Politecnico di Torino

FACULTY OF ENGINEERING
Computer Engineering LM-32

Elemento

Advanced Kubernetes provisioning, scaling and operations through kOps library

Candidato:

Paolo Beci

Relatore:

Prof. Alessio Sacco

Correlatore:

Prof. Guido Marchetto

Anno Accademico 2024-2025

Abstract

Kubernetes has emerged as the de facto standard for deploying and managing applications that scale from a handful of users to millions, becoming a cornerstone of the modern cloud-native ecosystem. It makes highly scalable and resilient infrastructures possible when paired with cloud computing. However, Kubernetes adoption remains hindered by two major obstacles: a steep learning curve, particularly for small or non-specialized teams, and the persistent issue of vendor lock-in within cloud platforms. Vendor-specific technologies, proprietary APIs, and economic constraints often limit portability and complicate multi-cloud strategies, creating barriers to innovation and flexibility. This thesis examines the challenges of Kubernetes adoption and proposes solutions to simplify cluster provisioning and management. An extensive review of the state of the art was conducted, categorizing and comparing the most widely used provisioning approaches, including self-managed installers such as Kubeadm, automation frameworks like Kubespray, Infrastructure-as-Code (IaC) tools such as Terraform and Pulumi, and managed services offered by major providers, including Amazon Elastic Kubernetes Service (EKS), Google Kubernetes Engine (GKE), and Azure Kubernetes Service (AKS). Each approach was evaluated in terms of complexity, supported environments, and trade-offs, with special focus on open-source alternatives that promote transparency and flexibility. The core contribution of this work is the development of an Elemento–kOps plug-in that extends kOps, an open-source library widely adopted for provisioning production-grade Kubernetes clusters, for the Elemento cloud platform. Elemento, an Italian deep-tech provider, distinguishes itself through its AtomOS hypervisor, which enables hybrid and multi-cloud deployments. By introducing this plug-in, the project delivers a unified interface for cluster lifecycle management, offering an alternative that combines portability, simplicity, and scalability. Methodologically, the research involved analyzing the kOps codebase, adopting debugging and logging strategies to navigate its architecture, and leveraging test-driven exploration of APIs and provider-specific implementations. The integration was achieved through a translation layer between kOps and the Elemento Go library, ensuring the seamless provisioning of virtual machines, storage, and networking resources via declarative tasks. Evaluation was performed by comparing the provisioning speed, reliability, and code complexity of the Elemento integration with those of conventional cloud providers. The results demonstrate the technical feasibility and potential advantages of supporting Elemento within kOps. The integration not only simplifies the provisioning process but also enables access to a distributed network of smaller cloud providers, contributing to reduced vendor lock-in and fostering a more open, flexible ecosystem. At the same time, limitations were observed in terms of reliability testing, as well as the need for broader validation across diverse environments. In conclusion, this work highlights the importance of open-source tools in shaping the future of cloud-native infrastructure. The Elemento–kOps integration demonstrates how collaborative development can enhance the accessibility of Kubernetes while addressing structural challenges in the cloud landscape. Future improvements may include enhancing system autonomy, expanding provider interoperability, and contributing further to the open-source community to support sustainable, vendor-neutral cloud adoption.

Contents

1 Introduction and goal **9**
- 1.1 What is Elemento? . . . 9
- 1.2 Cloud freedom . . . 9
  - 1.2.1 Vendor lock-in scenario . . . 10
  - 1.2.2 Vendor lock-in types . . . 10
- 1.3 Simplifying Kubernetes Adoption . . . 11
  - 1.3.1 The CNCF Landscape . . . 11
- 1.4 The importance of open source . . . 13
  - 1.4.1 Mechanics of Open Source development . . . 13
  - 1.4.2 Why Open Source is Crucial for Software Development . . . 14

2 Background **17**
- 2.1 Containers as a deployment standard . . . 17
- 2.2 Kubernetes: History and Technical Functionality . . . 18
- 2.3 Core Components of Kubernetes . . . 19
- 2.4 The Role of SDKs in Application Development . . . 21

3 State of the art **23**
- 3.1 Kubernetes cluster provisioning tools and methods . . . 23
  - 3.1.1 Self-Managed Installers . . . 23
  - 3.1.2 Infrastructure-as-Code Tools . . . 25
  - 3.1.3 Managed Cloud Kubernetes Services . . . 26
- 3.2 Architecture and Core Functions of kOps . . . 28
  - 3.2.1 How kOps works . . . 28
  - 3.2.2 How to use it . . . 29

4 Methodology **32**
- 4.1 Mastering Large Project Architecture: Insights from kOps . . . 32
  - 4.1.1 The importance of the debugger and error codes . . . 32
  - 4.1.2 The approach adopted in this specific case . . . 33
- 4.2 Systems integration . . . 34
  - 4.2.1 Handling of asynchronous operations . . . 35
- 4.3 How the Elemento support on kOps could be very useful . . . 36

5 Work development **37**
- 5.1 The integration workflow for adding Cloud Providers in kOps . . . 37

5.2 How I iterated the testing and development 38
5.3 Interesting workflow management of async actions by Hetzner Cloud . 39
5.4 Creating Cluster Manifests and Configurations 39
5.5 Installation and startup of the cluster 42

6 Work evaluation **47**
6.1 Testing the Kubernetes provisioning through kOps 47
6.1.1 Provisioning speed comparison 47
6.1.2 Complexity compared to other provisioning methods 49
6.1.3 Operating cost of the cluster 50
6.2 How reliable is it? 51

7 Conclusions **52**
7.1 Outcome of the thesis 52
7.2 Future improvements 52
7.2.1 Extend to multi-AtomOS server scenario 52
7.2.2 Auto-scaling 53
7.2.3 Observability 53

Bibliography **54**

List of Figures

4.1 System integration with kOps 34
4.2 Elemento contribution to an unbounded Cloud environment 36

5.1 Elemento create VM flow 37
5.2 Elemento Electros VM list 42
5.3 Elemento Electros VM list detail 42
5.4 Elemento Electros storage list 43
5.5 Elemento Electros storage list detail 43
5.6 Ansible role in the systems architecture 45
5.7 Cluster nodes list from external host 46
5.8 Cluster pods list from external host 46

6.1 Cluster creation time comparison based on 2023 data - source [14] . . 48
6.2 Cluster creation time comparison based on dimension 48
6.3 K8s clusters pricing comparison based on 2023 data - source [14] . . . 50

Chapter 1

Introduction and goal

1.1 What is Elemento?

Elemento [8] is an Italian deep-tech startup revolutionizing the way businesses build and manage their cloud infrastructures. Designed with flexibility, simplicity, and future scalability in mind, Elemento empowers organizations to deploy tailored cloud environments that adapt seamlessly to their specific needs, whether in the public cloud, private cloud, or both.

At the core of Elemento's offering are two proprietary technologies: the Cloud Gateway [10] and AtomOS [9], a next-generation hypervisor. These innovations offer a unified and intuitive interface for orchestrating and managing computing, storage, and networking resources across diverse environments. With Elemento, IT teams can monitor, scale, and operate their entire hybrid or multi-cloud stack, both public and private, through a single and integrated platform.

