTPS REST API. The integration layer encapsulates ticket and token-based authentication, node health polling, and asynchronous UPID (Unique Process Identifier) task tracking. Because Proxmox exposes long-running operations (such as cloning or disk resizing) as asynchronous tasks identified by a UPID rather than blocking HTTP calls, the integration layer is responsible for issuing the initial request, persisting the returned UPID against the corresponding domain resource, and then polling or subscribing to task status until completion or failure is confirmed. This status is subsequently reconciled with the internal resource state through the same Oban queues described in Section 3.6, ensuring that the platform's view of a virtual machine's lifecycle remains consistent with the authoritative state on the hypervisor cluster even in the presence of transient network interruptions.

3.8 Chapter Summary

3.8.1 Main Reasons for the Chosen Stack

The selected technology stack was chosen to satisfy the demanding requirements of multi-tenant infrastructure orchestration:

- **BEAM/Elixir** provides process-level fault isolation, massive non-blocking concurrency, and native supervision.
- **Ash Framework** centralizes domain logic, validations, and multi-tenant security policies in a declarative, auditable layer.
- **Phoenix LiveView** delivers low-latency, reactive user interfaces without the complexity of a decoupled frontend framework.
- **Oban and PostgreSQL** provide transactional, resilient background processing and durable persistence without requiring external message brokers.

3.8.2 Transition to the Core Application Architecture

With the technology foundation established, the following chapter examines the concrete application architecture. Chapter 4 presents the domain schema, resource models, authorization policies, and hypervisor coordination workflows that implement the system in practice.

Chapter 4

Core Platform Implementation & Domain Modeling

4.1 Introduction and Domain Architecture

4.1.1 Overview of the Implementation Scope

This chapter presents the software implementation, relational data structures, and operational algorithms of the multi-tenant cloud orchestration gateway. While Chapter 3 established the theoretical justification for the technology stack, evaluating the BEAM concurrency model, the declarative paradigm of the Ash Framework, Phoenix LiveView, and Oban job queues, this chapter translates those design decisions into a working system. Where Chapter 3 asked *why* this stack was chosen, this chapter answers *how* the chosen abstractions are composed into a coherent, operable platform. The remainder of the chapter follows the lifecycle of a request through the system. It begins with identity resolution and quota enforcement, then moves to the discovery and telemetry layer that keeps the application aware of the physical infrastructure. Next, it examines the catalog abstraction that mediates between raw hypervisor assets and end users, before concluding with the two asynchronous execution pipelines, provisioning and lifecycle management, that actually mutate hypervisor state.

The platform is designed to separate academic workflows from low-level hypervisor operations. In a university computing environment, students and professors need on-demand access to isolated virtual machines without having to deal with the complexity or security concerns of the underlying virtualization cluster. At the same time, system administrators need centralized control over physical compute nodes, resource quotas, operating system templates, and institutional identity mapping.

These two groups have different needs. End users want a simple, self-service experience rather than a complex hypervisor management console. Administrators, however, need enough visibility and control to manage the infrastructure safely and effectively. Balancing these needs within a single codebase is the main design challenge behind the domain boundaries described below.

To satisfy these requirements, the application architecture is organized into three primary domain bounded contexts:

1. **Identity and Access Domain (**Accounts**):** Governs user identities, institutional single sign-on mapping, academic role hierarchies, multi-dimensional hardware quotas, and distribution access permissions.
2. **Template and Distro Domain (**Catalog**):** Encapsulates operating system distribution offerings, providing an abstraction layer over raw hypervisor template disks with role-restricted access boundaries.
3. **Virtualization and Orchestration Domain (**Proxmox**):** Orchestrates physical hypervisor clusters, discovers hardware nodes, maintains in-memory telemetry caches, manages static network allocation via transactional database advisory locks, and executes asynchronous provisioning pipelines.

Each of these three contexts is deliberately kept narrow in scope and communicates with the others only through well-defined public interfaces exposed by the Ash domain layer, rather than through direct schema access. This mirrors standard bounded-context practice from domain-driven design: the Accounts domain has no knowledge of Proxmox API semantics, and the Proxmox domain has no knowledge of how roles are resolved from OAuth claims. The only shared vocabulary between domains is a small set of foreign-key references (for example, a virtual machine record's owning user identifier), which keeps the coupling between contexts low and makes each domain independently testable.

4.1.2 System Tiering and Control Flow Architecture

To separate user-facing operations from long-running virtualization tasks, the platform implements a four-tier architecture. Figure 4.1 illustrates the operational topology and boundary contracts governing the system.

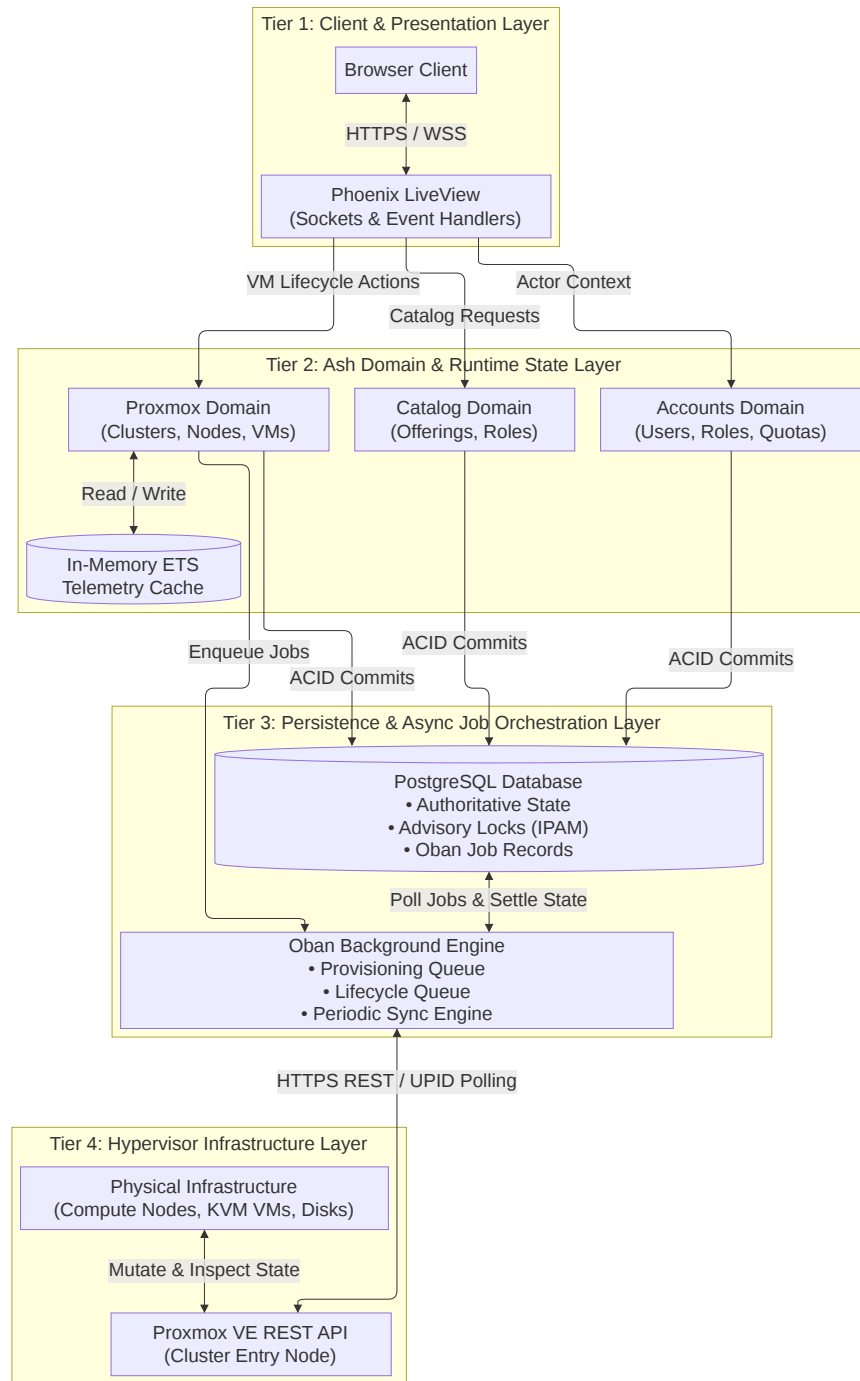


Figure 4.1: End-to-end platform architecture, tracing control flow from client LiveView sockets through the Ash domain layer, PostgreSQL and Oban orchestration, to the external Proxmox VE REST API.

The execution of operations across these boundaries proceeds through four core stages:

1. **Presentation Layer (Tier 1):** The client browser communicates over persistent WebSocket connections to Phoenix LiveView, ensuring low-latency interface updates and real-time state reflection without full-page reloads.
2. **Domain Application Layer (Tier 2):** LiveView actions dispatch into the Ash Framework domain layer (`Accounts`, `Catalog`, and `Proxmox`). High-frequency scheduling reads query an in-memory ETS telemetry cache directly, keeping the critical placement path free from database I/O bottlenecks.
3. **Persistence and Job Orchestration Layer (Tier 3):** Authoritative business rules, entity ownership, and deterministic IP leases (serialized via PostgreSQL advisory locks) [13] commit transactionally. Tasks involving hypervisor mutations are enqueued into dedicated Oban queues rather than blocking the web process.
4. **Hypervisor Infrastructure Layer (Tier 4):** Oban workers pull jobs, make authenticated REST calls against the cluster's entry node, poll task UPIDs to completion, and write final execution states back to PostgreSQL.

With the operational tiering established, the relational foundations supporting this control plane can be examined in detail.

4.1.3 Relational Entity-Relationship Model

The domain architecture is formalized as an integrated relational schema managed by the Ash Framework over a PostgreSQL database. Figure 4.2 illustrates the Entity-Relationship Diagram (ERD) defining the multi-tenant data model.

The schema establishes clear relational boundaries, cardinalities, and foreign key dependencies:

- **CLUSTER:** Represents an external Proxmox VE cluster. A cluster maintains a one-to-many relationship with physical `NODE` entities and discovered base `TEMPLATE` entities.
- **TEMPLATE:** Represents a disk template residing on a specific hypervisor node. Each template is abstracted into a user-facing `OFFERING` via a one-to-one association.
- **OFFERING:** Encapsulates a curated operating system distribution. An offering is linked to permitted `ROLE` entities through the `OFFERING_ROLE` join table, enforcing role-based catalog visibility.
- **ROLE:** Defines an academic privilege level (e.g., Bachelor Student, Master Student, Professor). A role has a one-to-many relationship with `AFFILIATION_MAPPING` rules and maintains a one-to-one baseline hardware limit via the `QUOTA` entity.

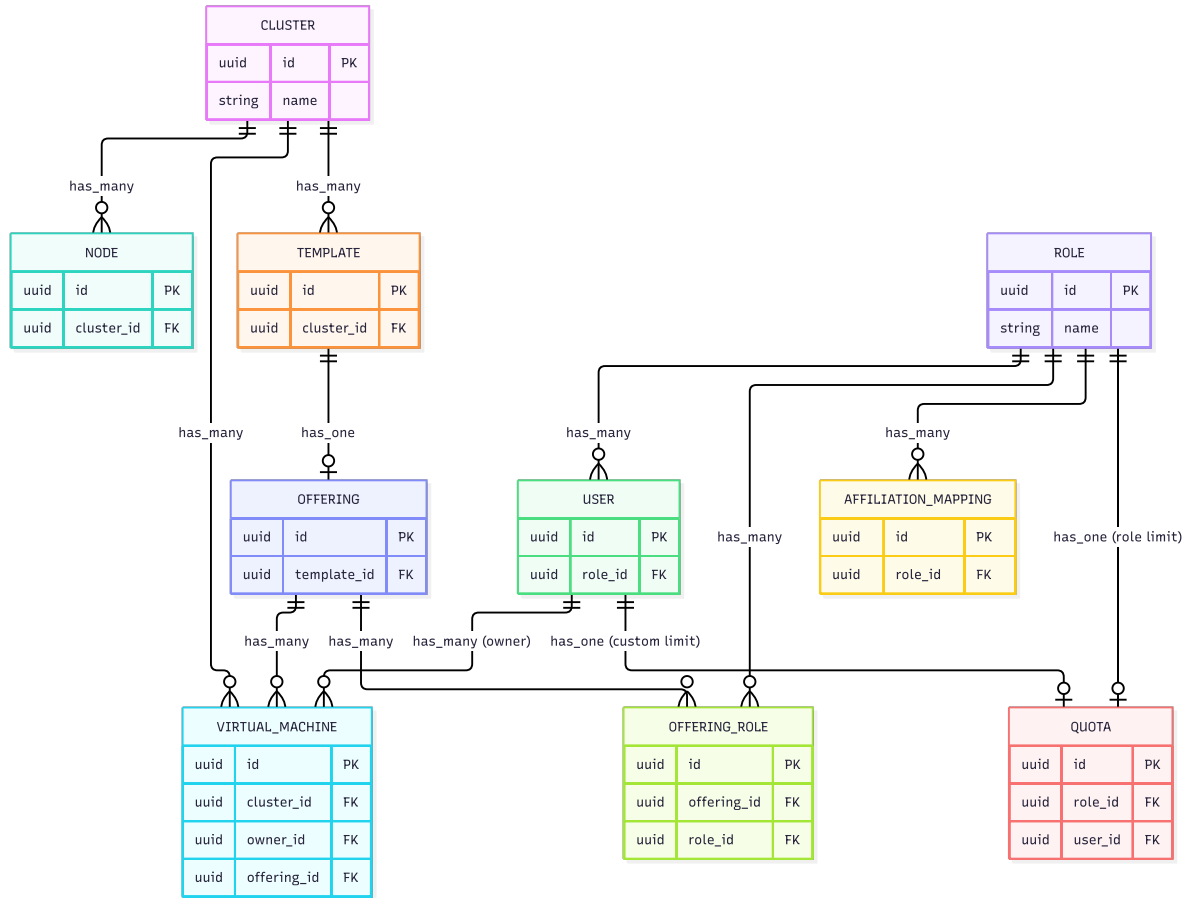


Figure 4.2: Entity-Relationship Diagram of the platform's domain architecture, illustrating cluster, catalog, user, role, quota, and virtual machine relationships.

- **USER:** Represents an authenticated institutional user. Each user belongs to a primary `ROLE` and can optionally possess a dedicated `QUOTA` override.
- **VIRTUAL_MACHINE:** Represents a provisioned virtual instance. A virtual machine is bounded by four distinct foreign keys: its parent `CLUSTER`, its execution `NODE`, its owning `USER`, and its base `OFFERING`.

Two design choices in this schema deserve explicit comment because they recur throughout the rest of the chapter. First, the `QUOTA` entity is modelled as a standalone resource with two nullable foreign keys (`role_id` and `user_id`) rather than as inline columns on `ROLE` and `USER`. This allows a single quota-resolution calculation, described in Section 4.2.4, to treat a role-level baseline and a user-level override identically, differing only in which foreign key is populated. Second, `TEMPLATE` and `OFFERING` are kept as separate resources rather than merged into one, even though the association is one-to-one at any given time. This separation exists because templates are *discovered* facts about the hypervisor (their existence, node placement, and clone availability are dictated entirely by what the synchronization pipeline observes), whereas offerings are *curated* facts maintained by administrators (their display name, minimum hardware requirements, and role visibility). Collapsing the two would force administrator-authored metadata to be overwritten on every synchronization pass, which is exactly the failure mode the separation is designed to avoid.

4.1.4 Separation of Application State and Hypervisor State

A fundamental design challenge in virtualization control planes is managing dual-state systems: the local application database stores user metadata and business rules, while the external hypervisor cluster maintains physical runtime execution state.

