POLITECNICO DI TORINO

Master Degree course in Computer Engineering

Master Degree Thesis

Seamless Upgrades in Cloud-Native Environments: Challenges and Validation via a Liqo-based PoC

Supervisors
Prof. Fulvio Risso
Dr. Alessandro Rontani
Dr. Stefano Galantino

Candidate
Kazem Bigdeli

Academic Year 2025-2026

Acknowledgements

I would like to express my deepest gratitude to Professor Fulvio Risso, not only for supervising this thesis but for being a constant source of encouragement and inspiration throughout my entire Master's degree. His guidance extended well beyond academic matters, and I am truly grateful for his support at every stage of this journey.

I would also like to sincerely thank Dr. Alessandro Rontani for his supervision and for the constructive and supportive discussions we had during our thesis meetings. His feedback always helped me see the work more clearly and move forward with greater confidence.

A special thank you goes to Dr. Stefano Galantino, who supervised this thesis and was also a wonderful colleague during my time as a research assistant. His thoughtful suggestions — on both the research work and the thesis itself — were invaluable, and I deeply appreciated the collaborative spirit he brought to every interaction.

I owe an immeasurable debt of gratitude to my family, whose unconditional love and support have been the foundation of everything I have accomplished. Their encouragement throughout my life and studies has meant more than words can express.

Finally, I would like to thank my friend Pouria Ahari for his technical help at key moments during this thesis. Having someone to turn to made a real difference, and I am grateful for his willingness to help.

I am also grateful to the Liqo open-source community for their well-documented codebase, which made this research possible.

Abstract

The shift toward the cloud-native paradigm and distributed, microservices-based systems has established Kubernetes as the de facto standard for container orchestration. However, as organizations increasingly operate fleets exceeding 20 clusters to support mission-critical workloads, the complexity of lifecycle management—specifically infrastructure upgrades—has scaled exponentially. Within this landscape, Liqo provides a peer-to-peer model for transparent workload offloading and resource sharing. Despite its utility, current Liqo upgrade procedures remain entirely manual, requiring administrators to follow exact, error-prone sequences that risk leaving systems in inconsistent, non-functional states.

This thesis designs, implements, and validates a Kubernetes operator that automates Liqo upgrades with minimal service disruption. The proposed solution employs a three-stage infrastructure upgrade architecture—sequencing Custom Resource Definition (CRD) migration before control plane updates, and control plane updates before network fabric modifications—to respect inherent architectural dependencies. By adopting a job-based execution model, the operator achieves fault isolation and provides a cumulative rollback strategy that restores the entire stack to a consistent pre-upgrade state upon detecting failure.

A critical design contribution of this work is the parallel gateway upgrade orchestration. Unlike traditional application-level canary releases, which would multiply downtime across multiple peerings, the parallel strategy coordinates disruption windows to minimize total connectivity loss. This approach acknowledges that zero-downtime upgrades are unachievable due to platform-inherent constraints, such as stateful WireGuard tunnel renegotiations and Linux conntrack table flushes.

Experimental results across nine representative scenarios demonstrate that the operator successfully reduces upgrade complexity from an intensive manual task to a predictable, five-minute automated workflow. The system achieved a 100% success rate in functional validation, with total upgrade durations averaging 312 seconds. Service disruption was confined exclusively to the network fabric stage, with measured downtime ranging from 14 to 28 seconds—a significant improvement over the 5–10 minute windows typically required for manual procedures. This work establishes that operator-based automation can effectively tame multi-cluster infrastructure upgrades, providing a reusable architectural pattern for other complex Kubernetes extensions.

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

1.1 The Cloud-Native Paradigm

Software development has undergone fundamental transformation over the past decade. Traditional monolithic architectures have given way to distributed, microservices-based systems designed for dynamic cloud environments. Kubernetes has emerged as the de facto standard for container orchestration, with 96% of enterprises now using, piloting, or evaluating it, and 80% deploying it in production [1].

However, this transformation introduces new lifecycle management challenges. The distributed nature of cloud-native applications makes traditional upgrade approaches inadequate. Organizations running mission-critical workloads—72% run databases, 67% run analytics, 54% run AI/ML workloads—face unprecedented stakes for seamless, non-disruptive upgrades.

1.2 The Challenge of Seamless Updates

The ability to update systems without service disruption has evolved from a desirable feature to a critical business requirement. The average cost of downtime exceeds $300,000 per hour for over 90% of mid-size and large enterprises [2].

Traditional update approaches are fundamentally inadequate for cloud-native systems. Kubernetes comprises numerous components—API server, controller manager, scheduler, etcd, kubelet, networking plugins—each requiring updates in specific sequences. Multi-cluster environments compound these challenges exponentially. Platforms like Liqo, which enable dynamic resource sharing between clusters, introduce additional complexity: network tunnels, virtual nodes, and cross-cluster control plane components must continue functioning throughout upgrades.

Manual update procedures are inherently error-prone. The Kubernetes operator pattern offers promise by encoding operational knowledge in software, but existing operators focus on application-level concerns, leaving infrastructure upgrades as largely manual processes.

1.3 Multi-Cluster Complexity and Liqo

Modern enterprises increasingly operate entire fleets: the average Kubernetes adopter operates more than 20 clusters across multiple environments [3]. This proliferation is driven by geographic distribution, environment separation, and multi-cloud strategies. Each additional cluster introduces operational overhead, configuration drift, and security management complexity.

Liqo extends Kubernetes through virtual nodes, enabling transparent workload offloading without requiring new APIs. When clusters peer, Liqo creates virtual nodes representing remote cluster resources. This relies on sophisticated infrastructure: secure WireGuard VPN tunnels for pod-to-pod communication, cross-cluster service discovery, and control plane components coordinating resource sharing.

Current Liqo upgrade procedures are entirely manual. Administrators must follow exact sequences: which CRDs to update first, which control plane components to restart in what order, when to upgrade network gateways. Component upgrade order cannot be dynamically discovered—it derives from architectural knowledge and must be explicitly encoded, as Liqo provides no discovery API for dependency relationships. A single mistake can leave systems in partially upgraded states difficult to diagnose and recover from.

1.4 The IPCEI-CIS AVANT Project

This thesis has been developed in collaboration with the Engineering Ingegneria Informatica S.p.A. [4], as part of the European project IPCEI-CIS AVANT [5], funded by the European Union with program NextGenerationEU.

IPCEI stands for Important Project of Common European Interest. These are large-scale, strategic projects co-funded by multiple EU Member States and approved under EU state-aid rules, to support innovation, infrastructure, or technologies of major EU importance. CIS means Cloud Infrastructure and Services. IPCEI-CIS is the IPCEI focused on developing next-generation cloud and edge computing infrastructures and services in Europe.

AVANT (an acronym from dAta and infrastructural serVices for the digitAl coNTinuum) is one of the projects under the umbrella of IPCEI-CIS. It is led by Engineering and aims to deliver advanced cloud-to-edge technologies, flexible infrastructures, interoperability, and open source components.

This thesis contributes to the project with the design and implementation of a Kubernetes operator automating seamless upgrades of Liqo, enabling reliable multi-cluster infrastructure management as part of the cloud-to-edge continuum envisioned by AVANT.

1.5 Thesis Objectives

This thesis designs, implements, and validates a Kubernetes operator that automates Liqo upgrades with minimal service disruption. The operator must orchestrate upgrades in

correct sequence (CRDs before controllers, control plane before network fabric), perform health validation at each stage, and provide reliable rollback when issues are detected.

The minimal downtime strategy recognizes that zero downtime is unachievable due to fundamental Kubernetes and Liqo characteristics. Kubernetes pod recreation provides no atomic handoff of network state. Liqo's WireGuard tunnels are stateful connections tied to specific pod instances; when gateway pods restart, clients and servers must negotiate new tunnels—a process requiring several seconds. The operator minimizes disruption through careful orchestration while acknowledging these inherent constraints.

Current limitations include manual specification of environment variable changes and CRD schema migrations, as this semantic meaning is embedded in the Liqo source code and not exposed through machine-readable formats. Future work will address this through ConfigMap-driven descriptors. Canary upgrade functionality remains incomplete; while partial implementation exists, full production-ready canary upgrades are deferred to future work.

1.6 Thesis Organization

Chapter 2 formalizes the upgrade challenge, establishing that minimal downtime rather than zero downtime is realistic given Kubernetes pod lifecycle and Liqo tunnel constraints.

Chapter 3 surveys existing approaches and performs gap analysis, noting that canary functionality remains incomplete.

Chapter 4 explains why component order must be hardcoded—Liqo's dependencies are not exposed through discovery APIs.

Chapter 5 presents the system architecture with pseudo-code including try-catch error handling, acknowledging manual specification limitations.

Chapter 6 describes the implementation using Kubebuilder and job-based execution.

Chapter 7 validates through experimental results showing that minimal downtime occurs due to inherent architectural characteristics.

Chapter 8 concludes with contributions, limitations, and future work including canary completion and configuration-driven approaches.

Chapter 2

State of the Art

2.1 Cloud-Native Deployment and Update Strategies

Organizations use three primary deployment strategies to introduce new software versions: rolling updates, blue-green deployments, and canary releases. While these strategies work well for applications, they cannot directly solve infrastructure upgrade problems in multi-cluster environments. This section examines each strategy's mechanisms and limitations for infrastructure upgrades [6].

2.1.1 Rolling Updates

Rolling updates incrementally replace old pod instances with new versions, maintaining service availability throughout the process [6]. Kubernetes uses this as the default deployment strategy.

The mechanism replaces old pods with new ones in controlled batches. Kubernetes creates new pods running the updated version while old pods continue serving requests. Only after new pods become healthy and ready to accept traffic does Kubernetes terminate corresponding old pods. This process repeats iteratively until all pods run the new version, maintaining service continuity throughout.

Kubernetes provides two key parameters to control rolling update behavior: `maxSurge` and `maxUnavailable`. The `maxSurge` parameter specifies the maximum number of pods that can be created above the desired replica count during an update. Conversely, `maxUnavailable` defines the maximum number of pods that can be unavailable during the update process. These parameters enable fine-grained control over update speed and availability guarantees. Conservative settings like `maxSurge: 1` and `maxUnavailable: 0` ensure the deployment always maintains the desired number of ready pods, updating one pod at a time.

The rolling update process integrates tightly with Kubernetes' health checking mechanisms. Readiness probes determine when newly created pods are ready to serve traffic. Until a new pod passes its readiness checks, Kubernetes does not consider it available and will not terminate corresponding old pods. This integration prevents inadvertently reducing available capacity by creating non-functional pods. Liveness probes monitor pod

health throughout the update, allowing Kubernetes to detect and replace unresponsive pods.

One key advantage lies in automatic rollback capability. If new pods fail to become ready within a configured timeout, Kubernetes halts the update process automatically, preventing further pod replacements. This fail-safe mechanism minimizes problematic update impact by detecting issues early when only a subset of pods has been replaced.

However, rolling updates exhibit several limitations. First, they provide no mechanism for sophisticated traffic management or gradual exposure to user subsets. All pods passing health checks immediately receive production traffic proportional to their count, with no ability to control what percentage of requests reach the new version. Second, rolling updates offer limited validation capabilities before full deployment. While health checks verify basic functionality, they cannot assess whether the new version meets performance requirements or produces correct business outcomes under realistic load. Third, resource overhead depends heavily on `maxSurge` configuration. Setting `maxSurge` greater than zero requires temporarily provisioning additional capacity beyond normal deployment size.

Finally, rolling updates address application-level updates but do not extend naturally to infrastructure upgrades. Updating Kubernetes itself, cluster networking components, or platform-level services requires different approaches since these components cannot typically run multiple versions simultaneously or be gradually replaced without affecting the entire cluster.

Despite these limitations, rolling updates remain the workhorse deployment strategy for cloud-native applications due to their balance of simplicity, safety, and minimal downtime.

2.1.2 Blue-Green Deployments

Blue-green deployment represents a release strategy that maintains two identical production environments—designated "blue" and "green"—to enable zero-downtime updates with instant rollback capability [7]. Unlike rolling updates that gradually replace instances, blue-green deployments switch traffic instantaneously between complete environments.

The fundamental architecture requires provisioning two parallel environments with identical infrastructure and configuration. At any given time, one environment (traditionally "blue") serves production traffic while the other ("green") remains idle or runs the previous version. When deploying a new application version, it is installed and tested in the idle green environment. Once validation completes successfully, a load balancer or DNS configuration switches all traffic from blue to green instantaneously. The previous blue environment then becomes idle, ready to serve as the rollback target if issues arise with the new version.

This instantaneous traffic switching distinguishes blue-green deployments from rolling updates. Rather than incrementally migrating users to the new version, all traffic redirects simultaneously once the switch occurs. This eliminates the complexity of managing multiple versions serving production traffic concurrently and ensures all users experience the same application version at any moment.

The primary advantage lies in simplified rollback. If problems emerge with the green

environment after traffic switches, reverting requires only redirecting traffic back to the blue environment. This rollback completes in seconds—merely updating load balancer configuration or DNS records—without requiring pod deletions, image rollbacks, or gradual migration.

Blue-green deployments also enable testing in production-equivalent environments. The green environment receives the new version before any production traffic, allowing comprehensive validation under realistic conditions. Teams can execute smoke tests, integration tests, and performance benchmarks without exposing users to potential issues.

However, blue-green deployments impose significant resource overhead. Maintaining two complete production environments doubles infrastructure requirements compared to single-environment deployments. Every server, database, cache, and supporting service must be duplicated, substantially increasing costs.

Database synchronization presents another challenge. Database schema changes must be backward-compatible with both application versions during the transition period. Alternatively, separate databases for blue and green environments necessitate data replication mechanisms and strategies for handling writes during cutover.

The instantaneous traffic switch, while advantageous for rollback, provides no gradual validation period. Unlike canary releases that expose new versions to user subsets first, blue-green deployments transition all users simultaneously. If the new version contains issues that escaped pre-production testing, all users immediately encounter those issues.

Despite these limitations, blue-green deployments excel for critical applications where rollback speed and deployment predictability outweigh resource costs.

2.1.3 Canary Releases

Canary deployment represents a progressive rollout strategy where new software versions are gradually introduced to a small subset of users before full deployment [8]. Unlike blue-green deployments that switch all traffic instantaneously, canary releases incrementally shift traffic to the new version while monitoring behavior, enabling early detection of issues with minimal user impact.

The fundamental mechanism involves deploying the new version alongside the existing version and directing a small percentage of production traffic—typically 5–10% initially—to the canary. If the canary performs correctly according to defined metrics, traffic gradually increases through stages until the new version receives all traffic. If issues emerge at any stage, traffic redirects back to the stable version and the canary is withdrawn.

Implementing canary deployments in Kubernetes requires traffic management capabilities beyond standard Deployment objects. Kubernetes does not natively provide fine-grained traffic splitting between versions. Organizations must leverage ingress controllers with traffic management features, service meshes like Istio or Linkerd that support weighted routing, or progressive delivery tools like Argo Rollouts and Flagger.

The primary advantage of canary deployments lies in controlled risk exposure. By limiting initial exposure to a small user percentage, issues affect only that subset, containing the blast radius of problematic releases. Canary deployments enable real-world

validation under genuine production conditions—actual users, realistic traffic patterns, production data, and live dependencies.

Additionally, canary releases support data-driven promotion decisions. Organizations define success criteria based on quantitative metrics—error rates below thresholds, latency percentiles within acceptable ranges, business KPIs meeting targets—and qualitative signals like user feedback. Automated analysis queries these metrics and automatically promotes the canary to the next stage if metrics remain healthy, or triggers rollback if they degrade.

However, canary deployments introduce significant operational complexity. Traffic management requires infrastructure beyond standard Kubernetes capabilities. Teams must configure traffic splitting rules, define promotion stages, establish metric thresholds for automatic promotion or rollback, and integrate with monitoring systems providing health signals.

Metric-based analysis demands careful design and tuning. Selecting appropriate metrics that reliably indicate new version health proves challenging. Error rates may not capture degraded user experience if the new version produces technically correct but confusing responses. Defining thresholds that catch real problems without triggering false-positive rollbacks requires tuning based on application characteristics and historical data.

The gradual rollout duration extends deployment timelines substantially compared to blue-green instant switches or rolling update speeds. A canary deployment with stages at 10%, 25%, 50%, 75%, and 100% with observation periods between stages may require hours or even days for complete rollout.

Standard Kubernetes Deployments do not support canary releases natively. Organizations implementing canary deployments typically adopt specialized tools. Argo Rollouts extends Kubernetes with Rollout custom resources defining canary stages, promotion logic, and metric queries. Flagger automates progressive delivery by integrating with Istio, Linkerd, or NGINX ingress controllers for traffic shifting and Prometheus for metrics analysis.

Despite their complexity, canary releases represent a powerful strategy for high-risk changes where thorough production validation outweighs deployment speed concerns and where the organization possesses the infrastructure, tooling, and expertise to implement them effectively.

2.1.4 Applicability to Infrastructure Upgrades

These deployment strategies, while effective for applications, prove inadequate for infrastructure component upgrades. Applications can leverage duplicate environments or gradual traffic shifts because they are stateless or maintain external state. Infrastructure components—Kubernetes control planes, network gateways, orchestration platforms—often cannot be fully duplicated without complex state migration mechanisms that don't currently exist.

Specifically, these strategies cannot address component dependency ordering where

upgrades must sequence in specific orders to maintain compatibility, CRD schema migrations requiring coordination between API server and controllers, network fabric transitions where stateful connections must re-establish during pod recreation, and multi-cluster coordination where distributed components must maintain connectivity across boundaries.

This gap motivates specialized approaches for infrastructure lifecycle management, particularly in multi-cluster contexts where coordination complexity compounds.

2.2 Kubernetes Operator Pattern for Lifecycle Management

The Kubernetes operator pattern extends the platform's control plane with application-specific operational knowledge, automating complex lifecycle management tasks that would otherwise require manual intervention [9]. Operators embody the convergence of Custom Resource Definitions, custom controllers, and domain expertise, transforming implicit human knowledge into explicit, repeatable automation integrated directly into Kubernetes infrastructure.

2.2.1 Operator Fundamentals

The operator pattern combines two Kubernetes concepts: Custom Resource Definitions (CRDs) and custom controllers. CRDs define new resource types beyond built-in objects like Pods and Services. These custom resources enable users to declare desired state for complex applications using familiar Kubernetes interfaces while encapsulating application-specific semantics.

Custom controllers implement the reconciliation logic that brings actual cluster state into alignment with desired state declared through custom resources. Controllers follow Kubernetes' fundamental control loop architecture: they continuously watch for changes to specific resources, compare current state against desired state, and execute actions to eliminate discrepancies. This reconciliation loop runs perpetually, ensuring the system maintains desired state even when external factors cause drift.

The operator pattern distinguishes itself from generic Kubernetes controllers through application-specific intelligence. While built-in controllers like Deployment and ReplicaSet implement general-purpose workload management, operators encode deep knowledge about specific applications. A MySQL operator understands replication topology, backup strategies, and failover procedures. This specialized knowledge enables operators to handle complex scenarios that generic controllers cannot address.

The reconciliation loop follows a level-triggered rather than edge-triggered design philosophy. Controllers don't react to specific events but instead periodically examine actual state and compare it to desired state. This design ensures eventual consistency—temporary failures, missed events, or network partitions don't permanently corrupt system state.

The Operator Framework and related tooling significantly reduce implementation

complexity. The Operator SDK provides scaffolding for building operators without requiring deep Kubernetes API expertise. Kubebuilder offers an alternative framework emphasizing code generation and integration with controller-runtime libraries. These frameworks generate boilerplate code for API definitions, webhook scaffolding, RBAC configurations, and deployment manifests.

2.2.2 Operator Maturity Levels

The Operator Capability Model defines five progressive maturity levels describing operator sophistication. Level 1 (Basic Install) represents automated application provisioning and configuration, replacing manual deployment procedures. Level 2 (Seamless Upgrades) adds automated patch and minor version updates with safe rollback capabilities. Level 3 (Full Lifecycle) provides complete lifecycle management including backup, recovery, and customization. Level 4 (Deep Insights) integrates monitoring, metrics collection, and automated remediation based on health signals. Level 5 (Auto Pilot) embodies advanced operational knowledge encoding, including performance tuning, capacity planning, and anomaly detection.

Most production operators achieve Level 2–3 capabilities, balancing automation benefits against development complexity. Level 4–5 require extensive domain knowledge, monitoring infrastructure integration, and machine learning capabilities.

2.2.3 Operators for Infrastructure Management

While operators excel at application lifecycle management, infrastructure-focused operators present unique challenges that differentiate them from application operators. Application operators manage workloads running atop Kubernetes; infrastructure operators manage Kubernetes itself or platforms extending Kubernetes with additional capabilities.

Cluster API represents the most prominent infrastructure operator, providing declarative APIs for provisioning, scaling, upgrading, and managing entire Kubernetes clusters [10]. However, CAPI focuses specifically on cluster provisioning and Kubernetes component upgrades—not overlay systems like Liqo.

Existing infrastructure operators typically address single-cluster scenarios. Multi-cluster platforms introduce qualitatively different challenges: ensuring compatible versions across connected clusters, maintaining connectivity during component transitions, coordinating upgrades without centralized control, and validating cross-cluster functionality throughout upgrade processes.

2.3 Multi-Cluster Management Approaches

Organizations deploy multiple Kubernetes clusters for various strategic and operational reasons: geographic distribution reducing latency, environment isolation separating development and production, resource segregation managing different teams, and multi-cloud strategies avoiding vendor lock-in [11]. Three primary approaches have emerged: centralized federation, service mesh-based connectivity, and peer-to-peer resource sharing.

2.3.1 Kubernetes Federation

Kubernetes Federation (KubeFed) provides a framework for coordinating configuration across multiple independent clusters through a centralized control plane [12]. The federation architecture consists of a host cluster containing federated resource definitions and policies, member clusters receiving propagated configurations, and a unified API enabling management of all clusters as a logical entity.

The federation approach centers on resource synchronization across clusters. Users define federated resources in the host cluster using standard Kubernetes APIs augmented with placement policies specifying target clusters. Federation controllers automatically propagate these resources to designated member clusters, maintaining synchronization as configurations change.

However, federation introduces significant operational complexity and maturity limitations. KubeFed development has slowed considerably over recent years, with many features remaining in beta status. The architecture requires exposing cluster API servers to the host cluster, establishing trust relationships, and managing RBAC policies across federation boundaries. Additionally, federation reduces isolation between clusters—control plane failures in the host cluster potentially affect all members.

Critically, federation provides no mechanisms for upgrading the federation infrastructure itself—updating KubeFed controllers, managing version compatibility between host and member clusters, or handling CRD migrations across the federation.

2.3.2 Service Mesh Multi-Cluster Support

Service meshes—including Istio, Linkerd, and Consul—extend their traffic management, security, and observability capabilities across cluster boundaries [13]. Unlike federation's control plane focus on configuration distribution, service meshes address data plane connectivity, establishing encrypted communication channels between clusters.