Elemento enables truly seamless hybrid and multi-cloud operations. Businesses can dynamically leverage resources from over 230 data centers across the world's five leading public cloud providers, while also integrating their own on-premises or private infrastructure, all without compromising on performance, security, or control.

By abstracting the underlying cloud complexity, Elemento makes modern cloud infrastructure accessible, efficient, and manageable, unlocking new agility for enterprises and service providers alike.

1.2 Cloud freedom

Vendor lock-in is a situation in cloud computing where a customer becomes heavily dependent on a specific cloud provider due to various reasons, such as technical aspects, contract terms, or other factors. This dependency can result in limited options for the customer to switch to another provider, even if the current provider is not performing well [17].

Vendor lock-in can occur when a cloud provider uses proprietary technologies, formats, or interfaces that are not easily interoperable with other providers, making it difficult for the customer to migrate their applications and data. It can also result from exclusive licensing terms, complex integration requirements, or custom

configurations that tie the customer to the specific provider's ecosystem. This lack of portability and flexibility can limit the customer's ability to adapt to changing needs or take advantage of competitive offerings, potentially impacting cost-effectiveness and innovation.

1.2.1 Vendor lock-in scenario

Company *TechWay*, a rapidly growing Software as a Service (SaaS) provider, initially selected Cloud Provider XYZ to host their application and oversee their cloud infrastructure. They were drawn by competitive rates, dependable services, and most features aligned with their needs [17].

Over time, *TechWay* integrated its application with Cloud Provider XYZ's offerings, relying on databases, serverless functions, and AI services for data processing. However, challenges arose when their evolving needs exceeded Cloud Provider XYZ's AI services. They turned to Cloud Provider ABC, which offered advanced machine learning options like natural language processing and image recognition.

At this point, the vendor lock-in issue emerged. Shifting from Cloud Provider XYZ to Cloud Provider ABC would be intricate and time-intensive. They would need to modify their code, adapt data formats, and restructure workflows to align with Cloud Provider ABC. Furthermore, high data egress charges by Cloud Provider XYZ added to the migration expenses. *TechWay* also faced the hurdle of their proficient developers specializing in Cloud Provider XYZ's tools. Adapting to Cloud Provider ABC would require training or hiring new talent familiar with their ecosystem [17].

1.2.2 Vendor lock-in types

In cloud computing, vendor lock-in can manifest in various forms: technical, data, service, contractual, and more. These lock-in mechanisms are often the result of providers offering proprietary technologies, tightly integrated services, or unique configurations that are not easily transferable across platforms. As customers adopt these specialized features, they gradually become dependent on the provider's ecosystem. This dependence significantly increases the cost and complexity of migration, requiring substantial redevelopment efforts, retraining, and system redesign [17].

The underlying motivation for these strategies is largely economic: by creating high switching costs, cloud providers aim to maximize revenue from each customer over time. The harder it is for a client to leave, the more likely they are to continue using the provider's services, and often to expand their usage. This approach ensures long-term customer retention and a steady, often increasing, revenue stream for the provider.

- **Technical**: Occurs when a cloud provider uses proprietary technologies, APIs, or data formats that are not easily portable. Customers relying on these unique features face significant redevelopment when migrating.
  Example: Using cloud-specific database functions that are incompatible with other platforms.

- **Data**: Happens when providers enforce proprietary data formats and structures. Migrating data requires transformation, which increases complexity, cost, and time.
  Impact: Structured data cannot be easily transferred without reformatting.

- **Service**: Arises when consumers integrate deeply with provider-specific services (e.g., analytics, serverless, machine learning). Switching providers requires major application changes.
  Example: AWS Lambda functions aren't easily portable to Azure or Google Cloud Functions.

- **Certification**: Found in regulated industries where providers offer specific compliance standards. Migrating requires recertification, making the process expensive and time-consuming.

- **Contract**: Results from long-term contracts with penalties or early termination fees. These financial constraints discourage switching providers mid-agreement.

- **Economic**: Caused by heavy investment in a provider's ecosystem, training, development, and tooling. Switching involves high costs and operational disruption.

- **Network**: Occurs when network configurations are tightly integrated with a provider's infrastructure. Migration demands major reconfiguration.
  Example: Custom networking (e.g., VPCs, VPNs) needs to be rebuilt on the new platform.

1.3 Simplifying Kubernetes Adoption

The objective of this thesis is to facilitate and promote the adoption of Kubernetes systems by enhancing the tools available for their management and deployment. Despite its widespread use, Kubernetes adoption remains challenging due to its steep learning curve, which often poses a significant barrier for small development teams.

Numerous tools and plug-ins have been developed to simplify various aspects of Kubernetes operations, including provisioning, cluster creation, management, monitoring, and security. This thesis will focus in particular on tools aimed at improving the provisioning and operations management of Kubernetes environments.

1.3.1 The CNCF Landscape

The Cloud Native Computing Foundation (CNCF) aims to make cloud native computing ubiquitous and sustainable by promoting applications and services natively designed for the cloud, with a strong emphasis on open-source technologies and a vendor-neutral ecosystem.

The Cloud Native Landscape [13], maintained by the CNCF, categorizes both open-source and proprietary solutions across the cloud native ecosystem. The landscape is structured into several categories and subcategories, reflecting the different layers and functionalities of cloud native architectures.
The primary categories are [13]:

Provisioning

Provisioning tools lay the foundation for cloud native platforms and applications. This category includes:

- Infrastructure creation, configuration, and management tools;
- Container image scanning and registry solutions;
- Security tools with embedded authorization and authentication mechanisms;
- Secret management and distribution systems.

Runtime

Runtime tools support the execution of containers within a cloud native environment. They provide:

- Persistent storage capabilities;
- Networking solutions (e.g., overlay networks);
- Tools for managing container lifecycles and runtime environments.

Orchestration and Management

With a secure and operational infrastructure in place, orchestration and management tools enable the coordination of containerized services across clusters of physical or virtual machines. This layer includes:

- Scheduling and orchestration frameworks;
- Service discovery and coordination systems;
- API gateways and service proxies;
- Service mesh architectures for inter-service communication.

Application Definition and Development

This layer encompasses tools that support developers in building cloud native applications, including:

- Databases and messaging/streaming systems;
- Image building and application definition tools;
- Continuous Integration and Continuous Delivery (CI/CD) pipelines that enhance code quality and accelerate development.

Observability and Analysis

Observability refers to the system's ability to be understood through its outputs, including metrics such as CPU usage, memory consumption, disk space, latency, and error rates. Analysis provides insight into these data points, enabling anomaly detection and rapid incident resolution. Tools in this category include:

- Logging, monitoring, and tracing frameworks;
- Chaos engineering tools for resilience testing.

Platform

Platforms consolidate various tools into unified environments, simplifying adoption for development teams by offering integrated solutions across the entire application lifecycle. While some organizations build in-house platforms, those with limited resources opt for existing solutions to ensure a sustainable cloud native strategy. Notably, nearly all modern platforms are built upon Kubernetes, reflecting its central role within the cloud native stack.

