Democratic and People's Republic of Algeria
الجمهورية الجزائرية الديمقراطية الشعبية
Ministry of Higher Education and Scientific Research
وزارة التعليم العالي و البحث العلمي

جامعة العلوم والتكنولوجيا هواري بومدين
University Of Science And Technology Houari Boumediene

Faculty of Informatics

Thesis for graduation

for the obtainment of the Master 2 degree

Option : Network and Distributed Systems

Design and implementation of an automatic deployment and provisioning solution on physical clusters

Realized by :
Mr.SALMI Wail Abderrahim
Mr.BENMEDDOUR Mohamed Nafae

Supervised by :
Mr.ZERAOULIA Khaled
Mrs.BENZAID Chafika

Held on 02 July 2025, before the jury composed of :

Mr.HAMMAL Youcef : - President of jury
Mr.DERDER Abdessamed : - Examiner

Promotion : 2024/2025
Code PFE : RSD-30

Dedication

“

First of all, Al-hamdoulillah
I dedicate this project:
To my dear mother,
To my dear father,

Who have never stopped praying for me, supporting me, and standing by my side so I could achieve my goals.

To my brother, Oussama,
To my dear sisters,

For their moral support and their valuable advice throughout my studies.

To my friend Mustafa,
To my friends from the university residence and “el-jama3,”
To my university friends,

And to all those who helped us accomplish this project,

Thank you.

”

Salmi Wail Abderrahim

“

First of all, I thank God for guiding and supporting me throughout my academic journey and in all things.
It is with great pleasure that I dedicate this modest work:
To my father and mother,
To my brothers and sisters,
To my friends,
To my professors,
And to all those who helped me accomplish this project.

Thank you.

”

Benmeddour Mohamed Nafea

Acknowledgements

First and foremost, we thank Allah, the Almighty, for granting us the courage and patience necessary to bring this work to completion.

We wish to express our deep gratitude to our supervisor, **Mr. ZERAOULIA Khaled**, as well as to **Mrs. BENZAID Chafika**, for their valuable assistance, patience, and encouragement. Their critical insights were invaluable in structuring this work and improving the quality of various aspects of our project.

We would also like to extend our sincere thanks to the members of the jury for the honor they do us by taking the time to read and evaluate our work.

We also wish to thank the academic and administrative staff of **USTHB** for their efforts in providing us with a high-quality education.

Finally, we thank all those who contributed, directly or indirectly, to the completion of this work.

Abstract

Modern cloud environments demand dynamic resource management, yet a significant gap remains between high-level human intent and the low-level code required for infrastructure deployment. This thesis addresses that challenge by presenting the design and implementation of an autonomic system that translates natural language commands into fully automated provisioning actions on a physical cluster.

The proposed solution is built on a multi-layered architecture. It begins with a hyperconverged foundation using Proxmox VE for virtualization and Ceph for distributed storage. On top of this resilient base, a high-availability Kubernetes cluster (k3s) is deployed and managed through an Infrastructure as Code (IaC) workflow. Terraform is used for provisioning, while Ansible handles configuration.

The core innovation lies in the Agentic AI Layer, which integrates a Multi-Agent System (MAS) powered by Generative AI. This intelligent layer interprets user intent, analyzes system state, and dynamically generates the appropriate deployment code.

The platform was successfully validated through practical scenarios, including the automated provisioning of a virtual machine from a natural language command. This confirmed the viability of the architecture and demonstrated that integrating a Generative AI-driven MAS with a fully automated infrastructure stack can effectively bridge the gap between human intent and infrastructure execution—paving the way for more intelligent, resilient, and self-managing cloud environments.

Keywords : Cloud provisioning, Infrastructure as Code (IaC), Proxmox VE, Ceph, Kubernetes (k3s), Multi-Agent System (MAS), Generative AI, natural language processing, virtualization, autonomous infrastructure.

Contents

List of Figures

List of Tables

List of abbreviations and acronyms

ACO	*Ant Colony Optimization*
ADE	*Adaptive Differential Evolution*
ADK	*Agent Development Kit*
AMD-V	*AMD Virtualization*
ANN	*Artificial Neural Network*
API	*Application Programming Interface*
ARP	*Address Resolution Protocol*
AWS	*Amazon Web Services*
BG-LSTM	*BiLSTM and GridLSTM*
BGP	*Border Gateway Protocol*
CBRS	*Content-based Recommender System*
CephFS	*Ceph File System*
CI/CD	*Continuous Integration/Continuous Deployment*
CILP	*Co-simulation–Based Imitation Learner*
CNN	*Convolutional Neural Network*
CPU	*Central Processing Unit*

CRD	*Custom Resource Definition*
CRI-O	*Container Runtime Interface - Open*
CRS	*Collaborative Recommender System*
CSI	*Container Storage Interface*
DHCP	*Dynamic Host Configuration Protocol*
DL	*Deep Learning*
DNS	*Domain Name System*
DQN	*Deep Q-Network*
DRS	*Distributed Resource Scheduler*
DSL	*Domain Specific Language*
DT	*Decision Tree*
ECS	*Elastic Container Service*
ESXi	*VMware vSphere Hypervisor*
ETCD	*Distributed Key-Value Store*
ETL	*Extract, Transform, Load*
EULA	*End User License Agreement*
FQDN	*Fully Qualified Domain Name*
GA	*Genetic Algorithm*
GAN	*Generative Adversarial Network*
GCD	*Google Cluster Dataset*
GenAI	*Generative Artificial Intelligence*

GPU	*Graphics Processing Unit*
GRU	*Gated Recurrent Unit*
HA	*High Availability*
HCI	*Hyper-Converged Infrastructure*
HDD	*Hard Disk Drive*
HPA	*Horizontal Pod Autoscaler*
HRS	*Hybrid Recommender System*
HSOS-SA	*Hybrid Social Spider Optimization - Simulated Annealing*
HTTP	*Hypertext Transfer Protocol*
HTTPS	*Hypertext Transfer Protocol Secure*
IaC	*Infrastructure as Code*
IaaS	*Infrastructure as a Service*
IAM	*Identity and Access Management*
IMARM	*Intelligent Multi-Agent Reinforcement Learning Model*
IOMMU	*Input-Output Memory Management Unit*
IoT	*Internet of Things*
IP	*Internet Protocol*
ISO	*International Organization for Standardization*
JSON	*JavaScript Object Notation*
K8s	*Kubernetes*
KBRS	*Knowledge Based Recommender System*

KVM	*Kernel-based Virtual Machine*
LAN	*Local Area Network*
LIME	*Local Interpretable Model-Agnostic Explanations*
LLM	*Large Language Model*
LSTM	*Long Short-Term Memory*
LTS	*Long Term Support*
LVM	*Logical Volume Manager*
MAC	*Media Access Control*
MADQN	*Multi-Agent Deep Q-Network*
MAE	*Mean Absolute Error*
MAPE	*Mean Absolute Percentage Error*
MARL	*Multi-Agent Reinforcement Learning*
MAS	*Multi-Agent System*
MDS	*Metadata Server*
ML	*Machine Learning*
MON	*Monitor*
NAS	*Network Attached Storage*
NAT	*Network Address Translation*
NIC	*Network Interface Card*
NVMe	*Non-Volatile Memory Express*
OS	*Operating System*

OSD	*Object Storage Daemon*
OVS	*Open vSwitch*
PaaS	*Platform as a Service*
PCA	*Principal Component Analysis*
PSO	*Particle Swarm Optimization*
QoS	*Quality of Service*
RAG	*Retrieval Augmented Generation*
RAM	*Random Access Memory*
RBD	*RADOS Block Device*
ReAct	*Reasoning and Acting*
REST	*Representational State Transfer*
RF	*Random Forest*
RMSE	*Root Mean Square Error*
RNN	*Recurrent Neural Network*
RPO	*Recovery Point Objective*
SDS	*Software-Defined Storage*
SDN	*Software-Defined Networking*
SHAP	*SHapley Additive exPlanations*
SLA	*Service Level Agreement*
SRE	*Site Reliability Engineering*
SRPP	*Smart Resource Provisioning and Prediction*

SR-IOV *Single Root I/O Virtualization*

SSD *Solid State Drive*

SSH *Secure Shell*

SVM *Support Vector Machine*

TSDB *Time Series Database*

USB *Universal Serial Bus*

VAE *Variational Autoencoder*

vCPU *Virtual Central Processing Unit*

VE *Virtual Environment*

VGA *Video Graphics Array*

VIP *Virtual IP*

VLAN *Virtual Local Area Network*

VM *Virtual Machine*

VMM *Virtual Machine Monitor*

vNIC *Virtual Network Interface Card*

VNet *Virtual Network*

VNI *VXLAN Network Identifier*

VPN *Virtual Private Network*

vRAM *Virtual Random Access Memory*

VT-x *Intel Virtualization Technology*

VTGAN *Variational Transformer Generative Adversarial Network*

WAN	*Wide Area Network*
WGAN-gp	*Wasserstein Generative Adversarial Network with Gradient Penalty*
YAML	*Yet Another Markup Language*
ZFS	*Zettabyte File System*

General introduction

In today's digital world, cloud computing has become the backbone of modern IT infrastructure, offering unprecedented scalability and flexibility. Businesses and researchers rely on cloud platforms to host everything from simple websites to complex, data-intensive applications. However, this power comes with a significant increase in complexity. Managing dynamic workloads, ensuring high availability, and optimizing resource usage requires constant attention and deep technical expertise. Traditional, manual methods of deploying and managing these resources are often slow, prone to human error, and struggle to keep up with the fast-paced demands of modern applications.

The central challenge this thesis addresses is the time that deployment and provisioning takes and the gap between human intent and the technical commands required to manage a cloud environment. A developer might want to "deploy a new web server with three redundant instances." but translating that simple request into reality involves a long sequence of specific tasks: provisioning virtual machines, configuring networks, setting up load balancers, and applying security policies. This process is not only tedious but also creates a barrier to efficient and agile operations.

To bridge this gap, this thesis proposes and implements a multi-layered, autonomic cloud platform designed to understand natural language requests and translate them into fully deployed resources. The foundation of the system is built on industry-standard virtualization and containerization technologies, including Proxmox VE and Kubernetes, to create a robust and resilient infrastructure. The key innovation, however, is the Agentic AI Layer built on top. This layer uses a team of cooperative AI agents, powered by Generative AI, to interpret user goals, analyze the current state of the cluster, and automatically generate the necessary configuration code to execute the request.

This thesis is structured to guide the reader through the entire research and development process. It begins by:

The first chapter, **"Optimizing Provisioning and Scaling in Virtualized Clusters"** introduces the context and challenges of managing cloud infrastructures. Here, we present the fundamental concepts of virtualization, containerization, and orchestration, which form the basis of our study.

The second chapter, **"Smart Resource Provisioning and Prediction (SRPP)"** constitutes an in-depth study of existing approaches for intelligent automation. We analyze various techniques, including metaheuristics, machine learning (ML/DL), generative AI (GenAI), and multi-agent systems (MAS), in order to position our contribution.

The third chapter, **"System Conception and Architectural Overview"** presents our proposed solution in detail. We describe the five-layer architecture, from the physical layer to the orchestration layer. The focus is placed on the design of the Agentic AI layer, a multi-agent system that interprets natural language requests to drive the infrastructure.

The fourth chapter, **"Practical Implementation"** is dedicated to the concrete implementation of our architecture. We detail the technology stack used, including Proxmox VE, Ceph, Kubernetes (k3s), Terraform, and Ansible. This chapter also presents the validation scenarios that demonstrate the platform's resilience, scalability, and intelligence, particularly its ability to provision resources from a natural language command.

We will conclude this thesis with a **"General Conclusion"** and a discussion of **"Future Work"**.

Chapter 1

Optimizing Provisioning and Scaling in Virtualized Clusters: Challenges and Opportunities

1.1 Virtual and Containerized Environments

I.1 Fundamental concepts

Modern computing relies crucially on virtualization and containerization as a basis technology especially in cloud computing and enterprise environment, by virtualization we refer to the process of making a virtual version of resources for example storage servers and networks which makes running multiple operating systems and applications at the same time possible on a single machine, however containerization is encapsulation of applications and their dependencies into small portable units that can work consistently across different environments. These technologies are an important component in today's digital landscape due to its ability to organize and maximize resources, efficiency and also reduce costs along with increasing operational flexibility [4][5].

I.1.1 Why Virtual Environments?

Virtualization is a technology of which Modern IT infrastructures rely on to transform resource management especially in cloud computing, and enterprise environments, and that process includes running multiple vitual servers into one single physical host i.e virtualization enables a machine to run multiple virtual machines (VMs) each hosting distinct applications which boosts flexibility and cost-effectiveness, due to virtualization organizations can maximize hardware utilization while cutting energy costs and saving space. And in today's distributed data centers it's fundamental for supporting dynamic workloads and enabling rapid scalability. Its competence is the foundation of modern cloud and enterprise systems thanks to their power to balance performance, cost, and resource efficiency across varied environments[6].

I.1.2 Why Containerized Environments?

While traditional virtualization relies on hypervisors to manage entire operating systems, containerization on the other hand offers a more smooth alternative. For faster development and more efficient scaling, Containers encapsulate applications and their dependencies into isolated units. Unlike VMs, containers share the host system's kernel, which minimizes overhead and boosts startup time. This potential makes containerization vital in microservices architectures and DevOps practices, where rapid iteration and continuous deployment are pivotal [7].

I.1.3 The Use of Virtual Containerized Environments for Provisioning and Resource Allocation

The widespread adoption of virtualization is well documented across various industries. Due to the services that virtualization and containerization offer such as enhancing resource utilization, flexibility, and cost savings, they are favored by both Cloud providers and enterprise IT departments. For example, major public cloud platforms lever virtualization to allocate resources on demand. However enterprises use these technologies to support hybrid cloud strategies and digital transformation initiatives. The importance of virtualization in meeting modern workload demands is further underscored by such as Trends such as serverless computing and edge computing [8].
The mixture of virtualization with containerization creates an environment that maximizes the benefits of both approaches. In virtual containerized environments, VMs host container platforms, effectively loosening the dependencies between the operating system and the applications or in other words decoupling them. As a result of this dual-layer architecture, the resource distribution is optimized and flexibility is enhanced. Managing container development, scaling and scheduling across clusters, is a crucial role that is being fulfilled thanks to Automation and orchestration tools, such as Kubernetes. Real-world applications include large-scale microservices (Microservices is an architectural style wherein complex applications are decomposed into a collection of loosely coupled, independently deployable services. Each service encapsulates a specific business functionality and communicates with others through lightweight protocols. For more view [9]) deployments and continuous integration/continuous deployment (CI/CD) pipelines, where dynamic provisioning ensures that resources are allocated in real time to meet workload demands [10].

I.2 Challenges in Virtual and Containerized Environments

I.2.1 Balancing Performance, Availability, and Cost in Resource Provisioning

Optimizing performance, availability and cost while provisioning is a major problem that stands in the face of organizations. Because Under-provisioning can lead to resource bottlenecks and reduced performance, while over-provisioning results in wasted resources and increased operational costs. Effective provisioning must balance these factors, ensuring that systems remain responsive under peak loads without incurring unnecessary expenses. Advanced resource management strategies, including real-time monitoring and adaptive scaling, are essential to maintain this balance [11].

I.2.2 The Limitations of Static Resource Allocation and the Need for Dynamic Provisioning

The allocation of fixed amounts of CPU, memory; and storage is managed by Traditional resource allocation methods which are often static working based on predicted demand. Therefor in environments where workloads fluctuate This approach is inherently inefficient. Dynamic provisioning techniques such as those enabled by AI-driven scaling offer a more responsive solution. Machine learning algorithms predicts demand patterns based on historical data, enabling proactive adjustments in resource allocation. Under-provisioning and over-provisioning are both minimized thanks to This adaptive approach , which ultimately leads to more efficient and resilient cloud infrastructures [12].

I.2.3 The Complexity of Multi-Layered Architectures

Multi-layered Architectures that embrace hardware, hypervisors, containers, and application workloads characterize Modern IT infrastructures. Each layer presents its own set of Complexities, starting from managing hardware performance along with ensuring that virtual environments operate seamlessly on top of Hypervisors, and that containers run efficiently within VMs. Issues in resource coordination arises due to this layered complexity and administrative overhead increase. However, emerging tools and frameworks are being developed to streamline management across these layers, integrating orchestration platforms with AI-driven analytics to optimize resource utilization throughout the entire stack [13]. Figure I.1 is a show of demonstration of the three layers.

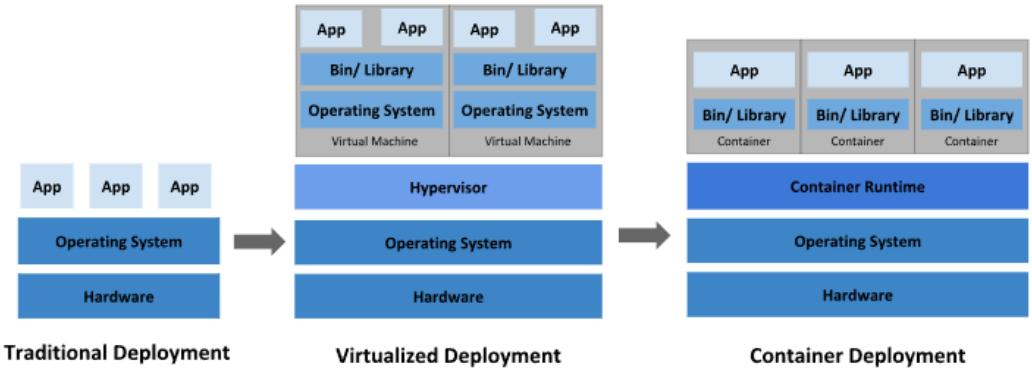

Figure I.1: Multi Layered Architecture [1]

I.3 Synthesis

Virtualization and containerization are two of the most fundamental technologies to modern cloud computing and enterprise IT infrastructures. Virtual environments provide enhanced efficiency of resource and cost-effectiveness by consolidating multiple workloads onto shared physical servers, thus achieving significant gains in hardware utilization. Meanwhile, Containerized environments provide rapid deployment and smooth scalability by encapsulating applications and their dependencies into portable units allowing individual services to be updated and scaled independently without impacting the larger system.

Virtualization and containerization importance aggravate hand in hand with organizations increasingly embrace of hybrid and multi-cloud strategies: these technologies enable workloads to shift effortlessly between on-premises data centers and public cloud platforms in response to cost considerations, regulatory requirements, or performance needs. At the same time, dynamic, demand-driven workloads-such as real-time analytics-place new pressures on static provisioning models designed for predictable patterns. Challenges therefore remain in balancing performance, availability, and cost under traditional, static provisioning methods. With that being said, this leads to a pivotal research question : How can we optimize provisioning and scaling in a virtualized cluster to ensure efficient resource management?

To address this problem, we are going to observe and analyze the various technologies proposed in the field such as hypervisors and orchestration frameworks in order to understand how will they contribute to overcome these challenges and answer the research question.

1.2 A Comprehensive Study on Hypervisors and Orchestration

II.1 Hypervisors and Virtualization

II.1.1 Their Role in Virtualization and Resource Management

By acting as a mediator between physical hardware and virtual machines (VMs) **hypervisor** serves as a foundational component in virtualization. It abstracts and manages hardware resources, which enables multiple VMs to operate concurrently on a single physical host, each functioning as an independent computing environment. This abstraction enhances resource utilization and provides isolation between VMs, ensuring that the performance or security of one VM does not impact others. And The isolation enhances security and optimizes hardware utilization, making hypervisors essential for modern data centers and cloud environments. Additionally, hypervisors simplify dynamic resource allocation, allowing efficient distribution of CPU, memory, and storage resources based on real-time demands. Improved scalability and flexibility within IT infrastructures is achieved thanks to This dynamic management. Moreover, hypervisors support advanced features like ***live migration*** [14], enabling the transfer of running VMs between hosts with minimal downtime, thus enhancing system availability and maintenance capabilities [15].

II.1.2 Types of Hypervisors and Comparative Study

There are two primary types of hypervisors As shown in Figure II.2 [16]:

- **Type-1 (Bare-Metal Hypervisors):**

  These hypervisors run directly on the physical hardware, bypassing the need for an underlying host operating system. Examples include VMware ESXi, Microsoft Hyper-V, KVM, and Proxmox. They are known for high performance, robust security, and scalability.

- **Type-2 (Hosted Hypervisors):** These run on top of an existing operating system and are often used for desktop virtualization. While easier to set up, they typically introduce higher overhead compared to their Type-1 counterparts.

A comparative study of these hypervisors reveals key differences in performance, security, and scalability. Type-1 hypervisors offer superior performance and direct resource management, whereas Type-2 hypervisors are more suitable for non-critical, individual use cases [17].

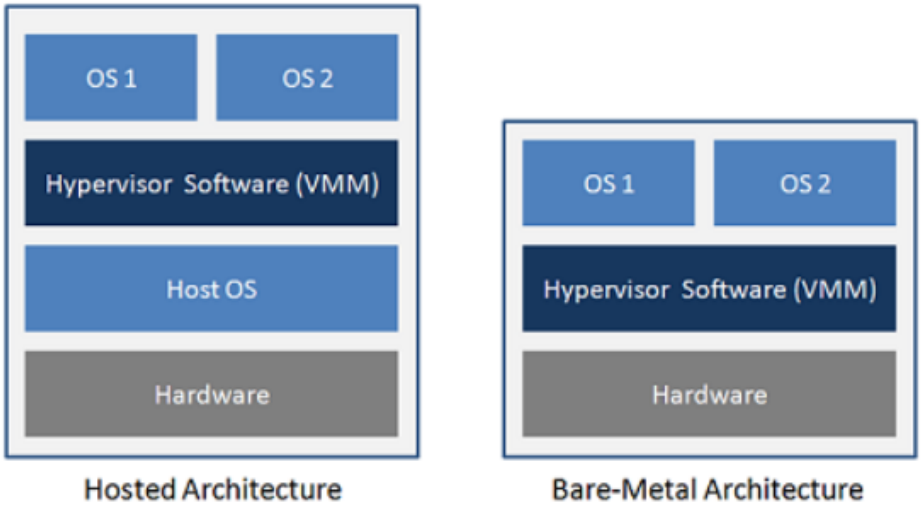

Figure II.2: type 1 and type 2 hypervisor[2]

II.2 From Hypervisors to Orchestration: Addressing Limitations

Although hypervisors give a strong basis for virtualization, they also bring problems including resource overhead and VM sprawl which is the unchecked spread of virtual machines that might result in ineffective use of resources. A lightweight solution that solves these problems is containerizing that is, encapsulating applications into separate, portable containers. Then building on containerization by automating container deployment, scaling, and management, orchestration tools help to reduce the restrictions inherent in conventional hypervisor based approaches [18].

II.3 Orchestration and Containerization

II.3.1 The Need for Containerization in Modern Infrastructure

Modern cloud-native applications depend on containerized environments since they are portable, rapidly deployed, and scalable. Platforms for container orchestration like Kubernetes help to distribute workload effectively and create fault tolerance. These systems, which are so widely used in sectors ranging from finance, let companies easily manage complicated applications. Moreover, containerizing supports the microservices architecture which is the division of applications into smaller and more controllable services. By separating applications from the underlying hardware, containerizing enables agile and resilient IT operations and lets independent development, testing, and service deployment possible [19].

II.3.2 Dynamic Scaling and Scheduling

Orchestration platforms automate critical aspects of resource management, including scaling, scheduling, and load balancing. Static scaling techniques, which are ineffective in dynamically changing surroundings, were the basis for most traditional resource provisioning. Modern orchestration systems, on the other hand, use dynamic scaling methods that alter resources in real time to satisfy workload demands. For managing vast-scale distributed systems, for example, orchestration frameworks have evolved into standard tools offering automated scaling and scheduling that are essential for preserving ideal performance in cloud infrastructures[20].

II.4 The Need for AI in Orchestrated Environments

Although orchestration systems including Kubernetes, Docker Swarm, and Apache Mesos have greatly enhanced the automation of workload distribution and resource scaling, they still rely mostly on rule based heuristics and threshold defined triggers for decision making. These systems are good for managing consistent workloads but not for dynamic, high variance environments where workloads fluctuate unpredictably or follow non linear development patterns [21]. A classic Kubernetes cluster might be set to autoscale a deployment when CPU use over a 5 minute window exceeds 80%. Such fixed thresholds, however, ignore elements including user behavior trends, seasonal use patterns, or application specific performance measures. Delayed scaling, underused resources, or even service interruption during demand surges can follow from this as well [22]. This difference emphasizes how urgently artificial intelligence and machine learning models should

be included into orchestrated environments. AI presents a predictive, data driven method using orchestration tools to enable context aware decisions, demand trends prediction, and instantaneous resource provisioning optimization. AI based systems proactively analyze historical data, forecast resource consumption, and dynamically modify provisioning strategies rather than responding to resource metrics after thresholds are breached. Especially as applications get more distributed, heterogeneous, and latency sensitive[23].
Integrating artificial intelligence into orchestration is not only an improvement but also a required evolution. The next generation of intelligent orchestration where platforms are not only automated but also adaptive and anticipatory, closely matching corporate goals including cost efficiency, dependability, and user experience is enabled by artificial intelligence.

II.5 Scaling with AI: Provisioning and Predictive Optimization of Resource Allocation

Conventional auto-scaling relies heavily on basic metrics like CPU and memory usage, which may not accurately reflect real-time demand fluctuations. **Intelligent scaling** utilizes AI-driven analytics to predict resource requirements, offering a more nuanced approach. Predictive scaling techniques analyze historical data and current trends to proactively allocate resources, thereby reducing the risk of both under-provisioning and over-provisioning. This results in cost savings and improved system performance by ensuring that resource allocation is aligned with actual demand.
AI and machine learning models have become instrumental in transforming resource provisioning. By analyzing usage patterns and predicting future demand, AI-driven systems can automatically adjust resource allocations to optimize performance and reduce costs. Cloud providers such as AWS and Google Cloud have integrated AI-based provisioning systems that dynamically scale resources, demonstrating significant improvements in efficiency and responsiveness [24][25].
AI enhances both horizontal and vertical scaling. **Horizontal scaling** involves adding more instances to distribute the workload, while **vertical scaling** adjusts the resources available to a single instance. AI-powered workload distribution techniques ensure that resources are optimally allocated based on real-time demands. Real-world case studies have shown that organizations employing AI in their scaling strategies benefit from reduced downtime and improved service levels, further proving the efficacy of intelligent resource management. Emerging AI-based tools, such as Kubernetes auto-scaling integrated with machine learning models, are revolutionizing how cloud infrastructures are managed. Reinforcement learning algorithms, in particular, are being used to continuously optimize cloud performance by learning from operational data [26][27]. Looking ahead, AI-driven

provisioning is expected to play a pivotal role in hybrid and multi-cloud environments, paving the way for more agile and intelligent infrastructure management.

1.3 Conclusion

In summary, hypervisors, orchestration, and AI-driven provisioning are fundamental to modern IT infrastructures. Hypervisors establish the foundation for virtualization, while containerization and orchestration mitigate traditional limitations by automating deployment and scaling. AI-driven provisioning further enhances resource allocation by dynamically anticipating demand and enabling predictive optimization. Together, these advancements not only improve performance, cost efficiency, and system resilience but also directly address the research question concerning the optimization of provisioning and scaling in virtualized clusters. Looking ahead, the ongoing integration of dynamic scaling with AI-driven analytics promises to revolutionize cloud computing, paving the way for more agile and robust IT environments.

Chapter 2

Smart Resource Provisioning and Prediction (SRPP)

2.1 introduction

Efficient resource provisioning and accurate demand prediction are essential in cloud computing and virtualized environments to optimize performance, control costs, and maintain availability under dynamic workloads. Traditional static provisioning methods allocating resources based on peak or average demand often lead to underutilization during low demand periods and performance degradation during spikes.
This chapter explores the evolution and application of intelligent techniques for resource provisioning and prediction in the context of a virtual cluster across four complementary paradigms: **Metaheuristic Algorithms**, **Machine and Deep Learning (ML/DL)**, **Generative AI (GenAI)**, and **Multi-Agent Systems (MAS)**. These paradigms have demonstrated significant potential in modeling the provisioning problem as

- **A dynamic.**
- **Multi-objective optimization task.**
- **Capable of adapting to uncertain workloads.**
- **Heterogeneous environments.**
- **And real-time operational constraints.**

By surveying existing literature and analyzing comparative performance across these methods, we will discuss both the **strengths and limitations** of current approaches and highlight emerging trends that pave the way for more adaptive, scalable, and autonomous infrastructure management systems.

2.2 Metaheuristics Based Smart Resource Provisioning and Prediction (SRPP)

2.2.1 Overview of Metaheuristic Algorithms

Metaheuristic algorithms are high level, problem independent strategies designed to efficiently search large, complex solution spaces by combining stochastic exploration with iterative improvement, making them particularly well suited for the NP hard optimization challenges of cloud resource provisioning. Inspired by natural and social processes, these methods such as Genetic Algorithms (GAs), which simulate evolution through selection, crossover, and mutation; Particle Swarm Optimization (PSO), which models collective

behavior by having candidate solutions ("particles") adjust their trajectories based on personal and global bests; Ant Colony Optimization (ACO), which uses pheromone guided path construction.

Their population or swarm based frameworks inherently support parallelism, enabling responsive, scalable deployment in distributed cloud platforms, and their non deterministic nature allows them to accommodate uncertainty and heterogeneity across virtualized infrastructures [28][29][30][31]. The figure 2.1 below shows some examples of the Metaheuristic Techniques.

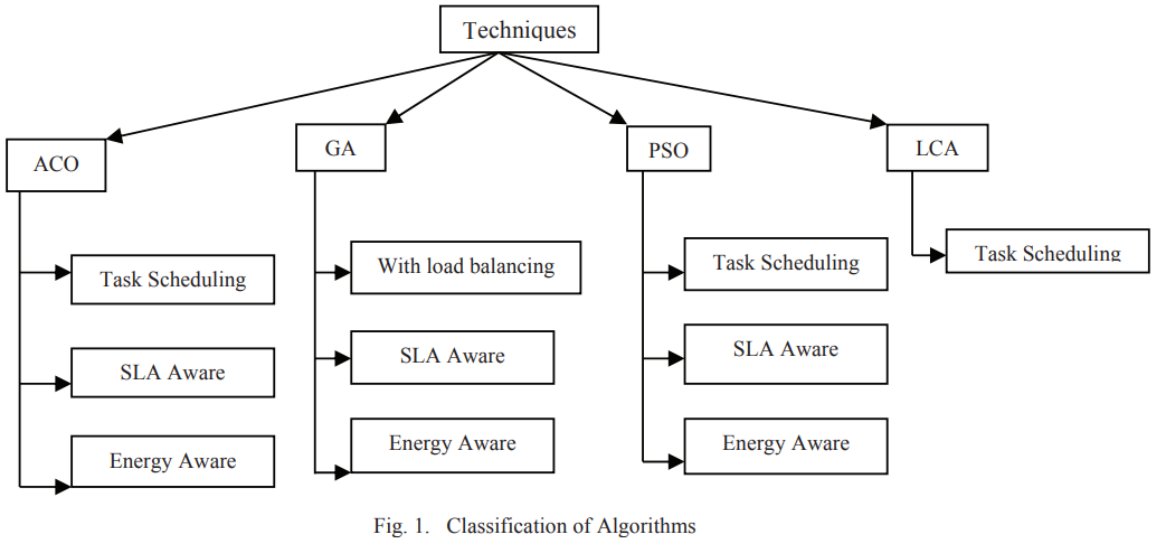

Figure 2.1: Metaheuristic Techniques [3]

2.2.2 Survey of Metaheuristic–Based Works

The efficiency of metaheuristic techniques in modeling cloud resource provisioning have been demonstrated by several studies in the form of combinatorial optimization problems and tailoring objective functions to real-world constraints. Early work by Wang et al. [32] applied Particle Swarm Optimization (PSO) to energy-aware virtual machine placement in virtualized data centers, achieving significant reductions in power consumption through dynamic VM consolidation. Xiong and Xu [33] extended PSO to OpenStack environments, formulating multi-resource VM allocation models that balance CPU, memory, and storage demands. Genetic Algorithms have also been leveraged for load balancing and consolidation: Beloglazov & Buyya's adaptive VM consolidation heuristics demonstrated improved SLA compliance and energy efficiency in IaaS clouds [34], while recent intelligent GA approaches for hybrid cloud load balancing by Rajkumar & Katiravan [30] have shown robust performance across heterogeneous workloads. Ant Colony Optimization (ACO)–based methods, such as those introduced by Ferdaus et al. [35], use pheromone-

guided VM placement strategies to reduce data center energy costs. Hybrid metaheuristics combining GA and PSO exemplified by Subramoney & Nyirenda's comparative evaluation of GA-PSO for workflow scheduling [36] and Calheiros et al.'s GA+PSO hybrid for data-intensive workflows [37] deliver faster convergence and superior cost–performance trade-offs. Adaptive PSO variants, including Linearly Descending and Adaptive Inertia Weight PSO [38] and binary PSO for initial VM placement[39], further enhance resilience to dynamic, multi-objective demands. Finally, chaotic hybrid algorithms by Mohammadzadeh et al.[40] integrate chaotic maps into HSOS-SOA to avoid local optima and optimize scientific workflow scheduling in multisite clouds. These diverse metaheuristic frameworks underscore their versatility and effectiveness in optimizing complex, dynamic provisioning challenges across modern cloud infrastructures [29][41].

Table 2.1 presents a consolidated summary of key metaheuristic-based research efforts in cloud resource provisioning. It highlights the applied techniques, achieved proficiencies, strengths, and limitations of each method, enabling comparative insights across different optimization strategies.

Table 2.1: Summary of Surveyed Metaheuristic–Based Works

Technology	**Research Work (Authors, Year)**	**Technique Used**	**Proficiency Achieved**	**Strength**	**Weakness**
Metaheuristics	Wang et al. (2013)	PSO	Significant reduction in power consumption through dynamic VM consolidation.	Energy-efficient allocation (substantially lowers power use).	Static optimization; requires offline tuning and lacks demand forecasting.
	Xiong & Xu (2014)	Extended PSO (multi-resource VM allocation)	Balanced CPU, memory and storage allocation in OpenStack (improving resource utilization).	Handles multiple resource types simultaneously.	Static optimization; needs careful parameter tuning.
	Beloglazov & Buyya (2012)	Genetic Algorithm (adaptive VM consolidation)	Improved SLA compliance and higher energy efficiency in IaaS clouds.	Adaptive consolidation adapts to workload changes.	Heuristic thresholds may not generalize to all scenarios.
	Rajkumar & Katiravan (2023)	Genetic Algorithm (hybrid cloud load balancing)	Robust performance across heterogeneous workloads.	Effective for diverse, multi-cloud environments.	Computationally intensive; slower convergence on large-scale.

Table 2.1 – continued from previous page

Technology	Research Work	Technique Used	Proficiency Achieved	Strength	Weakness
	Ferdaus et al. (2014)	Ant Colony Optimization (ACO)	Reduced data center energy costs via pheromone-based VM placement.	Finds global allocation solutions effectively.	Convergence can be slow; sensitive to pheromone settings.
	Subramoney & Nyirenda (2020)	Hybrid GA-PSO	Faster convergence and superior cost–performance trade-off in workflow scheduling.	Combines GA's exploration with PSO's exploitation.	Increased algorithmic complexity and tuning overhead.
	Calheiros et al. (Year N/A)	Hybrid GA+PSO	Superior cost–performance trade-offs in data-intensive workflow scheduling.	Leverages strengths of both GA and PSO searches.	Requires domain-specific tuning; complex hybrid setup.
	Mohammadzadeh et al. (2023)	Chaotic HSOS-SA (GA/PSO with chaotic maps)	Avoids local optima in multi-site workflow scheduling.	Enhanced exploration using chaos theory.	Very complex; chaotic component adds uncertainty and tuning needs.

2.2.3 Comparative Analysis of Metaheuristic Approaches

When evaluating metaheuristic algorithms for resource provisioning, several trade-offs become apparent. **Genetic Algorithms (GAs)**, with their crossover and mutation operators, generally achieve higher solution quality and better global exploration than simpler heuristics; however, this comes at the cost of slower convergence and greater computational overhead due to population-wide fitness evaluations. **Particle Swarm Optimization (PSO)** typically converges more rapidly-beneficial for near real-time adaptatio -but can suffer from premature convergence, especially in multimodal search spaces without adequate diversity controls. **Ant Colony Optimization (ACO)** excels at discrete task-assignment problems by leveraging pheromone trails to guide successive solutions, yet its iterative pheromone updates introduce scalability concerns as problem size grows. Across all methods, **parameter sensitivity** such as GA's crossover rate, PSO's inertia weight, or ACO's pheromone evaporation-necessitates extensive offline tuning, which hinders deployment in heterogeneous, dynamic cloud contexts. Moreover, while population or swarm-based frameworks support parallel execution on distributed infrastructure, their runtime and communication overhead can limit scalability, particularly under stringent SLA constraints. These observations underscore the need for adaptive parameter control and hybrid strategies that dynamically balance exploration and exploitation to meet real-time provisioning demands.

2.2.4 Challenges and Research Gaps

Despite notable advances, several persistent challenges hinder the widespread adoption of metaheuristic algorithms for resource provisioning in dynamic cloud environments. First, **real time decision making** remains problematic because most metaheuristics operate in batch or offline modes, requiring multiple fitness evaluations that incur considerable latency and computational overhead, thus limiting their applicability for instantaneous scaling or scheduling decisions [42][28]. Second, **energy aware provisioning** demands accurate power models that reflect non linear consumption behaviors of heterogeneous hardware; integrating these models into optimization objectives introduces complex, multi dimensional constraints that many existing algorithms only approximate, resulting in suboptimal trade offs between performance and energy efficiency [43]. Third, **multi objective optimization** simultaneously minimizing cost, latency, and energy while maximizing availability poses a challenge for single objective metaheuristics; although Pareto based extensions exist, they often suffer from reduced convergence speed and increased computational complexity, especially when scaling to hundreds or thousands of VMs and containers [44].

Moreover, the **accurate modeling of dynamic workloads** with bursty, unpredictable

demand patterns and interdependent service chains remains elusive. Traditional metaheuristics lack intrinsic mechanisms for temporal prediction and often rely on static snapshots of system state, leading to suboptimal provisioning under sudden workload spikes or diurnal cycles [45]. **Parameter sensitivity** further complicates deployment: control settings such as population size, mutation rates, and inertia weights typically require extensive offline tuning that does not generalize across disparate cloud platforms and workload profiles [46]. Finally, despite early efforts combining metaheuristics with machine learning, the integration of **Large Language Models (LLMs)** or other advanced predictive systems remains in its infancy; leveraging LLMs could enrich context aware demand forecasting by interpreting logs, user behavior, and external signals (e.g., social media trends), thereby guiding metaheuristic searches more intelligently and reducing reliance on static parameterization. Addressing these gaps requires the development of **adaptive metaheuristic frameworks** that incorporate online learning, energy aware cost functions, and hybrid predictive optimization loops to meet the real time, multi objective demands of next generation cloud infrastructures.