To prevent state desynchronization and avoid blocking web requests, the platform enforces a strict separation of concerns:

- **Authoritative Metadata Storage:** Entity ownership, role bindings, institutional affiliation mappings, static network leases, and quota ceilings are maintained authoritatively inside the local PostgreSQL database.
- **Derived Hardware State:** Virtual machine power states, physical node resource utilization, and template image availability are treated as dynamic external states that are queried out-of-band and synchronized into the application layer via background polling.

This distinction is not merely academic; it directly determines which failures are recoverable and which are not. If the application database is lost, ownership and quota records must be restored from backup because no other system holds that information. If the *derived* state cache becomes stale or is wiped—whether the in-memory telemetry cache described in Section 4.3.2 or a node's reported power state—the system self-heals on the next scheduled synchronization pass without any manual intervention, because the hypervisor cluster remains the single source of truth for that category of data. The remainder of this chapter is organized so that each domain's schema is presented alongside

an explicit statement of which of these two categories its attributes fall into, since that classification determines how the corresponding subsystem behaves under partial failure.

4.2 Identity, Affiliations, and Quota Enforcement

4.2.1 OAuth 2.0 Authorization Flow

Authentication is delegated to the central university identity provider using the OAuth 2.0 Authorization Code Grant Flow. Delegating authentication rather than implementing a local credential store was a deliberate choice: it means the platform never handles or stores institutional passwords, and it allows access to be revoked centrally the moment a student's institutional account is deactivated, without requiring any explicit deprovisioning logic inside the platform itself.

The authentication lifecycle proceeds through three distinct phases:

1. **Authorization Request (Step 1):** The user initiates authentication via `GET /auth/request`. The authentication controller redirects the browser (HTTP 302) to the institutional OAuth authorization endpoint requesting profile scopes.
2. **Callback Handling and Claim Extraction (Step 2):** The IdP redirects the browser back to `GET /auth/callback` with an authorization code. If an error occurs or access is denied, the application redirects the user to the landing page with a flash error. Upon receiving a valid code, the controller performs an out-of-band `POST /token` exchange to obtain an access token. It then queries `GET /profile` using the access token header, retrieving user metadata.
3. **User Upsert and Session Issuance (Step 3):** The application executes the user upsert routine within the domain layer, resolves academic roles from affiliation claims, generates an application JSON Web Token (JWT), establishes the authenticated user session, and redirects to `/portal`.

A design consequence of Step 2 is worth highlighting: the access token issued by the identity provider is used exactly once, to fetch the profile payload, and is never persisted. Only the derived application JWT—scoped to this platform and signed with a locally held secret is retained in the browser session. This keeps the blast radius of a leaked session cookie limited to the platform itself rather than to the broader set of institutional services the original IdP token might have granted access to. The application JWT carries the resolved user identifier and a short expiry window; LiveView socket connections revalidate the token on mount, so a revoked or expired session is rejected at the earliest possible point in the request lifecycle rather than being discovered mid-interaction.

The final phase of this flow, the user upsert, is where new academic identities are provisioned and existing ones are reconciled against the institution's records. Two lifecycle hooks fire as part of this step, and it is useful to introduce the first of them before turning to the supporting figure and schema.

4.2.2 First-User Bootstrap Hook

When the upsert routine described above runs, it first checks the total number of registered users in the system. If the record being created is the very first user the platform has ever seen, this hook overrides the incoming attributes to grant global administrative privileges, giving the institution's first login an initial administrator account without requiring a manual database seed script before the platform can be used. Every subsequent login is unaffected by this hook, since the user count check only ever succeeds once.

Figure 4.3 illustrates the end-to-end authentication sequence described above, covering the redirect, callback, token exchange, and upsert phases along with their error handling branches while Table 4.1 specifies the underlying relational entities, database constraints, and descriptions that the upsert routine reads from and writes to.

Table 4.1: Data schema for Identity, Role, and Quota entities.

Resource	Attribute	Type	Constraints	Description
User	id	UUID	Primary Key	Unique institutional user identifier.
	email	String	Unique, Required	Primary institutional email address.
	is_admin	Boolean	Default: false	Global administrative privilege flag.
	role_id	UUID	Foreign Key	Reference to assigned academic Role.
Role	id	UUID	Primary Key	Unique role identifier.
	name	String	Unique, Required	Display name (e.g., "Bachelor", "Professor").
	is_default	Boolean	Default: false	Fallback registration role indicator.
Quota	id	UUID	Primary Key	Unique quota record identifier.
	role_id	UUID	FK (Nullable)	Associated role limit reference.
	user_id	UUID	FK (Nullable)	Optional custom user override reference.
	max_cpu_cores	Integer	≥ 1 , Required	Hard ceiling on total provisioned vCPUs.
	max_ram_gb	Integer	≥ 1 , Required	Hard ceiling on total memory in gigabytes.
	max_storage_gb	Integer	≥ 1 , Required	Hard ceiling on persistent storage (GB).
	max_vms	Integer	≥ 1 , Required	Maximum allowed active virtual machines.
Affiliation Mapping	id	UUID	Primary Key	Unique mapping identifier.
	claim	String	Unique, Required	Institutional claim string from OAuth token.
	role_id	UUID	Foreign Key	Target Role assigned upon match.

Both this bootstrap hook and the affiliation resolution hook discussed next execute inside the same create/update transaction that performs the upsert itself. This guarantees that role resolution and admin bootstrapping cannot partially apply: either the entire upsert succeeds and the user record reflects a consistent role and privilege level, or the transaction rolls back and the previous session, if any, remains authoritative.

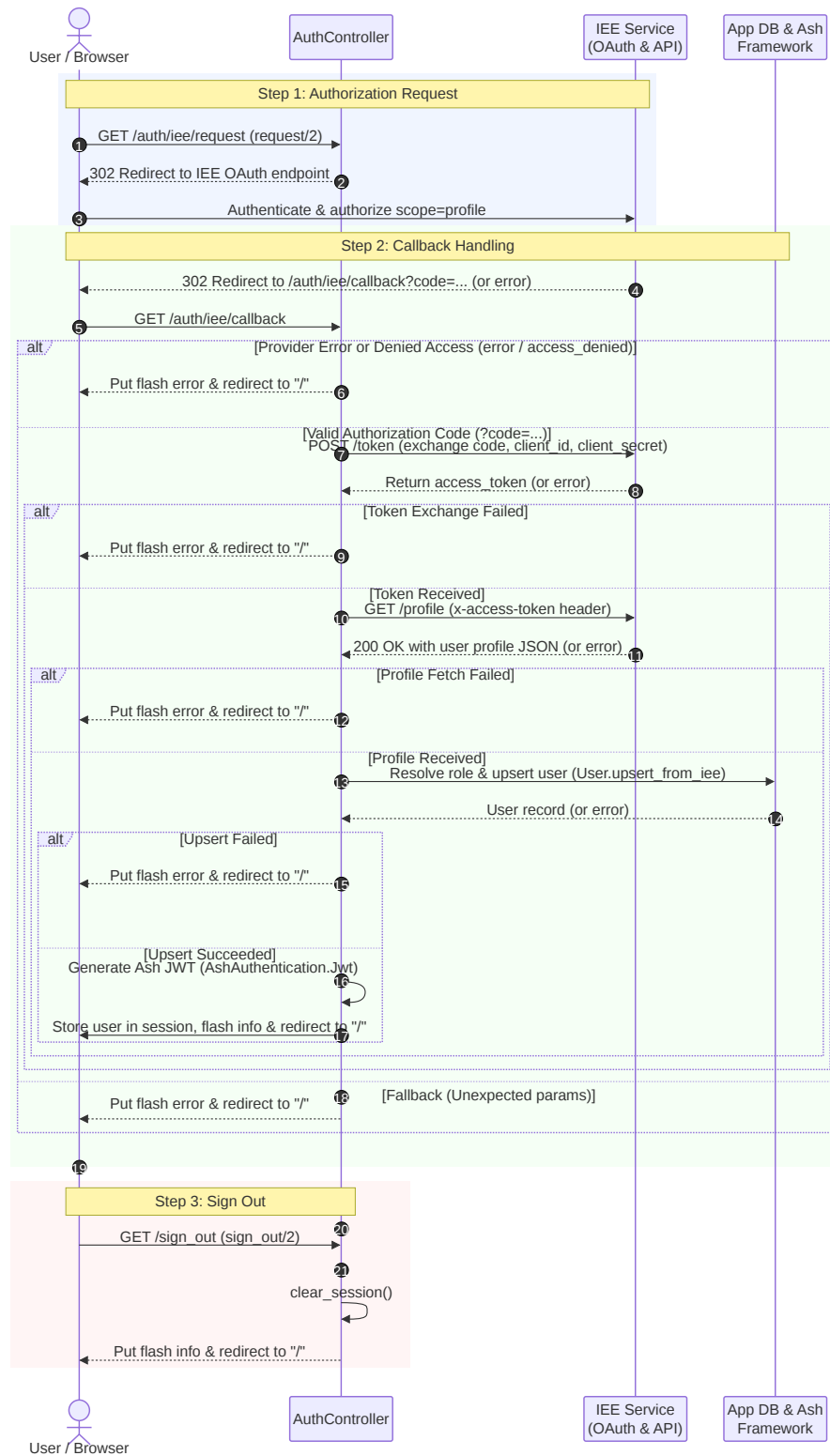


Figure 4.3: Sequence diagram of the OAuth 2.0 authentication, claim extraction, dynamic user upsert, and session establishment lifecycle.

4.2.3 Dynamic Affiliation Resolution

The second lifecycle hook resolves the user's institutional academic role. The user's institutional affiliation claims array (e.g., ["student", "undergraduate"]) is evaluated against active `AffiliationMapping` records from Table 4.1. If a matching claim is resolved, the user is bound to the corresponding `Role`. If no match is found, the user is assigned the system's designated default fallback role.

Because affiliation claims can legitimately change between logins—for instance, when a student is promoted from a Bachelor's to a Master's programme—this resolution hook re-evaluates on every login rather than only on first registration, so role changes made at the institutional level propagate into the platform automatically on the user's next sign-in, without requiring any administrator action inside the platform itself.

4.2.4 Mathematical Model of Resource Quota Verification

Resource allocation is governed by a quota verification step that runs as part of every virtual machine creation request. To prevent physical cluster exhaustion, every state except for fully deleted instances counts against a user's quota. The set of active states S_{active} is formally defined as:

$$S_{\text{active}} = \left\{ \begin{array}{l} \text{:pending, :provisioning, :running, :stopped,} \\ \text{:starting, :stopping, :shutting_down,} \\ \text{:rebooting, :restarting, :deleting, :locked_error} \end{array} \right\}$$

Notably, S_{active} includes `:locked_error` and `:deleting`. This is a deliberate conservative choice: an instance that failed mid-provisioning or is mid-deletion may still hold physical resources on the hypervisor (a partially cloned disk, an allocated static IP, a running process) even though it is not usable by the student. Excluding these transitional states from the quota calculation would allow a user to accumulate an unbounded number of failed or half-deleted machines while appearing, from the quota's perspective, to have full headroom available—effectively creating a resource-exhaustion loophole. By counting them, the quota system errs on the side of protecting the physical cluster at the minor cost of requiring an administrator or a cleanup routine to resolve stuck records before the user's headroom is restored.

Given a user u with assigned quota $\vec{Q}_u = (Q_{\text{cpu}}, Q_{\text{ram}}, Q_{\text{disk}}, Q_{\text{vms}})^T$ and the set of active virtual machines $V_u = \{v \in V \mid \text{owner}(v) = u \wedge \text{state}(v) \in S_{\text{active}}\}$, the current utilization vector $\vec{U}_u$ is computed as:

$$\vec{U}_u = \begin{pmatrix} U_{\text{cpu}} \\ U_{\text{ram}} \\ U_{\text{disk}} \\ U_{\text{vms}} \end{pmatrix} = \begin{pmatrix} \sum_{v \in V_u} v.\text{cpu_cores} \\ \sum_{v \in V_u} v.\text{ram_gb} \\ \sum_{v \in V_u} v.\text{storage_gb} \\ |V_u| \end{pmatrix}$$

When the user requests a new machine requiring hardware parameters $\vec{R}_{\text{req}} = (C_{\text{req}}, M_{\text{req}}, S_{\text{req}}, 1)^T$, the domain layer evaluates remaining headroom:

$$\vec{H}_{\text{remaining}} = \vec{Q}_u - \vec{U}_u$$

The provisioning action is permitted if and only if $\vec{H}_{\text{remaining}} \geq \vec{R}_{\text{req}}$ across all components:

$$\begin{cases} Q_{\text{cpu}} - U_{\text{cpu}} \geq C_{\text{req}} \\ Q_{\text{ram}} - U_{\text{ram}} \geq M_{\text{req}} \\ Q_{\text{disk}} - U_{\text{disk}} \geq S_{\text{req}} \\ Q_{\text{vms}} - U_{\text{vms}} \geq 1 \end{cases}$$

If any constraint is violated, the transaction is aborted before dispatching background jobs.

To make this concrete: a Bachelor Student role might carry a baseline quota of $\vec{Q} = (4, 8, 100, 2)$ —four vCPUs, 8 GB of RAM, 100 GB of storage, across at most two simultaneous machines. If the student already owns one running machine consuming $(2, 4, 40, 1)$, the remaining headroom is $\vec{H}_{\text{remaining}} = (2, 4, 60, 1)$. A request for a second machine requiring $(2, 4, 40, 1)$ would be accepted, since every component of $\vec{H}_{\text{remaining}}$ meets or exceeds the corresponding component of $\vec{R}_{\text{req}}$; a request for $(3, 4, 40, 1)$ would be rejected on the CPU dimension alone, even though RAM, storage, and VM-count headroom are all sufficient. This verification runs inside the same database transaction that would otherwise create the virtual machine record, so a denied request never reaches the point of allocating a static IP or enqueueing a provisioning job, keeping quota rejection an inexpensive, purely database-local operation.

4.3 Multi-Cluster Synchronization & Telemetry Architecture

With user identity, role, and quota resolution in place, the platform must also maintain an accurate, continuously refreshed picture of the physical infrastructure it manages. This is the responsibility of the Proxmox domain's discovery and telemetry subsystem, described next.

4.3.1 The Cluster Discovery and State Synchronization Engine

The platform interfaces with multiple independent Proxmox VE clusters through a periodic background synchronization pipeline, executed on a fixed cron-style schedule using Oban's periodic job scheduling capability, by default every 5 minutes. Figure 4.4 details the execution pipeline of the synchronization loop.