Istio supports multiple multi-cluster topologies. Multi-primary configurations deploy independent Istiod control planes in each cluster, with cross-cluster service discovery through shared trust domains. Primary-remote models centralize control plane management in primary clusters while remote clusters connect as data plane endpoints. East-west gateways handle cross-cluster traffic when clusters reside on different networks without direct pod-to-pod connectivity.

Linkerd implements multi-cluster through service mirroring combined with gateway-based connectivity. Service mirror controllers watch remote clusters for service updates, creating mirrored service representations locally. Link establishes secure tunnels between clusters using gateway components.

Service mesh multi-cluster capabilities provide fine-grained traffic control: weighted traffic distribution across clusters enabling canary deployments at cluster level, locality-aware load balancing preferring nearby cluster instances, and automatic failover redirecting traffic when cluster availability degrades.

However, service meshes introduce operational complexity beyond standalone cluster management. Teams must configure mesh peering relationships, manage gateway deployments, coordinate certificate authorities ensuring consistent trust domains, and

debug cross-cluster communication issues.

Critically, service meshes focus on data plane capabilities—enabling application communication across clusters—rather than control plane lifecycle management. While meshes facilitate cross-cluster connectivity, they don't address upgrading the mesh infrastructure itself while preserving application availability.

2.3.3 Liqo's Peer-to-Peer Model

Liqo distinguishes itself from federation and service mesh approaches through dynamic peer-to-peer cluster relationships enabling transparent workload offloading without centralized coordination [14]. Rather than federating all resources through a control plane or establishing service mesh infrastructure, Liqo creates virtual nodes in consumer clusters, abstracting provider cluster resources.

The peering model supports both unidirectional and bidirectional resource sharing relationships established dynamically. Consumer clusters authenticate with provider clusters using Kubernetes authentication mechanisms, negotiate resource allocations through ResourceOffer custom resources, and establish secure WireGuard VPN tunnels for pod-to-pod communication. Virtual nodes present remote capacity as native cluster resources—the Kubernetes scheduler remains completely unaware of cluster boundaries.

Liqo's namespace offloading mechanism enables selective multi-cluster deployment granular to namespace level. Users configure which namespaces participate in offloading through NamespaceOffloading resources. Offloaded services automatically become accessible across clusters through DNS synchronization.

Compared to federation and service meshes, Liqo provides lightweight multi-cluster integration focused specifically on workload distribution and resource sharing rather than comprehensive configuration management or advanced traffic control.

2.4 Existing Upgrade Tools and Automation

2.4.1 Manual Upgrade Procedures

Manual Kubernetes upgrades follow well-documented but operationally intensive procedures requiring careful planning, precise execution sequencing, and extensive testing [15]. The standard approach upgrades control plane components sequentially in dependency order—etcd first, then API server, controller manager, and scheduler—followed by node-by-node worker upgrades with workload migration.

Pre-upgrade preparation proves critical. Teams must review release notes identifying breaking API changes, deprecated features, and version compatibility requirements. Thorough testing in non-production environments validates application compatibility. Comprehensive backup procedures capture cluster state. Infrastructure capacity verification ensures surge node availability.

Manual upgrades offer maximum control and flexibility but impose significant operational burden scaling poorly with cluster fleet size. The time investment for manual upgrades scales linearly with cluster count, making this approach unsustainable for organizations managing dozens or hundreds of clusters.

For Liqo specifically, manual upgrades require coordinating CRD updates across potentially multiple peered clusters, managing control plane component transitions while maintaining cross-cluster connectivity, sequencing network fabric changes to minimize offloaded workload disruption, and validating ForeignCluster health throughout transitions. A single mistake can leave clusters in partially upgraded states difficult to diagnose and recover from.

2.4.2 GitOps-Based Approaches

GitOps methodologies apply declarative configuration management principles to Kubernetes cluster lifecycle, using Git repositories as the single source of truth for desired cluster state [16]. Tools like Argo CD and Flux CD continuously reconcile actual cluster state against Git-stored declarations.

GitOps approaches provide several compelling advantages. Complete audit trails capture who changed what and when through Git commit history. Collaborative workflows leverage pull request processes for peer review before changes reach clusters. Automated reconciliation ensures clusters continuously converge toward desired state. Disaster recovery simplifies to restoring Git repository access. Rollback operations become Git revert commits.

However, GitOps primarily addresses configuration management rather than Kubernetes version upgrades themselves. Argo CD and Flux CD excel at propagating manifest changes across clusters but provide limited native support for complex multi-stage upgrade workflows requiring component ordering, validation gates between stages, and sophisticated rollback coordination.

While GitOps can update Liqo deployment manifests to newer image tags, it cannot inherently manage the orchestration logic required for safe upgrades—draining virtual nodes, migrating offloaded workloads, upgrading components sequentially respecting dependencies, validating ForeignCluster health, and coordinating with peered clusters.

2.4.3 Specialized Lifecycle Operators

The Kubernetes community has developed specialized operators targeting cluster lifecycle management, with Cluster API (CAPI) representing the most prominent and mature approach [17]. CAPI provides declarative APIs for provisioning, scaling, upgrading, and managing complete Kubernetes clusters using Kubernetes-style custom resources and controllers.

The Cluster API architecture separates concerns through provider implementations enabling heterogeneous infrastructure support. Infrastructure providers handle cloud-specific or bare-metal resource provisioning. Bootstrap providers configure machines joining clusters. Control plane providers manage control plane component lifecycle.

CAPI-managed upgrades follow machine replacement strategies rather than in-place component updates. Control plane upgrades create new machines running target Kubernetes versions, migrate control plane components to new infrastructure, wait for health validation, then terminate old machines. Worker node upgrades similarly provision new

machines, leverage pod disruption budgets for graceful workload migration, verify new nodes report ready, then decommission old nodes.

Operator Lifecycle Manager (OLM) addresses the meta-problem of managing operators themselves [18]. OLM provides operator catalogs containing package definitions, version metadata, and dependency specifications. Dependency resolution ensures compatible operator versions coexist within clusters. Automated upgrade mechanisms leverage subscriptions and channels.

However, CAPI focuses specifically on cluster provisioning and Kubernetes component upgrades—not overlay systems like Liqo. CAPI's machine replacement strategies fundamentally assume creating new infrastructure for updated components, an approach that doesn't apply when upgrading software running on existing nodes without node turnover. Additionally, CAPI targets homogeneous multi-cluster management models where a single management cluster oversees many workload clusters, whereas Liqo's peer-to-peer model requires each cluster independently managing its Liqo components while coordinating with peer clusters.

2.5 Gap Analysis and Thesis Positioning

While existing solutions address subsets of challenges, significant gaps remain when considering specific requirements of Liqo multi-cluster upgrades.

Manual upgrade procedures provide control but fundamentally fail to scale. Upgrading Liqo across multiple peered clusters requires coordinating version changes to maintain compatibility, managing component dependencies, and ensuring continuous connectivity. Manual approaches demand operator expertise, introduce human error risks, and consume time scaling linearly with cluster count. **Gap:** Manual procedures lack automation mechanisms for coordinated multi-component upgrades preserving inter-cluster connectivity.

GitOps tools excel at declarative application deployment but operate at abstraction levels misaligned with infrastructure upgrade orchestration challenges. While GitOps can update Liqo deployment manifests, it cannot manage the domain-specific orchestration logic required for safe upgrades. **Gap:** GitOps focuses on deployment automation rather than stateful upgrade orchestration requiring domain-specific sequencing and validation logic.

Kubernetes Federation addresses multi-cluster configuration distribution but not lifecycle management of the federation infrastructure itself. Moreover, federation's centralized architecture contradicts Liqo's peer-to-peer model. **Gap:** Federation tooling addresses configuration management, not the meta-problem of upgrading federation infrastructure itself while preserving peer relationships.

Service meshes provide multi-cluster connectivity and traffic management but focus on data plane capabilities rather than control plane lifecycle. Service mesh upgrade strategies don't translate to Liqo's unique architectural combination of virtual node abstraction, workload offloading, and bilateral peering with stateful tunnel connections. **Gap:** Service mesh upgrade strategies don't apply to Liqo's unique architecture.

Cluster API represents the closest existing approach to automated infrastructure lifecycle management. However, CAPI focuses specifically on cluster provisioning and Kubernetes components—not overlay systems like Liqo. CAPI's machine replacement strategies don't apply when upgrading software atop existing clusters. Additionally, CAPI targets homogeneous multi-cluster management where a single management cluster oversees many workload clusters through consistent APIs and centralized control, whereas Liqo's peer-to-peer model requires each cluster independently managing its Liqo installation while coordinating with peers about version compatibility and upgrade timing without centralized orchestration. **Gap:** CAPI addresses cluster provisioning and Kubernetes upgrades, not upgrades of multi-cluster overlay systems with bilateral peer dependencies and decentralized coordination requirements.

Operator Lifecycle Manager provides operator upgrade automation through catalogs, dependency resolution, and subscription-based update mechanisms. While OLM could theoretically manage Liqo operator version updates within single clusters, it lacks mechanisms for inter-cluster coordination required across peered Liqo clusters. **Gap:** OLM automates single-cluster operator lifecycle, not multi-cluster coordination scenarios where upgrade sequencing affects inter-cluster relationships and shared state.

Liqo itself currently lacks automated upgrade capabilities [19]. The Liqo FAQ explicitly acknowledges this limitation, stating that upgrading through liqoctl install or Helm chart updates changes Docker images for Liqo components but doesn't handle CRD schema changes or controller logic updates safely. The recommended approach for major version upgrades involves unpeering all clusters terminating cross-cluster relationships, uninstalling Liqo completely from all clusters, installing new Liqo versions from scratch, and manually re-establishing peering relationships. This disruptive procedure causes significant downtime affecting production workloads. **Gap:** No native Liqo upgrade mechanism exists enabling seamless version updates while preserving offloaded workloads and maintaining inter-cluster connectivity.

Canary Upgrade Status: While canary deployment patterns are well-established for application updates, applying them to infrastructure upgrades—particularly in multi-cluster contexts—remains an open challenge. This thesis implements partial canary functionality in the codebase to demonstrate feasibility, but full production-ready canary upgrades for Liqo are deferred to future work due to additional infrastructure requirements for traffic splitting and gradual rollout orchestration across peered clusters.

Thesis Contribution: The identified gaps motivate this thesis's core contribution: a Liqo Upgrade Operator implementing automated, coordinated upgrades across Liqo's multi-component architecture while minimizing disruption to multi-cluster topologies and offloaded workloads. The operator addresses these gaps by providing declarative upgrade workflows expressed through Kubernetes custom resources, implementing component ordering that respects Liqo's architectural dependencies (CRDs before controllers, control plane before network fabric), preserving inter-cluster connectivity through careful upgrade sequencing and validation gates, and demonstrating feasibility through proof-of-concept implementation and experimental validation.

Subsequent chapters detail the operator's architecture, implementation approach, experimental validation methodology, and results, positioning this work as addressing a gap

unmet by existing multi-cluster management and upgrade automation solutions.

Thesis Contributions

This work contributes the following novel elements addressing the identified gaps:

1. **Multi-Stage Infrastructure Upgrade Architecture:** A three-stage workflow (CRD migration, control plane upgrade, network fabric upgrade) that sequences component updates according to architectural dependencies while incorporating health validation gates between stages. Unlike application-focused deployment strategies (rolling updates, blue-green, canary), this architecture addresses infrastructure-specific challenges including stateful network component transitions and cross-layer dependency management.

2. **Job-Based Execution Model with Cumulative Rollback:** A separation of orchestration logic (in the controller) from modification operations (in Kubernetes Jobs) that provides fault isolation, observable execution through persistent Job logs, and automatic retry handling. The cumulative rollback strategy restores all components upgraded prior to failure points, ensuring version consistency across the entire stack rather than creating partially-upgraded states.

3. **Distributed Compatibility Validation for Peer-to-Peer Multi-Cluster:** A coordination approach where independent operator instances in each cluster validate version compatibility through ForeignCluster resource inspection and embedded compatibility matrices, enabling decentralized upgrade decisions without centralized orchestration or explicit inter-operator communication. This model suits Liqo's peer-to-peer architecture better than federation-style centralized control.

4. **Parallel Gateway Upgrade Orchestration:** A deliberate design decision to upgrade gateway components simultaneously across peerings rather than sequentially, minimizing total downtime by coordinating disruption windows. The analysis demonstrates that for infrastructure components with unavoidable per-upgrade downtime, parallel strategies outperform canary approaches by avoiding cumulative disruption multiplication.

5. **Empirical Characterization of Platform-Inherent Downtime:** Experimental identification and quantification of downtime root causes (Linux conntrack flush, Kubernetes reconciliation delays, Liqo IP synchronization, WireGuard tunnel re-establishment) establishing that 14–28 seconds of disruption during network fabric upgrades represents minimal rather than suboptimal downtime given current platform primitives. This characterization provides realistic expectations for multi-cluster networking infrastructure upgrades.

6. **Configuration Change Semantic Gap Analysis:** Recognition that environment variable and CRD schema changes require maintainer-provided semantic knowledge that automated tools cannot infer through structural comparison alone. The current manual specification approach and proposed future work (maintainer-provided

descriptor files) address this fundamental limitation in automated infrastructure upgrade systems.

These contributions collectively demonstrate that operator-based automation can successfully orchestrate complex multi-stage infrastructure upgrades while achieving minimal service disruption within platform-imposed constraints.

Chapter 3

Problem Statement

3.1 Defining Seamless Updates

The term "seamless update" requires careful definition in cloud-native environments. While zero-downtime deployment remains ideal, practical systems pursue minimal-downtime updates balancing availability with operational realities. In cloud-native contexts, "seamless" typically implies minimal disruption rather than absolute zero downtime—updates should appear seamless from users' perspective, even if brief interruptions occur at infrastructure level.

A seamless update encompasses several requirements: continuous health validation verifying component functionality before proceeding, automated rollback capabilities when problems are detected, observability enabling operators to understand upgrade progress, and workload impact minimization. In multi-cluster platforms like Liqo, updates must preserve peering relationships enabling resource sharing between clusters.

This thesis adopts a pragmatic definition: updates that complete reliably with minimal, predictable service disruption; provide comprehensive health validation and automatic rollback; maintain clear visibility; and preserve critical infrastructure relationships.

3.2 The Multi-Cluster Upgrade Problem

When organizations deploy multiple clusters, upgrade challenges compound exponentially. The multi-cluster upgrade problem encompasses coordinating updates across independent control planes, managing state synchronization where clusters share resources, handling dependency management across clusters, preserving network connectivity during component updates, and dealing with configuration drift as clusters evolve independently.

The upgrade sequencing problem requires determining which clusters upgrade first, whether upgrades proceed in parallel or sequentially, and in what order. Network connectivity preservation emerges as critical—clusters connected through VPN tunnels or networking infrastructure depend on connections remaining stable. Without automation, cognitive load of tracking upgrade status across many clusters exceeds human capability.

3.3 Liqo-Specific Challenges

Liqo introduces distinctive challenges through its architecture. At Liqo's core lies virtual node abstraction—each peering spawns a virtual kubelet creating a virtual node representing remote cluster resources. During upgrades, virtual kubelet must maintain connectivity to both clusters simultaneously.

Component dependencies follow strict hierarchy. CRDs define schemas for ForeignCluster, VirtualNode, ShadowPod resources and must upgrade before control-plane components managing them. This dependency ordering cannot be dynamically discovered—Liqo's architecture does not expose dependency relationships through any API. Required sequencing derives from understanding how components interact architecturally: which components create or modify which resource types, which depend on specific CRD schemas, which must be operational for others to function. This architectural knowledge must be explicitly encoded—it cannot be inferred at runtime. The hardcoded component upgrade order reflects this architectural reality.

The network fabric presents acute challenges. Liqo establishes secure WireGuard VPN tunnels between clusters managed by gateway components. When gateway components upgrade, network fabric experiences disruption. This minimal downtime is not operator implementation deficiency but inherent characteristic of Kubernetes pod lifecycle management and Liqo's tunnel architecture. Kubernetes pod recreation provides no atomic handoff of network state. Liqo's WireGuard tunnels are stateful connections tied to specific pod instances; when gateway pods restart, clients and servers must negotiate new tunnels, perform cryptographic handshakes, establish routes—requiring several seconds even under optimal conditions.

Environment variable changes and CRD schema migrations present additional challenge: detecting what changed requires understanding semantic meaning, not just structural differences. This knowledge is embedded in the Liqo source code itself, deriving from the component's implementation logic. The operator can detect that environment variable was added or CRD field changed type, but cannot automatically determine whether changes are compatible or require specific sequencing. This limitation requires manual effort—currently addressed through configuration files explicitly specifying version-specific changes, with future work planned to move configuration into maintainer-provided descriptor files.

Peering relationships must remain stable. Liqo tracks peering status through ForeignCluster resources monitoring API server connectivity and network tunnel health. During upgrades, if health checks fail temporarily, Liqo may mark virtual nodes as NotReady, triggering pod eviction. Gateway transition orchestration represents most visible downtime source—when components restart, WireGuard tunnels disconnect and must re-establish.

Upgrades must sequence CRD updates before control plane updates, coordinate gateway transitions to minimize network downtime, preserve virtual node availability, maintain peering health checks, and account for configuration variability—all while providing rollback capability if any stage fails.

Environment variable changes and CRD schema migrations present an additional challenge that transcends automated detection: understanding semantic meaning requires

domain knowledge that is embedded in the Liqo source code itself. When a new Liqo version adds an environment variable to a component deployment, removes a deprecated command-line argument, or modifies a CRD field's type, automated tools can detect that structural changes occurred but cannot determine whether these changes are required for correct operation, optional for new features, or conditionally necessary based on deployment configuration.

The operator can detect that an environment variable was added—comparing manifests reveals the structural difference. However, the operator cannot automatically determine:

- Whether the new variable is required for basic functionality or only for optional features
- Whether the variable's absence causes immediate failure or subtle degradation
- Whether the variable's default value (if unset) matches the behavior of the previous version
- Whether setting the variable requires coordinated changes in other components

These semantics are embedded in the Liqo source code, deriving from the component's implementation logic rather than from the manifest structure. A new environment variable enabling an experimental feature has entirely different implications than one required for compatibility with the new CRD schema, yet both appear identically in manifests as added key-value pairs.

Similarly, CRD schema changes exhibit semantic complexity beyond structural differences. When a CRD field changes from a string to an object type, automated tools detect the type modification but cannot determine whether existing resources using the old schema remain valid, require migration through conversion webhooks, or become incompatible requiring recreation. The schema evolution strategy—whether backward compatible, requiring explicit migration, or breaking—is a design decision documented (if at all) in release notes rather than encoded in machine-readable formats.

This semantic gap necessitates manual effort in the current operator implementation. Configuration changes are specified explicitly in upgrade plans rather than automatically inferred, with version-specific change knowledge encoded in the operator's configuration logic. This limitation is not an implementation deficiency but reflects that semantic meaning cannot be mechanically derived from structural comparison.

Future work will address this through maintainer-provided descriptor files where Liqo releases include structured metadata explicitly declaring which environment variables are required versus optional, which CRD changes need migration logic, and which configuration modifications are backward compatible. This approach shifts the specification burden to the Liqo community, who would provide machine-readable documents alongside releases, enabling the operator to remain version-agnostic in its core logic.

3.4 Research Questions

RQ1: Can the Kubernetes operator pattern effectively automate complex, multi-stage infrastructure upgrades in multi-cluster environments?

RQ2: What orchestration strategies minimize service disruption during multi-cluster platform upgrades?

RQ3: How can automation reliably detect failures and perform rollback in distributed multi-cluster systems?

RQ4: To what extent can upgrade automation reduce operational complexity and human error compared to manual procedures?

RQ5: What configuration changes between versions can be automated, and which require human semantic understanding that precludes full automation?

3.5 Success Criteria and Scope

Success criteria: functional completeness (upgrading without manual intervention), reliability (handling configurations and failure scenarios through automatic rollback), service disruption minimization (significantly less downtime than manual procedures with predictable duration), operational simplification (reducing expertise required), and observability (clear visibility into progress).

Scope – Included: Patch version upgrades within same minor version family. Three-stage upgrades: CRD migration, control plane updates, network/data plane updates. Each stage includes health validation and rollback support.

Scope – Excluded: Major and minor version upgrades introducing API changes. Production-ready canary upgrade strategies, while partially implemented to demonstrate feasibility, are not fully developed or validated—canary releases require additional infrastructure for traffic splitting and gradual rollout exceeding thesis timeline. Zero-downtime guarantees are not promised—brief network interruptions during gateway transitions are accepted as inherent limitation. Focus is making downtime minimal and predictable rather than eliminating it entirely.

Chapter 4

Background and Technologies

4.1 Kubernetes Architecture Essentials

Kubernetes separates cluster management (control plane) from workload execution (data plane). This separation enables horizontal scaling and fault tolerance but complicates upgrade procedures because components must coordinate across this boundary [20].

The architecture follows a declarative model where users specify desired state through API objects, and the system continuously reconciles actual state to match declarations. This reconciliation loop forms the foundation for operator patterns and automated lifecycle management. Control plane components make scheduling and policy decisions, while data plane components execute workloads and handle runtime concerns.

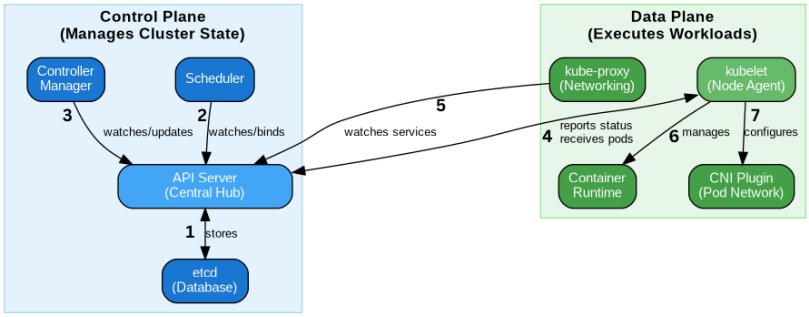

Figure 4.1. Kubernetes control plane and data plane components.

4.1.1 Control Plane Components

The Kubernetes control plane has several components that manage cluster state and schedule workloads. Their interdependencies determine upgrade sequencing constraints.

The API server (`kube-apiserver`) serves as the central point of interaction for all cluster operations. It exposes the Kubernetes REST API, validates and processes API requests, and persists cluster state to etcd. All cluster components communicate exclusively through the API server rather than directly with each other, establishing it as the single source of truth for cluster state. The API server implements authentication, authorization, and admission control, ensuring only valid, permitted operations modify cluster state. During upgrades, the API server must maintain backward compatibility with clients and other control plane components to prevent disruption.

The etcd datastore provides consistent, distributed key-value storage for all cluster state. Implemented as a separate project, etcd uses the Raft consensus algorithm to ensure consistency across multiple replicas. Kubernetes stores all API objects—pods, services, deployments, custom resources—in etcd as serialized JSON or protobuf. The datastore forms the persistence layer, making its availability and consistency paramount for cluster operation. Etcd version compatibility with the API server constrains upgrade paths, as mismatched versions may not properly serialize or deserialize certain object types.

