Politecnico di Torino

Master's Degree in Computer Engineering

A.y. 2024/2025

Graduation Session December 2025

Automating Deployment and Operation of a Scalable Bare-Metal Kubernetes Cluster

Supervisors:

Prof. Fulvio RISSO

MSc Attilio OLIVA

PhD Stefano GALANTINO

Candidate:

Ilkhom ABDUSAMATOV

Abstract

The growing reliance on cloud-native technologies has made Kubernetes a cornerstone of modern software infrastructures. However, deploying and managing Kubernetes clusters directly in bare-metal environments still poses significant challenges. The lack of native provisioning mechanisms, the diversity of hardware configurations, and the need to coordinate manually upgrades and scaling operations make these environments difficult to maintain. For organizations and research institutions that depend on on-premise resources — whether for performance, security, or data sovereignty reasons — achieving the same level of automation and resilience found in public clouds remains an open problem.

This thesis proposes an automated and extensible framework that brings cloud-grade manageability to bare-metal Kubernetes clusters. The framework follows a declarative and reproducible design that integrates the traditionally separate infrastructure and orchestration layers into a single, coherent workflow. Metal3 handles bare-metal provisioning, while KubeSpray leverages Ansible to automate cluster deployment and lifecycle operations such as scaling and upgrading. The networking layer is enhanced through the adoption of Cilium, providing advanced observability and ensuring efficient communication across workloads. At the operational level, Argo CD enables GitOps-based continuous delivery, ensuring version-controlled deployments and consistent synchronization of environments.

The resulting system demonstrates a scalable and maintainable approach to operating on-premise Kubernetes infrastructures while greatly reducing the need for manual intervention. It shows that automation principles typically associated with cloud providers can be effectively reproduced in self-managed settings, substantially decreasing administrative complexity while preserving flexibility and performance. The framework also proves its extensibility by enabling the seamless integration of additional functionalities, exemplified by the incorporation of GPU workload management, which highlights its adaptability to evolving computational requirements.

Acknowledgements

I would like to express my deepest gratitude to Professor Fulvio Risso for giving me the opportunity to undertake this work and for the invaluable knowledge he has shared throughout my studies.

My heartfelt thanks go to Federico Cucinella for guiding me into the world of cloud technologies and for his constant and immeasurable support.

This thesis would not have been possible without the continuous assistance and insightful advice of Attilio Oliva, Stefano Galantino, and Vincenzo Cosi, to whom I am sincerely grateful.

I also wish to thank all members of the Netgroup Research Group for their warm hospitality and for welcoming me into their community.

I am profoundly grateful to my friends, who accompanied me throughout this journey.

Finally, I wish to express my deepest appreciation to my parents and my family for their unconditional support and unwavering faith in me.

Table of Contents

Chapter 1

Introduction

Over the past decade, cloud-native technologies have reshaped the design and operation of modern computing systems. Practices centered on containerization, microservice architectures, and declarative configuration have enabled organizations to build scalable and resilient services with a high degree of automation. Kubernetes has emerged as the standard platform underpinning these developments, providing mechanisms for orchestrating containerized workloads and abstracting infrastructure complexity across a wide range of deployment environments.

Whereas cloud providers offer managed Kubernetes services with extensive automation and well-integrated operational tooling, deploying Kubernetes on self-managed bare-metal infrastructure presents substantial challenges. Operators must provision physical machines, configure networking, standardize system settings, and maintain cluster components without the automated support typically available in cloud environments. Furthermore, lifecycle tasks such as upgrades, scaling operations, and configuration changes — commonly referred to as Day-2 operations - often receive comparatively less attention during initial cluster design, especially in academic or research environments where long-term maintenance is not always the primary focus. As a result, clusters tend to accumulate operational complexity over time, making them harder to evolve or extend.

The CrownLabs platform at the Politecnico di Torino exemplifies this context. CrownLabs is an on-premise Kubernetes-based solution designed to provide students and researchers with remotely accessible virtualized environments. Its architecture has proven to be effective for educational purposes; however, the underlying Kubernetes cluster showed several areas where operational processes could be streamlined or modernized. Upgrades required substantial manual intervention, high-availability features were limited, and extensibility — such as the integration of GPU-accelerated workloads — was challenging within the existing setup. While these limitations did not undermine the functionality of the platform, they highlighted the need for a more automated and maintainable cluster foundation.

This thesis addresses these challenges by designing and implementing an automated, scalable, and reproducible bare-metal Kubernetes environment tailored to the operational needs of CrownLabs but generalizable to similar on-premise clusters. The work draws on established open-source technologies — such as Metal3 for bare-metal provisioning, Kubespray for cluster automation, and Argo CD for GitOps-based application delivery — to construct an infrastructure that reduces manual overhead, improves consistency, and supports the long-term evolution of the platform.

1.1 Goal

The objective of this thesis is to design and implement an automated bare-metal Kubernetes cluster that addresses the operational limitations encountered in the existing CrownLabs infrastructure. The new environment aims to:

- provide a reproducible and automated deployment workflow for bare-metal servers,
- support a scalable and maintainable cluster architecture with improved operational consistency,
- streamline Day-2 operations such as upgrades, configuration changes, and component lifecycle management,
- adopt a GitOps-based approach to ensure reliable, traceable, and declarative management of cluster components and applications.

These goals reflect a broader intent to improve the long-term sustainability of CrownLabs and to demonstrate how cloud-inspired operational practices can be applied to on-premise Kubernetes environments.

1.2 Structure of this thesis

This thesis is structured as follows:

- Chapter 2 – Background introduces the fundamental concepts underlying this work, including Kubernetes architecture, bare-metal infrastructure provisioning challenges, and an overview of the CrownLabs platform.
- Chapter 3 – Design presents the architectural decisions that guided the development of the new environment, covering bare-metal provisioning, cluster topology, automation workflows, and GitOps integration.

- Chapter 4 – Implementation describes the practical realization of the design, including deployment automation with Kubespray, GitOps workflows with Argo CD, and support for GPU-accelerated workloads.
- Chapter 5 – Results evaluates the effects of the proposed solution, discusses operational improvements, and reflects on scalability and maintainability of the system.
- Chapter 6 - Conclusion summarizes the contributions of the thesis and identifies areas for future work.

Chapter 2

Background

2.1 Kubernetes Fundamentals

Kubernetes has become the standard container orchestration platform for deploying and managing containerized applications at scale. Originally developed by Google and released open-source in 2014, it abstracts underlying infrastructure and provides a consistent API for managing workloads across diverse environments. This section reviews Kubernetes architecture and operational characteristics essential to understanding the challenges of bare-metal deployment.

2.1.1 Architecture Overview

Kubernetes separates concerns between cluster management and workload execution through two node types: control plane nodes and worker nodes.

The *control plane* maintains cluster state and makes global decisions about scheduling and scaling. Its core components include: the *API server*, which serves as the central REST interface for all cluster interactions; *etcd*, a distributed key-value store holding all cluster state; the *scheduler*, which assigns pods to nodes; and the *controller manager*, which runs reconciliation loops to maintain desired state.

Worker nodes execute containerized workloads through three essential components: the *kubelet*, which communicates with the control plane and manages pod lifecycle; the *container runtime*, which executes containers via the Container Runtime Interface (CRI); and *kube-proxy*, which maintains network rules for service routing.

This separation allows control plane optimization for management tasks while worker nodes focus on application execution, with distinct security and resource policies.

2.1.2 High Availability

Production clusters require control plane high availability to prevent API unavailability from blocking cluster management operations. This is typically achieved by deploying multiple control plane nodes (three or five instances) behind a load balancer, with all components running in active-active mode.

The etcd datastore uses Raft consensus across an odd number of members (typically three or five) to maintain consistency and tolerate $(n-1)/2$ member failures while preserving quorum. For bare-metal deployments, implementing this load balancing layer requires careful planning, adding complexity compared to cloud-managed solutions.

2.1.3 Cluster Upgrades and Version Management

Kubernetes follows a version skew policy: control plane components should run the same version, while worker kubelets can lag up to two minor versions behind the API server. This provides flexibility but requires coordinated upgrades.

Upgrading clusters involves multiple steps with operational risk: control plane components must be upgraded individually while maintaining API availability, followed by coordinated worker node upgrades. Each node upgrade requires draining pods to other nodes, potentially causing service interruptions. Beyond core components, clusters include numerous add-ons (CNI plugins, storage drivers, monitoring tools) with their own compatibility requirements, increasing upgrade complexity and risk. Manual upgrade procedures are time-consuming and error-prone, particularly in large clusters.

2.1.4 Bare-Metal vs Cloud-Managed Kubernetes

Cloud providers (GKE, EKS, AKS) abstract operational complexity: the provider manages control plane deployment, high availability, security patching, and worker node scaling automatically. Upgrades become simple API operations.

Bare-metal deployments require manual management of all lifecycle aspects: provisioning servers, installing operating systems, deploying Kubernetes components, implementing HA mechanisms, and managing networking and storage. Upgrades must coordinate OS, Kubernetes, runtime, and software updates across multiple machines. Scaling requires procuring and provisioning additional hardware.

The choice involves fundamental tradeoffs: bare-metal offers complete control and customization for specific workloads and compliance requirements, with potential cost advantages for existing hardware; cloud services provide operational simplicity and predictable costs but less control and higher per-workload expenses. Certain organizations require on-premise deployment for data sovereignty or regulatory compliance.

Understanding these differences motivates the thesis: bringing cloud-grade automation and operational practices to bare-metal Kubernetes deployments, thereby reducing the operational gap between deployment models.

2.2 Bare-Metal Provisioning

While cloud providers offer APIs for provisioning virtual machines and managing infrastructure programmatically, bare-metal servers lack such standardized interfaces. Deploying Kubernetes on physical hardware requires addressing the fundamental challenge of server lifecycle management: how to provision, configure, and maintain physical machines in a repeatable and automated manner. This section examines the most important challenges of bare-metal infrastructure and the role of out-of-band management.

2.2.1 Challenges of Bare-Metal Infrastructure

Bare-metal operations lack many cloud conveniences: there is no unified provisioning API, OS installs and low-level configuration are often manual or semi-automated (e.g., PXE), and hardware varies across generations and vendors. These realities make repeatable provisioning, fast scaling, and consistent configuration harder to achieve. Common pain points are:

- Absence of a single programmatic interface for creating and preparing servers, which complicates applying infrastructure-as-code practices.
- Per-server OS and firmware setup (network, partitions, drivers), which is time-consuming and sensitive to small differences that cause drift.
- Hardware heterogeneity (CPU, memory, storage, firmware) that requires flexible workflows and extra testing.
- Integration overhead for vendor-specific management interfaces and tooling, which raises automation and security effort.