The pipeline processes two distinct execution shapes:

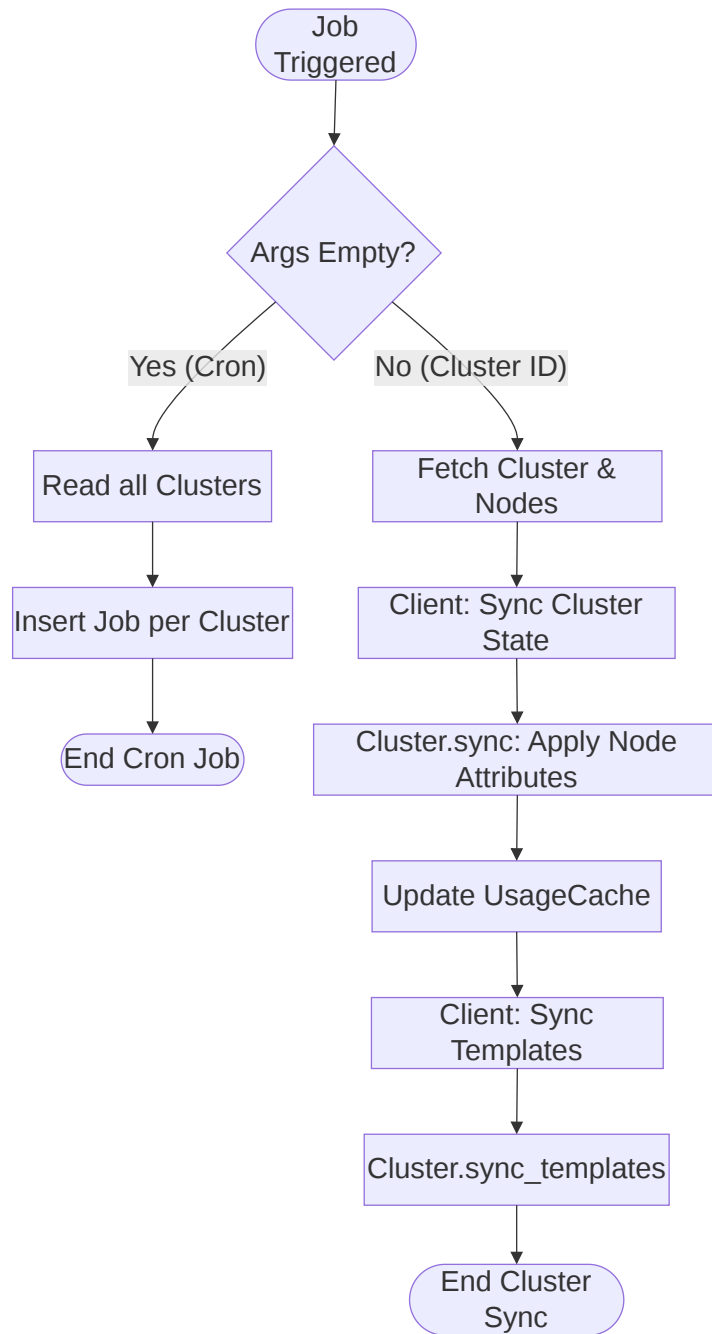


Figure 4.4: Execution flowchart of the periodic cluster discovery and synchronization pipeline.

1. **Periodic Fan-Out Phase:** The scheduled job reads all registered cluster entities from the database and inserts an independent background job for each cluster ID. This fan-out pattern isolates failure domains, ensuring that an unreachable or slow cluster cannot delay the synchronization of healthy clusters.
2. **Cluster Synchronization Phase:** Given a single cluster ID, a per-cluster job fetches the cluster record, reconnects through an available entry node, updates node attributes in the database, refreshes live in-memory telemetry, and discovers available template pools on that cluster.

The fan-out design also has a resilience benefit beyond isolation: because each per-cluster job is a discrete, independently retried background job, a transient failure against one cluster (for example, a momentary network partition to its entry node) is retried according to the job queue's standard exponential backoff policy without affecting the synchronization cadence of the other clusters. If a cluster fails synchronization repeatedly, its `status` attribute is updated to `:unreachable` or `:error`, and downstream consumers—most importantly the node placement logic described in Section 4.6.2—treat clusters in these states as ineligible for new provisioning requests, without requiring any special-case error handling at the call site. In effect, cluster reachability becomes just another dimension of the eligibility filter rather than a distinct failure mode that each caller must reason about separately.

Table 4.2 details the data attributes and relational structures used to model external hypervisor infrastructure.

The `allow_provisioning` flag on `Node` deserves separate mention, since it is the platform's primary manual override mechanism: an administrator can flip this flag to remove a node from placement consideration—for example, ahead of scheduled hardware maintenance—without deregistering the node entirely or interrupting virtual machines already running on it. Because the flag is consulted directly by the node placement algorithm in Section 4.6.2 rather than being encoded into the synchronization pipeline, its effect is immediate and does not depend on the next scheduled synchronization pass.

4.3.2 In-Memory Telemetry Layer

To support instantaneous target node placement without generating database write overhead on every sync cycle, node utilization metrics are stored in-memory using Erlang Term Storage (ETS), a concurrent, in-memory key-value store provided by the BEAM runtime. Persisting these metrics to PostgreSQL on every polling interval was considered and rejected: utilization figures are refreshed at least once per synchronization cycle and are consulted on the hot path of every provisioning decision, so routing them through disk-backed storage would introduce unnecessary write amplification and latency for data that, by definition, is already stale the moment the next poll completes. Keeping this data in a public, read-concurrent in-memory table instead allows any process—LiveView views rendering administrative dashboards, or the placement logic that runs during provisioning—to read current utilization with microsecond-level latency and no contention with the process that owns the writes.

Table 4.2: Data schema for Proxmox Infrastructure Entities.

Resource	Attribute	Type	Constraints	Description
Cluster	id	UUID	Primary Key	Unique cluster identifier.
	name	String	Unique, Required	Administrative cluster label.
	api_url	String	Required	Host IP/FQDN and port.
	token_id	String	Required	Proxmox API Token identifier.
	token_secret	String	Sensitive	Cryptographic API token secret.
	resource_pool	String	Optional	Target resource pool for templates.
	status	Atom	Enum	State: :online, :unreachable, :error.
Node	id	UUID	Primary Key	Unique node record identifier.
	name	String	Required	Physical node hostname.
	ip_address	String	Required	Node management IP address.
	status	Atom	Enum	Hardware state: :online, :offline.
	cpu	Float	≥ 0.0	Total physical CPU core capacity.
	ram_gb	Float	≥ 0.0	Total physical memory capacity (GB).
	allow_provisioning	Boolean	Default: true	Node placement eligibility toggle.
Template	id	UUID	Primary Key	Unique template record identifier.
	vmid	Integer	Required	Proxmox VMID of base template image.
	name	String	Required	Base template name.
	node_name	String	Required	Physical node hosting the template disk.
	available	Boolean	Default: true	Lock flag indicating clone availability.

The in-memory telemetry layer stores:

- **Per-Node Metrics:** Individual node entries containing CPU utilization percentage (`cpu_usage_percent` $\in [0.0, 100.0]$), memory utilization percentage (`ram_usage_percent` $\in [0.0, 100.0]$), and active instance count (`vm_count`).
- **Cluster-Wide Aggregates:** Arithmetic averages computed across active nodes (`avg_cpu_percent`, `avg_ram_percent`) and total cluster VM counts (`total_vms`).

The calculation of cluster-level metrics is expressed as:

$$\text{avg_cpu_percent} = \frac{1}{|N_{\text{active}}|} \sum_{n \in N_{\text{active}}} n.\text{cpu_usage_percent}$$

$$\text{avg_ram_percent} = \frac{1}{|N_{\text{active}}|} \sum_{n \in N_{\text{active}}} n.\text{ram_usage_percent}$$

$$\text{total_vms} = \sum_{n \in N_{\text{active}}} n.\text{vm_count}$$

Because this cache is process-local memory rather than a distributed store, it is intentionally scoped to a single application node; a horizontally scaled deployment would run one telemetry cache instance per application node, each independently refreshed by the same periodic synchronization schedule, rather than sharing a single global cache. Since every instance derives its state from the same authoritative Proxmox API responses on the same fixed interval, the caches remain eventually consistent with one another without requiring any cross-node synchronization protocol—a deliberate trade-off of strict consistency for operational simplicity, appropriate given that a few minutes of staleness in a load-balancing heuristic has no correctness impact, only a minor efficiency impact.

4.3.3 Cluster Statistics Calculation

A dedicated calculation layer combines persistent node capacity attributes from PostgreSQL (total physical CPU cores, total physical RAM in GB, and the `allow_provisioning` state) with dynamic in-memory telemetry from the cache described above. This layer provides a single, consistent source of truth for administrative LiveView dashboards, preventing data drift between cluster list views and detailed node inspection pages. By centralizing this join in one place rather than having each view independently query both the database and the cache, the platform avoids a class of bugs where two views of the same cluster disagree simply because they combined the two data sources in slightly different ways or at slightly different points in time.

4.4 Catalog Domain & Distro Offering Abstraction

Having established how the platform discovers and monitors physical infrastructure, the next question is how that infrastructure is exposed to end users in a controlled, curated form. To prevent end-users from interacting directly with raw hypervisor template IDs, the Catalog domain abstracts template disks into curated **Offerings**.

An `Offering` defines minimum hardware constraints (`min_cpu_cores`, `min_ram_gb`) and a default disk capacity. The `OfferingRole` join resource enables administrators to restrict offering visibility (e.g., exposing specialized GPU-enabled Linux images exclusively to Graduate Researchers while restricting General Undergraduates to standard lightweight Linux distributions). This indirection also gives administrators a safety mechanism independent of the underlying hypervisor: a template can be temporarily marked `available: false` at the `Template` level (for instance, while a base image is being patched) without deleting or modifying the `Offering` that references it, so the catalog entry simply becomes temporarily unselectable in the portal rather than needing to be recreated once the template is ready again.

Table 4.3 details the catalog schema layout and access mapping constraints.

Table 4.3: Catalog and Distribution Offering Data Schema.

Resource	Attribute	Type	Constraints	Description
Template	id	UUID	Primary Key	Unique local template identifier.
	vmid	Integer	Required	Hypervisor VMID of the template disk image.
	name	String	Required	Identifier of the base image.
	node_name	String	Required	Host node storing the template volume.
	available	Boolean	Default: <code>true</code>	Lock flag indicating template clone availability.
	cluster_id	UUID	Foreign Key	Reference to parent <code>Cluster</code> .
Offering	id	UUID	Primary Key	Unique offering identifier.
	name	String	Unique, Required	Public distribution name (e.g., "Ubuntu 24.04 LTS").
	template_id	UUID	Foreign Key	Underlying hypervisor <code>Template</code> image.
Offering Role	id	UUID	Primary Key	Unique join record identifier.
	offering_id	UUID	Foreign Key	Permitted distribution offering.
	role_id	UUID	Foreign Key	Target academic role permitted access.

At request time, the set of offerings a given user may select from is computed as the join of `OfferingRole` rows matching the user's resolved role, filtered further to offerings whose underlying `Template` is currently `available`. This filtering happens as a read-side authorization check inside the data query layer itself—rather than being enforced only in the user interface—so that even a direct, crafted request for a restricted offering's identifier is rejected at the domain layer, not merely hidden from the interface.

4.5 Deterministic Static Network Allocation

Once a user has selected a role-appropriate offering, the provisioning pipeline that ultimately fulfills that request needs a mechanism to assign the new machine a network identity. The platform addresses this with a self-contained static allocation component, described below.

4.5.1 Subnet Constraints and Allocation Pool

The platform manages static IPv4 assignments across a dedicated institutional subnet block without requiring an external IP Address Management (IPAM) daemon. A DHCP-based approach was considered during design but rejected for this deployment: academic virtual machines are frequently long-lived across a semester, are addressed directly by students for SSH access, and are sometimes referenced in coursework instructions by fixed address—properties that are all easier to guarantee with deterministic static allocation recorded directly against the owning `VirtualMachine` record than with a lease-based DHCP pool that could, in principle, reassign an address after an expiry window.

Rather than hard-coding these network parameters, the subnet base address, gateway, and allocatable host offset range are all read from environment variables at application startup. This allows the same codebase to be deployed against different institutional network layouts, or reconfigured as the available address space changes, without any code modification or redeployment beyond an environment update. For the deployment used during development and testing, these variables were configured as:

- Subnet Base: `192.168.1.30/24`.
- Gateway Address: `192.168.1.1`.
- Allocation Host Offset Range: Host `.32` to Host `.245` (reserving lower host addresses for network appliances and cluster management interfaces).

Under this particular configuration the pool yields 214 assignable addresses, but this figure is a property of the chosen host range rather than a fixed platform constant, and a production deployment against a different subnet would size its pool accordingly. Because the pool size is finite and administrator-defined, ip exhaustion is a real operational condition the system must surface rather than fail silently on; Algorithm 1 below explicitly returns a `:subnet_exhausted` error rather than allowing a request to proceed with a null or invalid address.

4.5.2 Concurrency Control via PostgreSQL Advisory Locking

Multiple provisioning jobs are permitted to execute concurrently, so concurrent jobs could, without coordination, read the same unassigned IP and attempt to assign duplicate addresses. To eliminate this race condition, address discovery is wrapped in a PostgreSQL transaction protected by a transactional

advisory lock, keyed by a stable hash derived from the allocator's namespace and the configured subnet and prefix:

$$\text{Lock Key} = \text{hash}(\text{allocator namespace, subnet, prefix})$$

An advisory lock was chosen over an alternative such as a unique database constraint on the `ip_address` column combined with optimistic retry, because the allocation routine needs to *search* for the first free address in an ordered range rather than simply validate a single candidate; a unique-constraint-and-retry strategy would require repeatedly re-querying and re-attempting under contention, whereas the advisory lock serializes the entire search-and-assign operation into a single atomic critical section, at the cost of brief lock contention when multiple provisioning jobs allocate addresses at nearly the same instant. Given the concurrency ceiling configured for the provisioning queue (five simultaneous jobs during development and testing, see Section 4.6), this contention window is small enough not to be a practical bottleneck. Algorithm 1 details the sequential static address allocation routine.

Algorithm 1: Deterministic Static IPv4 Allocation with Transactional Advisory Locking

Input : Subnet Base (S , read from configuration; $S = \{192, 168, 1, 30\}$ in the tested deployment), Host Range ($[H_{\text{start}}, H_{\text{end}}]$, read from configuration; $[32, 245]$ in the tested deployment)

Output: Allocated IPv4 String (IP) or Error (`:subnet_exhausted`)

begin

```

  BeginDatabaseTransaction();
  AcquirePostgresAdvisoryLock(LockKey);
  assigned_records ← QueryActiveVmIpsFromDatabase();
  assigned_set ← MapSet.new(assigned_records);
  allocated_ip ← nil;
  for host ←  $H_{\text{start}}$  to  $H_{\text{end}}$  do
    candidate_ip ← FormatIpString( $S$ , host);
    if candidate_ip  $\notin$  assigned_set then
      allocated_ip ← candidate_ip;
      break;
    end
  end
  if allocated_ip  $\neq$  nil then
    CommitDatabaseTransaction();
    return { :ok, allocated_ip };
  else
    RollbackDatabaseTransaction();
    return { :error, :subnet_exhausted };
  end
end

```

end

Because the advisory lock is held only for the duration of the search-and-record step and is auto-

matically released when the enclosing transaction commits or rolls back, a crashed or timed-out provisioning job cannot leave the lock permanently held; PostgreSQL releases session-scoped advisory locks when the holding connection terminates, which prevents a single failed job from deadlocking all subsequent IP allocation attempts.

4.6 Asynchronous Provisioning Pipeline

With a role-appropriate offering selected, quota headroom confirmed, and a mechanism in place to assign a static address, the platform is ready to describe the pipeline that turns an accepted request into a running virtual machine.

4.6.1 Pipeline Execution Flow

The complete automated provisioning pipeline is orchestrated by a dedicated background worker executing within a provisioning-specific job queue. The concurrency limit for this queue is not hard-coded; it is read from an environment variable at startup, and a limit of 5 concurrent jobs was the value used throughout development and testing. This job is deliberately kept out of the synchronous web request path: the end-to-end pipeline described below involves multiple sequential calls to the Proxmox API, several of which (disk cloning in particular) can take minutes to complete, which would be entirely unsuitable for a request-response HTTP cycle or even a LiveView event handler. Instead, the LiveView layer enqueues the job and immediately returns control to the user, who observes progress through the persisted VM state transitions rather than through a blocking wait. Figure 4.5 illustrates the 7-step execution workflow and error isolation paths.

Algorithm ?? formalizes the complete 7-step provisioning workflow.