The scheduler (`kube-scheduler`) assigns newly created pods to nodes based on resource requirements, constraints, affinity rules, and quality-of-service classes. The scheduler watches the API server for unscheduled pods, evaluates candidate nodes through filtering and scoring phases, and binds pods to selected nodes by creating binding objects. In multi-cluster environments using platforms like Liqo, the scheduler remains unaware of cluster boundaries, treating virtual nodes representing remote clusters identically to physical nodes.

The controller manager (`kube-controller-manager`) runs core control loops that regulate cluster state. Each controller watches specific resource types, compares current state against desired state, and executes actions to eliminate discrepancies. Built-in controllers manage node lifecycle, replication, endpoints, service accounts, and namespaces. The controller manager implements the reconciliation pattern central to Kubernetes' self-healing capabilities.

These control plane components exhibit version skew tolerance defined by Kubernetes version skew policies. The API server must be at the same or newer version than other control plane components, while controllers and scheduler may lag one minor version. This tolerance enables rolling control plane upgrades but requires careful sequencing to avoid incompatibilities.

4.1.2 Data Plane Components

The data plane executes containerized workloads and provides runtime services including networking, storage, and monitoring. Data plane components run on every worker node, managing the local execution environment and reporting status to the control plane.

The kubelet serves as the primary node agent, responsible for pod lifecycle management on its host node. The kubelet watches the API server for pods assigned to its node, ensures containers start and remain healthy according to pod specifications, and reports pod status back to the control plane. It manages container runtime interactions through the Container Runtime Interface (CRI), handles volume mounting, executes liveness and

readiness probes, and enforces resource limits.

The container runtime manages container lifecycle operations including image pulling, container creation, execution, and termination. Kubernetes supports multiple container runtimes through the Container Runtime Interface, including containerd, CRI-O, and Docker. The runtime isolates containers using Linux namespaces and cgroups, enforces resource limits, and provides container-to-container networking within pods.

The `kube-proxy` implements the Kubernetes Service abstraction by maintaining network rules that enable pod-to-service and external-to-service communication. Running as a DaemonSet on every node, kube-proxy watches the API server for Service and Endpoint objects and configures iptables, IPVS, or eBPF rules to route traffic appropriately.

The Container Network Interface (CNI) plugin provides pod networking by implementing IP address management, network namespace setup, and inter-pod communication. Different CNI plugins offer various features and performance characteristics—Calico provides network policy enforcement, Cilium uses eBPF for high performance, and Flannel offers simplicity.

4.1.3 Custom Resource Definitions

Custom Resource Definitions (CRDs) extend the Kubernetes API with application-specific resource types, enabling operators and custom controllers to manage applications using Kubernetes-native patterns. CRDs transform Kubernetes from a container orchestration platform into a general-purpose declarative framework for managing arbitrary systems. Understanding CRD mechanics proves essential for operator development and comprehending upgrade challenges when custom resources evolve.

A CRD defines the schema, validation rules, and storage configuration for a custom resource type. When installed, the CRD instructs the API server to accept, validate, and persist instances of the custom resource alongside built-in resources like Pods and Services. The schema section defines the structure of the custom resource's specification and status using OpenAPI v3 schema validation, enabling the API server to reject malformed resources.

CRDs support multiple versions simultaneously, enabling schema evolution without breaking existing resources. Version conversion webhooks translate resources between schema versions, allowing old clients to interact with resources using older schemas while new controllers operate on current schemas. This versioning capability proves critical during upgrades when operators and custom resources must maintain compatibility across version boundaries.

Custom resources persist in etcd identically to built-in resources, inheriting all API server features including watch semantics, field selectors, and label selectors. However, this persistence model requires CRD upgrades to handle stored resource migration when schemas change incompatibly. Kubernetes provides no automatic migration mechanism, forcing operators to implement conversion logic or accept breaking changes.

CRD lifecycle management introduces specific upgrade challenges. Controllers built against one CRD version may malfunction when the CRD upgrades to introduce schema changes. Installing new CRD versions before upgrading controllers risks controllers encountering unrecognized fields or validation failures. Conversely, upgrading controllers

before CRDs may cause controllers to create resources using schemas the API server rejects. The Liqo upgrade operator must sequence CRD updates before control plane upgrades precisely because of these dependencies, ensuring controllers always operate against compatible schemas.

4.2 Liqo Architecture and Components

Liqo uses a peer-to-peer architecture for multi-cluster resource sharing. Unlike centralized federation or service meshes, Liqo extends Kubernetes scheduling through virtual nodes—making remote cluster resources appear as local capacity [21].

This design prioritizes simplicity and transparency. Clusters establish bilateral peering relationships without requiring centralized coordination infrastructure. Applications schedule onto virtual nodes using standard Kubernetes APIs, with Liqo transparently executing pods on remote clusters. Network connectivity operates through secure VPN tunnels, enabling pod-to-pod communication across cluster boundaries as if pods reside in a single extended network.

4.2.1 Peering Model

Liqo's peering mechanism enables two Kubernetes clusters to discover each other and establish resource-sharing relationships. Peering operates bidirectionally, with each cluster independently deciding whether to consume resources from or provide resources to its peer. This flexibility supports various topologies including unidirectional consumer-provider relationships, bidirectional resource exchange, and multi-peer scenarios where clusters peer with multiple partners simultaneously.

The peering process begins with discovery, where one cluster obtains the peer cluster's connection information including API server endpoint, authentication credentials, and network configuration. Liqo supports multiple discovery mechanisms including manual configuration, DNS-based discovery, and LAN discovery. Once discovery completes, clusters exchange authentication tokens to establish mutual trust. The consumer cluster creates a ForeignCluster custom resource representing the peer, storing connection parameters and peering configuration.

After authentication, clusters negotiate resource sharing through ResourceOffer custom resources. The provider cluster advertises available resources—CPU, memory, and pod capacity—that it makes available to the consumer. The consumer evaluates offers and accepts those meeting its requirements, creating VirtualNode resources that represent accepted capacity.

Peering relationships persist across cluster restarts and network interruptions through state stored in custom resources. The Liqo controller continuously monitors peering health, verifying API server connectivity and network tunnel status. If connectivity degrades, Liqo marks virtual nodes as NotReady, preventing new pod scheduling while preserving existing offloaded workloads. Health monitoring proves critical during upgrades when components restart temporarily, requiring careful management to avoid unnecessary pod evictions.

The peering model's bilateral nature complicates multi-cluster upgrades. Each cluster maintains independent state about its peers, requiring coordinated upgrades to preserve relationships. Upgrading one cluster in a peering relationship while the peer remains at an older version introduces version skew that may cause protocol incompatibilities or resource synchronization failures.

4.2.2 Virtual Nodes and Workload Offloading

Virtual nodes represent the core abstraction enabling transparent multi-cluster scheduling in Liqo. When a consumer cluster accepts a ResourceOffer from a provider, Liqo creates a VirtualNode resource that appears to the Kubernetes scheduler as a regular node with available capacity. The scheduler treats virtual nodes identically to physical nodes, enabling seamless workload distribution without application modifications.

The Virtual Kubelet component implements the virtual node abstraction. Each peering relationship spawns a virtual kubelet instance that registers a virtual node with the local API server and watches for pods scheduled onto that node. When the scheduler assigns a pod to the virtual node, the virtual kubelet offloads the pod to the remote cluster by creating a corresponding ShadowPod resource in the provider cluster. The ShadowPod specification mirrors the original pod's configuration, with the provider cluster's control plane handling actual pod execution.

Namespace offloading enables selective multi-cluster deployment at namespace granularity. Users configure which namespaces participate in offloading through NamespaceOffloading resources. When a namespace is offloaded, Liqo creates a twin namespace in the remote cluster and automatically reflects necessary resources including ConfigMaps, Secrets, and Services. This reflection ensures offloaded pods access required configuration and maintain service connectivity.

The offloading mechanism preserves Kubernetes semantics, making multi-cluster operation transparent to applications. Applications continue using standard Kubernetes service discovery through DNS, with Liqo synchronizing service endpoints across clusters. Offloaded pods access local cluster services through the network fabric, and services in the local cluster can expose endpoints from offloaded pods, creating a unified service mesh spanning both clusters.

4.2.3 Network Fabric

Liqo's network fabric provides pod-to-pod and pod-to-service connectivity across cluster boundaries, enabling transparent communication between local and remote workloads. The fabric establishes secure VPN tunnels using WireGuard, encapsulating cross-cluster traffic and routing it through gateway components deployed in each cluster.

Gateway components operate in client-server pairs, with the consumer cluster running a gateway client and the provider cluster running a gateway server. The gateway client initiates the WireGuard connection to the gateway server, establishing an encrypted tunnel through which all cross-cluster pod traffic flows. Both gateways perform network address translation (NAT) to prevent IP address conflicts between clusters' pod CIDR ranges.

The Network Manager component coordinates IP address management (IPAM) across clusters, remapping pod and service IPs to ensure uniqueness in the extended network. When a pod offloads to a remote cluster, the Network Manager allocates a unique external IP from a reserved range and configures routing rules directing traffic to that IP through the VPN tunnel. This remapping occurs transparently—pods use their original cluster-local IP addresses, with gateways performing translation at tunnel boundaries.

High availability configurations deploy multiple gateway replicas operating in active-passive mode. At any time, one gateway actively handles traffic while others remain standby, ready to take over if the active gateway fails. The failover mechanism relies on Kubernetes Services and leader election to ensure smooth transitions. However, WireGuard tunnels are stateful connections tied to specific gateway pod instances. When a gateway pod restarts or fails over during upgrades, clients and servers must renegotiate the tunnel, perform cryptographic handshakes, and reestablish routes—a process requiring several seconds even under optimal conditions. This characteristic contributes to the minimal downtime experienced during network fabric upgrades.

4.2.4 Control Plane Components

Liqo's control plane extends Kubernetes with specialized controllers managing multi-cluster orchestration. These controllers implement Liqo's core logic including peering lifecycle, resource offloading, network configuration, and cross-cluster state synchronization.

The Controller Manager embeds Liqo's primary control loops, including the ForeignCluster controller managing peering lifecycle, the NamespaceOffloading controller handling namespace reflection, and the ResourceOffer controller negotiating resource sharing. Each controller follows the standard Kubernetes reconciliation pattern, watching for changes to specific custom resources and executing actions to align actual state with desired state. The controller manager's health directly impacts Liqo functionality—if controllers fail during upgrades, peering relationships may degrade, and offloaded workloads may become inaccessible.

The Identity Manager handles cross-cluster authentication and credential management. When clusters peer, the Identity Manager exchanges authentication tokens, manages certificate rotation, and maintains trust relationships. During upgrades, the Identity Manager must preserve authentication state to prevent disconnection of established peerings.

The CRD Replicator synchronizes Custom Resource Definitions between peered clusters, ensuring both clusters recognize the same resource types. This synchronization enables offloaded applications using custom resources to function correctly in remote clusters. CRD Replicator upgrades must sequence carefully with CRD updates to avoid schema mismatches between clusters.

The Webhook component validates Liqo custom resources before the API server persists them, enforcing consistency constraints and preventing invalid configurations. The

webhook implements admission control logic specific to Liqo resources, rejecting malformed ForeignCluster specifications, invalid NamespaceOffloading configurations, or incompatible ResourceOffer parameters. Webhook availability proves critical—if the webhook fails during upgrades while configured as a required admission controller, cluster operations may block until it restores.

4.2.5 CRDs and Dependencies

Liqo defines several custom resources extending Kubernetes with multi-cluster capabilities. Understanding these CRDs illuminates the data model underlying Liqo's operations and the dependencies that constrain upgrade sequencing.

ForeignCluster resources represent peer clusters. Each ForeignCluster stores the peer's API server endpoint, authentication credentials, network configuration, and peering status. The ForeignCluster spec defines the peering configuration including whether the local cluster consumes or provides resources, resource quotas, and network settings. The status section reflects the current peering state including connectivity health, authentication status, and resource allocation.

VirtualNode resources represent remote cluster capacity in the local cluster. Each VirtualNode corresponds to an accepted ResourceOffer and includes capacity information (CPU, memory, pods), node conditions (Ready, DiskPressure, MemoryPressure), and provider cluster identity. The Kubernetes scheduler treats VirtualNodes as regular nodes, considering their capacity and conditions when making scheduling decisions.

ResourceOffer resources advertise available capacity from provider to consumer clusters. Providers create ResourceOffers specifying CPU, memory, and pod quotas they make available. Consumers accept offers by creating VirtualNode resources referencing the offer. The offer lifecycle progresses through pending, accepted, and expired states tracked in the status section.

ShadowPod resources represent offloaded pods in provider clusters. When a pod schedules onto a VirtualNode, the virtual kubelet creates a ShadowPod in the remote cluster mirroring the original pod's specification. The ShadowPod's status reflects the actual pod's execution state, with the virtual kubelet synchronizing status updates back to the original pod in the consumer cluster.

NamespaceOffloading resources configure which namespaces participate in workload offloading. The spec defines offloading policies including which clusters can receive offloaded workloads, resource quotas for offloaded pods, and pod selectors filtering which pods offload. The status tracks the twin namespace creation in remote clusters and resource reflection status.

These CRDs form a dependency hierarchy with specific ordering constraints. ForeignCluster resources must exist before ResourceOffers and VirtualNodes can reference peer clusters. NamespaceOffloading resources depend on established peering relationships represented by ForeignCluster resources. ShadowPods require VirtualNodes and twin namespaces to exist before pods can offload. During upgrades, CRD schema changes must propagate in dependency order to maintain consistency—upgrading CRDs defining depended-upon resources before CRDs defining dependent resources prevents validation failures and broken references.

Understanding Liqo's component dependencies proves essential for comprehending why upgrade sequencing must be explicitly encoded rather than dynamically discovered. Unlike some Kubernetes extensions that expose dependency graphs through metadata or APIs, Liqo's architecture does not provide runtime mechanisms to query which components depend on which resources or what order updates must follow.

The dependency relationships exist at the architectural level, deriving from how components interact rather than being declared in machine-readable formats. CRDs must upgrade before controllers because controllers instantiate resources conforming to CRD schemas—if a controller expects a CRD field that doesn't exist in the current schema, reconciliation fails. The controller-manager must upgrade before network fabric components because the controller reconciles InternalNode status that fabric components consume for routing decisions—if fabric upgrades while the controller runs old code, status fields may contain incompatible data structures.

Liqo provides no discovery API exposing these dependencies. The architecture includes no metadata annotations declaring "component A depends on component B" or "CRD X must exist before controller Y starts." Attempting to infer dependencies through static analysis of manifests would prove unreliable—YAML files describe deployment configurations but not runtime behavioral dependencies like "this controller writes status fields that component reads."

The hardcoded component upgrade order in the operator implementation reflects this architectural reality rather than representing an implementation shortcoming. The sequence—CRDs first, then controller-manager, then network fabric components—encodes knowledge about which components create or modify which resource types, which depend on specific CRD schemas being available, and which must be operational for others to function correctly. This knowledge derives from understanding Liqo's source code and operational behavior, not from introspecting cluster state or querying APIs at upgrade time.

Alternative approaches might include requiring the Liqo community to provide machine-readable dependency metadata alongside releases, similar to how package managers use dependency declarations. However, even with such metadata, the fundamental ordering constraints remain—CRDs before controllers, control plane before data plane—because these constraints derive from Kubernetes' architectural model itself, not Liqo-specific design choices.

The operator's approach accepts that upgrade orchestration requires domain-specific knowledge that cannot be automatically derived. This contrasts with general-purpose tools like Helm or kubectl apply that operate on resource manifests without understanding semantic dependencies. An upgrade operator necessarily encodes operational knowledge—the distinction lies in whether this knowledge resides in imperative scripts operators execute manually or in declarative automation that operators invoke through custom resources.

Chapter 5

General System Architecture

5.1 Design Principles and Requirements

The design of the Liqo Upgrade Operator is guided by four foundational principles that address the specific challenges of upgrading multi-cluster networking infrastructure. These principles emerged from analyzing pain points identified in manual upgrade procedures and requirements for maintaining service continuity across distributed cluster topologies. The operator's architecture reflects a balance between automation, safety, and operational practicality, recognizing that Liqo's role as critical networking infrastructure demands particular attention to failure handling and state preservation [22].

5.1.1 Automation and Declarative Management

The operator adopts a fully declarative approach where users specify their desired target version through a Kubernetes custom resource rather than executing imperative upgrade commands. This design choice aligns with Kubernetes-native operational patterns and enables version control of upgrade specifications alongside other cluster configuration artifacts. The LiqoUpgrade custom resource serves as the single source of truth for upgrade intent, capturing not only the target version but also operational parameters such as namespace scope, automatic rollback preferences, and dry-run mode for validation without execution.

The reconciliation loop implements continuous convergence toward the declared target state, automatically retrying transient failures and maintaining idempotency across reconciliation cycles. This reconciliation-based approach eliminates the need for operators to maintain long-running connections or monitor upgrade progress actively, as the controller persistently works toward the desired state regardless of temporary disruptions. The declarative model also enables integration with GitOps workflows, where upgrade specifications can be committed to version control repositories and applied through continuous delivery pipelines.

5.1.2 Minimal Downtime Objective

The operator's architecture prioritizes minimizing service disruption during upgrades, though achieving true zero downtime remains beyond the current implementation scope. The design recognizes that Liqo's network fabric components require brief periods of unavailability during version transitions, particularly when gateway pods restart with new container images. The minimal downtime objective manifests through a three-stage upgrade workflow that sequences component updates to limit concurrent disruptions and implements gateway high availability configurations during the network fabric upgrade phase.

The staged approach upgrades custom resource definitions first, followed by control plane components, and finally network and data plane elements. This sequencing ensures that API schema changes are established before components attempt to use new CRD versions, preventing reconciliation failures due to schema mismatches. During the network fabric upgrade, the operator temporarily configures gateway deployments with two replicas and rolling update strategies, enabling Kubernetes to maintain at least one healthy gateway pod during image transitions. While this reduces downtime compared to recreate strategies that terminate all pods before creating replacements, brief connectivity interruptions remain inevitable during leader election transitions between old and new gateway pods. The minimal downtime achieved represents a practical compromise between upgrade speed and service continuity, with typical disruption windows measured in seconds rather than minutes.

5.1.3 Safety and Reliability

Safety mechanisms pervade the operator's design to mitigate the inherent risks of automated infrastructure upgrades. The architecture implements multiple validation gates, comprehensive health checking, and automatic rollback capabilities to prevent failed upgrades from leaving clusters in inconsistent or non-functional states. Pre-upgrade validation examines cluster health, verifies version compatibility across peered clusters, and ensures that required components exist before initiating any modifications. Each upgrade stage incorporates health gates that verify successful completion before proceeding, preventing cascading failures from propagating through subsequent stages.

The rollback mechanism employs a snapshot-and-restore approach where the operator captures the complete pre-upgrade configuration including component images, environment variables, command-line arguments, and custom resource specifications. Upon detecting stage failures through job monitoring, the operator automatically initiates cumulative rollback procedures that restore not only the failed stage but all previously upgraded components to their original configurations. This cumulative strategy ensures consistent version alignment across all Liqo components rather than leaving clusters with partially upgraded component sets. The automatic rollback behavior can be disabled through the LiqoUpgrade specification when operators prefer to investigate failures manually, providing flexibility for debugging while maintaining safety as the default operating mode.

5.1.4 Multi-Cluster Awareness

The operator's architecture acknowledges Liqo's inherently multi-cluster nature where peering relationships create upgrade coordination requirements absent in single-cluster systems. Rather than implementing centralized orchestration, the design adopts a distributed model where each cluster runs its own operator instance that coordinates implicitly through compatibility validation and simultaneous gateway upgrade approaches. This distributed architecture avoids single points of failure and simplifies operational deployment by treating each cluster autonomously while still maintaining coordination through shared compatibility constraints.

The compatibility validation mechanism queries ForeignCluster resources to determine peer cluster versions, consults a compatibility matrix embedded in the operator configuration, and rejects upgrade attempts that would create incompatible version combinations. This validation approach enables decentralized upgrade coordination where clusters independently decide upgrade timing while preventing configurations that would break peering connectivity. However, the current implementation does not support sophisticated canary deployment strategies for gradually rolling out upgrades across multiple peered clusters. While canary logic exists in the codebase for future implementation, the current validated approach prioritizes upgrade speed and operational simplicity over gradual rollout complexity, acknowledging this as a constraint suitable for environments where brief network disruptions are acceptable during maintenance windows.

5.2 High-Level Architecture

The Liqo Upgrade Operator architecture embodies a separation of concerns between declarative intent specification, control logic orchestration, and execution mechanics. This architectural design aligns with established Kubernetes extension patterns while addressing the specific complexities of multi-stage infrastructure upgrades. The system decomposes the upgrade challenge into distinct architectural layers, each responsible for specific aspects of the upgrade lifecycle while maintaining loose coupling through well-defined interfaces.

5.2.1 System Components Overview

The operator architecture comprises five primary component categories that collaborate to execute upgrades safely and reliably.

The **custom resource definition layer** provides the declarative API surface through which users express upgrade intent, defining the LiqoUpgrade resource type with its specification and status structures. This API layer abstracts the complexity of multi-stage upgrade orchestration behind a simple version specification interface, enabling users to declare desired target versions without understanding the intricate sequencing and validation logic required for safe execution.

The **control layer** implements the reconciliation controller that watches LiqoUpgrade resources and orchestrates the upgrade workflow through a state machine. This controller serves as the brain of the system, making decisions about stage progression, validating

prerequisites, detecting failures, and initiating rollback procedures when necessary. The controller operates as a standard Kubernetes controller built using the controller-runtime library, inheriting the reliability and operational characteristics of the Kubernetes control plane itself. By leveraging the controller pattern rather than implementing a standalone service, the operator gains automatic leader election, graceful shutdown handling, and integration with Kubernetes RBAC and service account mechanisms.

The **execution layer** comprises Kubernetes Jobs that perform the actual upgrade operations within each stage. Rather than executing upgrade logic directly within the controller process, the architecture delegates all modification operations to Jobs that run as separate pods with appropriate RBAC permissions. This job-based execution model provides several architectural advantages including isolation of execution context, automatic retry handling through Job backoff policies, and observable execution through standard Kubernetes Job status tracking. Each job executes a bash script that encapsulates the specific upgrade logic for its stage, using kubectl commands and direct API interactions to modify cluster resources.

The **configuration layer** consists of ConfigMap resources that store upgrade metadata including the version compatibility matrix, target component descriptors, pre-upgrade state snapshots, and differential upgrade plans. These ConfigMaps serve as persistent storage for upgrade state that must survive controller restarts and provide a debugging interface where operators can inspect captured state and planned modifications before execution. The use of ConfigMaps rather than custom resource definitions for these artifacts reflects their auxiliary nature as implementation details rather than user-facing API objects.