In practice, PXE and scripted installers reduce manual steps, but they do not eliminate the need for reliable remote management, inventory, and health instrumentation to build truly repeatable, scalable bare-metal fleets.

2.2.2 Out-of-Band Management

Out-of-band management provides the control plane needed to automate physical servers independently of their OS. Modern servers expose these capabilities via a

BMC (Baseboard Management Controller) and vendor interfaces; the most relevant aspects are:

- IPMI: a long-standing standard for power control, console access, and sensor reading. Widely available but dated — implementations and security vary by vendor.
- Redfish: a newer, RESTful standard (JSON/HTTP) designed for automation and better security. Increasingly the preferred interface for programmatic hardware management.
- Typical BMC features used in automation: remote power on/off/reset, serial or KVM console access, one-time boot selection or virtual media for OS installation, and telemetry (temperature, fans, power).

These interfaces enable automated provisioning workflows (powering a node, forcing PXE boot, installing an image, and inspecting hardware). However, differences in vendor behavior, firmware quirks, and security configurations (credentials, network isolation) remain practical obstacles to fully hands-off bare-metal automation.

2.3 GitOps and Operational Automation

Once a Kubernetes cluster is running, operational consistency and safe change management become primary concerns. GitOps is a practical model that brings version control, review, and continuous reconciliation to both applications and cluster components. The following subsections summarize the core principles and how they apply to Kubernetes operations, including Day-2 tasks such as upgrades and configuration changes.

2.3.1 GitOps Principles

GitOps treats Git as the canonical, auditable source of desired state. Operators express cluster and application configuration as declarative manifests stored in repositories; an automated agent continuously reconciles the live cluster to match that repository.

Key ideas:

- *Declarative source of truth*: All desired state (apps, infrastructure, and cluster components) is captured in Git. This makes intent explicit and reversible.
- *Automated reconciliation*: A controller in the cluster pulls repository state and applies changes, detecting and correcting drift rather than relying on ad-hoc pushes.

- *Review and audit*: Changes go through the normal code review process (PRs), providing peer review, CI validation, and a timestamped history of who changed what and why.
- *Safe rollbacks and reproducibility*: Reverting to a previous commit restores the prior desired state, enabling fast, reliable rollbacks and reproducible environments for recovery or testing.
- *Day-2 operations as code*: Routine operational tasks—upgrades, configuration updates, policy changes—are expressed and validated in Git. This reduces manual steps, documents the intent, and enables coordinated rollouts across components.

Practically, GitOps encourages separating concerns: CI builds and publishes artifacts (images, charts), while the GitOps repository declares which artifacts and configuration should run. The cluster-side agent requires appropriate access, but credentials remain inside the cluster, improving security compared with externally pushing changes.

2.3.2 Continuous Delivery for Kubernetes

Continuous delivery for Kubernetes is naturally pull-based in a GitOps model. The pipeline commonly splits responsibilities:

- *CI*: Builds and tests code, produces immutable artifacts (container images), and optionally updates a staged manifest or a metadata file (e.g., image tag) in a Git branch.
- *CD (pull-based)*: An in-cluster reconciler monitors Git and applies manifests to reach the declared state; it performs health checks, reports drift, and can be configured to require manual approval for risky changes.

Benefits of the pull model:

- Reduces distribution of cluster credentials — the reconciler holds credentials inside the cluster.
- Improves reliability — if the reconciler restarts it will re-sync from Git and restore intended state.
- Enables consistent multi-cluster deployments — multiple clusters can sync from the same repository or branch structure to ensure consistent configuration across environments.

Coordination patterns for complex changes:

- *Atomic change bundles*: Use a single PR to update multiple related manifests so changes roll out together.
- *Phased rollout and health gates*: Split changes into waves and require health checks before proceeding to the next phase to avoid cascading failures.
- *Mixing automated and manual approvals*: Configure automated sync for low-risk updates and manual approval for disruptive operations (major upgrades, schema changes).

2.4 CrownLabs Platform

2.4.1 Purpose and Context

CrownLabs is an on-premise, Kubernetes-based platform developed at Politecnico di Torino to provide remote access to preconfigured laboratory environments for teaching and research. Originally created in 2020 to support remote education during the COVID-19 emergency, the platform has since become a persistent service used by students and researchers to access course labs, configure development sandboxes, and create testbeds. Its design prioritizes reproducible, isolated environments and easy provisioning of course-specific setups.

2.4.2 Existing Infrastructure

The initial CrownLabs deployment was on a best-efforts basis. The cluster was bootstrapped with kubeadm and ran a single control-plane instance as a VM, with six physical worker nodes providing user compute. Networking relied on Calico configured for direct routing and MetalLB for service IP advertisement via BGP. This composition enabled rapid delivery of lab environments but left many operational responsibilities — provisioning, upgrades, and availability — to manual processes and ad-hoc procedures.

Single Control Plane Configuration

The initial deployment used a single control plane node running as a virtual machine on one of the physical worker nodes. This configuration simplified initial setup and reduced resource overhead but introduced availability risks. During control plane upgrades or maintenance, the Kubernetes API server would be temporarily unavailable, preventing management operations and potentially affecting workload scheduling. While running workloads remained operational during control plane outages, any pods that needed to be rescheduled or any new instances that users tried to create would fail until the control plane recovered.

This single point of failure was a known limitation but represented a practical tradeoff for a system developed quickly under resource constraints. As the platform grew in importance and user base, improving control plane resilience became a priority.

Networking Configuration

The cluster used Calico as its Container Network Interface (CNI) plugin. Calico was configured to provide pod networking without overlay encapsulation. This direct routing approach offers excellent performance and simplicity when all nodes exist on the same layer-2 network segment.

For external service exposure and load balancing, the cluster deployed MetalLB, which implemented BGP-based load balancer services. MetalLB announced service external IPs through BGP, integrating with the existing network infrastructure to make services accessible from outside the cluster.

Physical Infrastructure

The cluster consisted of six physical worker nodes, providing the compute resources for running user workloads and cluster infrastructure components. These nodes, along with the control plane VM, had served the platform well during its initial years of operation.

Chapter 3

Design

3.1 Requirements Analysis

3.1.1 Identified Limitations

As CrownLabs usage expanded and new pedagogical needs emerged, several structural limitations of the existing Kubernetes infrastructure became increasingly evident. These issues spanned resource management, operational procedures, networking capabilities, and support for emerging workload types. The most significant limitations can be summarised as follows:

- **Resource constraints and limited scalability:** Peak-period CPU utilisation regularly approached cluster capacity, and the absence of automated provisioning prevented rapid expansion of compute resources.
- **Complex and disruptive upgrade procedures:** Manual Kubernetes upgrades introduced prolonged maintenance windows, API downtime, and elevated operational risk due to the lack of automated or validated upgrade workflows.
- **Insufficient high availability:** A single control plane constituted a critical single point of failure, compromising cluster operability during maintenance or unplanned outages.
- **Networking restrictions across subnets:** The existing Calico configuration did not support seamless cross-subnet communication, preventing integration of additional servers located in other datacenter segments.
- **Limited support for new capabilities:** The introduction of GPU-accelerated workloads required cluster-wide reconfiguration, which the current architecture could not accommodate safely or efficiently.

Collectively, these limitations demonstrated that the original architecture had reached the boundaries of its scalability and operational maintainability. To sustain CrownLabs' continued growth, a re-design was necessary to establish a more automated, resilient, and extensible foundation.

3.1.2 Requirements

The requirements for the new cluster were derived directly from the limitations identified above and the operational needs of CrownLabs. The redesigned infrastructure must support automated provisioning of bare-metal servers while minimising manual intervention, thereby enabling rapid expansion during periods of increased demand. The deployment process must be reproducible and fully declarative to ensure consistency across environments. To remove the single point of failure inherent in the previous architecture, a highly available control plane is essential, along with the ability to scale the cluster seamlessly by adding or removing nodes without service disruption.

Given the operational challenges associated with previous upgrade procedures, the new design must provide streamlined, low-risk upgrade mechanisms for both Kubernetes and its supporting components. Furthermore, the networking architecture must enable cross-subnet communication so that resources distributed across multiple datacenter segments can participate uniformly in the cluster. The infrastructure must also support GPU-accelerated workloads, reflecting the evolving academic use cases in machine learning and compute-intensive applications.

Operational management should transition toward a GitOps-based workflow, ensuring version-controlled configuration, automated reconciliation, and comprehensive auditability of changes. These requirements collectively define the foundation upon which technology choices and architectural decisions in subsequent chapters are built.

3.2 Architecture Overview

The architectural design of the new provisioning and management framework is structured around a three-layer approach that separates the concerns of infrastructure preparation, Kubernetes cluster creation, and operational management. This model provides a conceptual foundation that remains valid regardless of the specific technologies ultimately selected during the design process.

3.2.1 Three-Layer Approach

The first layer focuses on preparing and configuring the underlying physical machines that will form the cluster. At the time of designing the framework, different strategies were under evaluation. One option involved integrating infrastructure provisioning directly into a unified control plane, while another treated provisioning as an independent stage preceding cluster creation. Both approaches shared a common goal: abstracting low-level interactions with server hardware and ensuring that compute nodes are delivered in a consistent and predictable state prior to cluster deployment.

The second layer encompasses the deployment and lifecycle management of the Kubernetes cluster itself. Various tooling options were considered, some offering tight integration with the provisioning layer and others operating independently on an already prepared set of machines. Regardless of the eventual choice, this layer is responsible for transforming provisioned hosts into a functional cluster and for facilitating ongoing lifecycle tasks such as scaling or upgrading. While the overarching design emphasises declarative specifications wherever feasible, it also recognises that certain stages of cluster deployment—particularly those relying on procedural automation—combine declarative intent with imperative execution.

The third layer introduces operational automation through a Git-driven workflow. Here, configuration and application state are expressed declaratively and reconciled automatically by a controller operating within the cluster. Although specific tooling had not yet been formally introduced at this point in the design, the decision to adopt a Git-centric operational model had already been made. This layer ensures consistent, traceable, and version-controlled management of cluster resources and system components.

3.2.2 Design Philosophy

Across all layers, the architecture is guided by the principle of explicit desired-state definition. Wherever possible, configuration is expressed declaratively so that automated systems may reconcile actual state with the specified target. This approach reduces the need for manual intervention and enhances both reproducibility and operational safety.

Central to this philosophy is the use of Git as the authoritative source of truth. All modifications to cluster configuration flow through version control, enabling structured review processes, auditability, and straightforward rollback.

Another defining principle is reproducibility. The architecture aims to allow the entire environment — from machine provisioning through application deployment — to be reconstructed deterministically from version-controlled specifications. This facilitates disaster recovery, reduces configuration drift, and enables safe experimentation and testing in isolated environments.