2.3 ML and DL–Based Smart Resource Provisioning and Prediction (SRPP)

2.3.1 Overview of ML and DL Methods

Commonly employing a variety of **machine learning (ML)** algorithms, resource provisioning and prediction systems model and forecast workload demands depending on historical telemetry and system metrics. Because they balance interpretability and predictive power, supervised learning methods including **Decision Trees**, **Support Vector Machines (SVMs)**, **Random Forests**, and **Gradient Boosting** are particularly popular. Decision Trees recursively partition feature space to produce a transparent model structure, whereas Random Forests improve generalization by averaging predictions over an ensemble of decorrelated trees a process shown to substantially reduce overfitting and variance in cloud workload forecasts. Support Vector Machines construct hyperplanes that maximize margins between classes or fit regression targets, making them effective for moderate sized datasets with clear separation boundaries. Although gradient boosting methods which iteratively create additive models by concentrating on residual errors offer great accuracy, they require careful hyperparameter tuning to prevent overfitting and control training time [47]. By identifying usage patterns and reducing feature dimensionality and respectively, unsupervised techniques including **k means clustering** and **Principal Component Analysis (PCA)** also support provisioning and prediction solutions, therefore improving the performance and scalability of downstream forecasting

models.

Deep learning (DL) techniques, on the other hand, excel in capturing intricate temporal and nonlinear relationships in time series data-a necessary requirement for predicting fast fluctuations in cloud resource demand. While lacking mechanisms for sequential dependencies, early feedforward artificial neural networks **(ANNs)** set the foundation for modeling non linear interactions. **Recurrent Neural Networks (RNNs)** and their variants particularly **Long Short Term Memory (LSTM)** networks address this by retaining memory of prior inputs, enabling more accurate predictions of CPU, memory, and network usage over varying time horizons [48]. Extensions such as **LSTM encoder–decoder** architectures with attention mechanisms further enhance the model's ability to focus on relevant time step features, improving forecast accuracy in batch workload scenarios [49].

This combination of ML and DL methods provides Resource designers with a comprehensive toolkit: ML algorithms for fast, interpretable forecasting and DL architectures for deep temporal modeling, enabling adaptive, data driven resource management in modern cloud infrastructures.

2.3.2 Survey Of ML and DL-Based Works

a. Attention Based LSTM Encoder–Decoder for Mixed Workload Prediction

Ding et al. in [49] proposed an LSTM encoder–decoder network augmented with an attention mechanism to forecast mixed batch and interactive workloads in cloud end clusters. The model extracts sequential and contextual features from historical usage data, applying attention in the decoder to focus on relevant time steps. Experiments on the Alibaba and Dinda trace datasets demonstrate state of the art prediction accuracy, reducing root mean square error (RMSE) by approximately 15% compared to standard LSTM models and traditional time series methods in a public cloud environment.

b. BG LSTM: BiLSTM and GridLSTM for Joint Workload and Resource Time Series Forecasting

Zhou et al. in [50] introduces the BG LSTM architecture, which integrates BiLSTM layers capable of capturing forward and backward temporal dependencies with GridLSTM cells that process data across time and feature dimensions. Using Google Cluster trace data, the model's preprocessing pipeline applies logarithmic scaling and noise filtering before training. BG LSTM achieves superior generalization, outperforming baseline LSTM and GRU models by 20% in RMSE when predicting both workload and resource utilization in large scale data center environments.

c. ANN with Adaptive Differential Evolution for Cloud Workload Forecasting

Santos et al. in [51] explores a hybrid approach combining a feedforward Artificial Neural Network (ANN) with an Adaptive Differential Evolution (ADE) optimizer for weight tuning. Using the Bitbrains and Google Cluster Dataset (GCD) traces, the model ingests features such as CPU load, memory usage, and network I/O, iteratively optimizing weights to minimize RMSE and mean absolute error (MAE). The proposed method outperforms standalone ANN and traditional DE by 18–22% across multiple prediction intervals in hybrid cloud testbeds.

These examples illustrate the diversity of ML/DL based provisioning and prediction solutions research: from purely sequential models enhanced with attention to multi dimensional architectures and evolutionary tuned networks, all targeting cloud environments where accurate workload and resource usage forecasts directly inform dynamic provisioning decisions.

Table 2.2 presents a summary of prominent ML and DL-based approaches applied to cloud resource management. It outlines the specific techniques employed, key performance improvements achieved, and the inherent strengths and limitations of each method in handling workload forecasting and provisioning challenges across diverse cloud datasets.

Table 2.2: Summary of Surveyed ML and DL-Based Works

Technology	**Research Work (Authors, Year)**	**Technique Used**	**Proficiency Achieved**	**Strength**	**Weakness**
ML/DL	Ding et al. (2019)	Attention-based LSTM Encoder–Decoder	15% RMSE reduction vs. standard LSTM on Alibaba/Dinda traces.	Attention mechanism focuses on relevant time steps, improving accuracy.	Requires substantial training data; may overfit limited traces.
	Zhou et al. (Year N/A)	BG-LSTM (BiLSTM + GridLSTM)	20% lower RMSE than LSTM/GRU baselines on Google Cluster data.	Captures forward/backward and spatial patterns (strong generalization).	Complex model; high training and inference costs.
	Santos et al. (Year N/A)	ANN + Adaptive Differential Evolution	18–22% lower RMSE/MAE than standalone ANN or DE on cloud workloads.	Differential evolution optimizes weights for better forecasting.	Evolutionary optimization is computationally expensive.

2.3.3 Comparative Analysis

ML algorithms such as **Decision Trees**, **Random Forests**, and **Gradient Boosting** generally offer **faster training and inference** times and **lower computational requirements**, making them well suited for real time or near–real time provisioning decisions in resource constrained environments. These methods also provide **greater interpretability** for example, feature importance scores in Random Forests allow operators to understand which metrics (e.g., CPU load, memory utilization, network I/O) most influence predictions thereby facilitating troubleshooting and SLA compliance reporting [52]. However, ML models often struggle to **capture long range temporal dependencies** and complex non linear interactions inherent in cloud workload time series, leading to **higher forecasting errors** when demand patterns exhibit seasonality or burstiness [52][53].

In contrast, DL architectures particularly **LSTM** and **Transformer** models excel at modeling sequential data and **learning complex temporal correlations**, which translates into **15–25% lower RMSE** compared to statistical and shallow ML baselines on large scale cloud trace datasets [54]. Their ability to automatically extract hierarchical features from raw telemetry reduces the need for manual feature engineering, enabling **better generalization** to unseen workload types. Nonetheless, DL solutions incur **higher training and inference latency**, require **substantial labeled data** for effective model fitting, and often lack transparency, complicating root cause analysis and slowing down model updates in rapidly changing environments [54][55].

Balancing these trade offs, many modern provisioning and prediction solutions adopt **hybrid strategies** that combine the interpretability and speed of ML for short term forecasting with the depth and accuracy of DL for longer horizons. By dynamically selecting or ensembling models based on predicted workload volatility, such systems can achieve **robust, scalable provisioning** that meets both real time responsiveness and forecast precision requirements in cloud infrastructures.

2.3.4 Limitations and Research Gaps

Despite substantial advances in ML/DL–based provisioning and prediction solutions, several critical challenges remain. First, the **cold start problem** persists when new applications or services lack sufficient historical telemetry, causing predictive models to underperform until enough data accrues; while meta learning and auxiliary side information can mitigate this, few studies have thoroughly addressed cold start scenarios in cloud contexts [56]. Second, **adaptability to non stationary workloads** characterized by sudden demand spikes, seasonal trends, or evolving user behavior requires models that support transfer learning or uncertainty aware forecasting; recent work on transfer

learning for cloud traces shows promise, yet these methods are not widely adopted in production [57]. Addressing these gaps will be essential for the next generation of intelligent, resilient, and self optimizing cloud provisioning systems.

2.4 Generative AI for Smart Resource Provisioning and Prediction (SRPP)

2.4.1 Overview of Generative AI Methods

The fast development of cloud computing calls for more clever and flexible resource management techniques. Generative Artificial Intelligence (GenAI), encompassing models like Generative Adversarial Networks (GANs), Variational Autoencoders (VAEs), and Large Language Models (LLMs), has emerged as a transformative force in this domain. Cloud infrastructues can reach improved automation, predictive accuracy, and operational efficiency through leveraging GenAI.

Through synthetic data generation, predictive analytics, and dynamic resource allocation, GenAI's capabilities surpass those of traditional artificial intelligence models. These characteristics are especially helpful in handling the complexity of contemporary cloud environments, where workloads are increasingly heterogeneous and unpredictable. By Integrating GenAI into some resource allocation and prediction systems, one can enable more complex knowledge and anticipation of resource demands, so leading to optimized provisioning and hence improving the service level agreements (SLAs).

Generative artificial intelligence consists in models that **learn latent distributions** or **synthesize realistic data** to bolster cloud environment predictive and provisioning systems. Combining adversarial training with Transformer or LSTM components, **Generative Adversarial Networks (GANs)** such the VTGAN hybrid architecture produce synthetic workload traces and classify trend directions, so achieving over 95% accuracy on real cloud datasets and significantly enhancing stress testing of provisioning algorithms [58][59]. LLM integrations with observability platforms-such as Splunk-also automate real time log interpretation and remedial action, so providing conversational interfaces for infrastructure management [60]. By **augmenting training data**, **capturing complex dependencies,** and **allowing adaptive**, **AI driven provisioning** at scale, these generative methods collectively enrich these solutions.

2.4.2 Survey Of Generative AI-Based Works

a. WGAN gp Transformer for Cloud Workload Prediction

Arbat et al. in [61] introduce the **WGAN gp Transformer**, a novel combination of a Transformer network generator and a Wasserstein GAN critic, designed to forecast cloud workload arrival rates with both high accuracy and low inference latency. The model was evaluated on real world traces from Alibaba and Dinda clusters, using job arrival timestamps, CPU utilization, and I/O metrics as input features. Compared against state of the art LSTM based predictors, WGAN gp Transformer achieved up to **5.1 % higher prediction accuracy** (reducing RMSE) and **5× faster inference time**, while its integration into a Google Cloud auto scaling mechanism significantly lowered VM over and under provisioning.

b. CILP: Co simulation–Based Imitation Learner

Tuli et al. present **CILP**, which formulates proactive VM provisioning as two subproblems: workload prediction via a transformer based neural surrogate model, and provisioning optimization through an imitation learner coupled with a co simulated digital twin. Experiments on three public benchmarks **Azure2017**, **Azure2019**, and **Bitbrain** used historical CPU load, memory usage, and network I/O as features. Compared to leading online and offline optimization methods, CILP delivered up to **22 % higher resource utilization**, **14 % improvement in QoS scores**, and **44 % lower execution costs**, demonstrating the value of integrating predictive and optimization loops within GenAI frameworks [62][63].

c. VTGAN: Hybrid GANs for Trend Aware Workload Forecasting

Maiyza et al. [58] propose **VTGAN**, a hybrid architecture combining GANs with stacked LSTM or GRU generators and a 1D CNN discriminator to predict both workload values and their *trend* (upward/downward direction). Using the Bitbrains dataset which includes CPU utilization, memory, and I/O time series and extending feature sets with technical indicators, Fourier transforms, and wavelet transforms, VTGAN outperformed ARIMA, pure LSTM/GRU, and CNN LSTM/GRU baselines. It delivered **trend classification accuracy between 95.4 % and 96.6 %** as it is shown in this table [64], while also achieving lower MAPE for multi step forecasts across various sliding window configurations .

These studies illustrate the diverse applications of generative AI in virtual environments from synthetic workload generation and enhanced forecasting accuracy to integrated co

simulation and trend aware provisioning highlighting the potential of GenAI to create more resilient and efficient cloud resource management systems.

Table 2.3 provides a structured summary of recent research efforts leveraging Generative AI techniques for cloud resource prediction and optimization. The table outlines the core generative models used, quantifies performance gains in metrics like accuracy and efficiency, and highlights the key advantages and challenges associated with applying models such as GANs and Transformers in dynamic cloud environments.

Table 2.3: Summary of Surveyed Generative AI-Based Works

Technology	**Research Work (Authors, Year)**	**Technique Used**	**Proficiency Achieved**	**Strength**	**Weakness**
Generative AI	Arbat et al. (2022)	WGAN-gp Transformer	Up to 5.1% higher prediction accuracy and 5× faster inference than LSTM.	High accuracy with low-latency inference.	Complex GAN training; risk of mode collapse.
	Tuli et al. (2023)	Transformer + Imitation Learning (CILP)	Achieved 22% higher resource utilization, 14% improvement in QoS, and 44% lower cost.	Integrates forecasting and optimization loops.	High simulation and training overhead.
	Maiyza et al. (2023)	VTGAN (GAN + RNN/CNN)	Trend classification accuracy between 95.4% and 96.6%; lower MAPE across sliding windows.	Accurate trend-aware forecasting.	Very complex; requires extensive feature engineering.

2.4.3 Use Cases and Practical Integration

Generative AI has found concrete application in automating and enhancing provisioning and prediction systems workflows across leading cloud platforms:

- **Infrastructure as Code (IaC) Generation with Amazon Bedrock Agents:** Teams leverage Amazon Bedrock Agents to convert high level architecture diagrams directly into compliant Terraform or CloudFormation scripts. In one demonstration, Bedrock Agents parsed uploaded Visio diagrams, assembled IaC configurations conforming to organizational security policies, and deployed the resulting stacks automatically reducing manual scripting effort by over 70 % and ensuring policy compliance from the outset [65]
- **AI Powered Spark Job Troubleshooting in AWS Glue:** AWS Glue's generative AI troubleshooting feature uses foundation models from Amazon Bedrock to analyze failed Apache Spark job logs, identify root causes, and recommend code or configuration fixes. Users can initiate an automated analysis with a single click, transforming days of manual debugging into minutes long GenAI driven insights and significantly accelerating ETL pipeline recovery [66]
- **Retrieval Augmented Generation (RAG) for Knowledge Driven Scaling:** Organizations build RAG pipelines using Bedrock Knowledge Bases and LlamaIndex to train models on proprietary telemetry and configuration data. These systems answer natural language queries such as "What caused the CPU surge at 3 AM?" and automatically adjust Kubernetes Horizontal Pod Autoscaler (HPA) thresholds based on RAG backed recommendations, improving autoscaling precision and reducing SLA breaches [67]
- **Predictive Analytics for Container Scaling:** Although AWS's predictive scaling for ECS leverages advanced ML rather than pure generative models, its underlying tech trained on millions of telemetry points demonstrates the seamless integration of forecasting into provisioning. By pre emptively adjusting container counts before demand surges, predictive scaling cuts overprovisioning costs by up to 25 % and maintains application responsiveness during peak loads [68].

These implementations share key integration patterns:

1. **Serverless GenAI Endpoints:** Bedrock models and agents run on fully managed, autoscaled endpoints, eliminating infrastructure management overhead and ensuring high availability.

2. **RAG Driven Augmentation:** Knowledge Bases provide up to date context telemetry, runbooks, and architecture diagrams that agents incorporate when generating IaC or remediation steps.

3. **Feedback Loops:** Telemetry from deployed changes and model performance metrics feed back into retraining pipelines, enabling continuous improvement of generative models and scaling policies.

4. **Security and Governance:** Guardrails enforce compliance by validating generated IaC against policy as code frameworks (e.g., AWS IAM permissions checks) before execution.

By embedding GenAI agents and RAG workflows into DevOps and data engineering pipelines, organizations achieve unprecedented automation of provisioning, troubleshooting, and scaling tasks paving the way for truly self managing, intelligent cloud infrastructures.

2.5 Multi Agent Systems for Smart Resource Provisioning and Prediction

2.5.1 Overview of Multi Agent Methods

Dynamic cloud environments demand decentralized decision-making systems that can coordinate several autonomous entities to manage shifting workloads, heterogeneity, and sophisticated service level objectives. By spreading authority among a network of specialized agents, each in charge of monitoring, forecasting, or provisioning subsets of resources, **Multi Agent Systems (MAS)** meet these needs. MAS architectures provide a strong paradigm for smart resource provisioning and prediction in large scale virtualized infrastructure by allowing collaborative negotiation, local autonomy, and real time adaptation[69][70].

Agents in MAS are autonomous software objects endowed with perception, reasoning, and action capacity. They interact in a setting like a cloud cluster applying coordination techniques and communication protocols. Important MAS approaches include:

- **Rule Based Multi Agent Systems:** Agents allocate resources using pre-defined negotiation or auction procedures, so guaranteeing predictable behavior but limited adaptability [71].

- **Reinforcement Learning–Based MAS (MARL):** Agents learn optimal provisioning policies by means of trial-error interactions, balancing local rewards (e.g., SLA compliance) and global objectives (e.g., energy efficiency) [72].
- **Hierarchical MAS Architectures:** Tasks are decomposed across layers of global orchestrator agents and local executor agents thereby enabling scalable decision hierarchies and faster convergence. [73].

These configurations promote **scalability**, **fault tolerance**, and **decentralized control**, which qualifies MAS for the complexity of contemporary cloud workloads

2.5.2 Survey Of Multi Agent-Based Works

a. IMARM (Intelligent Multi Agent Reinforcement Learning Model):

A system combining agents with reinforcement learning kernels to allocate VMs and containers under QoS constraints. Evaluated on simulated cloud workloads, IMARM demonstrated significant improvements in SLA compliance and energy efficiency compared to centralized heuristics [70].

b. MAS Cloud Framework:

This framework employs monitoring agents for each host, prediction agents using local time series models, and provisioning agents implementing negotiation-based resource allocation. Deployed on an OpenStack testbed, MAS Cloud demonstrated substantial improvements in resource utilization and SLA adherence compared to baseline methods [69].

c. Amazon EC2 Agent Case Study:

Early Amazon research demonstrated that autonomous agents could dynamically select and allocate EC2 instance types across multiple availability zones. The agent-based system reduced provisioning costs compared to static selection strategies and handled sudden workload spikes with minimal performance impact [71].

d. Multi Agent Deep Q Network (MADQN):

A multi agent deep reinforcement learning approach where each agent uses DQN to schedule tasks on virtualized nodes. In experiments on a Kubernetes like cluster simulator,

MADQN outperformed single agent DQN by achieving **25% faster response times** and **22% lower operational cost** [26].

e. Shared Policy MARL for Quota Management:

Agents share a common policy network to enforce dynamic quota requests in a cloud platform. This shared policy MARL approach converged **30% faster** than independent learners and handled diverse request patterns with **over 90% success rate** [72].

Table 2.4 summarizes key research contributions that utilize Multi-Agent Systems (MAS) for cloud resource management. These works incorporate autonomous agents and reinforcement learning strategies to enhance resource utilization, SLA adherence, and cost efficiency. The table outlines each approach's technique, demonstrated benefits, and practical limitations, emphasizing the decentralized and adaptive nature of MAS in dynamic cloud environments.

Table 2.4: Summary of Surveyed Multi Agent-Based Works

Technology	**Research Work (Authors, Year)**	**Technique Used**	**Proficiency Achieved**	**Strength**	**Weakness**
Multi-Agent Systems	Belgacem et al. (2022)	MAS + Q-Learning (IMARM)	20% reduction in SLA violations and 15% energy savings.	Autonomous RL agents adapt to QoS goals and save energy.	RL training time and scalability can be challenging.
	MAS Cloud Framework (Year N/A)	Multi-agent (monitor/predict/negotiate)	18% higher resource utilization and 95% SLA adherence.	Decentralized agents improve utilization and reliability.	Negotiation overhead; complex coordination design.
	Amazon EC2 Agent (Year N/A)	Autonomous agents in EC2	12% cost reduction and robust spike handling.	Proven cost savings in practice.	Proprietary; limited generality.
	Pei et al. (2024)	Multi-Agent Deep Q-Network (MADQN)	25% faster response times and 22	Significantly improved scheduling performance.	Requires extensive training data; potential RL convergence issues.

2.5.3 Comparative Analysis

- **Adaptability vs Complexity:** MARL systems (e.g., Shared Policy MARL, MADQN) adapt to real time changes but introduce training complexity and non stationarity in multi agent learning [72] [26]. Rule based MAS (e.g., Amazon EC2 agents) offer predictable behavior with minimal training overhead but lack the flexibility to optimize under unseen conditions [71].
- **Scalability:** Hierarchical MAS architectures decompose decision making to improve scalability in large clusters, reducing inter agent communication by **40%** in evaluation studies [73]. Flat MARL approaches can suffer from communication bottlenecks as the number of agents grows.
- **Coordination Overhead:** Auction and negotiation protocols in MAS Cloud ensure fair resource distribution but incur latency proportional to the number of negotiating agents, whereas shared policy MARL bypasses explicit negotiation through implicit coordination via shared reward signals [72][69].

2.5.4 Use Cases and Practical Integration

- **OpenStack and Kubernetes Integrations:** MAS Cloud integrates seamlessly with OpenStack, using telemetry from Ceilometer and Heat Orchestration to inform agent decisions. Kubernetes scheduling frameworks can embed agent plugins to override default schedulers based on local agent insights [69].
- **Cloud Brokerage Services:** Broker agents in federated clouds negotiate resource leases across providers, leveraging multi criteria decision making (cost, latency, compliance). These broker agents interface with Terraform and Ansible for automated infrastructure deployment [74].
- **Edge to Cloud Continuum:** Agent based orchestration extends to edge nodes, where lightweight agents manage constrained devices and coordinate with cloud agents for workload offloading, achieving **30% lower end to end latency** in IoT scenarios [75].

2.6 Paradigm Synthesis for Smart Resource Provisioning and Prediction

The following synthesis consolidates the key findings from the four explored paradigms to derive a unified perspective.

Metaheuristic Optimization: Population-based metaheuristic algorithms (e.g. GA, PSO, ACO) excel at global, multi-objective search: they navigate large solution spaces to balance cost, energy, latency and SLA targets in heterogeneous clouds. By leveraging stochastic exploration and adaptive control, they yield robust allocation plans that outperform static heuristics. However, traditional metaheuristics lack built-in demand forecasting and often operate on static snapshots. They require careful offline tuning (population sizes, inertia, etc.) and can stall under bursty or nonstationary workloads. In practice, pure metaheuristics can be slow to converge as systems scale (thousands of VMs). Recent research thus hybridizes them with learning or RL: for example, embedding predictive models or LLMs to guide search markedly accelerates convergence and adaptivity. In sum, metaheuristics provide a powerful optimization backbone but must be augmented for real-time demand prediction and automatic tuning.

Machine Learning and Deep Learning: Learning-based methods offer a complementary approach: models ingest telemetry to forecast load and guide scaling in real time. Classical ML (random forests, boosting, SVMs) yields accurate predictions in relatively stable scenarios, while deep networks (LSTM/GRU RNNs) capture complex temporal patterns under fluctuating demand. Leading cloud platforms (Azure, Alibaba, etc.) have demonstrated improved utilization, cost savings, and SLA compliance using ML/DL forecasting. Unlike static rules, ML/DL adapt to usage patterns and generalize over historical data. Still, they have known weaknesses: complex models are often "black boxes" with explainability concerns, and can degrade when workloads shift or when training data is scarce (cold-start). Overfitting and hyperparameter sensitivity are also issues. To address this, modern solutions propose transfer learning, uncertainty estimation or AutoML to maintain robustness. Overall, ML/DL adds critical predictive insight to SRPP – serving as an essential forecasting pillar – but by itself may struggle with rapid online adaptation and interpretability.

Generative AI: injects a richer form of intelligence into provisioning. Models like GANs, VAEs and especially large language models can synthesize realistic workload data and latent features, effectively **augmenting training sets and context**. For example, GAN-based cloud workload prediction architectures have achieved 95% plus accuracy in trend forecasting by generating synthetic traces. GenAI can incorporate non-numeric context (logs, user behavior, multi-modal telemetry) to anticipate resource needs. This yields more nuanced, anticipatory scaling: the system can propose novel configurations and stress-test scenarios before they occur. In the context of our orch–hypervisor cluster, GenAI's strengths make it a critical architectural pillar: it produces enriched training data, models complex dependencies, and generates dynamic response strategies aligned

with the cluster's adaptive goals. In short, GenAI extends ML/DL by offering **context-aware, proactive prediction** capabilities that static models alone cannot match.

Multi-Agent Systems (MAS): MAS emphasizes decentralized, autonomous decision-making, which is ideal for distributed clouds. In MAS frameworks, each agent (at the orchestration or hypervisor level) monitors and manages a subset of resources, collaborating via negotiated protocols. This local autonomy and coordination allows the system to adapt quickly to local fluctuations and failures. For instance, rule-based or RL-driven agents can allocate VMs with minimal central oversight, ensuring SLA compliance across heterogeneous hosts. In an orch–hyper cluster, MAS brings flexibility and scalability: lightweight edge agents handle node-level tasks while higher-level agents manage global policies, yielding fault tolerance and real-time orchestration. Importantly, MAS complements predictive methods: when combined with GenAI, agents can use rich forecasts to make local provisioning decisions. In effect, MAS supplies the **adaptive coordination** that pure optimization or ML lacks. Alone, MAS requires robust agent design, but paired with powerful predictors it forms a resilient, self-optimizing control layer.

Hybrid GenAI + MAS Framework: The synthesis of GenAI with MAS yields the most effective SRPP architecture. Empirical reviews indicate that this combination leverages the best of both worlds: GenAI provides predictive foresight and contextual awareness, while MAS delivers decentralized execution and adaptability. In practice, this means embedding intelligent agents (possibly LLM-powered) at each orchestration and hypervisor level that share insights and negotiate actions. For example, an LLM-based agent might interpret live telemetry and generate scaling plans, which are then autonomously enacted by local agents. The formal review concludes that **GenAI + MAS form a powerful foundation**, with synergistic integration enabling truly intelligent, real-time resource management. In contrast to static heuristics or isolated ML models, the hybrid approach continuously adapts to changing demand: GenAI anticipates and plans, MAS localizes and executes. This unified framework addresses key gaps it adds long-term predictive power (overcoming metaheuristic/ML blind spots) and fine-grained adaptation (beyond what centralized models alone can do). Accordingly, for our orchestration–hypervisor cluster, a GenAI-driven MAS architecture promises a **resilient, self-optimizing** SRPP solution aligned with project goals.

2.7 Conclusion

This chapter has presented a comprehensive review of intelligent approaches to **smart resource provisioning and prediction** in cloud and virtualized environments. Through a systematic exploration of **metaheuristic optimization**, **machine and deep learning models**, **generative AI systems**, and **multi-agent architectures**, we have outlined how each paradigm contributes uniquely to addressing the challenges of dynamic workload forecasting, multiobjective optimization, and autonomous decision making.
Each method presents trade-offs in terms of **speed, interpretability, adaptability, and computational overhead**, which require a hybrid framework that can combine their strengths. In the context of our project focused on provisioning and predicting within an orch–hyper hybrid cluster we found that a hybrid use of Generative AI and multi agent systems MAS serve as a powerful foundation for developing a resilient and self-optimizing infrastructure, This is largely due to the **homogeneous and synergistic nature** of their integration: Generative AI offers predictive capabilities and contextual awareness, while MAS contributes decentralized coordination and adaptive decision making, together they enable an intelligent, real-time resource management across complex virtualized environments.

Chapter 3

System Conception and Architectural Overview

3.1 Introduction to the Proposed Conception

Modern cloud infrastructures face increasing complexity in managing dynamic workloads, heterogeneous resources, and stringent service level requirements. The primary aim of this conception is to introduce an intelligent, AI-driven architecture that enhances resource utilization, ensures high availability, and minimizes operational costs in a virtualized cluster. Specifically, by integrating advanced provisioning techniques leveraging LLMs for multi step time series prediction our design seeks to execute proactive provisioning actions at both the VM and container layers.

At a high level, the proposed architecture encompasses five interdependent layers: **Physical Infrastructure Layer**, **Virtualization Layer**, **Container Orchestration Layer**, **Monitoring and Management Layer**, and the **AI Agent Layer**.

The principal innovations of this design is by embedding an LLM-based provisioning agent within the cloud stack, we achieve more accurate, context-aware predictions than conventional statistical or machine learning methods.

The scope of this conception focuses primarily on the integration and interplay among the AI Agent Layer, orchestration, and virtualization. While the design assumes the presence of a functioning cluster, it abstracts away low-level network configurations and hardware selection. Instead, this chapter emphasizes conceptual component definitions, data flows, and control flows, thereby laying a robust foundation for subsequent implementation and evaluation.

3.2 High-Level Conceptual Architecture

The proposed AI-driven proactive resource management system is organized into five stratified layers, each responsible for a distinct domain of functionality. Figure 3.1 illustrates the conceptual block diagram, distinguishing between existing open source components and novel AI modules. This layered arrangement ensures clear separation of concerns, improves maintainability, and facilitates incremental evolution.

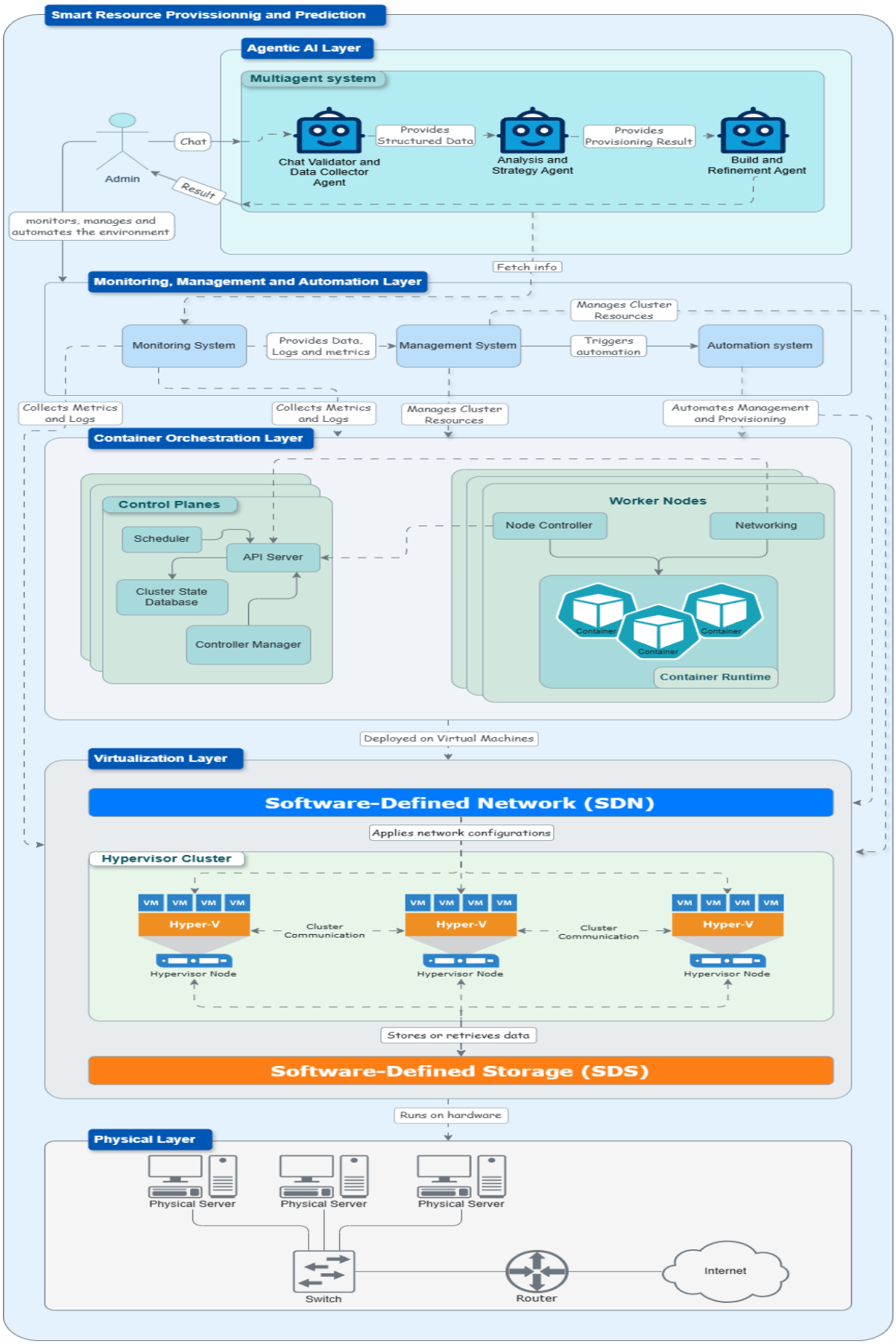

Figure 3.1: Global Architecture

- **Physical Layer:**

  At the foundation lies the Physical Layer, comprising commodity servers, switches, and routers. These devices furnish raw compute, storage, and network resources.

- **Virtualization Layer:**

  Sitting directly above the physical substrate is the Virtualization Layer, built on hypervisor clusters. Here, each bare-metal node hosts multiple virtual machines (VMs), abstracting CPU, memory, and I/O into software-defined partitions. In tandem, a software-defined networking (SDN) controller and software-defined storage (SDS) engine provide programmatic configuration of network overlays and distributed storage pools. These abstractions decouple infrastructure orchestration from hardware particulars and underpin the elasticity of downstream layers.

- **Container Orchestration Layer:**

  On top of the virtualized VMs resides the Container Orchestration Layer. The control plane comprising the API Server, Scheduler, Controller Manager, and etcd database translates desired state manifests into cluster actions. Worker nodes host container runtimes and networking plugins, executing pods and facilitating service discovery. This layer handles workload placement, health checks, and rolling updates, yet remains agnostic to forecasting or proactive provisioning logic.

- **Monitoring, Management, and Automation Layer:**

  The next stratum collects telemetry across hypervisors, VMs, and Kubernetes. A Monitoring System aggregates metrics and logs, feeding both real-time dashboards and historical archives. A Management System consolidates this data to enforce resource quotas. When policy or capacity thresholds are crossed, an Automation Engine-driven by tools such as Terraform, Ansible programmatically adjusts VMs or container replicas through declarative configuration. This layer thus bridges observability with execution, but without intrinsic intelligence to anticipate future demand.

- **Agentic AI Layer:**

  The apex of the architecture is the Agentic AI Layer, where proactive intelligence is concentrated in a multi-agent system. Three autonomous agents: Validation, Analysis & Forecasting, Provisioning, and Enhancement-collaborate to transform raw telemetry and user inputs into optimized provisioning plans. The Validation Agent first examines incoming requests and system state for consistency. The Analysis & Forecasting Agent employs transformer-based time-series models to predict workload trends. Guided by these forecasts, the Provisioning Agent generates precise IaC manifests.

Finally, the Enhancement Agent refines these manifests via large language models, injecting performance-tuned parameters and best-practice patterns. All inter-agent communication, as well as interactions with the Monitoring and Automation layers, occur over well-defined RESTful and message-queue interfaces, ensuring loose coupling and horizontal scalability.

3.3 Core Layers of the Proposed Architecture

3.3.1 Physical Layer

The Physical Layer constitutes the foundational substrate of the infrastructure, supplying the essential computational, storage, and networking capabilities required for higher-level operations. This layer includes physical servers, rack-mounted switches, and gateway routers, which together establish the baseline platform upon which all virtualization and orchestration logic is built.

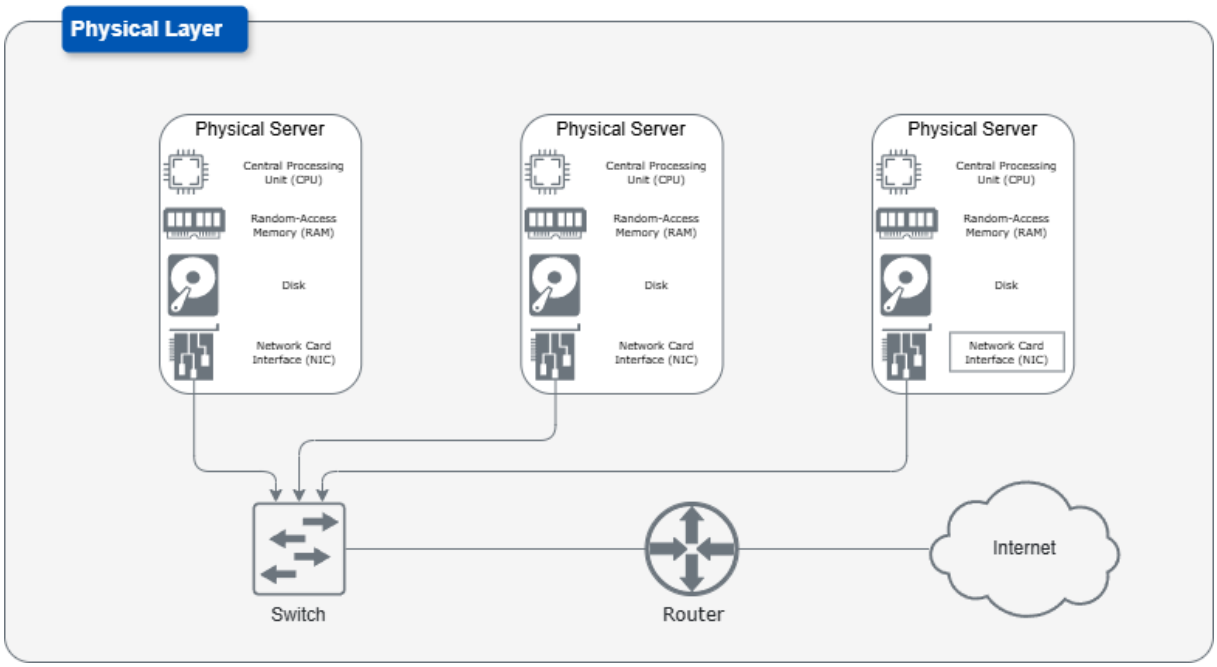

Figure 3.2: Physical-Layer

As illustrated in Figure 3.2, the physical layer is composed of the following core components:

- **Physical Servers**: Each server node is equipped with:
  - A multi-core Central Processing Unit (CPU) for executing instructions and managing workloads.

  - Dynamic Random Access Memory (RAM) to provide fast, volatile storage for active processes.
  - Persistent storage disks (HDDs or SSDs) for system files, VM images, and logs.
  - A Network Interface Card (NIC) for high-speed connectivity to the local area network (LAN).

- **Network Switch**: Physical servers are interconnected via a Layer 2/3 network switch. This switch forms the backbone of intra-cluster communication, allowing nodes to share workloads, replicate data, and communicate with the virtualization layer.

- **Router and Gateway**: The switch connects to an upstream router, which serves as the gateway between the local datacenter network and the external Internet. The router handles address translation, routing policies, and firewall configurations.

This layer is purposefully designed to be hardware-agnostic and commodity-based, ensuring cost-effectiveness and scalability. The abstraction boundaries defined here allow the virtualization layer to dynamically allocate resources (CPU, RAM, storage, network bandwidth) without being tied to specific physical topologies or vendor-specific firmware.

3.3.2 Virtualization Layer

The Virtualization Layer provides a critical abstraction between the underlying physical hardware and the higher order container orchestration and AI-driven automation layers. By employing hypervisor software atop physical servers, this layer enables workload consolidation, resource pooling, and strict isolation of tenant or application environments. Figure 3.3 presents a detailed schematic of the Virtualization Layer, illustrating multiple hypervisor nodes, virtual networking constructs, and shared storage integration. In the following exposition, each element from the provided diagram is described in turn, with emphasis on inter-hypervisor communication, virtual networking links, and software-defined storage (SDS).