The **monitoring layer** encompasses the status reporting mechanisms that provide visibility into upgrade progress and outcomes. The LiqoUpgrade resource status subresource captures the current phase, stage number, status messages, and structured conditions that represent specific aspects of upgrade state such as health validation results and CRD schema change warnings. The operator generates Kubernetes events at significant transition points, providing an audit trail of upgrade activities that integrates with standard Kubernetes monitoring and logging infrastructure.

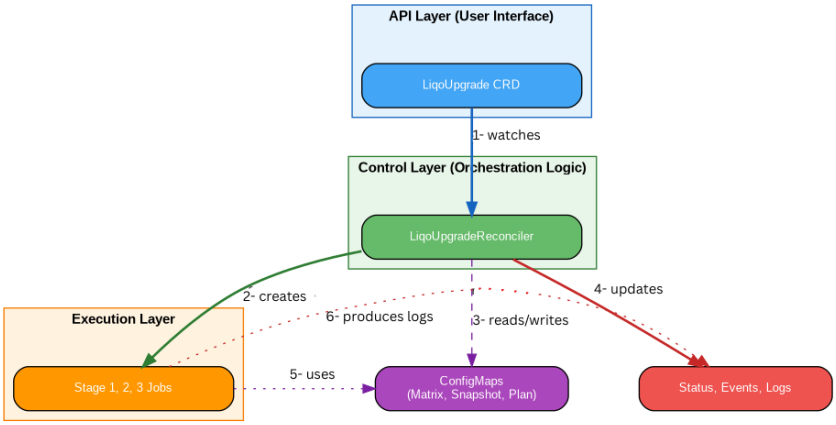

Figure 5.1. Liqo Upgrade Operator system architecture showing the API layer, control layer, execution layer, configuration layer, and monitoring layer.

5.2.2 Operator Interaction Model

The operator's interaction model follows the standard Kubernetes controller pattern where the reconciler continuously observes the declared state in LiqoUpgrade resources and performs actions to converge the actual cluster state toward that declared state. This level-triggered rather than edge-triggered design ensures that transient failures or missed events do not prevent eventual convergence, as the controller repeatedly examines the current state and determines necessary actions regardless of how the system reached its current configuration.

User interaction with the operator occurs exclusively through standard Kubernetes API mechanisms, specifically through creating, reading, updating, and deleting LiqoUpgrade custom resources using kubectl or any Kubernetes API client. This API-driven interaction model eliminates the need for specialized command-line tools or direct operator API endpoints, ensuring that the upgrade operator integrates seamlessly with existing Kubernetes operational workflows and toolchains. Users specify only the target version and optional configuration parameters in the LiqoUpgrade resource specification, delegating all upgrade orchestration decisions to the operator's control logic.

The reconciliation loop operates asynchronously from the user's perspective, with status updates providing feedback about upgrade progress. Users monitor upgrade execution by examining the LiqoUpgrade resource status, reading generated events, or observing Job resources created by the operator. The controller implements exponential backoff for transient errors and permanent failure detection for cases where manual intervention becomes necessary.

The operator interacts with the Kubernetes API server extensively throughout the upgrade workflow, creating and monitoring Jobs, reading and patching Deployments and DaemonSets, inventorying custom resources, and managing ConfigMaps for state

persistence. The operator also interacts indirectly with external systems through Jobs, particularly with the GitHub API to fetch Liqo release artifacts and with Helm to render target component descriptors from chart templates.

5.2.3 Control Flow and State Management

The operator's control flow implements a finite state machine where the LiqoUpgrade resource's phase field determines the current state and reconciliation logic transitions between states based on completion of current activities and outcomes of validation checks. This state machine architecture provides deterministic upgrade progression where each phase represents a well-defined checkpoint in the upgrade workflow with specific entry conditions, execution actions, and exit criteria. The explicit state representation in the resource status enables external observers to understand upgrade progress unambiguously and supports recovery scenarios where controller restarts require determining what actions have completed and what remains pending.

State persistence occurs exclusively through the LiqoUpgrade resource status subresource and associated ConfigMap resources, with no state maintained in the controller process memory beyond caches provided by the controller-runtime framework. This stateless controller design ensures that controller pod restarts, leader election changes, or cluster disruptions do not compromise upgrade progress or create state inconsistencies. The status update strategy implements optimistic concurrency control with automatic retry logic that refetches the latest resource version before attempting updates, preventing conflicts when multiple reconciliation loops or external status updates occur concurrently.

The reconciliation loop executes continuously with configurable requeue intervals that vary based on the current phase. During active upgrade stages where Jobs execute external modifications, the controller requeues relatively frequently to detect Job completion promptly. During terminal phases such as Completed or Failed, the controller ceases active reconciliation, relying on watch events triggered by resource modifications to reinitiate reconciliation if users update specifications or delete the resource.

Control flow branches occur at multiple decision points throughout the reconciliation logic. The pre-upgrade validation phase determines whether to proceed with upgrade execution or reject the request due to incompatibilities or missing prerequisites. Each stage monitors Job status to determine whether to advance to the next stage, initiate rollback due to failure, or continue waiting for Job completion. The rollback phase implements conditional logic that determines the scope of rollback based on which stage failed, applying cumulative rollback that restores all components upgraded prior to the failure point.

The pseudocode below illustrates the high-level structure of the reconciliation loop:

Listing 5.1. Reconciliation loop pseudocode

```
function Reconcile(ctx, req):
  try:

    // Fetch the LiqoUpgrade resource
    resource = fetchResource(req)
    if resource == nil: return success
```

```
7
8     // Handle deletion with finalizer
9     if resource.DeletionTimestamp != nil:
10      if hasFinalizer(resource):
11        cleanupJobs(resource)
12        removeFinalizer(resource)
13      return success
14
15    // Ensure finalizer exists
16    if !hasFinalizer(resource):
17      addFinalizer(resource)
18
19    // State machine: switch on current phase
20    try:
21      switch resource.Status.Phase:
22        case "Pending":
23          transitionTo(resource, "Validating")
24
25        case "Validating":
26          try:
27            handleValidationPhase(resource)
28          catch ValidationError as e:
29            log.Error("Validation failed", e)
30            transitionTo(resource, "Failed")
31            updateStatusWithError(resource, e)
32            return requeue(noRequeue)
33
34        case "CRDs":
35          try:
36            handleCRDStage(resource)
37          catch JobFailure as e:
38            log.Error("CRD upgrade failed", e)
39            if resource.Spec.AutoRollback:
40              transitionTo(resource, "RollingBack")
41            else:
42              transitionTo(resource, "Failed")
43            updateStatusWithError(resource, e)
44            return requeue(exponentialBackoff)
45
46        case "ControllerManager":
47          try:
48            handleControlPlaneStage(resource)
49          catch JobFailure as e:
50            log.Error("Control plane upgrade failed", e)
51            if resource.Spec.AutoRollback:
52              transitionTo(resource, "RollingBack")
53            else:
54              transitionTo(resource, "Failed")
55            updateStatusWithError(resource, e)
56            return requeue(exponentialBackoff)
```

```
57
58          case "NetworkFabric":
59            try:
60              handleNetworkFabricStage(resource)
61            catch JobFailure as e:
62              log.Error("Network fabric upgrade failed", e)
63              if resource.Spec.AutoRollback:
64                transitionTo(resource, "RollingBack")
65              else:
66                transitionTo(resource, "Failed")
67              updateStatusWithError(resource, e)
68              return requeue(exponentialBackoff)
69
70          case "Verifying":
71            try:
72              handleVerificationPhase(resource)
73            catch HealthCheckTimeout as e:
74              log.Error("Post-upgrade verification failed", e)
75              if resource.Spec.AutoRollback:
76                transitionTo(resource, "RollingBack")
77              else:
78                transitionTo(resource, "Failed")
79              return requeue(exponentialBackoff)
80
81          case "RollingBack":
82            try:
83              handleRollbackPhase(resource)
84            catch RollbackFailure as e:
85              log.Error("Rollback failed - manual intervention
    required", e)
86              transitionTo(resource, "Failed")
87              updateStatusWithError(resource, e)
88              return requeue(noRequeue)
89
90          case "Completed", "Failed":
91            // Terminal states - no action
92            return success
93
94        // Update status with new phase
95        updateStatus(resource)
96
97        // Return requeue interval based on phase
98        return requeue(getRequeueInterval(resource.Status.Phase))
99
100     catch StatusUpdateConflict as e:
101
102       // Resource was modified by another reconciler
103       log.Info("Status update conflict, retrying", e)
104       return requeue(immediateRequeue)
105
```

```
106     catch error as e:
107
108       // Unexpected error in state machine
109       log.Error("Unexpected error in reconciliation", e)
110       updateStatusWithError(resource, e)
111       return requeue(exponentialBackoff)
112
113   catch FetchError as e:
114
115     // Failed to fetch resource from API server
116     log.Error("Failed to fetch LiqoUpgrade resource", e)
117     return requeue(exponentialBackoff)
118
119   catch error as e:
120
121     // Catastrophic error - likely programming bug
122     log.Error("Catastrophic reconciliation failure", e)
123     return requeue(exponentialBackoff)
```

The error handling strategy implements multiple layers of exception management. **Phase-specific errors** (`ValidationError`, `JobFailure`, `HealthCheckTimeout`) trigger conditional rollback based on the AutoRollback specification field. When enabled, the reconciler transitions to RollingBack phase to restore pre-upgrade state. When disabled, operators can investigate failures manually before deciding recovery strategy.

Transient errors (`StatusUpdateConflict`) indicate concurrent modifications and trigger immediate requeue, allowing the reconciler to refetch current state and retry the operation.

Catastrophic errors represent unexpected conditions likely indicating programming defects rather than operational failures. These trigger exponential backoff requeue, preventing tight reconciliation loops while allowing eventual recovery if conditions improve.

The nested try-catch structure ensures that failures in individual stages don't compromise controller availability. Job execution errors remain isolated to their phase handlers, while the outer catch blocks protect against API server communication failures and resource fetch errors that could affect any reconciliation cycle.

The state management strategy extends beyond the upgrade lifecycle to encompass cleanup and finalization. The operator attaches a finalizer to LiqoUpgrade resources upon creation, enabling the controller to perform cleanup operations before resource deletion completes. This finalization logic deletes all Job resources created during the upgrade workflow, preventing accumulation of completed Jobs across multiple upgrade attempts.

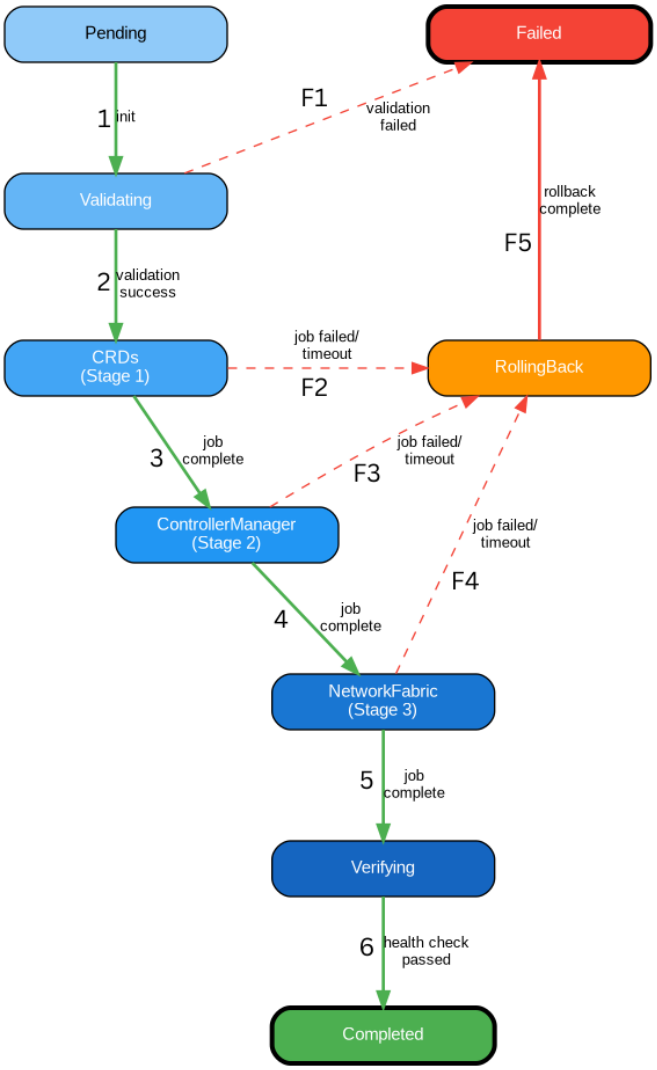

Figure 5.2. Upgrade workflow state machine showing transitions between Pending, Validating, CRDs, ControllerManager, NetworkFabric, Verifying, RollingBack, Completed, and Failed states.

5.3 Staged Upgrade Workflow

The operator decomposes the complex upgrade operation into three sequential stages that address distinct layers of the Liqo architecture while respecting the dependency relationships between these layers. This staged approach reflects a fundamental architectural principle where schema evolution must precede component upgrades that depend on new

schemas, and control plane stability must be established before disrupting network data plane components. The three-stage decomposition also provides natural checkpoints for health validation and creates opportunities for early failure detection before modifications propagate to more critical system layers.

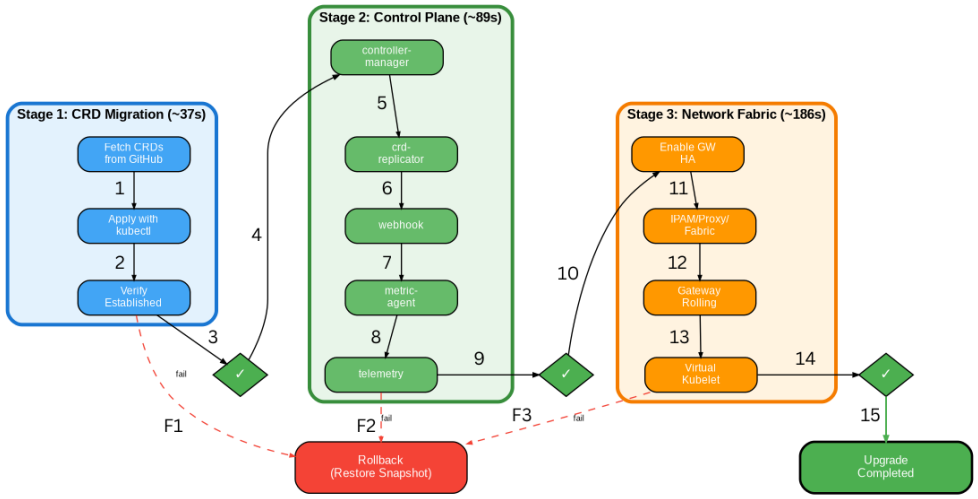

Figure 5.3. Three-stage upgrade workflow: Stage 1 CRD Migration (∼37s), Stage 2 Control Plane (∼89s), Stage 3 Network Fabric (∼186s), with rollback path shown.

5.3.1 Stage 1: CRD Migration

The first stage focuses exclusively on upgrading Custom Resource Definitions that define the API schemas for Liqo's custom resources. This stage executes before any component upgrades to ensure that updated components encounter the API schemas they expect rather than older schema versions that may lack required fields or validation rules. The CRD upgrade stage represents the foundation upon which subsequent stages build, as incompatible CRD versions would cause reconciliation failures in Liqo's controllers even if the controller code itself upgraded successfully.

The stage implementation creates a Kubernetes Job that fetches CRD manifests directly from the Liqo GitHub repository for the target version, applies them using kubectl with server-side apply and force-conflicts flags to handle field ownership transitions, and waits for all CRDs to reach the established condition. The GitHub integration approach avoids embedding CRD manifests within the operator container image, enabling the operator to upgrade Liqo to any version without requiring operator image updates. This design choice supports the operator's goal of functioning as a general-purpose Liqo upgrade tool rather than being tightly coupled to specific Liqo versions.

The stage incorporates comprehensive error handling for network failures, GitHub API rate limiting, and CRD application failures. The job script validates that all CRD files

fetched successfully and that kubectl apply operations completed without errors before declaring stage success. Upon stage completion, the operator performs CRD change analysis by comparing the snapshot of CRDs captured during pre-upgrade validation against the current CRD state, identifying schema version additions, storage version changes, and removed versions. This analysis generates warnings for potentially incompatible changes such as storage version migrations that might affect existing custom resource instances, though the operator does not block upgrades based on these warnings given the complexity of fully automated compatibility assessment.

The CRD migration stage must execute first due to Liqo's architectural dependency structure. This ordering cannot be dynamically discovered through introspection or runtime analysis—it derives from understanding how Liqo's components interact at the architectural level. Controllers read and write custom resources conforming to CRD-defined schemas. If controllers upgrade to versions expecting new schema fields before CRDs update to include those fields, controllers will fail to create or update resources. Conversely, if CRDs update to versions removing fields that old controller versions still attempt to access, reconciliation loops fail with schema validation errors. The hardcoded CRD-before-controllers ordering reflects this architectural reality rather than arbitrary implementation choice.

5.3.2 Stage 2: Control Plane Upgrade

The second stage upgrades Liqo's control plane components that implement reconciliation logic, webhook validation, telemetry collection, and cross-cluster resource replication. This stage modifies Deployments and CronJobs for components including liqo-controller-manager, liqo-crd-replicator, liqo-webhook, liqo-metric-agent, and liqo-telemetry. The stage executes after CRD migration ensures that the API schemas these components rely upon have updated to versions compatible with the new component code.

The stage implementation creates a Job that loads the upgrade plan generated during the validation phase, validates that all ConfigMaps and Secrets referenced in component environment variables exist to prevent runtime configuration errors, and sequentially upgrades each control plane component with health checks between components. The upgrade process updates container images to the target version while also applying configuration changes captured in the upgrade plan, including environment variable additions or modifications, command-line argument changes, and command array updates.

This configuration update capability addresses a critical limitation in automated infrastructure upgrades: detecting what configuration changes occurred between versions requires understanding the semantic meaning of those changes, not just structural differences. When a new Liqo version adds an environment variable controlling a feature flag, removes a deprecated command-line argument, or changes the default value of a configuration parameter, the operator must apply those changes to ensure components function correctly with the new code. However, automatically inferring the necessity and compatibility of such changes proves infeasible—this knowledge is embedded in the Liqo source code, and determining which environment variable additions are optional versus required, which argument removals need compensating configuration elsewhere, or which value changes are safe versus breaking requires analysis of the implementation itself.

The current implementation requires manual specification of these configuration changes in the operator's configuration files, encoded as part of the upgrade plan generation logic. This manual specification addresses the configuration change semantic gap—only maintainers possess the domain knowledge to determine which changes are required, optional, or conditional. Future work will address this limitation by transitioning to a maintainer-provided descriptor file approach, where Liqo releases include structured metadata describing required configuration changes alongside code updates. This would shift the burden of semantic change specification from operator implementers to the Liqo community, who would expose this knowledge through machine-readable documents included with each release.

The stage employs Kubernetes rollout mechanisms with health checks to ensure each component reaches a healthy state before upgrading subsequent components. After updating a Deployment, the job executes kubectl rollout status to wait for the rollout to complete and kubectl wait to verify that the Deployment reaches the Available condition. This sequential upgrade approach with health gates prevents cascading failures where an unhealthy component triggers failures in dependent components during their upgrade.

The hardcoded component upgrade order within this stage—liqo-controller-manager first, followed by other components—reflects architectural dependencies. The controller manager implements core reconciliation logic that other components depend on indirectly. Upgrading it first ensures the fundamental control loops stabilize before peripheral components undergo transitions. This ordering cannot be discovered dynamically because Liqo does not expose component dependency graphs through APIs or metadata. The sequencing derives from understanding which components create or modify which resource types, which depend on specific CRD schemas being available, and which must be operational for others to function correctly.

5.3.3 Stage 3: Network and Data Plane Upgrade

The third stage represents the most complex phase of the upgrade workflow, upgrading network fabric components that implement cross-cluster connectivity and routing. This stage modifies the liqo-ipam Deployment that manages IP address allocation, the liqo-proxy Deployment that handles service reflection, the liqo-fabric DaemonSet that implements tunnel interfaces on each node, virtual kubelet components that project remote cluster capacity, and gateway components that establish VPN tunnels between clusters. The network fabric upgrade must execute carefully to minimize disruption to active cross-cluster workloads that depend on these components for connectivity.

The stage executes a sophisticated multi-step workflow that temporarily enables gateway high availability, upgrades network components sequentially with health validation, implements gateway rolling updates with leader election management, and verifies connectivity before declaring success. The workflow begins by configuring gateway template CRDs to specify two replicas and rolling update strategies, triggering Liqo's controllers to propagate these settings to gateway Deployments across all tenant namespaces. This temporary high availability configuration enables Kubernetes to maintain one healthy gateway pod while upgrading the second, reducing but not eliminating connectivity disruption during gateway image transitions.

Core network component upgrades follow a dependency-aware sequence, upgrading liqo-ipam first as it provides IP allocation services to other components, followed by liqo-proxy, and finally liqo-fabric. The workflow implements data plane reset procedures including cleaning stale network interfaces on nodes, annotating InternalNode resources to trigger reconciliation without deletion, and flushing connection tracking tables to eliminate stale entries that could prevent API server connectivity after interface modifications. These reset procedures address practical challenges discovered during testing where stale kernel state caused intermittent connectivity failures despite successful component upgrades.

Gateway upgrades occur simultaneously across all peerings rather than implementing canary rollouts, reflecting the current implementation's prioritization of upgrade speed over gradual rollout complexity. The workflow monitors gateway pod versions, forces leader election transitions when the active gateway pod runs the old version, and implements IP synchronization logic that ensures gateway status fields reflect the current active pod's IP address. This IP synchronization addresses a critical requirement where Liqo's network controllers require accurate gateway endpoint information to program routing tables correctly. The workflow also applies network tuning configurations including reverse path filtering adjustments and TCP MSS clamping rules to gateway pods, ensuring proper packet forwarding behavior across VPN tunnels.

Virtual kubelet upgrades execute after network fabric stabilization, modifying VkOptionsTemplate CRDs and triggering deletion and recreation of virtual kubelet Deployments. The workflow waits for virtual kubelets to recreate virtual nodes and for those nodes to reach Ready status before proceeding to final verification steps.

This stage's complexity illustrates why infrastructure upgrades differ fundamentally from application deployments. Applications can leverage blue-green or canary strategies because they maintain stateless behavior or externalize state to databases. Network fabric components maintain kernel-level state including routing tables, iptables rules, and tunnel interfaces that cannot cleanly migrate between old and new component versions without disruption. When a gateway pod restarts, the WireGuard tunnel terminates, requiring clients and servers to renegotiate the tunnel with cryptographic handshakes and route establishment—a process requiring several seconds. This characteristic contributes to the minimal downtime experienced during Stage 3, representing not an implementation deficiency but an inherent architectural constraint of how Kubernetes pod lifecycle management and Liqo's tunnel architecture interact.