3.2.3 Component Interaction

The interaction between the architectural layers follows a conceptual sequence. First, the infrastructure layer prepares the nodes that will participate in the cluster, regardless of whether this preparation is tightly integrated with the cluster management system or performed as a separate step. The cluster lifecycle layer then consumes these nodes to create and maintain a Kubernetes cluster, establishing both control plane and worker roles as required. Once the cluster is operational, the operational automation layer assumes responsibility for managing system components and applications through declarative definitions stored in version control.

While each layer builds upon the outputs of the previous one, the architecture intentionally preserves clear boundaries between them. This separation enables flexibility in selecting or replacing individual components without undermining the overall design principles or workflow.

3.3 Bare-Metal Provisioning Layer

The bare-metal provisioning layer constitutes the foundation upon which the entire cluster architecture is built. Its purpose is to prepare physical servers for inclusion in the Kubernetes environment by abstracting heterogeneous hardware characteristics, enforcing a consistent baseline configuration, and supplying the cluster-deployment layer with machines that are fully initialized and ready for orchestration. Through this layer, raw infrastructure is transformed into predictable and reproducible compute nodes, ensuring that subsequent automation stages can operate reliably and without manual intervention.

During the design phase, several tooling options for implementing this provisioning layer were evaluated, with MAAS and Metal3 emerging as the two most viable candidates. Both solutions support large-scale management of bare-metal servers, but they differ significantly in their architectural integration models and operational paradigms. The final design adopts Metal3 for two primary reasons. First, Metal3 offers native compatibility with the Cluster API (CAPI), which at the time was being investigated as a promising candidate for the cluster lifecycle management layer due to its declarative and extensible design. Although CAPI was ultimately not selected, its strong suitability during early evaluations made Metal3 a strategically aligned choice. Second, a Metal3-based environment was already available within the university through a research group that had deployed and validated the technology in a production-adjacent context. This prior institutional experience provided both practical insights and operational confidence, further motivating the decision.

The remainder of this section presents the architecture, workflow, and operational role of Metal3 within the provisioning layer, and explains how it integrates with the overall automation pipeline established for the cluster.

3.3.1 Metal3 Overview

Metal3 integrates bare-metal provisioning directly into the Kubernetes control plane by exposing physical servers as custom resources. This Kubernetes-native approach allows operators to manage heterogeneous hardware using declarative workflows that align with standard cluster operations. Instead of maintaining separate provisioning systems, Metal3 enables bare-metal machines to participate in the same reconciliation and version-controlled processes that govern other infrastructure components.

At the center of Metal3 is its reliance on OpenStack Ironic, a mature and feature-complete provisioning service responsible for low-level hardware management. Metal3 acts as an abstraction layer that translates Kubernetes resource specifications into Ironic operations, combining Ironic's hardware support with Kubernetes' declarative configuration model.

Provisioning Workflow

The primary resource introduced by Metal3 is the `BareMetalHost`, which defines the metadata, power-management configuration, and provisioning parameters for a single physical server. When such a resource is created, the Metal3 controllers register the host with Ironic, perform optional hardware inspection, and initiate the provisioning workflow. Throughout the process, the resource's status reflects progress and potential errors, making the entire lifecycle observable through standard Kubernetes tools.

Provisioning typically proceeds through registration, inspection, OS image deployment, configuration via mechanisms such as cloud-init, and a final transition to a ready state. At that point, the server becomes suitable for integration into a Kubernetes cluster or assignment to other roles. Deprovisioning reverses this process, wiping the machine and returning it to an available state.

The Metal3 architecture consists of three main components: the Baremetal Operator, which reconciles `BareMetalHost` resources; the Ironic services, which perform the actual provisioning actions; and the Ironic Python Agent, which executes image deployment tasks on the server itself. Together, these components provide a streamlined, reproducible, and Kubernetes-aligned mechanism for managing bare-metal infrastructure.

3.3.2 Metal3 Management Cluster

The Metal3 setup is illustrated in figure 3.1. It consists of a dedicated Proxmox virtual machine that hosts a minimal single-node Kubernetes cluster. This management cluster runs the essential Metal3 components: the baremetal-operator controller manager, which reconciles BareMetalHost resources, and the Ironic deployment, which performs the actual provisioning operations. The setup includes a custom Ironic Python Agent (IPA) developed by the research group to address specific provisioning requirements. An nginx web server running on the same VM serves as the image service, hosting operating system images that will be deployed to bare-metal servers.

3.4 Cluster Deployment and Lifecycle

The cluster deployment layer is responsible for transforming provisioned bare-metal servers into a functional, highly available Kubernetes cluster and for managing its lifecycle through upgrades, scaling operations, and configuration changes. In this section, two alternative solutions for implementing this layer — Kubespray and Cluster API — are examined. Each is evaluated in terms of its architectural model, operational workflow, and suitability for integration within the overall system design.

3.4.1 Kubespray

Kubespray is an Ansible-based framework for deploying and managing production-grade Kubernetes clusters. It combines Ansible's mature configuration-management capabilities with a modular and opinionated collection of playbooks that automate all stages of cluster installation and lifecycle control.

Ansible-Based Deployment Model

Kubespray builds on Ansible's agentless automation model, requiring only SSH access and a Python environment on target machines. Its playbooks, written in YAML, express desired system state declaratively while leveraging Ansible's idempotency, dependency handling, and parallel execution. Through a structured set of roles, Kubespray automates the installation of container runtimes, system prerequisites, etcd, control-plane components, worker-node services, and essential add-ons. The modular organisation of these roles enables selective customisation while maintaining well-tested defaults.

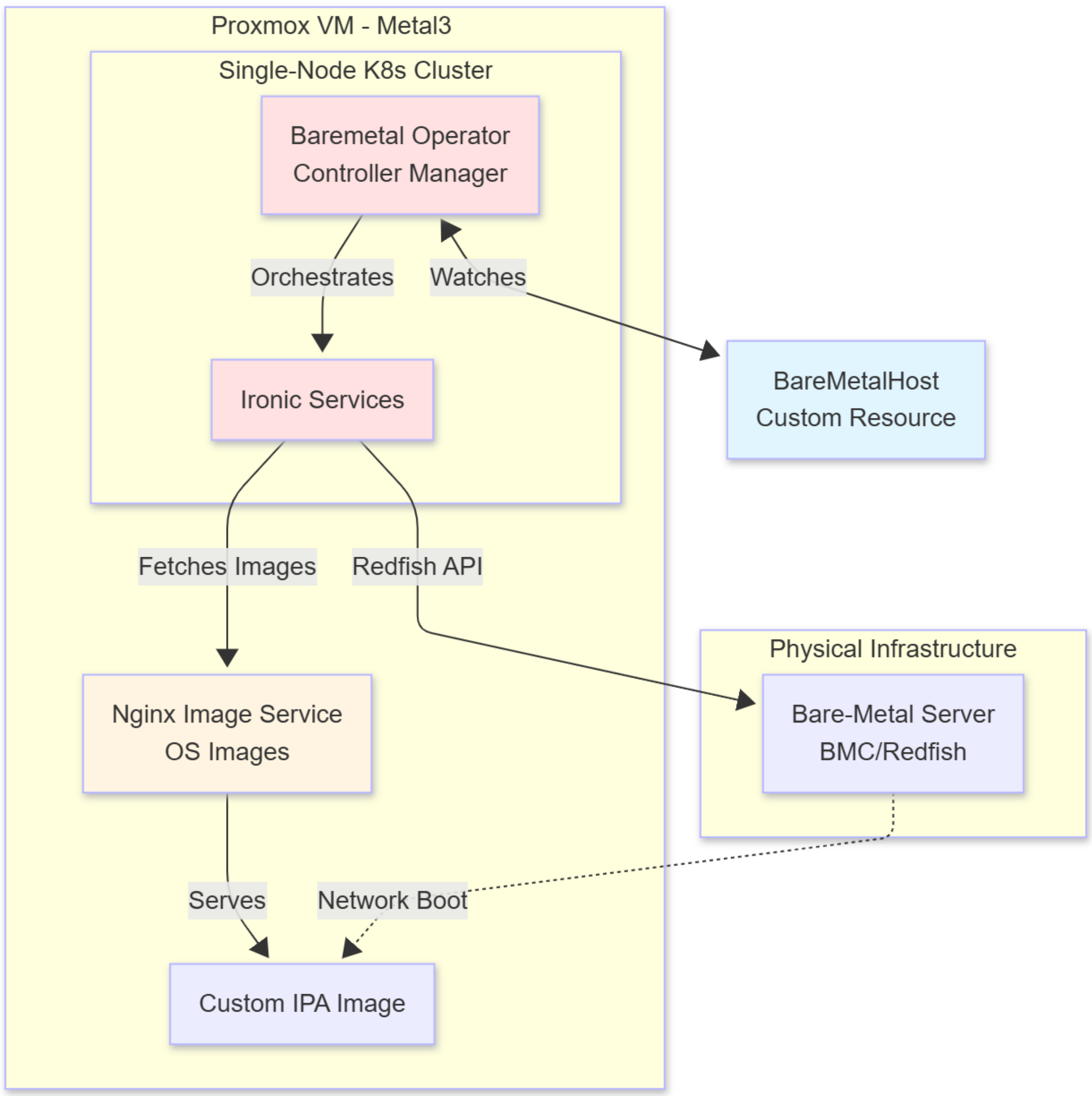


Figure 3.1: Metal3 Management Plane Architecture

Declarative Configuration through Inventory

Cluster configuration in Kubespray is defined through Ansible inventory files, which specify node roles (control-plane, worker, etcd) and encode key parameters such as Kubernetes version, CNI selection, network CIDRs, and feature toggles. The inventory thereby serves as a version-controlled declaration of cluster topology and configuration intent. Applying configuration changes consists of modifying inventory variables and re-executing the relevant playbooks, allowing Kubespray to converge the system toward the updated desired state.

Network Flexibility and CNI Support

Kubespray supports a wide set of CNI plugins, including Calico, Cilium, Flannel, and Weave, configurable directly from inventory variables. Operators may customise pod and service CIDR ranges, DNS backend options, and other network characteristics, enabling deployments that adapt to diverse network environments—from simple flat L2 networks to more complex segmented topologies.

Lifecycle Operations

Beyond initial provisioning, Kubespray provides robust Day-2 operational support:

- *Scaling*: Adding or removing nodes is achieved by editing the inventory and running the corresponding scale or remove-node playbook.
- *Upgrades*: Kubernetes version upgrades follow a controlled workflow defined in dedicated upgrade playbooks, respecting Kubernetes version-skew policies and orchestrating control-plane and worker upgrades safely.
- *Node replacement*: Failed or retired nodes can be cleanly removed and substituted with freshly provisioned hosts using standard playbooks.

These capabilities provide repeatable, validated procedures for common cluster maintenance tasks.