Two structural properties of Algorithm ?? are worth drawing out explicitly. First, Steps 1 and 2—node placement and identifier/address acquisition—execute *outside* the surrounding try/catch block. This is intentional: a failure at this stage means no hypervisor-side resources have been created yet, so the recovery path is simply to record the failure reason and release the VM record back to an error state, with nothing to unwind on the Proxmox side. From Step 3 onward, however, the job has begun mutating external hypervisor state (a cloned disk, a partially configured guest), so every subsequent failure is caught by the shared handler, which transitions the record to `:locked_error` rather than leaving it in an ambiguous intermediate status; a locked machine is deliberately *not* automatically re-tried, since automatically re-running a clone or configuration step against a partially created VMID risks producing an inconsistent guest, and instead surfaces for administrator or user-initiated cleanup. Second, the synthetic system actor used to fetch and mutate the VM record allows the background worker—a process with no HTTP request or session context of its own—to reuse the exact same authorization policies that govern interactive requests, rather than requiring a parallel set of "background job" permission rules; this synthetic actor is simply granted the equivalent of administrator rights for the duration of the job, scoped narrowly to the single VM it was enqueued to provision.

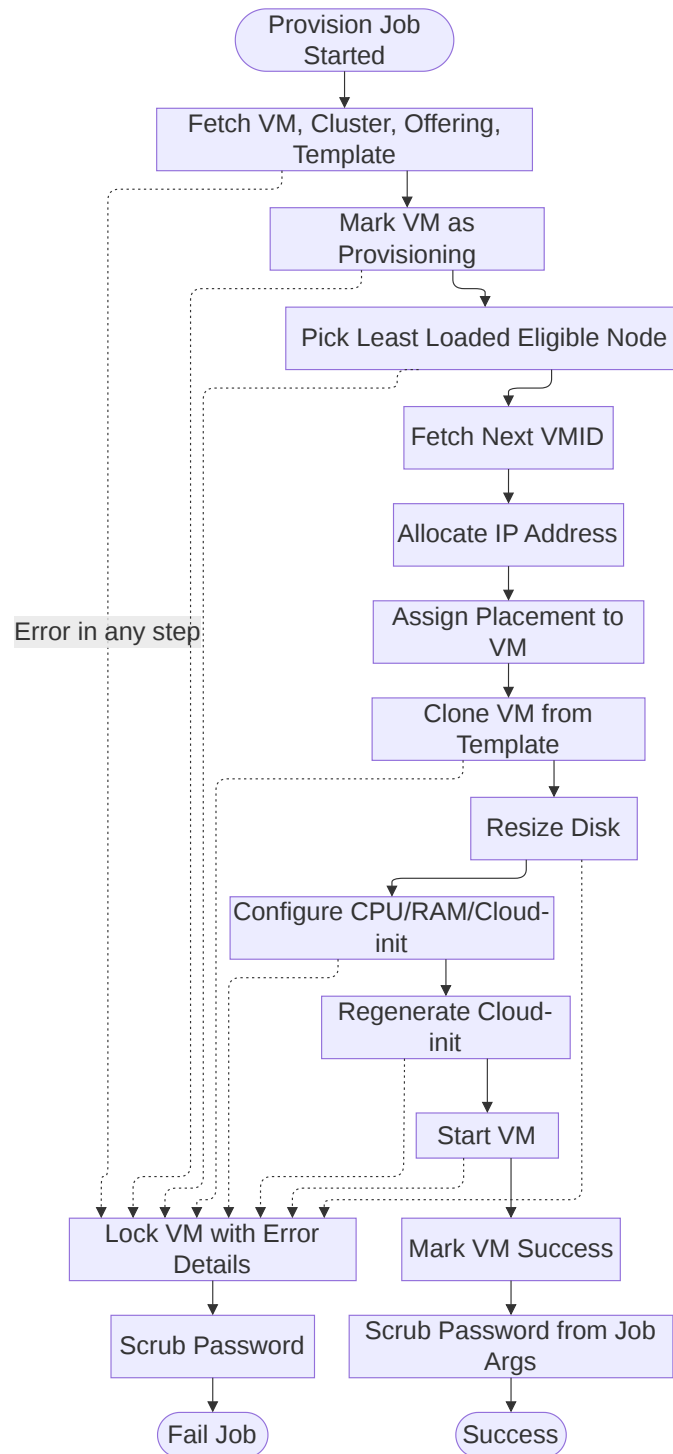


Figure 4.5: Execution flowchart of the automated virtual machine provisioning pipeline, illustrating step progression, error recovery paths, and database password scrubbing.

Algorithm 2: Virtual Machine Provisioning Pipeline

Input : Job ID (*job_id*), VM ID (*vm_id*), Root Password (*password*), VM Specs (*cores, ram, storage*)

Output: Execution Result (Success or Error)

actor $\leftarrow$ SystemAdmin;

vm $\leftarrow$ FetchVM(*vm_id*, *actor*);

MarkProvisioning(*vm*, *actor*);

creds $\leftarrow$ GetCredentials(*vm.cluster*);

target_node $\leftarrow$ PickLeastLoadedNode(*vm.cluster.nodes*, *creds*, *vm.template*);

if *target_node* = $\emptyset$ **then**

 LockVMWithError(*vm*, NoEligibleNode);

 ScrubPassword(*job_id*);

return Error;

end

new_vmid $\leftarrow$ AllocateNextVMID(*creds*);

ip_address $\leftarrow$ AllocateStaticIP();

AssignPlacement(*vm*, *new_vmid*, *target_node*, *ip_address*);

try {

 CloneTemplate(*creds*, *vm.template*, *target_node*, *new_vmid*, *vm.name*);

 ResizeDisk(*creds*, *target_node*, *new_vmid*, "scsi0", *storage*);

ci_config $\leftarrow$ BuildConfig(*cores*, *ram*, "root", *password*, *ip_address*);

 ConfigureGuest(*creds*, *target_node*, *new_vmid*, *ci_config*);

 RegenerateCloudInit(*creds*, *target_node*, *new_vmid*);

 PowerOn(*creds*, *target_node*, *new_vmid*);

 MarkSuccess(*vm*);

result $\leftarrow$ Success;

}

catch (Exception e) {

 LockVMWithError(*vm*, *reason*);

result $\leftarrow$ Error;

}

finally {

 ScrubPassword(*job_id*);

}

return *result*;

4.6.2 Algorithmic Node Placement Optimization

Target node selection evaluates hardware constraints followed by least-loaded scoring:

1. **Eligibility Filtering:** The candidate node must satisfy `node.allow_provisioning = true` and `node.status = :online`.
2. **Feature Capability Check:** The node must support cloning the specific template disk format across the required storage pools, which is verified via a hypervisor capability check against the target node.
3. **Least-Loaded Node Selection:** The placement engine queries the in-memory telemetry layer to find the node minimizing the combined load score:

$$\text{Target Node} = \arg \min_{n \in N_{\text{capable}}} (\text{cpu_usage_percent}(n) + \text{ram_usage_percent}(n))$$

This combined-load scoring function is a deliberately simple heuristic rather than a full bin-packing solver: summing CPU and RAM percentage utilization treats the two resources as equally weighted, which is a reasonable approximation for the relatively homogeneous academic workload profile this platform targets (small, general-purpose Linux images), but it does mean the scoring function does not account for storage utilization or network topology when selecting a node. This is an accepted simplification for the current deployment scale, and is revisited as a direction for future work in Section 4.8. Because the score is read directly from the in-memory telemetry layer rather than by querying the Proxmox API synchronously at placement time, the entire eligibility-and-scoring computation in Step 1 of Algorithm ?? completes in-process, without incurring any additional external API latency on the critical provisioning path.

4.6.3 Post-Execution Database Credential Scrubbing

To inject initial guest credentials without manual terminal interaction, plaintext passwords must be passed to the Cloud-Init drive configuration. Because Oban persists job arguments as JSONB data in PostgreSQL, persistent password storage would create security vulnerabilities in database backups.

To neutralize this risk, the provisioning worker executes an SQL parameter scrubbing query immediately upon job termination:

```
UPDATE oban_jobs SET args = args - 'password' WHERE id = $1
```

This guarantees that sensitive passwords exist in persistent storage only during the active provisioning window and are purged immediately upon success or failure. Crucially, this scrubbing statement executes in a job-completion callback that Oban guarantees to run regardless of whether the job body returned `:ok` or an error tuple, or raised an unhandled exception—so a crash partway through Step 5 or 6 of the pipeline does not leave the plaintext password lingering in the job queue's storage table.

The only residual exposure window is the interval between job insertion and the scrubbing statement's execution, which is bounded by the pipeline's own runtime and is not persisted beyond that job's lifetime; no other table or log sink in the platform is written with the plaintext value.

4.7 Lifecycle Power Management

Once a virtual machine has been successfully provisioned, ongoing interaction with it is dominated not by further disk or network changes but by simple power-state transitions, which the platform handles through a separate, lighter-weight worker.

Fast lifecycle power commands (`start`, `stop`, `shutdown`, `reboot`, `restart`) are executed asynchronously within a dedicated lifecycle job queue, distinct from the provisioning queue described in Section 4.6. Like the provisioning queue, its concurrency limit is configured via an environment variable rather than fixed in code; a limit of 5 concurrent jobs was likewise used throughout development and testing. This queue is kept distinct from the provisioning queue even though both ultimately call the same Proxmox client, because lifecycle actions are short-lived (typically completing in seconds) and latency-sensitive from the user's perspective, whereas provisioning jobs are long-running; sharing a single queue would allow a backlog of multi-minute clone operations to starve quick power-toggle requests behind them.

The worker maps action identifiers to internal handler operations, the corresponding Proxmox API endpoint, and a target terminal state, as summarized in Table 4.4.

Table 4.4: Action Dispatch and State Settling Mapping for Lifecycle Power Commands.

Action Name	Handler Operation	Proxmox Endpoint	Target State	Semantics
"start"	Power On	/status/start	:running	Power on virtual machine.
"stop"	Hard Stop	/status/stop	:stopped	Hard instantaneous power off.
"shutdown"	Graceful Shutdown	/status/shutdown	:stopped	Graceful ACPI OS shutdown.
"reboot"	Graceful Reboot	/status/reboot	:running	Graceful ACPI OS reboot.
"restart"	Hard Reset	/status/reset	:running	Hard hardware reset.

When an action executes, the worker invokes the corresponding handler operation, marks the returned task identifier (UPID) in PostgreSQL for UI progress tracking, polls the task status, and marks the VM state as successful. If an exception occurs, the worker transitions the VM to `:locked_error` and records the error diagnostics.

Dispatching through a lookup table keyed by action name, rather than a conditional chain of branches per action, keeps the addition of future lifecycle actions (for example, pause/resume or snapshot operations) to a single new table row plus the corresponding client call, without touching the worker's control flow. Because every action in the table shares the same task-tracking-and-poll pattern, this dispatch design also guarantees that every lifecycle action, regardless of which Proxmox endpoint it targets, is observed by the user through the identical progress-and-settlement mechanism—avoiding

a situation where, say, `reboot` feels responsive while `restart` appears to hang because it was implemented with a slightly different polling strategy.

4.8 Limitations and Directions Suggested by the Implementation

While the design decisions described above satisfy the platform's core requirements, several are explicit simplifications made in light of the deployment's current scale, and are worth stating plainly rather than leaving implicit. The node placement heuristic in Section 4.6.2 does not account for storage utilization or physical network locality, which would matter more at a scale involving many more nodes or heterogeneous hardware generations. The static IPv4 allocator described in Section 1, although its subnet and host range are externally configurable rather than fixed in code, still assumes a single flat subnet per deployment; supporting multiple subnets per cluster or IPv6 addressing would require generalizing the allocator beyond its current single-subnet assumption. Finally, a `:locked_error` state is, by design, never automatically retried, which prevents compounding failures. However, this also means that transient errors—such as an intermittent API timeout during Step 3 of provisioning—currently require manual intervention rather than being distinguished from genuinely non-recoverable failures. These limitations are treated as scope boundaries of the present implementation rather than defects and are briefly revisited in the discussion of future work.

4.9 Chapter Summary

This chapter detailed the concrete software implementation and domain architecture of the platform, tracing the path of a request from authentication through to a running, network-addressable virtual machine.

We presented the Relational Entity-Relationship Diagram and specified the data schemas for accounts, roles, quotas, clusters, nodes, templates, offerings, and virtual machines, and explained the rationale behind key modelling choices such as the separation of `Template` and `Offering`. We detailed the OAuth 2.0 sequence flow, first-user administrative bootstrapping, dynamic affiliation mapping, and the mathematical formalization of quota headroom verification, including a worked numerical example of the headroom calculation. We explored periodic cluster discovery and synchronization, in-memory telemetry caching, and cluster statistics calculations, along with the operational rationale for isolating cluster failures through per-cluster job fan-out. We formalized deterministic static IPv4 allocation—configurable via environment variables rather than fixed in code—protected by PostgreSQL advisory locking, contrasting it against a DHCP-based alternative. Finally, we analyzed the 7-step provisioning pipeline, its node placement heuristic, its post-execution password scrubbing guarantee, and its configurable concurrency limits, alongside power lifecycle dispatching, and closed with an explicit statement of the implementation's current scope limitations.

With the platform implementation established, Chapter 5 presents platform operations, user work-

flows, and qualitative system reliability verification—evaluating, in practice, how the architectural guarantees described in this chapter hold up under real academic usage.

Chapter 5

Platform Operations, Workflows, and System Verification

5.1 Introduction and Operational Overview

5.1.1 Operational Paradigm and Interface Tiers

The cloud orchestration platform functions as a centralized, real-time control plane positioned between academic users and physical Proxmox VE hypervisor clusters. The system translates high-level web actions into asynchronous hypervisor operations, abstracting complex virtualization mechanics into a streamlined self-service experience while enforcing strict multi-tenancy and resource governance.

The presentation tier is implemented using Phoenix LiveView, establishing stateful, bidirectional WebSocket connections between client browsers and server processes. This architecture eliminates the state duplication and serialization latency inherent in traditional single-page applications (SPAs).

The system architecture organizes user interaction into two primary interfaces:

1. **Student and Faculty Self-Service Portal:** An interface for non-administrative academic users to monitor allocated quotas, provision virtual compute instances from curated operating system templates, and manage instance power states.
2. **Administrator Control Console:** A comprehensive management suite for system operators to register hypervisor clusters, monitor node telemetry, manage distribution offerings, configure institutional affiliation mappings, oversee the global VM fleet, and apply custom quota overrides.

5.1.2 Real-Time State Synchronization and Event Delivery

A core requirement of cloud management portals is the live delivery of asynchronous infrastructure updates without manual browser refreshes. When a user provisions an instance or issues a power command, the task is dispatched to dedicated Oban worker queues.

The platform leverages Phoenix PubSub to broadcast state changes across isolated topics. When an Oban worker settles an instance transition (e.g., transitioning from `:provisioning` to `:running`) or records an error condition in PostgreSQL, an event payload is published. Subscribed Live-View processes handle the event, update their internal state assigns, calculate minimal HTML diffs, and push binary patches across WebSockets to surgically update the client interface.

5.2 Institutional Onboarding and Authentication Flow

Access to the platform is gated by the university's central identity provider (IdP) via OAuth 2.0. Figure 5.1 illustrates the public landing page and the institutional authentication interface.