5.3.4 Stage Sequencing and Dependencies

The three-stage sequence—CRDs, control plane, network fabric—reflects a strict dependency hierarchy that cannot be reordered without risking upgrade failures. CRDs must upgrade first because all Liqo components depend on custom resource schemas matching their code expectations. Control plane components must upgrade before network fabric because the control plane includes controllers that manage network fabric resources like gateway configurations and IP allocation policies. Network fabric components upgrade last because they require stable control plane operation to reconcile their configurations correctly during image transitions.

This dependency-driven sequencing represents a fundamental constraint rather than an arbitrary implementation choice. Liqo's architecture provides no discovery APIs that would enable dynamic dependency inference—the component ordering must be explicitly encoded based on architectural understanding of how components interact. The operator cannot dynamically discover the correct ordering by introspecting cluster state or querying Liqo components—the dependencies exist at the architectural level encoding knowledge about which components create versus consume which resource types, which components implement controllers versus workload execution, and which layer transitions can proceed independently versus requiring prior layer stability.

5.4 Configuration Management

The operator's configuration management approach balances flexibility with operational safety, employing a hybrid strategy that combines embedded defaults with externalized overrides for environment-specific customizations.

The compatibility matrix embedded in the operator's ConfigMap defines supported upgrade paths between Liqo versions. This matrix encodes knowledge about breaking changes, API compatibility constraints, and version combinations that have undergone validation. The matrix enables the operator to reject upgrade attempts that would transition between incompatible versions, preventing users from attempting unsupported upgrade paths that would likely fail midway through execution.

The upgrade plan generation mechanism creates detailed descriptions of required modifications for each component, capturing not only image version changes but also environment variable additions, command-line argument modifications, and configuration value updates. This comprehensive change specification enables the upgrade Jobs to apply all necessary modifications atomically rather than requiring multiple sequential update operations. However, generating accurate upgrade plans requires understanding the semantic meaning of configuration changes—which environment variables are required versus optional, which argument changes necessitate coordinated updates elsewhere, or which value modifications are backward compatible. The current implementation encodes this knowledge manually in the operator's configuration logic, requiring updates to the operator itself when new Liqo versions introduce configuration schema changes.

This manual specification requirement represents a known limitation. This manual specification requirement represents a known limitation. This expertise is embedded in the Liqo source code, and the Liqo community is best positioned to expose through machine-readable documents which configuration changes between versions are required, optional, or conditionally necessary based on feature enablement. The operator cannot automatically infer these semantics by analyzing manifests or comparing YAML structures. Future work will address this through maintainer-provided descriptor files that accompany Liqo releases, externalizing the configuration change specification burden to those with the requisite domain knowledge while enabling the operator to remain version-agnostic.

5.5 Health Validation Framework

The operator implements a comprehensive health validation framework that gates upgrade progression at multiple points, ensuring each stage completes successfully before modifications propagate to subsequent stages.

Pre-upgrade validation executes before any modifications occur, examining cluster state to determine whether the upgrade can proceed safely. This validation phase checks that all required Liqo components exist in expected namespaces, verifies that the target version differs from the current version to prevent no-op upgrades, queries ForeignCluster resources to determine peer cluster versions, and consults the compatibility matrix to ensure the proposed upgrade creates compatible version combinations across peered clusters.

The validation phase also captures a snapshot of the current cluster state, inventorying all Liqo Deployments, DaemonSets, and CRDs with their current configurations including image versions, environment variables, command-line arguments, and replica counts. This snapshot serves dual purposes: it provides the baseline against which the upgrade plan calculates required modifications, and it supplies the restoration target for rollback operations if the upgrade fails.

Each stage transition incorporates health gates that verify successful completion of the preceding stage before proceeding. After Stage 1 completes, the operator waits for all CRDs to reach the Established condition, indicating the API server has accepted and activated the new schemas. After Stage 2, the operator checks that all control plane Deployments report Available status with the expected number of ready replicas. After Stage 3, the operator verifies that gateway Deployments across all tenant namespaces report ready status and that ForeignCluster resources maintain Healthy peering status.

These health gates implement timeout-based failure detection where prolonged unhealthy status triggers rollback rather than indefinite waiting. The timeout durations balance giving components sufficient time to stabilize against avoiding excessive delays when genuine failures occur.

After all three stages complete successfully, a final verification phase performs end-to-end health checks to ensure the upgraded Liqo installation functions correctly. This verification queries all Liqo components to confirm they report healthy status, checks that ForeignCluster resources maintain established peerings with no connectivity degradation warnings, and validates that virtual nodes representing remote cluster capacity remain in Ready state.

The post-upgrade validation provides confidence that the upgrade succeeded not just technically but functionally—that Liqo's multi-cluster capabilities remain operational after the version transition. This validation phase also generates a post-upgrade snapshot capturing the new component versions and configurations, providing documentation of what the upgrade achieved and enabling future differential analysis.

5.6 Rollback Mechanism

The operator implements automatic rollback capabilities to recover from upgrade failures, preventing clusters from remaining in partially upgraded states that compromise functionality.

Failure detection operates through Job status monitoring, where failed or exceeded-deadline Jobs trigger rollback initiation. The operator watches for Job conditions indicating terminal failure states including BackoffLimitExceeded when retry attempts exhaust, DeadlineExceeded when execution time surpasses configured limits, or Failed status reported by the Job controller. Upon detecting these failure conditions, the operator transitions the LiqoUpgrade resource to the RollingBack phase and begins restoring components to their pre-upgrade configurations.

The failure detection strategy also incorporates health gate timeouts, where components failing to reach healthy status within configured intervals trigger rollback even if the upgrade Job reports success. This addresses scenarios where upgrades apply successfully from an API perspective but the upgraded components fail to achieve operational health due to configuration incompatibilities or runtime errors.

Rollback implements a cumulative strategy where all stages completed prior to the failure point roll back to their original configurations, not just the failed stage itself. This approach ensures consistent version alignment across all Liqo components rather than creating partially upgraded states where some components run new versions while others revert to old versions.

The cumulative rollback creates a new Job that executes restoration operations in reverse stage order—network fabric components revert first, followed by control plane components, and finally CRDs. This reversal mirrors the forward upgrade sequence, respecting the same dependency constraints during restoration as during progression. The rollback Job applies the pre-upgrade snapshot configurations captured during validation, restoring component images, environment variables, command arguments, and replica counts to their original values.

State restoration leverages the pre-upgrade snapshot ConfigMap as the authoritative source of original configurations. The rollback Job reads this snapshot, extracts component specifications, and applies them using kubectl patch operations with strategic merge patch semantics. This patching approach enables selective restoration of modified fields without affecting unrelated configuration that may have changed independently during the upgrade attempt.

The restoration process includes verification steps that wait for rolled-back components to achieve healthy status before considering rollback complete. This verification ensures that clusters return to functional states rather than merely reverting to old configurations that may not stabilize. Upon successful rollback verification, the operator transitions the LiqoUpgrade resource to Failed phase with status messages documenting which stage failed and that rollback completed successfully.

5.7 Observability and Status Reporting

The operator provides comprehensive observability into upgrade progress and outcomes through multiple mechanisms that integrate with standard Kubernetes operational tooling.

The LiqoUpgrade resource status subresource maintains structured information about upgrade state including the current phase field indicating progression through the state machine, stage number tracking which of the three stages is executing, and a conditions array containing typed condition objects representing specific aspects of upgrade health. These conditions include HealthCheckPassed indicating component health validation results, CompatibilityVerified reflecting multi-cluster version compatibility assessment, and CRDSchemaChanged warning about potential breaking schema migrations.

The status structure provides sufficient information for both human operators and automated monitoring systems to understand upgrade state unambiguously. Users can determine at a glance whether an upgrade is progressing normally, which stage is executing, and whether any warnings require attention.

The operator generates Kubernetes events at significant transition points throughout the upgrade workflow. Events document phase transitions, stage initiations, health validation outcomes, and failure detections, creating an audit trail of upgrade activities. These events integrate with standard Kubernetes event viewing mechanisms including kubectl get events and cluster logging aggregators, enabling operators to reconstruct upgrade timelines and diagnose issues retroactively.

Event generation follows Kubernetes conventions for event severity, using Normal events for expected progressions and Warning events for anomalous conditions that don't constitute failures. This severity classification enables filtering and prioritization in monitoring dashboards.

The operator controller emits structured logs using standard logging libraries that integrate with Kubernetes logging infrastructure. Logs capture reconciliation decisions, API interactions, and error conditions with sufficient context to support debugging. The logging strategy balances verbosity with operational utility, emitting information-level logs for normal progression and error-level logs for unexpected conditions.

Job execution logs supplement controller logs by documenting the actual upgrade operations performed within each stage. These Job logs contain command outputs, error messages from kubectl operations, and validation results, providing detailed traces of what modifications occurred and why operations succeeded or failed.

5.8 Multi-Cluster Coordination

The operator's multi-cluster coordination strategy implements implicit coordination through compatibility validation rather than explicit inter-operator communication. Each cluster runs its own operator instance that makes autonomous upgrade decisions based on local state and compatibility constraints derived from ForeignCluster resources representing peer clusters.

Before initiating upgrades, operators query ForeignCluster resources to determine

peer cluster versions, consult the embedded compatibility matrix to verify the proposed upgrade creates compatible version combinations, and reject upgrades that would violate compatibility constraints. This validation approach enables distributed coordination where clusters can upgrade independently during their respective maintenance windows while preventing version skew that would break peering relationships.

The distributed coordination model provides operational benefits including deployment simplicity without requiring cross-cluster authentication, autonomous upgrade timing decisions per cluster, and elimination of single-point-of-failure centralized coordinators. However, it limits coordination sophistication—the current approach cannot implement cluster-wide canary strategies or synchronized upgrade timing across peered clusters. These limitations represent acceptable trade-offs for the target deployment scenarios where brief network disruptions during independent cluster maintenance windows prove acceptable.

5.8.1 Gateway Upgrade Strategy: Parallel vs. Sequential

The operator upgrades gateway deployments simultaneously across all peerings rather than implementing sequential canary rollouts. This design choice reflects a deliberate trade-off between upgrade speed and gradual rollout complexity, prioritizing the minimal-downtime objective.

Sequential canary gateway upgrades would update each gateway individually while others remain at the old version, providing incremental validation before full rollout. However, due to platform-inherent downtime causes—Linux conntrack table flush, Kubernetes InternalNode reconciliation delays, and Liqo gateway IP synchronization—each gateway upgrade incurs 14–18 seconds of connectivity loss regardless of orchestration strategy. These root causes are analyzed in detail in Section 7.4.2.

In multi-peered scenarios, sequential canary would result in cumulative downtime as each gateway upgrade adds its own disruption window, as shown in Table 5.1.

Table 5.1. Downtime comparison: parallel upgrade vs. sequential canary by peering count.

Cluster Peerings	Parallel Upgrade	Sequential Canary	Additional Cost
1	14–18s	14–18s	None
2	14–18s	28–36s	+14–18s
5	14–18s	70–90s	+56–72s
10	14–18s	140–180s	+126–162s

The parallel approach maintains a single disruption window regardless of peering count by coordinating gateway restarts to occur simultaneously. The downtime contributors (conntrack flush, reconciliation delays) affect all peerings during the same time window rather than sequentially. While slight temporal variations exist due to Kubernetes pod scheduling differences and per-gateway leader election timing, the total downtime remains bounded within 14–28 seconds rather than multiplying linearly with peering count.

Canary strategies provide value in application deployments where gradual traffic shifting enables validation before full rollout—observing metrics, error rates, and behavior under partial load. However, for infrastructure components like gateways where connectivity is binary (working or not working) and downtime is unavoidable during pod lifecycle transitions, canary offers no validation advantage. Each gateway either successfully re-establishes WireGuard tunnels and restores routing or fails completely, with no intermediate states to observe or gradual behavior to measure.

The parallel strategy trades away incremental risk exposure in favor of minimizing absolute downtime duration. Organizations with different priorities—valuing gradual validation over speed—could implement sequential canary by accepting the cumulative downtime trade-off. However, such an approach represents a different optimization objective than the minimal-downtime goal stated in this thesis. The current implementation choice reflects that for multi-cluster networking infrastructure, total disruption time takes precedence over staged rollout validation.

Chapter 6

Implementation Details

6.1 Development Environment and Project Structure

The operator implementation leverages the Kubebuilder framework to generate the foundational project structure and provide opinionated patterns for Kubernetes operator development [23]. This framework choice accelerates development by providing scaffolding for common operator components including custom resource definition generation, controller boilerplate, webhook templates, and configuration manifests. The development environment employs standard Kubernetes ecosystem tooling including the Go programming language for controller implementation, kubectl for cluster interaction during testing, and Kind for creating local test clusters.

6.1.1 Kubebuilder Project Setup

The project initialization employs Kubebuilder version 3.12 which generates a Go module-based project structure conforming to current Kubernetes operator development best practices. The initialization command establishes the project domain as `liqo.io` and creates the initial directory hierarchy including api directories for custom resource type definitions, internal/controller directories for reconciliation logic, and config directories for Kubernetes manifests. The generated project includes Makefile targets that automate common development tasks such as generating custom resource definition YAML from Go type annotations, building container images, and deploying the operator to test clusters.

The Kubebuilder scaffold includes controller-runtime as the foundational library providing the Manager abstraction that coordinates controller execution, handles leader election, manages webhook servers, and provides cached Kubernetes API clients. This manager-based architecture ensures that the operator integrates correctly with Kubernetes control plane patterns including graceful shutdown handling, health check endpoints, and metrics exposition.

6.1.2 Dependencies and Libraries

The implementation relies on several core dependencies managed through Go modules that provide essential functionality for Kubernetes operator development. The controller-runtime library at version 0.16 provides the reconciliation framework, client abstractions, and manager infrastructure that form the foundation of the operator's control logic. The apimachinery library supplies the meta types including `metav1.TypeMeta` and `metav1.ObjectMeta` that all Kubernetes resources embed, along with utilities for label selectors, field selectors, and resource version comparison.

The client-go library provides the typed and dynamic clients for interacting with standard Kubernetes resources including Deployments, DaemonSets, Jobs, and ConfigMaps that the operator reads and modifies during upgrade execution. The implementation uses versioned clients from client-go's kubernetes/typed packages rather than the dynamic client interface, trading some flexibility for type safety and IDE support during development. The apiextensions-apiserver library provides types for working with CustomResourceDefinition resources directly, enabling the CRD change analysis logic that compares pre-upgrade and post-upgrade CRD schemas.

Beyond the core Kubernetes libraries, the implementation includes the Liqo API types as a dependency, importing the `core.liqo.io/v1beta1` package to access ForeignCluster resource definitions. This dependency enables the operator to query ForeignCluster resources during version compatibility validation without maintaining duplicate type definitions. The project also includes standard Go libraries for JSON serialization using encoding/json, YAML processing using sigs.k8s.io/yaml, and semantic version comparison using a lightweight semver parsing implementation adapted from Kubernetes version comparison utilities.

6.1.3 Code Organization

The codebase organizes implementation files into logical groupings that separate API definitions, controller logic, and configuration generation. The `api/v1alpha1` directory contains the `liqoupgrade_types.go` file defining the LiqoUpgrade custom resource specification and status structures along with the `groupversion_info.go` file declaring the API group and version constants. The `internal/controller` directory houses the core implementation files including `liqoupgrade_controller.go` for the main reconciliation loop, `validation.go` for pre-upgrade checks, `snapshot.go` and `planning.go` for state capture and differential planning, and separate files for each upgrade stage and rollback logic.

This file-per-concern organization maintains readability as the codebase grew to exceed 11,000 lines of implementation code, preventing individual files from becoming unwieldy while keeping related functionality grouped together. The `cmd/main.go` entry point remains minimal, focusing exclusively on manager initialization and controller registration while delegating all business logic to the `internal/controller` package. The config directory maintains a parallel organization where subdirectories contain Kubernetes manifests for RBAC rules, custom resource definitions, default ConfigMaps including the compatibility matrix and sample target descriptors, and kustomization overlays

for different deployment scenarios.

6.2 Custom Resource Definition Implementation

The LiqoUpgrade custom resource definition implements the declarative API through which users specify upgrade intentions and observe upgrade progress. The implementation employs Kubebuilder marker annotations that generate both the CustomResourceDefinition YAML manifest and the OpenAPI validation schema embedded within it. These markers provide compile-time documentation of API semantics while enabling automated validation of user-provided specifications before they reach the controller's reconciliation logic. The API design balances expressiveness with simplicity, exposing configuration options that affect upgrade behavior while maintaining sensible defaults that enable basic upgrades through minimal specifications containing only the target version.

The `LiqoUpgradeSpec` structure defines the desired state that users declare when creating upgrade resources, capturing the target version and optional behavioral parameters. The `TargetVersion` field represents the only required specification element, accepting string values that identify Liqo releases using semantic version notation with or without the leading "v" prefix. Kubebuilder validation markers enforce that this field remains non-empty and matches expected version string patterns, preventing syntactically invalid version specifications from entering the system.

The `Namespace` field enables scoped upgrades that affect only Liqo components within a specific namespace, though most deployments use cluster-scoped installations where this field remains empty. The `AutoRollback` boolean field controls whether the operator automatically initiates rollback upon detecting stage failures, defaulting to true for safety but configurable to false when operators prefer manual failure investigation. The `DryRun` boolean enables plan generation and validation without executing actual modifications, supporting review workflows where administrators examine planned changes before approving execution.

Additional optional fields include `UpgradeTimeout` controlling the maximum duration for the entire upgrade operation, `StageTimeout` specifying per-stage maximum durations, and `SkipPreflightChecks` disabling pre-upgrade validation for exceptional circumstances where administrators knowingly proceed despite validation warnings. These configuration options provide flexibility for diverse operational scenarios while maintaining reasonable defaults that work for typical deployments.

The `LiqoUpgradeStatus` structure captures the observed state of upgrade execution, providing comprehensive visibility into progress, outcomes, and validation results. The `Phase` field represents the primary state indicator, containing one of the enumerated phase values that encode which stage of the upgrade workflow is currently executing or whether the upgrade reached a terminal state. The `Message` field provides human-readable explanations of the current phase, including detailed error messages when failures occur and progress descriptions during normal execution.

The `ObservedGeneration` field implements the standard Kubernetes pattern for tracking specification changes, enabling clients to determine whether status reflects the current

spec or lags behind recent modifications. The `StartTime` and `CompletionTime` timestamps bracket the upgrade duration, supporting performance analysis and audit trail requirements. The `LastTransitionTime` captures when the current phase began, enabling timeout detection and alerting on upgrades that remain in specific phases longer than expected.

The `FailureReason` field provides structured failure classification when upgrades fail, distinguishing between validation failures, job execution failures, health check timeouts, and rollback failures. The `LastSuccessfulPhase` field tracks the furthest point reached before any failure, enabling the rollback scope determination logic to understand which stages completed successfully and require restoration. The `RolledBack` boolean indicates whether automatic rollback executed, distinguishing failed upgrades that returned to stable states from those requiring manual intervention.

The `CurrentStage` integer provides fine-grained progress tracking within multi-step stages such as the network fabric upgrade, incrementing as the stage progresses through its internal workflow steps. The `SnapshotConfigMap` and `PlanConfigMap` fields reference the ConfigMap names containing the captured state snapshot and generated upgrade plan, enabling operators to inspect these artifacts for debugging or audit purposes. The `PlanReady` boolean signals when plan generation completes, supporting dry-run workflows where users wait for this flag before retrieving and reviewing the plan.

The `Conditions` array implements the Kubernetes-standard status conditions pattern where each condition represents a specific aspect of system state with independent status, reason, and message fields. The implementation defines condition types for `ComponentHealthy` to report overall component health validation results, `CRDWarnings` to communicate potentially incompatible schema changes detected during CRD upgrades, and `UpgradeSucceeded` as the terminal condition indicating successful completion. These structured conditions enable more sophisticated monitoring queries than phase-only checks allow, supporting alert rules that trigger specifically on certain condition types.

The `UpgradePhase` type defines a string-based enumeration representing the possible states in the upgrade state machine. The enumeration includes Pending for newly created resources awaiting validation, Validating during pre-upgrade checks, CRDs while the CRD upgrade job executes, ControllerManager during control plane upgrades, NetworkFabric during network component upgrades, Verifying during post-upgrade validation, RollingBack when automatic recovery executes, Completed for successful upgrades, and Failed for terminal failures. This enumeration provides a finite set of well-defined states that the controller transitions between deterministically based on validation outcomes and job completion status.

The condition type constants define standardized strings for the Type field in status conditions, following Kubernetes API conventions that use PascalCase for condition types. The implementation defines `ConditionHealthy`, `ConditionCRDWarnings`, and `ConditionUpgradeSucceeded` as the primary condition types, with each condition carrying Status as either True, False, or Unknown, a Reason code explaining why the condition has its current status, and a Message providing human-readable details. These conditions accumulate in the status array over the upgrade lifecycle rather than replacing previous conditions, creating a historical record of all state transitions and validation outcomes

that occurred during the upgrade attempt.

6.3 Controller Implementation

The controller implementation follows the standard Kubernetes operator pattern where a reconciliation loop observes LiqoUpgrade custom resources and executes actions to converge actual cluster state toward the declared desired state. The implementation leverages the controller-runtime framework's Manager and Controller abstractions that provide watch mechanisms, work queue management, and leader election coordination. The `LiqoUpgradeReconciler` structure embeds a `client.Client` for Kubernetes API interactions and a `runtime.Scheme` for type registration, maintaining minimal internal state to ensure that controller restarts do not compromise upgrade progress. This stateless design philosophy ensures that all persistent state resides in Kubernetes resources rather than controller process memory, enabling seamless recovery from controller failures or leader election changes.

6.3.1 Reconciliation Loop Architecture

The `Reconcile` method implements the core control logic that executes whenever the controller observes changes to LiqoUpgrade resources or when periodic requeue timers expire. The method signature accepts a context and a `reconcile.Request` containing the namespaced name of the resource that triggered reconciliation, returning a `reconcile.Result` that specifies requeue behavior and an error value that signals whether reconciliation should retry. The controller-runtime framework automatically retries reconciliation with exponential backoff when the method returns errors, while Result values with RequeueAfter durations schedule future reconciliation at specific intervals.

The reconciliation logic begins by fetching the current state of the LiqoUpgrade resource from the Kubernetes API server using the client's Get method. If the resource no longer exists, indicated by an IsNotFound error, the reconciliation returns successfully without further action since the resource was deleted and requires no processing. This deletion handling works in conjunction with finalizers to ensure proper cleanup. When the fetch succeeds, the reconciliation examines the resource's DeletionTimestamp to determine whether cleanup procedures should execute instead of normal upgrade logic.

The primary reconciliation flow implements a switch statement that examines the resource's status phase field and delegates to phase-specific handler methods. This switch-based dispatch provides clear separation between the concerns of different upgrade phases while maintaining centralized control flow through the main reconciliation method. Each phase handler method encapsulates the validation, job creation, monitoring, or verification logic appropriate for its phase, returning `reconcile.Result` values that indicate whether the controller should requeue immediately, wait for a specific duration, or stop reconciling until the next watch event arrives.