Execution Workflow

Operationally, Kubespray follows a standard Ansible workflow: an operator invokes `ansible-playbook` with a chosen inventory and playbook, after which Ansible connects to each target host via SSH, collects system facts, and executes the necessary tasks to converge the node to the declared state. Tasks include package installation, configuration file deployment, service management, and component orchestration. Thanks to Ansible's idempotent semantics, these playbooks can be rerun safely, supporting verification and iterative configuration without risk of unintended changes.

3.4.2 Cluster API

Cluster API (CAPI) adopts a fundamentally Kubernetes-native model for cluster lifecycle management, shifting the entire process of creating, upgrading, and scaling Kubernetes clusters into the Kubernetes control plane itself. Rather than invoking external automation tools, Cluster API introduces a set of custom resources whose reconciliation is handled by controllers running in a dedicated management cluster.

Kubernetes-Native Lifecycle Management

CAPI extends the Kubernetes API with resources that describe complete clusters. A management cluster hosts the Cluster API controllers, which watch these resources and ensure that corresponding workload clusters match the declared specifications. Operators interact with clusters through standard Kubernetes mechanisms — `kubectl` and declarative YAML manifests — treating clusters as first-class API objects analogous to deployments or stateful sets. This enables uniform GitOps workflows, version-controlled state, and API-driven operations.

Reconciliation and Controller Logic

Cluster API follows the Kubernetes operator pattern: reconciliation loops continuously compare desired state, expressed in custom resources, with the actual state of workload clusters. The controllers provision nodes, adjust cluster size, replace failed machines, and orchestrate version upgrades as required. This continuous reconciliation provides robustness against failures, ensures automatic convergence toward the declared topology, and yields fine-grained observability through events and status fields. Standard Kubernetes RBAC and admission control apply directly to cluster resources, enabling consistent governance.

Infrastructure Providers and Machine Abstractions

A key design feature of CAPI is its provider abstraction. Infrastructure providers implement platform-specific logic for provisioning compute resources across a broad range of environments, including major clouds (AWS, Azure, GCP), virtualization platforms (vSphere, OpenStack), and bare-metal systems through Metal3. The `Machine` custom resource serves as a platform-agnostic representation of a single node, while higher-level abstractions such as `MachineDeployment` and `MachineSet` provide rolling-update semantics analogous to Kubernetes workload controllers. This layered architecture cleanly separates infrastructure concerns from cluster-level orchestration.

Declarative Cluster Composition

Cluster API defines cluster state entirely through custom resource definitions:

- *Cluster*: declares cluster-wide properties such as networking configuration and control-plane endpoint.
- *Machine*: represents an individual node, referencing an infrastructure-specific template and desired Kubernetes version.

- *MachineDeployment* and *MachineSet*: manage groups of machines with declarative rolling updates.
- *KubeadmControlPlane*: manages control-plane nodes, including HA configuration and the orchestration of upgrades.

Together, these resources compose full cluster specifications in a declarative, version-controlled form. Updating cluster configuration consists simply of modifying resource manifests, while the controllers handle the operational details required to enact the new desired state.

3.4.3 Technology Selection: KubeSpray vs Cluster API

The choice of cluster deployment technology was a critical design decision. A systematic comparison was conducted across multiple dimensions, listed in table 3.1.

Table 3.1: Properties Comparison

Features	**CAPI**	**Kubespray**
Metal^3 Integration	✓	X
API-based Cluster Lifecycle Management	✓	X
Scalability and Extensibility	✓	✓
Management Cluster Independence	X	✓
Granular Control over Components	X	✓
Multi-Platform Flexibility	X	✓
In-place Upgrades for Bare Metal	X	✓

Metal3 Integration: Cluster API offers native integration with Metal3 through the Cluster API Provider for Metal3 (CAPM3). This integration allows clusters to be managed entirely through Kubernetes custom resources, providing a unified operational model. KubeSpray, in contrast, operates independently of Metal3, treating provisioned servers as generic hosts accessible via SSH.

API-Based Cluster Lifecycle Management: Cluster API provides declarative cluster management through Kubernetes APIs. Clusters are represented as custom resources, and controllers continuously reconcile desired state with actual state. This Kubernetes-native approach offers elegant conceptual consistency and enables programmatic cluster management. KubeSpray uses an imperative execution model: operators explicitly invoke Ansible playbooks to perform actions, though these playbooks operate on declarative inventory.

Management Cluster Requirement: Cluster API requires a separate management cluster to host the CAPI controllers that manage workload clusters. This management cluster must remain available and operational for cluster lifecycle operations to function. KubeSpray has no such dependency: it runs from an operator's workstation or automation server and requires no persistent management infrastructure. For a small-scale deployment, this additional management overhead represents a significant consideration.

Granular Component Control: KubeSpray provides extensive configurability through Ansible variables, allowing fine-grained control over Kubernetes component flags, configuration file contents, and deployment procedures. The declarative inventory can specify detailed settings for networking, security policies, feature gates, and numerous other parameters. Cluster API, while configurable, provides a more opinionated and abstracted interface. Its focus on standardization and portability means less flexibility in customizing low-level component behavior.

Multi-Platform Flexibility: KubeSpray is platform-agnostic: as long as target hosts are accessible via SSH and meet basic requirements (supported OS, Python installation), KubeSpray can deploy Kubernetes. Mixing virtual machines and bare-metal servers in the same cluster is straightforward. Cluster API's provider model expects homogeneous infrastructure within a cluster: all nodes come from the same infrastructure provider (Metal3 for bare-metal, vSphere for VMs, etc.). While technically possible to mix providers, this scenario is not well-supported and introduces significant complexity.

In-Place Upgrade Strategy: KubeSpray performs in-place upgrades: a dedicated Ansible playbook upgrades Kubernetes components on existing nodes without reprovisioning them. Nodes remain operational throughout the upgrade, with control plane components and kubelets updated sequentially. This approach is straightforward and doesn't require additional bare-metal capacity. Cluster API, particularly when integrated with Metal3, follows a different paradigm: upgrades typically involve provisioning new nodes with updated images and replacing old nodes. This immutable infrastructure approach ensures clean upgrades but requires either spare bare-metal capacity or tolerance for node reprovisioning time (deprovision old node, provision new node with updated image, join to cluster). While Cluster API can be configured for in-place upgrades, this deviates from its typical operational model and reduces some of its advantages.

Decision Rationale

Given the specific context of this deployment — a small cluster (under 10 nodes), limited spare bare-metal capacity, and a team with Ansible experience — KubeSpray emerged as the more pragmatic choice. The absence of management cluster overhead reduces operational complexity. The in-place upgrade model aligns better with the constraints of bare-metal provisioning times and available hardware. The granular configurability provides flexibility to tune the cluster precisely to

CrownLabs requirements. The platform-agnostic nature future-proofs against potential architectural changes, such as mixing VM-based control planes with bare-metal workers.

While Cluster API's Kubernetes-native approach offers conceptual elegance, particularly in large-scale or multi-cluster environments, the operational overhead and architectural assumptions were less aligned with the immediate requirements. For a larger deployment with dozens of nodes or multiple clusters, the balance might shift in favor of Cluster API's strengths in standardization and declarative management.

Finalizing the choice of Kubespray unlocked several downstream design decisions that depended on the deployment model. With the cluster management approach established, it became possible to validate assumptions regarding control plane topology, node allocation, and the feasibility of in-place versus rolling upgrade strategies.

3.4.4 High Availability Control Plane

Kubespray's support for heterogeneous, multi-platform cluster configurations made it feasible to decouple the control plane from the bare-metal worker nodes. As a result, the design strategically places the control plane on VMware virtual machines provisioned by the university, ensuring they are hosted on robust and secure infrastructure separate from the bare-metal worker nodes and allowing the physical servers to be dedicated entirely to workload execution.

To eliminate the single point of failure present in the original CrownLabs deployment, the architecture adopts a high-availability control plane with three dedicated control plane nodes. Each node hosts the full suite of Kubernetes control plane components — the API server, controller manager, scheduler, and an etcd instance. Together, the three etcd members form a Raft-based consensus cluster, ensuring data replication and enabling the system to tolerate the loss of one member while preserving quorum and operational continuity.

API server traffic is balanced by an internal HAProxy instance deployed on the worker nodes, which exposes a stable virtual endpoint for kubelet and other system components. HAProxy monitors the health of the API servers and forwards requests only to responsive control plane nodes, thereby maintaining connectivity even in the presence of individual node failures.

This high-availability configuration ensures that the cluster can withstand the loss of a control plane node without service disruption. It also enables planned maintenance to be performed sequentially across the control plane nodes, maintaining uninterrupted cluster availability throughout the procedure.

3.5 Network Architecture Design

The network architecture addresses the primary limitation that hindered the expansion of the original CrownLabs infrastructure: the lack of cross-subnet pod communication. The redesigned networking stack not only resolves this constraint but also modernizes the cluster's networking capabilities to support future growth and operational flexibility.

3.5.1 CNI Plugin Selection: Cilium

Although the legacy infrastructure relied on Calico, the new design adopts Cilium as the cluster's Container Network Interface (CNI). While Calico with an overlay configuration could have met the functional requirement, Cilium was selected for its more modern architecture and its alignment with current industry direction.

Cilium is built on eBPF (extended Berkeley Packet Filter), a Linux kernel technology enabling in-kernel packet processing and introspection without modifying kernel code. This provides several advantages: lower CPU overhead compared to iptables-based models, improved performance through efficient datapath execution, and comprehensive observability via tools such as flow logs and per-pod network metrics.

Beyond its performance characteristics, Cilium integrates functionality that would otherwise require additional components. Native NetworkPolicy enforcement offers fine-grained traffic control, while its built-in BGP support removes the architectural need for MetalLB. Optional service-mesh capabilities, though not required at present, offer a clear path for functional expansion without future CNI migration.

The adoption of Cilium therefore serves both immediate and strategic goals: it enables the required overlay networking while positioning the system for evolving networking needs.

3.5.2 Overlay Networking for Cross-Subnet Communication

The central networking challenge was enabling seamless pod-to-pod communication across nodes located in distinct datacenter subnets. Cilium addresses this requirement through the use of VXLAN (Virtual Extensible LAN) overlay networking.

In VXLAN mode, Cilium encapsulates pod traffic within UDP-based VXLAN tunnels that span L3 network boundaries. When a pod on one node communicates with a pod on a node in a different subnet, the Cilium agent encapsulates the packet with a VXLAN header and transmits it across the underlay network. The receiving node decapsulates the packet and forwards it to the destination pod. This process is fully transparent to workloads, effectively extending a virtual L2 network across the datacenter's heterogeneous network topology.

Because VXLAN operates over standard UDP, it requires only minimal accommodation from the physical network—namely, allowing the relevant UDP ports through any intermediate firewalls or ACLs. No specialized routing configuration is required.