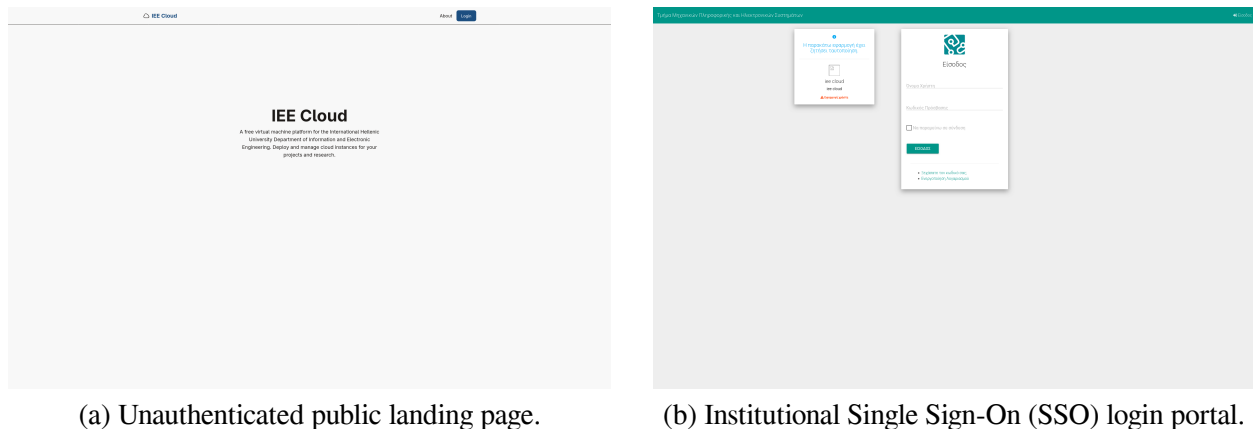


Figure 5.1: Public landing page and institutional OAuth 2.0 Single Sign-On (SSO) login portal of the Department of Information and Electronic Engineering.

The authentication workflow proceeds as follows:

1. **Authentication Initiation:** An unauthenticated user accesses the platform landing page and selects the "Login" action.
2. **Institutional Redirection:** The authentication controller redirects the browser to the university identity provider via an authorization code request.
3. **Credential Verification:** The user enters their institutional credentials on the university SSO page (confirming identity assertions for departmental systems).

4. **Token Exchange and Claim Extraction:** Upon callback, the platform exchanges the authorization code for an access token via back-channel communication and queries the user profile endpoint.
5. **Dynamic Role Resolution and Session Creation:** The platform parses the user's institutional affiliation claims, maps the user to their designated academic role and quota envelope, creates or updates the local user entity in PostgreSQL, issues a secure JSON Web Token (JWT), and establishes the active session.

5.3 Student and Faculty Self-Service Operations

5.3.1 Overview Dashboard and Quota Visualization

Figure 5.2 illustrates the primary user dashboard upon successful login.

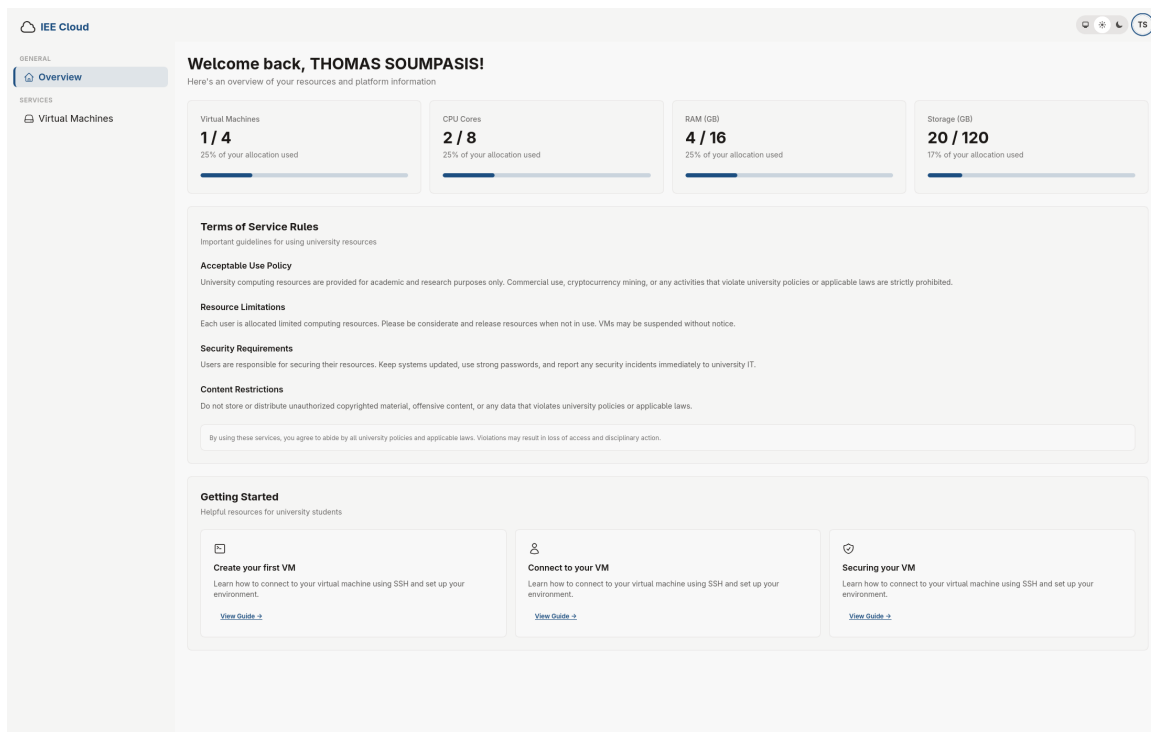


Figure 5.2: Student Self-Service Overview Dashboard, displaying active resource quota consumption bars, compute summaries, terms of service rules, and getting started guides.

The dashboard provides real-time visibility into personal resource utilization through four dedicated quota metric cards:

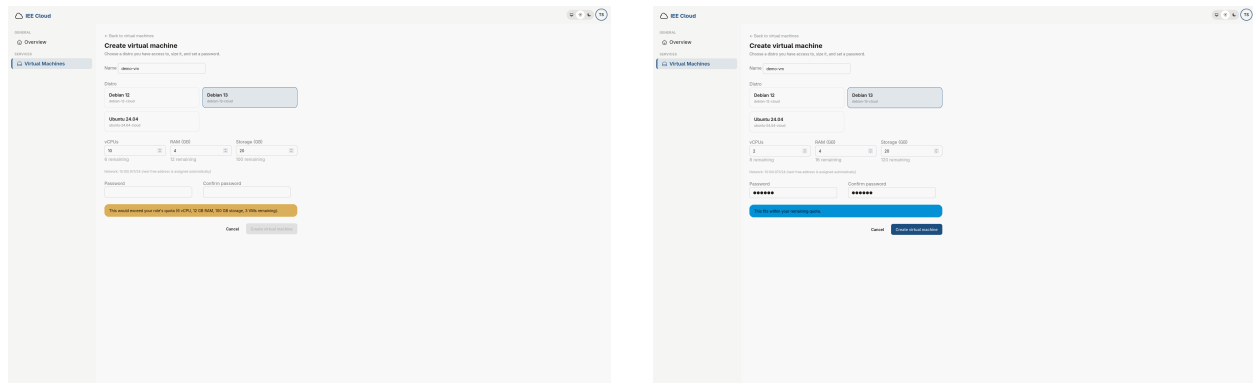
- **Virtual Machines:** Displays total active virtual instances relative to the allowed limit (e.g., 1/4 instances, 25% utilization).

- **CPU Cores:** Displays allocated virtual processor cores against the quota ceiling (e.g., 2/8 cores, 25% utilization).
- **RAM Allocation:** Displays cumulative memory allocation in gigabytes (e.g., 4/16 GB, 25% utilization).
- **Storage Allocation:** Displays allocated persistent storage capacity in gigabytes (e.g., 20/120 GB, 17% utilization).

Below the quota indicators, the dashboard renders institutional Terms of Service Rules (covering acceptable use, resource limits, security obligations, and content restrictions) and interactive Getting Started guides ("Create your first VM", "Connect to your VM", "Securing your VM").

5.3.2 Virtual Machine Provisioning Wizard and Quota Validation

When an academic user initiates the creation of a new instance, the platform presents an interactive provisioning wizard. Figure 5.3 demonstrates the real-time client validation under both quota-exceeded and within-quota conditions.



(a) Reactive quota violation warning (10 vCPUs requested, 6 remaining).

(b) Validated configuration within remaining quota headroom (2 vCPUs, 4 GB RAM).

Figure 5.3: Interactive Virtual Machine Provisioning Wizard, illustrating dynamic client-side quota headroom calculation, offering selection, and automated subnet assignment.

The provisioning wizard executes a multi-step configuration sequence:

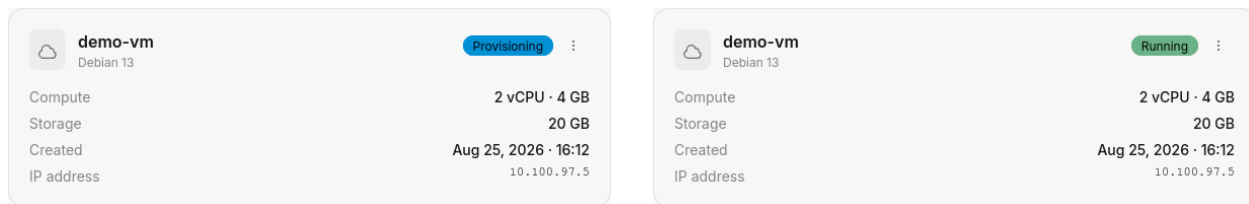
1. **Hostname Assignment:** The user specifies an instance label (e.g., `demo-vm`).
2. **Distribution Selection:** The user selects a base operating system from the curated distro offerings assigned to their role (e.g., Debian 12, Debian 13, or Ubuntu 24.04 LTS).
3. **Hardware Sizing and Real-Time Quota Checking:** The user configures requested vCPU cores, RAM (GB), and persistent storage (GB). LiveView evaluates remaining quota headroom

dynamically. If the configuration exceeds limits (e.g., requesting 10 vCPUs when only 6 remain), an amber warning banner is displayed: "This would exceed your role's quota (6 vCPU, 12 GB RAM, 100 GB storage, 3 VMs remaining)", and submission is disabled. When the parameters fit within available headroom, a confirmation banner appears, and the submission action is enabled.

4. **Static Network and Subnet Indication:** The interface informs the user of the assigned network block (e.g., `10.100.97.1/24`), indicating that the next available static host address will be allocated automatically.
5. **Authentication Configuration:** The user enters and confirms a secure root password for Cloud-Init injection.

5.3.3 Asynchronous State Progression and UI Updates

Upon form submission, Ash validates the transaction via `WithinQuota`, stages the record in state `:pending`, and enqueues a job into Oban's `:vm_provisioning` queue. Figure 5.4 illustrates the real-time visual progression of the virtual machine card.



(a) Transitional Provisioning badge with pre-allocated static IP.

(b) Terminal Running badge with active lifecycle management menu.

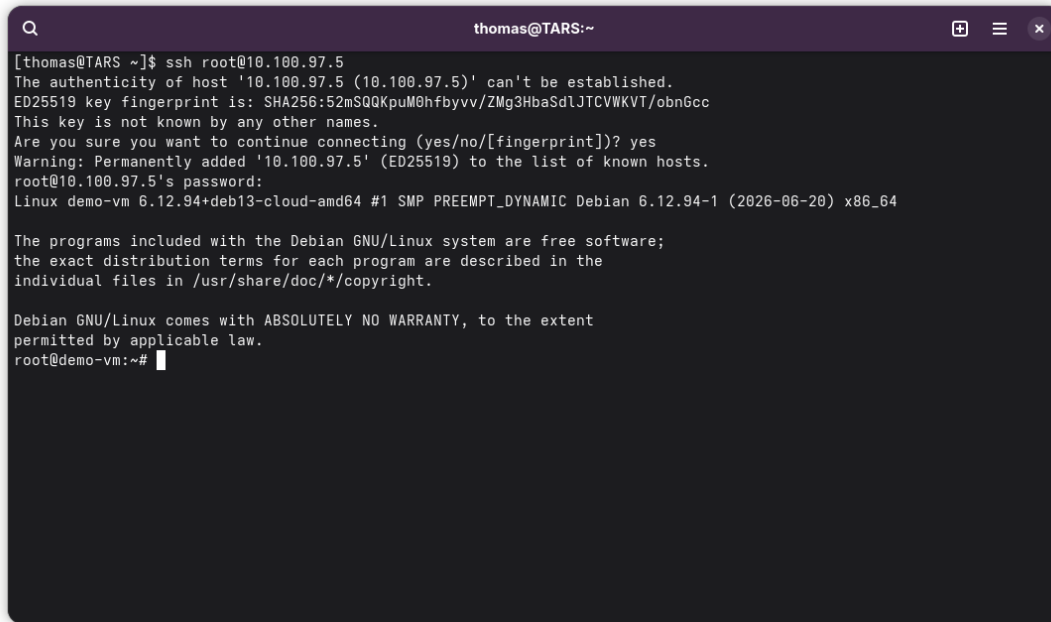
Figure 5.4: Real-time virtual machine card state progression in the self-service dashboard.

The interface reflects the instance lifecycle in real time:

- **Transitional Provisioning State:** The instance card displays a blue Provisioning badge, the assigned operating system (Debian 13), allocated resources (2 vCPU · 4 GB RAM, 20 GB Storage), creation timestamp, and the dynamically allocated static IPv4 address (`10.100.97.5`).
- **Terminal Running State:** When `VmProvisionWorker` completes the cloning, disk resizing, Cloud-Init injection, and power-on sequence, Phoenix PubSub broadcasts the completion event. The card switches dynamically to a green Running badge and enables the contextual power actions menu.

5.3.4 Remote Instance Connectivity Verification

Following successful provisioning, the virtual machine becomes directly reachable over the institutional network using standard protocols. Figure 5.5 illustrates an external terminal establishing an SSH session to the newly deployed instance.



```
thomas@TARS:~  
[thomas@TARS ~]$ ssh root@10.100.97.5  
The authenticity of host '10.100.97.5 (10.100.97.5)' can't be established.  
ED25519 key fingerprint is: SHA256:52mSQKpuM0hfbyvv/ZMg3HbaSdLJTCVWKVT/obnGoc  
This key is not known by any other names.  
Are you sure you want to continue connecting (yes/no/[fingerprint])? yes  
Warning: Permanently added '10.100.97.5' (ED25519) to the list of known hosts.  
root@10.100.97.5's password:  
Linux demo-vm 6.12.94+deb13-cloud-amd64 #1 SMP PREEMPT_DYNAMIC Debian 6.12.94-1 (2026-06-20) x86_64  
  
The programs included with the Debian GNU/Linux system are free software;  
the exact distribution terms for each program are described in the  
individual files in /usr/share/doc/*/copyright.  
  
Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent  
permitted by applicable law.  
root@demo-vm:~#
```

Figure 5.5: Terminal session establishing an authenticated SSH connection to `root@10.100.97.5`, confirming automated Cloud-Init network configuration, hostname injection, and guest OS initialization.

The successful terminal connection verifies several underlying infrastructure operations:

1. The assigned static IPv4 address (10.100.97.5) was injected into the guest network configuration and initialized by Cloud-Init upon boot.
2. The hostname was configured to `demo-vm`.
3. The root password injected during provisioning was accepted by the guest authentication subsystem.
4. The Linux kernel (Debian 6.12 SMP PREEMPT_DYNAMIC) initialized successfully on the underlying KVM hypervisor node.

5.4 Administrative Management Workflows

The administrator console provides a dedicated management interface partitioned into General Overview, User Management (Users, Roles), and Infrastructure Management (Clusters, Templates, Virtual Machines).

5.4.1 Cluster Registration and Hardware Node Telemetry

Figure 5.6 illustrates the 2-step cluster registration wizard, and Figure 5.7 shows the detailed cluster telemetry view.

(a) Step 1: Endpoint URL and API Token credentials.