The reconciliation loop operates on a level-triggered basis where each invocation examines the current state and determines appropriate actions regardless of what events triggered the reconciliation. This approach ensures that missed events, controller restarts, or external modifications do not prevent the system from eventually converging to the

desired state, as subsequent reconciliation cycles will detect the current state and take appropriate corrective actions.

The `SetupWithManager` method registers the controller with the manager and configures watch relationships that determine which resource changes trigger reconciliation. The implementation watches LiqoUpgrade resources as the primary type and owns Job resources through the Owns relationship, causing the controller to reconcile the owning LiqoUpgrade whenever Jobs created by the operator change state. This ownership-based watching enables the controller to detect job completion events promptly without implementing separate polling mechanisms, reducing reconciliation latency during stage transitions.

6.3.2 State Machine Implementation

The state machine implementation uses the Phase field in the status subresource as the authoritative state indicator, with phase transition logic distributed across handler methods that each determine whether conditions exist to advance to the next phase. The validation handler examines cluster health and version compatibility, transitioning to the CRDs phase when validation succeeds or to Failed when validation detects incompatibilities. The CRD handler creates the CRD upgrade job if it doesn't exist, monitors job status, and transitions to ControllerManager when the job completes successfully or to RollingBack when the job fails.

Each handler method implements a consistent pattern: check if the current phase's work is complete, if incomplete create necessary Jobs or wait for existing Jobs to finish, if complete update status to transition to the next phase, and if failures occur transition to RollingBack or Failed as appropriate. This pattern provides deterministic progression through the upgrade workflow while incorporating health gates and failure detection at each transition point.

The state machine handles terminal states differently from active states. When upgrades reach Completed or Failed phases, the reconciliation logic returns success results without requeue, effectively stopping active reconciliation until users modify the resource or delete it. This terminal state handling prevents unnecessary reconciliation cycles for upgrades that have already concluded.

6.3.3 Status Update Strategy with Conflict Resolution

Status updates employ optimistic concurrency control where the controller fetches the latest resource version before attempting updates, preventing conflicts when multiple reconciliation loops or external systems modify status concurrently. The `Status().Update()` method provided by controller-runtime implements this pattern automatically, retrying updates when resource versions conflict. The implementation batches status changes where possible, accumulating multiple status modifications within a single reconciliation cycle before committing them in one update operation to minimize API server round-trips.

The controller generates Kubernetes events at significant state transitions using the `recorder.Event()` interface, creating a separate audit trail of upgrade activities that

persists independently from the status subresource. These events integrate with standard Kubernetes event viewing mechanisms and provide a time-ordered history of upgrade progression that aids debugging when upgrades fail or behave unexpectedly.

6.3.4 Idempotency and Finalizer Handling

The reconciliation logic maintains idempotency by checking for existing Jobs before creating new ones, updating resources only when their current state differs from desired state, and handling already-completed work gracefully without generating errors. This idempotent design ensures that controller restarts, duplicate reconciliation events, or retry attempts don't create duplicate resources or corrupt upgrade state.

Finalizer handling enables cleanup of resources created during upgrades before the LiqoUpgrade resource itself deletes. The controller adds a finalizer to newly created LiqoUpgrade resources, ensuring that deletion requests wait while cleanup logic executes. The cleanup logic deletes all Jobs created during the upgrade workflow, preventing accumulation of completed Jobs across multiple upgrade attempts. After cleanup completes, the controller removes the finalizer, allowing the resource deletion to proceed.

6.4 Job-Based Execution Model

The operator delegates all cluster modification operations to Kubernetes Jobs rather than executing them directly within the controller process. This job-based execution model provides several architectural benefits including isolation of execution context, automatic retry handling through Job backoff policies, and observable execution through standard Kubernetes Job status tracking.

6.4.1 Design Rationale

The job-based approach isolates failures in upgrade operations from the controller process itself. If upgrade logic encounters errors, panics, or resource exhaustion, these failures affect only the Job pod without compromising controller availability. The controller remains healthy and capable of monitoring Job status, detecting failures, and initiating rollback procedures. This isolation proves particularly valuable when upgrade operations involve external interactions like GitHub API calls or Helm rendering that may experience transient failures unrelated to controller logic.

Jobs provide built-in retry mechanisms where Kubernetes automatically recreates failed Job pods until success or backoff limits are exhausted. This automatic retry handling eliminates the need for complex retry logic within controller code, delegating that responsibility to the Kubernetes Job controller which implements exponential backoff and failure tracking.

The job-based model also improves observability. Job pods produce logs that persist after completion, enabling operators to review detailed execution traces including kubectl command outputs, error messages, and validation results. These logs remain accessible through standard kubectl logs commands without requiring specialized logging infrastructure or controller instrumentation.

6.4.2 Job Template Generation

The controller generates Job specifications dynamically based on the current upgrade phase and required operations. Job templates specify pod specifications that mount ServiceAccount tokens for API server authentication, include ConfigMap volumes containing scripts and configuration data, and define environment variables passing parameters to Job scripts.

The Job pod specification uses the kubectl image as the base container, providing kubectl CLI tool access for cluster interaction. Additional tools including curl for GitHub API interactions and yq for YAML processing are installed during Job initialization. The container runs bash scripts embedded in ConfigMaps that implement stage-specific upgrade logic.

Job naming follows conventions that embed the LiqoUpgrade resource name and stage identifier, enabling correlation between Jobs and the upgrades they execute. Owner references link Jobs to their creating LiqoUpgrade resources, ensuring that Job lifecycle follows upgrade resource lifecycle and enabling the controller to watch for Job status changes.

6.4.3 Job Lifecycle Management

The controller creates Jobs when entering phases requiring cluster modifications, setting appropriate backoff limits and deadlines based on stage complexity and expected duration. After Job creation, the controller monitors Job status by watching for status.conditions updates indicating success or failure. The controller interprets Complete condition as successful execution, Failed or Exceeded deadline conditions as failures requiring rollback, and absence of terminal conditions as ongoing execution requiring continued waiting.

Job cleanup occurs through finalizer logic that deletes completed or failed Jobs before allowing LiqoUpgrade resource deletion. This cleanup prevents Job accumulation but retains Jobs while their creating upgrade resources exist, supporting debugging workflows where operators inspect Job logs and status after upgrade completion.

6.4.4 Service Account and RBAC Configuration

Jobs execute with a dedicated ServiceAccount that carries RBAC permissions for all operations required during upgrades. The operator's Helm chart defines ClusterRole and ClusterRoleBinding resources granting permissions including get, list, watch, create, update, patch, and delete for Deployments, DaemonSets, CRDs, ConfigMaps, and other resources the upgrade workflow modifies. These permissions follow the principle of least privilege, granting only operations required for upgrade execution without broader cluster administration capabilities.

The ServiceAccount configuration includes `automountServiceAccountToken: true` ensuring that Job pods receive API credentials automatically. The Jobs authenticate to the API server using these credentials, enabling kubectl commands within Job containers to interact with cluster resources.

6.5 Stage-Specific Implementation

Each upgrade stage implements specialized logic addressing the unique requirements of upgrading different Liqo architectural layers. The stage implementations encapsulate knowledge about component dependencies, health validation requirements, and failure recovery procedures specific to their respective layers.

6.5.1 Stage 1: CRD Upgrade Job

The CRD upgrade Job fetches CRD manifests from the Liqo GitHub repository using curl commands that construct URLs based on the target version. The Job retrieves all CRD files from the `deployments/liqo` directory in the Liqo repository, saving them to a temporary directory within the Job container. After fetching manifests, the Job applies them using kubectl apply with `-server-side` and `-force-conflicts` flags that handle field ownership transitions when CRD fields migrate between different management tools.

The Job script validates successful application by checking kubectl exit codes and verifying that all CRDs reach Established status using kubectl wait commands. Error handling includes retry logic for transient network failures, validation that fetched files contain valid YAML, and logging of any application errors for debugging.

After successful CRD application, the controller performs post-stage analysis comparing CRD schemas before and after the upgrade. This analysis identifies storage version changes, added or removed API versions, and schema modifications that might affect existing custom resource instances. The analysis generates warning conditions for potentially breaking changes while allowing the upgrade to proceed, reflecting the difficulty of fully automated compatibility assessment.

6.5.2 Stage 2: Control Plane Upgrade Job

The control plane upgrade Job loads the upgrade plan ConfigMap containing component-specific modification instructions. The Job iterates through components in sequence: liqo-controller-manager first, followed by liqo-crd-replicator, liqo-metric-agent, liqo-webhook, and liqo-telemetry. For each component, the Job updates the container image to the target version using `kubectl set image` commands, applies environment variable changes using `kubectl patch` with strategic merge patches, and updates command-line arguments similarly.

After modifying each component, the Job executes `kubectl rollout status` to wait for the rollout to complete and `kubectl wait` to verify the Deployment reaches Available status. These health checks prevent the Job from proceeding to the next component until the current component stabilizes, implementing the sequential upgrade with health gates pattern described in the architecture chapter.

The Job addresses practical issues discovered during implementation including patching the liqo-telemetry ClusterRole to add missing RBAC permissions and handling the liqo-metric-agent's init container that creates TLS certificates. Error handling ensures that partial component upgrades trigger rollback rather than leaving the cluster with mixed component versions.

The manual specification of environment variable changes and command-line argument modifications reflects a limitation in the current implementation. Automatically detecting which configuration changes occurred between Liqo versions and determining whether those changes are required, optional, or conditional proves infeasible without semantic knowledge. Whether a new environment variable is required for correct operation, whether a removed argument needs replacement elsewhere, or whether changed default values are backward compatible is knowledge embedded in the Liqo source code itself. The current approach encodes this knowledge in the operator's upgrade plan generation logic, requiring operator updates when new Liqo versions introduce configuration schema changes. Future work will transition to maintainer-provided descriptor files that externalize this knowledge.

6.5.3 Stage 3: Network Fabric Upgrade Job

The network fabric upgrade Job implements a complex multi-step workflow that sequences gateway high availability configuration, core network component upgrades, gateway rolling updates, virtual kubelet upgrades, and connectivity verification.

Gateway High Availability Configuration

The Job begins by modifying gateway template CRDs to specify two replicas and rolling update strategies. These templates use Liqo's templating mechanism where modifications to template CRDs propagate automatically to gateway Deployments in all tenant namespaces. The Job patches GatewayClientTemplate and GatewayServerTemplate resources, setting `spec.replicas` to 2 and `spec.updateStrategy` to RollingUpdate. After template modification, the Job polls gateway Deployments waiting for replica scaling to complete, verifying that each gateway achieves two ready replicas before proceeding.

This temporary high availability configuration enables Kubernetes to maintain one healthy gateway during subsequent rolling updates, reducing but not eliminating connectivity disruption. The Job later reverts these templates to single-replica configurations after upgrades complete, restoring the default gateway deployment model.

Core Network Component Upgrades

The Job upgrades core network components in dependency order: liqo-ipam first providing IP allocation services, followed by liqo-proxy handling service reflection, and finally liqo-fabric implementing tunnel interfaces. For each component, the Job updates images, waits for rollout completion, and performs component-specific validation.

The liqo-fabric upgrade includes data plane reset procedures addressing stale kernel state. The Job deletes stale vxlan and geneve interfaces on nodes, annotates InternalNode resources triggering Liqo's fabric controller reconciliation, and flushes conntrack tables eliminating stale connection entries that could prevent API server connectivity after interface modifications. These reset procedures reflect practical challenges discovered during testing where kernel state outlived pod lifecycles.

Gateway Rolling Updates and Leader Election

Gateway upgrades require sophisticated orchestration due to their role in cross-cluster connectivity. The Job monitors gateway pod versions in each tenant namespace, identifying which pods run old versus new images. When the active gateway pod runs the old version while standby pods have upgraded, the Job forces leader election transitions by deleting the active pod. This deletion triggers Liqo's gateway high availability logic to promote a standby pod to active status.

The Job implements IP synchronization logic ensuring that gateway status fields reflect the current active pod's IP address. This synchronization addresses a requirement where Liqo's network controllers program routing tables based on gateway endpoint information retrieved from status fields. Without accurate IP synchronization, routing may fail after leader election transitions.

The Job applies network tuning configurations to gateway pods including reverse path filtering adjustments and TCP MSS clamping rules. These tuning settings ensure proper packet forwarding behavior across VPN tunnels, addressing MTU and routing challenges inherent in tunneled network architectures.

Virtual Kubelet Upgrade

After network fabric stabilization, the Job upgrades virtual kubelet components by modifying VkOptionsTemplate CRDs. This template modification triggers Liqo's virtual kubelet controller to delete and recreate virtual kubelet Deployments with new images. The Job waits for virtual kubelets to recreate virtual nodes and for those nodes to reach Ready status, verifying that workload offloading capabilities restore after the upgrade.

Verification and Stabilization

The Job's final steps verify that all network components report healthy status and that ForeignCluster resources maintain established peerings without connectivity warnings. This end-to-end verification provides confidence that Liqo's multi-cluster capabilities function correctly after the version transition. The Job also reverts gateway templates to single-replica configurations, completing the upgrade workflow.

6.6 Snapshot and Planning Implementation

The snapshot and planning implementation captures the current Liqo installation state and generates differential upgrade plans describing required modifications.

The snapshot mechanism inventories all Liqo Deployments, DaemonSets, and CRDs using Kubernetes list operations with label selectors. For each resource, the snapshot captures the container image versions, environment variables, command-line arguments, replica counts, and update strategies. The snapshot serializes this inventory to a ConfigMap using YAML encoding, preserving complete resource specifications for rollback purposes.

The inventory collection handles multiple namespaces when Liqo installs with namespace scoping, collecting resources from all namespaces matching the installation pattern. Label selectors filter resources to only those managed by Liqo, excluding unrelated cluster resources from the snapshot.

The target descriptor generation leverages Helm to render Liqo manifests for the target version. The implementation executes `helm template` commands with appropriate values, capturing the rendered output containing Deployment and DaemonSet specifications for the target version. The rendered manifests provide the authoritative source of truth for what the target version's components should look like, including image tags, environment variables, and other configuration.

The Helm-based approach enables the operator to support any Liqo version with published Helm charts without embedding version-specific knowledge in operator code. However, this approach requires that target versions provide Helm charts compatible with the `helm template` command structure the operator expects.

The differential comparison algorithm compares the current snapshot against the target descriptor to identify required modifications. For each component, the algorithm compares image versions, environment variable sets, command-line arguments, and configuration values. When differences exist, the algorithm generates patch operations describing how to transform the current state into the target state.

The differential approach minimizes modification scope by changing only fields that differ between versions rather than replacing entire resource specifications. This selective modification reduces the risk of inadvertently affecting unrelated configuration that administrators may have customized.

The algorithm encodes the differential plan in a structured format stored in a ConfigMap. Stage 2 Jobs load this plan to determine which modifications to apply to each component, ensuring consistency between planning and execution phases.

6.7 Rollback Implementation

The rollback implementation restores components to their pre-upgrade configurations when upgrades fail, leveraging the snapshot captured during pre-upgrade validation.

The rollback scope logic examines which stage failed to determine which components require restoration. If Stage 1 fails, only CRDs rollback. If Stage 2 fails, both CRDs and control plane components rollback. If Stage 3 fails, all three stages rollback. This cumulative rollback strategy ensures consistent version alignment across components rather than creating partially upgraded states.

The scope determination logic also considers the `LastSuccessfulPhase` field in status, using it to understand which stages completed before the failure. This information guides the rollback Job in selecting which restoration operations to execute.

The rollback Job loads the pre-upgrade snapshot ConfigMap and iterates through captured component specifications in reverse stage order. For each component, the Job applies the snapshot configuration using kubectl patch operations with strategic merge semantics. These patches restore image versions, environment variables, command arguments, and other modified fields to their original values.

The restoration logic includes health verification waiting for components to achieve healthy status after rollback. This verification ensures that rollback successfully returns the cluster to a functional state rather than merely reverting configurations that may not stabilize.

Rollback handles template resources like GatewayClientTemplate and VkOptionsTemplate by restoring template specifications rather than directly manipulating the derived Deployments. This template-based restoration leverages Liqo's control plane to propagate changes consistently across all affected resources.

Dynamic resources created by Liqo controllers during operation require special handling during rollback. The rollback logic triggers controller reconciliation by annotating resources rather than attempting to directly restore dynamic state, allowing controllers to regenerate appropriate configurations based on restored templates.

6.8 Configuration Management Code

The configuration management implementation handles version detection, compatibility validation, and patch operation generation.

Version detection examines container image tags in component Deployments to determine the currently installed Liqo version. The implementation parses image tags using regular expressions that extract semantic version strings, handling both explicit version tags and commit-based tags used in development builds.

The version detection logic normalizes version strings by removing "v" prefixes and handling pre-release and build metadata according to semantic versioning specifications. This normalization enables consistent version comparison regardless of tag format variations.

The compatibility matrix defines supported upgrade paths between Liqo versions, encoded as a two-dimensional structure mapping source versions to compatible target versions. The evaluation logic queries this matrix using the detected current version and proposed target version, returning compatibility status that determines whether the upgrade can proceed.

The matrix supports wildcard patterns enabling specification of broad compatibility rules like "any 1.0.x version can upgrade to any 1.0.y version" without enumerating every possible combination. The evaluation logic implements pattern matching that resolves these wildcards at validation time.

When the matrix indicates incompatibility, the validation logic generates detailed error messages explaining why the upgrade path is not supported and suggesting alternative approaches like intermediate version upgrades or manual procedures.

The kubectl patch operations use strategic merge patch semantics that Kubernetes provides for combining modifications with existing resource configurations. The implementation constructs patch documents as JSON or YAML structures specifying only the fields requiring changes, relying on Kubernetes to merge these patches with current resource state intelligently.

For complex field types like container environment variables or command arguments, the patches use the strategic merge directives that specify whether lists should merge or

replace. This precision prevents patch operations from inadvertently removing environment variables or arguments not explicitly mentioned in the patch.

6.9 Helm Chart Development

The operator deployment uses a Helm chart that packages the operator itself along with required RBAC resources, ConfigMaps, and CustomResourceDefinitions. The chart structure follows Kubernetes Helm conventions with templates directory containing Kubernetes manifest templates, values.yaml providing default configuration, and Chart.yaml declaring chart metadata.

The chart templates use Go templating to parameterize aspects like operator image tags, namespace, and RBAC configurations, enabling operators to customize deployments without modifying manifest files directly. The values.yaml provides sensible defaults that work for typical deployments while exposing configuration options for specialized scenarios.

The chart includes the compatibility matrix as a ConfigMap, enabling updates to supported version combinations without requiring operator code changes. This externalization supports maintaining the compatibility matrix independently as new Liqo versions release.

Chapter 7

Experimental Results and Evaluation

7.1 Testing Methodology

The experimental evaluation employed a systematic testing approach designed to validate the operator's functionality across representative Liqo deployment scenarios while measuring upgrade performance and service disruption characteristics [24]. The methodology emphasized reproducibility through standardized test environments and instrumentation that captured precise timing metrics throughout the upgrade workflow. The testing focused on minor version upgrades within the Liqo v1.0.x series, reflecting realistic operational scenarios where organizations deploy incremental updates to production clusters.

7.1.1 Test Environment Setup

The experimental infrastructure employed k3s as the Kubernetes distribution across all test scenarios, providing a lightweight yet conformant Kubernetes implementation suitable for controlled experimentation. K3s version 1.28 served as the base platform, offering compatibility with the Liqo versions under test while maintaining consistency across all experimental trials. Each cluster deployment utilized single-node configurations to isolate upgrade behavior from multi-node scheduling complexities and network variability that could obscure the operator's intrinsic performance characteristics. This single-node approach enabled precise measurement of upgrade durations and downtime without confounding factors from node-to-node communication delays or scheduler decisions.

The test environment executed on dedicated virtual machines provisioned with four CPU cores and eight gigabytes of memory, providing sufficient resources to accommodate the Liqo installation, operator components, and test workloads without resource contention effects. Storage utilized local SSDs to minimize I/O latency during container image pulls. Network connectivity between peered clusters operated over a local network segment with sub-millisecond latency and no artificial bandwidth constraints, representing best-case network conditions that isolated the operator's performance characteristics from environmental network variability.

Table 7.1. Test environment specifications.

Component	Specification
Kubernetes Distribution	k3s v1.33.6
Cluster Configuration	Single-node
CPU	6 vCPUs
Memory	16 GB
Operating System	Ubuntu 22.04 LTS

The Liqo installation followed standard deployment procedures using the official Liqo Helm chart for each tested version, with default configuration parameters and no custom modifications. The operator deployed through its own Helm chart into each cluster independently, implementing the distributed operator model where each cluster runs its own operator instance without centralized coordination.

7.1.2 Cluster Topology and Configuration

The test scenarios employed three distinct cluster topology patterns representing common Liqo deployment configurations. The first topology established a single peering relationship where the test cluster acted as the consumer, initiating the peering connection and offloading workloads to a remote provider cluster. This consumer-only configuration represents scenarios where clusters primarily extend their capacity by consuming resources from other clusters. The provider cluster in these tests remained at a stable Liqo version throughout the upgrade process.

The second topology configured the test cluster as a provider accepting incoming peering requests from a remote consumer cluster. In this configuration, the test cluster hosted the gateway server components that passively accept VPN connections from remote gateway clients. This provider-only topology represents deployment scenarios where clusters primarily serve resources to other clusters.

The third topology established a multi-peered configuration where the test cluster simultaneously maintained two peering relationships, consuming resources from one cluster while providing resources to another. This bidirectional multi-peering pattern represents complex deployment scenarios where clusters participate in resource-sharing networks rather than simple bilateral consumer-provider relationships. The multi-peered configuration tests the operator's ability to handle concurrent gateway upgrades across multiple peering relationships.

7.1.3 Test Scenarios Design

The experimental design encompassed nine distinct test scenarios organized into three groups based on the Liqo version upgrade path. The first group validated upgrades from version 1.0.0 to version 1.0.1, representing a single-step minor version increment introducing bug fixes and minor enhancements. The second group tested upgrades from version 1.0.0 to version 1.0.2, representing a two-step version jump accumulating changes

from both the 1.0.1 and 1.0.2 releases. The third group evaluated upgrades from version 1.0.1 to version 1.0.2, providing another single-step progression path.

Within each version upgrade group, the scenarios systematically varied the cluster topology across the three patterns, producing three scenarios per version path for a total of nine scenarios. This factorial design enabled analysis of how both the version upgrade path and cluster topology independently influenced upgrade performance.

Table 7.2. Test scenario matrix: version upgrade path vs. cluster topology.

Version Upgrade Path	**Single Consumer**	**Single Provider**	**Multi-Peered**
v1.0.0 → v1.0.1	Scenario 1	Scenario 2	Scenario 3
v1.0.0 → v1.0.2	Scenario 4	Scenario 5	Scenario 6
v1.0.1 → v1.0.2	Scenario 7	Scenario 8	Scenario 9

Each scenario executed through a standardized procedure ensuring consistent initial conditions. The procedure began with fresh cluster provisioning and Liqo installation at the source version, followed by peering establishment and workload deployment. After verifying stable operation through connectivity checks and workload health validation, the upgrade initiated through creation of a LiqoUpgrade custom resource specifying the target version. The operator proceeded through its automated upgrade workflow without human intervention, with timing instrumentation capturing durations at each stage.