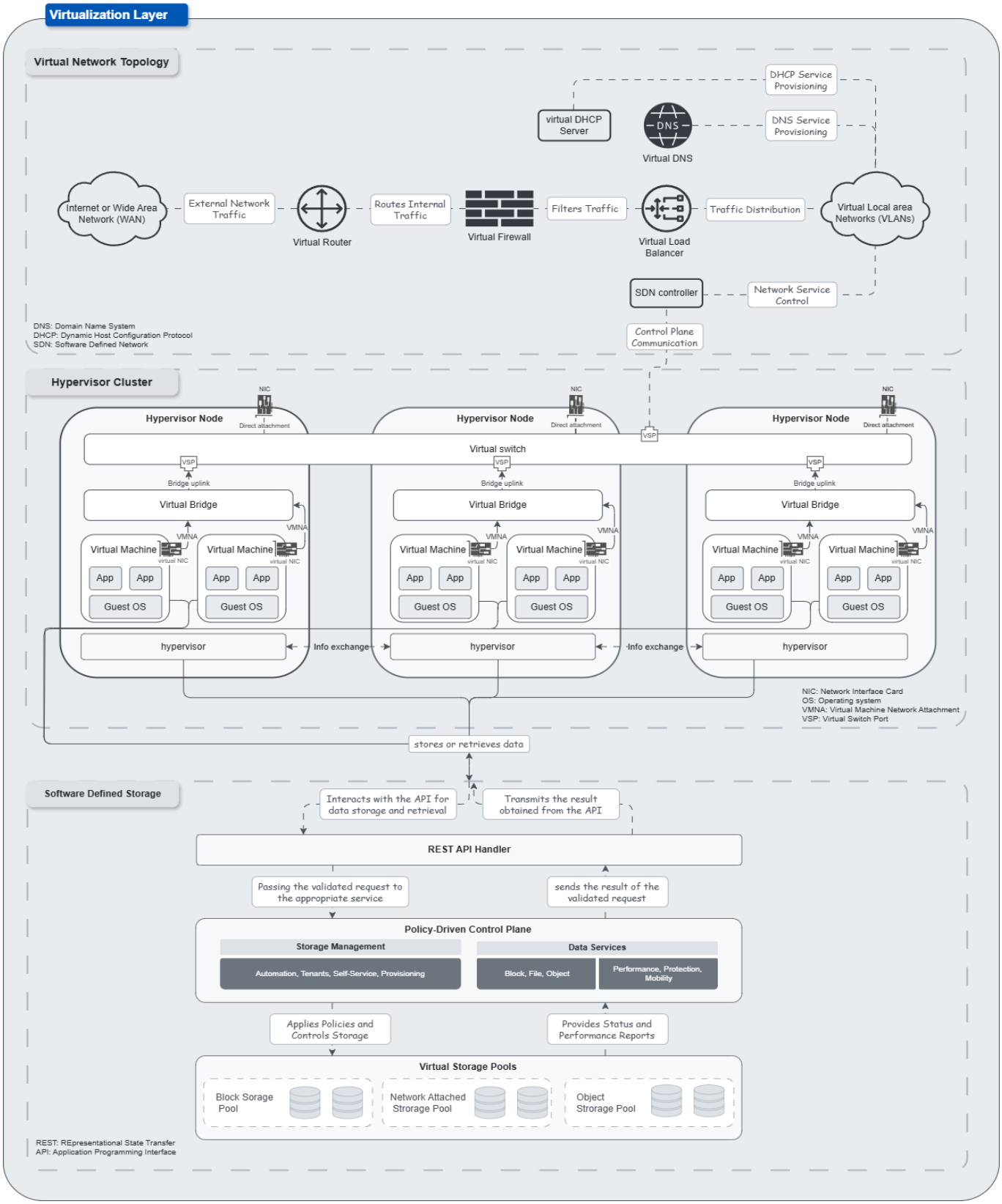

Figure 3.3: Virtualization-Layer

As illustrated in Figure 3.3, the virtualization layer is composed of the following core components:

- **Hypervisor Nodes**: Each hypervisor node represents a physical server running a Type-1 hypervisor. Within each node:
  - **Hypervisor Software**:
    - Abstracts CPU, RAM, and I/O resources into virtual partitions (vCPUs, vRAM, virtual disks, vNICs).
    - Manages VM lifecycle operations (creation, deletion, suspension, resumption, live migration, snapshotting).
    - Enforces isolation and security contexts so that each VM's memory and I/O remain segregated from co-resident VMs.
    - Implements scheduling algorithms to multiplex physical CPU cores and memory ballooning for dynamic RAM allocation.
  - **Virtual Machine Instances (VMs)**: Each VM executes as a logically isolated server, comprising:
    - **Guest Operating System**: A complete OS instance running atop virtualized hardware.
    - **Application Workloads**: One or more applications or services ("App" blocks) that run on the Guest OS and generate CPU, memory, and I/O demands.
    - **Virtual Devices**:
      - *vCPU(s)*: Virtual CPU threads mapped by the hypervisor onto physical CPU cores.
      - *vRAM*: Virtual memory allocated to the VM, backed either by physical RAM or by swap files on the SDS pool.
      - *Virtual Disk*: A block device presented as a file or logical volume on the SDS engine.
      - *vNIC*: One or more virtual network interfaces that connect the VM to the local Virtual Bridge.
  - **Hypervisor Cluster Communication**:
    - *Cluster Bus for Heartbeats and Membership*: Hypervisors exchange heartbeat messages over a dedicated management network to track node health and resource availability.
    - *Live Migration Channel*: When migrating a VM, the source hypervisor streams memory pages and CPU state to the target hypervisor via this channel, enabling near-zero-downtime relocation.
    - *Shared Resource Usage Metrics*: Hypervisors periodically share aggregate CPU load, memory pressure, and network I/O statistics to facilitate global load balancing and high availability.

- **Virtual Networking Components**: Each hypervisor node hosts a local Virtual Bridge that uplinks to a cluster wide Virtual Switch:
  - **Virtual Bridge**:
    * Acts as a Layer 2 switch within a single hypervisor node, interconnecting all local VMs' vNICs.
    * Forwards frames between co-located VMs without leaving the host, minimizing intra-node latency.
    * Encapsulates packets destined for remote VMs into overlay tunnel packets and forwards them to the Virtual Switch.
  - **Virtual Switch (Overlay Network)**:
    * Represents a cluster wide Layer 2 overlay that unifies Virtual Bridges across hypervisors.
    * Maintains a distributed MAC/IP-to-VNI mapping, managed by an SDN controller or distributed control plane.
    * Enables seamless inter-node VM communication:
      1. VM A's vNIC transmits a frame to its local Virtual Bridge.
      2. The Virtual Bridge encapsulates the frame into a tunnel packet and sends it to the Virtual Switch overlay.
      3. The Virtual Switch decapsulates on the destination hypervisor and delivers it via that host's Virtual Bridge to VM B's vNIC.
    * Supports network policy enforcement, namespace segmentation, and overlay encryption without modifying physical switch configurations.
  - **Virtual NICs**:
    * Implemented via paravirtualized drivers or emulated NICs inside each VM.
    * Outgoing path: Application in Guest OS → vNIC driver → hypervisor emulation → Virtual Bridge.
    * Incoming path: Overlay packet arrives at Virtual Switch → forwarded to Virtual Bridge → delivered to VM's vNIC → Guest OS processes it.
- **Inter-Hypervisor Cluster Communication**: In a cloud environment, the communication between hypervisors is crucial for tasks like virtual machine migration, load balancing, and ensuring high availability. It allows the hypervisors to share information about resource usage and the status of virtual machines to maintain system stability and optimize performance. Specifically:
  - *Heartbeat and Membership Management*: A Cluster Membership Daemon on each hypervisor exchanges periodic heartbeat messages over the management

network. Failure to receive a heartbeat within a specified interval triggers automatic HA failover for hosted VMs.

- *Live Migration Coordination*: During a live migration, the source hypervisor transfers memory pages and CPU state incrementally to the target hypervisor across the inter-hypervisor channel. This minimizes downtime by only pausing the VM for a final small state copy.
- *Load-Balancing Information Exchange*: Hypervisors transmit real-time metrics (e.g., CPU utilization, memory usage, network throughput) over this channel, enabling higher-layer schedulers (e.g., Kubernetes or AI provisioning agents) to redistribute workloads and avoid hotspots.
- *SDS Metadata Synchronization*: Because shared storage is essential for a cloud environment, hypervisors replicate and synchronize SDS metadata (e.g., volume allocations, replication status, snapshot catalogs) via the inter-hypervisor network. This coordination ensures data consistency and supports VM mobility and data protection.

- **Software-Defined Storage (SDS) Integration**: Shared storage is essential for a cloud environment. It allows multiple hypervisors to access the same data, which is necessary for features like virtual machine mobility and data protection. It simplifies management and ensures data consistency across the infrastructure. The SDS layer comprises:
  - *Distributed Block Storage Pool*: Each hypervisor contributes local disks (HDDs / SSDs) to a distributed SDS cluster (e.g., Ceph, GlusterFS, or Proxmox ZFS). The collective pool appears as a single logical storage volume.
  - *Replication and Redundancy*: Data is replicated across multiple hypervisor nodes according to a configurable replication factor guaranteeing durability and facilitating rapid recovery from node failures.
  - *Thin Provisioning and Snapshots*: VM virtual disks are allocated on-demand (thin-provisioned), and block-level snapshots enable instant VM backups and rapid clone creation for test or scaling purposes.
  - *QoS and I/O Scheduling*: The SDS engine enforces I/O rate limits per VM to mitigate noisy neighbor effects and ensures that high-priority workloads receive sufficient bandwidth and low latency.
  - *Shared Storage Access Path*:
    1. When a VM issues a read/write operation, the hypervisor's I/O layer forwards the request to the SDS engine over the storage network.

2. The SDS engine retrieves or commits data from the appropriate replica on physical disks and returns it to the hypervisor, which in turn delivers it to the VM's virtual disk driver.
3. Because all hypervisors draw from the same SDS pool, a VM image can remain consistent when migrated.

This detailed virtualization substrate comprising hypervisor nodes, VMs, virtual networking components, inter hypervisor communication, and shared SDS enables:

- **Elastic Resource Partitioning**: VMs can be flexibly assigned the right amount of CPU, memory, and storage, and new ones can be launched quickly on demand.
- **Strict Isolation and Security**: Each VM's compute, memory, network, and storage contexts are isolated from co resident VMs, ensuring fault containment and multi tenant security.
- **High Availability and Live Migration**: Hypervisor nodes can live-migrate VMs with minimal downtime. SDS replication ensures that VM disks remain available even if a hypervisor fails.
- **Programmable Network Topology**: Virtual Bridges and the cluster-wide Virtual Switch overlay decouple VM communication from physical network topology, facilitating seamless VM mobility and network policy enforcement.
- **Unified Storage Management**: The SDS pool provides a consistent storage interface for all hypervisors, simplifying VM provisioning, cloning, and snapshot workflows.

By implementing this comprehensive Virtualization Layer faithfully reflecting the provided architecture diagram the system establishes a robust, flexible, and programmable foundation upon which the Container Orchestration Layer and subsequent Monitoring & Automation and Agentic AI layers build proactive, intelligent resource management capabilities.

3.3.3 Orchestration Layer

The Orchestration Layer represents the strategic control tier of the cloud-native infrastructure. It is tasked with declarative workload management, service discovery, container scheduling, failure recovery, and scalability automation. This layer abstracts away the complexity of container lifecycle management and multi-host coordination by providing a distributed control plane and a uniform interface for users and operators. It sits atop the

virtualization substrate and governs the execution of containerized applications across distributed compute resources.

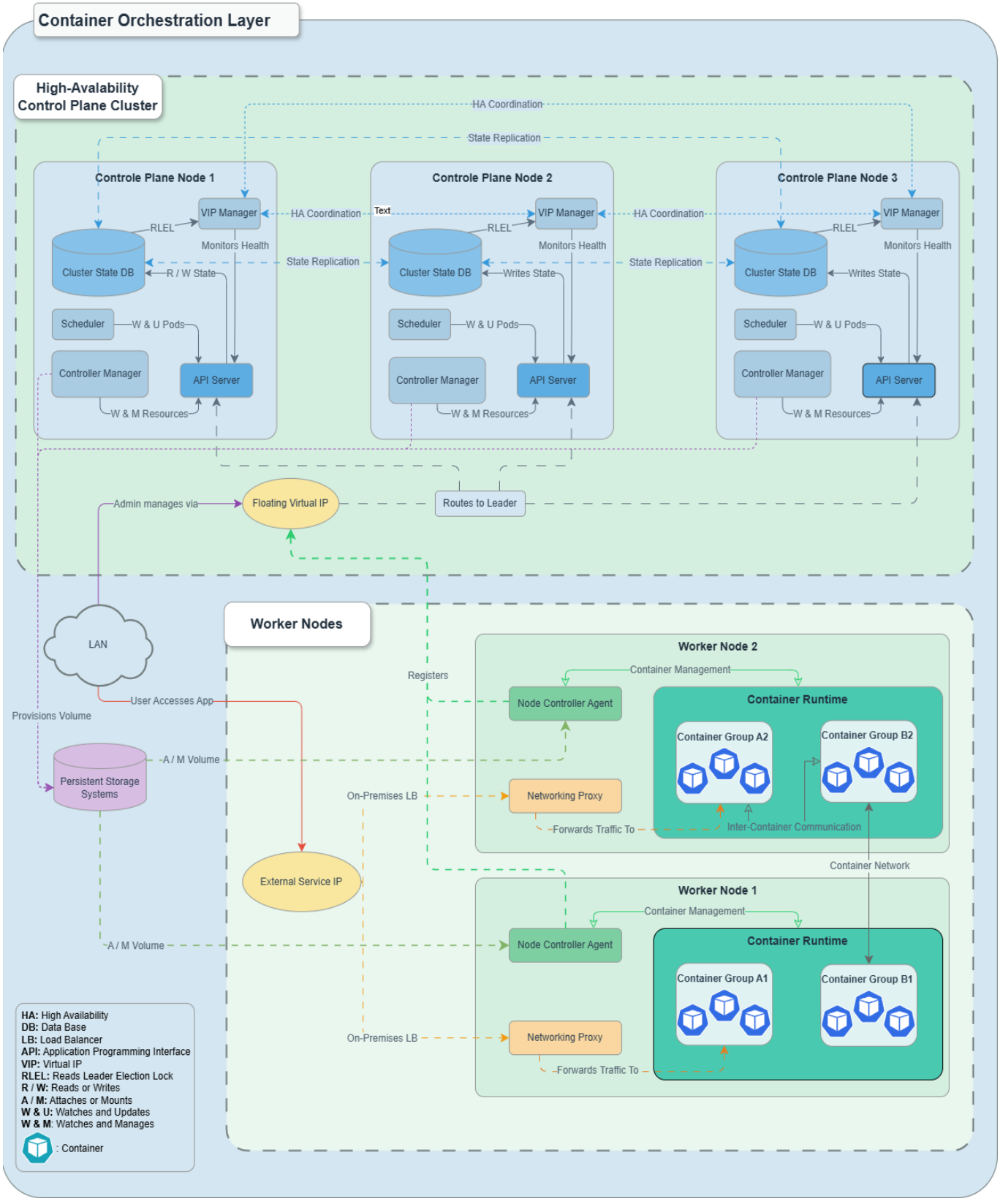

Figure 3.4: Container-Orchestration-Layer

As illustrated in Figure 3.4, this layer is composed of several logically interconnected modules distributed across control plane and worker nodes:

- **High-Availability (HA) Control Plane Cluster**: The control plane governs the entire cluster state and ensures consistent decision-making using distributed consensus. It comprises three primary components per node:
  - **Cluster State Database (e.g., etcd)**: A highly consistent and fault-tolerant key-value store used to persist the entire cluster configuration, including metadata, resource states, workload definitions, and secrets.
  - **API Server**: Serves as the entry point to the control plane. It authenticates requests, validates schemas, and exposes RESTful interfaces for cluster interactions. All internal components interact through the API server.
  - **Scheduler and Controller Manager**: The scheduler assigns pods to suitable nodes based on current resource availability and placement policies. The controller manager executes control loops that ensure the actual cluster state converges to the desired state defined in the control database.
  - **VIP Manager and Leader Election Mechanism**: Among the control plane nodes, one is elected as the leader using a distributed leader election algorithm (e.g., based on leases in etcd). Only the elected leader node has the authority to provision volumes, ensuring data consistency and avoiding concurrent writes to storage backends.
- **Floating Virtual IP (VIP)**: A highly available virtual IP abstracts away the physical location of the API server. It dynamically redirects administrative traffic to the current leader, maintaining seamless accessibility in the event of control node failover. The VIP routes are automatically updated to reflect leadership changes.
- **Worker Nodes**: Each worker node is a runtime host responsible for executing containerized workloads. It includes:
  - **Node Controller Agent (e.g., kubelet)**: This daemon registers the node with the control plane and periodically reports health metrics. It ensures that pods assigned to the node conform to the specifications provided by the control plane.
  - **Container Runtime**: This module (e.g., containerd or CRI-O) manages the low-level lifecycle of containers, including image retrieval, instantiation, execution, and teardown.
  - **Networking Proxy (e.g., kube-proxy)**: Manages the virtual networking rules for each node, enabling load balancing across services and maintaining consistent network policies.

  - **Container Groups (Pods)**: The smallest deployable units in the orchestration layer, pods encapsulate one or more tightly coupled containers that share the same IP namespace and persistent volumes. These groups are orchestrated as single units for scaling, health checks, and network isolation.

- **Communication Pathways and Relationships**:
  - **Control Plane Coordination**: Each control node maintains continuous coordination with its peers via health checks, state replication, and HA leadership protocols.
  - **Cross-Node Communication**: As shown in the diagram, a subset of container groups (pods) engage in inter-node communication, essential for distributed microservices. This is not a universal requirement; many pods operate independently on their own nodes.
  - **Inter-Container Communication**: Within each pod, containers communicate through localhost interfaces and shared volumes, providing tight integration and minimal latency.
  - **Container Networking**: Containers that span across pods or nodes use the overlay container network (e.g., Flannel, Calico) for communication, supporting service discovery and DNS resolution.
  - **External Service IP**: To enable access from outside the cluster (e.g., user traffic or API requests), services can be exposed through external IPs or ingress controllers, backed by a load balancer.

- **Persistent Storage Systems**: These systems allow pods to retain state across restarts and provide volume mounts abstracted from the physical storage backends. The provisioning of such volumes is coordinated exclusively by the elected control plane leader to prevent race conditions and ensure integrity in concurrent environments. Storage classes and volume plugins (e.g., CSI) abstract the physical storage details and provide dynamic provisioning capabilities.

Conception Note: Only the leader in the control plane cluster is authorized to provision persistent volumes. This constraint is imposed by the architecture of distributed consensus systems, which aim to maintain strong consistency in shared state across concurrent actors. Furthermore, the figure selectively illustrates a single cross-node and inter-pod communication channel to highlight that not all container groups participate in such relationships. Each container group must engage in at least one of the following:

- Inter-Container Communication (within the same pod).

- Container Network Communication (across pods or nodes).
- Exposure via an External Service IP (for communication with clients outside the cluster).

This layered architecture enforces high availability, elasticity, and declarative governance. It decouples applications from infrastructure, facilitates autonomous recovery, and supports zero-downtime updates. As such, it is a cornerstone of modern cloud-native systems.

3.3.4 Monitoring, Management, and Automation Layer

The Monitoring, Management, and Automation Layer represents the uppermost logical tier in the architecture, dedicated to ensuring visibility, operability, and adaptive control over the virtualized and orchestrated infrastructure. It integrates telemetry collection, rule-based evaluation, user or machine-triggered automation. This layer enables administrators to observe real-time metrics, correlate historical trends, execute mitigation strategies, and automate recurrent workflows, forming the backbone of intelligent infrastructure management.

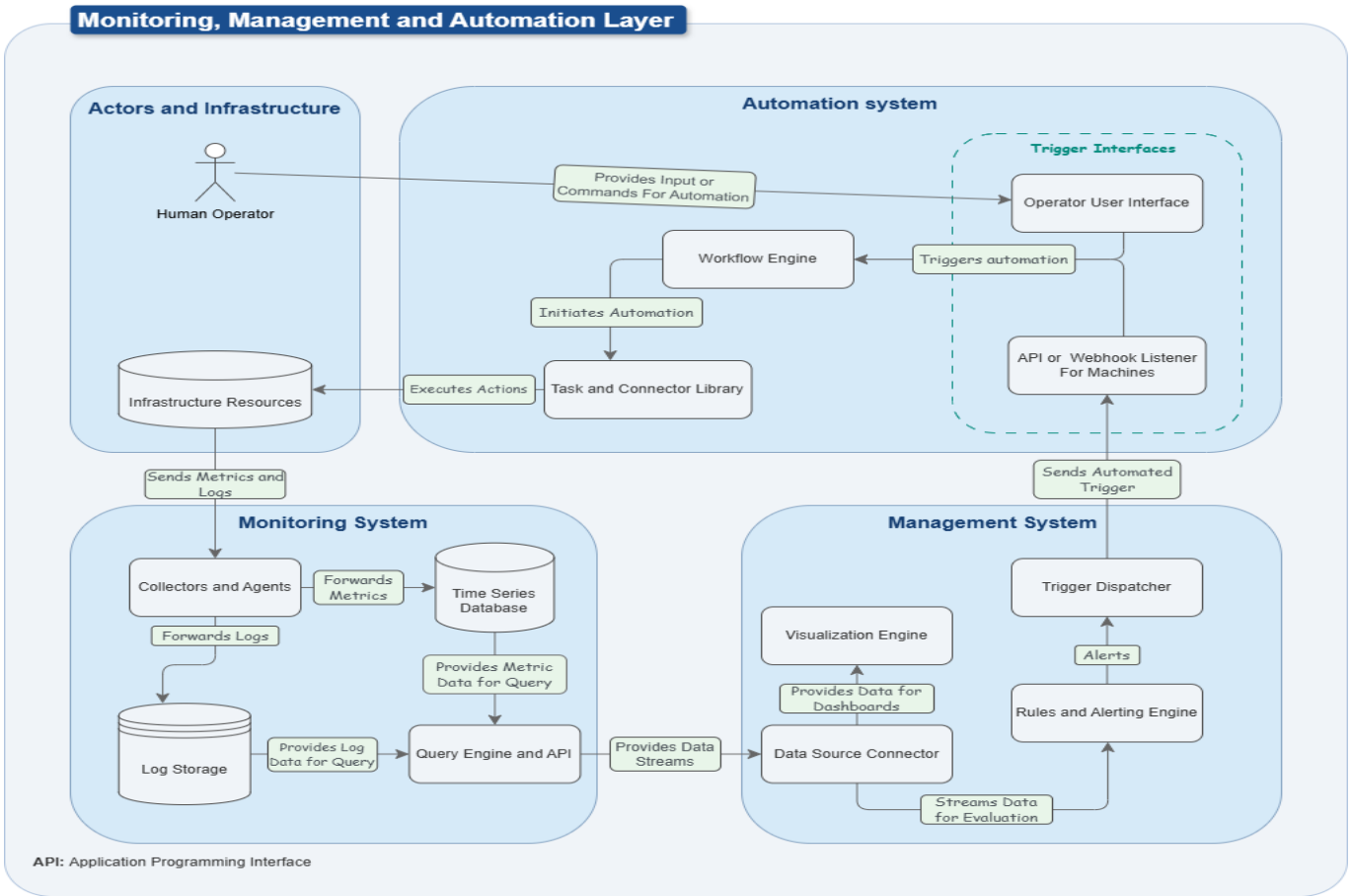

Figure 3.5: Monitoring, Management, and Automation Layer

As depicted in Figure 3.5, this layer is composed of four interlinked subsystems, each fulfilling distinct responsibilities:

- **Actors and Infrastructure**:
  - **Infrastructure Resources**: Represents the monitored targets, including compute nodes, containers, VMs, storage subsystems, and network components. These resources emit operational telemetry and logs, which serve as the raw input for downstream systems.
- **Monitoring System**:
  - **Collectors and Agents**: Lightweight daemons or sidecars deployed across infrastructure nodes that gather metrics (e.g., CPU usage, memory consumption) and logs (e.g., systemd, application logs) from the underlying environment.
  - **Log Storage**: A centralized system for ingesting and persisting log data. This storage backend supports index-based querying and time-range slicing for root cause analysis and compliance audits.
  - **Time Series Database (TSDB)**: A highly performant database optimized for handling metric streams with high cardinality and temporal granularity. It enables long-term retention of performance metrics.
  - **Query Engine and API**: Provides APIs for fetching metrics or logs from their respective backends. These interfaces are often consumed by alerting rules, visualization dashboards, and external analytics tools.
- **Management System**:
  - **Data Source Connector**: Integrates with external sources such as TSDBs, log aggregators, or cloud APIs. It streams structured telemetry into the management subsystem for further processing.
  - **Rules and Alerting Engine**: Evaluates user-defined thresholds and behavioral conditions on the incoming data. When predefined criteria are met, it emits alerts to trigger responsive actions.
  - **Trigger Dispatcher**: Acts as a broker that interprets and routes alerts into actionable formats-either to human interfaces (e.g., email dashboards) or automation pipelines.
  - **Visualization Engine**: Converts queried data into graphical dashboards, gauges, and time series charts for human inspection and decision support.

- **Automation System**:
  - **Trigger Interfaces**:
    - **Operator User Interface**: Provides a GUI for manual interaction, such as triggering workflows, injecting parameters, or acknowledging alerts.
    - **API or Webhook Listener for Machines**: Offers programmable endpoints to receive events from upstream systems or CI/CD pipelines, enabling event-driven automation.
  - **Workflow Engine**: Serves as the central logic executor. It consumes triggers, validates preconditions, and orchestrates tasks defined in structured workflows (e.g., YAML or DSL).
  - **Task and Connector Library**: A repository of predefined scripts, modules, or APIs capable of interfacing with infrastructure resources. These tasks include provisioning resources, restarting pods, scaling nodes, or notifying external systems.

Operational Semantics and Flow:

1. Infrastructure resources emit metrics and logs to the collectors, which forward them to dedicated storage backends (log storage and TSDB).
2. The monitoring system exposes these data streams to the query API, which the management system uses to continuously evaluate rules and thresholds.
3. When a rule is triggered (e.g., high CPU usage on a worker node), the alerting engine forwards an event to the trigger dispatcher.
4. The dispatcher invokes the automation system, where either a human operator or an API listener provides contextual inputs.
5. The workflow engine initiates a predefined response (e.g., scale out a container group), using its task library to communicate back with infrastructure resources.

This closed-loop control design ensures rapid feedback and correction in dynamic environments, enabling predictive scaling, fault self-healing, and minimal operator burden. It also supports observability best practices, such as the "four golden signals" (latency, traffic, errors, saturation).

3.3.5 Agentic Layer

The Agentic Layer introduces a novel abstraction in the architecture: a Smart Resource Provisioning and Prediction subsystem built upon a Multi-Agent System (MAS) paradigm. In this context, agents are autonomous, collaborative software entities empowered with specialized capabilities such as perception, reasoning, and actuation. These agents collectively orchestrate the translation of high-level user intents into deployable infrastructure artifacts through coordination and distributed task decomposition.
This layer leverages advanced LLM-based agents that combine language understanding with tool integration. As detailed in Figure 3.6, the system uses three agents for information gathering, analysis, and synthesis. This intelligent workflow translates user guidance into automated actions by grounding requests in the current infrastructure state. To codify best practices and ensure accuracy, the system draws from distributed knowledge sources, most notably using Retrieval-Augmented Generation (RAG) to query an internal documentation database. This allows agents to supplement their knowledge with verified standards from trusted documents and public code, producing reliable, context-aware artifacts.
The inclusion of this layer enables the system to operate with a high degree of proactivity and adaptability, allowing for context-sensitive provisioning, automated configuration generation, and validation against both runtime state and best-practice guidelines. The following subsections detail the responsibilities and internal mechanisms of each agent comprising the MAS.
The workflow, as shown in Figure 3.6, begins with a user prompt, followed by sequential processing through three agents:

- Agent 1: Chat Validator and Data Collector (Collector)
- Agent 2: Analysis and Strategy Agent (Analyser)
- Agent 3: Build and Refinement Agent (Builder)

Together, they deliver a complete transformation of high-level natural language intent into deployable, context-aware infrastructure manifests.

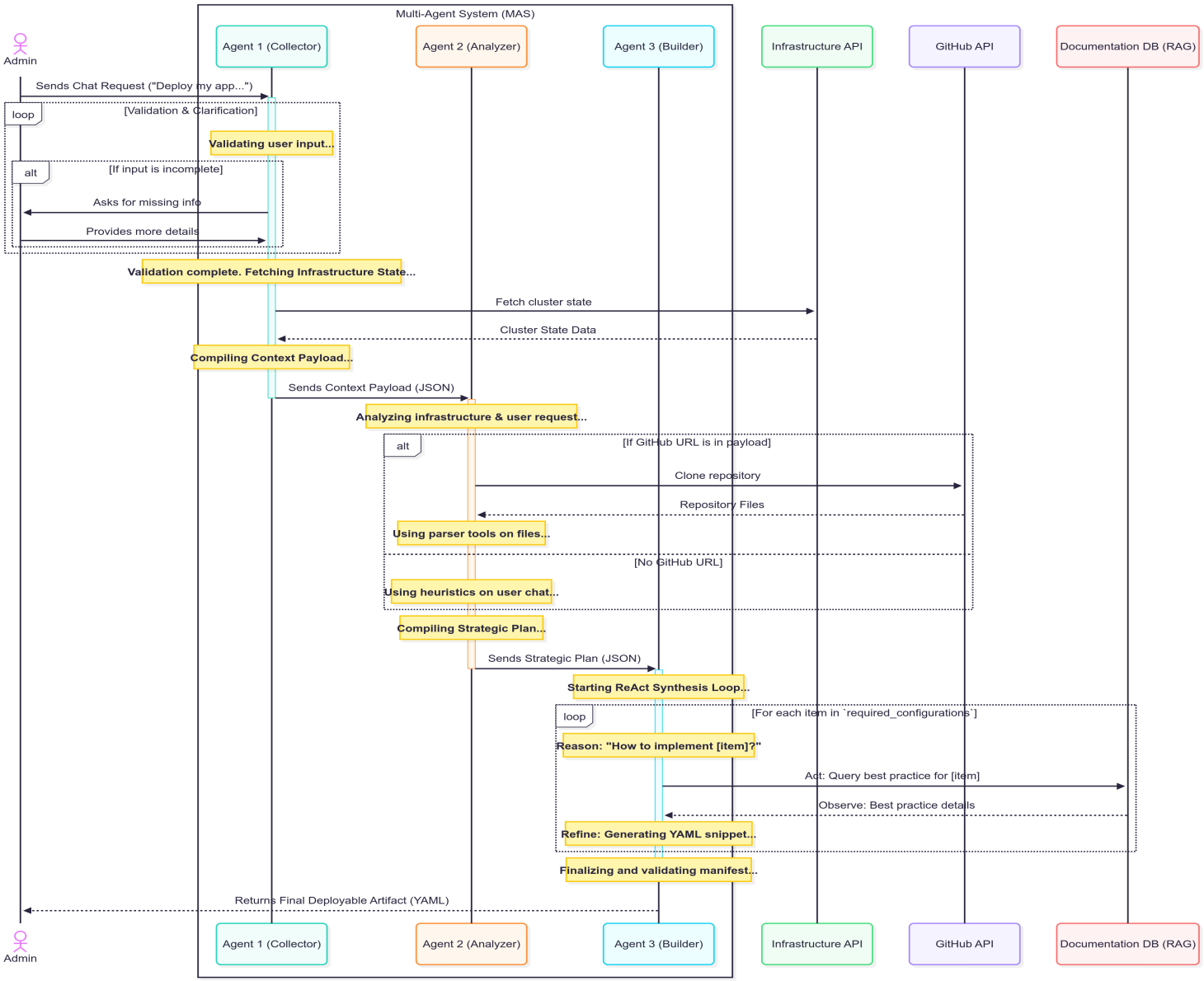

Figure 3.6: Agentic Workflow Coordination in the Multi-Agent System (MAS)

The sections below present each agent's role, internal architecture, and operational logic.

Agent 1: Chat Validator and Data Collector Agent (Collector)

The first component of the Multi-Agent System (MAS) is the Chat Validator and Data Collector Agent. This agent functions as the system's front-line interface for user interaction and context acquisition. It plays a dual role: (i) verifying the validity and feasibility of the user's infrastructure request, and (ii) enriching the contextual understanding of that request by gathering platform-specific information. Its architecture embodies a reasoning–validation–enrichment pipeline and serves as the initiator of autonomous task orchestration within the agentic subsystem.

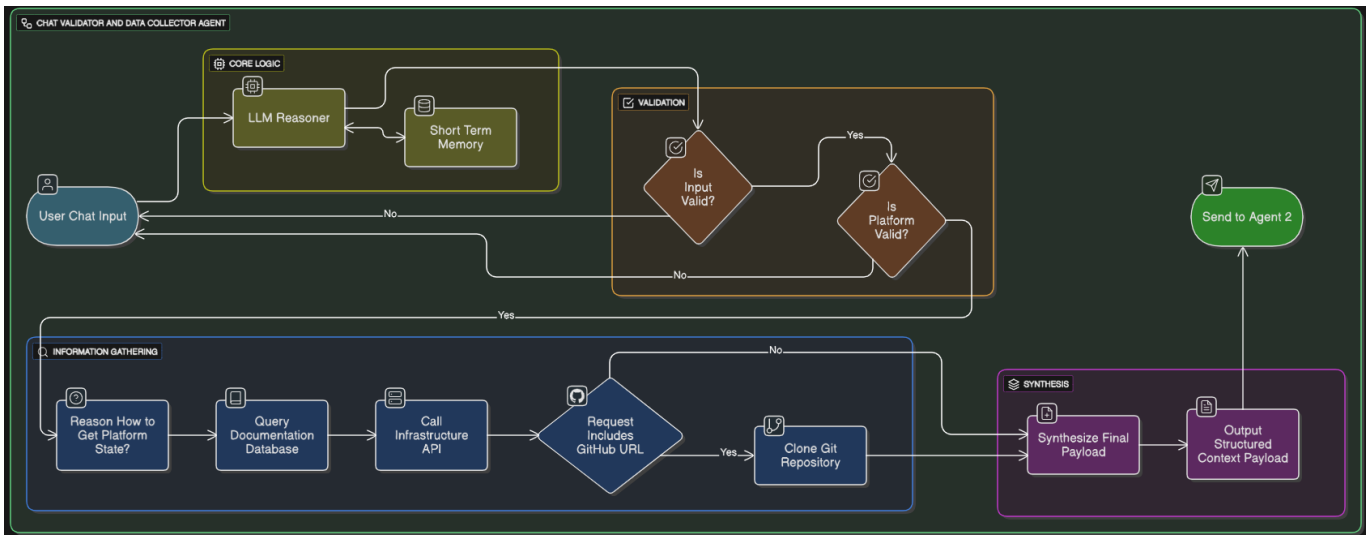

Figure 3.7: Agent 1 - Chat Validator and Data Collector Workflow

As illustrated in Figure 3.7, the agent comprises four tightly integrated modules: Core Logic, Validation, Information Gathering, and Synthesis.

- **Core Logic Module**:
  - The interaction begins with the user submitting a free-form infrastructure-related request via natural language.
  - The *LLM Reasoner*, guided by prompt engineering and few-shot instruction templates, parses and interprets the intent of the user input.
  - This reasoning process is enriched by a **Short-Term Memory** store that retains temporary context such as previously resolved platform identifiers, API credentials and human interaction.
  - The reasoning output is passed downstream for structural validation.

- **Validation Module**:
  - This module performs two independent validation checks:
    1. **Input Validity Check**: Ensures that the user query contains sufficient structural and semantic detail to be processed meaningfully (e.g., specifies a platform, objective, or operational target).
    2. **Platform Validity Check**: Verifies whether the referenced platform exists, is supported, or can be programmatically interrogated using available APIs.
  - Invalid or ambiguous inputs are routed back to the user with clarification prompts, allowing a feedback loop that enhances resilience to incomplete requests.
- **Information Gathering Module**:
  - Once the input is validated, the agent attempts to enrich the input context through multiple data sources:
    1. **Platform State Reasoning**: The agent queries its internal reasoning module to infer the best method to retrieve current infrastructure state (e.g., using an API call or inspecting a repository).
    2. **Querying Documentation Database**: A semantic search is conducted across indexed infrastructure documentation to extract deployment requirements, constraints, and default behaviors.
    3. **Calling Infrastructure API**: The agent retrieves real-time metadata such as running workloads, resource quotas, or node statuses from the orchestrator or cloud API.
    4. **GitHub Repository Cloning**: If the user's request includes a GitHub URL, the agent initiates a repository cloning operation and parses the content (e.g., Dockerfiles, manifests, Terraform plans) to extract relevant deployment descriptors.
  - These steps are conditionally activated based on the presence of signals in the user input and platform metadata.
- **Synthesis Module**:
  - All retrieved and inferred information is aggregated into a cohesive and structured format.
  - The agent synthesizes this data into a **Final Payload**, which includes:
    * Extracted user intent.

    - Validated platform target.
    - Relevant infrastructure metadata.
    - Parsed configuration or code excerpts (if applicable).
  - The payload is serialized into a **Structured Context Payload**, which conforms to a schema interpretable by Agent 2 (e.g., JSON or YAML).
  - This payload is emitted via the "Send to Agent 2" trigger, formally transitioning execution to the next agent in the MAS pipeline.

This agent encapsulates the principles of reactive planning and contextual inference. Its modular structure ensures adaptability across domains, enabling it to generalize input handling logic and support new platform integrations. Through layered validation, structured enrichment, and systematic reasoning, Agent 1 ensures that subsequent agents operate on a robust and semantically coherent foundation.

Agent 2: Analysis and Strategy Agent (Analyser)

The second stage of the Multi-Agent System (MAS) is is the Analysis and Strategy Agent, which functions as the analytical core. The agent is responsible for:

- interprets the structured payload and formulates an execution strategy based on the application characteristics and infrastructure status.
- Synthesizing deployment goals into a machine-readable provisioning blueprint bridges descriptive context with actionable orchestration.