1.4 The importance of open source

Open source software (OSS) projects are defined by their publicly available source code, which anyone may inspect, modify, and redistribute. This model fosters rapid innovation and collaboration, as multiple organizations and individuals contribute solutions to shared problems. Major studies show that leading companies leverage OSS to improve product quality and reduce costs [1] but also customers value open standards and the interoperability that open source enables.

1.4.1 Mechanics of Open Source development

OSS projects follow distinct mechanics of licensing, governance, contribution, and sustainability that distinguish them from proprietary development.

Licensing

Open source licenses grant users broad rights to reuse and modify code. For example, Kubernetes is licensed under the permissive Apache License 2.0. Under Apache 2.0, users are explicitly permitted to use, modify, distribute, and sublicense the code with minimal restrictions [12]. Notably, Apache 2.0 grants patent rights from contributors to users, easing corporate adoption. Because it does not require derivative works to remain open (unlike GPL-style licenses), it allows companies to integrate Kubernetes into closed-source products without legal concerns. In practice, Apache's permissive terms have encouraged enterprise use: companies can adapt Kubernetes for internal use or commercial offerings without fear of losing proprietary code, and the license's patent clause provides a safety net [12].

Community governance

Every open source project has some form of governance or framework for making decisions and defining roles, even if informal. Well-defined governance helps contributors understand how to participate and make the project sustainable. Many projects use a meritocratic model: contributors earn influence by the value of their work ("do-ocracy"), and decisions are made by those actively contributing code or reviews [5]. Foundations like the CNCF or Apache provide a neutral umbrella to enforce rules that level the playing field. For example, foundations often require projects to hold trademarks and require that no single company can unilaterally change the license. The Linux Foundation describes its role as ensuring "distributed ownership" of code and an "open community governance" model. Kubernetes itself operates under a charter and subproject model overseen by a Steering Committee, where technical leads (SIGs) and maintainers guide development by consensus. In short, community governance in OSS is about defining roles (maintainer, reviewer, contributor, etc.) and decision processes so that contributions are fair and transparent [5]. Contributions to OSS projects are typically made via a public version-control system (such as Git) and issue tracking. Developers fork or branch the code, make changes, and submit pull requests (PRs). Other contributors then review and discuss these changes. Popular platforms like GitHub or GitLab facilitate this workflow, enabling lightweight, asynchronous collaboration.

Project sustainability

Maintaining an OSS project over years requires sustainable support. Funding and resources often come from a mix of sources. Individual developers might contribute in their spare time, but large projects usually rely on corporate backing or community funding. Companies with vested interests often sponsor developers or donate to foundations (as discussed below). Non-profit organizations (e.g. Linux Foundation, CNCF, Mozilla) offer grants or legal infrastructure. Platforms like GitHub Sponsors, Open Collective, and Patreon enable individuals to donate money to projects or maintainers. Corporate sponsors also play a major role: firms often treat contributing to OSS as part of their engineering roadmap or corporate social responsibility. Foundations organize conferences (e.g. KubeCon, Open Source Summit) and training that generate revenue to support infrastructure and staff.

1.4.2 Why Open Source is Crucial for Software Development

Open source methodology has transformed the software world in several broad ways:

Innovation and quality

Open source accelerates innovation through global collaboration. By exposing code to wide scrutiny, projects benefit from many "eyes on the code," leading to rapid bug fixes and feature development [1]. Historically, projects like the Linux kernel, Apache web server, and Python language have jumped ahead of proprietary alternatives by leveraging community invention. In short, open source lowers barriers

to experimentation, enabling companies and individuals to build on each other's work. An academic review notes that open standards and open source share core values of openness and consensus, which lay the groundwork for innovation and fair competition [1].

Cost-effectiveness

By eliminating licensing fees and allowing reuse of existing code, OSS can drastically cut software costs. Large enterprises report saving millions by deploying open source software. Beyond raw dollars, OSS avoids vendor lock-in: because the source code is available, customers are not tied to a single supplier or forced into costly upgrades. The ResearchGate [1] study notes that companies dislike "vendor lock-in" in critical components, and thus prefer OSS solutions that allow them to switch providers or modify code as needed. This flexibility effectively reduces the long-term cost and risk of software adoption. Moreover, open source encourages sharing of infrastructure: containerization (enabled by open tools like Docker and Kubernetes) avoids repeating effort in every project [1].

Standardization and industry adoption

Many open source projects become de facto industry standards precisely by virtue of wide adoption. Open source projects often crystallize around open formats and specifications, sometimes even influencing formal standards bodies. The CNCF and other bodies promote common API versions (e.g. Kubernetes API Stability Guarantees) so that code written today runs on future versions. This standardization reduces fragmentation: developers can write to a single Kubernetes API rather than a dozen proprietary ones [1].

The effect of Google open-sourcing Kubernetes was dramatic: within just a few years, it became the most widely used container orchestration platform and the second-largest open source project after Linux. Moreover, the project's success catalysed the emergence of an extensive ecosystem of complementary tools and frameworks such as Helm, Prometheus, and Istio, further reinforcing Kubernetes' central role within the cloud-native paradigm and demonstrating the critical importance of community-driven governance in sustaining large-scale technological adoption.

Chapter 2

Background

2.1 Containers as a deployment standard

According to Docker, "A container is a standard unit of software that packages up code and all its dependencies so the application runs quickly and reliably from one computing environment to another." [11]

A Docker container image is "a lightweight, standalone, executable package of software that includes everything needed to run an application: code, runtime, system tools, system libraries, and settings." [11]

At runtime, container images become containers on the Docker Engine and run identically across Linux and Windows hosts, isolating software from infrastructure variation.

History of Docker

Docker emerged in March 2013 when dotCloud released the first version of the Docker Engine as an open-source project based on LXC. It quickly gained traction by simplifying Linux container management through a more developer-friendly interface. In early 2014, Docker replaced LXC with its own ‘libcontainer‘ implementation written in Go, improving portability and maintainability. From that point on, Docker advanced rapidly by adopting high-performance union filesystems like OverlayFS and contributing runtimes such as ‘containerd‘ to the broader ecosystem to solidify its role as the de facto standard for building, sharing, and running container images [11].

Docker Architecture and Underlying Mechanisms

Docker uses a client–server architecture. The Docker client communicates with the Docker daemon over a REST API via Unix sockets or TCP. The daemon handles core tasks: building and running containers, managing images, volumes, and networks, and orchestrating object lifecycles. Under the hood, Docker relies on key Linux kernel mechanisms [11]:

- **Namespaces**: create isolated contexts for processes, filesystems, networking,

process IDs, and inter-process communication (IPC), ensuring containers operate independently from the host and each other.

- **Control Groups (cgroups)**: limit and monitor resource usage (CPU, memory, I/O), preventing any container from monopolizing host resources.

- **Union file systems (e.g., overlay2)**: support layered images where each Dockerfile instruction adds a new layer; at runtime, a writable top layer is added for runtime changes. This layering optimizes storage and improves build efficiency.