Add cluster

1 Connect 2 Review & confirm

Node URL

Any single node in the cluster — the rest are discovered automatically.

API token ID

API token secret

(b) Step 2: Discovered nodes review and resource pool selection.

Add cluster

1 Connect 2 Review & confirm

Cluster name	Olympus
Nodes discovered	6
Nodes online	6 / 6

aris	online
poseidon	online
hephaestus	online
athena	online

Resource pool

Figure 5.6: Two-step Administrative Cluster Registration wizard.

The cluster management workflow operates across several key mechanisms:

1. **Two-Step Cluster Connection Wizard:** In Step 1, the administrator enters a single entry node URL (e.g., `https://10.0.0.10:8006`), the API token ID (`root@pam!platform`), and the secret. In Step 2, the platform queries the cluster status out-of-band, discovers the cluster name (e.g., "Olympus"), verifies online member nodes (e.g., `aris`, `poseidon`, `hephaestus`, `athena`), and allows the operator to bind a designated template resource pool (e.g., `IEE-Cloud`).
2. **Live Cluster Health and Aggregate Telemetry:** The cluster dashboard displays health indicators, total nodes (6 enabled), total hosted VMs (130 instances), total CPU capacity (464 cores

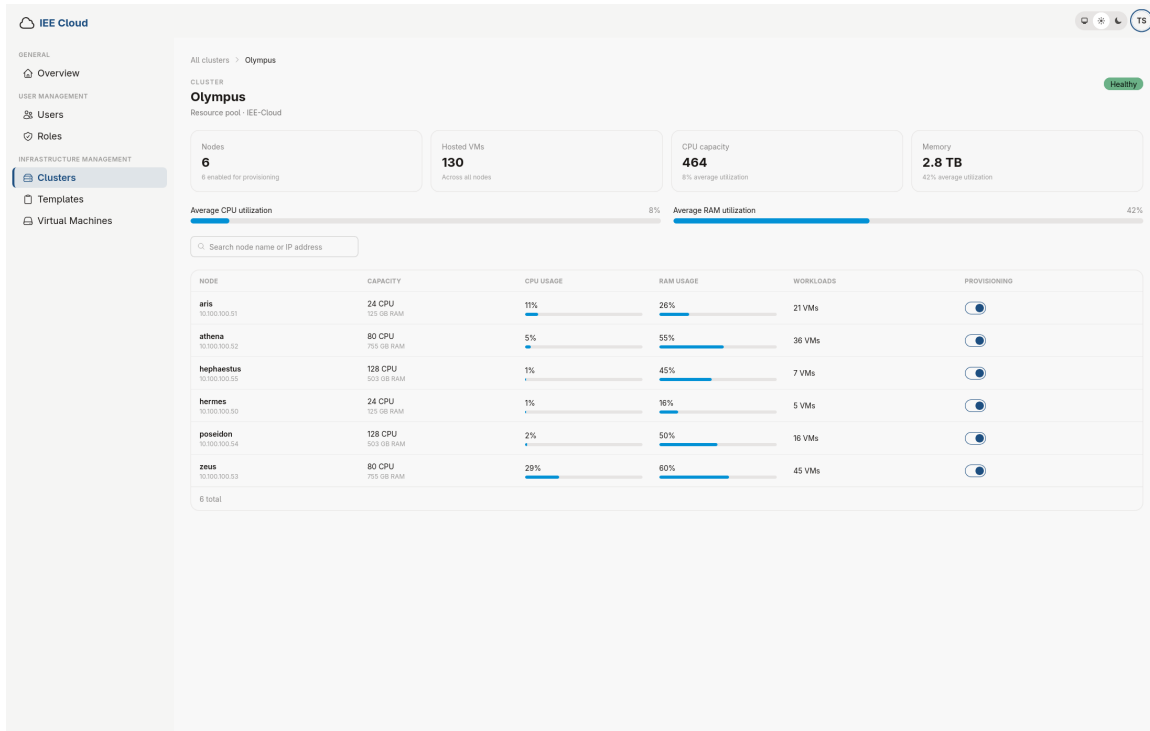


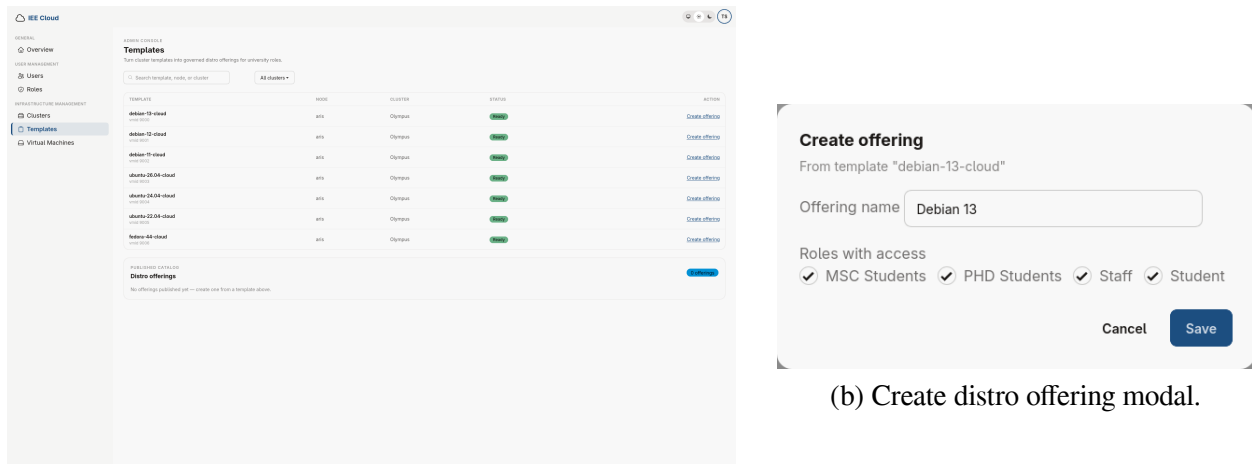
Figure 5.7: Cluster Telemetry and Node Management dashboard for the “Olympus” cluster, showing aggregate compute/memory capacity, live utilization bars, per-node workload metrics, and provisioning toggles.

at 8% average utilization), and cumulative RAM capacity (2.8 TB at 42% average utilization).

3. **Per-Node Workload Monitoring and Provisioning Toggles:** The node table presents per-host capacity, live CPU/RAM usage percentages, and hosted VM workloads across nodes (aris, athena, hephaestus, hermes, poseidon, zeus). Administrators can toggle the `allow_provisioning` switch on individual nodes to isolate hosts during maintenance without terminating active workloads.

5.4.2 Template Discovery and Distro Offering Governance

Figure 5.8 illustrates the template discovery table and the distro offering creation modal.



(a) Discovered hypervisor template images table.

(b) Create distro offering modal.

Figure 5.8: Operating system template discovery and distribution offering governance console.

The template subsystem bridges raw hypervisor disks with user offerings:

- **Automatic Template Discovery:** Discovered images within the designated resource pool are listed with their VMID (debian-13-cloud vmid 9000, debian-12-cloud vmid 9001, ubuntu-24.04-cloud vmid 9004, etc.), host node (aris), cluster ("Olympus"), and availability status.
- **Offering Creation and Role Scoping:** Clicking "Create offering" opens a configuration modal to define the user-facing title (e.g., "Debian 13") and select role access permissions via checkboxes ("MSC Students", "PHD Students", "Staff", "Student").

5.4.3 Academic Role Hierarchy and Quota Ceilings

Figure 5.9 illustrates the configured roles grid, while Figure 5.10 shows the creation and modification modals.

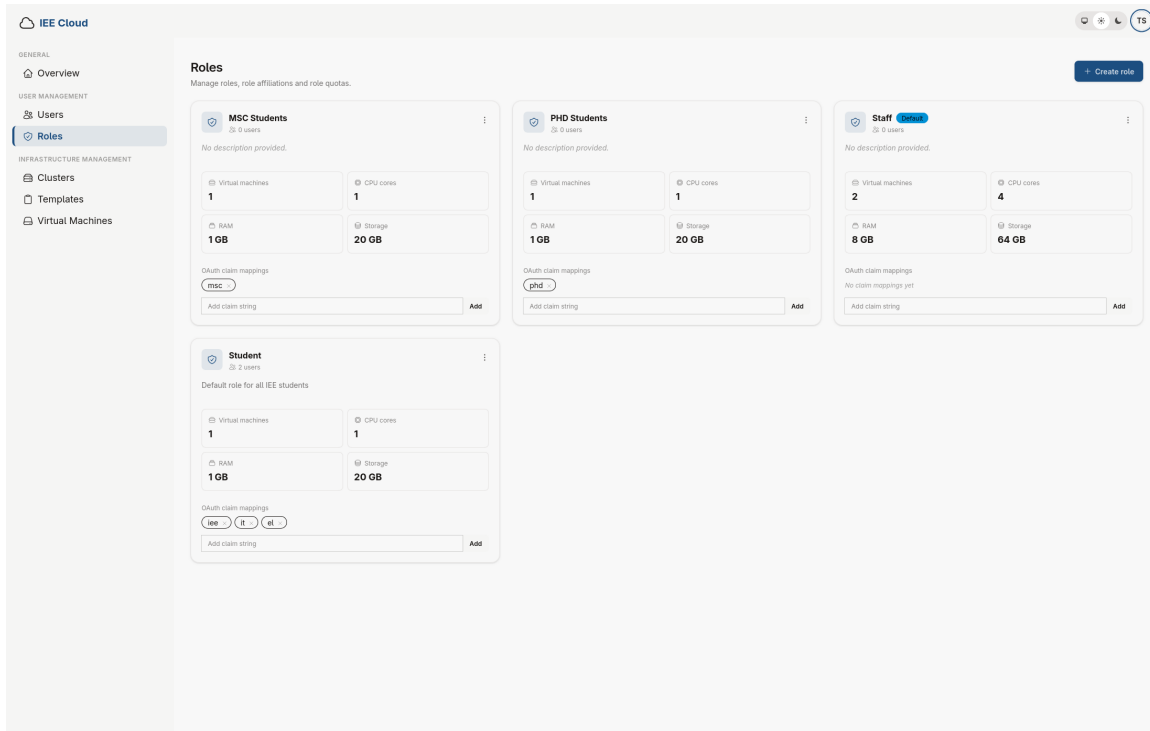


Figure 5.9: Academic Role Management dashboard, displaying configured role tiers, baseline quotas, default flags, and mapped OAuth claim tags.

Create new role

Define a new role and its standard limits.

Role Name
e.g. Student, Researcher

Description
e.g. Coursework and teaching lab access

QUOTA LIMITS

Max VMs: 1
Max CPU cores: 1
Max RAM (GB): 1
Max storage (GB): 20

Cancel Create role

Modify role

Update this role's name and description.

Role Name
Student

Description
Default role for all IEE students

QUOTA LIMITS

Max VMs: 1
Max CPU cores: 1
Max RAM (GB): 1
Max storage (GB): 20

Cancel Save changes

(a) Create new role modal.

(b) Modify role and baseline quota modal.

Figure 5.10: Modal dialogs for defining and adjusting academic roles and baseline quota envelopes.

The role subsystem governs multi-tenant hardware ceilings:

- **Configured Academic Tiers:** Displays cards for configured roles (e.g., “Student” with Default flag, “MSC Students”, “PHD Students”, “Staff”).
- **Baseline Quota Limits:** Defines standard limits for Max VMs, Max CPU cores, Max RAM (GB), and Max Storage (GB). For example, the default Student role is bounded to 1 VM, 1 CPU, 1 GB RAM, 20 GB Storage, while Staff is allocated 2 VMs, 4 CPUs, 8 GB RAM, 64 GB Storage.
- **OAuth Claim Mappings:** Lists mapped institutional claims (e.g., claims `ieee`, `it`, `el` map to Student; `msc` maps to MSC Students; `phd` maps to PHD Students).

5.4.4 User Directory, Role Reassignment, and Custom Quotas

Figure 5.11 illustrates the administrative users table alongside the role reassignment and custom quota override modals.

The user management console provides administrative controls over individual accounts:

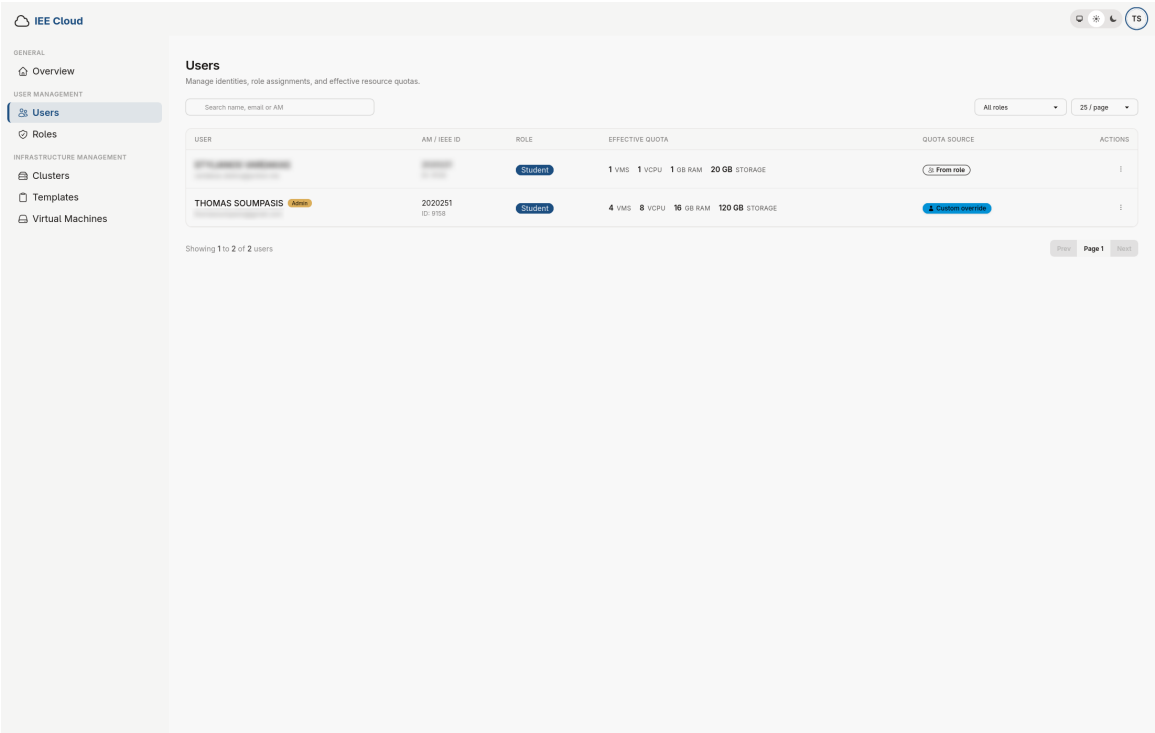
1. **User Table:** Details institutional email, administrative badge (Admin), academic ID (AM / IEEE ID), active role pill, effective quota, and quota source badge (“From role” or “Custom override”).
2. **Role Reassignment:** Allows administrators to change a user's assigned role via a dropdown dialog, instantly updating their baseline resource allowance.
3. **Granular Custom Quotas:** For research or special laboratory projects, administrators can configure a custom quota override (e.g., upgrading a specific user to 4 VMs, 8 vCPUs, 16 GB RAM, 120 GB Storage). The modal displays the base role quota and includes a “Clear custom quota” option to revert the account to standard limits.