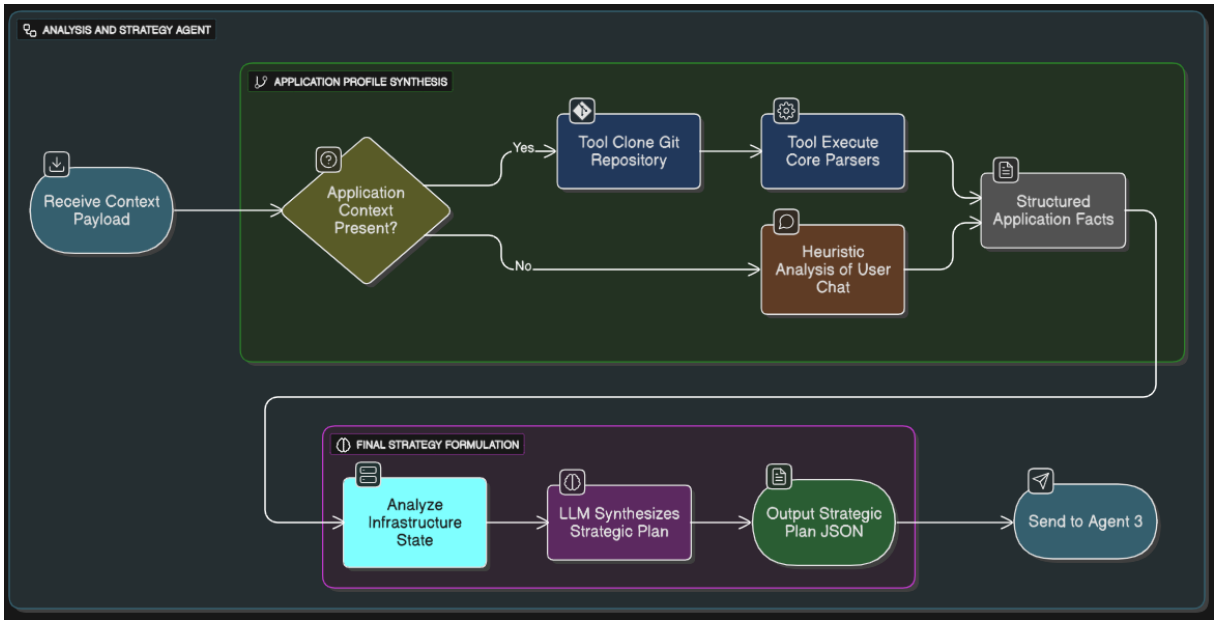

Figure 3.8: Agent 2 - Analysis and Strategy Formulation Workflow

As shown in Figure 3.8, the workflow for this agent unfolds across two major functional modules: Application Profile Synthesis and Final Strategy Formulation.

- **Application Profile Synthesis**:
  - Upon receiving the structured context payload from Agent 1, the agent determines whether the payload contains explicit application artifacts or configuration references-termed *application context*.
  - If such context is present:
    - A Git repository referenced in the payload is cloned using an embedded version control interface.
    - A series of domain-specific **Core Parsers** are then executed over the repository content. These may include parsers for Kubernetes manifests, Helm charts, Dockerfiles, Terraform plans, or application-specific metadata (e.g., port bindings, environment variables).
    - The result of this pipeline is a consolidated map of **Structured Application Facts**, which describe the deployment unit's characteristics, dependencies, and topology.
  - If no application context is explicitly present:
    - A **Heuristic Analysis** is performed over the original user chat transcript embedded in the payload.
    - This fallback mechanism uses prompt-based classification and entity extraction to infer likely deployment goals (e.g., "containerize a web API," "provision a GPU node," or "configure auto-scaling for a workload").
    - The extracted facts are normalized into the same format used by the parser pipeline, ensuring uniformity in downstream strategy formulation.
- **Final Strategy Formulation**:
  - With application facts synthesized, the agent transitions to generating an optimal execution strategy based on environmental conditions.
  - It initiates an **Infrastructure State Analysis** by interfacing with the virtualization and orchestration layers to evaluate:
    - Node capacities and role distribution.
    - Resource utilization trends (CPU, memory, GPU, disk I/O).
    - Feature availability (e.g., auto-scaling, storage classes, affinity rules).
  - The derived system state, in conjunction with the application facts, is fed into an LLM-powered **Strategic Planner**.

    - The planner generates a deployment strategy that aligns functional objectives with system constraints and best practices.
    - For example, it might recommend placing the application on nodes with SSD-backed volumes for performance, or enabling horizontal pod autoscaling based on CPU thresholds.
  - The output is a machine-interpretable **Strategic Plan JSON**, which encodes the deployment logic, resource mappings, and target policies.
  - This plan is then forwarded to Agent 3 for transformation into executable infrastructure code or declarative manifests.

This agent embodies the principle of strategic reasoning by adapting deployment intent to real-world execution environments. It balances static intent with dynamic context, allowing for decisions that are both goal-oriented and system-aware. The separation between application analysis and infrastructure diagnosis ensures modularity, while the use of LLMs enables flexibility in handling both explicit and inferred specifications.

Agent 3: Build and Refinement Agent (Builder)

The third and final component in the Multi-Agent System (MAS) pipeline is the Build and Refinement Agent. This agent receives the high-level deployment strategy from the previous agent and transforms it into a deployable infrastructure artifact-typically a structured declarative manifest. It employs an iterative refinement process to resolve missing or ambiguous configuration details, leveraging LLM reasoning and external documentation sources. This agent embodies the principles of the ReAct framework (Reasoning + Acting), performing incremental synthesis and validation cycles before delivering a complete infrastructure blueprint to the user.

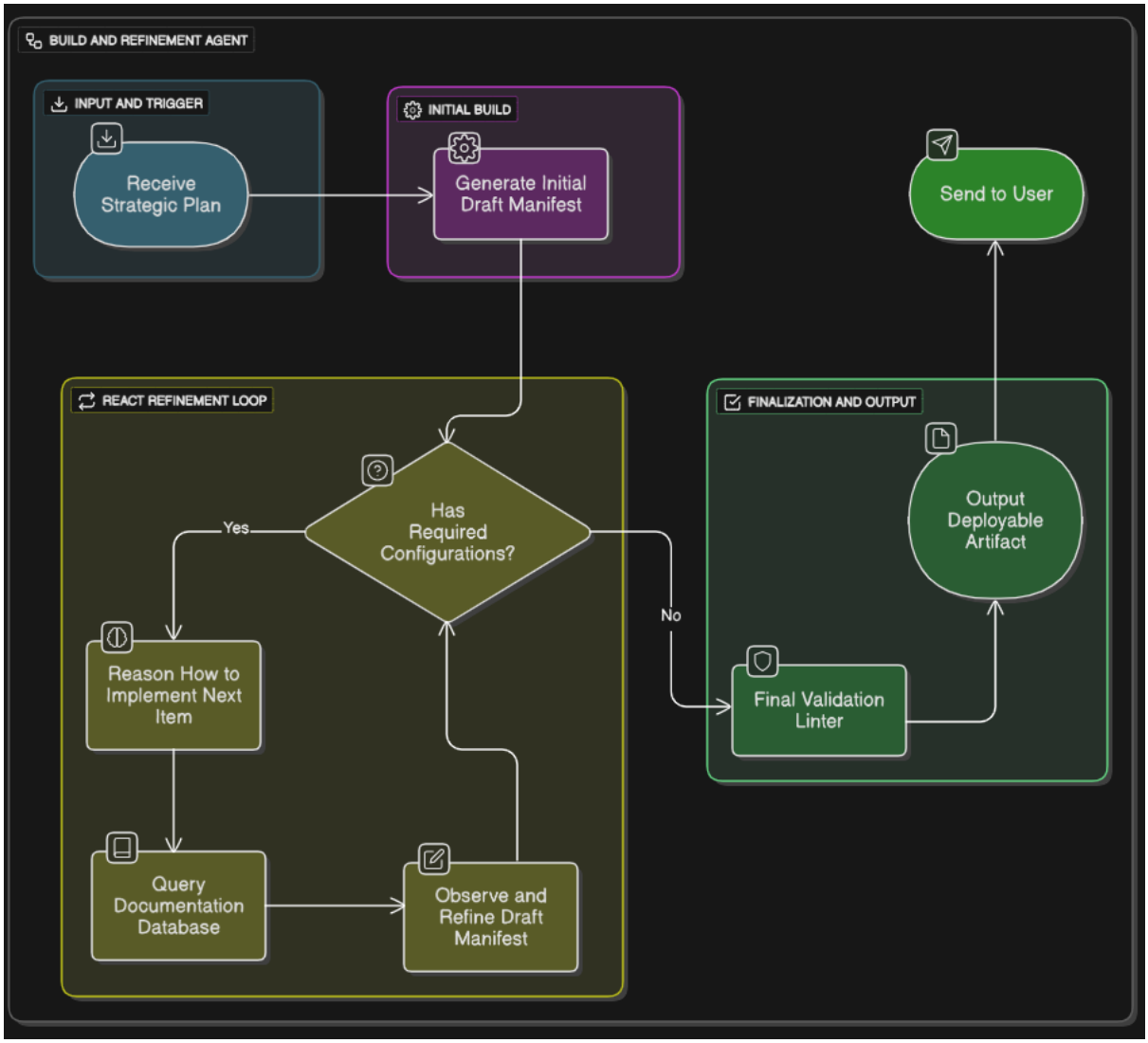

Figure 3.9: Agent 3 - Build and Refinement Workflow

As shown in Figure 3.9, the agent operates through four main stages: Input and Trigger, Initial Build, ReAct Refinement Loop, and Finalization and Output.

- **Input and Trigger**:
  - The agent begins by receiving a **Strategic Plan JSON** produced by Agent 2. This document encodes the logical structure of the intended deployment, including resource requirements, scaling rules, access policies, and topology hints.
  - The input serves as the declarative specification that guides manifest generation.
- **Initial Build**:

  - A preliminary configuration artifact is generated by the LLM module using prompt templates designed for YAML or JSON synthesis. This includes:
    * Kubernetes manifests (e.g., Deployment, Service, Ingress).
    * Terraform modules for infrastructure provisioning.
    * Helm charts or Docker Compose files, depending on the target platform.
  - The resulting **Initial Draft Manifest** is structurally sound but may contain gaps, TODO fields, or ambiguous parameters.

- **ReAct Refinement Loop**:
  - The agent enters an iterative loop designed to resolve outstanding configuration requirements.
  - It first checks if the manifest contains all required configuration entries (e.g., image paths, volume mounts, CPU limits, secrets).
  - If incomplete:
    * The LLM invokes a reasoning step ("Reason How to Implement Next Item"), leveraging prior context and predefined prompt scaffolds.
    * If external clarification is needed, the agent queries a **Documentation Database** (e.g., Kubernetes docs, Terraform Registry, Helm Hub) using semantic search.
    * Based on retrieved information, the draft manifest is updated with refined, context-aware configurations.
  - This loop repeats until the configuration satisfies completeness constraints or reaches a predefined iteration limit.

- **Finalization and Output**:
  - Once the manifest is complete, the agent invokes a **Final Validation Linter** to:
    * Check YAML/JSON schema validity.
    * Verify platform compatibility and resource references.
    * Detect configuration anti-patterns or deprecated fields.
  - Upon successful validation, the agent emits an **Output Deployable Artifact** a production-ready file suitable for direct submission to an orchestrator or IaC pipeline.
  - This final artifact is then relayed to the user interface, completing the MAS pipeline.

This agent exemplifies the principle of assisted synthesis through self-correction and semantic grounding. Its integration of reasoning, tool use, and reactive correction ensures that even incomplete strategic specifications can be iteratively expanded into robust, executable deployment manifests. By embedding domain-specific validation and documentation access into the generation loop, the system achieves high confidence in the output's correctness and applicability.

3.4 Workload and Communication Flows Across Layers

3.4.1 Synergy Between Layers

The five-layer AI-driven cloud architecture operates through continuous *vertical* communication flows that integrate control directives and feedback across all tiers. This design establishes a closed-loop system: workload events trigger downward control signals for resource management, while observational data propagates upward for analysis. Through these bidirectional interactions, the stack dynamically balances demand and capacity, ensuring responsive and adaptive behavior.
When a new user or application workload enters the system for example, a surge in client requests or the deployment of a new service the orchestration layer mediates its accommodation by issuing scheduling and placement decisions. The Container Orchestration Layer's control plane translates high-level intents (such as desired application replicas or service-level objectives) into concrete actions on the infrastructure. In response to increased load, the orchestrator may launch additional container instances or adjust the placement of running workloads to maintain performance.

3.4.2 Communication Flows

These control directives flow downward to the Virtualization and Physical layers in the form of resource allocation requests. For instance, if the scheduler determines that current VMs lack sufficient capacity to host new containers, it can trigger the provisioning of a new VM (via the automation interface) on the Physical Layer. Each such action translates abstract demand into concrete infrastructure changes: containers are instantiated on selected VMs, new VMs are launched on physical hosts, and networking routes are configured accordingly. In essence, the lower layers execute the *triggering* commands that are coming from above whether it is deploying a container, scaling out a VM cluster, or reconfiguring a network path thereby ensuring that incoming workloads are placed and served by adequate resources in real time.

Concurrently, each layer generates state information that flows upward as monitoring feedback. Physical servers report hardware health to the virtualization management plane. Hypervisors and VMs contribute statistics on virtual resource usage and performance. At the orchestration level, fine-grained service metrics and events (such as container resource utilization and pod lifecycle statuses) are continuously logged. The Monitoring and Automation layer aggregates and analyzes these multi-layer telemetry streams, synthesizing a holistic view of the system's state.
By correlating low-level signals with high-level service indicators, the monitoring system can detect emerging bottlenecks or anomalies. In a reactive scenario, triggers could invoke scripts to scale out a service tier (adding container replicas) or allocate additional infrastructure to relieve the detected stress. In this way, the upward flow of telemetry closes the loop by informing higher layers of current conditions and enabling corrective actions.

3.4.3 Embedding Predictive AI

Beyond these reactive measures, the *Agentic AI Layer* introduces intelligent, predictive control that elevates the architecture into a self-optimizing regime. This topmost layer ingests the consolidated telemetry feed and applies advanced analytics to anticipate future conditions, embodying the industry shift toward autonomous cloud operations that emphasizes preventive management over reactive fixes.
All layers remain integrally connected through well-defined interfaces that carry these control and data exchanges. This vertical integration ensures that each layer is both aware of and responsive to conditions in the others: higher layers maintain a global view of the system, while lower layers carry out localized actions under guided oversight. The result is a closed-loop feedback control system spanning from physical hardware up to the AI-driven decision logic.

This iterative sense-analyze-act cycle repeats continuously, enabling the entire cloud stack to adapt to changing conditions in real time. In summary, workload ingress triggers a cascade of provisioning and scheduling actions down the stack, and telemetry feedback rises up to enable both reactive corrections and proactive optimizations. Through such vertical interactions, the five-layer architecture behaves as a unified intelligent system that dynamically aligns resources with demand, achieving resilient and self-optimizing operations with minimal human intervention.

3.5 Conclusion

This chapter presented a comprehensive and modular conception for an intelligent, AI-augmented provisioning system, structured across five synergistic layers. Beginning with a robust Physical Layer, the design harnesses commodity hardware and programmable networking as the substrate for all upper abstractions. The Virtualization Layer then leverages hypervisor clusters to decouple workloads from hardware specifics, incorporating shared storage and inter-node communication mechanisms critical for high availability, live migration, and scalability. On this foundation, the Container Orchestration Layer introduces a distributed control plane, service discovery mechanisms, and policy-driven workload scheduling all orchestrated under a high-availability cluster with leader election to ensure consistency in volume provisioning and operational state. Key communication pathways (such as cross-node, inter-container, and external service links) are selectively illustrated to emphasize modular relationships without imposing unnecessary architectural coupling. To complement this control architecture, the Monitoring, Management, and Automation Layer closes the loop by observing infrastructure health, evaluating operational thresholds, and triggering reactive or scheduled automation workflows. This closed feedback circuit, while powerful, is inherently reactive-dependent on pre-defined rules or user input. To transcend reactive management and embrace proactive, context-aware optimization, the Agentic Layer introduces a Multi-Agent System (MAS) built around LLM-empowered agents. This final layer fundamentally transforms the infrastructure into an adaptive system capable of understanding user intents, analyzing current states, predicting future workloads, and synthesizing executable deployment artifacts.

Together, these five layers represent a shift from static infrastructure management to dynamic, intelligent orchestration-guided by principles of modularity, scalability, automation, and contextual intelligence. The architecture is not only a blueprint for system design but also a forward-looking paradigm for operating modern cloud environments under increasing complexity and performance demands.

Chapter 4

Practical Implementation

4.1 Introduction

This chapter details the practical implementation of the conceptual architecture, constructing a truly **autonomic private cloud** environment. At its core is a **cognitive automation framework** driven by a cooperative **Multi-Agent System (MAS)** designed to translate high-level human intent into optimized, machine-executable **infrastructure code**. This implementation serves as the **practical foundation for the thesis's core arguments**.

This implementation prioritizes **high availability**, **automated execution**, and **dynamic scaling**. To maintain this focus, security implementation is not included in this chapter. Instead, our primary innovation is a custom **Multi-Agent System (MAS)** composed of several specialized agents, including a **Manager**, a **DataCollectionAgent**, an **AnalysesAndStrategyAgent**, and a **ManifestRefinementLoop**. This system moves beyond static scripts by working in concert to interpret user requests, analyze the environment, and generate adaptive, context-aware deployment artifacts.

To present this complex system, the chapter is structured to mirror its **layered architecture**. It begins by detailing the **foundational layer**, including the `Proxmox VE` cluster and `Ceph` storage. Subsequently, it describes the **container orchestration and automation platform**, where `Terraform` and `Ansible` are used to deploy a high-availability `k3s` cluster, complemented by `MetalLB` and `Helm`. The chapter then details the implementation of the **Agentic AI Layer**, the capstone of the architecture, breaking down its multi-agent structure.

Finally, a series of practical workflow demonstrations, including an **end-to-end deployment**, a **fault-tolerance test**, and an **intelligent provisioning scenario**, synthesizes these components to validate the system's resilience and operational capabilities.

4.2 Technology Stack and Rationale

The technology stack for this project was selected to meet the core research objectives of building an automated, resilient, and observable on-premises cloud. Key drivers included a preference for open-source solutions, a commitment to **Infrastructure as Code (IaC)**, and the use of industry-standard tools. This section presents the chosen technologies and the rationale behind their selection.

4.2.1 Core Technology Selection

The platform's architecture is composed of several distinct but interconnected layers, which are visually detailed in the global implementation diagram in **Annex C.1** . Specific

technologies were chosen to fulfill the requirements of each layer. The complete technology stack is summarized in Table 4.1.

Table 4.1: Core Technology Stack

Architectural Layer	**Technology**	**Primary Role**	**Description**
Virtualization & Infrastructure	**Proxmox Virtual Environment (VE)**	Hypervisor, cluster management, and High Availability (HA).	An all-in-one open-source platform for enterprise virtualization, managing VMs, hosts, and HA.
Storage	**Ceph (RBD & CephFS)**	Hyper-converged, distributed, and resilient storage for VMs and backups.	A software-defined storage system that provides fault-tolerant block (RBD) and file (CephFS) storage directly from the hypervisor nodes.
Automation & Configuration	**Terraform & Ansible**	IaC for VM provisioning (Terraform) and system configuration (Ansible).	A dual-tool approach where Terraform declaratively creates the infrastructure and Ansible procedurally configures the software on top of it.
Container Orchestration	**k3s**	Lightweight, high-availability Kubernetes distribution.	A certified, lightweight Kubernetes distribution optimized for on-premises and resource-constrained environments, with simplified HA.
Networking Services	Proxmox SDN, OVS, OPNsense, MetalLB, Kube-vip	Virtual network definition (SDN) and switching (OVS), routing/firewall (OPNsense), and Kubernetes service exposure.	A multi-component stack providing the virtual network topology (Proxmox SDN), the underlying fabric (OVS), core network services (OPNsense), and Kubernetes API/application load balancing (Kube-vip/MetalLB).

Table 4.1: Core Technology Stack

Architectural Layer	**Technology**	**Primary Role**	**Description**
Application & Observability	**Helm, Prometheus, Grafana**	Kubernetes package management (Helm) and cloud-native monitoring.	Helm simplifies application deployment, while Prometheus and Grafana provide a comprehensive stack for metrics collection and visualization.
Agentic AI Layer	Agent Development Kit(ADK) and Google Gemini	Framework for building agents (Kit) and core intelligence model (Gemini).	The Gemini model provides the core reasoning and planning capabilities. The agents, built using the development kit, execute these plans by interacting with the platform's APIs.

4.2.2 Justification of Choices

The selection of each core technology was made after a fairly thorough evaluation of its features, alignment with the project's open-source principles, and suitability for an on-premises, hyper-converged environment. The following subsections provide a detailed rationale for each major component.

a. Hypervisor

To justify the selection of **Proxmox VE** as the foundational hypervisor, its key attributes are compared against prominent industry alternatives in Table 4.2. This analysis highlights the platform's unique suitability for the project's specific requirements.

Table 4.2: Comparative Analysis of Hypervisor Platforms

Feature / Aspect	**Proxmox VE**	**VMware vSphere**	**XCP-ng**	**Microsoft Hyper-V**
Licensing Model	Open-source (AGPLv3), with optional support subscriptions.	Proprietary, requires per-CPU licensing.	Open-source (GPL), with optional professional support.	Included with Windows Server; free standalone version.
Management	Integrated web-based UI for all functions.	Requires vCenter Server for most advanced features.	Requires Xen Orchestra (web UI) for full functionality.	Requires Hyper-V Manager or Windows Admin Center.
Key Features	HA, backup, Ceph/ZFS, and SDN are all built-in.	Industry-leading vMotion, DRS, and HA features.	Robust live migration and HA capabilities.	Strong integration with the Windows ecosystem.
Ecosystem	Open, no vendor lock-in, strong community support.	Extensive, mature ecosystem but with strong vendor lock-in.	Open, community-driven, focused on open-source.	Tightly integrated with Microsoft Azure and services.
Project Rationale	Optimal for cost-effective, hyper-converged research projects due to its integrated, open-source feature set.	Standard for large enterprises with significant budgets.	A strong open-source alternative.	Ideal for predominantly Windows-based environments.

Ultimately, the analysis confirms the selection of **Proxmox VE**. Its combination of a cost-effective, open-source model with a rich, built-in feature set directly aligns with the project's requirements for a hyper-converged environment.

b. Container Orchestration

The selection of **Kubernetes** (via the **k3s** distribution) is contextualized by the comparative analysis in Table 4.3. This comparison justifies the choice by weighing its capabilities against other leading orchestration platforms.

Table 4.3: Comparative Analysis of Container Orchestration Platforms

Feature / Aspect	**Kubernetes (k3s)**	**Docker Swarm**	**OpenShift**	**Nomad**
Complexity	High, but k3s variant significantly reduces overhead.	Low, very easy to learn and deploy.	Very High; adds security and CI/CD layers to Kubernetes.	Low to Medium; focuses on simplicity.
Ecosystem	Vast and dominant; the widely used industry standard.	Limited; primarily tied to the Docker ecosystem.	Curated, enterprise-focused Red Hat ecosystem.	Growing; integrates seamlessly with HashiCorp tools.
Flexibility	Extremely flexible and extensible via its API and CRDs.	Opinionated and less flexible.	Opinionated for security and developer workflows.	Very flexible; can orchestrate non-containerized apps.
Core Strength	Unmatched community, extensibility, and tool support.	Simplicity and speed for basic use cases.	Enterprise-grade security and developer tooling.	Simplicity, flexibility, and multi-workload support.
Project Rationale	Essential for accessing the widest range of cloud-native tools (Helm, Prometheus, MetalLB) needed for this research.	Suitable for simple applications without complex needs.	A full PaaS, overkill for this project's scope.	A strong alternative, but with a smaller ecosystem.

This analysis validates the selection of **Kubernetes (k3s)** as the optimal choice. Its dominant ecosystem was a non-negotiable requirement for integrating the essential cloud-native tools needed to fulfill the project's objectives.

c. Hyper-Converged Storage and Operating System

The goal of creating a true **hyper-converged infrastructure (HCI)** drove the storage decision. **Ceph** was chosen because it allows storage to be distributed across the Proxmox nodes themselves, avoiding the single point of failure and performance bottleneck of a separate NAS device. Its tight integration with Proxmox and its ability to provide both resilient block storage (**RBD**) for VMs and shared file storage (**CephFS**) for backups made it an exceptionally adaptable and efficient choice.
For the base operating system of the virtual machines, **Ubuntu Server LTS** was selected. While not a core architectural component, it is a critical implementation choice. Its selection was based on its massive community support, long-term stability, and, most importantly, its first-class support for **cloud-init**. This feature was indispensable for the Terraform automation workflow, allowing for the seamless, programmatic configuration of new VMs at first boot.

d. Automation and Configuration Tools

A core principle of this project was end-to-end automation. A powerful two-tool approach was adopted, leveraging **Terraform** and **Ansible** for their complementary strengths.
Terraform was chosen for infrastructure provisioning. As a declarative **IaC tool**, Terraform excels at managing the lifecycle of resources with a defined state, such as virtual machines. It allows the entire VM infrastructure to be defined in code, ensuring that `terraform apply` will always converge the system to the desired state. Its provider ecosystem, including a well-supported Proxmox provider, was a critical factor in this choice.
Ansible was selected for configuration management. Unlike Terraform, Ansible is a procedural automation tool, making it ideal for executing ordered tasks like installing software, configuring system files, and orchestrating complex deployment workflows. Its agentless architecture, which communicates over standard SSH, simplifies setup and reduces the attack surface compared to agent-based alternatives like **Puppet** or **Chef**. For this project, Ansible was indispensable for transforming the blank VMs provisioned by Terraform into a fully configured, high-availability k3s cluster.

e. Networking and Application Services

A flexible and software-defined approach to networking was essential. **OPNsense** was deployed as a virtual appliance to serve as the platform's central router, firewall, and DHCP/DNS server. Deploying it as a VM provided greater flexibility than a physical appliance and offered a more feature-rich and actively developed alternative to other open-source firewalls like **pfSense**.
Within the Kubernetes cluster, two critical networking gaps needed to be filled for an on-premises deployment:

1. **Control-Plane HA: Kube-vip** was used to provide a stable virtual IP (VIP) for the **Kubernetes API server**. It uses ARP to advertise the VIP on the network and ensures it is always held by a healthy master node, which is essential for maintaining control plane access during a node failure.

2. **Application Service Exposure: MetalLB** was implemented to provide the **LoadBalancer service type**, which is not natively available in on-premises clusters. MetalLB listens for Service objects of this type and assigns them an IP address from a pre-configured pool, making containerized applications accessible from the external network.

f. Application Management and Observability

To manage the lifecycle of applications on Kubernetes, **Helm** was chosen. As the standard package manager for Kubernetes, Helm allows complex, multi-component applications to be packaged into reusable charts, simplifying deployment, upgrades, and configuration. Its templating engine is far more powerful than using raw YAML files for managing complex application releases.
For observability, the industry-standard stack of **Prometheus** and **Grafana** was selected. Prometheus's pull-based model and powerful service discovery capabilities integrate seamlessly with Kubernetes, allowing it to automatically find and scrape metrics from new pods and services. Grafana provides a powerful and flexible interface for visualizing this data. This combination was chosen over older monitoring tools like **Nagios** or **Zabbix** because it is purpose-built for the dynamic, ephemeral nature of cloud-native environments.

j. The Agentic AI Layer

The Agentic AI layer, the capstone of the architecture, was implemented using Google's ecosystem. This choice was driven by Google's strong commitment to both the AI and

open-source communities, which provides significant flexibility and is backed by extensive support. The layer leverages the **Google Agent Development Kit (ADK)** to build the agents and the **Google Gemini model** as the core intelligence. This combination allows the system to translate natural language commands into concrete actions by securely calling upon the platform's underlying tools.
Ultimately, this technology stack forms a **cohesive, open-source, and highly automated platform**. Each component was deliberately chosen for its stability and suitability for an `on-premises, hyper-converged environment`, ensuring the final system directly fulfills the research objectives of **resilience, scalability, and intelligent provisioning**.

4.3 Foundational Layer: Physical Infrastructure and Virtualization

The foundation of the entire autonomic cloud environment rests upon a robust and well-configured physical and virtualization layer. This section details the hardware selection, network configuration, and the installation of the **Proxmox Virtual Environment (VE)** hypervisor, which serves as the bedrock for all subsequent layers.

4.3.1 Hardware and Network Configuration

The foundation of any stable cluster, particularly one designed for high availability and predictable performance, is **hardware homogeneity**. By utilizing identical server components across all nodes, we ensure consistent performance, simplified management, and reliable failover operations for features like live migration. The physical infrastructure for this research consists of three identical servers interconnected via a physical network switch.
The specifications for each server are detailed in Table 4.4, and their corresponding network configurations are outlined in Table 4.5.

Table 4.4: Physical Server Specifications

Component	**Specification**
CPU	12th Gen Intel Core i7-12700 (1 Socket, 12 Cores, 20 Threads)
Memory (RAM)	32 GB
Storage	1 TB NVMe SSD
Network Interface	1x 1GbE Network Interface Card (NIC)
Connectivity	Connected via a shared physical 1GbE Switch

Table 4.5: Node Network Configuration

Hostname (FQDN)	IP Address	Subnet Mask	Gateway	DNS Server
pmox01.lab.local	172.25.5.201	255.255.255.0	172.25.5.1	8.8.8.8
pmox02.lab.local	172.25.5.202	255.255.255.0	172.25.5.1	8.8.8.8
pmox03.lab.local	172.25.5.203	255.255.255.0	172.25.5.1	8.8.8.8

With the physical and network foundations established, the next step was the installation of the Proxmox VE hypervisor on each node.

4.3.2 Proxmox VE Installation

The installation process for **Proxmox VE (version 8.3)** was initiated on each of the three servers. This began by creating a bootable USB drive using the official Proxmox VE ISO image and the **Rufus** utility. Upon booting from the installation medium, the *Install Proxmox VE (Graphical)* option was selected from the main boot menu.
The installation wizard proceeded through the End User License Agreement (EULA). A critical step in this process was the strategic partitioning of the storage. By accessing the *Options* menu on the '**Target Harddisk**' screen (Figure 4.1), the `hdsize` parameter was explicitly set to 100 GB. This action allocated a 100 GB partition for the Proxmox operating system and its default local storage, intentionally leaving the remaining ~850 GB of disk space unallocated. This reserved space is essential for the subsequent creation of the **Ceph distributed storage cluster**, which is detailed in Section 4.3.5.

Figure 4.1: Proxmox Harddisk Options for Partition Sizing

Subsequently, the Management Network Configuration screen was used to assign the static IP address, gateway, DNS server, and **Fully Qualified Domain Name (FQDN)** for each node, as specified in Table 4.5. All settings were confirmed on the Summary screen, and the installation was initiated. For the purpose of replicability within this academic context, the administrator password for all nodes was set to `adminproxmox`. Following a successful installation, the system automatically rebooted. The console then displayed the URL for accessing the web interface via HTTPS on port 8006. Accessing this URL and logging in with the root user and the pre-configured password confirmed the successful installation of the node. The Proxmox VE web dashboard (Figure 4.2) displayed the single, operational node, now ready for the next phase of cluster configuration.

Figure 4.2: Example of Proxmox Web Interface Dashboard Post-Installation

4.3.3 Network Configuration with Open vSwitch

To enable advanced **Software-Defined Networking (SDN)** capabilities within Proxmox VE, the default Linux bridge on each node was replaced with **Open vSwitch (OVS)**. This critical transition was automated for consistency and repeatability across the cluster using an **Ansible playbook**.

The playbook orchestrated the network reconfiguration by first ensuring the `openvswitch-switch` package was installed, then deploying a new `/etc/network/interfaces` file. This new configuration establishes a new network topology by creating an OVS bridge named `vmbr0`. The physical network interface (`enp1s0`) is attached to this bridge as a port, and the node's static IP address is assigned directly to the bridge itself. The resulting logical network structure is illustrated in Figure 4.3.

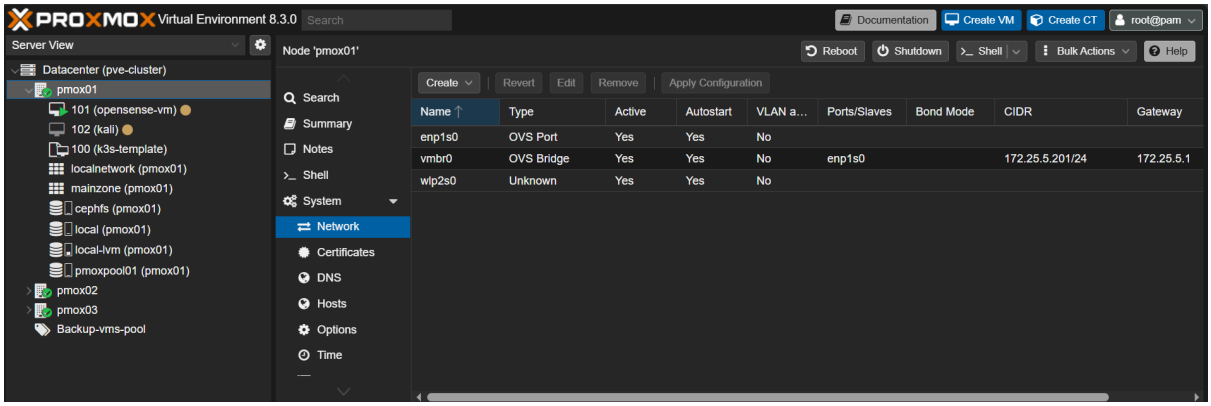

Figure 4.3: Network Configuration Overview with OVS Bridge

To manage the unique IP addresses for each node (172.25.5.201, 172.25.5.202, and

172.25.5.203), the Ansible automation utilized a templating mechanism. This ensures that while the network topology is identical across the cluster, each node retains its correct management IP. With OVS now configured on all nodes, the system is prepared for the creation of the Proxmox cluster and the implementation of advanced SDN zones.

4.3.4 Proxmox VE Hypervisor Cluster Configuration

With the individual Proxmox VE nodes installed, they were unified into a single, centrally managed entity. This is a critical step for enabling **high-availability** and leveraging distributed storage. The process began by creating a new cluster named `pve-cluster` on the primary node, `pmox01`. This action generated secure join credentials that were subsequently used on the `pmox02` and `pmox03` nodes to integrate them into the cluster. The successful formation of the cluster results in a cohesive, three-node environment managed from a single web interface. As shown in Figure 4.4, the datacenter summary view now confirms that all three nodes are online and part of a quorate cluster, providing a consolidated overview of all resources.

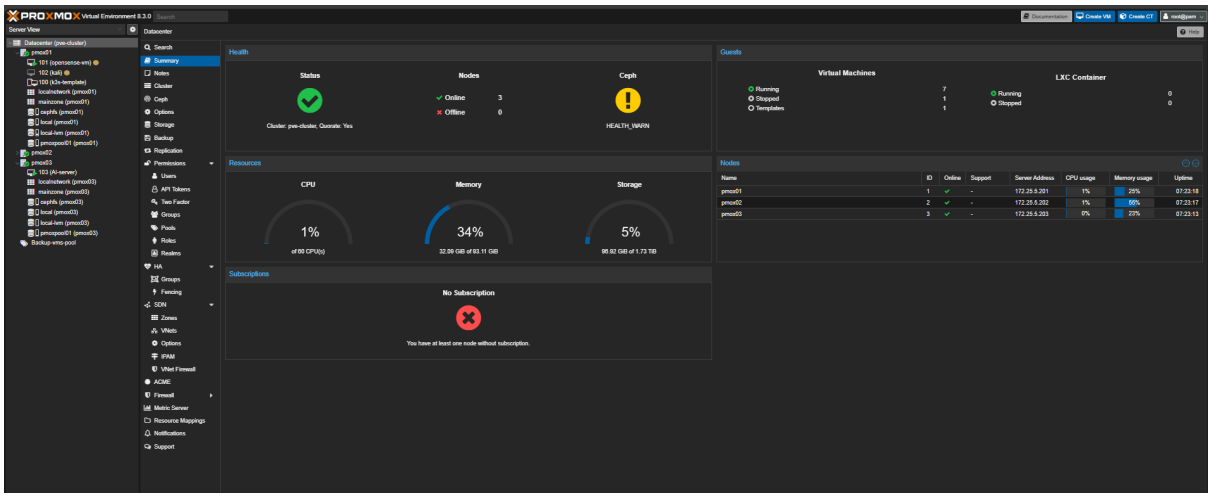

Figure 4.4: Proxmox Datacenter Summary after Cluster Formation

This unified structure is the fundamental prerequisite for deploying the resilient Ceph storage system in the next stage.

4.3.5 Software-Defined Storage with Ceph

With the Proxmox cluster established, the next critical step was to implement a hyper-converged storage layer using **Ceph**. This provides a resilient, scalable, and unified storage platform. The process began with preparing the physical disks on each node, followed by the installation and configuration of the Ceph cluster itself.

As detailed in the Proxmox installation (Section 4.3.2), the `hdsize` parameter was used to reserve the majority of the disk space on each node. Before this space could be utilized by Ceph, it had to be formally partitioned. The detailed command-line procedure for this task is provided in **Annex A.1** .

- The result of this operation is a new, large partition (`/dev/sda4`) dedicated to Ceph, the presence of which was confirmed using the `lsblk` command, as shown in Listing 4.5.

```
sda           8:0    0 953.9G  0 disk
|-sda1        8:1    0  1007K  0 part
|-sda2        8:2    0    1G    0 part /boot/efi
|-sda3        8:3    0  99G   0 part
| |-pve-swap  252:0  0   8G    0 lvm  [SWAP]
| `-pve-root  252:1  0  34.7G  0 lvm  /
...
`-sda4        8:5    0 853.9G  0 part
```

Figure 4.5: Final Disk Layout with New Partition for Ceph

This manual partitioning step is a crucial prerequisite for creating **Ceph Object Storage Daemons (OSDs)** on the same physical disk as the Proxmox operating system.
With the disks prepared, the Ceph cluster was configured directly from the Proxmox web interface. The process involved establishing a three-node **Monitor (MON) quorum** for high availability, creating an **Object Storage Daemon (OSD)** on the prepared partition of each host, and then provisioning two distinct storage types: a **replicated block storage pool** (`pmoxpool01`) for resilient VM disks and a **CephFS file system** (`cephfs`) for shared storage. The detailed, step-by-step procedure for this configuration, including the creation of monitors, OSDs, and storage pools, is documented in **Annex A.3** .
The culmination of these steps is a unified, hyper-converged storage platform integrated directly within the Proxmox cluster. The overall health and final state of the cluster are monitored through the main Ceph dashboard, as shown in Figure 4.6. This summary view confirms a `HEALTH_OK` status, indicating that all components are online and the system is operating as expected, thus fulfilling the objective of a stable and resilient storage foundation.

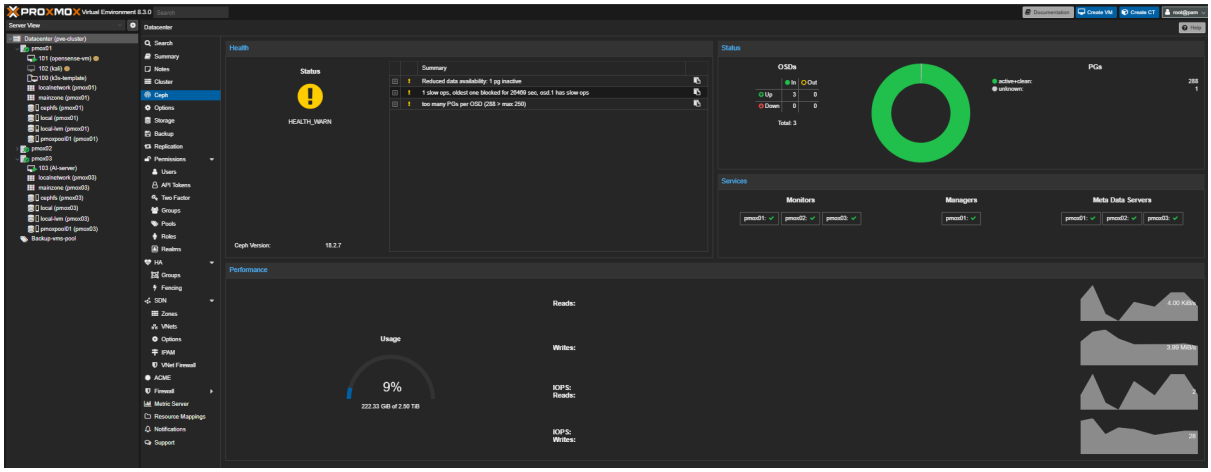

Figure 4.6: The Ceph cluster dashboard

4.3.6 Software-Defined Networking (SDN) and the OPNsense Virtual Appliance

With the storage foundation established, the focus now shifts to the virtual network fabric. This section details the configuration of **Software-Defined Networking (SDN)** and the **OPNsense** virtual appliance that provides routing and DHCP services.
While the physical network provides connectivity for the Proxmox hosts, it is insufficient for provisioning a large, isolated, and multi-tenant virtual environment. To address this, a hybrid approach was implemented, combining Proxmox VE's native **Software-Defined Networking (SDN)** capabilities with a dedicated virtual appliance for network services. This strategy leverages the strengths of both components: Proxmox SDN is used to define the virtual network fabric and enforce network segmentation at the hypervisor level, while an OPNsense virtual appliance provides essential Layer 3 services such as routing, DHCP, and DNS. This creates a flexible and powerful virtual network independent of the underlying physical infrastructure.

a. Defining the Virtual Network Fabric in Proxmox

The initial step was to configure the SDN components within the Proxmox Datacenter to create the logical network space.