This overlay-based approach provides the essential functionality missing from the earlier deployment: nodes in different datacenter network segments can now join the same cluster while supporting direct, reliable pod-to-pod communication.

3.5.3 BGP Configuration for Service Exposure

For exposing Kubernetes services externally, the design implements BGP-based load balancing using Cilium's integrated BGP support. This replaces the MetalLB deployment used in the original infrastructure, reducing the number of separate components to manage.

The cluster operates within a private service network for internal server management and communication. External services are exposed using public IP addresses specially alocated to the cluster. Cilium's BGP Control Plane is configured to peer with the datacenter's router infrastructure and announce these service IPs.

The physical network topology includes a router with a point-to-point connection to the datacenter's NEXUS switch infrastructure. Cilium BGP speakers running on designated nodes establish BGP sessions with this router, advertising routes for LoadBalancer service IPs. The router, in turn, propagates these routes through the datacenter's network, enabling external clients to reach services running in the cluster.

This BGP-based approach provides several benefits: traffic is distributed across multiple nodes advertising the same service IP through ECMP (Equal-Cost Multi-Path) routing, failed nodes are automatically removed from service as their BGP advertisements withdraw, and the integration with existing datacenter routing infrastructure is clean and standard.

3.6 GitOps and Operational Design

The operational layer of the new cluster infrastructure adopts a GitOps-based workflow centered around ArgoCD. This design replaces the manual and error-prone operational practices of the previous environment—where components were deployed through ad-hoc Helm commands—and introduces a fully declarative, auditable, and automated approach to managing cluster state.

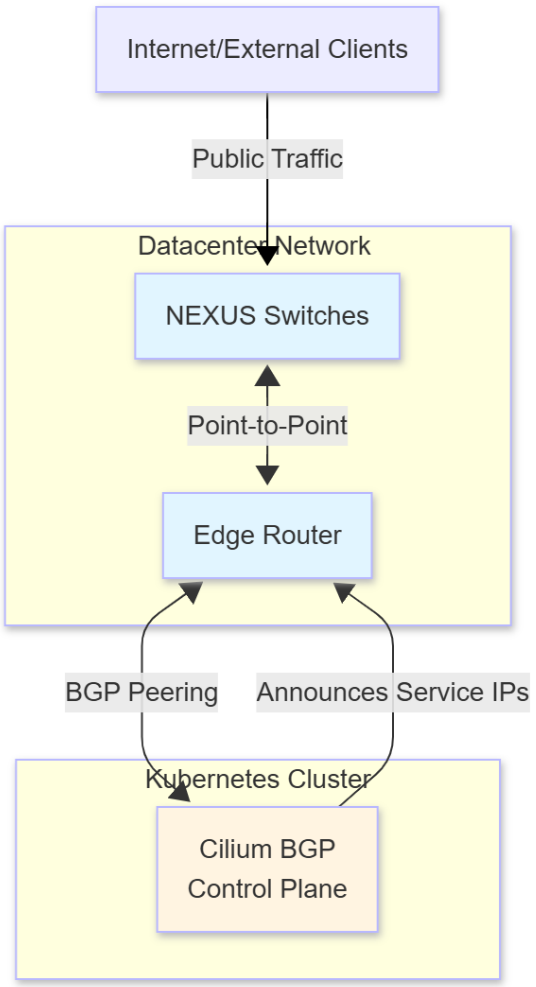


Figure 3.2: BGP Routing Topology

3.6.1 Introduction to ArgoCD and GitOps Workflow

ArgoCD is a Kubernetes-native continuous delivery system designed around GitOps principles. It continuously monitors one or more Git repositories and ensures that Kubernetes clusters match the desired state declared in version control. By treating Git as the single source of truth, ArgoCD provides deterministic and reproducible deployments, configuration transparency, and built-in drift detection.

Git as the Declarative Source of Truth

ArgoCD tracks the contents of Git repositories and reconciles them with the live cluster state. Each ArgoCD Application defines a mapping between a repository path and a deployment destination (cluster + namespace). ArgoCD observes these Applications and automatically applies any changes detected in Git. The system supports multiple configuration formats—including plain YAML, Helm charts, and Kustomize—allowing teams to retain established templating workflows.

Reconciliation and Drift Detection

The ArgoCD controller continuously compares desired state from Git with actual cluster state. When discrepancies arise, it marks the Application as OutOfSync and, depending on policy, automatically or manually restores alignment. This automated drift detection is especially valuable in environments where manual edits or external controllers may modify deployed resources.

Multi-Environment and Multi-Cluster Capabilities

A single ArgoCD installation can manage multiple clusters, each associated with its own applications or environments. By structuring Git repositories into separate directories or branches (e.g., staging and production), ArgoCD enables clean separation of environments while retaining a unified operational workflow.

Together, these capabilities provide the foundation for the operational layer described below.

3.6.2 Declarative Application Management

With ArgoCD deployed in the management cluster, all platform services and user applications are declared in a dedicated GitOps repository. ArgoCD monitors this repository and ensures that the cluster configuration continuously matches the committed manifests.

Adding a new application requires only placing its manifest or Helm chart reference into the appropriate directory. ArgoCD automatically creates or updates the corresponding Application resource, synchronizing it with the live cluster. Synchronization policies are configurable: foundational components may rely on automatic reconciliation, while production workloads can require manual approval.

This model transforms Git into a complete and auditable representation of the live system. Deployment reviews occur through pull requests, and any rollback can be performed by reverting or amending commits. Configuration drift — formerly a significant source of instability — is both easily detectable and automatically correctable.

3.6.3 CI/CD Integration Strategy

ArgoCD complements rather than replaces existing CI/CD pipelines. For the CrownLabs application, development workflows continue to use GitHub Actions for building container images and running tests. However, instead of the CI pipeline directly deploying to Kubernetes, it updates the GitOps repository to reference new image tags.

The planned workflow enhancement involves the CI pipeline committing image tag updates to the GitOps repository for staging and production environments. ArgoCD detects these commits and synchronizes the new versions to the appropriate clusters or namespaces. This separation of concerns means the CI system handles application building and testing, while ArgoCD handles deployment and state management.

This integration model provides clear separation between code repositories (where applications are developed) and configuration repositories (where deployments are declared). Access controls can be different: developers have write access to code repositories but may have read-only access to production configuration, with deployments requiring approval from operations staff.

3.6.4 Operational Benefits and Extensibility

Adopting ArgoCD and GitOps principles substantially improves operational reliability and maintainability:

- *Transparency*: the entire cluster state is stored in human-readable manifests under version control.
- *Auditability*: changes are reviewed through pull requests and recorded as commits.
- *Consistency*: development, staging, and production follow the same operational pattern.
- *Rollback safety*: reverting to a previous system state requires only reverting Git history.
- *Extensibility*: introducing new services or platform components requires no custom scripts—only new manifests committed to the repository.

The GitOps model therefore reduces operational complexity, minimizes configuration drift, and provides a scalable foundation for future growth of the CrownLabs platform.

Chapter 4

Implementation

4.1 Cluster Deployment with Kubespray

Kubespray was selected as the automation framework for deploying the Kubernetes cluster due to its maturity, flexibility, and full support for highly available, production-grade configurations. This section details the complete implementation workflow, from repository preparation to inventory design and cluster configuration, culminating in the final execution of the Kubespray playbooks.

4.1.1 Kubespray Deployment Workflow

The deployment process begins by retrieving the official Kubespray repository and preparing an isolated Python environment for installing the required Ansible dependencies. Kubespray is distributed as a collection of Ansible playbooks, and therefore a controlled Python virtual environment ensures reproducibility and prevents dependency conflicts with system-level Python packages.

The following commands were used to acquire Kubespray (version `v2.28.0`, the latest release at the moment of implementation) and prepare the environment:

```
git clone https://github.com/kubernetes-sigs/kubespray.git
cd kubespray
git checkout release-2.28

python3 -m venv venv
source venv/bin/activate
pip install -r requirements.txt
```

Once Kubespray and its dependencies are installed, an inventory workspace is created by copying the bundled sample inventory directory:

```
cp -rfp inventory/sample inventory/ha-cluster
```

All subsequent configuration is performed exclusively within the `inventory/ha-cluster` directory, which becomes the working environment for defining cluster nodes, component settings, and network parameters.

4.1.2 Configuring the Inventory

Kubespray uses an Ansible inventory to define the servers composing the Kubernetes cluster and their assigned roles. According to the official documentation:[1]

> "The inventory is composed of 3 groups:
>
> - **kube_node**: list of Kubernetes nodes where the pods will run.
> - **kube_control_plane**: list of servers where Kubernetes control plane components (apiserver, scheduler, controller) will run.
> - **etcd**: list of servers to compose the etcd server. You should have at least 3 servers for failover purpose."

Kubespray provides a default `inventory.ini` file, but YAML inventory format is also supported and was adopted for clarity. The inventory created for this deployment is shown below:

```
all:
  hosts:
    control_plane1:
      ansible_host: 192.168.10.11
      ansible_user: root
      etcd_member_name: k8s-01
    control_plane2:
      ansible_host: 192.168.10.12
      ansible_user: root
      etcd_member_name: k8s-02
    control_plane3:
      ansible_host: 192.168.10.13
      ansible_user: root
      etcd_member_name: k8s-03

    worker1:
      ansible_host: 192.168.10.21
```

[1]Kubespray Documentation: Inventory Structure

```
      ansible_user: root
      etcd_member_name: ""
    worker2:
      ansible_host: 192.168.10.22
      ansible_user: root
      etcd_member_name: ""
    worker3:
      ansible_host: 192.168.10.23
      ansible_user: root
      etcd_member_name: ""

  children:
    kube_control_plane:
      hosts:
        control_plane1:
        control_plane2:
        control_plane3:

    etcd:
      hosts:
        control_plane1:
        control_plane2:
        control_plane3:

    kube_node:
      hosts:
        worker1:
        worker2:
        worker3:
```

Listing 4.1: Kubespray Inventory file

The `all` section defines SSH profiles for each server. When `ansible_user` is not `root`, privilege escalation must be explicitly enabled. In such cases, the playbooks must be executed using the `-become` flag, and if a password is required, additionally `-ask-become-pass`.

4.1.3 Cluster Configuration Parameters

Cluster-specific configurations are defined within the `group_vars` directory of the working inventory. Only a subset of parameters needs modification for a standard highly available deployment. The relevant directory structure is:

```
inventory/ha-cluster
|---group_vars
|   |---all
|   |   |---all.yml
|    |---k8s_cluster
|        |---addons.yml
|        |---k8s-cluster.yml
|        |---k8s-net-cilium.yml
|---inventory.yml
```

The most important configuration groups are outlined below.

Global Parameters (all.yml).