5.4.5 Global Virtual Machine Fleet Supervision and Error Recovery

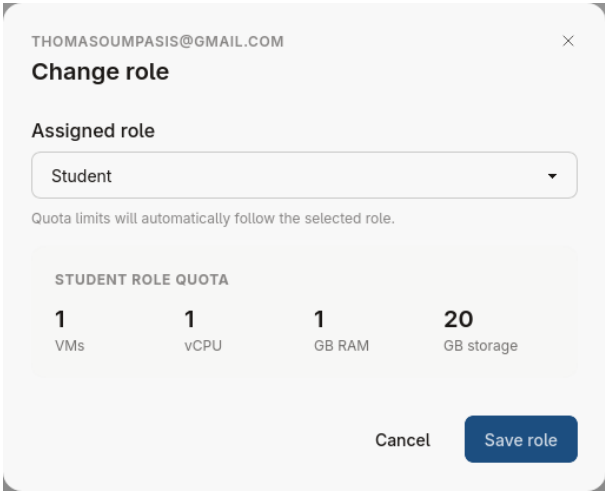
Figure 5.12 illustrates the global administrative Virtual Machines table and the diagnostic Unlock VM modal.

The global VM fleet view enables system-wide oversight and fault recovery:

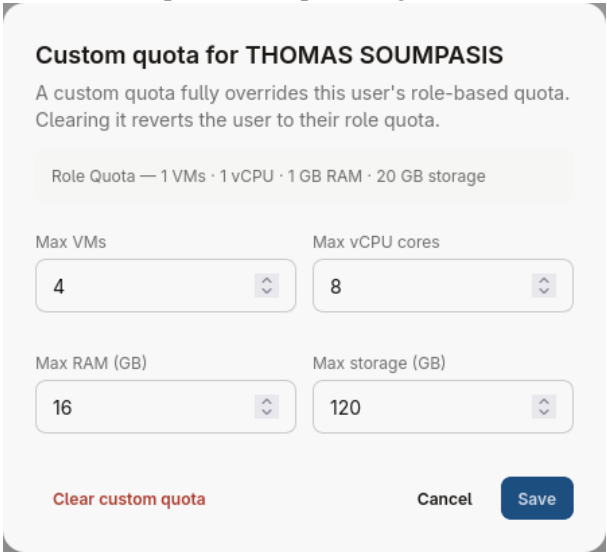
- **Global Fleet Supervision:** Lists all active virtual machines across all users, detailing instance name, hypervisor VMID (e.g., `vmid 209`), owner name and email, distro offering, parent cluster (“Olympus”), live execution state (Running), compute specs (2 vCPU · 4 GB · 20 GB), static IPv4 address (10.100.97.5), and contextual actions.



(a) User directory displaying institutional IDs, effective quotas, and quota origins.



(b) Role reassignment modal.



(c) Custom quota override modal.

Figure 5.11: User management console, role reassignment, and granular custom quota override dialogs.

(a) Global Virtual Machines fleet table. (b) Unlock VM error recovery modal.

Figure 5.12: Global virtual machine fleet supervision and administrative state unlock error recovery modal.

- **Administrative Error Unlocking:** If an instance enters a locked error state during a failed operation, administrators can open the Unlock dialog. The modal displays the exact hypervisor error message (e.g., `{:http_error, 500, %{ "message" => "can't remove VM 209 - protection mode enabled" }}`) and provides a dropdown allowing the operator to reset the record to a verified terminal state (e.g., `Stopped`).

5.5 Background Job Orchestration and Pipeline Execution

The platform delegates all long-running hypervisor operations to Oban background worker queues to preserve web server responsiveness and ensure operational resiliency.

5.5.1 The 7-Step Provisioning Pipeline (`VmProvisionWorker`)

When `VmProvisionWorker` picks up a job from the `:vm_provisioning` queue (concurrency limit: 5), it executes a sequential 7-step pipeline:

1. **Node Placement Selection:** Evaluates candidate nodes within the target cluster, filters hosts where `allow_provisioning = true` and cloning is supported, and selects the host minimizing total utilization via `UsageCache`:

$$\text{Target Node} = \arg \min_{n \in N_{\text{eligible}}} (\text{UsageCache}(n).\text{cpu_usage_percent} + \text{UsageCache}(n).\text{ram_usage_percent})$$

2. **ID and Static Network Acquisition:** Queries the Proxmox cluster for the next available VMID and calls `IpAllocator.allocate/0` to reserve the next available IPv4 address within a PostgreSQL transaction protected by an advisory lock.
3. **Template Cloning:** Dispatches an asynchronous full clone request for the selected template to the Proxmox REST API, records the returned task UPID in PostgreSQL, and blocks until the hypervisor confirms completion via `wait_for_task/4`.
4. **Storage Volume Expansion:** Resizes virtual disk slot `scsi0` to the user-requested storage capacity.
5. **Cloud-Init Configuration Injection:** Formulates the Cloud-Init payload, setting CPU cores, memory size, root user credentials, and static IP configuration (`ip=10.100.97.5/24, gw=10.100.97.`
6. **Cloud-Init ISO Regeneration:** Rewrites the Cloud-Init drive definition (`ide2`) on the hypervisor to trigger the rebuild of the bootable ISO image.
7. **Power-On and Password Scrubbing:** Issues the startup command to Proxmox, transitions the database state to `:running`, and executes an SQL scrubbing query to purge the plaintext password from the `oban_jobs` table.

5.5.2 Power Lifecycle State Dispatch (`VmLifecycleWorker`)

Instance lifecycle operations (Start, Stop, Shutdown, Reboot, Restart) execute in the `:vm_lifecycle` queue. The worker resolves the action function pointer, dispatches the HTTP mutation to Proxmox, tracks the active task UPID, and settles the database record into its terminal state (`:running` or `:stopped`).

Table 5.1 summarizes the operational behaviors and badge transitions for each lifecycle action.

Table 5.1: Summary of Virtual Machine Lifecycle Operations and Interface Behaviors.

Action	Initial State	Transitional Badge	Operational Behavior
Start	<code>:stopped</code>	STARTING	Dispatches start command to Proxmox and waits for QEMU startup.
Stop	<code>:running</code>	STOPPING	Issues hard power-off command to immediately terminate guest execution.
Shutdown	<code>:running</code>	SHUTTING_DOWN	Sends ACPI shutdown signal to guest OS for graceful filesystem unmounting.
Reboot	<code>:running</code>	REBOOTING	Issues graceful ACPI reboot signal to restart the guest operating system.
Restart	<code>:running</code>	RESETTING	Issues hard hardware reset command to force immediate instance reboot.
Delete	<code>:stopped</code> , <code>:locked</code>	DELETING	Destroys hypervisor VM, purges disk volumes, and recycles assigned IPv4 address.

5.6 Qualitative System Verification and Operational Reliability

System correctness and resilience were verified through structural qualitative evaluation across five core operational criteria:

5.6.1 Quota Boundary Enforcement and Real-Time Guarding

- **Operational Condition:** A user with an active quota of 8 vCPUs and 2 vCPUs already consumed attempts to configure a new VM requiring 10 vCPUs.
- **Observed Behavior:** LiveView performs real-time client-side calculation, displays the amber quota warning banner, and disables the submission button. When tested against direct API submissions, the Ash domain engine evaluates `WithinQuota`, aborting the transaction before any database record or Oban job is staged.

5.6.2 Deterministic Static Network Allocation

- **Operational Condition:** Multiple virtual machines are created concurrently within the `:vm_provisioning` queue.
- **Observed Behavior:** `IpAllocator` acquires a PostgreSQL transactional advisory lock, queries assigned IP addresses, and sequentially selects the next free host offset in `10.100.97.0/24` (assigning `10.100.97.5`, `10.100.97.6`, etc.). Zero address collisions occurred across all provisioning cycles.

5.6.3 Cloud-Init Parameter Injection and Network Reachability

- **Operational Condition:** Provisioning a virtual machine from a Debian 13 base template with custom root credentials and static IP parameters.
- **Observed Behavior:** Upon boot, Cloud-Init applies the static network lease, configures the gateway, and initializes root access. The instance responded to network ICMP pings and accepted authenticated SSH connections on port 22.

5.6.4 Hypervisor Fault Isolation and Administrative Error Recovery

- **Operational Condition:** Simulating an external hypervisor error during instance deletion (e.g., hypervisor protection mode enabled).
- **Observed Behavior:** `VmLifecycleWorker` catches the error returned by Proxmox and invokes `VirtualMachine.lock_with_error/3`. The record transitions to state `:locked_error`, storing the error diagnostic string `({:http_error, 500, %{"message" => "can't remove VM 209 - protection mode enabled"}})` in `error_details`. The administrator uses the `Unlock` modal to inspect the diagnostic message and reset the VM state to `Stopped`.

5.6.5 Transient Credential Sanitization

- **Operational Condition:** Querying the persistent PostgreSQL `oban_jobs` table following completed provisioning tasks.
- **Observed Behavior:** Direct SQL inspection confirmed that while operational metadata (`vm_id`, node name, queue) remained recorded for audit purposes, the plaintext `"password"` key was stripped from the JSONB `args` column across all completed and discarded jobs.

5.7 Chapter Summary

This chapter presented the operational workflows, user interface interactions, asynchronous background job execution, and qualitative reliability verification of the platform.

We walked through the institutional OAuth 2.0 onboarding sequence, the self-service student overview dashboard, dynamic quota validation within the VM creation wizard, and remote instance access verification via SSH. We detailed the administrative control console, covering the 2-step cluster connection wizard, node telemetry inspection, template cataloging, role hierarchy governance, custom quota overrides, and global VM fleet error recovery. Finally, we analyzed the asynchronous background pipelines in `VmProvisionWorker` and `VmLifecycleWorker`, and verified system reliability across quota boundaries, static network allocation, Cloud-Init reachability, hypervisor error recovery, and credential scrubbing.

With platform operations and system verification established, Chapter 6 presents the summary of technical contributions, architectural conclusions, and future extensions.

Chapter 6

Conclusion

6.1 Summary

This diploma thesis presented the complete architectural design, domain modeling, implementation, and qualitative verification of a lightweight, reactive, multi-tenant web platform engineered for academic hypervisor cluster orchestration. The project was developed to directly resolve the operational bottlenecks and administrative overhead associated with computing resource provisioning in higher education. Historically, academic computer science and engineering laboratories have relied either on manually maintained physical workstations or on manual administrative ticketing workflows to provision virtual machines. These legacy methods introduce significant latency, waste valuable administrative labor, and restrict student access. Conversely, exposing the native administration dashboards of enterprise hypervisors like Proxmox (PVE) directly introduces unacceptable security vulnerabilities, lacks granular multi-tenancy and fails to enforce dynamic quotas.

To address these challenges, the platform was built as a dedicated middleware control plane that bridges untrusted academic users with backend physical virtualization clusters. The application was implemented in the functional Elixir programming language on the Erlang Virtual Machine (BEAM) runtime, combining the declarative domain modeling of the Ash Framework, real-time WebSocket interfaces via Phoenix LiveView, and durable database-backed background job queuing using Oban.

The primary technical achievements delivered by the completed platform include:

- **Institutional OAuth 2.0 Integration**
- **Automated Academic Role and Quota Mapping**
- **Multi-Cluster Infrastructure Orchestration**
- **Curated Distribution Offering Abstraction**
- **Automated Multi-step Asynchronous Provisioning Pipeline**

- **Fault-Isolated Power Lifecycle State Machine**
- **Reactive Real-Time User Interfaces**

6.2 Conclusions

The design, implementation, and qualitative verification of the platform yield several significant architectural and practical conclusions regarding cloud infrastructure in educational environment:

First, the project demonstrates that managing academic virtualization clusters does not necessitate deploying heavy, complex enterprise cloud suites such as OpenStack. While enterprise platforms provide expansive features sets, their substantial hardware footprint, complex multi-node control planes, and demanding maintenance requirements make them impractical for small-to-medium university departments. A focused, lightweight middleware gateway executing on modern web runtimes can successfully provide robust multi-tenancy, granular access control, dynamic hardware quotas, and automated self-service provisioning over existing hypervisor clusters with negligible resource overhead.

Second, from a software engineering perspective, the Erlang Virtual Machine (BEAM) and the Elixir ecosystem proved to be exceptionally well-suited for hardware virtualization control planes. Infrastructure lifecycle tasks such as cloning disk volumes or waiting for guest operating systems to initialize are inherently slow, asynchronous, and prone to transient network latency. The BEAM's Actor Model, lightweight processes, and preemptive reduction scheduling ensure that hundreds of concurrent background hypervisor tasks can execute without starving web worker threads or degrading user interface responsiveness. Furthermore, the runtime's supervision tree architecture guarantees absolute fault isolation, ensuring that an unexpected hypervisor timeout or network failure in one student's provisioning task is cleanly handled without affecting the stability of the broader system.

Third, the adoption of the Ash Framework validated the advantages of declarative domain modeling over traditional imperative web architectures. Embedding Role-Based Access Control (RBAC) policies, academic affiliation mappings, dynamic quota validations, and state machine guards directly into the resource layer ensures uniform security enforcement. Whether an infrastructure mutation is triggered by a student clicking an action on a LiveView dashboard or by an automated background worker executing a scheduled task, the domain policies are evaluated identically at the data entry boundary, completely eliminating privilege escalation risks and data leaks caused by omitted controller guards.

Fourth, delegating long-running and short-lived infrastructure operations to persistent, database-backed Oban queues established operational durability and performance isolation. Separating slow disk-cloning operations into the `:vm_provisioning` queue while routing fast power commands to the `:vm_lifecycle` queue prevents slow jobs from blocking quick administrative interactions. Implementing database credential scrubbing immediately following Cloud-Init injection demonstrated that transient credentials can be utilized securely without lingering in persistent job logs or database backup archives.

6.3 Future Extensions

While the completed platform delivers a robust, production-ready foundation for academic virtual machine management, the system's modular domain architecture opens several promising possibilities for future technical expansion

6.3.1 Embedded In-Browser Web Console Access (noVNC and SPICE)

In the current implementation, users access their provisioned virtual machines over the network via SSH or remote desktop protocols using the assigned static IP address. A valuable immediate enhancement is the integration of an interactive, in-browser graphical and terminal console. By establishing secure, token-authenticated WebSocket proxy tunnels between Phoenix LiveView and Proxmox's native `noVNC` or `SPICE` endpoints, students could access an interactive command-line shell or graphical desktop interface directly within their web browser. This would enable seamless terminal interaction without requiring students to install local SSH clients, configure firewall port forwards, or expose virtual machine ports to public networks.

6.3.2 Multi-Subnet IPAM Integration and VLAN Pool Management

The current networking layer operates over a single configured institutional subnet, assigning static host addresses sequentially. To accommodate larger campus environments spanning multiple departments and physical server rooms, future versions could expand the network engine into a full multi-subnet IP Address Management (IPAM) layer. This would allow administrators to define multiple IP address pools, associate subnets with specific VLAN tags and cluster nodes, and automatically manage IP recycling across distinct departmental network segments.

6.3.3 Automated Idle Resource Reclamation and Power Scheduling

In academic environments, provisioned virtual machines are often left running continuously during nights, weekends, and holiday breaks, consuming unnecessary electrical power and host memory. A valuable operational extension would be the implementation of an automated resource reclamation engine. By tracking live CPU and network I/O activity via Proxmox cluster telemetry, the platform could identify idle virtual machines, send automated notification warnings to students, and automatically suspend or power down inactive instances.

6.3.4 Multi-Service Cloud Expansion (DBaaS and Object Storage)

While this thesis focused strictly on the full automated lifecycle management of virtual compute instances, the underlying Ash domain architecture was intentionally designed to support multi-service

cloud expansion. Future development could introduce dedicated Database-as-a-Service (DBaaS) modules (providing on-demand PostgreSQL and MySQL instances) and S3-compatible Object Storage bucket management (abstracting backend storage clusters such as Ceph or MinIO), transforming the platform into a comprehensive, multi-tiered academic cloud computing environment.