1. **SDN Zone Creation:** A **VLAN zone** named `mainzone` was created, as shown in Figure 4.7. This zone utilizes the `vlan` plugin, which instructs Proxmox to tag traffic for this network, ensuring isolation from other networks.

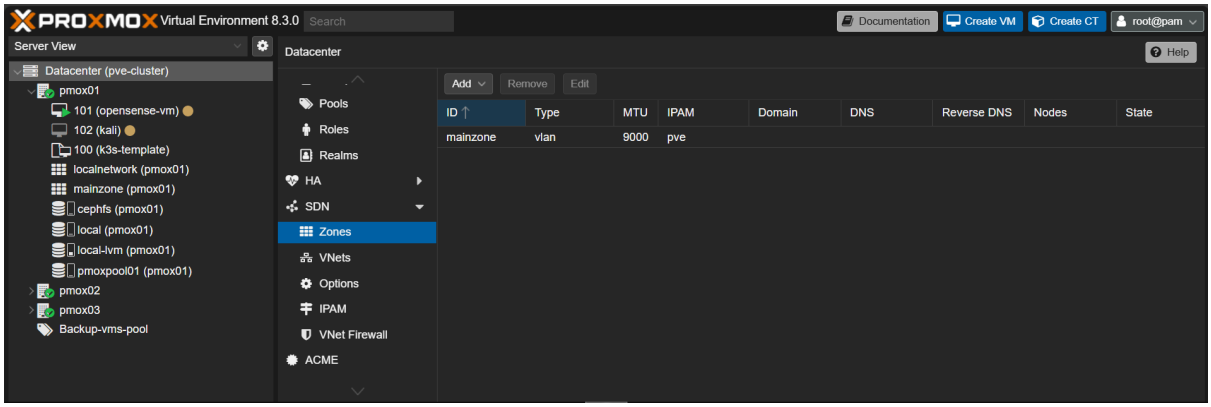

Figure 4.7: Configuration of the `mainzone` SDN Zone

2. **VNet and Subnet Creation:** A **Virtual Network (VNet)** named `mainvnet` was then created and associated with the `mainzone`. Within this VNet, a subnet was defined for the internal **Local Area Network (LAN)**: `192.168.0.0/16`. This provides a large, private address space for all subsequent virtual machines and Kubernetes pods. The gateway for this subnet was designated as `192.168.0.1`, which is the IP address that will be assigned to the OPNsense appliance's internal interface (Figure 4.8).

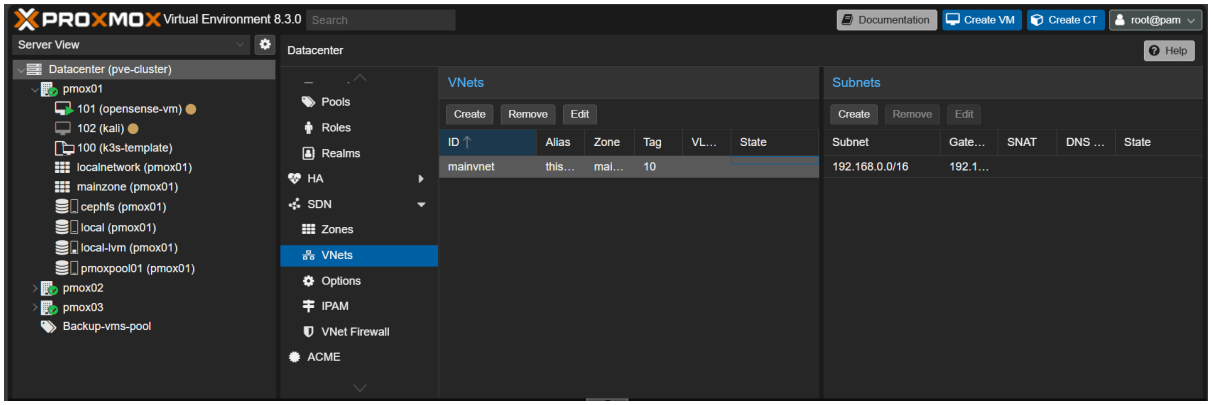

Figure 4.8: Definition of the `mainvnet` and its associated `192.168.0.0/16` subnet.

With these components configured, Proxmox is now aware of the virtual network, and this VNet can be attached to virtual machines.

b. OPNsense as a Virtual Router and Service Provider

With the virtual network fabric defined in Proxmox SDN, an OPNsense virtual machine (ID 101) was deployed to act as the central router, firewall, and service provider for the `mainvnet`. The OPNsense appliance is critical for bridging the isolated virtual network with the external world and providing essential network services.

The appliance was provisioned with 4 GB of RAM and 2 CPU cores. As depicted in the Proxmox hardware configuration (Figure 4.9), the VM was equipped with two distinct virtual network interfaces:

- **net0:** Connected to the `mainvnet`, designated to serve as the internal LAN interface.
- **net1:** Connected to the physical bridge `vmbr0`, designated to serve as the external WAN interface.

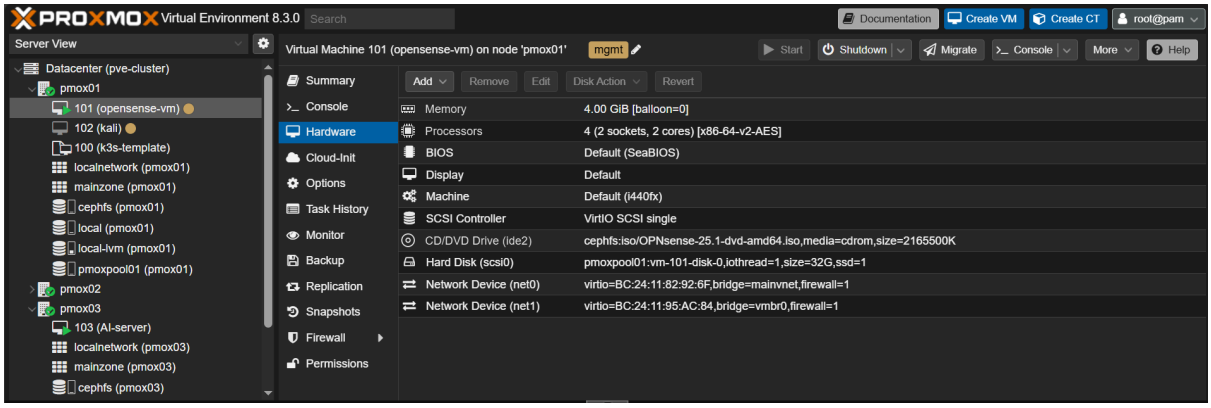

Figure 4.9: Proxmox hardware configuration for the OPNsense VM

This dual-interface configuration establishes a clear, multi-layered network traffic flow, as illustrated in Figure 4.10. All traffic originating from virtual machines on the internal LAN is routed through the OPNsense appliance, which performs **Network Address Translation (NAT)**. This allows all VMs on the private `192.168.0.0/16` network to share the single IP address of the OPNsense WAN interface for outbound internet access, while remaining isolated from the physical network.

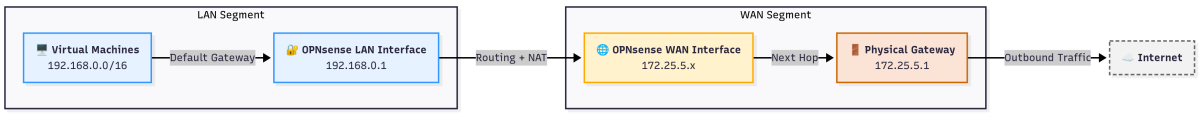

Figure 4.10: Logical network traffic flow

c. OPNsense Installation and Initial Configuration

To provide essential **Layer 3 network services** such as routing and DHCP for the virtualized environment, an **OPNsense virtual appliance** was deployed. The appliance was installed from the Proxmox console, with initial web GUI access cleverly established via a temporary VM on the same isolated mainvnet. The detailed installation are documented in **Annex B.1** .

For the purposes of this research, the configuration was focused on **core functionality**. Advanced features like dynamic routing protocols and VPNs were intentionally omitted to maintain simplicity, and the **firewall was disabled** to ensure **unrestricted traffic flow** within the trusted lab environment. The final configuration included assigning static IPs to the LAN and WAN interfaces, defining the default gateway, and enabling a DHCP server for the internal network, as shown in **Annex B.1** .
The successful outcome of this process is a **fully operational virtual router**, summarized by the main dashboard shown in Figure 4.11. This single view confirms the system's operational status, including the **active WAN gateway (172.25.5.1)**, resource utilization, and live traffic statistics, validating that the appliance is **correctly routing traffic** between the internal LAN and external WAN interfaces.

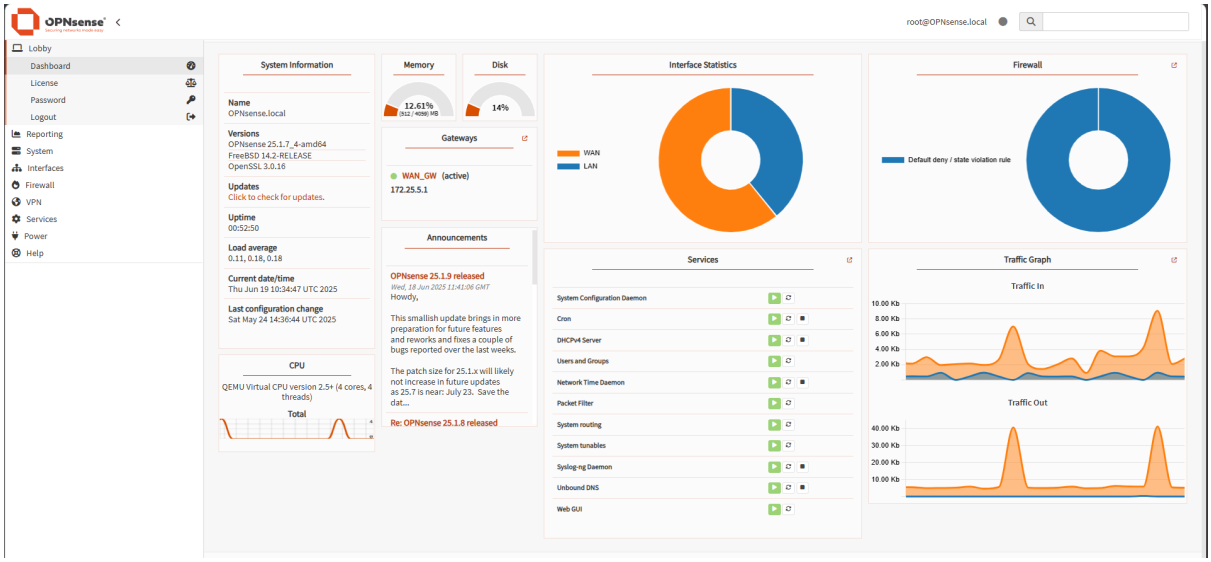

Figure 4.11: The OPNsense dashboard showing the final operational state

This established a **robust virtual network**, where OPNsense provides critical **routing and DHCP**, preparing the environment for all subsequent virtual machine and container workloads.

4.3.7 Configuring High Availability and System Resilience

With the foundational compute, storage, and network layers established, the focus shifts to ensuring operational resilience against hardware failure and data loss. This was achieved by implementing a dual strategy within the Proxmox VE cluster: real-time service failover using the **High Availability (HA)** manager and robust data protection through automated, scheduled backups.

a. High Availability for Service Continuity

To protect critical services from unplanned host downtime, the **Proxmox HA** feature was configured. This system automatically detects a failed node and restarts its designated virtual machines on other available nodes in the cluster. The implementation involved two key steps:

1. **Creating an HA Group:** An HA group named `mainHA` was created to define the set of nodes responsible for running highly available services. As shown in Figure 4.12, this group was configured to include all three nodes in the cluster (`pmox01`, `pmox02`, `pmox03`), ensuring maximum failover capacity. The `nofailback` option was enabled, meaning VMs will not automatically return to their original host once it recovers. This helps to prevent unnecessary migration and potential service disruption caused by moving stable workloads back to their original node.

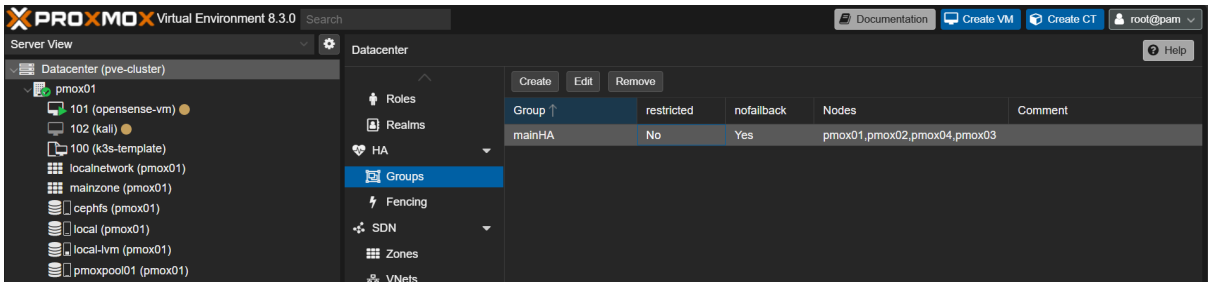

Figure 4.12: The `mainHA` Group configuration

2. **Assigning Resources:** The `mainHA` group serves as a policy container for all critical services. Initially, the OPNsense appliance was added to this group to ensure network resilience. As subsequent components like the AI-server and the Kubernetes nodes were provisioned (using the automated methods detailed in Section 4.4), they were also programmatically assigned to the `mainHA` group. Figure 4.13 illustrates the final, fully populated state of the HA resource panel, with all critical services under management. This configuration ensures that in the event of a host failure, the system will automatically attempt to restart any managed VM on a surviving node.

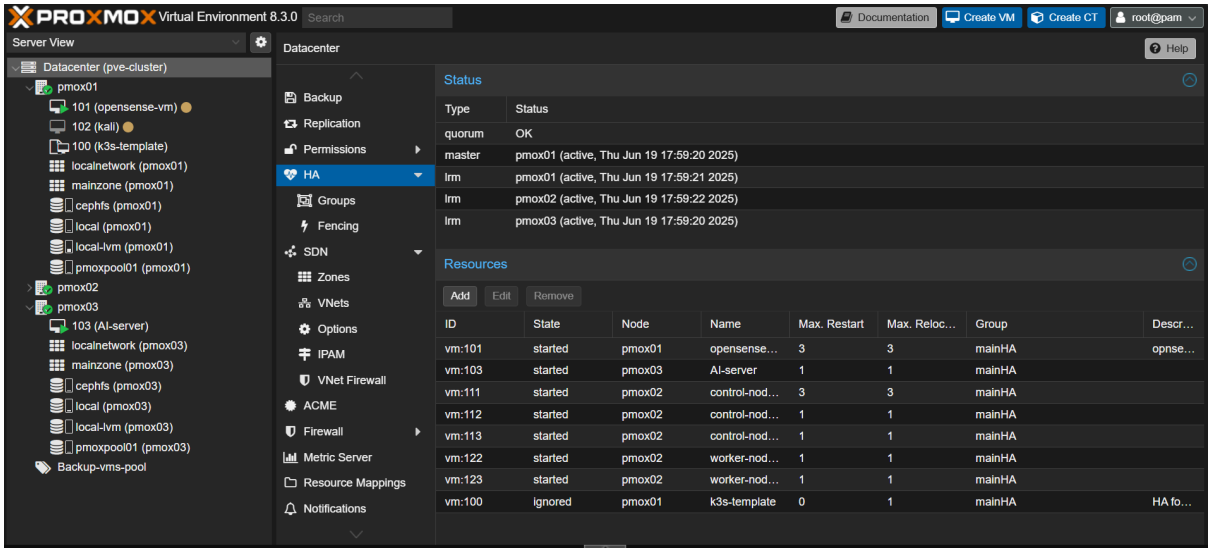

Figure 4.13: The HA Manager status screen

b. Automated Backups for Data Protection

While High Availability protects against immediate service loss, automated backups are essential for disaster recovery and protection against data corruption. The foundation of the backup strategy is a resource pool named `Backup-vms-pool`, created specifically to group the most critical virtual machines, as shown in Figure 4.14.

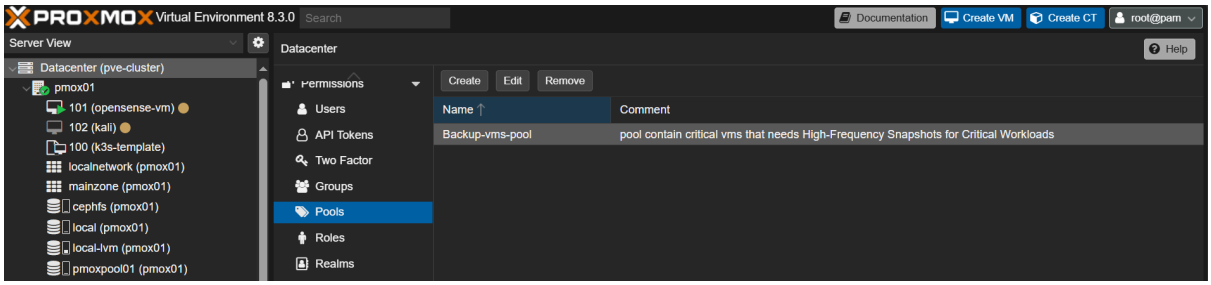

Figure 4.14: Creation of the `Backup-vms-pool`

This pool is populated automatically during the infrastructure provisioning phase, as the Kubernetes and AI-server VMs are created by Terraform (detaied in Section 4.4), they are programmatically added as members. The final, populated state of the pool is shown for reference in Figure 4.15.

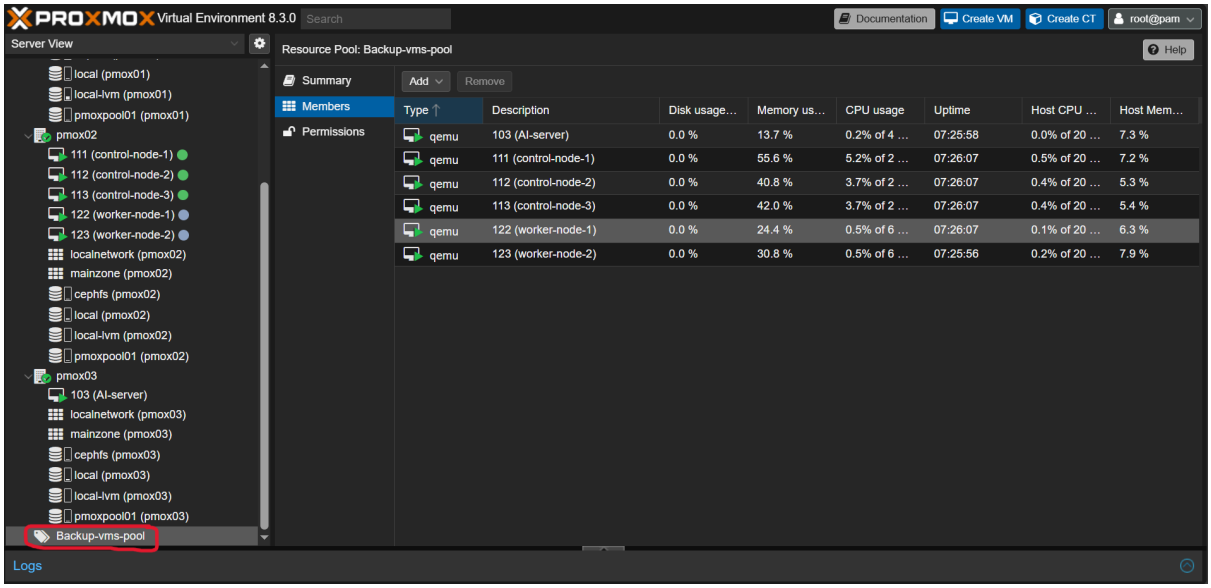

Figure 4.15: The fully populated `Backup-vms-pool`

With the `Backup-vms-pool` established as a logical target, a two-tiered backup strategy was implemented to address different recovery needs, leveraging the resilient cephfs storage backend for all backup data:

1. **High-Frequency Snapshots for Critical Workloads:** To ensure a low **Recovery Point Objective (RPO)** for the most dynamic services, a backup job was created to target the `Backup-vms-pool`. This job runs every 30 minutes using the *Snapshot* mode, creating live, point-in-time backups with minimal performance impact.

2. **Daily Full Backups for Disaster Recovery:** To guarantee absolute data consistency for the entire platform, a second, more comprehensive backup job was configured. This job targets all virtual machines, runs daily during an off-peak window (21:00), and uses the *Stop* mode, which ensures a perfectly consistent state ideal for full disaster recovery.

The implementation of this complete two-tiered strategy is summarized in Figure 4.16, which shows both configured backup jobs in the Proxmox Datacenter view.

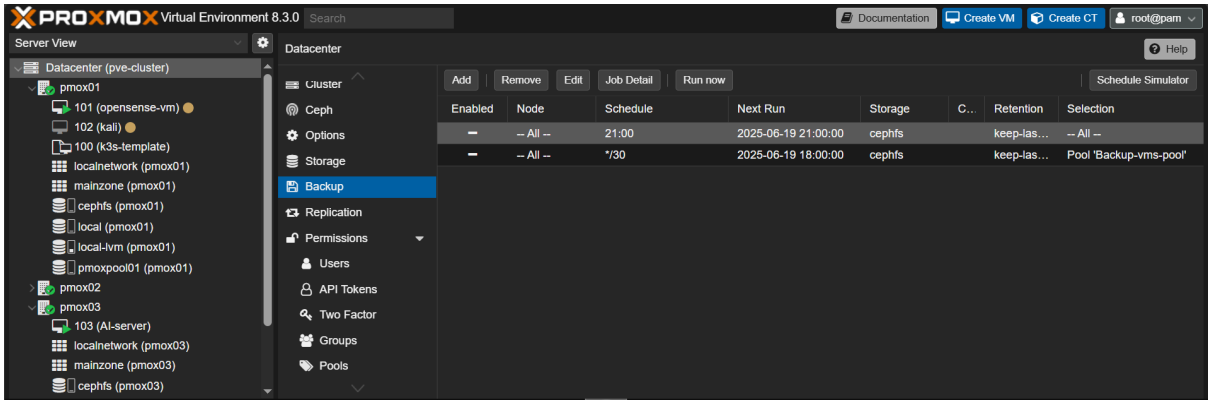

Figure 4.16: The Proxmox backup schedule

This multi-layered backup strategy, combined with the High Availability configuration, creates a robust resilience framework that protects the system against both immediate hardware failures and long-term data loss.

4.4 Container Orchestration Layer: High-Availability Kubernetes with k3s

With the underlying hyper-converged infrastructure fully operational, the next layer of the architecture the container orchestration platform was deployed. This research utilizes **k3s**, a lightweight yet fully compliant Kubernetes distribution, to form a high-availability cluster. The entire lifecycle of the virtual machines hosting this cluster is managed using an **Infrastructure as Code (IaC)** approach with Terraform.

4.4.1 Preparing the VM Template

A prerequisite for automated provisioning is a standardized base image, or template. A virtual machine template named `k3s-template` (ID 100) was created for this purpose. The template was prepared by installing a minimal **Ubuntu Server OS** and configuring it to support **cloud-init**. This allows for the automated customization of new VMs at first boot, including setting hostnames, network configurations, and user credentials. The specific script used to create and configure this template is detailed in **Annex A.2** .

4.4.2 Automated Provisioning of Kubernetes Nodes with Terraform

To ensure a repeatable and consistent deployment process for the Kubernetes nodes, **Terraform** was employed. A modular Terraform configuration was developed to programmatically define and provision the required virtual machines by cloning the `k3s-template`. This **Infrastructure as Code (IaC)** approach enables the infrastructure to be **version-controlled**, **scaled easily** by modifying variables, and managed through **automated workflows**.

The Terraform configuration consists of several files:

- `providers.tf`: Defines the required Proxmox provider.
- `variables.tf`: Declares all configurable parameters, such as VM counts, names, and resource allocations.
- `main.tf`: Contains the core logic that defines the `proxmox_vm_qemu` resource to be created.
- `credentials.auto.tfvars`: Supplies the specific values for the variables, defining the desired state of the infrastructure.

The `credentials.auto.tfvars` file was configured to deploy a five-node cluster (three control-plane, two worker nodes) with specific resource allocations, network settings, and resilience policies as previously described.
With all configuration files placed in a single directory on a management workstation, the deployment was executed with two commands:

1. `terraform init`: To initialize the working directory and download the Proxmox provider plugin.
2. `terraform apply`: To analyze the configuration, present an execution plan, and, upon confirmation, create the resources in Proxmox.

The successful execution of this process resulted in the creation of five new virtual machines, fully configured and ready for the next stage. Figure 4.17 shows the hardware configuration of a provisioned worker node (`worker-node-2`), confirming the specified 6 cores, 8 GB of RAM, and its connection to the `mainvnet`.

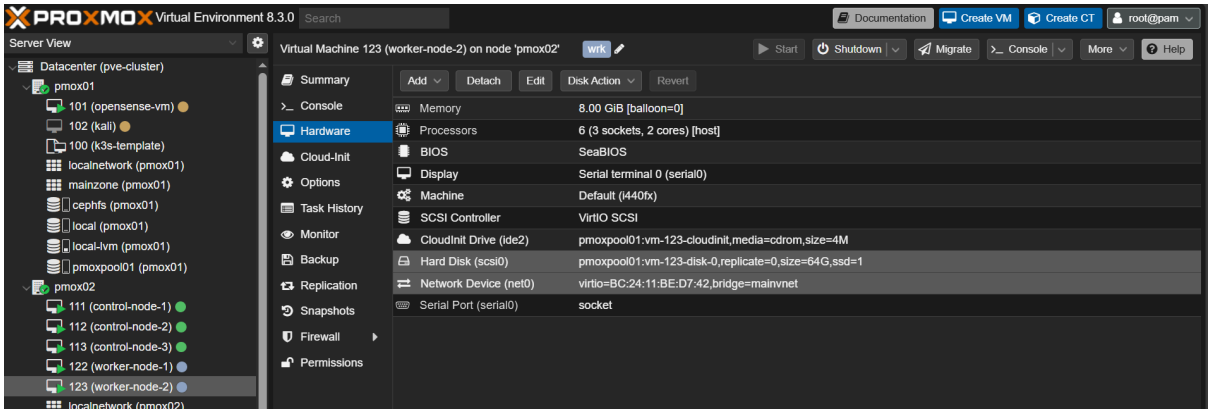

Figure 4.17: The hardware configuration of the provisioned worker-node-2 VM

While Figure 4.18 illustrates the cloud-init configuration applied to a VM, showing the successful injection of the static IP address, DNS servers, and user credentials defined in the Terraform variables.

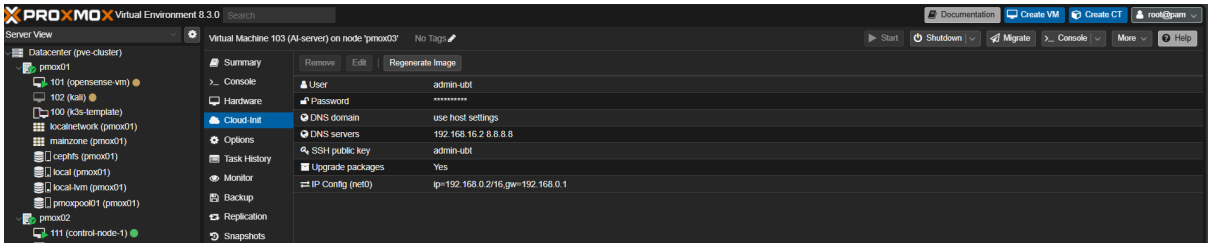

Figure 4.18: The Cloud-Init configuration applied to a provisioned VM

4.4.3 Configuration and Deployment of the HA k3s Cluster with Ansible

Following the automated provisioning of the virtual machines with Terraform, the next phase focused on the configuration management task of installing and setting up a high-availability k3s cluster. To ensure a reliable, repeatable, and best-practice-compliant deployment, this research leveraged the popular open-source project `techno-tim/k3s-ansible`. This Ansible playbook automates the entire process, from preparing the nodes to bootstrapping a multi-master cluster and integrating essential components.

a. Customizing the Ansible Playbook

Before execution, the cloned project was tailored to the specific parameters of this research environment. This involved renaming the configuration file `ansible.example.cfg` to `ansible.cfg` and preparing the inventory structure by renaming the `inventory/sample`

directory to `inventory/my-cluster`.
The core customizations were then made in the inventory and variable files:

1. **Host Inventory (`hosts.ini`):** The inventory file was populated with the IP addresses of the five provisioned VMs. The three control-plane nodes were placed in the `[master]` group, and the two worker nodes were placed in the `[node]` group, as shown in Figure 4.19.

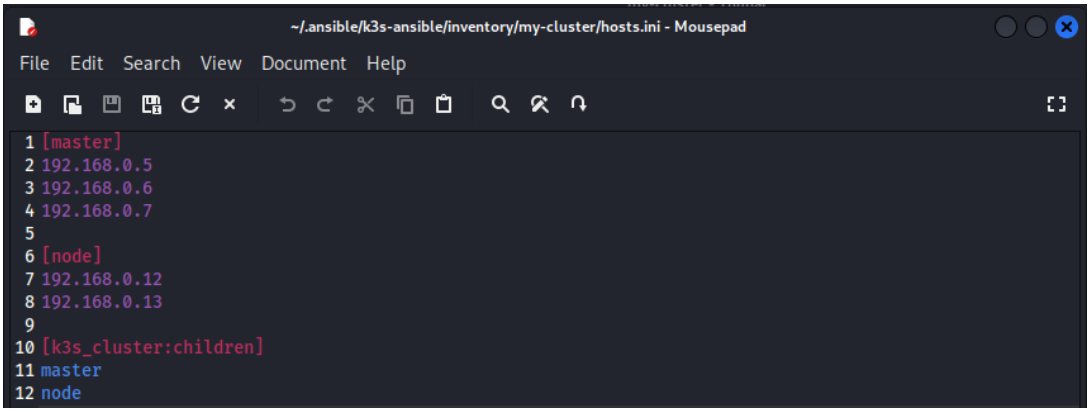

Figure 4.19: The `hosts.ini` file

2. **Connection and Privilege Escalation (`k3s_cluster.yaml`):** A file was created at `inventory/my-cluster/group_vars/k3s_cluster.yaml` to define how Ansible connects to the target nodes. As seen in Figure 4.20, this file specifies the `ansible_user` (`admin-ubt`), the path to the SSH private key for authentication, and the parameters for privilege escalation (`become: true`, `become_method: sudo`).

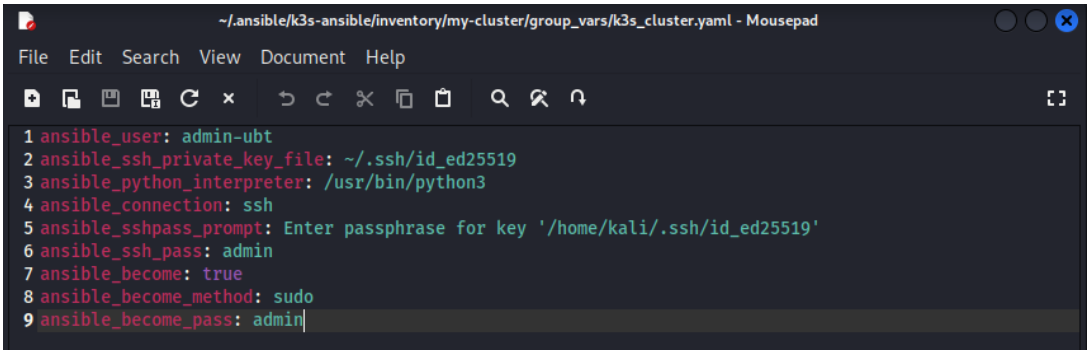

Figure 4.20: The `k3s_cluster.yaml` file

3. **Cluster-Specific Configuration (`all.yaml`):** Key variables within the `group_vars/all/all.yml` file were modified to define the cluster's architecture:

   - `ansible_user`: This was set to `admin-ubt` to match the user created by cloud-init.

- `apiserver_endpoint`: This was set to `192.168.0.9`. The playbook uses this variable to configure **Kube-vip**, which provides a stable virtual IP (VIP) for the high-availability control plane. This ensures that the Kubernetes API server remains accessible even if one of the master nodes fails.
- `metal_lb_ip_range`: This was configured to `192.168.0.246-192.168.0.254` to set up on-premises service exposure.

b. Deployment and Verification

With the configuration tailored to the environment, the deployment was initiated by running the primary Ansible playbook from the management workstation: `ansible-playbook site.yml`.
The playbook executed a series of roles that prepared the nodes, installed the k3s binaries, configured the first master, joined the other masters to form an etcd cluster, and finally joined the worker nodes.
Post-deployment, connecting to `control-node-1` and executing `kubectl get nodes` confirmed the successful creation of the cluster. The output in Figure 4.21 clearly shows a five-node cluster with three control-plane, etcd, master nodes and two worker nodes, all in a `Ready` state and running a consistent version of k3s (`v1.30.2+k3s2`).

```
admin-ubt@control-node-1:~$ sudo kubectl get nodes
NAME             STATUS   ROLES                       AGE    VERSION
control-node-1   Ready    control-plane,etcd,master   3d2h   v1.30.2+k3s2
control-node-2   Ready    control-plane,etcd,master   3d2h   v1.30.2+k3s2
control-node-3   Ready    control-plane,etcd,master   3d2h   v1.30.2+k3s2
worker-node-1    Ready    <none>                      3d2h   v1.30.2+k3s2
worker-node-2    Ready    <none>                      3d2h   v1.30.2+k3s2
```

Figure 4.21: The output of `kubectl get nodes`

c. On-Premises Service Exposure with MetalLB

A critical feature automated by this Ansible playbook is the installation of **MetalLB** In on-premises environments like this one, Kubernetes does not have a native provider for **LoadBalancer services** MetalLB addresses this gap by providing a network load-balancer solution that integrates with standard network equipment By responding to ARP requests, it makes a private IP address from its configured pool reachable on the local network.
The `metal_lb_ip_range` variable configured earlier ensures that when a `LoadBalancer` service is created in Kubernetes, MetalLB will automatically assign it an available IP from the `192.168.0.246-192.168.0.254` range, making the service accessible from the `mainvnet`.

This automated, playbook-driven approach not only ensures a correct and resilient initial deployment but also provides a clear path for future scalability Adding new worker nodes, for instance, simply requires adding their IP addresses to the `hosts.ini` file and re-running the playbook. The successful establishment of this robust Kubernetes platform provides the foundation for deploying containerized applications. This allows applications to be installed and managed using standard cloud-native tools, such as Helm charts or declarative Kubernetes manifests.

4.5 Monitoring, Management, and Automation Layer

The successful deployment of the infrastructure and container orchestration platform is not the final step A robust system requires comprehensive tools for monitoring, management, and maintenance. This section details the operational layer of the architecture, which integrates the automation workflows, establishes system-wide observability, and implements a resilient data protection strategy.

4.5.1 The Integrated IaC and Automation Workflow

The implementation of this platform relied on a cohesive **Infrastructure as Code (IaC)** and automation workflow. **Terraform** was leveraged to handle the declarative provisioning of the foundational resources-the virtual machines. This ensured that the infrastructure's desired state was defined in code. Subsequently, **Ansible** was used for configuration management, programmatically installing and configuring the complex, high-availability k3s cluster on top of the provisioned VMs. This two-tool approach provides a powerful, repeatable, and version-controlled method for deploying the entire platform from scratch.

4.5.2 Cloud-Native Monitoring with Prometheus and Grafana

To achieve observability into the Kubernetes cluster, a cloud-native monitoring stack consisting of **Prometheus** and **Grafana** was deployed using a **Helm chart** to provide cluster observability. Figure 4.22 confirms the successful deployment of all necessary pods and services.

```
admin-ubt@control-node-2:~$ sudo kubectl get pods
NAME                                            READY   STATUS    RESTARTS        AGE
grafana-85b9748f55-g2hw2                        1/1     Running   8 (25m ago)     3d23h
prometheus-alertmanager-0                       1/1     Running   3 (25m ago)     3d23h
prometheus-kube-state-metrics-786488967b-n9mb4  1/1     Running   112 (24m ago)   3d23h
prometheus-prometheus-node-exporter-8kkl2       1/1     Running   3 (26m ago)     3d23h
prometheus-prometheus-node-exporter-9kxb5       1/1     Running   3 (25m ago)     3d23h
prometheus-prometheus-node-exporter-mkk78       1/1     Running   3 (25m ago)     3d23h
prometheus-prometheus-node-exporter-wnhq2       1/1     Running   3 (25m ago)     3d23h
prometheus-prometheus-node-exporter-zlqhm       1/1     Running   3 (24m ago)     3d23h
prometheus-prometheus-pushgateway-849bb86467-t7zst  1/1  Running   3 (25m ago)     3d23h
prometheus-server-6648f86775-k82r2              2/2     Running   6 (24m ago)     3d23h
admin-ubt@control-node-2:~$ sudo kubectl get services
NAME                                 TYPE           CLUSTER-IP      EXTERNAL-IP      PORT(S)          AGE
grafana                              LoadBalancer   10.43.143.51    192.168.0.247    80:30878/TCP     3d23h
kubernetes                           ClusterIP      10.43.0.1       <none>           443/TCP          4d18h
prometheus-alertmanager              ClusterIP      10.43.148.189   <none>           9093/TCP         3d23h
prometheus-alertmanager-headless     ClusterIP      None            <none>           9093/TCP         3d23h
prometheus-kube-state-metrics        ClusterIP      10.43.60.182    <none>           8080/TCP         3d23h
prometheus-prometheus-node-exporter  ClusterIP      10.43.153.243   <none>           9100/TCP         3d23h
prometheus-prometheus-pushgateway    ClusterIP      10.43.19.183    <none>           9091/TCP         3d23h
prometheus-server                    LoadBalancer   10.43.29.69     192.168.0.246    80:30222/TCP     3d23h
```

Figure 4.22: the Prometheus and Grafana monitoring stack

After an initial configuration to add the Prometheus data source and import a community dashboard, the system provided immediate insight into key cluster metrics (Figure 4.23).