Relevant options include:

- API server load balancer configuration (internal or external).
- Upstream DNS resolvers.
- Proxy settings (if required in the environment).
- Certificate management mode (`script` for automatic generation).
- RHEL subscription parameters (if applicable).

Cluster Parameters (k8s-cluster.yml).

Key settings include:

- OIDC integration, required for CrownLabs authentication.
- Network plugin selection (Cilium, Calico, Flannel, Weave, Kube-OVN, etc.).
- Pod and service CIDR blocks.
- Cluster name and DNS domain.
- Container runtime selection (default: containerd).
- Optional NVIDIA GPU acceleration (not used due to outdated implementation).

All the values adopted for the deployment are summarized below:

```
loadbalancer_apiserver_localhost: true
loadbalancer_apiserver_type: haproxy
upstream_dns_servers:
  - 8.8.8.8
  - 8.8.4.4
cert_management: script

```

```
kube_network_plugin: cilium
kube_service_addresses: 10.1.0.0/16
kube_pods_subnet: 172.23.0.0/16
cluster_name: cluster.local
container_manager: containerd
```

These parameters provide the minimal necessary configuration required for a fully functional Kubernetes cluster. Kubespray exposes numerous additional variables for fine-grained customization, but these were outside the scope of the deployment.

4.1.4 Configuring Cilium

Cilium functionality is configured in the `k8s-net-cilium.yml` file. For the purposes of this work, several Cilium features were relevant:

- *Overlay Networking:* VXLAN encapsulation was enabled to support cross-subnet pod communication.
- *Kube-proxy replacement:* Tested successfully but later disabled due to incompatibility with an external open-source component planned for integration.
- *Hubble:* Enabled to provide network observability, with the option for integration into Prometheus and Grafana.
- *BGP Control Plane:* A critical feature for integrating the cluster with the datacenter router.

Cilium BGP Control Plane

Cilium provides two BGP integration mechanisms: the legacy *BGP Peering Policy* and the modern *BGP Control Plane*. Although the legacy mechanism is simpler to configure, it is deprecated and scheduled for removal; therefore, the BGP Control Plane was used due to its long-term support and granular control.

According to Cilium's documentation:[2]

> "CiliumBGPClusterConfig: Defines BGP instances and peer configurations that are applied to multiple nodes.
>
> CiliumBGPPeerConfig: A common set of BGP peering settings. It can be used across multiple peers.

[2]Cilium Documentation: BGP Control Plane

> CiliumBGPAdvertisement: Defines prefixes that are injected into the BGP routing table.
>
> CiliumBGPNodeConfigOverride: Defines node-specific BGP configuration to provide a finer control."

The deployment required no node-specific overrides; therefore, only the first three resources were configured through Kubespray. The resulting Cilium configuration is shown below:

```
cilium_enable_bgp_control_plane: true

cilium_loadbalancer_ip_pools:
  - name: "public"
    cidrs:
      - "130.192.XX.XX/2X"

cilium_bgp_cluster_configs:
  - name: "cilium-bgp"
    spec:
      bgpInstances:
      - name: "instance-64512"
        localASN: 64512
        peers:
        - name: "peer-64512-tor1"
          peerASN: 64512
          peerAddress: "<bgp_router_ip_address>"
          peerConfigRef:
            name: "cilium-peer"
      nodeSelector:
        matchExpressions:
          - {key: somekey, operator: NotIn, values: ['never-used-value']}

cilium_bgp_peer_configs:
  - name: cilium-peer
    spec:
      gracefulRestart:
        enabled: true
        restartTimeSeconds: 15
      families:
        - afi: ipv4
          safi: unicast
          advertisements:
            matchLabels:
              advertise: "public"

cilium_bgp_advertisements:
  - name: bgp-advertisements
```

```
    labels:
      advertise: public
    spec:
      advertisements:
        - advertisementType: "Service"
          service:
            addresses:
              - LoadBalancerIP
          selector:
            matchExpressions:
              - {key: somekey, operator: NotIn, values: ['never-used-value']}
```

Listing 4.2: Cilium BGP Control Plane Configuration

This configuration assigns a load balancer pool of publicly routable addresses, defines the BGP instance and its peer (the datacenter edge router), specifies BGP capabilities for the peer (IPv4 unicast), and advertises Kubernetes `LoadBalancer` services into the upstream routing domain.

4.1.5 Executing the Deployment

Once the inventory and configuration files are fully defined, the cluster can be provisioned by executing the main Kubespray playbook:

```
ansible-playbook -i inventory/ha-cluster/inventory.yml \
  cluster.yml -b -v --private-key=~/.ssh/<private_ssh_key_file>
```

The deployment completes in approximately 30 minutes depending on node count and network conditions. Upon completion, the new Kubernetes cluster is operational, fully configured with high availability, Cilium networking, and BGP integration through the Cilium control plane.

Kubespray also facilitates lifecycle operations through dedicated playbooks. To scale the cluster by adding or removing nodes, the operator simply updates the inventory file to reflect the desired set of servers and executes the same command while substituting the playbook name with `scale.yml` or `remove-node.yml`. The `remove-node.yml` playbook not only removes the node from the cluster membership but also ensures that all Kubernetes components, services, and configurations previously installed by Kubespray are thoroughly cleaned from the target machine.

Finally, if the intention is to dismantle the entire cluster, including all control plane and worker nodes, Kubespray provides the `reset.yml` playbook. Executing this playbook reverts all machines in the inventory to a clean state by removing the full set of Kubernetes-related artifacts, thereby restoring the servers to a pre-deployment condition.

4.2 ArgoCD Setup

Following the successful deployment of the Kubernetes cluster using Kubespray, the next step in establishing a fully automated and declarative management workflow is the installation and configuration of ArgoCD. As the central component of the GitOps model adopted in this thesis, ArgoCD is responsible for continuously reconciling the cluster state against the manifests stored in a dedicated Git repository. This section describes the deployment architecture of ArgoCD, the structure of the GitOps repository created for this environment, and the process of populating the repository with the full CrownLabs stack and supporting components.

4.2.1 ArgoCD Deployment Architecture

ArgoCD is deployed using Helm through a composed chart structure that includes two subcharts: the official `argo-cd` chart, which installs the ArgoCD control plane together with all required Custom Resource Definitions (CRDs), and the `argocd-apps` chart, which bootstraps the initial set of ArgoCD *Application* resources. This two-stage design ensures that the core ArgoCD installation becomes available before the management of higher-level applications begins.

The values supplied to the `argo-cd` chart define the configuration of the ArgoCD instance installed in the cluster, including repository credentials, RBAC policies, resource settings, and networking options. Crucially, this chart also installs all ArgoCD CRDs, enabling the cluster to recognise and process `Application`, `AppProject`, and related custom resource types before any such resources are created.

In the second stage, the `argocd-apps` chart is deployed with a set of values that describe ArgoCD `Application` resources representing both ArgoCD itself and the `argocd-apps` bootstrap configuration. These definitions include references to the GitOps repository created for this project, specifying the repository URL, the directory paths containing configuration manifests, and the desired synchronisation policies. As a result, immediately after installation, ArgoCD begins to manage and continuously reconcile its own configuration. Updates to ArgoCD functionality—such as changes to resource limits, RBAC, projects, or application definitions—are therefore performed by modifying the manifests stored in the Git repository, eliminating the need for external tooling to configure the GitOps controller.

GitOps Repository Structure

To support a self-managed GitOps workflow, a dedicated repository was designed following a clear, hierarchical structure that separates concerns and enables predictable automation:

```
repository-root/
    |---argocd/
    |   |---apps/
    |   |   |---<project-name>/
    |   |   |     |---<application-manifests>.yaml
    |   |   |---...
    |   |---install/
    |       |---<argocd-helm-chart-configuration>/
    |---charts/
        |---<application-name>/
        |     |---<helm-chart-or-values>/
        |---...
```

The `argocd/install` directory contains the Helm chart configuration for deploying ArgoCD itself. This directory serves as the source for the `argocd` application, enabling ArgoCD's full self-management cycle: any modifications committed to this directory are automatically detected and applied by ArgoCD.

The `argocd/apps` directory contains the manifests defining ArgoCD applications and, optionally, their associated projects. Each subdirectory represents a logical project grouping, such as infrastructure components, monitoring systems, or the CrownLabs application stack. Each application manifest specifies the repository path, Helm chart or values file to use, destination namespace, and synchronisation policy governing its deployment. These manifests are consumed by the `argocd-apps` bootstrap application, which tracks and synchronises all applications defined within this structure.

The `charts` directory stores custom Helm charts or values files for applications requiring configuration beyond upstream defaults. When an application needs customised parameters, a directory is created here containing the corresponding values file. Applications referencing unmodified upstream charts instead point directly to the upstream Helm repository without requiring a local directory. This hybrid design preserves flexibility while maintaining clarity regarding which components are customised and which are deployed from vendor-maintained charts.

4.2.2 Repository Population and CrownLabs Stack Migration

With the GitOps repository and ArgoCD bootstrap configuration in place, the main implementation effort of this thesis involved migrating all CrownLabs components into the new declarative management structure. Each existing service was analysed individually—either by inspecting its previous deployment manifests or by examining the values used in its Helm release—to reconstruct a complete GitOps-compatible specification. During this process, several components were upgraded to more recent versions, and others were reconfigured to better align with the requirements of the new highly available cluster architecture.

All applications were added to the repository as ArgoCD `Application` resources within the appropriate project directory. Their configuration was expressed exclusively through Helm values files stored under the `charts` directory when customisation was needed. This approach ensured consistency across applications and simplified future maintenance.

As part of enhancing the capabilities of the new cluster, the NVIDIA GPU Operator was introduced to support GPU-accelerated workloads. Its deployment illustrates the simplicity and repeatability of adding new services under ArgoCD management. The following `Application` resource was committed to the repository:

```yaml
apiVersion: argoproj.io/v1alpha1
kind: Application
metadata:
  name: gpu-operator
  namespace: argocd
spec:
  destination:
    namespace: gpu-operator
    server: https://kubernetes.default.svc
  project: cluster-base
  source:
    helm:
      valueFiles: [./Values.yaml]
    path: charts/gpu-operator
    repoURL: git@github.com:git-user/gitRepo.git
    targetRevision: HEAD
  syncPolicy:
    syncOptions:
      - CreateNamespace=true
      - ServerSideApply=true
```

Listing 4.3: GPU Operator Application Manifest

To deploy any new application, this manifest can be reused by adjusting the application name, destination namespace, and the path referencing the correct Helm values file.