Therefore, a Docker container essentially functions as an isolated process (or group of processes) running under the host kernel yet confined by namespaces and cgroups.

Images vs. Containers

A Docker image is an immutable, read-only template comprised of stacked layers defined via a Dockerfile. Each layer corresponds to commands such as installing packages or copying files. When instantiated, the image becomes a container with an added writable layer on top, enabling runtime modifications; destroying the container discards this layer by default [11].

Technical Benefits

Because containers share the host OS kernel instead of virtualizing hardware, they are far more lightweight and efficient than virtual machines. Images typically occupy tens of megabytes, and containers can start in milliseconds. Shared kernel usage permits high-density deployment and rapid scaling.

Isolation provided by namespaces and cgroups enhances security, prevents dependency conflicts, and promotes consistent behaviour across environments—from development to production [11].

2.2 Kubernetes: History and Technical Functionality

Origins and the Need for Orchestration

Before Kubernetes, Google had already developed Borg and Omega, internal systems to manage containers at a large scale. Kubernetes was announced in June 2014 and made open source to provide a community-driven container orchestrator inspired by Borg's architecture [16][6]. It filled the gap created by the lack of a standard system for orchestration, scheduling, and cluster management across many machines.

Core Concepts: Orchestration, Scheduling, Cluster Management

A container orchestrator is software that makes multiple machines act like a single powerful compute substrate, coordinating deployment, resource assignment, and cluster health [16]. Kubernetes combines these responsibilities—coordinated operations, assignment of workloads to nodes (scheduling), and integrating machines into a reliable, fault-tolerant cluster into one cohesive control plane [16][6]. This unified approach resolved concerns businesses had about supporting diverse runtime environments.

Key Functional Capabilities

Kubernetes automates deployment, scaling, and lifecycle management of containerized applications through declarative specifications. Service discovery and load balancing are native; they can automatically expose containers via DNS names or IPs and distribute traffic across Pod replicas. Automated rollouts and rollbacks maintain application availability without downtime.

Self-healing ensures failed containers are restarted and unhealthy nodes are replaced. Auto scaling enables horizontal scaling of Pods based on metrics like CPU usage, and cluster auto scaling adjusts node counts as needed [6]. Declarative APIs enable advanced deployment patterns such as blue–green or canary releases, facilitating continuous deployment workflows [16].

Kubernetes abstracts infrastructure differences: by mapping Service definitions to cloud-provider-specific resources like load-balancers, it remains provider-agnostic while still leveraging native features [16].

Impact on Cloud-Native Infrastructure

Just as Docker standardized the packaging of applications, Kubernetes established a uniform platform for running containerized workloads. By abstracting orchestration, scheduling, networking, storage, and scaling, it underpins modern CI/CD pipelines, microservices architectures, and cloud-native deployments with resilient, reproducible, and efficient infrastructure.

2.3 Core Components of Kubernetes

The fundamental architecture of Kubernetes is divided into two principal categories: the Control Plane and the Worker Nodes. These elements collaborate to ensure that the desired application state is consistently realized and upheld [4].

1. Control Plane (Master Node)

The Control Plane, often referred to as the Master Node, serves as the central coordinator of the Kubernetes cluster. It governs the cluster's overall status, orchestrates

scheduling, and oversees the integrity and operation of the system. It includes several key components that determine what should be executed and on which node [6].

Primary Control Plane components:

- **API Server:** Acts as the front-end for the Kubernetes control plane, exposing the Kubernetes API. It handles REST operations, performs validation, and updates the cluster's state accordingly.
- **Scheduler:** Selects the appropriate worker node for deploying pods, based on resource availability and defined constraints.
- **Controller Manager:** Continuously monitors and enforces the desired cluster state, such as ensuring the correct number of pod replicas are running.
- **etcd:** A distributed key-value store that maintains configuration details and the overall state of the cluster, including data about nodes, pods, and services.
- **Cloud Controller Manager** (when using cloud providers): integrates node, routing, and load-balancer state with the hosting environment.

2. Worker Nodes (Node Pool)

Worker Nodes, sometimes referred to as the Node Pool, are tasked with executing the application workloads. Each node includes essential services required to manage and run pods, such as a container runtime, kubelet, and networking proxy [6].

Core components on Worker Nodes:

- **Kubelet:** A daemon that runs on every worker node, ensuring containers in a pod are running as expected. It interfaces with the API server to provide updates on node health and status.
- **Container Runtime:** The engine used to run containers (e.g., Docker engine, containerd, or CRI-O). It handles container lifecycle and execution.
- **Kube-Proxy:** Manages the networking for pods, enabling communication across nodes and ensuring traffic is correctly routed to the appropriate pods.

3. Pods

Pods are the most basic deployable units in Kubernetes. They encapsulate one or more containers and represent a single instance of a running application within the cluster. Pods offer an abstraction over containers, allowing Kubernetes to orchestrate them uniformly [6].

Notable Pod characteristics:

- **Multi-container Pods:** A single pod can house multiple containers that share the same network space and storage. This setup is useful for tightly coupled components, such as a main app and a logging sidecar.

- **Pod IP:** Each pod is assigned a unique local IP address, facilitating inter-container communication using `localhost`.
- **Ephemeral Nature:** Pods are inherently transient. If a pod crashes or is deleted, Kubernetes automatically provisions a replacement to maintain the specified application state.
- **Resource Requests and Limits:** Pods can define their CPU and memory requirements, and Kubernetes schedules them on nodes that can fulfil these resource needs.

2.4 The Role of SDKs in Application Development

SDK (Software Development Kit) help software developers create applications for a specific platform, system, or programming language. Typically, a basic SDK will include a compiler, debugger, and application programming interfaces (APIs) but also documentation, libraries and drivers. Some examples of popular software development kits are the Java development kit (JDK), the Windows 7 SDK, the MacOS X SDK, the Android SDK, and the iOS SDK. As a specific example, the Kubernetes operator SDK can help you develop your own Kubernetes operator. It contains high-level APIs, tools for scaffolding and code generation, and extensions to cover common operator use cases [15]. There are a lot of benefits using SDKs, some of them:

- **Simpler integrations:** SDKs reduce the complexity of integrations by simplifying processes and providing standardized interfaces, such as those used to access device hardware or external services.
- **Abstraction:** SDK often use APIs in order to let your product or service communicate with other products and services without having to know how they're implemented.
- **Documentation and instructions:** SDKs provide built-in support and expertise with written code and support documentation, tutorials, and more, removing the need to search for answers or outsource.
- **Cost savings:** SDKs built within apps offer cost savings because they eliminate the need to rebuild common features, cutting development time and accelerating delivery. The integrations also don't require specialized technical skills, allowing your team to perform in-house integrations.

Cloud SDKs simplify how you interact with a cloud provider's infrastructure services such as storage, compute, and databases. They are often used to:

- Manage resources like storage buckets, databases, or virtual machines.
- Automate provisioning, scaling, and deployment workflows.
- Monitor system performance and usage.