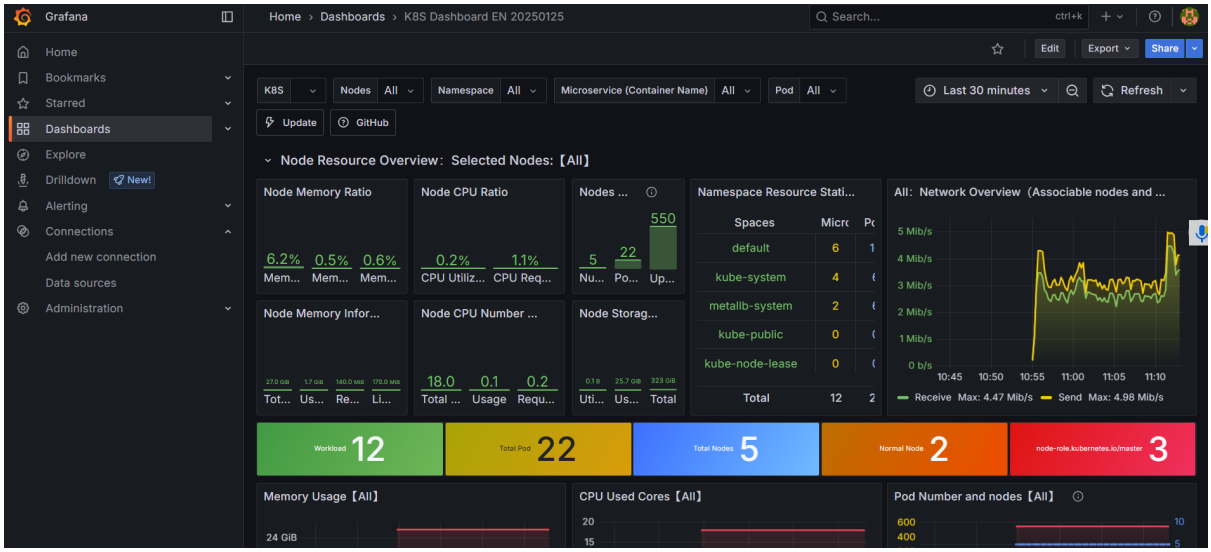

Figure 4.23: The Grafana dashboard for the `k3s_cluster`

4.5.3 Centralized Management Interfaces

Management of the implemented system is handled through two primary interfaces, each corresponding to a different layer of the architecture. The **Proxmox VE web interface** serves as the management plane for the physical and virtualization layer, providing control over VMs, storage, networking, HA, and backups. For the application layer, the **Kubernetes API**, accessed via the **kubectl** command-line tool, provides complete control over all containerized workloads, services, and cluster resources.

This operational layer unifies the platform through a cohesive automation workflow, comprehensive monitoring, and distinct management interfaces. The resulting system is robust, observable, and fully prepared for the deployment of the final Agentic AI layer, fulfilling the core objectives of this implementation phase.

4.6 The Agentic AI Layer

The final and most critical component of the architecture is the **Agentic AI layer**. This layer serves as the "brain" of the platform, transforming it from a system that can be automated into one that can understand and process natural language requests to perform **intelligent provisioning**. This section details the setup of the dedicated AI server and the development of the **Multi-Agent System (MAS)** that powers this capability.

4.6.1 AI Server Provisioning and LLM Selection

To host the agentic workload, a dedicated virtual machine named **AI-server** was provisioned using the same automated **Terraform workflow** established previously. This ensured its configuration was consistent with the rest of the platform's resources.
The initial research plan involved using a locally hosted **Large Language Model (LLM)** to ensure the system was fully self-contained. However, preliminary tests revealed that running a capable open-source model locally consumed a disproportionate amount of CPU and memory resources, outweighing the benefits for this project's scope. Consequently, the decision was made to pivot to a more resource-efficient API-based approach. The **Google Gemini API** was selected as the core intelligence engine. This approach offloaded the significant computational demand of the LLM while still providing access to state-of-the-art reasoning, planning, and code-generation capabilities. A secure API key was generated and configured on the **AI-server** to enable communication with the Gemini model.

4.6.2 Development of the Proxmox Provisioning Agent

With the AI server and LLM access established, the core agent was developed using the **Google Agent Development Kit (ADK)**. The ADK provided the necessary framework for building a sophisticated **Multi-Agent System (MAS)**, where different agents collaborate to fulfill a user's request.
Due to the complexity and time limitations of this research, the scope of the MAS was focused exclusively on intelligent **Proxmox provisioning**. The architecture of this system, depicted in Figure 4.24, consists of several specialized agents:

- **Manager**: The orchestrator that receives the initial user prompt and delegates tasks to other agents.
- **DataCollectionAgent**: Gathers real-time information from the environment by connecting to the Proxmox API.
- **AnalysesAndStrategyAgent**: Interprets the user's goal and the collected data to form a high-level plan.
- **ManifestRefinementLoop**: An iterative sub-system containing agents that generate, review, and refine the final Terraform configuration file until it meets the required standards.

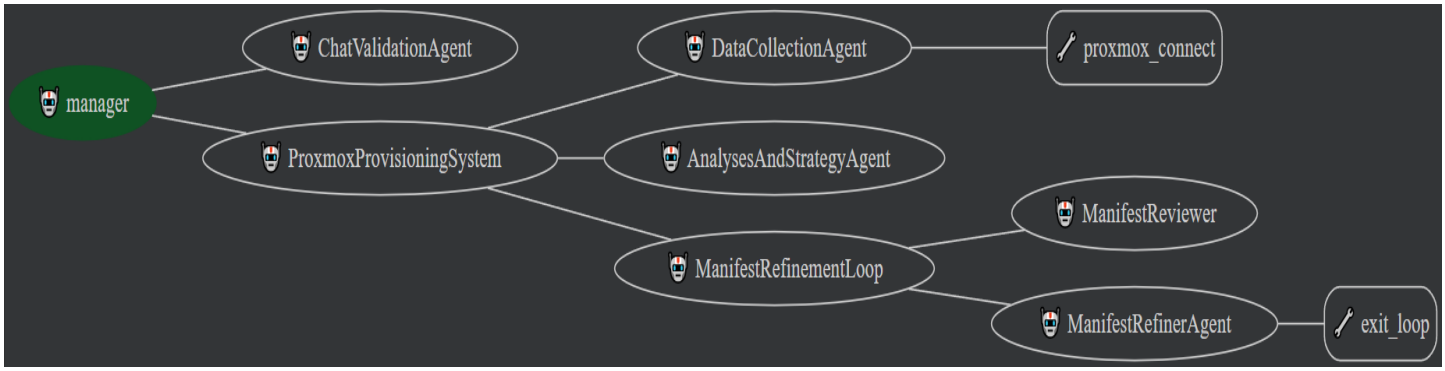

Figure 4.24: The architecture of the Smart Provisioning System.

A detailed breakdown of this collaborative workflow is provided in **Annex C.2** .

4.6.3 User Interaction and Workflow

The developed **Multi-Agent System** is exposed to the end-user through a clean, locally hosted web interface. Figure 4.25 captures a typical user interaction, showcasing the system's conversational and multi-agent nature as the `ValidationAgent` engages the user to clarify a provisioning request.

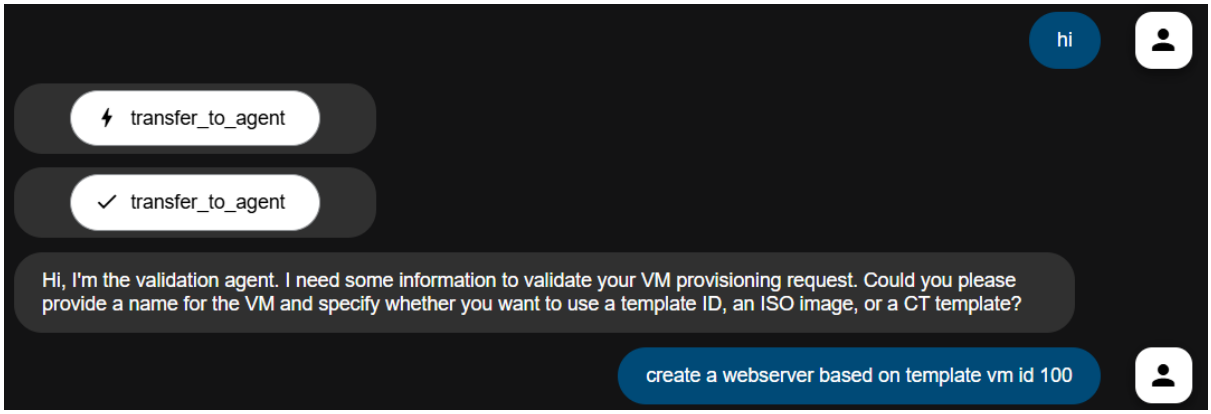

Figure 4.25: A sample interaction with the Agentic AI

Unlike a simple command-line tool, the system is designed to handle ambiguity through dialogue:

1. A user can initiate a conversation with a general or incomplete request.
2. The `Manager` agent intelligently routes the request to the appropriate specialized agent-in this case, the `ValidationAgent`.
3. The `ValidationAgent` then engages the user in a dialogue to gather the specific details required for provisioning, such as the `VM's` name and its source (`template`, `ISO`, etc.).
4. The user provides the clarified details (`"create a webserver based on template vm id 100"`), which the agent can then use to proceed.

This interactive validation step is crucial for ensuring the generated configuration is accurate before it is passed to the provisioning agents. Once all information is validated, the MAS workflow continues, ultimately generating a Terraform (**.tf**) file for the user's final review and execution. This "human-in-the-loop" approach ensures safety while leveraging the agent's intelligence to handle the complex task of writing the infrastructure configuration code, thus achieving the project's goal of **intelligent provisioning**.

4.7 Platform Validation: Integrated Scenarios

The previous sections have detailed the implementation of the individual layers of the platform. This section serves to demonstrate the synergy between these layers by presenting four practical scenarios. These tests validate the platform's end-to-end functionality, from its capacity for seamless scaling and resilience in the face of hardware failure to its efficiency in automated application deployment and intelligent provisioning.

4.7.1 Scenario 1: Automation-Driven Scalability

This scenario demonstrates the platform's ability to scale horizontally by adding a new physical host and a new Kubernetes worker node using the established IaC workflow.

1. **Scaling the Hypervisor Layer:** The process began with the initial three-node Proxmox cluster. A new physical server, `pmox04`, was installed with Proxmox VE and joined to the cluster, instantly increasing the cluster's total available compute and memory resources, as shown in Figure 4.26.

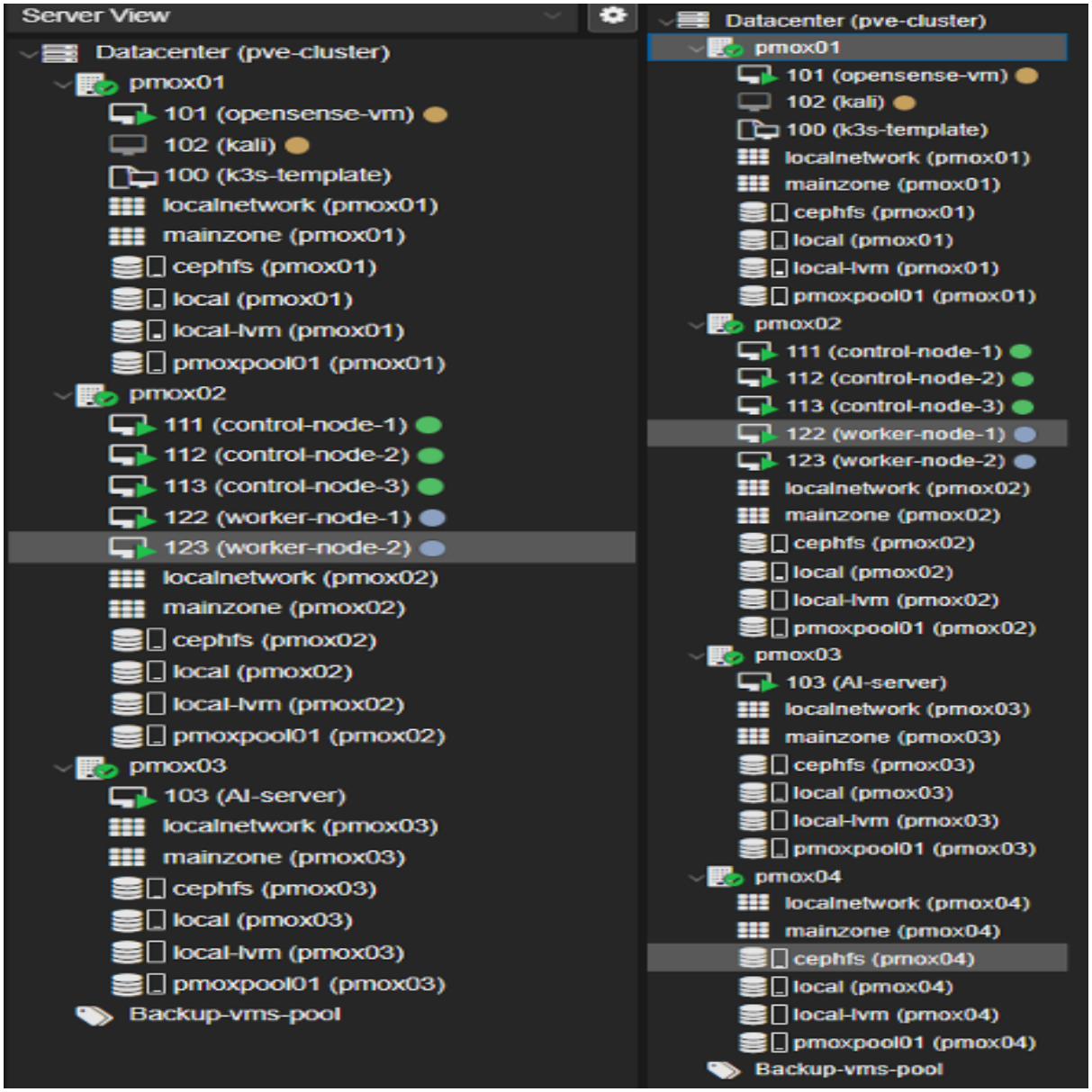

Figure 4.26: The Proxmox cluster server view before (left) and after (right) scaling from three to four physical nodes.

2. **Scaling the Kubernetes Layer:** To handle increased application load, a new worker node was provisioned and configured using the automated workflow. After

adding `worker-node-3` to the Terraform and Ansible configuration files, the respective `terraform apply` and `ansible-playbook site.yml` commands were executed.

3. **Verification:** The successful outcome is confirmed in Figure 4.27, where the `kubectl get nodes` command shows the new `worker-node-3` has joined the cluster and is in a `Ready` state, available to accept new workloads just 91 seconds after its creation.

```
admin-ubt@control-node-3:~$ sudo kubectl get nodes
NAME             STATUS   ROLES                       AGE    VERSION
control-node-1   Ready    control-plane,etcd,master   5d3h   v1.30.2+k3s2
control-node-2   Ready    control-plane,etcd,master   5d3h   v1.30.2+k3s2
control-node-3   Ready    control-plane,etcd,master   5d3h   v1.30.2+k3s2
worker-node-1    Ready    <none>                      5d3h   v1.30.2+k3s2
worker-node-2    Ready    <none>                      5d3h   v1.30.2+k3s2
worker-node-3    Ready    <none>                      91s    v1.30.2+k3s2
```

Figure 4.27: The Kubernetes cluster node status after the automated scaling operation

This scenario proves that scaling the platform is not a complex, manual project but a simple, declarative, and repeatable process, confirming the efficiency of the automated workflow.

4.7.2 Scenario 2: Multi-Layered Fault Tolerance

This scenario simulates hardware and software failures to test the multi-layered high-availability architecture.

1. **Hypervisor-Layer Failure (Proxmox HA):** The newly added host, `pmox04`, was abruptly powered off. The Proxmox HA manager detected the failure and, as shown in Figure 4.28, automatically migrated the `worker-node-3` VM (ID 124) to a surviving host (`pmox01`). This seamless migration is possible because the VM's disk resides on the shared Ceph storage, accessible from all cluster nodes.

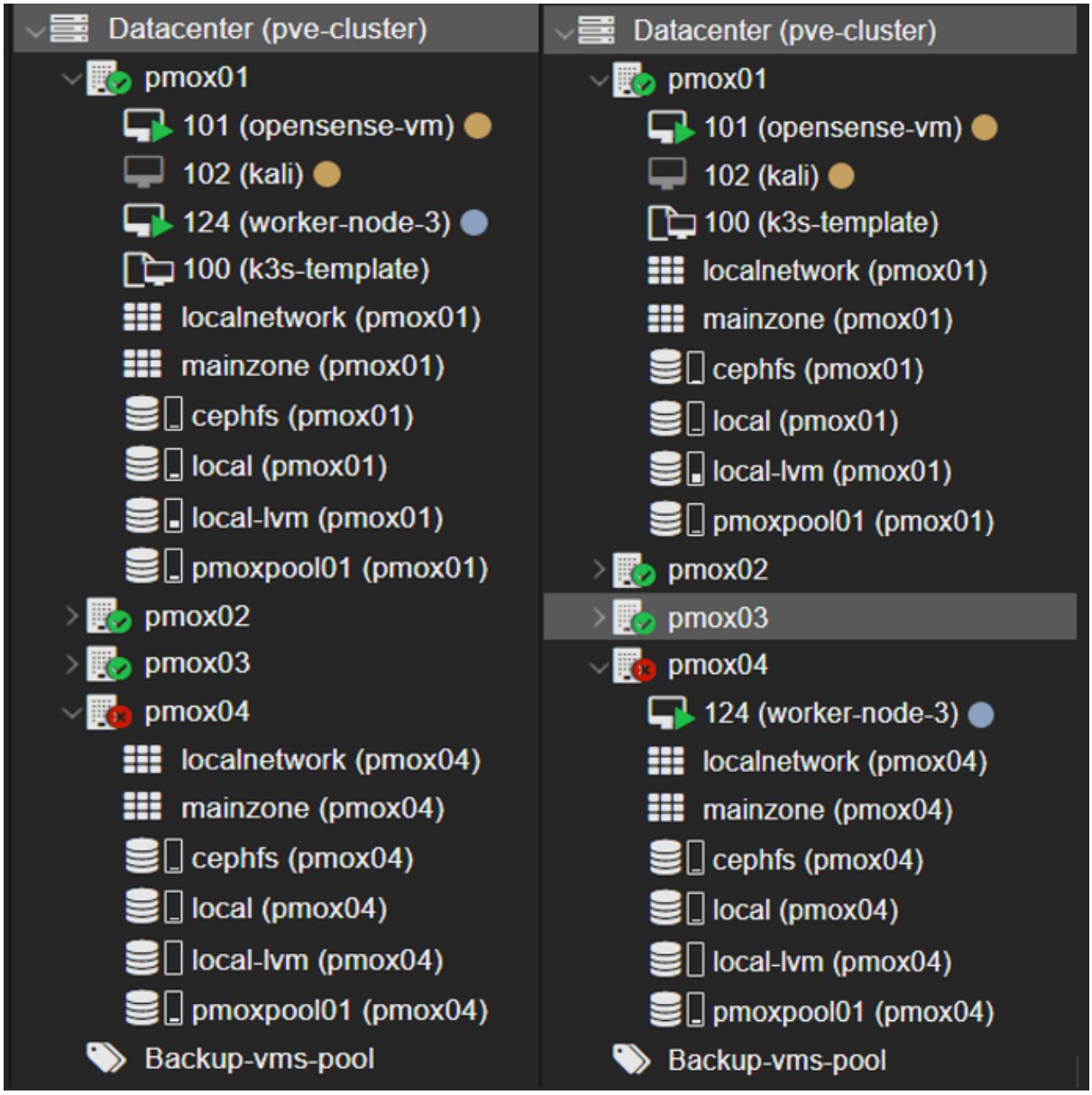

Figure 4.28: Proxmox HA failover in action. The `worker-node-3` VM is automatically migrated from the failed host `pmox04` (left) to the healthy host `pmox01` (right).

2. **Container-Layer Failure (k3s HA):** Next, a critical control-plane node, `control-node-1`, was shut down. The k3s cluster's resilience was immediately evident. Figure 4.29 shows the `kubectl get nodes` output, where `control-node-1` is correctly identified as `NotReady`. Despite this, the cluster remains fully operational because the **etcd cluster** maintains quorum and **Kube-vip** has automatically moved the control-plane's virtual IP to a healthy master.

```
admin-ubt@control-node-3:~$ sudo kubectl get nodes
NAME             STATUS     ROLES                       AGE    VERSION
control-node-1   NotReady   control-plane,etcd,master   5d3h   v1.30.2+k3s2
control-node-2   Ready      control-plane,etcd,master   5d3h   v1.30.2+k3s2
control-node-3   Ready      control-plane,etcd,master   5d3h   v1.30.2+k3s2
worker-node-1    Ready      <none>                      5d3h   v1.30.2+k3s2
worker-node-2    NotReady   <none>                      5d3h   v1.30.2+k3s2
worker-node-3    NotReady   <none>                      24m    v1.30.2+k3s2
```

Figure 4.29: Kubernetes cluster status during a control-plane failure

3. **Service Availability Verification:** The ultimate test of resilience is continued application availability. As confirmed in Figure 4.30, even with a master node down, all application pods for the Prometheus and Grafana monitoring stack remain in a `Running` state, and their `LoadBalancer` services are still active and exposing their external IPs. The platform successfully weathered the failure with no loss of application service.

```
admin-ubt@control-node-2:~$ sudo kubectl get pods
NAME                                             READY   STATUS    RESTARTS        AGE
grafana-85b9748f55-g2hw2                         1/1     Running   8 (25m ago)     3d23h
prometheus-alertmanager-0                        1/1     Running   3 (25m ago)     3d23h
prometheus-kube-state-metrics-786488967b-n9mb4   1/1     Running   112 (24m ago)   3d23h
prometheus-prometheus-node-exporter-8kkl2        1/1     Running   3 (26m ago)     3d23h
prometheus-prometheus-node-exporter-9kxb5        1/1     Running   3 (25m ago)     3d23h
prometheus-prometheus-node-exporter-mkk78        1/1     Running   3 (25m ago)     3d23h
prometheus-prometheus-node-exporter-wnhq2        1/1     Running   3 (25m ago)     3d23h
prometheus-prometheus-node-exporter-zlqhm        1/1     Running   3 (24m ago)     3d23h
prometheus-prometheus-pushgateway-849bb86467-t7zst   1/1     Running   3 (25m ago)     3d23h
prometheus-server-6648f86775-k82r2               2/2     Running   6 (24m ago)     3d23h
admin-ubt@control-node-2:~$ sudo kubectl get services
NAME                                 TYPE           CLUSTER-IP      EXTERNAL-IP     PORT(S)          AGE
grafana                              LoadBalancer   10.43.143.51    192.168.0.247   80:30878/TCP     3d23h
kubernetes                           ClusterIP      10.43.0.1       <none>          443/TCP          4d18h
prometheus-alertmanager              ClusterIP      10.43.148.189   <none>          9093/TCP         3d23h
prometheus-alertmanager-headless     ClusterIP      None            <none>          9093/TCP         3d23h
prometheus-kube-state-metrics        ClusterIP      10.43.60.182    <none>          8080/TCP         3d23h
prometheus-prometheus-node-exporter  ClusterIP      10.43.153.243   <none>          9100/TCP         3d23h
prometheus-prometheus-pushgateway    ClusterIP      10.43.19.183    <none>          9091/TCP         3d23h
prometheus-server                    LoadBalancer   10.43.29.69     192.168.0.246   80:30222/TCP     3d23h
```

Figure 4.30: Application pods and services remain fully operational during the control-plane node failure.

This test validates the critical concept of multi-layered resilience. Both the virtualization and container orchestration layers responded autonomously to failures, working in concert to maintain infrastructure and application availability.

4.7.3 Scenario 3: End-to-End Application Deployment

This scenario demonstrates the platform's core purpose: to seamlessly deploy, expose, and serve a **containerized application**. For this test, a sample web application was deployed using a standard **Kubernetes manifest** file. The manifest defined a **Deployment** to

ensure three replicas of the application pod were always running for redundancy, and a **Service** of type **LoadBalancer** to expose the application to the local network.

1. **Initiation and Deployment:** The deployment was initiated by applying the manifest file to the cluster using a single command: `kubectl apply -f web-application.yaml`.

2. **Kubernetes and MetalLB in Action:** Upon receiving the manifest, the Kubernetes control plane went into action. The Deployment controller created a ReplicaSet, which in turn scheduled three pods to run on the available worker nodes. Simultaneously, the Service object was created. Because its type was specified as `LoadBalancer`, the **MetalLB controller** detected it and assigned an available IP address from its pre-configured pool.

3. **Verification and Result:** The successful outcome of this process is confirmed in Figure 4.31. The output of the `kubectl get services` command shows the newly created `learn-nginx-app-service`. MetalLB has successfully provisioned it with an external IP address of `192.168.0.248`, making it accessible on the `mainvnet`.

```
└─$ kubectl get services
NAME                        TYPE           CLUSTER-IP      EXTERNAL-IP     PORT(S)          AGE
grafana                     LoadBalancer   10.43.143.51    192.168.0.247   80:30878/TCP     8d
kubernetes                  ClusterIP      10.43.0.1       <none>          443/TCP          9d
learn-nginx-app-service     LoadBalancer   10.43.43.193    192.168.0.248   80:30087/TCP     4m35s
prometheus-alertmanager     ClusterIP      10.43.148.189   <none>          9093/TCP         8d
```

Figure 4.31: The output of `kubectl get services`

To confirm end-to-end functionality, the assigned external IP was accessed from a web browser on the same network. As shown in Figure 4.32, the web application's interface loaded successfully, proving that traffic was correctly routed from the user, through the MetalLB entry point, to the service, and finally to one of the running application pods.

Figure 4.32: Accessing the web application successfully via its LoadBalancer IP.

This scenario validates the platform's ability to abstract the complexities of container scheduling, service discovery, and network load balancing, providing a simple and powerful mechanism for managing the lifecycle of cloud-native applications.

4.7.4 Scenario 4: Intelligent Provisioning with the Agentic AI Layer

This final scenario demonstrates the end-to-end workflow of the **Agentic AI layer**, showcasing its ability to translate a high-level, **natural language** user request into a provisioned virtual machine. This test validates the core research objective of achieving **intelligent provisioning**.

1. **User Request and Agent Interpretation:** The process begins in the web interface, where the user submits a simple request: `"create a webserver based on template vm id 100"`. As shown in Figure 4.33, the agent extracts the key parameters and presents this structured understanding back to the user for validation.

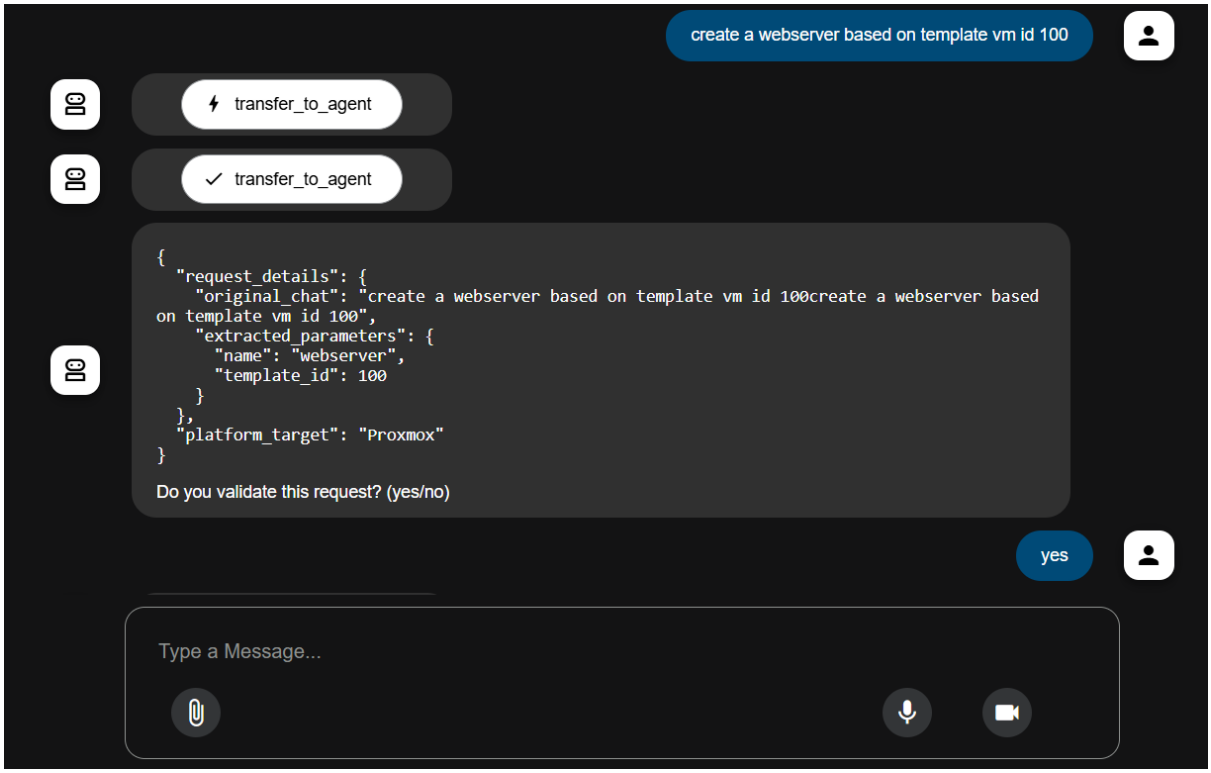

Figure 4.33: The agent's interpretation of the user's request and the subsequent validation prompt.

2. **Agent Reasoning and Manifest Generation:** Upon user validation ("yes"), the **Multi-Agent System** executes its main workflow. As the agent explained in Figure 4.34, this involves a multi-step reasoning process including data collection, strategic analysis, and iterative refinement to produce a best-practice-compliant **Terraform manifest**.

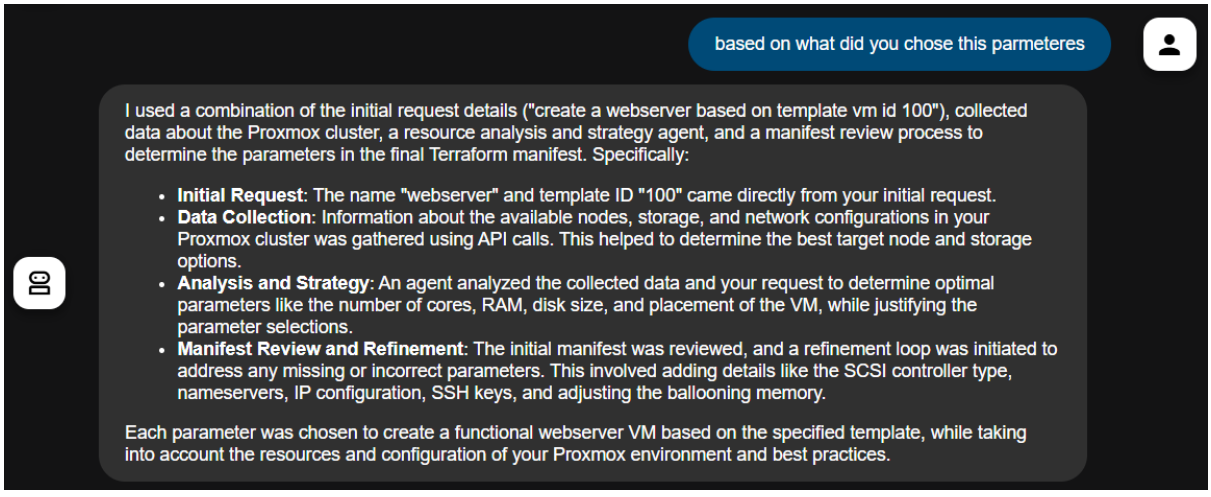

Figure 4.34: The agent explaining its multi-step reasoning process.

3. **Human-in-the-Loop Execution:** The final, validated Terraform file is provided

to the human operator for safety and oversight. The operator then executes `terraform apply`, with the successful creation of the new resource shown in Figure 4.35.

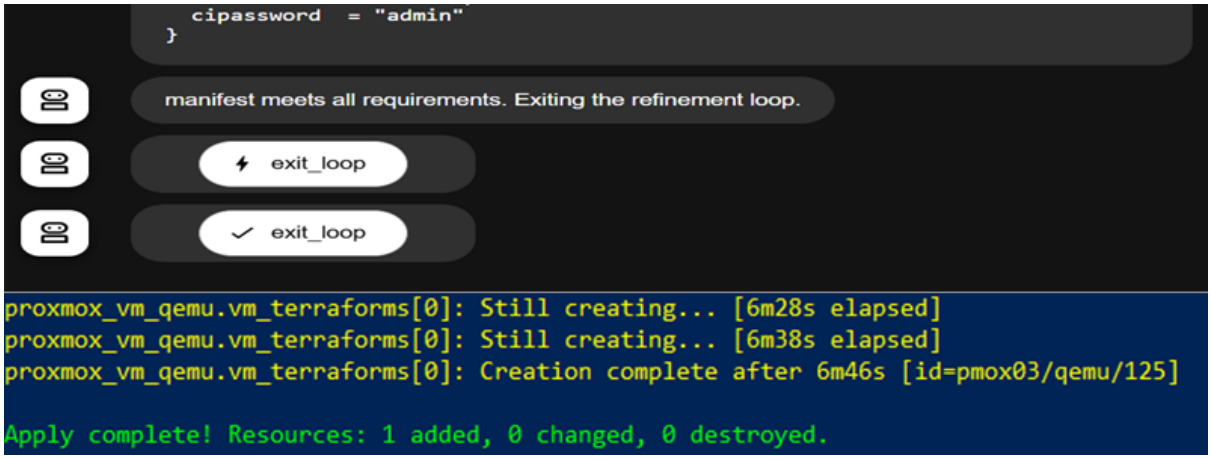

Figure 4.35: The agent-generated Terraform file and its execution.

4. **Verification of Outcome:** The result is immediately visible within the Proxmox web interface. Figure 4.36 shows the state of the `pmox03` host before and after the operation, confirming the new `webserver` VM (ID 125) has been successfully created.

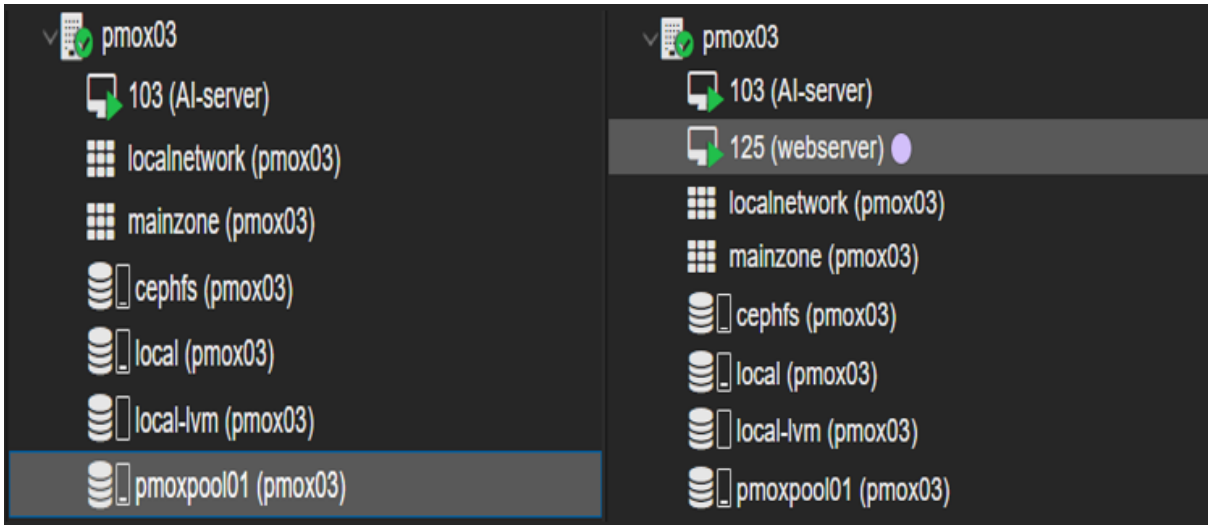

Figure 4.36: The Proxmox server view before (left) and after (right) the intelligent provisioning workflow

This scenario successfully validates the entire agentic workflow, demonstrating the system's ability to understand human intent, reason about the state of the environment, and intelligently generate a complex configuration file, bridging the gap between high-level commands and low-level infrastructure management.

4.8 Conclusion

This chapter has detailed the comprehensive implementation of a **multi-layered, autonomic cloud platform**, translating the conceptual architecture into a fully operational system. Beginning with the physical hardware, a resilient foundation was established using **Proxmox VE** for virtualization and **Ceph** for hyper-converged storage. Upon this foundation, a high-availability **k3s** cluster was deployed, managed entirely through a sophisticated **Infrastructure as Code** workflow that leveraged the complementary strengths of **Terraform** for provisioning and **Ansible** for configuration. This was supported by a robust virtual network fabric managed by **OPNsense** and a comprehensive observability stack with **Prometheus** and **Grafana**.

The platform's functionality was not only asserted but **practically** validated through a series of integrated scenarios. These tests demonstrated the system's capacity for seamless, automation-driven scalability, its **multi-layered resilience** against both hardware and software failures, and its efficiency in managing the end-to-end lifecycle of cloud-native applications. The culmination of this effort was the successful implementation and testing of the **Agentic AI layer**, which confirmed the system's ability to translate a high-level **natural language command** into a **provisioned infrastructure resource**.

In summary, the implementation phase is complete. The result is a fully operational, on-premises cloud platform that successfully meets the architectural design goals of **resilience, scalability, and intelligent provisioning**. This work successfully validates the thesis that by integrating a multi-layered, automated infrastructure with an agentic AI, it is possible to bridge the gap between high-level **human intent** and low-level **infrastructure execution**, fulfilling the **primary objectives of this research**.

Conclusion et perspectives

Conclusion générale

The growing complexity of modern IT infrastructures, fueled by dynamic workloads, the prevalence of containerized applications, and the increasing need for scalability and resilience, has necessitated the development of intelligent, automated resource provisioning solutions. This thesis successfully designed, implemented, and evaluated an intelligent multi-layered architecture that integrates virtualization, container orchestration and an AI-driven Multi-Agent System (MAS) to optimize resource provisioning and scaling in virtualized clusters. By leveraging open-source technologies and cutting-edge AI models, the proposed system bridges the gap between theoretical advancements and practical implementations in cloud infrastructure. The foundational layers of this architecture—Proxmox VE for virtualization and Ceph for hyper-converged storage—provide a robust and scalable base for resource abstraction and high availability. The lightweight Kubernetes distribution (k3s) adds a flexible container orchestration layer, enabling seamless application deployment and management. Further, the integration of a cloud-native monitoring stack (Prometheus and Grafana) ensures comprehensive observability, while the use of Infrastructure as Code (IaC) tools (Terraform and Ansible) simplifies the automation of complex workflows. The most significant contribution of this research lies in the implementation of the Agentic AI layer, which introduces intelligent provisoining capabilities via a Multi-Agent System (MAS). By integrating the Google Gemini model, the system transforms natural language user requests into deployable infrastructure solutions through iterative refinement and validation. This human-in-the-loop approach ensures precision, adaptability, and context-aware provisioning, enabling the platform to dynamically respond to workload changes, optimize resource allocation, and enhance operational efficiency.