For the GPU Operator itself, only the parameters deviating from chart defaults were specified:

```yaml
gpu-operator:
  nfd:
    enabled: false

  driver:
    enabled: false

  sandboxWorkloads:
    enabled: true

  toolkit:
    env:
      - name: CONTAINERD_CONFIG
        value: /etc/containerd/config.toml
      - name: CONTAINERD_SOCKET
        value: /run/containerd/containerd.sock
      - name: CONTAINERD_RUNTIME_CLASS
        value: nvidia
      - name: CONTAINERD_SET_AS_DEFAULT
        value: "false"
```

Listing 4.4: GPU Operator Helm Values

Once these files were pushed to the repository, the `argocd-apps` bootstrap application detected them and prepared the GPU Operator for deployment. If automatic synchronisation were enabled, the operator would be deployed immediately without requiring manual approval. This mechanism applies uniformly to all components in the system, demonstrating the effectiveness and scalability of the GitOps workflow.

The final step in populating the repository consisted of deploying CrownLabs itself as an ArgoCD application. Throughout the testing and validation phases, CrownLabs was redeployed multiple times by simply updating the referenced commit in the GitOps repository, rather than relying on fixed versioned releases. This demonstrated both the flexibility of the GitOps setup and the ease of recovery from misconfigurations or deployment failures: by resetting the commit reference, the entire CrownLabs stack could be restored to any known-good state in a matter of minutes.

This concludes the implementation chapter, having established the cluster infrastructure, deployed ArgoCD with a fully self-managed configuration, and migrated the entire CrownLabs ecosystem into a unified and declarative GitOps repository.

Chapter 5

Results

This chapter presents the outcomes of the implementation described in the previous chapter. First, it evaluates the improvements in deployment, scaling, and operational efficiency achieved by using KubeSpray and ArgoCD.

5.1 Deployment and Scaling Efficiency

Using the newly configured Kubespray-based cluster deployment workflow, several concrete improvements in operational efficiency were observed:

Reproducible initial deployment. After performing the repository checkout, virtual environment setup, inventory configuration, and parameterization, executing the primary playbook produced a fully functional, HA-enabled Kubernetes cluster. The process followed a repeatable, documented path — eliminating the ad hoc, manual steps that characterised the previous infrastructure setup.

Reduced manual effort for node management. The same playbook machinery, with minimal modifications to the inventory, was used to add and remove worker nodes. The automation included installing necessary software, joining or draining nodes, and cleaning up when nodes were removed.

Improved time-to-deployment. According to the recorded data (see Figures 5.1, 5.2, 5.3), cluster deployment, node addition, and node removal all completed in under 20–40 minutes depending on the number of nodes — a major improvement over the manual server provisioning and configuration process previously used. These results demonstrate that frequent scaling operations (for example, to accommodate concurrent lab sessions in CrownLabs) are now practical without extensive manual labor or long waiting times.

This efficiency gain is essential for educational and research environments, where hardware availability and peak loads can vary significantly and unpredictably.

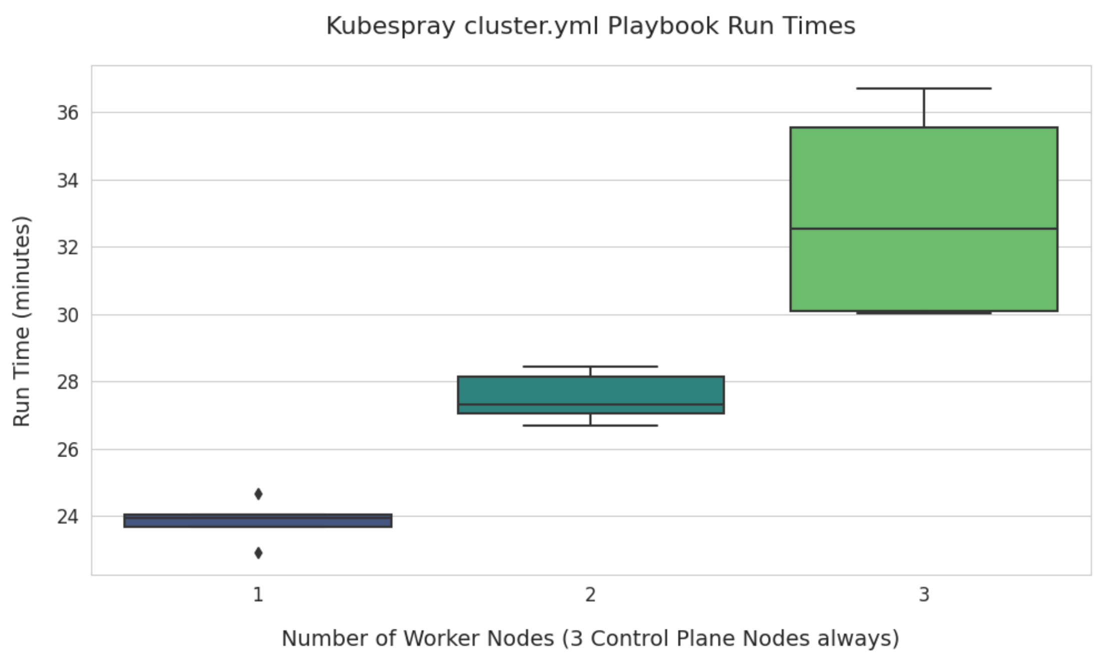


Figure 5.1: Cluster Deployment Playbook Boxplot

5.2 Operational Management: GitOps with ArgoCD

Once the cluster was deployed and operational, the introduction of ArgoCD as the operational layer proved transformative for application and configuration management:

- Immediate self-management of ArgoCD. Because ArgoCD manages its own configuration, any change to its settings—such as RBAC, project definitions, or helm chart values—can be made simply by updating manifests in the Git repository and committing them. ArgoCD synchronises itself accordingly, establishing a closed self-managed loop.

- Clear, reproducible deployment of applications. The GitOps repository structure (with separate directories for ArgoCD installation, applications, and Helm charts/values) provided a clean separation of concerns and predictable deployment mechanics. With this structure, deploying an application required only committing its manifest (and optional Helm values) and letting ArgoCD handle the rest.

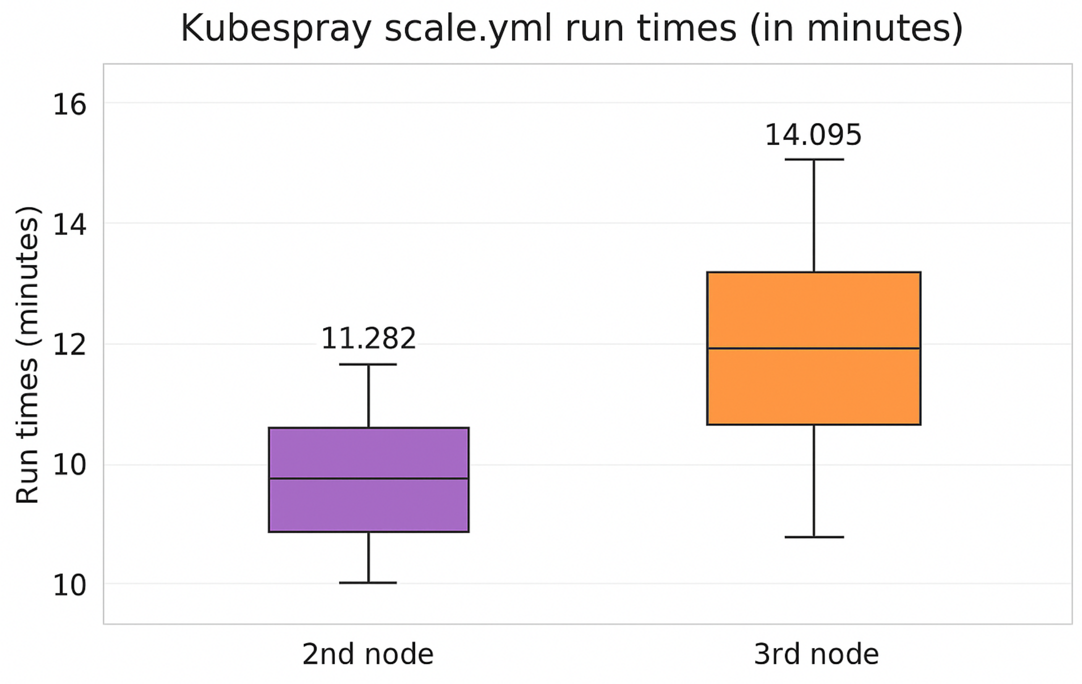


Figure 5.2: Scale Playbook Boxplot

- Simplified stack migration. All existing CrownLabs services, previously deployed manually or via ad-hoc scripts, were migrated into the repository. Many component versions were updated; others reconfigured for HA or compatibility with the new cluster setup. Because all configuration is now version-controlled, it is straightforward to track changes, roll back misconfigurations, and replicate the setup in testing or staging clusters.

- Rapid deployment on fresh clusters. On a newly provisioned cluster, a single commit of the GitOps repository sufficed to deploy the entire CrownLabs stack — demonstrating that the combination of Kubespray and ArgoCD effectively provides "cluster-as-code" + "infrastructure-as-code" for bare-metal environments.

These operational improvements dramatically reduce administrative overhead, minimize chances for configuration drift, and provide a reproducible, auditable, and maintainable deployment model.