Like many areas in IT, Cloud Computing relies heavily on SDKs, especially those designed to interact with and manage cloud provider resources. Go is the preferred language in this space, thanks to its speed, efficiency, built-in concurrency support, and strong tooling. Some of the most popular include: AWS Go SDK, Google Cloud Client Libraries SDK for Go, Azure Go SDK.

Chapter 3

State of the art

3.1 Kubernetes cluster provisioning tools and methods

Kubernetes clusters can be provisioned in many ways, depending on the environment (cloud, on-prem, edge) and desired automation. Broadly, we can classify approaches into self-managed installers, infrastructure-as-code (IaC) tools, Kubernetes-style cluster management (Cluster API), and managed cloud services. Each has different supported environments, use cases, and trade-offs.

3.1.1 Self-Managed Installers

Kubeadm

The official Kubernetes bootstrap tool. It only initializes a cluster on existing machines (does not provision VMs). Kubeadm is very lightweight and portable: it can run on bare metal, VMs, laptops, or even a Raspberry Pi. It is ideal for quickly getting a "minimum viable" cluster up for testing or development [2].

- Use cases: simple single-cluster installs, custom setups, development environments, or as a building block for higher-level automation.
- Pros: Fast and minimal; no vendor lock-in; runs almost anywhere (cloud or bare-metal); produces best-practice control-plane setup.
- Cons: Requires manual steps: you must pre-provision nodes (with your own scripts or cloud console). It does not manage networking, add-ons, or node lifecycle. Itself doesn't build "production-ready" clusters.

kOps

An open-source tool (by the Kubernetes SIGs) that can create and manage clusters as long as the underlying cloud infrastructure. It acts as "Kubectl for clusters". kOps automates the entire cluster lifecycle: it provisions AWS (and other cloud) resources for control plane and node groups, configures Kubernetes (including HA masters, etc.), and supports upgrades. Officially, kOps supports AWS and GCP

(Google Cloud) out of the box with Azure in alpha and community support for DigitalOcean, OpenStack, and Hetzner [19].

- Use cases: Production-grade Kubernetes on supported cloud providers. Good when you want more control over cluster config than managed services provide.
- Pros: Creates "production-ready" clusters with HA master nodes, VPCs, load balancers, networking and so on. Integrates with cloud features (IAM, ELB, Auto-scaling). Supports cluster upgrades and add-ons. Widely used for AWS Kubernetes.
- Cons: Limited to supported providers (primarily AWS/GCP). Has more moving parts than Kubeadm (config state). Upgrades can be complex. Some manual steps (e.g., kubeconfig management) remain.

Kubespray

A community Ansible-based installer (also a CNCF project) that deploys Kubernetes clusters on many platforms. Kubespray uses Ansible playbooks (under the hood using Kubeadm) to configure a cluster, and supports virtually all major environments: AWS, GCP, Azure, OpenStack, vSphere, Equinix, and generic bare-metal servers. It can create highly available clusters (multiple master nodes), and is highly configurable (choice of CNI plug-ins, OS distributions, etc.) [20].

- Use cases: Deploying clusters across diverse on-prem or cloud platforms, when you need maximum flexibility. Good for heterogeneous environments or when a custom configuration is needed.
- Pros: Multi-cloud and bare-metal support. Built-in support for HA, choice of OS (Ubuntu, CentOS, etc.), and network plug-ins. Active development and CI-tested releases. Uses familiar Ansible, so you can customize easily.
- Cons: Does not provision machines or networks itself. You must supply inventory of hosts (manually or via another tool like Terraform). More complex setup (SSH keys, inventory files). Longer bootstrap time (Ansible runs many tasks).

Rancher RKE/RKE2

Rancher's Kubernetes Engine for on-prem or cloud. They can install Kubernetes on any set of existing nodes (bare-metal or cloud VMs) or provision new nodes via supported providers. RKE2 (also called RKE "Government") is fully conformant and can even be provisioned via Rancher's UI or Terraform (it uses Cluster API under the hood) [24].

- Use cases: Organizations using Rancher for multi-cluster management. Hybrid or on-prem deployments where Rancher is the management plane. Deploying clusters in private data centers or across clouds with a unified interface.

- Pros: Vendor-agnostic (any infrastructure via Rancher node drivers). Integrated lifecycle: Rancher can auto-scale node pools, replace failed nodes, and handle upgrades. RKE2's tight coupling with Cluster API simplifies multi-cloud provisioning.
- Cons: Requires running Rancher server. Tied to Rancher ecosystem. Steep learning curve to set up Rancher.

3.1.2 Infrastructure-as-Code Tools

Terraform

A popular declarative IaC tool. Terraform can't directly install Kubernetes software on a set of nodes, but it can fully provision managed Kubernetes clusters and all their associated infrastructure from cloud providers. For example, Terraform's AWS provider can create VPCs, EC2 instances, and an EKS cluster resource; similarly for Azure AKS or Google Cloud GKE. Terraform has providers for most environments (AWS, Azure, GCP, VMware, OpenStack, etc.), and many community "modules" for cluster resources [23].

- Use cases: Integrate Kubernetes cluster creation into existing IaC workflows or pipelines. Automate cloud resource setup (networks, load balancers, compute) in a reproducible way. Manage cluster state alongside other infrastructure.
- Pros: Broad provider support on multi-cloud or on-prem. Mature ecosystem, state management, Terraform Registry modules. Good for auditing and rollbacks.
- Cons: Not usable on every VM but only into the defined ones on a single vendor at a time, creating vendor lock-in. Cluster lifecycle beyond creation (e.g., upgrades, node scaling) often requires separate steps or external tools. HCL (Terraform's domain-specific language) is less flexible than general-purpose languages, and secret handling requires extra configurations.

Pulumi

An open-source IaC platform where infrastructure is defined in general-purpose languages (TypeScript, Python, Go, etc.). Pulumi can call the same cloud APIs (via providers) as Terraform, and also supports Kubernetes-specific logic. For example, Pulumi has templates for deploying EKS/GKE/AKS clusters on AWS, GCP, or Azure. It also lets you use Helm charts or Kubernetes YAML directly as part of your program [23].

- Use cases: Teams who prefer real programming languages and IDE support for IaC. Complex orchestration or dynamic logic (e.g., loops, conditions) in cluster provisioning. Integrating Kubernetes and cloud resources in one codebase.
- Pros: Use rich language features (loops, packages, modules) for infrastructure logic; built-in secret encryption; strong multi-cloud support. Immediate access to new cloud features via native SDKs.

- Cons: Newer ecosystem than Terraform and requires coding skills. Slightly higher overhead vs. simple declarative configs. Not all organizations accept running code in pipelines.

Kubernetes Cluster API

The Cluster API (CAPI) is a Kubernetes SIG project that brings "Kubernetes-style" declarative management to clusters themselves. With CAPI, you run a management cluster that includes the Cluster API controllers. You then define Cluster custom resources (and related Machine/VM resources) in YAML. The controllers will provision and manage entire clusters on the target infrastructure, just like any other Kubernetes resource [25].