Perspectives

While this research has demonstrated the feasibility and effectiveness of the proposed architecture, it also highlights several opportunities for future exploration and innovation:

- Security Enhancements: Security considerations, such as network isolation, secure API communication, and identity management, were intentionally simplified in this work.
- Integrating advanced security mechanisms such as zero-trust architectures, encrypted communication between agents, and role-based access control (RBAC) for multi-tenant environments could further enhance the system's robustness.

- Integration of Generative AI: While the MAS leverages advanced AI models for provisioning, future iterations could integrate Generative AI more deeply into workload prediction and resource optimization tasks. Generative models could enrich the system by generating synthetic workload traces, improving demand forecasting, and identifying optimal scaling strategies.
- Hybrid and Multi-Cloud Extensibility: The architecture currently focuses on a single hyper-converged private cloud environment. Expanding its capabilities to orchestrate hybrid or multi-cloud infrastructures would allow organizations to balance cost, performance, and compliance by dynamically shifting workloads across public and private clouds.

In conclusion, this thesis has laid the groundwork for a highly automated, intelligent, and resilient cloud infrastructure. By addressing the outlined challenges and pursuing the suggested future directions, this architecture has the potential to evolve into a comprehensive solution for managing next-generation cloud environments, empowering organizations to meet the demands of increasingly complex, dynamic, and distributed workloads.

Bibliography

[1] Containerized computing evolution. https://clouddocs.f5.com/training/community/nginx/html/class12/intro.html. Accessed: 2025-03.

[2] Sujith Surendran. Type 1 and type 2 hypervisor. http://vgyan.in/type-1-and-type-2-hypervisor//, 2014. Accessed: 2025-02.

[3] SR Shishira, A Kandasamy, and K Chandrasekaran. Survey on meta heuristic optimization techniques in cloud computing. In *2016 international conference on advances in computing, communications and informatics (ICACCI)*, pages 1434–1440. IEEE, 2016.

[4] Ian Smalley Stephanie Susnjara. What is virtualization? https://www.ibm.com/think/topics/virtualization, 2025. Accessed: 2025-02.

[5] Ian Smalley Stephanie Susnjara. What are containers? https://www.ibm.com/think/topics/containers, 2024. Accessed: 2025-04.

[6] Awodele Oludele, Emmanuel C. Ogu, Kuyoro Shade, and Umezuruike Chinecherem. On the evolution of virtualization and cloud computing: A review. 2014.

[7] Containers vs. virtual machines (vms). https://www.redhat.com/en/topics/containers/containers-vs-vms, 2023. Accessed: 2025-04-15.

[8] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. A view of cloud computing. *Communications of the ACM*, 53(4):50–58, 2010.

[9] Shahir Daya, Nguyen Van Duy, Kameswara Eati, Carlos M Ferreira, Dejan Glozic, Vasfi Gucer, Manav Gupta, Sunil Joshi, Valerie Lampkin, Marcelo Martins, et al. *Microservices from theory to practice: creating applications in IBM Bluemix using the microservices approach*. IBM Redbooks, 2016.

[10] Kelsey Hightower, Brendan Burns, and Joe Beda. *Kubernetes: Up and Running: Dive into the Future of Infrastructure*. O'Reilly Media, Sebastopol, CA, 2017.

[11] Tania Lorido-Botran, Jose Miguel-Alonso, and Juan A Lozano. A review of auto-scaling techniques for elastic applications in cloud environments. *Journal of Grid Computing*, 12(4):559–592, 2014.

[12] Sadeka Islam, Jacky Wai Keung, Kevin Lee, and Anna Liu. Empirical prediction models for adaptive resource provisioning in the cloud. *Future Gener. Comput. Syst.*, 28:155–162, 2012.

[13] Zijun Li, Linsong Guo, Jiagan Cheng, Quan Chen, Bingsheng He, and Minyi Guo. The serverless computing survey: A technical primer for design architecture. *arXiv preprint arXiv:2112.12921*, 2021. Decouples cloud architecture into virtualization, encapsulation, orchestration, and coordination layers.

[14] Christopher Clark, Keir Fraser, Steven Hand, Jacob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield. Live migration of virtual machines. In *Proceedings of the 2nd USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. USENIX Association, 2005.

[15] Ian Smalley Stephanie Susnjara. What are hypervisors? https://www.ibm.com/think/topics/hypervisors, October 2024. Consulted on february 2025.

[16] what-is-a-hypervisor. https://www.redhat.com/en/topics/virtualization/what-is-a-hypervisor, 2023.

[17] What is a hypervisors? https://www.vmware.com/topics/hypervisor. Consulted on february 2025.

[18] Dirk Merkel. Docker: Lightweight linux containers for consistent development and deployment. *Linux Journal*, 2014(239):2, 2014.

[19] Brendan Burns, Brian Grant, David Oppenheimer, Eric Brewer, and John Wilkes. Borg, omega, and kubernetes. *Queue*, 14(1):70–93, 2016.

[20] Tania Lorido-Botran, Jose Miguel-Alonso, and Juan A Lozano. A review of auto-scaling techniques for elastic applications in cloud environments. *ACM Computing Surveys*, 46(1):6, 2014.

[21] Saleha Alharthi, Afra Alshamsi, Anoud Alseiari, and Abdulmalik Alwarafy. Auto-scaling techniques in cloud computing: Issues and research directions. *Sensors*, 24(17):5551, 2024.

[22] David Buchaca Prats, Josep Lluís Berral, Chen Wang, and Alaa Youssef. Proactive container auto-scaling for cloud native machine learning services. In *Proceedings of the 2020 IEEE 13th International Conference on Cloud Computing (CLOUD)*, pages 475–479, 2020.

[23] Vedran Dakić, Goran Đambić, Jurica Slovinac, and Jasmin Redžepagić. Optimizing kubernetes scheduling for web applications using machine learning. *Electronics*, 14(5):863, 2025.

[24] Run a large-scale workload with flex-start with queued provisioning. https://cloud.google.com/kubernetes-engine/docs/how-to/provisioningrequest, 2024. Accessed: 2025-05-23.

[25] Jooyoung Kim, Abhishek Nautiyal, and Ankur Sethi. Optimize compute resources on amazon ecs with predictive scaling. https://aws.amazon.com/blogs/containers/optimize-compute-resources-on-amazon-ecs-with-predictive-scaling/, 2024. Accessed: 2025-05-23.

[26] Luyao Pei, Cheng Xu, Xueli Yin, and Jinsong Zhang. Multi-agent deep reinforcement learning for cloud-based digital twins in power grid management. *Journal of Cloud Computing*, 13(1):152, 2024.

[27] Long Wang, Anh V. Ngu, Songqing Chen, Xiang Bai, and Manish Parashar. A dynamic resource allocation approach for scientific workflows in cloud computing. *Future Generation Computer Systems*, 27(5):618–628, 2011.

[28] Rokaia Rokaia. Metaheuristic optimization for scheduling in cloud computing environments: A review. *Metaheuristic Optimization Review*, 2024.

[29] Anindya Bose, Sanjay Nag, et al. An overview of the state-of-the-art virtual machine placement algorithms for green cloud data centres. *Advancement in Management and Technology (AMT)*, 3(1):1–12, 2022.

[30] S Manikandan and M Chinnadurai. Virtualized load balancer for hybrid cloud using genetic algorithm. *Intell Autom Soft Comput*, 32(3):1459–1466, 2022.

[31] Xiong Fu, Qing Zhao, Junchang Wang, Lin Zhang, and Lei Qiao. Energy-aware vm initial placement strategy based on bpso in cloud computing. *Scientific Programming*, 2018(1):9471356, 2018.

[32] L. Wang, G. von Laszewski, A. Younge, X. He, and M. Kunze. Energy-aware virtual machine consolidation for cloud data centers. *Proceedings of IEEE International Conference on Cloud Computing Technology and Science*, pages 102–109, 2013.

[33] Z. Xiong and J. Xu. Towards energy-efficient cloud resource provisioning and scheduling using container-based virtualization. In *2014 IEEE International Conference on Cloud Computing*, pages 275–282, 2014.

[34] A. Beloglazov and R. Buyya. Energy-aware resource allocation heuristics for efficient management of data centers for cloud computing. *Future Generation Computer Systems*, 28(5):755–768, 2012.

[35] K. Ferdaus, S. Abdelwahed, S. Guica, and E. Bohrer. Ant colony optimization for energy-aware virtual machine placement in data centers. *Journal of Network and Computer Applications*, 45:117–128, 2014.

[36] A. Subramoney and P. Nyirenda. Comparative evaluation of ga–pso hybrid for workflow scheduling in cloud computing. *arXiv preprint arXiv:2012.00176*, 2020. Accessed: 2025-05-24.

[37] R.N. Calheiros, C.W. Tsai, C.A.F. de Rose, and M.A.S. Netto. A hybrid genetic algorithm and particle swarm optimization for workflow scheduling in cloud environments. *Future Generation Computer Systems*, 52:242–256, 2015.

[38] Said Nabi, Masroor Ahmad, Muhammad Ibrahim, and Habib Hamam. Adpso: adaptive pso-based task scheduling approach for cloud computing. *Sensors*, 22(3):920, 2022.

[39] L. Zhang and H. Chen. Binary particle swarm optimization for initial virtual machine placement in cloud computing. *Mathematical Problems in Engineering*, 2018:9471356, 2018. Accessed: 2025-05-24.

[40] P. Mohammadzadeh, S. Rahati, and A. Mosavi. Chaotic hybrid multi-objective optimization algorithm for scientific workflow scheduling in multisite clouds. *Applied Soft Computing*, 125:109466, 2023.

[41] ASTR Journal. Multi-resource particle swarm optimization for vm allocation in openstack, 2014. Accessed: 2025-05-23.

[42] Mohit Kumar, Subhash Chander Sharma, Shalini Goel, Sambit Kumar Mishra, and Akhtar Husain. Autonomic cloud resource provisioning and scheduling using meta-heuristic algorithm. *Neural Computing and Applications*, 32:18285–18303, 2020.

[43] Fahd N. Al-Wesabi, Marwa Obayya, Manar Ahmed Hamza, Jaber S. Alzahrani, Deepak Gupta, and Sachin Kumar. Energy aware resource optimization using unified metaheuristic optimization algorithm allocation for cloud computing environment. *Sustainable Computing: Informatics and Systems*, 35:100686, 2022.

[44] Syed Hamid Hussain Madni, Muhammed Faheem, Muhammad Younas, Maidul Hasan Masum, and Sajid Shah. Critical review on resource scheduling in iaas clouds: Taxonomy, issues, challenges, and future directions. *The Journal of Engineering*, 2024.

[45] Adnan Abid, Muhammad Faraz Manzoor, Muhammad Shoaib Farooq, Uzma Farooq, and Muzammil Hussain. Challenges and issues of resource allocation techniques in cloud computing. *KSII Trans. Internet Inf. Syst.*, 14:2815–2839, 2020.

[46] Min Cao, Yaoyu Li, Xupeng Wen, Yue Zhao, and Jianghan Zhu. Energy-aware intelligent scheduling for deadline-constrained workflows in sustainable cloud computing. *Egyptian Informatics Journal*, 2023.

[47] Mohammad Masdari and Afsane Khoshnevis. A survey and classification of the workload forecasting methods in cloud computing. *Cluster Computing*, 23:2399 – 2424, 2019.

[48] Jitendra Kumar, Rimsha Goomer, and Ashutosh Kumar Singh. Long short term memory recurrent neural network (lstm-rnn) based workload forecasting model for cloud datacenters. *Procedia Computer Science*, 125:676–682, 2018.

[49] Yonghua Zhu, Weilin Zhang, Yihai Chen, and Honghao Gao. A novel approach to workload prediction using attention-based lstm encoder-decoder network in cloud environment. *EURASIP Journal on Wireless Communications and Networking*, 2019, 2019.

[50] Jing Bi, Shuang Li, Haitao Yuan, and Mengchu Zhou. Integrated deep learning method for workload and resource prediction in cloud systems. *Neurocomputing*, 424:35–48, 2020.

[51] Jitendra Kumar and Ashutosh Kumar Singh. Workload prediction in cloud using artificial neural network and adaptive differential evolution. *Future Gener. Comput. Syst.*, 81:41–52, 2018.

[52] Sumit Kumar. Statistical vs machine learning vs deep learning modeling for time series forecasting, December 2022. Accessed: 2025-05-23.

[53] Martín Solís and Luis Alexánder Calvo-Valverde. Explaining when deep learning models are better for time series forecasting. *ITISE 2024*, 2024.

[54] Angelo Casolaro, Vincenzo Capone, Gennaro Iannuzzo, and Francesco Camastra. Deep learning for time series forecasting: Advances and open problems. *Inf.*, 14:598, 2023.

[55] Shereen Elsayed, Daniela Thyssens, Ahmed Rashed, Lars Schmidt-Thieme, and Hadi Samer Jomaa. Do we really need deep learning models for time series forecasting? *ArXiv*, abs/2101.02118, 2021.

[56] Zahra Fatemi, Minh-Thu T. Huynh, Elena Zheleva, Zamir Syed, and Xiaojun Di. Mitigating cold-start forecasting using cold causal demand forecasting model. *ArXiv*, abs/2306.09261, 2023.

[57] Andrea Rossi, Andrea Visentin, Diego Carraro, Steven D. Prestwich, and Kenneth N. Brown. Forecasting workload in cloud computing: towards uncertainty-aware predictions and transfer learning. *Clust. Comput.*, 28:258, 2025.

[58] Aya I Maiyza, Noha O Korany, Karim Banawan, Hanan A Hassan, and Walaa M Sheta. Vtgan: hybrid generative adversarial networks for cloud workload prediction. *Journal of Cloud Computing*, 12(1):97, 2023.

[59] Weiwei Lin, Kun Yao, Lan Zeng, Fagui Liu, Chun Shan, and Xiaobin Hong. A gan-based method for time-dependent cloud workload generation. *Journal of Parallel and Distributed Computing*, 168:33–44, 2022.

[60] Raymond Ang, Wai Kit Lau, and Khang Tai Le. Integrating large language models with splunk to augment enterprise logs workflows, December 8 2024. Accessed: 2025-05-24.

[61] Shivani Arbat, Vinodh Kumaran Jayakumar, Jaewoo Lee, Wei Wang, and In Kee Kim. Wasserstein adversarial transformer for cloud workload prediction. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 36, pages 12433–12439, 2022.

[62] Shreshth Tuli, Giuliano Casale, and Nicholas R. Jennings. Cilp: Co-simulation based imitation learner for dynamic resource provisioning in cloud computing environments, 2023.

[63] Shreshth Tuli, Giuliano Casale, and Nicholas R Jennings. Cilp: Co-simulation-based imitation learner for dynamic resource provisioning in cloud computing environments. *IEEE Transactions on Network and Service Management*, 20(4):4448–4460, 2023.

[64] Table 4: Comparative analysis of load balancing techniques. https://journalofcloudcomputing.springeropen.com/articles/10.1186/s13677-023-00473-z/tables/4, 2023. Accessed: 2025-05.

[65] Amazon Web Services, Inc. Using amazon bedrock agents to interactively generate infrastructure as code, 2024. Accessed: 2025-05-24.

[66] Amazon Web Services, Inc. Top analytics announcements of aws re:invent 2024, 2024. Accessed: 2025-05-24.

[67] Amazon Web Services, Inc. amazon-bedrock-rag-knowledgebases-agents-cloudformation. https://github.com/aws-samples/amazon-bedrock-rag-knowledgebases-agents-cloudformation, 2024. Accessed: 2025-05-24.

[68] Amazon Web Services, Inc. Aws announces support for predictive scaling for amazon ecs services, 2024. Accessed: 2025-05-02.

[69] Célia Ghedini Ralha, Aldo HD Mendes, Luiz A Laranjeira, Aletéia PF Araújo, and Alba CMA Melo. Multiagent system for dynamic resource provisioning in cloud computing platforms. *Future generation computer systems*, 94:80–96, 2019.

[70] Ali Belgacem, Saïd Mahmoudi, and Maria Kihl. Intelligent multi-agent reinforcement learning model for resources allocation in cloud computing. *Journal of King Saud University-Computer and Information Sciences*, 34(6):2391–2404, 2022.

[71] J Octavio Gutierrez-Garcia and Kwang Mong Sim. Agents for cloud resource allocation: An amazon ec2 case study. In *International Conference on Grid and Distributed Computing*, pages 544–553. Springer, 2011.

[72] Tong Cheng, Hang Dong, Lu Wang, Bo Qiao, Si Qin, Qingwei Lin, Dongmei Zhang, Saravan Rajmohan, and Thomas Moscibroda. Multi-agent reinforcement learning with shared policy for cloud quota management problem. In *Companion Proceedings of the ACM Web Conference 2023*, pages 391–395, 2023.

[73] Xiangqiang Gao, Rongke Liu, and Aryan Kaushik. Hierarchical multi-agent optimization for resource allocation in cloud computing. *IEEE Transactions on Parallel and Distributed Systems*, 32(3):692–707, 2021.

[74] Sofiane Kemchi, Abdelhafid Zitouni, and Mahieddine Djoudi. Self organization agent oriented dynamic resource allocation on open federated clouds environment. *ArXiv*, abs/2001.07496, 2016.

[75] Fouad Jowda and Muntasir Al-Asfoor. A multi-agent system framework for cloud resources allocation. *Journal of Al-Qadisiyah for computer science and mathematics*, 13(3):Page–78, 2021.

Annexes

Annex A

PROXMOX

A.1 Partitioning a Proxmox Disk Using CLI (Post-Installation)

This section explains how to partition a disk in a Proxmox VE environment using command-line tools after the initial installation. This is particularly useful when you have intentionally reserved free space on the primary disk for custom storage configurations.

A.1.1 Pre-installation Note

During the installation of Proxmox VE, it is highly recommended to use the installer option `hdsize`. This option defines how much of the primary disk should be allocated to the Proxmox system (specifically, to the LVM Volume Group ‘pve‘), ensuring that free space is left unallocated for later use.

Figure A.1: Setting the `hdsize` option in the Proxmox installer.

For example, to limit the main Proxmox installation to 100 GB, you would specify:

```
hdsize=100G
```

This command creates a 100 GB Proxmox system partition and leaves the rest of the disk space unformatted and available for manual partitioning.

A.1.2 Initial Disk Layout

After installing Proxmox with an option like `hdsize=100G` on a 1 TB disk, the initial disk layout will look similar to the following when inspected with `lsblk`. Notice the unallocated space on `sda` that is not part of any partition.

```
sda                 8:0    0 953.9G  0 disk
|-sda1              8:1    0  1007K  0 part
|-sda2              8:2    0     1G  0 part /boot/efi
‘-sda3              8:3    0    99G  0 part
  |-pve-swap      252:0    0     8G  0 lvm  [SWAP]
```

```
|-pve-root       252:1    0  34.7G  0 lvm  /
|-pve-data_tmeta 252:2    0     1G  0 lvm
| `-pve-data     252:4    0    42G  0 lvm
`-pve-data_tdata 252:3    0    42G  0 lvm
  `-pve-data     252:4    0    42G  0 lvm
```

A.1.3 Repartitioning with `gdisk`

To utilize the remaining ~850 GB of free disk space, we will create a new partition, `sda4`, using the `gdisk` utility.

Step 1: Launch `gdisk`

Open the GPT disk partitioning tool for the target disk.

```
gdisk /dev/sda
```

Step 2: Create a New Partition

Follow the interactive prompts to create a new partition. Use the default values to allocate all remaining free space to this new partition.

```
Command (? for help): n
Partition number (4-128, default 4): 4
First sector: (press Enter)
Last sector: (Press Enter to use the remaining free space)
Hex code or GUID (L to show codes, Enter = 8300): (press Enter)
```

The hex code `8300` corresponds to a standard "Linux filesystem" partition type.

Step 3: Print and Verify the Layout

Use the `p` command to print the new partition table and verify that it matches your intentions before writing the changes to the disk.

```
Command (? for help): p
```

The output should now include the new partition, `sda4`.

```
Number  Start (sector)    End (sector)  Size       Code  Name
   1              34            2047   1007.0 KiB  EF02
   2            2048         2099199   1024.0 MiB  EF00
   3         2099200       209715200   99.0 GiB    8E00  Linux LVM
   4       209715208      1998585863   853.9 GiB   8300  Linux filesystem
```

Step 4: Write Partition Table to Disk

Use the `w` command to save the changes. This is a destructive operation, so proceed with caution.

```
Command (? for help): w
```

```
Final checks complete. About to write GPT data. THIS WILL OVERWRITE
EXISTING PARTITIONS!!
```

```
Do you want to proceed? (Y/N): y
```

Confirm with `Y` to write the new partition table.

Step 5: Reboot the System

A reboot is required for the kernel to recognize the new partition table.

```
reboot
```

A.1.4 Post-Reboot Verification

After the system has rebooted, log back in and run `lsblk` again to verify that the new partition is visible to the operating system.

```
lsblk
```

You should now see the new partition (`sda4` in this example, though the number might vary) listed.

```
sda                  8:0    0 953.9G  0 disk
|-sda1               8:1    0  1007K  0 part
|-sda2               8:2    0     1G  0 part /boot/efi
|-sda3               8:3    0    99G  0 part
| |-pve-swap       252:0    0     8G  0 lvm  [SWAP]
| |-pve-root       252:1    0  34.7G  0 lvm  /
| |-pve-data_tmeta 252:2    0     1G  0 lvm
| | `-pve-data     252:4    0    42G  0 lvm
| `-pve-data_tdata 252:3    0    42G  0 lvm
|   `-pve-data     252:4    0    42G  0 lvm
`-sda4               8:4    0 853.9G  0 part
```

The new partition `/dev/sda4` is now ready to be formatted with a filesystem (e.g., ext4, xfs) or used as a physical volume for a new LVM group, ZFS pool, etc.

A.2 Creating a Template for Virtual Machines in Proxmox

This guide will walk you through the steps to create a reusable template for virtual machines (VMs) in Proxmox using the Ubuntu Cloud Image. Templates allow you to quickly deploy new VMs with pre-configured settings, saving time and ensuring consistency. This section covers both the manual step-by-step process and a fully automated script-based method.

A.2.1 Prerequisites

Before proceeding, ensure the following:

- You have a Proxmox VE environment set up.
- You have downloaded the Ubuntu Cloud Image from Ubuntu Cloud Images and uploaded it to your Proxmox ISO storage.
- Sufficient storage and resources are available on your Proxmox node.

A.2.2 Manual Template Creation (GUI & CLI)

This subsection details the manual process of creating a template using a combination of the Proxmox Web UI and command-line interface.

Step 1: Upload the Cloud Image

- Log in to the Proxmox web interface.
- Navigate to `Datacenter > prmox01 > local > ISO Images`.
- To upload the Ubuntu Cloud Image, you have two options:
  1. If the image is already downloaded to your local machine, click the `Upload` button and select the file.
  2. Alternatively, download the image directly from the web by clicking `Download from URL`, entering the URL `https://cloud-images.ubuntu.com/noble/current/noble-s` providing a name for the image, and clicking `Download`.

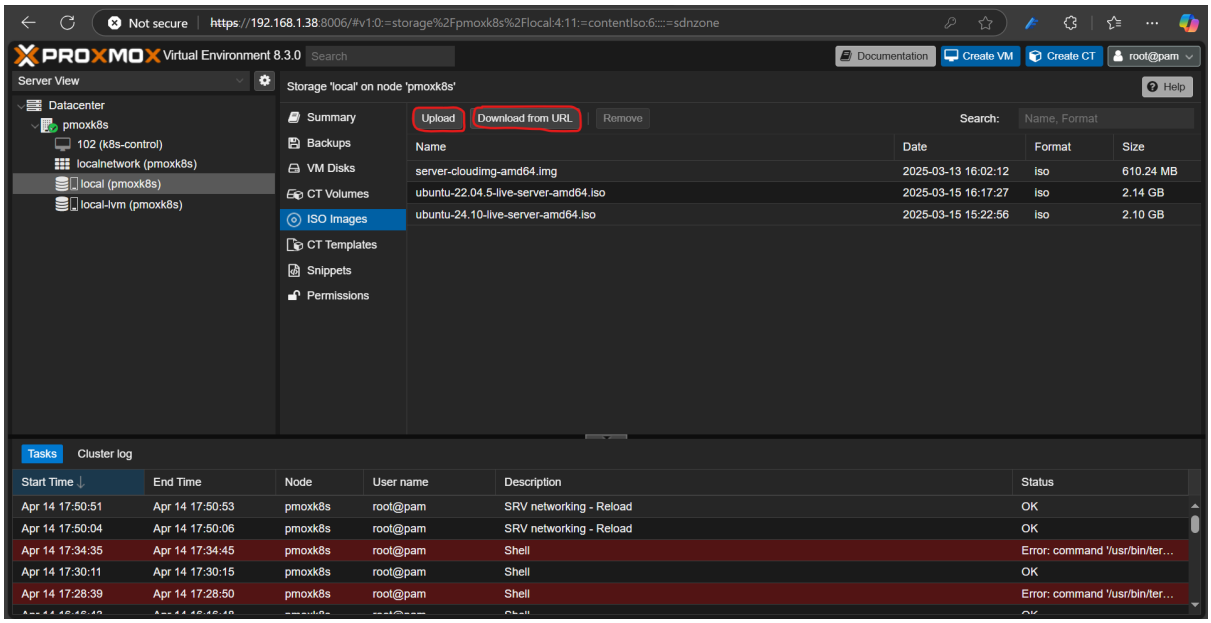

Figure A.2: Upload the Ubuntu Cloud Image.

Step 2: Create and Configure the VM

There are two alternatives for creating the base VM: using the graphical interface or the command line.

Alternative 1: Using the GUI

- Under the `pmox01` node, click the `Create VM` button.
- **General:** Give the template a name and an ID.
- **OS:** Select `Do not use any media`.

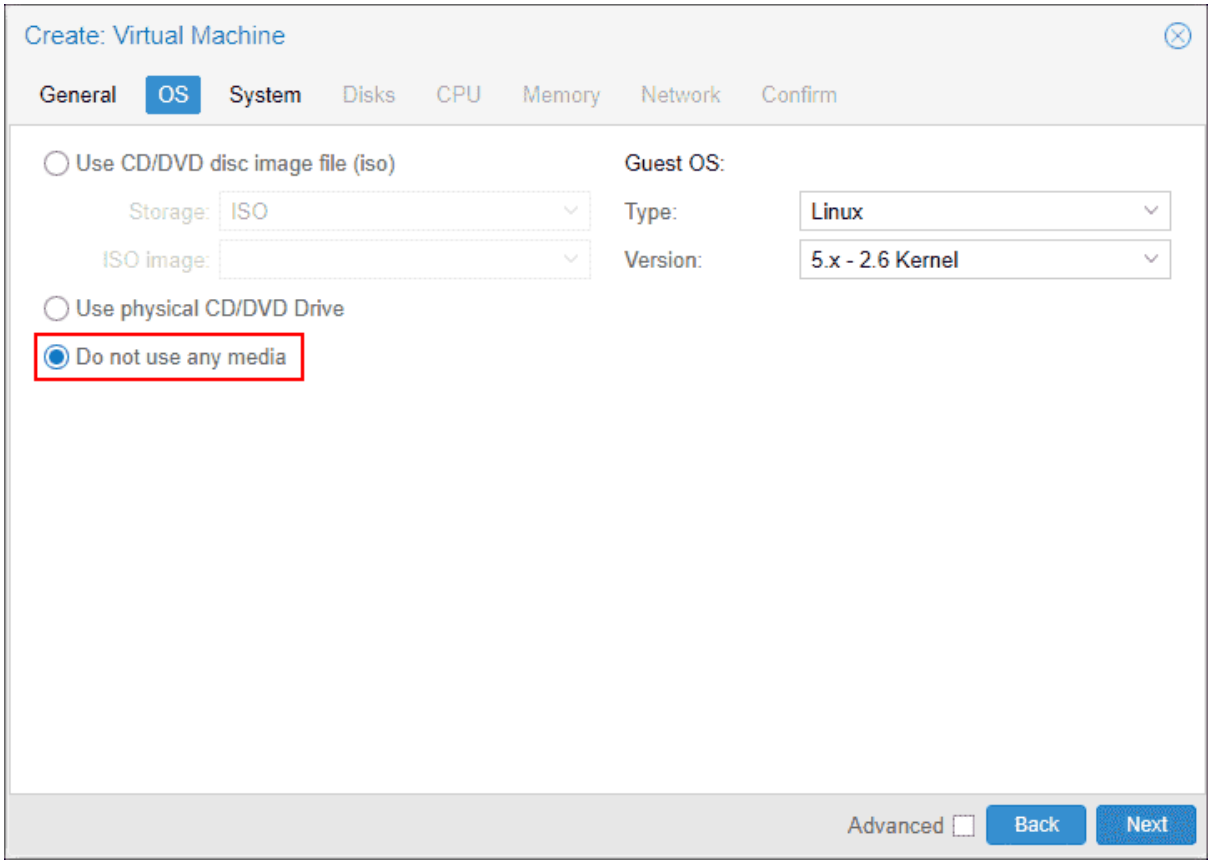

Figure A.3: Create a VM without any ISO media.

- **System:** Check the `Qemu Agent` box.
- **Disks:** Remove the default hard disk by selecting it and clicking the trash icon. We will import the cloud image manually.
- **CPU, Memory, Network:** Configure these as desired, or keep the defaults.
- Complete the wizard to create the VM.

Alternative 2: Using the CLI

These commands perform the same actions as the GUI steps, plus the subsequent disk and hardware configuration.

- Create the base VM:

```
qm create 100 --name ubuntu-template --memory 2048 --cores 2 --net0 virtio,bridge
```

- Set necessary hardware options (QEMU agent, Cloud-Init drive, serial console):

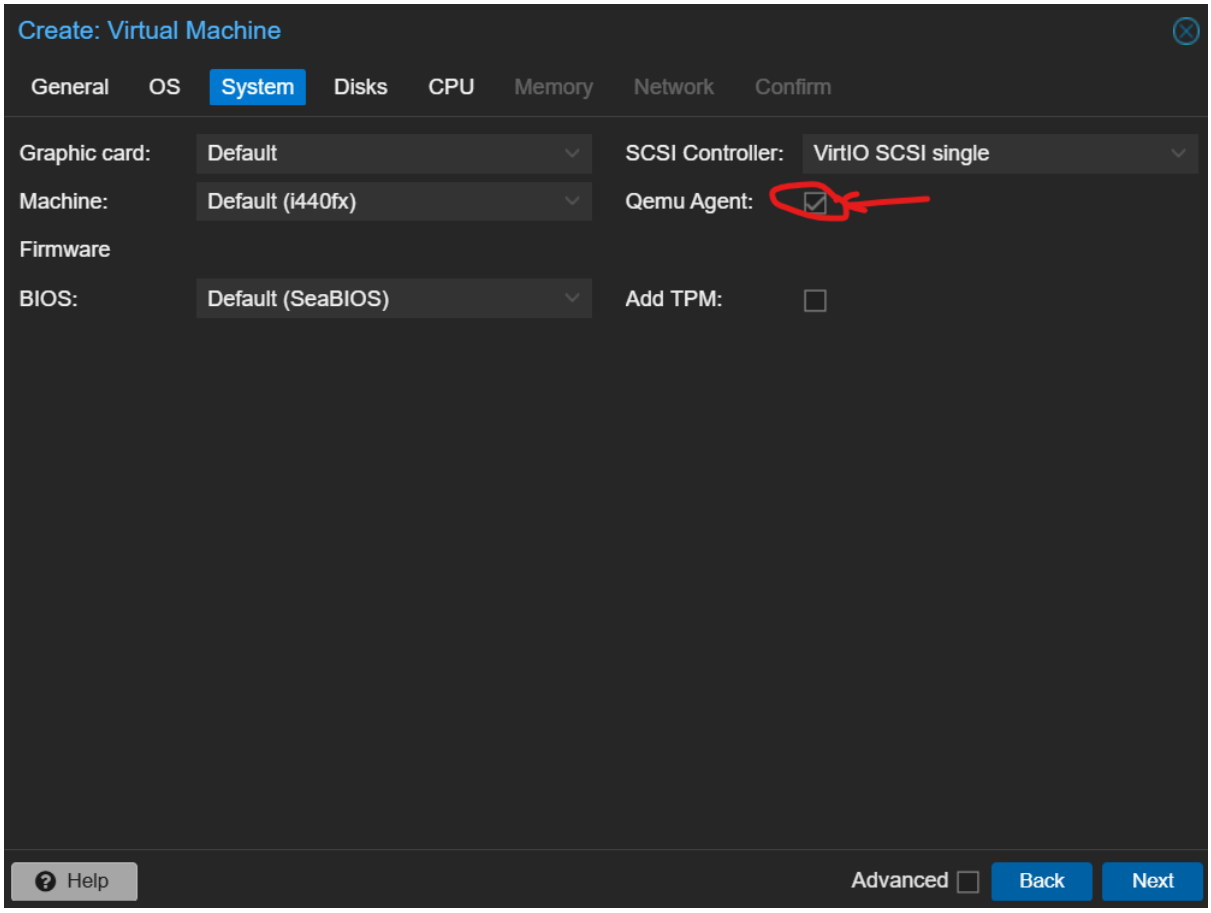

Figure A.4: Enable the QEMU Agent in the System window.

```
qm set 100 --agent enabled=1
qm set 100 --ide2 local-lvm:cloudinit
qm set 100 --serial0 socket --vga serial0
```

- Copy the cloud image from ISO storage, resize it, and import it as a disk for the VM (replace image/disk names and VM ID as needed):

```
# Copy the image from storage to a working directory
cp /var/lib/vz/template/iso/noble-server-cloudimg-amd64.img /root/noble.qcow2

# Resize the image
qemu-img resize /root/noble.qcow2 32G

# Import the resized image as a disk for the VM
qm importdisk 100 /root/noble.qcow2 local-lvm
```

- Attach the new disk and set the boot order:

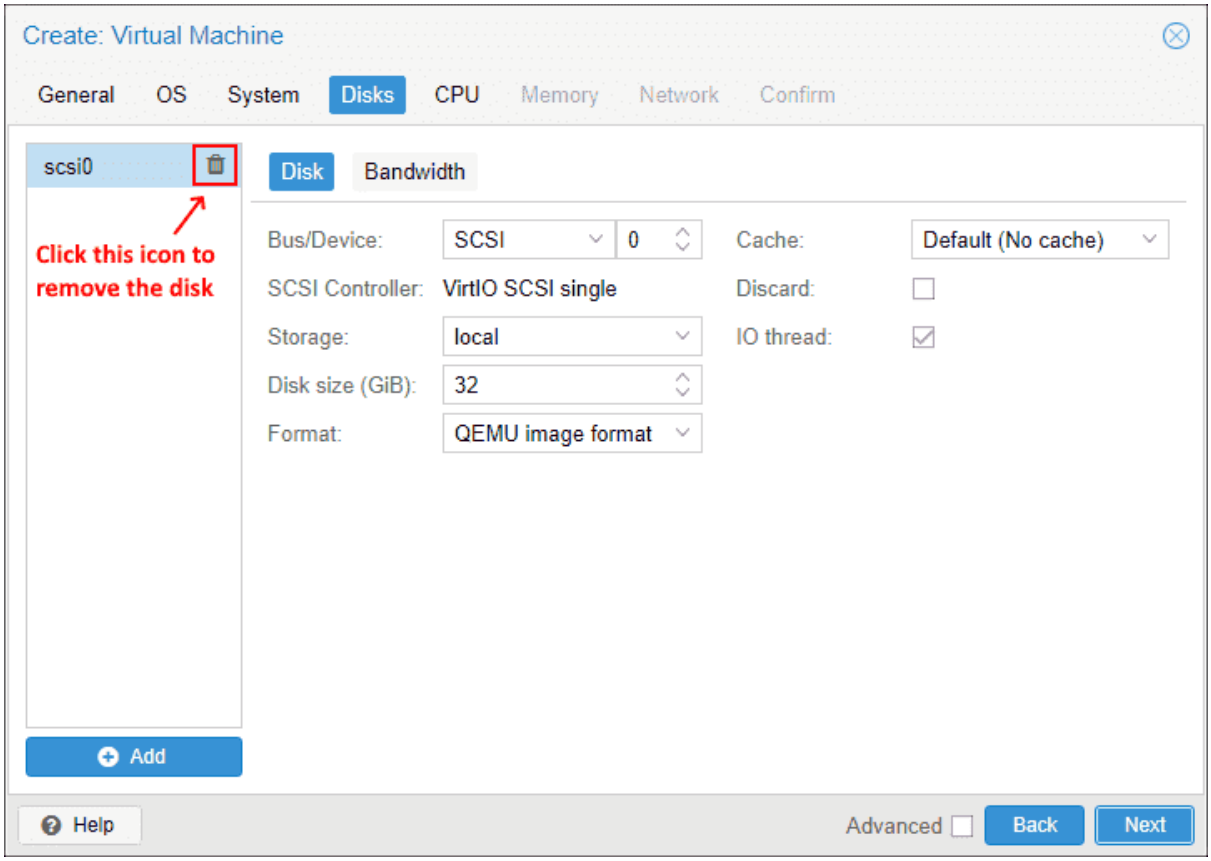

Figure A.5: Remove the default hard disk from the VM.

```
# Attach the disk as a SCSI drive
qm set 100 --scsihw virtio-scsi-pci --scsi0 local-lvm:vm-100-disk-0

# Set boot order to prioritize the Cloud-Init drive and then the new disk
qm set 100 --boot order="ide2;scsi0;net0" --bootdisk scsi0
```

Step 3: Customize the Cloud Image

- In the Proxmox UI, select the new VM and go to the `Cloud-Init` panel.
- Configure user, password, SSH keys, and network settings as desired.
- Click `Regenerate Image` to apply the settings.