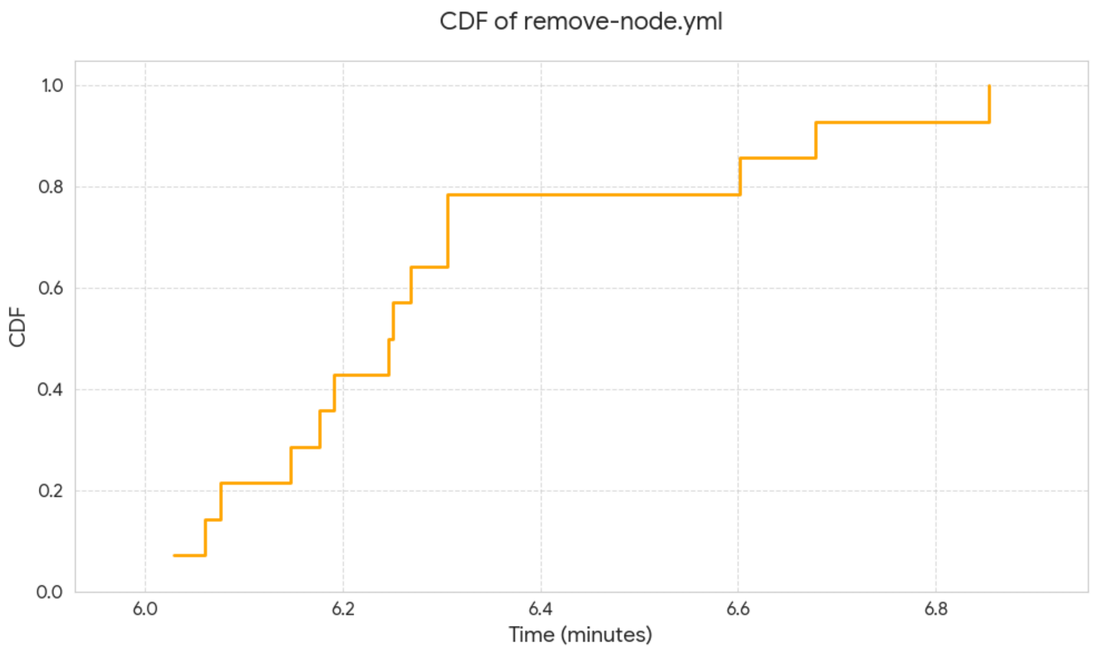
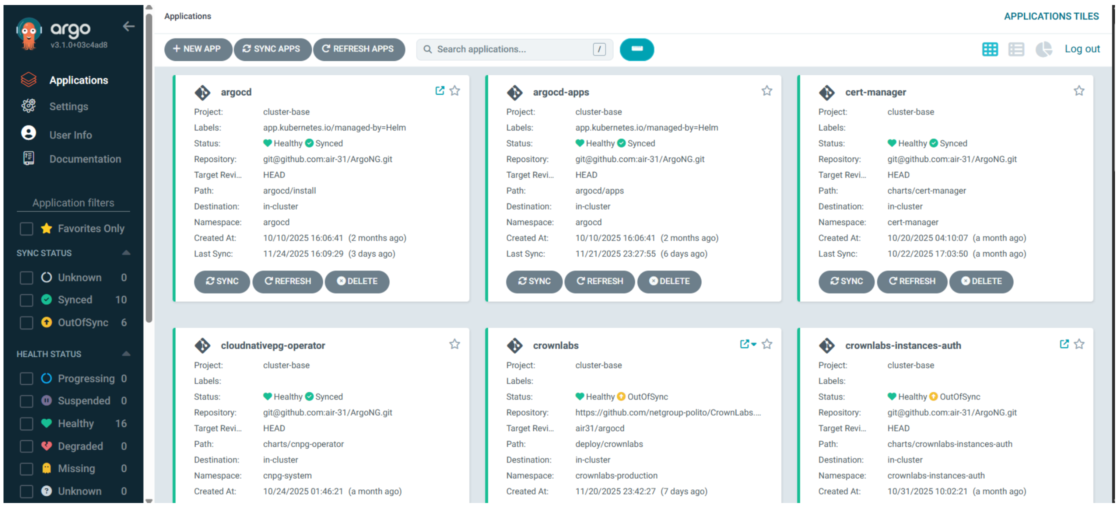


Figure 5.3: Node Removal Playbook CDF



argocd

Project:	cluster-base
Labels:	app.kubernetes.io/managed-by=Helm
Status:	Healthy Synced
Repository:	git@github.com:air-31/ArgoNG.git
Target Revi...	HEAD
Path:	argocd/install
Destination:	in-cluster
Namespace:	argocd
Created At:	10/10/2025 16:06:41 (2 months ago)
Last Sync:	11/24/2025 16:09:29 (3 days ago)

argocd-apps

Project:	cluster-base
Labels:	app.kubernetes.io/managed-by=Helm
Status:	Healthy Synced
Repository:	git@github.com:air-31/ArgoNG.git
Target Revi...	HEAD
Path:	argocd/apps
Destination:	in-cluster
Namespace:	argocd
Created At:	10/10/2025 16:06:41 (2 months ago)
Last Sync:	11/21/2025 23:27:55 (6 days ago)

cert-manager

Project:	cluster-base
Labels:	
Status:	Healthy Synced
Repository:	git@github.com:air-31/ArgoNG.git
Target Revi...	HEAD
Path:	charts/cert-manager
Destination:	in-cluster
Namespace:	cert-manager
Created At:	10/20/2025 04:10:07 (a month ago)
Last Sync:	10/22/2025 17:03:50 (a month ago)

cloudnativepg-operator

Project:	cluster-base
Labels:	
Status:	Healthy Synced
Repository:	git@github.com:air-31/ArgoNG.git
Target Revi...	HEAD
Path:	charts/cnpg-operator
Destination:	in-cluster
Namespace:	cnpg-system
Created At:	10/24/2025 01:46:21 (a month ago)

crownlabs

Project:	cluster-base
Labels:	
Status:	Healthy OutOfSync
Repository:	https://github.com/netgroup-polito/CrownLabs....
Target Revi...	air31/argocd
Path:	deploy/crownlabs
Destination:	in-cluster
Namespace:	crownlabs-production
Created At:	11/20/2025 23:42:27 (7 days ago)

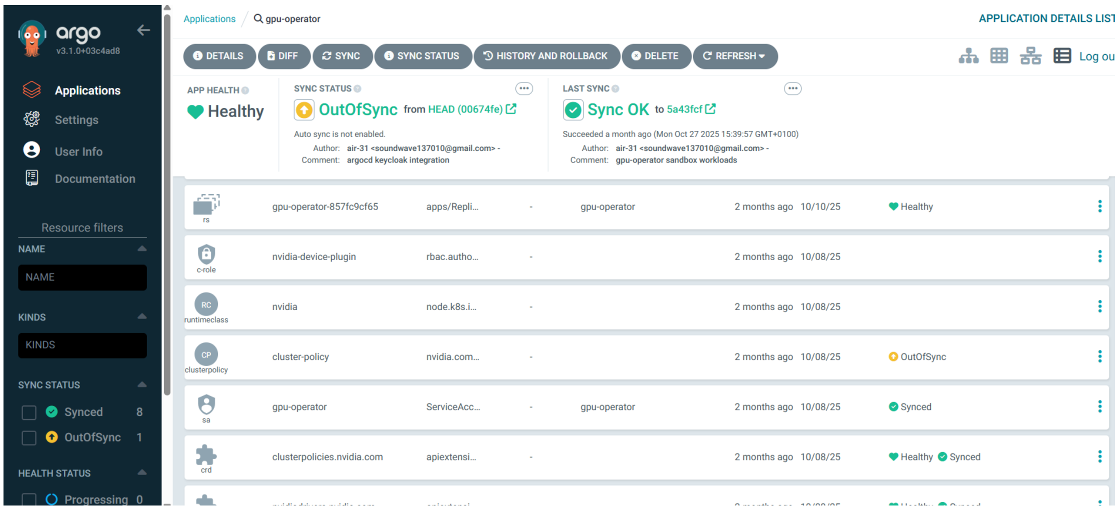
crownlabs-instances-auth

Project:	cluster-base
Labels:	
Status:	Healthy OutOfSync
Repository:	git@github.com:air-31/ArgoNG.git
Target Revi...	HEAD
Path:	charts/crownlabs-instances-auth
Destination:	in-cluster
Namespace:	crownlabs-instances-auth
Created At:	10/31/2025 10:02:21 (a month ago)

Figure 5.4: ArgoCD Applications

5.3 Introduction of GPU-Accelerated Workloads

A key objective of the redesign was to enable GPU-accelerated workloads for courses related to machine learning, scientific computing, or graphics processing.



APP HEALTH
Healthy

SYNC STATUS
OutOfSync from HEAD (00674fe)
Auto sync is not enabled.
Author: air-31 <soundwave137010@gmail.com> -
Comment: argocd keycloak integration

LAST SYNC
Sync OK to 5a43fcf
Succeeded a month ago (Mon Oct 27 2025 15:39:57 GMT+0100)
Author: air-31 <soundwave137010@gmail.com> -
Comment: gpu-operator sandbox workloads

rs	gpu-operator-857fc9cf65	apps/Repli...	-	gpu-operator	2 months ago	10/10/25	Healthy
c-role	nvidia-device-plugin	rbac.autho...	-		2 months ago	10/08/25	
runtimeclass	nvidia	node.k8s.i...	-		2 months ago	10/08/25	
clusterpolicy	cluster-policy	nvidia.com...	-		2 months ago	10/08/25	OutOfSync
sa	gpu-operator	ServiceAcc...	-	gpu-operator	2 months ago	10/08/25	Synced
crd	clusterpolicies.nvidia.com	apiextensi...	-		2 months ago	10/08/25	Healthy Synced

Figure 5.5: ArgoCD Resource View

The implementation successfully demonstrated this capability:

- The NVIDIA GPU Operator was added to the GitOps repository and deployed via ArgoCD like any other application, using a Helm chart with minimal overrides.
- A GPU-enabled workspace was instantiated for the "AI-Enhanced Architecture" course, providing students with access to virtualized environments with access to NVIDIA GPUs and schedulable via Kubernetes.

5.4 Addressing Original Limitations

By combining a robust provisioning baseline, automated cluster deployment, and declarative operational management, the new infrastructure successfully overcomes the key limitations identified in the original CrownLabs setup:

- Scalability constraints have been alleviated: nodes can be added or removed swiftly, enabling the platform to respond to varying demand without long delays.
- Manual upgrade burden and maintenance windows have been replaced by predictable and repeatable procedures, reducing downtime and operational risk.
- Single point of failure in the control plane has been mitigated by deploying a highly available control plane (on VMs), while worker nodes run on bare-metal servers.

- Networking inflexibility — particularly the previous inability for cross-subnet pod communication — has been resolved via Cilium overlay + BGP configuration, enabling distributed resources to join the same cluster.
- Lack of support for GPU workloads has been addressed by deploying the GPU operator and verifying GPU-enabled environments.
- Configuration drift and inconsistent deployments have been replaced by GitOps-based declarative management, ensuring reproducibility, auditability, and version-controlled changes.

Chapter 6

Conclusions

The work presented in this thesis — "Automating the Deployment and Operation of a Scalable Bare-Metal Kubernetes Cluster" — demonstrates that combining mature open-source tools with a well-thought-out architecture can successfully bring cloud-style automation to on-premise infrastructure in academic and research environments.

By leveraging Kubespray for cluster deployment and lifecycle management, and ArgoCD for operational GitOps workflows, the resulting solution provides:

- reproducible and efficient cluster provisioning;
- automated scaling and upgrade procedures;
- high availability and fault tolerance;
- a fully declarative, version-controlled, and auditable configuration management process.

In the context of CrownLabs, the new infrastructure removes previous operational bottlenecks and significantly reduces manual workload for administrators. The ability to redeploy the full system from a clean slate by simply applying the GitOps repository illustrates that bare-metal infrastructure can indeed achieve a level of manageability and resilience comparable to public clouds.

Overall, this thesis shows that with appropriate tooling and architecture, on-premise bare-metal environments can be managed efficiently, reliably, and flexibly — making them a viable, and sometimes preferable, alternative for academic and research institutions.

Future work may explore further enhancements such as automated bare-metal provisioning integration (e.g., via cluster-API), dynamic scaling policies based on workload metrics, tighter integration between GPU scheduling and user workflows, and multi-cluster or multi-region deployments. However, the results obtained here constitute a strong foundation and prove the viability of the approach.