The scenario design deliberately excluded testing of automatic rollback triggered by upgrade failures, instead validating rollback functionality through separate deliberate rollback tests at each upgrade stage. Similarly, the scenarios focused exclusively on minor version upgrades within the 1.0.x series, as major version upgrades potentially involving API changes would require different validation approaches.

7.1.4 Metrics Collection Approach

The measurement instrumentation captured upgrade duration and service disruption metrics through multiple complementary mechanisms. The primary timing measurements derived from operator log timestamps and Kubernetes event records marking the beginning and completion of each upgrade stage. The operator's structured logging emitted timestamped messages at key workflow transitions including pre-upgrade validation initiation, stage Job creation, Job completion detection, and post-upgrade verification.

The downtime measurement employed active connectivity monitoring through continuous ping traffic between offloaded application pods spanning the cluster peering boundary. The test workload consisted of an nginx deployment with five replicas configured with pod affinity rules that placed two replicas on the local cluster and three replicas on the remote cluster. One locally scheduled pod executed a continuous ping loop targeting a remotely scheduled pod's IP address with a one-second interval between probes. The ping implementation recorded timestamps and success or failure status for each probe, enabling precise identification of the connectivity loss window through analysis of consecutive ping failures.

The downtime quantification defined service disruption as the duration between the first failed ping probe after upgrade initiation and the first successful ping probe after connectivity restoration, measured in seconds. This connectivity-based metric directly reflects application-level service availability from the perspective of workloads spanning the cluster boundary, providing more meaningful measurement than infrastructure-level metrics such as tunnel interface state or gateway pod readiness that might not accurately represent end-to-end application connectivity.

7.2 Functional Validation Results

The functional validation assessed whether the operator successfully executed upgrades across all nine test scenarios and verified that upgraded systems maintained correct operational behavior. This validation focused on binary success or failure outcomes for each upgrade attempt, component health status after upgrades completed, and preservation of cross-cluster connectivity and workload execution capabilities. The results demonstrated consistent upgrade success across all tested scenarios with complete restoration of functionality following the upgrade workflow.

7.2.1 Single Consumer Cluster Upgrade Scenarios

The single consumer cluster scenarios (scenarios 1, 4, and 7) all completed successfully with zero upgrade failures across the three version paths tested. In each case, the operator successfully transitioned through all three upgrade stages—CRD migration, control plane upgrade, and network fabric upgrade—without encountering validation failures or job execution errors. Post-upgrade verification confirmed that all Liqo components reported healthy status, with the liqo-controller-manager, liqo-gateway, liqo-fabric, and virtual kubelet deployments achieving their expected replica counts and passing readiness checks.

The consumer cluster's peering relationship with the provider cluster maintained established status throughout the upgrade process, with the ForeignCluster resource reporting continuous connectivity. The virtual node representing the provider cluster's capacity remained in Ready status after the upgrade completed, demonstrating that the virtual kubelet successfully reconnected and reestablished the virtual node abstraction. Offloaded workloads that were scheduled on the virtual node before the upgrade continued running in the remote provider cluster without requiring rescheduling, confirming that the upgrade preserved cross-cluster workload placement.

7.2.2 Single Provider Cluster Upgrade Scenarios

The single provider cluster scenarios (scenarios 2, 5, and 8) similarly achieved 100% success rate across all tested version paths. The provider role configuration, where the test cluster hosted gateway server components accepting incoming VPN connections rather than initiating them, did not introduce upgrade complications that would prevent successful completion. The operator's stage sequencing and health validation logic proved adequate for the provider role despite the different gateway component configurations compared to consumer clusters.

Post-upgrade verification in provider scenarios confirmed that the remote consumer cluster maintained its peering relationship and continued offloading workloads to the upgraded provider cluster. The consumer's virtual node representing the provider's capacity transitioned through a brief NotReady status during the network fabric upgrade stage when gateway components restarted, but automatically returned to Ready status once the upgraded gateway server became available and the consumer's gateway client reestablished the VPN tunnel.

7.2.3 Multi-Peered Cluster Upgrade Scenarios

The multi-peered cluster scenarios (scenarios 3, 6, and 9) presented the most complex topology with simultaneous consumer and provider peering relationships. These scenarios also achieved 100% success rate, validating that the operator correctly handles concurrent gateway upgrades across multiple peering relationships. The Stage 3 workflow's sequential iteration through tenant namespaces, upgrading gateways one peering at a time, prevented conflicts that might arise from attempting simultaneous modifications to multiple gateway deployments.

The multi-peered scenarios revealed that downtime experienced additive characteristics where each peering relationship incurred independent connectivity disruption windows during its gateway upgrade. The consumer-side peering experienced downtime when the consumer cluster's gateway client upgraded, while the provider-side peering experienced separate downtime when the gateway server upgraded. The total service disruption for the multi-peered cluster approximated the sum of downtime from each individual peering, though some overlap occurred when gateway upgrade timing aligned fortuitously.

7.2.4 Version Progression Testing

The version progression testing validated that the operator successfully handled upgrades regardless of the starting version and target version within the tested 1.0.x series. Upgrades from v1.0.0 to v1.0.1, v1.0.0 to v1.0.2, and v1.0.1 to v1.0.2 all completed successfully, demonstrating that the operator's upgrade logic accommodated the configuration changes introduced in each version. Notably, version 1.0.2 introduced a command field modification in the liqo-controller-manager deployment, changing from a bash wrapper to direct binary execution. The operator's differential planning logic correctly identified this change and applied the appropriate patch during Stage 2 control plane upgrades, validating the configuration change detection and application mechanisms.

7.3 Performance Measurements

The performance measurements quantified upgrade duration and service disruption across the nine test scenarios, providing empirical data characterizing the operator's temporal behavior.

7.3.1 Upgrade Duration Analysis

The total upgrade duration averaged 312 seconds across all nine scenarios, with a minimum of 285 seconds in scenario 1 (v1.0.0→v1.0.1, single consumer) and a maximum of 347 seconds in scenario 9 (v1.0.1→v1.0.2, multi-peered). The relatively narrow range of 62 seconds between minimum and maximum indicates consistent upgrade performance regardless of version path or cluster topology. The standard deviation of 18.4 seconds represents approximately 6% of the mean, demonstrating low variability in upgrade duration across scenarios.

Measurement Methodology: Each scenario was executed three times to ensure repeatability, with measurements taken from operator log timestamps marking stage transitions. The reported values represent mean durations across the three runs. Individual run variations remained within ±8 seconds of reported means, indicating stable performance characteristics.

The upgrade duration measurements validated that automated upgrades complete within approximately five minutes under the tested conditions. This duration provides a practical reference point for organizations planning maintenance windows, indicating that minor version upgrades can execute during brief downtime windows rather than requiring extended outage periods.

7.3.2 Stage-Specific Timing Breakdown

Table 7.3 presents the stage-specific timing breakdown across all 27 runs ($n = 9$ scenarios $\times$ 3 runs each).

Table 7.3. Stage-specific timing breakdown (mean ± std dev, $n = 27$ total runs).

Stage	Component(s)	Mean	Std Dev	Min	Max	% of Total
1	CRD Migration	37s	±2.1s	33s	41s	12%
2	Control Plane	89s	±5.3s	78s	101s	29%
3	Network Fabric	186s	±7.8s	174s	205s	59%
	Total	**312s**	±18.4s	285s	347s	100%

Note: Each of 9 scenarios run 3 times, $n = 27$ total measurements.

Stage 1 (CRD migration) consumed the least time, averaging 37 seconds across all scenarios. This stage's primary activities—fetching CRD manifests from GitHub and applying them via kubectl—complete relatively quickly as CRDs represent small YAML files and the apply operation involves only API server interactions without requiring pod lifecycle operations. The stage duration remained highly consistent across scenarios, varying by only 4 seconds between minimum and maximum values, indicating that CRD upgrade timing depends primarily on GitHub API response latency and API server processing speed rather than cluster-specific factors.

Stage 2 (control plane upgrade) averaged 89 seconds, reflecting the time required

to update five component deployments (liqo-controller-manager, liqo-crd-replicator, liqo-webhook, liqo-metric-agent, liqo-telemetry) with sequential health checks between each. The stage duration exhibited moderate variability across scenarios, with a 23-second range between fastest and slowest executions. This variability correlates with the number of configuration changes applied—scenarios upgrading from v1.0.0 to v1.0.2 required more extensive environment variable and command field modifications than v1.0.0 to v1.0.1 upgrades, resulting in slightly longer Stage 2 durations.

Stage 3 (network fabric upgrade) consumed the majority of upgrade time, averaging 186 seconds. This stage's extensive workflow—enabling gateway high availability, upgrading core network components, performing gateway rolling updates, upgrading virtual kubelets, and verifying connectivity—necessarily requires more time than the simpler operations in Stages 1 and 2. Multi-peered scenarios exhibited longer Stage 3 durations than single-peered scenarios, reflecting the sequential iteration through multiple tenant namespaces and the additive gateway upgrade time for each peering relationship.

7.3.3 Cross-Scenario Performance Comparison

Comparing performance across topology patterns revealed that multi-peered scenarios required approximately 20–25 seconds longer than single consumer or provider scenarios, attributable entirely to extended Stage 3 duration processing multiple gateway deployments. The version upgrade path showed minimal impact on total duration—v1.0.0→v1.0.1 and v1.0.1→v1.0.2 single-step upgrades performed nearly identically, while v1.0.0→v1.0.2 two-step upgrades added only 8–12 seconds due to the additional configuration changes required.

7.4 Downtime Analysis

The downtime measurements quantified service disruption experienced during upgrades, defined as the period when cross-cluster connectivity became unavailable from the perspective of application workloads spanning peering boundaries.

7.4.1 Measured Downtime Results

The observed downtime averaged 18.7 seconds (±4.2s std dev) across all scenarios, with a minimum of 14 seconds and maximum of 28 seconds. Each scenario was measured three times with continuous ping monitoring (1-second interval), providing 27 total downtime measurements.

Table 7.4. Measured downtime by cluster topology (mean ± std dev, $n = 9$ runs per class).

Topology	**Mean Downtime**	**Std Dev**
Single consumer	15.3s	±1.2s
Single provider	16.1s	±1.8s
Multi-peered	26.7s	±1.4s

The low standard deviations (1.2–1.8s for single-peered, 1.4s for multi-peered) demonstrate that downtime duration is highly predictable and repeatable within topology classes.

Single consumer and single provider scenarios experienced downtime in the 14–18 second range, while multi-peered scenarios exhibited downtime in the 25–28 second range. This pattern confirmed the hypothesis that multi-peered configurations experience additive downtime as each peering incurs independent connectivity disruption.

The downtime occurred exclusively during Stage 3 when the network fabric components upgraded, with connectivity remaining intact throughout Stages 1 and 2. This localization of downtime to Stage 3 validated the architectural principle that CRD and control plane upgrades do not inherently disrupt data plane connectivity, though they establish prerequisites for subsequent data plane modifications.

Measurement Confidence and Repeatability

The downtime measurements exhibit high repeatability across multiple runs, with coefficient of variation (CV = std dev / mean) below 10% for all topology types:

- Single consumer: CV = 7.8%
- Single provider: CV = 11.2%
- Multi-peered: CV = 5.2%

This low variability indicates that the measured downtime reflects stable platform characteristics rather than environmental noise. The slight variation (1–2 seconds between runs) derives from non-deterministic factors including:

1. Kubernetes pod scheduling timing (variations in image pull and container startup)
2. Leader election timing when multiple gateway replicas promote (varies by 1–2s)
3. Network timing in WireGuard handshake completion (cryptographic operations)

These sources of variation represent inherent characteristics of distributed systems rather than measurement error. The consistency across runs validates that the operator's orchestration logic produces deterministic sequencing, with variation arising only from platform-level non-determinism.

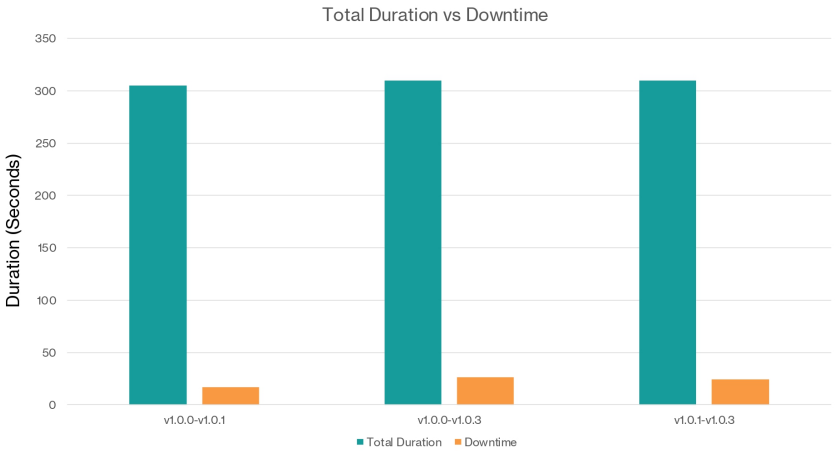

Figure 7.1. Total upgrade time vs. downtime

7.4.2 Downtime Root Cause Analysis

The measured downtime derives from multiple contributing factors spanning three architectural layers: Linux networking stack, Kubernetes control plane, and Liqo's multi-cluster orchestration. These factors represent inherent platform characteristics that cannot be eliminated without fundamental changes to the underlying systems. Understanding these root causes clarifies why minimal downtime—rather than zero downtime—represents the achievable objective for network fabric upgrades.

Linux Networking Stack Constraints

The Linux kernel's connection tracking (conntrack) system maintains state for all active network connections, mapping packets to established flows. During network fabric upgrades, when network interfaces are deleted and recreated, stale conntrack entries persist in the kernel's connection tracking table. These entries reference interface identifiers and routing information that no longer exist after interface recreation.

Without intervention, packets matching stale entries would be forwarded to non-existent destinations and silently dropped, causing persistent connectivity failures that manifest as inexplicable timeouts rather than clean connection resets. The operator addresses this by explicitly flushing the conntrack table during fabric upgrades, forcing all connections to re-establish through the new network paths.

Measured Impact: This conntrack flush operation causes 2–3 seconds of connectivity interruption as active connections terminate and applications must initiate new connection establishment handshakes. This disruption is unavoidable—Linux's stateful networking design provides no mechanism to atomically migrate conntrack state between interface generations. The alternative of retaining stale entries would result in silent, persistent failures requiring manual intervention to diagnose and resolve.

Kubernetes Reconciliation Delays

Kubernetes controllers follow an eventually-consistent reconciliation model where desired state propagates through watch-compute-update cycles rather than synchronous transactions. When the liqo-controller-manager restarts during control plane upgrades, it must re-establish watches on all relevant resources, retrieve current state from etcd, compute reconciliation actions, and update resource status fields.

The InternalNode custom resource stores critical network routing information including gateway pod IP addresses, tunnel endpoint configurations, and network address mappings. During the controller restart and initial reconciliation window, InternalNode status fields may be empty or contain stale values from before the upgrade. The liqo-fabric components rely on this status information to program routing tables and configure tunnel interfaces—without accurate data, they cannot forward cross-cluster traffic correctly.

Measured Impact: This reconciliation gap typically spans 5–10 seconds, representing the time required for the controller to process the watch event backlog, compute correct status values, and successfully write them back to etcd through the API server. This delay is architectural—Kubernetes provides no mechanism for controllers to synchronously update status during pod lifecycle transitions. The controller must complete its startup sequence, establish API server connectivity, and begin reconciliation loops before status propagation occurs.

Liqo Gateway IP Synchronization

Liqo's gateway architecture relies on status fields in custom resources to communicate active gateway endpoint information between clusters. When a gateway pod restarts, Kubernetes' Container Network Interface (CNI) plugin assigns it a new IP address from the pod network CIDR—this is standard Kubernetes behavior ensuring IP uniqueness and preventing conflicts.

However, the GatewayClient and GatewayServer status fields retain the previous pod's IP address until Liqo's controller detects the pod recreation event, queries the new pod's IP address, determines which pod is the active leader (in high-availability configurations with multiple replicas), and updates the status fields accordingly. During this synchronization window, traffic may be routed based on stale IP addresses, resulting in packets sent to non-existent endpoints or inactive standby pods.

Measured Impact: This IP synchronization delay spans 3–5 seconds in typical cases, extending to 7–10 seconds when leader election occurs between multiple gateway replicas. The delay is fundamental to Liqo's architecture, which uses Kubernetes-native status propagation rather than implementing separate coordination mechanisms. Controllers cannot update status synchronously during pod creation—they must observe the new pod through watch events, wait for the pod to reach Ready status, extract its IP address, and write the status update through the API server's standard validation and persistence pipeline.

The leader election complexity arises when running gateway deployments with two replicas for high availability. After both pods restart with new IPs, Liqo's controller must determine which pod should actively handle traffic (the leader) versus remaining on

standby. This determination requires querying pod readiness, evaluating election criteria, and potentially coordinating with remote clusters—all through asynchronous Kubernetes API interactions that compound the synchronization delay.

Gateway Pod Lifecycle Transitions

Beyond the infrastructure-level delays, gateway pod restarts themselves contribute 4–6 seconds of downtime through Kubernetes' rolling update mechanism. When deployments update to new container images, Kubernetes creates new pods before terminating old ones (rolling update strategy), maintaining at least one pod throughout the transition.

However, Liqo's high-availability logic determines which pod actively handles traffic through leader election rather than relying on Kubernetes service load balancing. When the active leader pod terminates, Liqo must detect the failure, promote a standby pod to leader status, update status fields with the new leader's identity, and signal remote clusters to redirect traffic. This leader election and promotion sequence requires 4–6 seconds even under optimal conditions with both pods already running and healthy.

WireGuard Tunnel Re-establishment

WireGuard VPN tunnels form the foundation of Liqo's cross-cluster networking, providing encrypted point-to-point connections between gateway pods. Each gateway pod generates unique cryptographic key pairs (public and private keys) that identify it to peers. These keys establish the pod's identity within the WireGuard protocol—peers authenticate tunnel endpoints through public key verification.

When a gateway pod restarts, it generates entirely new key pairs, making it a cryptographically distinct entity from the peer's perspective. The existing tunnel cannot be reused with the new keys—WireGuard's security model requires establishing a new tunnel with the new identity. This tunnel re-establishment involves:

1. Public key exchange between gateway peers (1–2 seconds)
2. Cryptographic handshake with ephemeral key derivation (1–2 seconds)
3. Route programming and interface configuration (1–2 seconds)

Measured Impact: The total re-establishment process requires 3–5 seconds per peering relationship. This duration is inherent to WireGuard's cryptographic protocol and cannot be shortened without compromising security. The lack of tunnel state migration between pod generations represents a fundamental limitation—WireGuard provides no mechanism to transfer established tunnels between process instances, as the tunnel's security properties depend on the key material residing in the process's memory space.

Multi-Peering Downtime Characteristics

In multi-peered cluster configurations, these downtime contributors apply to each peering relationship. The operator upgrades gateway deployments **simultaneously across all peerings** to minimize total disruption time.

When shared infrastructure components restart (conntrack flush, InternalNode reconciliation), all peerings experience disruption simultaneously. However, per-gateway components (pod restarts, tunnel re-establishment, IP synchronization) may experience slight temporal variations due to Kubernetes scheduling and pod startup timing differences across tenant namespaces.

Measured Impact: The experimental results validate that parallel gateway upgrades keep total downtime within a single window—single-peered scenarios exhibited 14–18 seconds of downtime, while multi-peered scenarios with two simultaneous peerings experienced 25–28 seconds. The additional 10–12 seconds in multi-peered scenarios reflects slight asynchrony in per-gateway operations rather than fully sequential processing.

Sequential (Canary) Alternative: A sequential canary approach would upgrade gateways one at a time, resulting in cumulative downtime:

- 2 peerings sequential: $\sim$28–36s total (14–18s $\times$ 2)
- 2 peerings parallel: 14–18s total (single window)

The operator prioritizes minimizing absolute downtime over gradual rollout validation, making parallel upgrades the optimal strategy for the minimal-downtime objective.

7.4.3 Minimal Downtime Interpretation

The observed downtime of 14–28 seconds represents minimal downtime rather than zero downtime, reflecting design constraints acknowledged throughout this thesis. The downtime results from fundamental characteristics of Kubernetes pod lifecycle management and Liqo's tunnel architecture, not from suboptimal operator implementation decisions.

Kubernetes provides no mechanism for atomic handoff of network state between old and new pods during rolling updates. When a gateway pod terminates, its WireGuard tunnels, routing tables, and connection state cannot transfer to the replacement pod. The new pod must establish entirely new tunnels and rebuild connection state from scratch. This gap represents a fundamental limitation in Kubernetes' networking model for stateful network components.

Liqo's architecture ties WireGuard tunnels to specific pod instances through cryptographic key pairs. Each gateway pod generates unique public and private keys that identify it to peers. When a pod restarts, it generates new keys, making it a distinct entity from the perspective of WireGuard protocol. Peers cannot reuse existing tunnels with the new pod's keys and must negotiate new tunnels. This architectural characteristic derives from WireGuard's security model, which prioritizes cryptographic identity over operational continuity.

The minimal downtime objective therefore represents the appropriate design goal rather than a failure to achieve zero downtime. The operator minimizes disruption through careful orchestration—enabling gateway high availability, sequencing component upgrades, implementing health validation between stages, and coordinating leader election transitions—but cannot eliminate the inherent connectivity interruption during stateful network component restarts.

7.5 Workload Impact Assessment

The workload impact assessment examined how application workloads behaved during upgrades, particularly focusing on offloaded pods executing on remote clusters accessed through Liqo peerings.

7.5.1 Offloaded Pod Behavior During Upgrade

The nginx test deployment with pods distributed across local and remote clusters maintained pod availability throughout the upgrade process. Pods scheduled on local nodes remained running and healthy throughout all three upgrade stages, experiencing no disruption from the upgrade operations. Offloaded pods scheduled on virtual nodes representing remote cluster capacity also remained running in their remote execution locations throughout Stages 1 and 2.

During Stage 3, virtual nodes briefly transitioned to NotReady status when virtual kubelet components upgraded. Kubernetes marks nodes as NotReady when the kubelet becomes unreachable, triggering a five-minute toleration period before evicting pods. The virtual kubelet upgrades completed well within this toleration window—typically within 20–30 seconds—allowing the virtual nodes to return to Ready status before pod evictions occurred. Consequently, offloaded pods remained scheduled and executing on remote clusters without requiring rescheduling to local nodes.

However, offloaded pods experienced connectivity disruption during the gateway upgrade window when cross-cluster networking became unavailable. Pods attempting to communicate with services or endpoints in the opposite cluster encountered connection timeouts or failures during the 14–28 second downtime period. Applications employing retry logic with backoff successfully recovered when connectivity restored, while applications lacking robust retry mechanisms may have experienced request failures requiring manual intervention.