Key points:

- Declarative & Multi-Cloud: You specify cluster size, machine templates, and networking in Kubernetes manifests, and CAPI handles the provisioning of VMs, load balancers, etc. in the target environment. Because CAPI is provider-neutral, it can be extended to any infrastructure (AWS, Azure, vSphere, OpenStack, bare-metal, Equinix Metal, etc.). There are official/integrated providers like CAPA on AWS, CAPZ on Azure, CAPV on vSphere, and community ones (GCP, Docker, Metal3 for bare-metal, etc.).

- Built for Scale: CAPI is production-ready and aimed at large-scale operations. It is upstream in tools like Rancher and Mirantis Kubernetes Engine. It enables managing hundreds or thousands of clusters (e.g., edge sites, multi-cloud clusters) using a unified API.

- How it Works: A bootstrap cluster creates a management cluster, which runs the CAPI controllers. You run clusterctl or YAML manifests to generate a cluster specification. The controllers create Machine resources that spin up VMs (via the infrastructure provider) and join them into a new "workload" cluster. You can later upgrade or scale clusters by updating the manifests.

- Use cases: Multi-cluster management, GitOps-style cluster creation, hybrid/-cloud consistency. Ideal when you need uniform automation across clouds, on-prem, and edge.

- Pros: Declarative, consistent cluster lifecycle; reusable across environments; rich ecosystem of providers; integrates with Kubernetes toolchain.

- Cons: Higher complexity and learning curve. Requires an initial management cluster. Less straightforward for one-off single clusters.

3.1.3 Managed Cloud Kubernetes Services

Cloud providers offer "managed" Kubernetes with control planes handled for you. These simplify cluster provisioning but have trade-offs:

Amazon EKS

AWS's managed Kubernetes. Amazon runs the control plane across multiple availability zones (for HA), and you pay a nominal hourly fee (around $0.10/hr per cluster). According to CNCF surveys, EKS is the most widely used managed K8s service [22].

- Key features: Control plane is fully managed (auto-replaced on failure), integrated with AWS IAM, ALB, and other services. Supports serverless and outposts.
- Use cases: AWS-centric environments needing managed K8s. Existing AWS automation (CloudFormation, Terraform) can provision EKS easily.
- Pros: High reliability (99.95% SLA); deep AWS ecosystem integration; RBAC with IAM.
- Cons: Fewer out-of-the-box conveniences than GKE (more manual upgrades). There is an extra per-cluster charge. Some setup (VPC networking, node CNI) requires manual steps.

Google GKE

Google's managed Kubernetes (inherited from Borg experience). GKE is known for its advanced features and automation [22].

- Key features: Auto-managed control plane and node upgrades; release channels (Rapid, Regular, Stable) for version control; GKE Autopilot (hands-off mode) and GKE Edge options.
- Use cases: Multi-cloud or Google-centric use; when you want the smoothest management experience. Good for easily staying on the latest K8s versions.
- Pros: Automatic upgrades for control plane and nodes; quick access to new K8s versions; free zonal control plane. Excellent dashboard and Cloud Operations integration.
- Cons: Google Cloud dependency; regional clusters cost around $0.10/hr to match EKS SLA. No dedicated government cloud.

Azure AKS

Microsoft's managed Kubernetes. AKS integrates tightly with the Azure platform (AD, Azure Monitor, Policies). It historically offered fast availability of new K8s versions. The control plane is offered free, and you pay only for nodes [22].

- Key features: Integrated Azure AD and Azure Policy; automatic node repair; Azure Monitor/Log Analytics integration; Azure Dev Spaces and DevOps tools.

- Use cases: Enterprises invested in Azure/Microsoft stack. For example, Azure AD or Active Directory integration for multi-tenant identity.
- Pros: Free control plane makes it cost-effective; first to roll out new Kubernetes versions; good Windows container support.
- Cons: Historically, cluster upgrades were semi-manual, though Microsoft is automating more. Some features, such as network policies, must be enabled at creation. Limited to the Azure environment.

3.2 Architecture and Core Functions of kOps

I selected kOps over other available projects primarily because it is open source, scalable, and offers a high level of completeness and complexity. Its approach, which relies on cloud provider APIs and Kubernetes APIs, is more robust and maintainable compared to solutions based on Ansible scripts or proprietary tools. One of the most significant advantages of kOps is that it allows users to manage not only the creation and deployment of production-grade Kubernetes clusters but also their ongoing operations. This distinguishes it from tools that are limited to setting up minimal viable products.

3.2.1 How kOps works

kOps is a command-line tool that enables creation, management, upgrading, and maintenance of production-grade Kubernetes clusters by provisioning the required cloud infrastructure in an automated and declarative fashion [19]. The following paragraphs provide a detailed examination of the subsystems involved. Each section will describe the purpose, internal mechanisms, and interactions of the components involved in the lifecycle of a Kubernetes cluster managed by kOps.

Command-Line Interface (CLI)

The entire operation of kOps is driven by its CLI, implemented using the Cobra library in Go. Each command, such as `kops create cluster`, is defined in the `/cmd/kops` directory. The primary entry point for cluster creation is the function `RunCreateCluster()` located in `create_cluster.go`. This function orchestrates the workflow to build the cluster according to user inputs [19].

State Store and Storage Abstraction

kOps stores cluster configuration state in a remote backend, often referred to as the state store. Access to this storage is abstracted through a clientset interface, obtained via a factory object. Common functions include `Get()`, `Create()`, `Update()`, and `List()` for managing cluster resources. Importantly, kOps does not store information in the state store that can be inferred directly from the actual cloud state. Usually, the state store is stored in a remote S3 bucket [19].

API Layer

At the core of kOps lies its API, defined with Go structs and coupled with Kubernetes API machinery for version control. The principal structure is `api.Cluster{}`, whose configuration is held in `cluster.Spec`. This specification defines cluster-level attributes such as networking, Kubernetes version, and add-ons. Cluster nodes and roles are defined via Instance Groups, which are represented as a slice of pointers to `kops.InstanceGroup` objects. Each group specifies parameters such as machine type, disk volume, and instance count [19].

Cloudup Engine

Once a valid cluster specification and instance groups are defined, kOps constructs an `ApplyClusterCmd` object. This object encapsulates the cluster, clientset, selected cloud provider, output directory, instance group definitions, and run settings such as dry-run flags or task timeouts. Invoking its `Run()` method launches the core orchestration sequence, which includes the following stages [19]:

- Input sanitisation and validation to ensure consistency and correctness.
- Construction of a provider-specific cloud object based on the cluster's cloud provider setting.
- Model building phase, which translates the high-level cluster specification into discrete tasks. Each task represents an atomic cloud API operation (for example, creating a VPC or launching an instance).
- Task execution phase, in which each task is inspected (via a `Find()` method) and then rendered (via `Render()`) to call the actual cloud API.

Node Bootstrapping (Nodeup)

Nodeup is a separate binary executed during node initialization. It is invoked through cloud-init scripts when new instances launch. Nodeup installs and configures necessary dependencies on each node, ensuring that it joins the Kubernetes control plane and becomes fully operational as part of the cluster [19].