Appendix A

Modernization of Departmental Infrastructure

A.1 Context and Previous State

The software orchestration capabilities detailed in the previous chapters required a robust, highly available physical foundation. Prior to this collaborative initiative, the department's server room operated on legacy infrastructure. The environment was largely unmaintained and consisted of a handful of aging servers, constrained by 1 Gbps network connections and HDD-based storage. This hardware could not reliably support the concurrent virtualized workloads required in a modern academic setting.

A.2 Infrastructure Overhaul and Modernization

To support the new self-service cloud portal, the server room was entirely re-architected to prioritize redundancy, performance, and disaster recovery. The modernization effort included the following key improvements:

- **Compute Virtualization:** The legacy server were replaced with a robust 6-node Proxmox cluster consisting of modern hardware, providing the compute capacity and high-availability features required.
- **Storage Area Network (SAN):** The storage backend was upgraded from solely local storage to highly available iSCSI SANs with proper multipathing.
- **Dedicated Network Segmentation:** To eliminate bottlenecks, network traffic was distributed across bonded 10Gbps interfaces. Storage, virtual machine traffic, cluster quorum, backup, and management traffic were explicitly isolated onto separate interfaces. This allowed the storage

network to utilize jumbo frames as a dedicated, high-throughput path, while ensuring the Proxmox cluster's quorum mechanism met its strict low latency requirements on its own isolated links.

- **Core Services and Security:** A highly available pfSense firewall configuration was deployed at the edge. Centralized DNS and DHCP were established alongside strict VLAN segmentation to isolate administrative traffic from regular user workloads.
- **Backup and Disaster Recovery:** Two dedicated Proxmox Backup Server (PBS) were integrated into the environment, featuring automated scheduling tiered by virtual machine priority. Furthermore, comprehensive documentation detailing the entire architecture and rebuild procedures was authored to guarantee operational continuity.

A.3 Storage Area Network (SAN) Architecture

The SAN backend consists of an HPE MSA-2060 dual-controller array, connected to two dedicated Cisco switches (Switch-A and Switch-B) which in turn connect to the iSCSI network interfaces of each Proxmox node, as illustrated in the cabling diagram (Figure A.1). Each node maintains independent physical paths to both controllers, giving four paths per LUN per node once both switches and both controllers are accounted for.

A.3.1 Multipath I/O Configuration

Path redundancy and failover are managed at the operating system level using Linux `multipathd`, configured as follows:

```
defaults {
    user_friendly_names yes
    find_multipaths yes
    flush_on_last_del always
    no_path_retry queue
    polling_interval 2
}

devices {
    device {
        vendor "HP"
        product "MSA2060"
        path_grouping_policy "multibus"
        path_selector "round-robin 0"
        failback "immediate"
        path_checker "readsector"
```

```
        features "1 queue_if_no_path"
        dev_loss_tmo 60
    }
}

blacklist {
    wwid .*
}

blacklist_exceptions {
    wwid "3600c0ff000faee3796a5f66701000000"
}

multipaths {
    multipath {
        wwid "3600c0ff000faee3796a5f66701000000"
        alias HPE-MSA2060
    }
}
```

Several settings in this configuration reflect deliberate design choices rather than defaults:

- `no_path_retry queue` instructs the kernel to queue I/O rather than fail it immediately if all paths are lost, which is appropriate for a storage-backed hypervisor cluster where a transient path outage should not cause guest I/O errors.
- The explicit `blacklist/blacklist_exceptions` pairing blacklists all devices by default and then whitelists only the MSA's WWID, preventing local disks or unrelated LUNs from being accidentally pulled into multipath management.
- `dev_loss_tmo 60` gives a 60-second grace period before a path is considered permanently lost, balancing fast failover against tolerance for brief switch-level interruptions (e.g. during a switch reload).

Verification and ALUA Behavior

The output of `multipath -ll` confirms that all eight paths to the array are active and healthy, and illustrates the array's use of ALUA (Asymmetric Logical Unit Access):

```
HPE-MSA2060 (3600c0ff000faee3796a5f66701000000) dm-77 HPE,MSA 2060 iSCSI
size=7.0T features='1 queue_if_no_path' hwhandler='1 alua' wp=rw
|+- policy='service-time 0' prio=50 status=active
|  |- 4:0:0:1 sdf 8:80  active ready running
```

```
| |- 7:0:0:1 sdg 8:96  active ready running
| |- 9:0:0:1 sdj 8:144 active ready running
| `-- 5:0:0:1 sdk 8:160 active ready running
`-+- policy='service-time 0' prio=10 status=enabled
    |- 3:0:0:1 sde 8:64  active ready running
    |- 6:0:0:1 sdh 8:112 active ready running
    |- 8:0:0:1 sdi 8:128 active ready running
    `-- 2:0:0:1 sdd 8:48  active ready running
```

The two priority groups are not a fault condition but the expected behavior of a dual-controller array under ALUA. The higher-priority group (`prio=50`) contains the four paths that reach the controller currently owning the volume, and is used for all active I/O. The lower-priority group (`prio=10`) contains the four paths that reach the array through its partner controller; these remain enabled and ready, and are only promoted to active use if the owning controller or its paths become unavailable. This confirms that the "proper multipathing" referenced in Section 2 is implemented not only at the cabling level, but is correctly recognized and managed by the host operating system.

A.3.2 iSCSI Switch Configuration

The two switches dedicated to iSCSI traffic required storage-specific configuration beyond a standard Layer 2 setup, covering jumbo frames, flow control, and port documentation. The following configuration was applied identically to both switches unless otherwise noted.

Jumbo Frames

The MTU was raised to accommodate 9000-byte jumbo frames end-to-end, matching the MTU configured on the Proxmox iSCSI interfaces (Section A.4). Where the switch platform supports a global MTU setting, it was applied as follows:

```
conf t
system mtu 9216
end
reload
```

Note that a global MTU change on Cisco IOS typically requires a reload to take effect, and the value configured here includes Layer 2 overhead; a system MTU of 9216 bytes comfortably accommodates a 9000-byte IP payload.

Flow Control

Flow control was enabled to reduce the likelihood of dropped frames during bursts of storage I/O:

```

conf t
interface range TenGigabitEthernet1/0/1 - 4
    flowcontrol receive on
end

```

Flow control was matched to the capabilities of the connected host NICs; where a Proxmox NIC only supports receiving pause frames, the switch side was configured symmetrically to avoid a mismatched negotiation.

Port Documentation

Each SFP+ port was labeled with its connected device and role, both to aid troubleshooting and to keep the physical topology traceable against the cabling diagram:

```

interface TenGigabitEthernet1/0/1
    description Proxmox01 NIC1 (iSCSI-A)

```

This was repeated for every port connected to a Proxmox iSCSI interface and every MSA 2060 iSCSI port.

A.4 Proxmox Host Network Interface Design

Each Proxmox node is configured with three physically independent NIC pairs, deliberately isolating traffic classes so that latency-sensitive and throughput-sensitive workloads never contend for the same link. The relevant portion of `/etc/network/interfaces` is summarized below.

Interface(s)	Bridge	Role
eno1	vmbr0	Proxmox management (static IP + gateway)
eno2	vmbr1	Corosync ring 0
eno3	vmbr2	Corosync ring 1 (redundant ring)
eno4	vmbr3	Proxmox Backup Server traffic (dedicated VLAN)
eno5np0	---	iSCSI path A (10.10.10.0/24, MTU 9000)
eno6np1	---	iSCSI path B (10.10.20.0/24, MTU 9000)
ens3f0np0, ens3f1np1	vmbr999 (via bond0)	VM traffic (LACP, VLAN-aware trunk)

Table A.1: Physical-to-logical interface mapping per Proxmox node

```

auto eno5np0
iface eno5np0 inet static

```

```
        address 10.10.10.12/24
        mtu 9000
#ISCSI-1

auto eno6np1
iface eno6np1 inet static
        address 10.10.20.12/24
        mtu 9000
#ISCSI-2

auto bond0
iface bond0 inet manual
        bond-slaves ens3f0np0 ens3f1np1
        bond-miimon 100
        bond-mode 802.3ad
        bond-xmit-hash-policy layer2

auto vmbr999
iface vmbr999 inet manual
        bridge-ports bond0
        bridge-stp off
        bridge-fd 0
        bridge-vlan-aware yes
        bridge-vids 2-4094
#VM-Traffic
```

Two design decisions in this layout are worth highlighting explicitly, as they reflect an understanding of why generic "bond everything" advice does not apply uniformly:

- **The iSCSI interfaces are intentionally left unbonded.** Each iSCSI NIC sits on its own subnet (10.10.10.0/24 and 10.10.20.0/24) rather than being combined into a single bonded interface. Path redundancy is instead delegated entirely to Linux `multipathd`. Bonding the iSCSI NICs would introduce a second, independent failover mechanism operating underneath `multipath`, which can mask path failures from `multipathd` or cause it to make decisions based on an inconsistent view of link state. Keeping the two paths separate and letting `multipath` own failover is the standard recommended practice for iSCSI MPIO.
- **VM traffic uses LACP bonding with a VLAN-aware trunk.** Unlike the iSCSI links, VM traffic benefits from both throughput aggregation and switch-side link redundancy, which 802.3ad (LACP) with a `layer2` hash policy provides. The resulting bridge (`vmbr999`) is VLAN-aware across the full range (2-4094), allowing individual VMs to be assigned to any tenant or workload VLAN without requiring a dedicated bridge per VLAN.

A.5 Cluster Quorum Considerations

Corosync, which governs cluster membership and quorum for the 6-node Proxmox cluster, is configured to use two independent physical rings (`eno2/vmbr1` and `eno3/vmbr2|`), fully isolated from iSCSI, VM, backup, and management traffic. This isolation is a direct response to corosync's sensitivity to latency and jitter: because cluster quorum decisions and fencing are time-bounded, any contention on the corosync links from unrelated bulk traffic (for example, a large VM migration or a backup job) could risk spurious quorum loss. Dedication of two physical NICs solely to corosync, rather than sharing them with any other bridge, removes this risk at the network layer rather than relying on QoS alone. Ring-level tuning values (e.g. totem timers) are not reproduced here, as they fall outside the scope of the interface design covered in this appendix.

A.6 Core Network Services and Security

A highly available pfSense firewall pair was deployed at the network edge to avoid a single point of failure for routing, NAT, and perimeter filtering. Centralized DNS and DHCP services were consolidated to simplify address management across the growing number of VLANs. Strict VLAN segmentation separates administrative and infrastructure traffic (management interfaces, iSCSI, corosync, backup) from regular user and tenant workloads, so that a compromised or misbehaving user VM cannot directly reach infrastructure-plane interfaces.

A.7 Backup and Disaster Recovery

Two Proxmox Backup Server (PBS) instances were deployed to provide redundant backup targets. Backup traffic from each Proxmox node is routed over a NIC dedicated exclusively to this purpose (`eno4/vmbr3`), which terminates on a VLAN reserved solely for PBS traffic. This isolates backup jobs at both the host and switch level: scheduled backups cannot compete for bandwidth with production storage or VM traffic, and, conversely, a backup window cannot be disrupted by unrelated network load elsewhere in the environment. Backup scheduling is tiered by virtual machine priority, ensuring that critical workloads are protected more frequently than lower-priority ones.

A.8 Documentation and Operational Continuity

To guarantee that the environment can be understood, maintained, and rebuilt by future administrators, comprehensive documentation was authored alongside the infrastructure itself. This includes:

- Physical cabling diagrams for server-to-switch and switch-to-storage connections (see Figure A.1).

- Switch configuration procedures for iSCSI-specific settings (jumbo frames, flow control, port labeling).
- Host-level network configuration for every traffic class (storage, VM, quorum, backup, management).
- Multipath and storage configuration required to reproduce the SAN setup on replacement hardware.

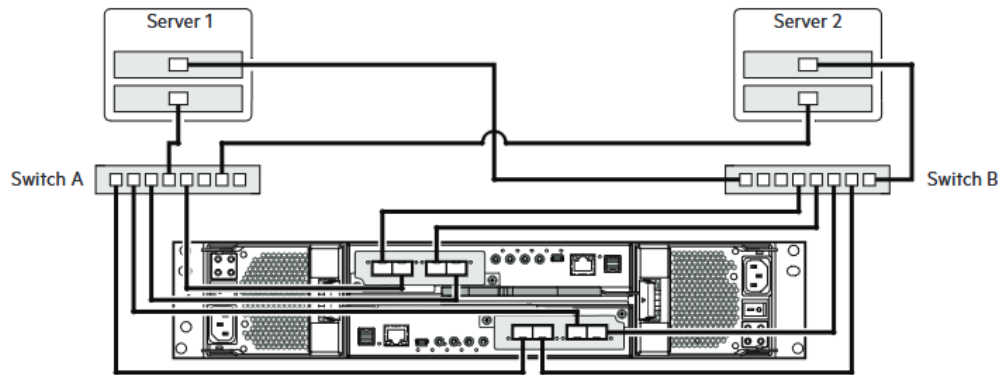


Figure A.1: iSCSI cabling topology: dual-server, dual-switch connectivity to the HPE MSA 2060 array.

A.9 Summary

The physical and network architecture described in this appendix directly enabled the orchestration platform discussed in the preceding chapters. By isolating storage, VM, quorum, backup, and management traffic onto independent physical or logical paths, and by validating that redundancy is genuinely realized at both the storage layer (ALUA multipathing) and the network layer (switch pairing, LACP bonding), the modernized server room provides a stable foundation on which automated provisioning pipelines and role-based quotas can execute safely and at scale.

Bibliography

- [1] G. J. Popek and R. P. Goldberg, "Formal requirements for virtualizable third generation architectures," *Communications of the ACM*, vol. 17, pp. 412--421, July 1974.
- [2] Proxmox Server Solutions GmbH, "Proxmox VE Documentation." <https://pve.proxmox.com/pve-docs/>, 2026. Accessed: Sep. 6, 2026.
- [3] Proxmox Server Solutions GmbH, "Proxmox VE API Viewer." <https://pve.proxmox.com/pve-docs/api-viewer/index.html>, 2026. Accessed: Sep. 6, 2026.
- [4] OpenStack Foundation, "Openstack architecture overview," 2026. Accessed: Sep. 6, 2026.
- [5] OpenStack Foundation, "Horizon: The openstack dashboard project," 2026. Accessed: Sep. 6, 2026.
- [6] OpenStack Foundation, "Keystone: Openstack identity service," 2026. Accessed: Sep. 6, 2026.
- [7] D. E. Hardt, "The oauth 2.0 authorization framework," RFC 6749, IETF, Oct. 2012.
- [8] Elixir Core Team, "Elixir Programming Language." <https://elixir-lang.org/>, 2026. Accessed: Sep. 6, 2026.
- [9] Erlang/OTP Team, "Erlang/OTP Documentation." <https://www.erlang.org/docs>, 2026. Accessed: Sep. 6, 2026.
- [10] Ash Framework Team, "Ash Framework Documentation." <https://www.ash-hq.org/>, 2026. Accessed: Sep. 6, 2026.
- [11] Phoenix Framework Team, "Phoenix Framework Documentation." <https://www.phoenixframework.org/>, 2026. Accessed: Sep. 6, 2026.
- [12] Oban.pro, "Oban --- Reliable, Extensible Job Processing for Elixir." <https://oban.pro>, 2026. Accessed: Sep. 6, 2026.
- [13] The PostgreSQL Global Development Group, "PostgreSQL Documentation." <https://www.postgresql.org/docs/>, 2026. Accessed: Sep. 6, 2026.
- [14] Ecto (Elixir) Core Team, "Ecto Documentation." <https://hexdocs.pm/ecto/>, 2026. Accessed: Sep. 6, 2026.