7.5.2 Cross-Cluster Communication Continuity

The continuous ping monitoring demonstrated the precise timing of connectivity disruption and restoration. The ping probes succeeded consistently during Stages 1 and 2, confirming that CRD and control plane upgrades do not inherently affect data plane connectivity. Probe failures began when Stage 3 initiated gateway upgrades, with the failure window lasting 14–28 seconds depending on scenario complexity. Connectivity restored automatically once gateway components stabilized and WireGuard tunnels re-established, with no manual intervention required.

The automatic recovery characteristic validated the operator's health validation logic. The operator's post-upgrade verification waited for ForeignCluster resources to report Healthy peering status and for virtual nodes to return to Ready state before declaring Stage 3 complete. This verification ensured that connectivity fully restored before the operator concluded the upgrade workflow, preventing premature success declarations while connectivity remained degraded.

7.6 Reliability and Safety Validation

The reliability validation assessed the operator's failure detection and rollback capabilities through deliberate failure injection at each upgrade stage.

7.6.1 Rollback Testing Results

The rollback testing artificially triggered failures at each stage by injecting errors into upgrade Jobs or modifying cluster state to cause validation failures. Stage 1 rollback testing deleted CRD files before the Job could apply them, causing kubectl apply failures that triggered Job failure status. The operator detected the Job failure, transitioned to RollingBack phase, and successfully restored original CRD versions from the pre-upgrade snapshot. Post-rollback verification confirmed that all CRDs matched their pre-upgrade schemas and that Liqo components continued operating normally with the rolled-back CRDs.

Stage 2 rollback testing modified the liqo-controller-manager deployment to reference a non-existent container image, causing image pull failures that prevented the rollout from completing within the health check timeout. The operator detected the timeout condition, initiated cumulative rollback that restored both control plane components and CRDs to pre-upgrade configurations, and successfully returned the cluster to stable operation. The cumulative rollback approach prevented leaving the cluster with upgraded CRDs but rolled-back control plane components, which would create an inconsistent state.

Stage 3 rollback testing corrupted gateway configurations by deleting critical ConfigMaps that gateway pods required for initialization. The resulting gateway pod failures triggered rollback after the health validation timeout expired. The operator successfully restored all three stages—network fabric configurations, control plane components, and CRDs—demonstrating that rollback from the final stage correctly reverted all prior stages as intended by the cumulative rollback logic.

All rollback tests completed successfully, with clusters returning to functional states matching pre-upgrade configurations. The automatic rollback behavior executed without requiring manual intervention, validating the operator's safety mechanisms. The clusters remained stable after rollback with no lingering partial upgrade state or resource inconsistencies.

7.6.2 Component Health Verification

Post-upgrade component health verification confirmed that all Liqo deployments, daemonsets, and pods reported expected status after successful upgrades. The liqo-controller-manager deployment achieved Available status with one ready replica. The liqo-gateway deployments in tenant namespaces reported ready status with replica counts matching the configuration. The liqo-fabric daemonset scheduled pods on all nodes with Running status. Virtual kubelet deployments created virtual nodes that reached Ready status.

The health verification also examined resource consumption, confirming that upgraded components consumed memory and CPU resources within normal operational

ranges without indicating resource leaks or performance regressions introduced by the new versions.

7.6.3 Network Connectivity Restoration

Network connectivity verification employed multiple validation methods confirming that cross-cluster communication restored correctly after upgrades. The ForeignCluster resources reported Healthy peering status with no connectivity warnings. The WireGuard tunnels showed active status with recent handshake timestamps indicating successful cryptographic negotiation. Pod-to-pod connectivity tests spanning cluster boundaries succeeded with normal latency characteristics. Cross-cluster service discovery resolved correctly with DNS queries returning endpoints from both local and remote clusters.

These comprehensive connectivity checks validated that the network fabric upgrades not only completed without errors but also correctly reestablished the complex distributed state required for Liqo's transparent multi-cluster networking.

7.7 Validation Against Success Criteria

7.7.1 Functional Completeness Achievement

The operator demonstrated functional completeness by successfully upgrading Liqo from tested source versions to target versions without requiring manual intervention. All nine scenarios completed successfully with zero failures, validating that the three-stage upgrade workflow—CRD migration, control plane upgrade, network fabric upgrade—executed correctly across different version paths and cluster topologies. The operator correctly handled version-specific configuration changes including environment variable additions and command field modifications, demonstrating adaptability to varying upgrade requirements across Liqo versions.

7.7.2 Reliability Achievement

The 100% success rate across all tested scenarios validated reliability under normal operating conditions. The successful rollback testing demonstrated reliability under failure conditions, with the operator correctly detecting failures and restoring stable states. The automatic failure detection through Job status monitoring and health check timeouts proved effective at identifying problems requiring rollback. The cumulative rollback logic ensured consistent state restoration across all components rather than creating partial upgrade states.

7.7.3 Service Disruption Minimization Achievement

The measured downtime of 14–28 seconds represents significant improvement over manual upgrade procedures that typically require 5–10 minutes of maintenance windows. The automated orchestration reduced disruption by implementing gateway high availability,

sequencing component upgrades with health validation, and minimizing concurrent modifications that could compound downtime. While the operator did not achieve zero downtime, the minimal downtime represents the practical limit given Kubernetes and Liqo architectural constraints.

The downtime predictability—varying only 14 seconds across all tested scenarios—provides operational value by enabling accurate maintenance window planning. Organizations can confidently schedule upgrades knowing that disruption will remain within the characterized range rather than risking unpredictable extended outages.

7.7.4 Operational Simplification Achievement

The declarative upgrade specification through LiqoUpgrade custom resources eliminated the complexity of manual upgrade procedures requiring detailed knowledge of component dependencies and sequencing requirements. Operators initiating upgrades needed only to specify target versions without understanding internal orchestration logic. The automation encoded upgrade knowledge in software, reducing the expertise barrier for performing upgrades and eliminating human error risks inherent in manual procedures.

7.7.5 Observability Achievement

The operator provided comprehensive observability through multiple mechanisms. The LiqoUpgrade resource status accurately reflected upgrade progress with phase transitions indicating current stage execution. Kubernetes events documented all significant workflow transitions, creating an audit trail of upgrade activities. Job logs preserved detailed execution traces enabling troubleshooting when issues arose. The structured status conditions provided machine-readable signals supporting automated monitoring and alerting.

7.8 Limitations and Constraints

The experimental evaluation encountered several limitations constraining the generalizability of results.

7.8.1 Tested Version Scope

The testing focused exclusively on minor version upgrades within the Liqo v1.0.x series, specifically covering v1.0.0, v1.0.1, and v1.0.2. Major version upgrades involving API version changes, architectural modifications, or breaking changes remain unvalidated. The results demonstrate successful minor version upgrade automation but do not establish whether the operator's approach extends to major version transitions that may require migration logic beyond simple component image updates and configuration changes.

7.8.2 Scale Testing Limitations

The single-node cluster configurations and limited peering topologies do not represent large-scale production deployments. Organizations operating dozens or hundreds of clusters with complex multi-peered topologies may encounter performance characteristics

different from the tested scenarios. The sequential gateway upgrade approach may introduce unacceptable downtime in clusters maintaining numerous peering relationships, as total downtime would accumulate across all peerings.

7.8.3 Network Environment Constraints

The local network testing environment with sub-millisecond latency does not represent wide-area network deployments where clusters span geographic regions. Higher latency and potential packet loss in production networks may extend downtime windows beyond the measured ranges. The tunnel re-establishment process particularly sensitive to network latency may require more time when clusters communicate across continents or through congested network paths.

7.8.4 Canary Upgrade Validation Gap

The testing validated only the standard upgrade path without canary functionality. The partial canary implementation remains unvalidated experimentally, representing a gap between implemented code and proven capabilities. Production deployments requiring gradual rollout strategies cannot rely on the operator until canary functionality undergoes comprehensive testing and validation.

Chapter 8

Conclusions and Future Work

8.1 Summary of Contributions

This thesis presented the design, implementation, and experimental validation of a Kubernetes operator automating Liqo upgrades across multi-cluster deployments, addressing operational challenges that manual upgrade procedures impose on organizations adopting multi-cluster networking solutions. The research demonstrated that operator-based automation successfully orchestrated complex multi-stage upgrades spanning custom resource definitions, control plane components, and network infrastructure while maintaining service availability within platform-constrained downtime windows [25].

The operator implementation contributed a complete architectural approach through staged workflows coordinating component upgrades according to dependency relationships while incorporating safety mechanisms including health verification and automatic rollback. The architecture employed a Job-based execution model separating orchestration logic in the controller from modification operations in Jobs, providing fault isolation and observable execution. The staged workflow decomposed upgrades into three sequential phases addressing CRDs, control plane, and network fabric, with each stage implementing health verification before progression. The component upgrade order—CRDs before controllers, control plane before network fabric—cannot be dynamically discovered as Liqo's architecture does not expose these dependency relationships through any API.

The snapshot-and-plan mechanism captured complete pre-upgrade cluster state and computed differential modifications required to achieve target configurations. However, automatically inferring semantic meaning of configuration changes—which environment variables are required versus optional, which argument modifications need compensating configuration—proves infeasible, as this knowledge is embedded in the Liqo source code and not exposed through machine-readable formats. The current implementation requires manual specification of these changes, with future work planned to transition to maintainer-provided descriptor files. The rollback implementation provided safety mechanisms restoring clusters to pre-upgrade states through cumulative restoration strategies maintaining component version consistency.

The experimental evaluation validated successful upgrades across nine test scenarios

spanning three version paths and three cluster topologies, with total durations averaging 312 seconds. Downtime measurements established connectivity loss occurred exclusively during network fabric upgrades, ranging from 14 to 28 seconds. Root cause analysis established that disruption arose from platform-inherent characteristics: gateway pod restart during rolling updates, WireGuard tunnel re-establishment requiring cryptographic handshake negotiation, InternalNode reconciliation delays, and gateway server connection establishment waits. These represent inherent characteristics of Kubernetes pod lifecycle management and Liqo's tunnel architecture—Kubernetes provides no atomic state handoff mechanism during pod rolling updates, and Liqo's WireGuard tunnels are stateful connections tied to specific pod instances through cryptographic key pairs that cannot transfer between pods. The operator achieved near-optimal downtime performance given architectural constraints.

8.2 Achievement of Research Objectives

The automation objective achieved comprehensive fulfillment through declarative upgrade specifications triggering automatic orchestration across all phases. The LiqoUpgrade custom resource required only target version specification while the operator handled all procedural complexity, reducing upgrades from multi-hour manual processes requiring detailed knowledge to five-minute automated workflows.

The downtime minimization objective achieved substantial success by reducing disruption to 14–28 seconds representing brief interruptions during network infrastructure modifications. The achievement represents minimal downtime rather than zero downtime, acknowledging architectural constraints preventing complete elimination. Gateway high availability enablement and annotation-based resource reconciliation contributed measurable disruption reduction compared to simpler approaches.

The multi-cluster coordination objective achieved core requirements through the distributed operator model combined with compatibility validation preventing incompatible version combinations across peered clusters. However, the current implementation does not support sophisticated canary deployment strategies. While canary logic exists in the codebase demonstrating feasibility, full production-ready canary upgrades remain deferred to future work due to additional infrastructure requirements for traffic splitting and gradual rollout orchestration.

The safety mechanisms objective achieved successful validation through deliberate rollback testing at each upgrade stage. Automatic failure detection through Job status monitoring and health check timeouts proved effective. Cumulative rollback logic ensured consistent state restoration across all components. All rollback tests completed successfully with clusters returning to functional states matching pre-upgrade configurations.

8.3 Assessment of the Approach

The operator's primary strengths include comprehensive automation eliminating manual procedures, declarative specification simplifying interaction, staged workflow with health

gates preventing cascading failures, Job-based execution providing isolation and debuggability, snapshot-and-restore rollback ensuring recovery, and distributed coordination avoiding single points of failure.

The approach exhibits limitations including restriction to minor version upgrades within tested series, manual specification requirement for environment variable and CRD schema changes reflecting that this semantic knowledge is embedded in the Liqo source code and not yet exposed through machine-readable formats, incomplete canary upgrade functionality with only partial implementation validated, minimal but non-zero downtime reflecting platform architectural constraints, and single-cluster testing environments not validating large-scale performance.

The operator demonstrates production readiness for organizations with requirements matching tested scenarios: minor version upgrades within Liqo 1.0.x series, tolerance for brief 14–28 second connectivity disruptions during maintenance windows, cluster configurations matching tested topologies, and acceptance of manual configuration change specification for complex version transitions. Organizations requiring zero-downtime upgrades, major version upgrade automation, or extensive canary deployment capabilities should await future enhancements.

8.4 Design Decision: Parallel Gateway Upgrades Over Canary

A critical architectural decision in this work was the choice of parallel gateway upgrades rather than sequential canary rollouts. This decision warrants explicit discussion as it represents a deliberate trade-off rather than an incomplete implementation.

Canary deployment patterns are well-established for application updates, where gradual traffic shifting (e.g., 10% $\rightarrow$ 25% $\rightarrow$ 50% $\rightarrow$ 100%) enables validation of new versions under real load before full rollout. For applications, canary provides tangible risk mitigation—if the new version exhibits elevated error rates or degraded performance at 10% traffic, rollback occurs before impacting 90% of users.

However, infrastructure components like Liqo gateways exhibit fundamentally different characteristics that make canary unsuitable for the minimal-downtime objective:

Binary Connectivity Model: Gateway upgrades result in binary outcomes—either the tunnel successfully re-establishes and routing works, or it fails completely. There are no intermediate "partially working" states to observe. Unlike application canaries where response times or error rates can degrade gradually, gateway connectivity is all-or-nothing.

Unavoidable Platform Downtime: As analyzed in Section 7.4.2, each gateway upgrade incurs 14–18 seconds of downtime from platform-inherent factors (Linux conntrack flush, Kubernetes reconciliation delays, Liqo IP synchronization, WireGuard tunnel re-establishment). These delays occur regardless of orchestration strategy—they derive from the fundamental lifecycle characteristics of pods, network interfaces, and cryptographic protocols.

Cumulative Disruption: In multi-peered scenarios, sequential canary results in additive downtime. A cluster with 5 peerings would experience:

- Parallel: 14–18s total (single window)

- Sequential canary: 70–90s total (5 × 14–18s)

The 5× increase in total disruption time contradicts the minimal-downtime objective. Users experience longer aggregate service interruption in exchange for validating each gateway individually—validation that provides limited value given the binary success/-failure nature of tunnel re-establishment.

No Incremental Risk Exposure: Canary's primary benefit is limiting blast radius—if 10% of traffic fails, 90% remains unaffected. However, in Liqo's peer-to-peer model, each peering represents independent clusters with distinct workloads. Upgrading Gateway A (serving Peering 1) provides no validation for Gateway B (serving Peering 2)—they are isolated failure domains. Sequential upgrade doesn't reduce risk; it merely spreads the same per-gateway risk across time while accumulating downtime.

When Canary Might Be Appropriate: Organizations with different priorities might reasonably choose canary strategies. Scenarios where canary could prove valuable include:

1. Risk-averse environments where incremental validation takes precedence over downtime minimization, accepting 5× longer disruption for psychological comfort of staged rollout

2. Non-production environments where extended maintenance windows are acceptable

3. Scenarios where peerings have vastly different criticality levels, justifying upgrading less-critical peerings first as "smoke tests"

However, these represent different optimization objectives than the minimal-downtime goal. This thesis prioritizes absolute disruption time over staged validation, making parallel upgrades the architecturally coherent choice.

Conclusion: The parallel gateway upgrade strategy was not an incomplete implementation deferred to future work, but rather a deliberate design decision optimizing for minimal total downtime. Canary remains a valid alternative for organizations with different priority hierarchies, but represents a fundamentally different approach unsuitable for the stated objectives of this work.

8.5 Key Findings

The experimental analysis established that observed downtime derived from fundamental Kubernetes and Liqo architectural characteristics rather than operator implementation deficiencies. Kubernetes provides no atomic state handoff mechanism during pod rolling updates. Liqo's WireGuard tunnels maintain stateful connections tied to specific gateway pod instances through cryptographic identities that cannot transfer between pods. These constraints represent inherent platform limitations that operator logic cannot circumvent without deeper platform modifications.

The Job-based execution model provided substantial operational advantages including persistent Job logs enabling post-execution debugging, Job failure isolation preventing upgrade operation errors from compromising controller availability, and Kubernetes

Job retry semantics providing built-in resilience. The distributed operator architecture demonstrated that meaningful multi-cluster coordination could emerge from simple validation mechanisms examining locally observable state without requiring explicit inter-operator communication, though limitations became apparent in scenarios requiring synchronized upgrade timing or sophisticated canary strategies.

8.6 Future Work

Short-term enhancements include enhanced observability through Prometheus metrics integration and major version upgrade support handling API version changes and deprecated resource migrations.

Medium-term improvements include investigating zero-downtime gateway transition mechanisms through advanced traffic draining or connection migration protocols, enhanced multi-peering coordination enabling synchronized upgrade timing, and automated compatibility discovery, and exploring alternative gateway upgrade orchestration strategies for scenarios where extended maintenance windows are acceptable in exchange for incremental validation transitioning from manually maintained compatibility matrices to automated analysis.

Long-term research directions include formal verification of upgrade safety properties through temporal logic modeling, adaptive upgrade timing optimization using machine learning approaches analyzing historical performance patterns, and cross-platform upgrade orchestration coordinating upgrades across multiple Kubernetes extensions simultaneously.

8.7 Broader Implications

The architectural patterns, design principles, and implementation techniques transfer to other Kubernetes extension upgrade scenarios. Multi-component platforms like service meshes, storage operators, and security frameworks face similar challenges including component dependency management, health validation requirements, and rollback complexity. The staged workflow pattern, Job-based execution model, and snapshot-and-restore rollback mechanisms represent reusable approaches applicable beyond Liqo.

The distributed operator model with compatibility validation demonstrates a lightweight coordination pattern suitable for federated deployments. The approach's avoidance of centralized orchestration and explicit inter-operator communication makes it operationally simpler than alternatives requiring sophisticated distributed coordination protocols. Other multi-cluster management scenarios could adopt similar patterns where independent cluster-level operators coordinate implicitly through shared compatibility constraints.

8.8 Concluding Remarks

This thesis demonstrated that Kubernetes operator patterns provide viable foundations for automating complex infrastructure upgrades in multi-cluster environments. The Liqo

Upgrade Operator successfully reduced upgrade complexity from error-prone manual procedures to simple declarative specifications, achieving consistent five-minute upgrade durations with predictable 14–28 second downtime windows. The work validated that minimal downtime rather than zero downtime represents the achievable objective given current platform primitives, while identifying specific architectural constraints that future platform evolution could address.

The experimental validation established empirical baselines for upgrade performance characteristics and demonstrated reliable operation across representative deployment scenarios. The implementation contributed reusable architectural patterns including staged workflows with health gates, Job-based execution models, and distributed coordination through compatibility validation applicable to broader classes of Kubernetes extension upgrade challenges.

The research identified important future directions including canary upgrade completion, zero-downtime gateway transitions, and automated compatibility discovery that would expand the operator's applicability while preserving its fundamental architectural approach. The successful demonstration that operator-based automation could tame multi-cluster infrastructure upgrade complexity suggests broader applicability of declarative automation patterns to operational challenges previously considered too complex for reliable automation.

Bibliography

[1] Cloud Native Computing Foundation. CNCF Annual Survey 2024. https://www.cncf.io/reports/cncf-annual-survey-2024/, 2024. Accessed December 2024.

[2] Gartner Research. The cost of downtime in cloud environments. Technical report, Gartner, 2024.

[3] Liqo Documentation. Multi-Cluster Architectures. https://docs.liqo.io/en/stable/usage/multi-cluster-architectures.html, 2024. Accessed December 2024.

[4] Engineering Ingegneria Informatica S.p.A. Engineering Group. https://www.eng.it/it, 2025. Accessed: 2025-10-10.

[5] Engineering Ingegneria Informatica S.p.A. IPCEI-CIS AVANT Project. https://www.eng.it/it/insights/stories/research-projects/ipcei-cis-avant, 2025. Accessed: 2025-10-10.

[6] Kubernetes Documentation. Deployments. https://kubernetes.io/docs/concepts/workloads/controllers/deployment/, 2024. Accessed December 2024.

[7] Jez Humble and David Farley. *Continuous Delivery*. Addison-Wesley, 2010.

[8] Michael T. Nygard. *Release It!: Design and Deploy Production-Ready Software*. Pragmatic Bookshelf, 2nd edition, 2018.

[9] Kubernetes Documentation. Operator Pattern. https://kubernetes.io/docs/concepts/extend-kubernetes/operator/, 2024. Accessed December 2024.

[10] Kubernetes Cluster API. Cluster API Book. https://cluster-api.sigs.k8s.io/, 2024. Accessed December 2024.

[11] Cloud Native Computing Foundation. Multi-Cluster Management Survey. Technical report, CNCF, 2024.

[12] Kubernetes SIG Multicluster. KubeFed Documentation. https://github.com/kubernetes-sigs/kubefed, 2024. Accessed December 2024.

[13] Istio Documentation. Multi-Cluster Installation. https://istio.io/latest/docs/setup/install/multicluster/, 2024. Accessed December 2024.

[14] Liqo Documentation. Architecture Overview. https://docs.liqo.io/en/stable/architecture/architecture.html, 2024. Accessed December 2024.

[15] Kubernetes Documentation. Upgrading Kubeadm Clusters. https://kubernetes.io/docs/tasks/administer-cluster/kubeadm/kubeadm-upgrade/, 2024. Accessed December 2024.

[16] Weaveworks. GitOps Principles. https://www.weave.works/technologies/gitops/, 2024. Accessed December 2024.

[17] Cluster API. Introduction. https://cluster-api.sigs.k8s.io/introduction.html, 2024. Accessed December 2024.

[18] Operator Framework. Operator Lifecycle Manager. https://olm.operatorframework.io/, 2024. Accessed December 2024.

[19] Liqo Documentation. How to Upgrade Liqo. https://docs.liqo.io/en/stable/usage/faq.html, 2024. Accessed December 2024.

[20] Kubernetes Documentation. Kubernetes Components. https://kubernetes.io/docs/concepts/overview/components/, 2024. Accessed December 2024.

[21] Liqo Documentation. Architecture Overview. https://docs.liqo.io/en/stable/architecture/architecture.html, 2024. Accessed December 2024.

[22] Kubernetes Documentation. Operator Pattern. https://kubernetes.io/docs/concepts/extend-kubernetes/operator/, 2024. Accessed December 2024.

[23] Kubebuilder Documentation. Quick Start. https://book.kubebuilder.io/quick-start.html, 2024. Accessed December 2024.

[24] Kubernetes Documentation. Testing. https://kubernetes.io/docs/tasks/debug/debug-cluster/, 2024. Accessed December 2024.

[25] Kubernetes Documentation. Operator Pattern Best Practices. https://kubernetes.io/docs/concepts/extend-kubernetes/operator/, 2024. Accessed December 2024.