Operator Reconciliation (kops-controller)

In addition to provisioning logic, kOps includes a runtime controller that runs as a DaemonSet on master node(s). This controller performs reconciliation, such as applying needed labels on Kubernetes Node objects. For example, it can map a Node to its Instance Group by querying provider APIs and applying metadata tags accordingly [19].

3.2.2 How to use it

kOps provides a suite of command-line operations that support the complete lifecycle management of a Kubernetes cluster. These operations include registering a cluster,

applying or updating infrastructure, rolling updates, querying state, and cleanup [18].

Cluster Creation and Registration

The command `kops create` registers a cluster. There are two possible approaches. One involves using a cluster specification YAML file, for example:

```
kops create -f <cluster_spec_file>
```

Once the cluster has been registered in the state store, the following command applies the specification and creates the necessary cloud infrastructure:

```
kops update cluster --yes
```

Alternatively, the command

```
kops create cluster <clustername>

Example:
kops create cluster --cloud=elemento --name=test.k8s
--state=s3://test-kops --kubernetes-version=1.32.4
--zones=europe --node-count=3 --node-volume-size=64
--control-plane-volume-size=64 --network-cidr 10.0.0.0/24
```

constructs the cloud specification directly, using flags such as zones, node count, image, and topology. In most cases, the cluster specification is edited manually via `kops edit cluster`.

Applying or Updating Infrastructure

The command `kops update cluster <clustername>` reconciles the actual cloud infrastructure to match the desired cluster specification. Before finalizing changes, it is advisable to preview the planned operations using:

```
kops update cluster --name <clustername>
```

Once the preview matches expectations, applying changes is performed by including the `-yes` flag:

```
kops update cluster --name <clustername> --yes
```

Cluster Rollout

To deploy changes that require restarting or replacing nodes,

```
kops rolling-update cluster <clustername>
```

is used. It is again possible to preview the actions before execution, and then to proceed securely using the `-yes` flag. Both when applying or updating the configuration, kOps executes all necessary modifications in sequence so that the actual cluster state aligns as closely as possible with the desired specification, following the same declarative and reconciliation-driven approach used by Kubernetes.

Querying Registered Clusters

The command `kops get clusters` lists all clusters currently registered in the state store. This command is useful for auditing and overview purposes.

Deleting a Cluster

To delete all cloud resources associated with a cluster—including DNS records, load balancers, volumes, instances, and network elements—and to remove the cluster from the registry, use:

```
kops delete cluster --name <clustername>
```

As with other operations, a preview is recommended, followed by confirmation using the `-yes` flag:

```
kops delete cluster --name <clustername> --yes
```

Other Utility Commands

`kops toolbox template` generates reusable cluster specification templates using Go templating, which is particularly useful for managing multiple clusters with consistent configurations. The command `kops version` reports the version of the kOps binary currently in use [18].

Chapter 4

Methodology

4.1 Mastering Large Project Architecture: Insights from kOps

To effectively navigate and comprehend a large, pre-existing codebase, a structured approach is essential since it is much more difficult than starting a project from scratch. The initial and fundamental step involves successfully getting the code to run in a local development environment. This action is critical because it isolates potential issues related to the local setup from those inherent in the codebase itself, providing a stable foundation for subsequent work.

Once the environment is functional, a deep understanding of the product's functionality is essential. This can be achieved through various methods, such as studying documentation and trying to execute the code itself. Open-source projects, like the one discussed in this thesis, are often thoroughly documented from a usage perspective but tend to lack equally detailed documentation on their internal structure and functioning, making it hard for new developers to jump in.

When documentation is scarce, one of the best ways to understand the flow and logic of an application is by examining its tests. Tests often serve as an implicit form of documentation, and when well written, they can reveal everything needed to know about the codebase's underlying logic. Finally, reading the code itself can reveal the specific logic of a function or API call. However, since it is more time-consuming, it is usually considered a last resort. Ideally, well-written code should 'explain itself' [21].

4.1.1 The importance of the debugger and error codes

When fixing errors or implementing new functionalities, programmers often encounter multiple bugs and logical errors. Identifying the root cause can be complex, and following a systematic approach helps speed up the process. Typically, the debugging process consists of the following steps:

- **Error Identification**: Developers, testers, and end users report bugs encountered during software testing or usage. Developers then locate the exact line of code or module responsible for the bug. This process can be tedious and

time-consuming; this step can be streamlined using AI tools or detailed error outputs in order to reduce the waste.

- **Error Analysis**: Programmers analyse the bug by recording all program state changes and data values. They also prioritize bug fixes based on the severity of impact on software functionality.

- **Correction and Validation**: Developers fix the bug and perform tests to ensure that the software continues to function as expected. They may also write new tests to prevent the bug from recurring in the future.

Errors can take various forms, depending on their origin: syntax errors, semantic errors, logical errors, or runtime errors caused by edge cases not anticipated during the programming phase. One widely used approach to solving bugs is incremental development, where programs are built in manageable sections that can be frequently tested. This allows programmers to detect bugs early and address them one at a time, rather than confronting multiple errors after completing large portions of code. Similarly, the divide-and-conquer method encourages breaking the code into smaller, focused areas, making it easier to isolate and resolve specific issues.

Another traditional technique is backtracking, particularly effective for smaller programs. Developers work backward from the point where an error manifests to identify its origin in the code. While this method becomes more challenging as the codebase grows, it remains a valuable tool for tracing the root cause of problems. In modern contexts, remote debugging is often necessary, allowing developers to analyse applications running in separate environments, such as servers or cloud platforms. Alongside this, logging plays a key role: programs frequently record internal data, execution times, and system states in log files, which developers can examine to track down errors and understand program behaviour.

Practical techniques also include strategic use of print statements to observe variable values at critical points, systematic testing of edge cases to uncover hidden bugs, and careful use of version control systems like Git to track changes and identify when and where a bug was introduced [3].

4.1.2 The approach adopted in this specific case

The debugger used in this project is Delve [7], a tool built specifically for Go. Since the kOps codebase is very large and complex, I first experimented with its basic functionalities before turning to the documentation to gain a deeper understanding of its logic. However, because the implementation details were not extensively documented, I adopted a black-box approach: I provided specific inputs, analysed the corresponding outputs, and traced variable values using the debugger to uncover the chain of function calls.

As my task was to introduce support for a new cloud provider into kOps, I began by studying the existing implementations already integrated into the project. I selected the Hetzner Cloud implementation as the reference model, both for its relative simplicity and for the balance it offered between ease of understanding and the range of features it supported, which made it well aligned with the requirements

of Elemento. The most useful debugging method I relied on throughout the implementation was the extensive use of logs. Both the logs already present in kOps and those I added in my implementation proved invaluable in tracing the root causes of bugs.

4.2 Systems integration

The system is made up of three main components that work closely together: the kOps library, the Elemento library, and the AtomOS instance.