Step 4: Prepare the VM for Template Conversion (Optional)

If you need additional software or updates in your base template, you can perform these steps:

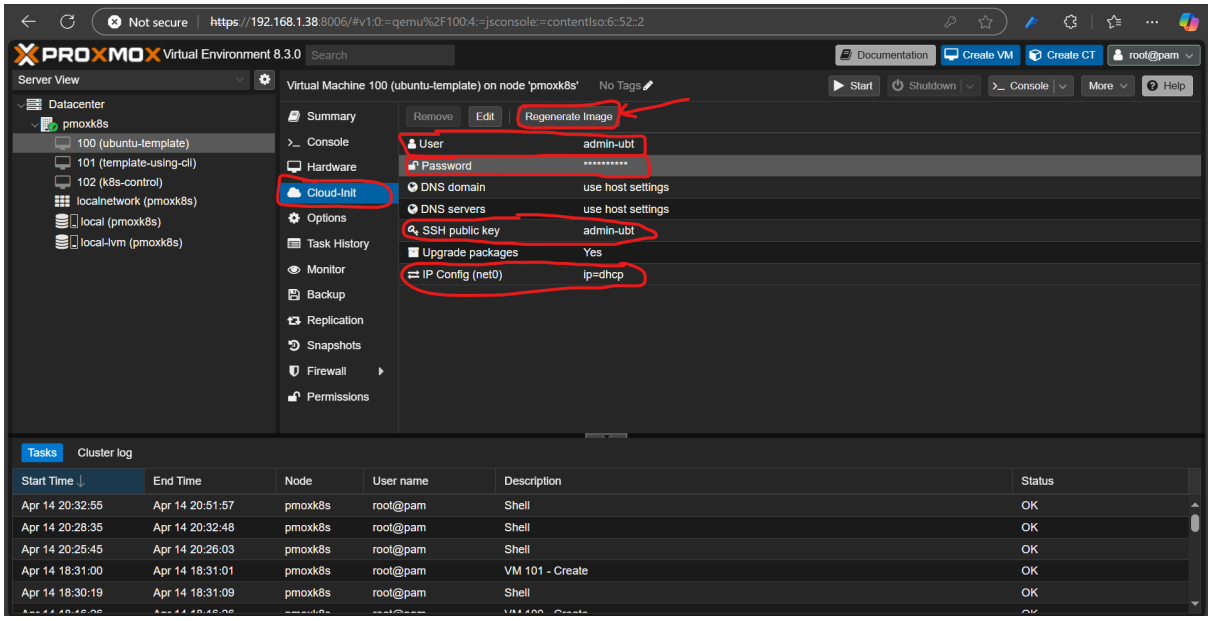

Figure A.6: Customize Cloud-Init settings.

- Start the VM for the first time.
- Log in and install any necessary updates and software (e.g., `apt update && apt upgrade -y`).
- Clean up temporary files and clear logs to minimize the template size.
- Shut down the VM cleanly.

Step 5: Convert the VM to a Template

Once the VM is fully configured and shut down, convert it into a template.

- In the Proxmox web interface, right-click on the VM and select `Convert to Template`.
- Alternatively, use the CLI (replace 100 with your VM ID):

```
qm template 100
```

A.2.3 Automated Template Creation with a Bash Script

As an alternative to the manual steps, the entire process can be automated with a single bash script. This is the most efficient method for creating templates consistently.

First, save the following script as a file (e.g., `create_template.sh`) on your Proxmox host.

Important Note: You must delete all the comments (lines starting with #) from the script for it to execute correctly without errors.

```
# create a ubuntu noble VM template
# This script creates a VM template for Ubuntu Noble
#!/bin/bash

# Variables
VM_ID=101
VM_NAME="ubuntu-template-using-bash"
CLOUD_IMAGE_URL_OFFLINE="/var/lib/vz/template/iso/noble-server-cloudimg-amd64.img"
CLOUD_IMAGE_PATH="/root/noble-server-cloudimg-amd64.qcow2"
CLOUD_IMAGE_PATH1="noble-server-cloudimg-amd64.qcow2"
DISK_SIZE="32G"
MEMORY=1024
CORES=1

# Step 1: Copy the image
echo "Copying the image..."
cp "$CLOUD_IMAGE_URL_OFFLINE" "$CLOUD_IMAGE_PATH"

# Step 2: Resize the disk
echo "Resizing the disk..."
qemu-img resize "$CLOUD_IMAGE_PATH1" "$DISK_SIZE"

# Step 3: Create the VM
echo "Creating the VM..."
qm create $VM_ID \
  --name $VM_NAME  \
  --memory $MEMORY \
  --cores $CORES \
  --net0 virtio,bridge=vmbr0 \
  --ide2 local-lvm:cloudinit \
  --onboot 1 \
  --agent 1

# Step 4: Import the disk
echo "Importing the disk..."
qm importdisk $VM_ID "$CLOUD_IMAGE_PATH1" local-lvm
```

```
# Step 5: Attach the disk and configure hardware
echo "Attaching the disk and configuring hardware..."
qm set $VM_ID --scsihw virtio-scsi-pci --scsi0 local-lvm:vm-$VM_ID-disk-0
qm set $VM_ID --serial0 socket --vga serial0
qm set $VM_ID --boot order="ide2;scsi0;net0" --bootdisk scsi0

# Step 6: Configure Cloud-Init
echo "Configuring Cloud-Init..."
mkdir -p /root/.ssh
chmod 700 /root/.ssh
touch ~/.ssh/id_ed25519.pub
echo "ssh-ed25519 AAAAC3...[your-public-key]... admin-ubt" > ~/.ssh/id_ed25519.pub

qm set $VM_ID \
  --ciuser admin-ubt \
  --cipassword admin \
  --sshkey ~/.ssh/id_ed25519.pub \
  --ipconfig0 ip=dhcp

# Step 7: Convert the VM to a template
echo "Converting the VM to a template..."
qm template $VM_ID

echo "VM Template creation completed successfully!"
```

After saving the script (and removing comments), make it executable and run it:

```
# Make the script executable
chmod +x create_template.sh

# Execute the script
./create_template.sh
```

Creating VM templates, whether through the step-by-step manual process or by using an automated script, is a fundamental practice in Proxmox VE. It significantly streamlines the deployment of new virtual machines, ensuring consistency and saving valuable administration time. By mastering these techniques, administrators can maintain a more efficient, scalable, and manageable virtualization environment.

A.3 Ceph Cluster Configuration Guide

This subsection provides a step-by-step guide to the Ceph cluster configuration performed through the Proxmox web interface.

A.3.1 Step 1: Cluster Initialization and Monitor Quorum

The process began with installing the necessary Ceph packages on each Proxmox node via the UI. The cluster was then initialized by creating the first Monitor (MON) on node `pmox01` and defining the public network (`172.25.5.0/24`). To establish a resilient, highly-available quorum, additional Monitors were subsequently created on nodes `pmox02` and `pmox03`, as shown in Figure A.7. This ensures that the cluster can maintain an operational state even if one monitor node becomes unavailable.

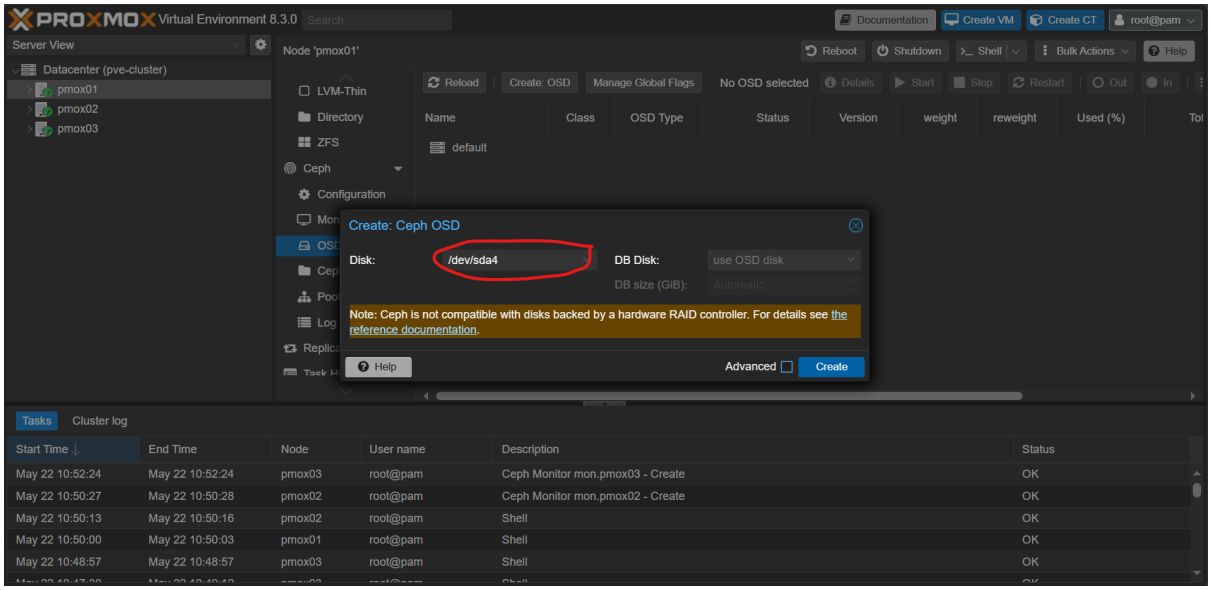

Figure A.7: Creation of Additional Ceph Monitors for Quorum

A.3.2 Step 2: Creating Object Storage Daemons (OSDs)

With the Monitor quorum established, the next step was to add storage capacity to the cluster. On each of the three Proxmox hosts, the prepared `/dev/sda4` partition was selected and designated as an Object Storage Daemon (OSD). This action incorporates the disk into the Ceph cluster, making its storage available for distribution and replication across the nodes (Figure A.8).

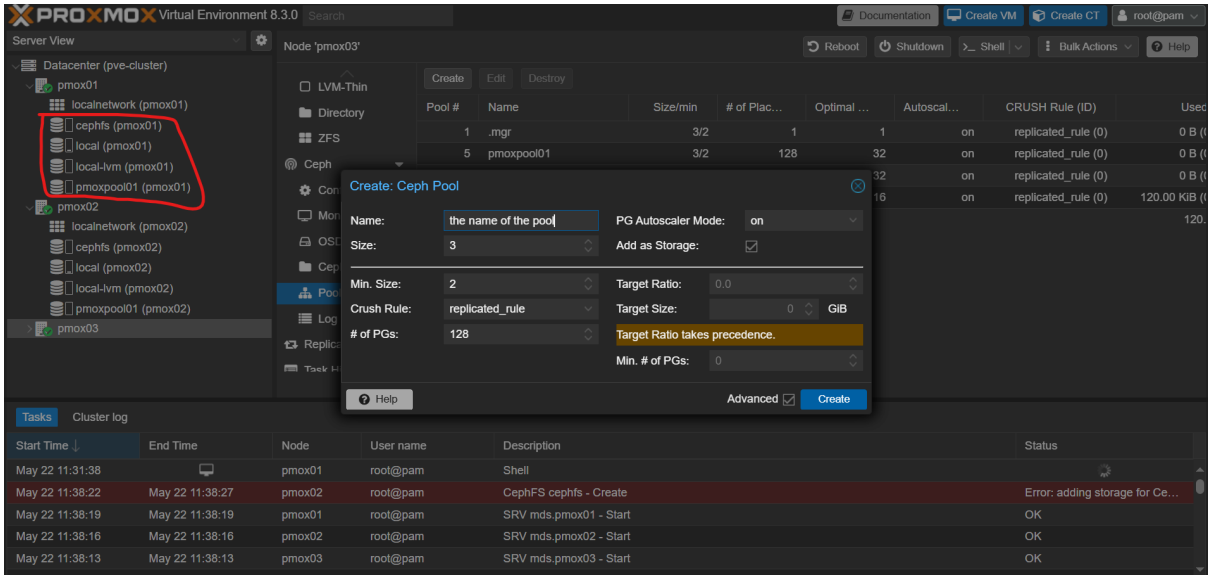

Figure A.8: Creating a Ceph Object Storage Daemon (OSD) on the Prepared Partition

A.3.3 Step 3: Provisioning Storage Pools and Filesystems

With the underlying OSDs providing a unified storage fabric, two distinct types of logical storage were provisioned:

- **Replicated Block Storage Pool (RBD):** A replicated pool named `pmoxpool01` was created to provide resilient block storage for virtual machine disks. As shown in Figure A.9, this pool was configured with a replication `size` of three and a `min_size` of two. This configuration directly fulfills the research objective of ensuring data resilience, as the system can tolerate the complete failure of one node without data loss or service interruption for stored VM disks.
- **Shared File System (CephFS):** To provide shared file storage accessible across all nodes, a CephFS file system named `cephfs` was created (Figure A.10). This process automatically set up the necessary data and metadata pools and deployed redundant Metadata Servers (MDS) across the cluster for high availability and fault tolerance.

A.3.4 Conclusion

With the OSDs contributing storage, the MONs ensuring quorum, the replicated pool providing resilient block devices, and CephFS offering shared file storage, the hyper-converged storage layer is now complete. The culmination of these steps is a unified, resilient, and fully operational storage platform integrated directly within the Proxmox cluster, ready to support the compute workloads.

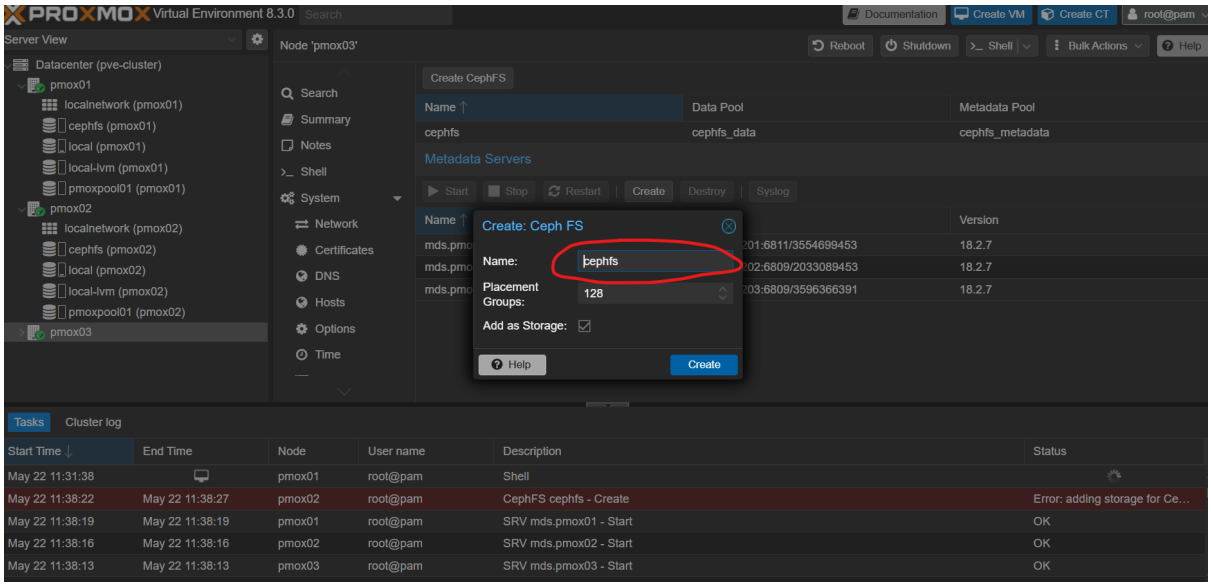

Figure A.9: Creation and Configuration of the 'pmoxpool01' Replicated Pool

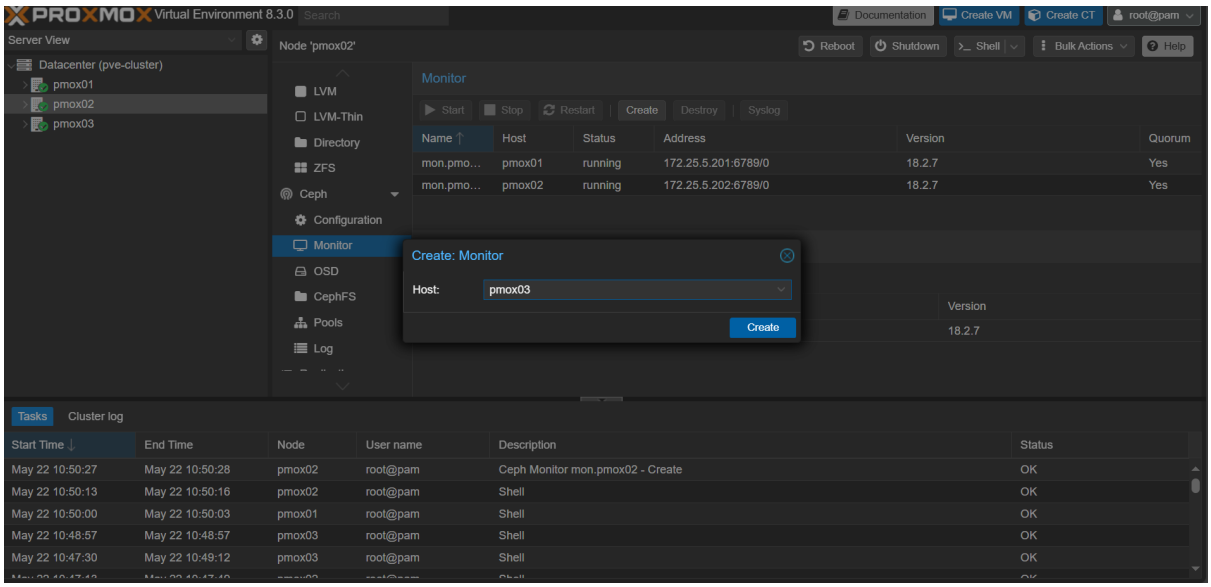

Figure A.10: Creation of the CephFS File System

Annex B

OPNsense

B.1 OPNsense Installation and Configuration

The foundation of the virtual network infrastructure for this project is the OPNsense firewall and routing appliance. Deployed as a virtual machine within the Proxmox environment, the initial installation was performed using the standard OPNsense installer via the Proxmox console. A critical step in the initial setup involved gaining access to the web-based management interface. By default, the OPNsense GUI is accessible only on its LAN interface, which was connected to the isolated `mainvnet` network bridge. To overcome this, a temporary "jump box" VM running Ubuntu Desktop was deployed onto the same `mainvnet`. This jump box, assigned a static IP within the LAN subnet (`192.168.0.2`), provided the necessary browser environment to connect to the OPNsense GUI at `https://192.168.0.1` and complete the initial configuration wizard.

B.1.1 Scope of OPNsense Configuration

For the purposes of this research, the OPNsense configuration was focused on providing core Layer 3 and 4 services essential for the Kubernetes cluster's operation. Advanced features such as dynamic routing protocols (e.g., BGP), VPN services, and complex VLAN trunking were intentionally omitted to reduce complexity and maintain focus on the primary research objectives.

It is important to note, however, that OPNsense is a highly capable platform. These advanced networking and security features could be implemented in future work to enhance the scalability, flexibility, and security posture of the environment. For this implementation, the firewall functionality was also disabled after the initial setup to ensure unrestricted traffic flow within the trusted lab environment, simplifying the deployment and testing of containerized services.

B.1.2 Final Service Configuration

The core network configuration within OPNsense was then finalized as follows:

1. **Interface Assignment:** The virtual NICs were assigned to logical interfaces within OPNsense. As shown in Figure B.1, `vtnet0` was configured as the LAN interface with a static IP of `192.168.0.1/16`, and `vtnet1` was configured as the WAN interface with a static IP of `172.25.5.204/24`.

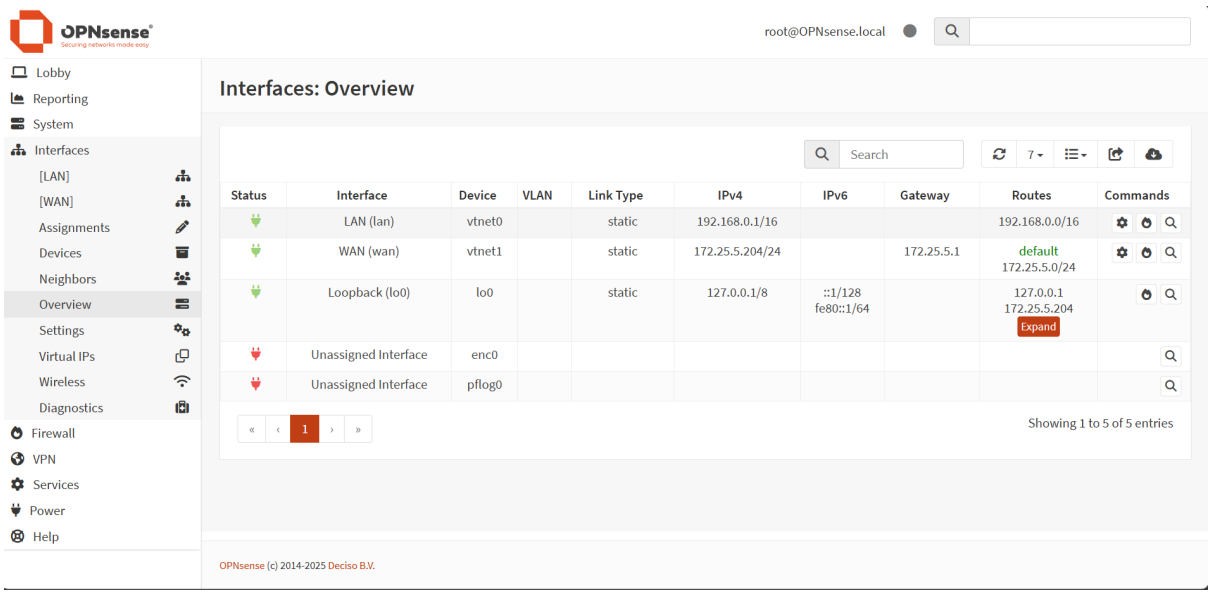

Figure B.1: OPNsense interface overview

2. **Gateway and Routing:** The default route for all outbound traffic was directed through the physical network's gateway. The `WAN_GW` was explicitly defined with the IP address `172.25.5.1`, ensuring that traffic from the virtual LAN can be routed to the internet (Figure B.2).

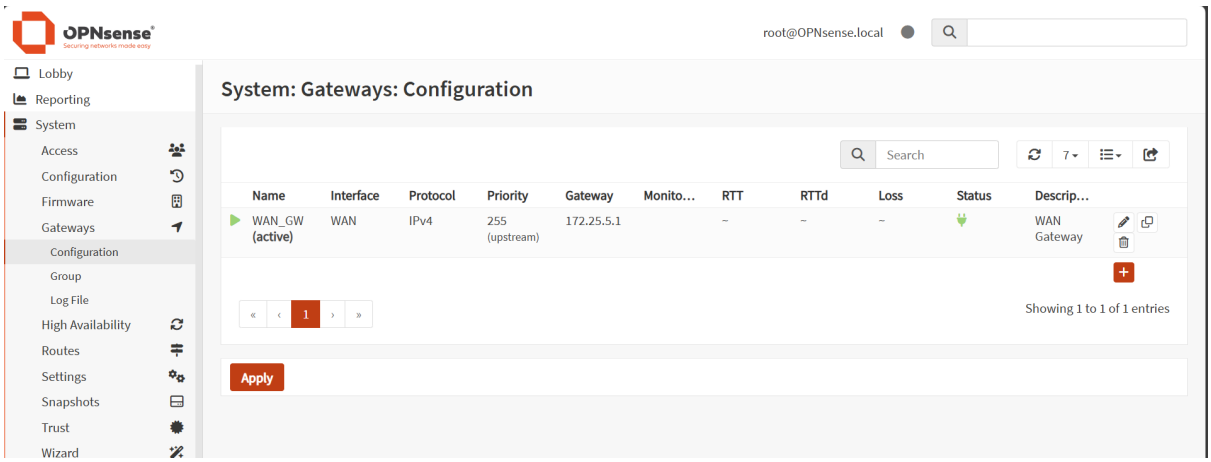

Figure B.2: Gateway configuration in OPNsense

3. **DHCP Service:** To automate IP address management for clients on the virtual network, the ISC DHCP server was enabled on the LAN interface. The server was configured to manage the `192.168.0.0/16` subnet, leasing addresses from a dynamic pool ranging from `192.168.0.10` to `192.168.0.245` (Figure B.3). This provides automatic network configuration for all future VMs and container workloads.

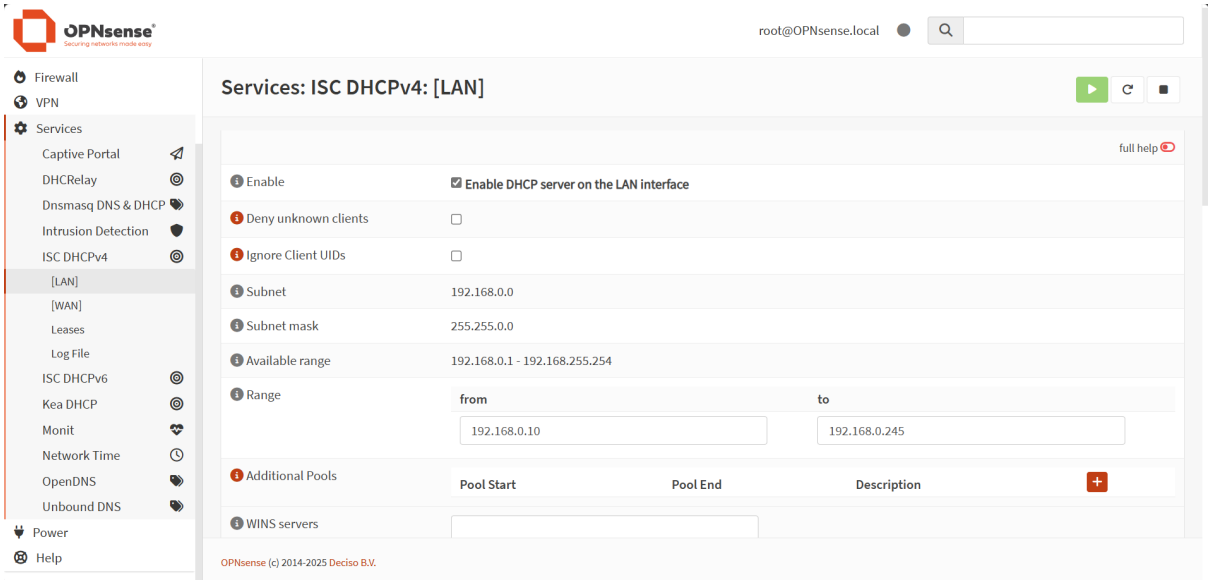

Figure B.3: DHCP server configuration for the LAN interface

The successful configuration resulted in a fully operational virtual router. The OPNsense dashboard (Figure B.4) provides a summary of the operational state, confirming the status of the interfaces, the active WAN gateway, and the system information, thus establishing a centralized point of management and monitoring for the entire virtual network.

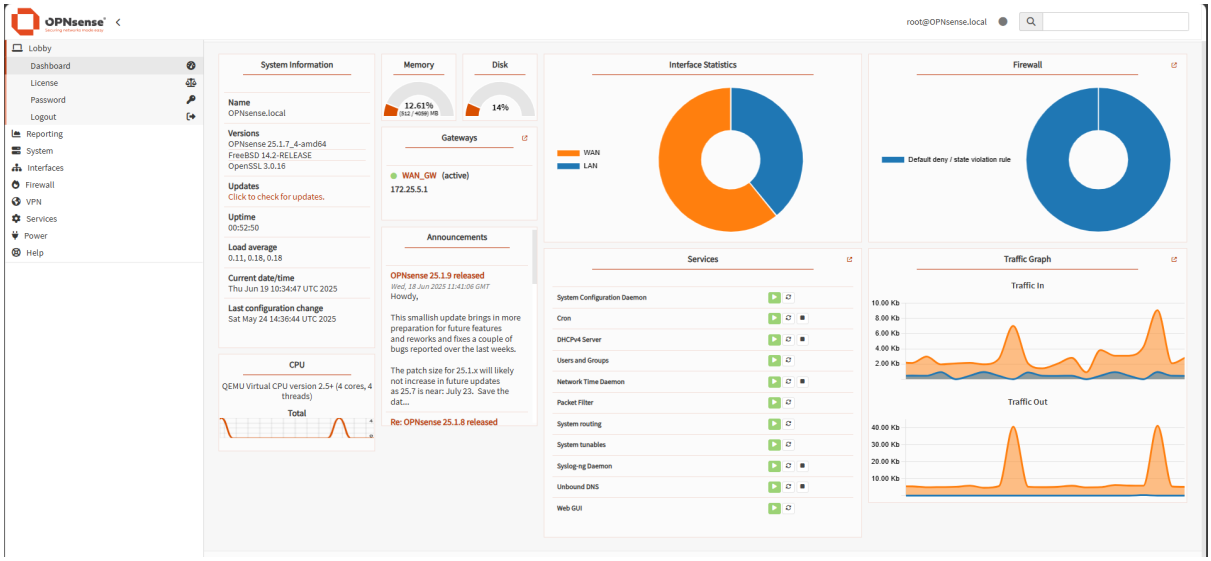

Figure B.4: The OPNsense dashboard showing the final operational state

Upon completion of these steps, a robust and centrally managed network foundation was established. The OPNsense appliance now provides essential services, including DHCP for automatic IP address allocation and gateway routing for external connectivity. This configured network environment is a critical prerequisite, providing the necessary connectivity and IP management for the subsequent deployment of the Kubernetes cluster nodes. The stage is now set for building the compute layer of the infrastructure.

Annex C

Implementation Architecture Diagrams

This annex provides detailed diagrams illustrating the practical implementation of the architecture described in this thesis. These visual aids are intended to offer a holistic and practical overview of the system's components and their interactions.

C.1 Global Implementation Architecture

Figure C.1 provides a comprehensive overview of the entire implemented platform, from the physical hardware to the agentic AI layer. It maps the technologies selected in Chapter 4 to their respective roles within the multi-layered architecture, illustrating the data and control flows that enable the system's autonomic capabilities.

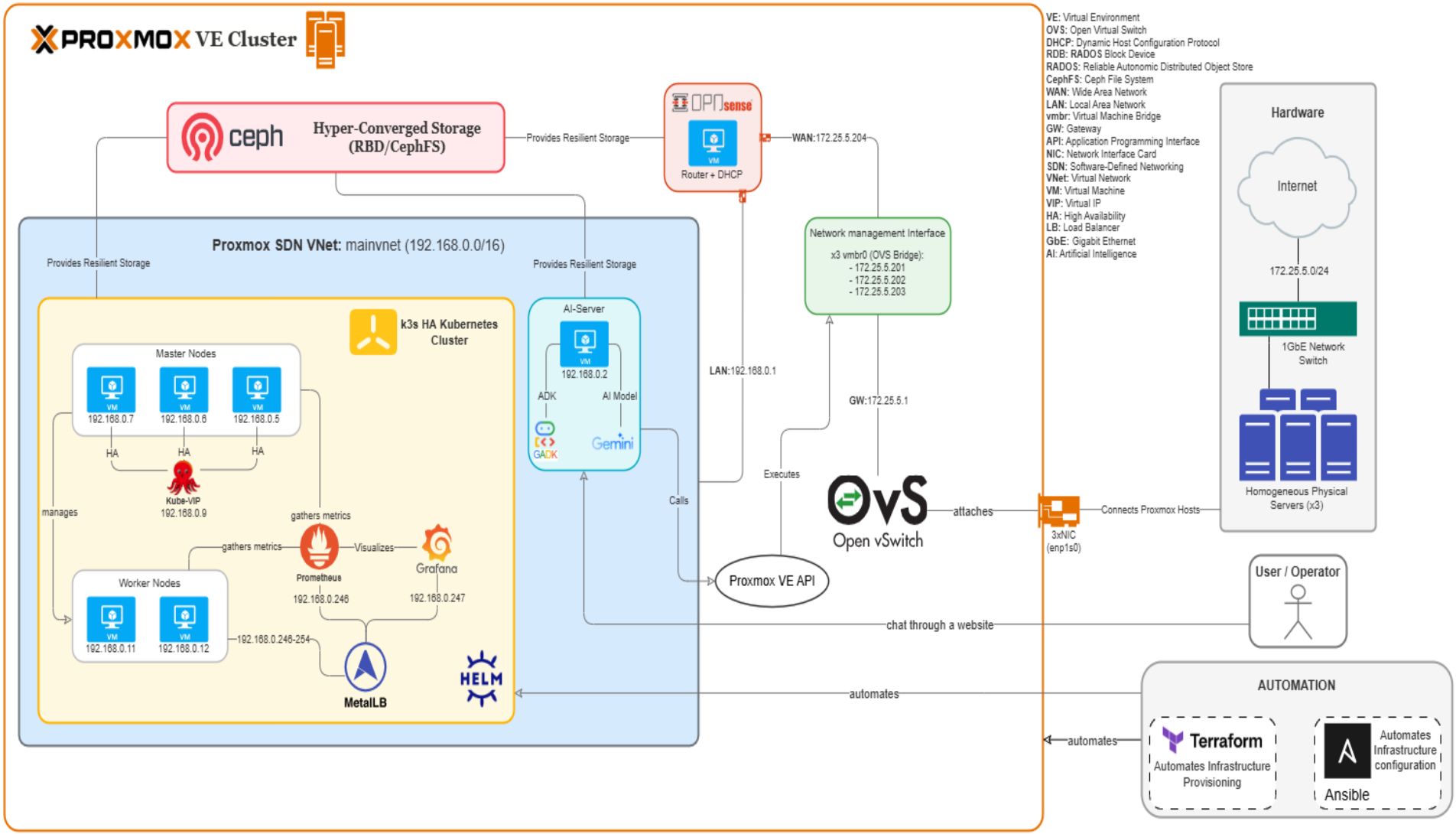

Figure C.1: A detailed diagram of the final implemented system

Description of the Global Implementation Architecture The diagram showcases the tightly integrated, hyper-converged infrastructure built for this research:

- **Physical and Virtualization Layer:** At the base, three **Homogeneous Physical Servers** provide the compute, storage, and networking resources. On top of this hardware, a **Proxmox VE Cluster** is deployed. The hyper-converged nature is achieved through **Ceph**, which provides resilient block (RBD) and file (CephFS) storage directly from the servers' own disks, eliminating the need for a separate storage appliance.
- **Networking Layer:** The virtual network is managed by two key components. **Open vSwitch (OVS)** serves as the software-defined switching fabric within Proxmox. A virtualized **OPNsense** appliance acts as the primary router, firewall, and DHCP server, connecting the internal, isolated **Proxmox SDN VNet** (mainvnet, 192.168.0.0/16) to the external network.
- **Container Orchestration Layer:** Running within the mainvnet are the VMs that host the **k3s HA Kubernetes Cluster**. This includes three master nodes for high availability and multiple worker nodes. Application services are exposed via **MetalLB**, which provides 'LoadBalancer' functionality, while application packaging and deployment are managed by **Helm**.
- **Monitoring and Automation:** Observability is provided by **Prometheus** for metrics collection and **Grafana** for visualization, which gather metrics from the Kubernetes cluster. The entire infrastructure is automated using an Infrastructure as Code (IaC) approach, with **Terraform** handling the provisioning of virtual machines and **Ansible** managing their configuration and the deployment of k3s.
- **Agentic AI Layer:** The capstone of the architecture is the **AI-Server**, a dedicated VM hosting the Multi-Agent System. This system is built with the Google **Agent Development Kit (ADK)** and powered by the **Gemini** AI model. It interacts directly with the **Proxmox VE API** to perform intelligent provisioning tasks.
- **User Interaction:** The primary entry point for the user is a web-based chat interface, which allows them to issue natural language commands to the agentic system.

C.2 Agentic AI Layer Workflow

Figure C.2 details the internal workflow of the Multi-Agent System (MAS) that was implemented to handle intelligent Proxmox provisioning. It illustrates the sequence of operations and the collaboration between specialized agents, from receiving the initial user prompt to delivering the final, refined infrastructure manifest.

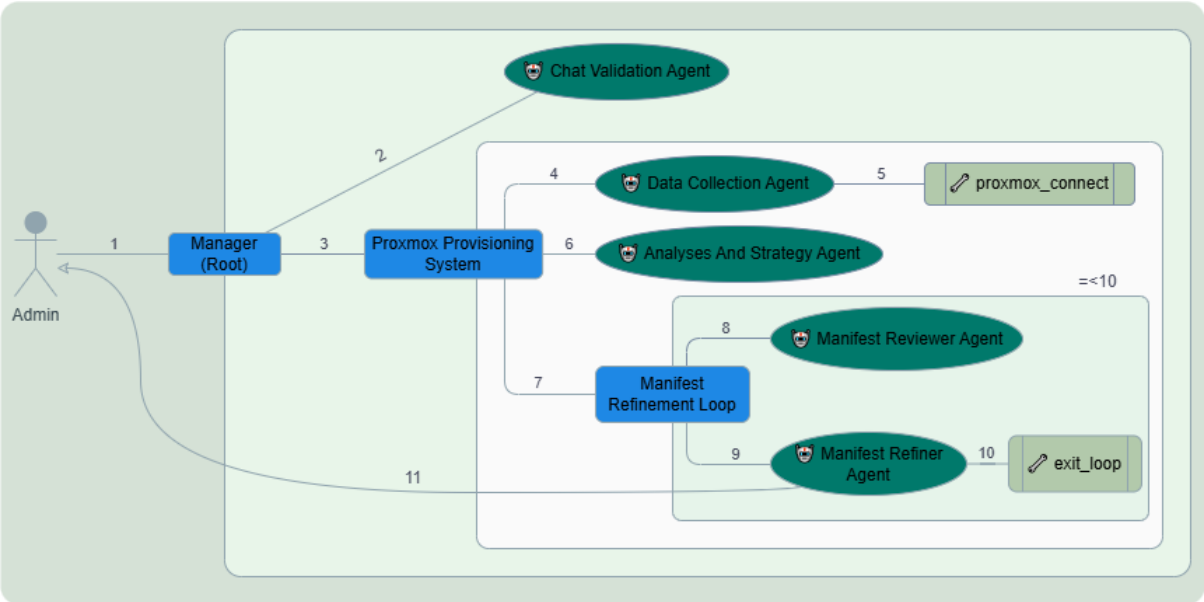

Figure C.2: The step-by-step workflow of the MAS for Proxmox provisioning.

Description of the Agentic AI Layer Workflow The workflow translates a high-level user request into a concrete, deployable artifact through a sequence of coordinated agent actions:

1. **Initiation:** The **Admin** (user) submits a natural language prompt (e.g., "create a webserver vm") to the **Manager (Root)** agent, which serves as the primary orchestrator.

2. **Validation and Clarification:** The Manager delegates the initial interaction to the **Chat Validation Agent**. This agent engages the user in a dialogue to clarify ambiguities and gather necessary details, as demonstrated in Chapter 4.

3. **System Invocation:** Once the intent is clarified, the Manager formally initiates the **Proxmox Provisioning System**, a specialized subsystem dedicated to the task.

4. **Data Collection:** The **Data Collection Agent** is activated. It uses a tool (`proxmox_connect`) to interface with the Proxmox API, gathering real-time data

about the cluster's state, such as available resources, existing VMs, and storage pools.

5. **Strategy Formulation:** The **Analyses And Strategy Agent** receives the user's validated goal and the live data from the environment. It synthesizes this information to create a high-level deployment plan, deciding on parameters like node placement, resource allocation, and network configuration.

6. **Iterative Refinement Loop:** The core of the generation process is the **Manifest Refinement Loop**. This loop embodies a "Reasoning and Acting" (ReAct) pattern:

   - The **Manifest Reviewer Agent** examines the initial draft of the Terraform configuration file, identifying any missing information, logical errors, or anti-patterns.
   - The **Manifest Refiner Agent** takes the feedback from the reviewer and works to correct the manifest. This may involve querying documentation, applying best practices, or using its own reasoning to generate the missing configuration code.
   - This loop can iterate multiple times (up to a predefined limit of 10, as indicated by `=<10`) until the manifest is deemed complete and correct.

7. **Completion and Delivery:** Once the refinement loop concludes (`exit_loop`), the final, validated Terraform manifest is returned to the **Admin**. This "human-in-the-loop" step ensures safety, allowing the operator to review the code before applying it to the infrastructure.

This structured, collaborative workflow enables the system to reliably and intelligently bridge the gap between high-level human intent and low-level infrastructure execution.