The kOps library is the entry point of the process. It receives the commands required to create the cluster configuration and to trigger the provisioning phase. It is being extended to support Elemento as a new cloud provider, ensuring that it can integrate seamlessly with the rest of the infrastructure. The Elemento Go library acts as an adapter between kOps and the Elemento platform. Its role is to translate function calls from kOps into API calls that can be understood by Elemento systems. This translation layer allows kOps to remain agnostic of provider-specific APIs and instead rely on standardized functions, making it easier to support multiple cloud providers with minimal changes. Finally, the AtomOS instance represents the underlying execution environment. It is Elemento's Linux-based KVM hypervisor, responsible for actually provisioning the requested resources, such as storage, virtual machines, and networking, based on the API calls it receives.

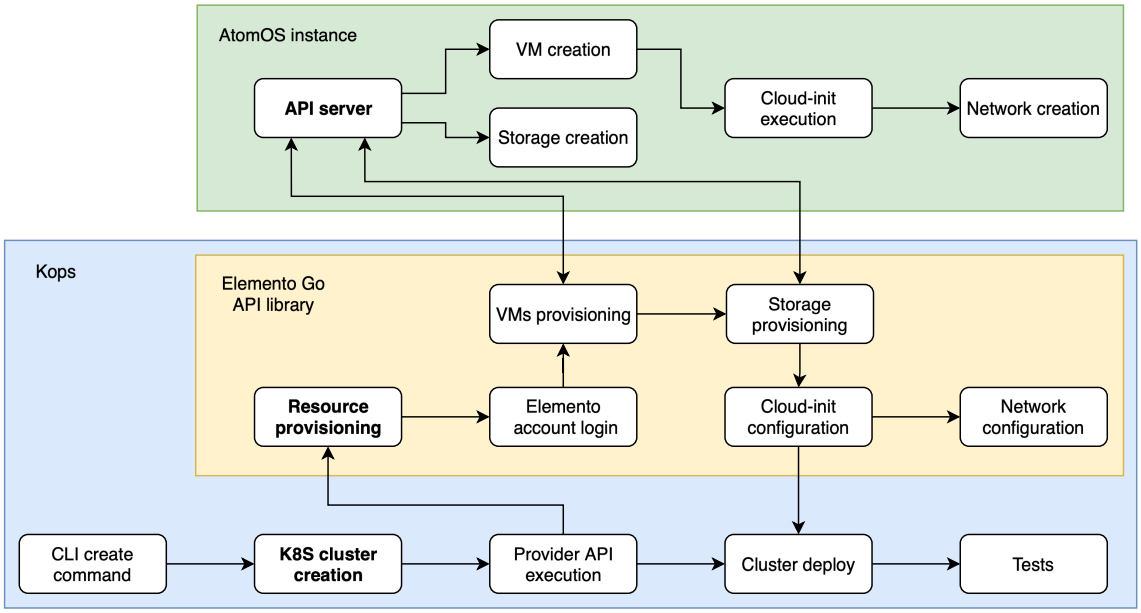

Figure 4.1: System integration with kOps

4.2.1 Handling of asynchronous operations

In the kOps codebase, the *fi* package provides the abstraction layer through which cloud infrastructure is defined and reconciled, and at the heart of this system are the so-called tasks. An *fi* task is a Go struct that models a specific resource, such as a server group, a subnet, or an IAM role, and encapsulates both the configuration values that describe the resource and the behaviour required to manage its lifecycle. Rather than invoking cloud provider APIs directly in an imperative manner, kOps expresses desired state as a graph of interdependent tasks, with each task responsible for representing one unit of infrastructure. The task graph is then traversed and executed by the *fi* engine, which applies the necessary changes in a deterministic order, ensuring that dependencies are respected and the resulting cluster state matches the declared specification in a typical k8s logic fashion. The execution relies on around 40 fi tasks that execute all the necessary actions needed to provision the desired cluster configuration.

This approach makes tasks both declarative and idempotent: they provide a high-level description of resources, while the *fi* runtime determines whether they need to be created, updated, or left unchanged. By relying on generated boilerplate code from the fitask tool, tasks implement a common set of interfaces such as name resolution, lifecycle management, and string formatting that allows them to integrate seamlessly into the provisioning engine and be handled uniformly across different cloud providers' Go SDKs. Tasks that can be executed in parallel, such as provisioning resources for different nodes, are often carried out in this way, thereby further reducing overall provisioning time.

4.3 How the Elemento support on kOps could be very useful

By integrating Elemento into the list of cloud providers supported by kOps, I'm not merely adding a single provider but enabling access to an entire network of small cloud providers that stand to benefit significantly from this integration. The system leverages a unified architecture in which all Elemento hypervisors communicate with each other and function as a single abstract cloud system. This design allows for the provisioning of nodes across multiple cloud environments. Future enhancements could further enable the system to autonomously heal itself by allowing the kOps controller APIs to manage AtomOS provisioning capabilities across multiple instances hosted on different cloud providers.

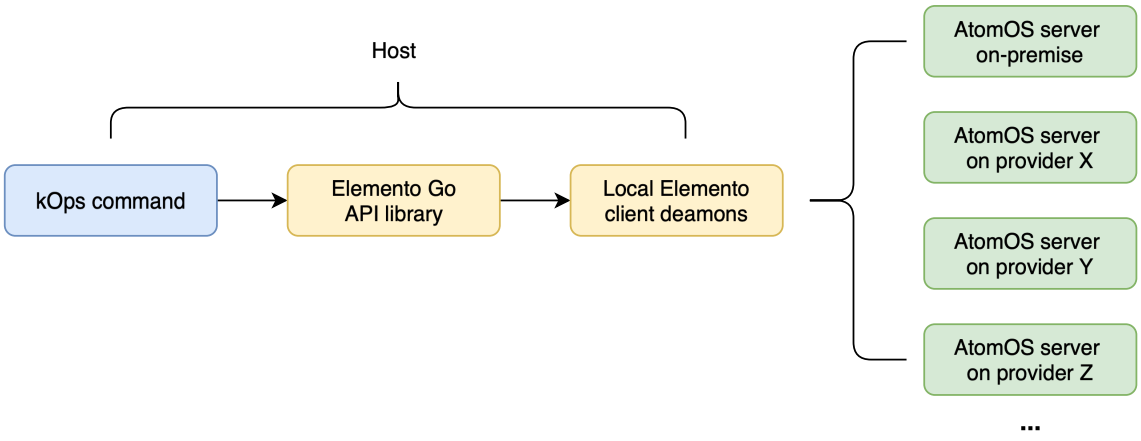

Figure 4.2: Elemento contribution to an unbounded Cloud environment

Chapter 5

Work development

5.1 The integration workflow for adding Cloud Providers in kOps

Unfortunately, there is no single standardized procedure for integrating a new cloud provider into the kOps project. While the project defines a set of common interfaces and shared logic, each provider must still be integrated in a slightly different manner. This results in greater complexity, both in terms of understanding the overall system and in ensuring its long-term maintainability. A key challenge arises from the fact that cloud providers often implement equivalent resources in significantly different ways, owing to proprietary technologies, patents, and distinct architectural choices.

The kOps code has been forked by Elemento, where the specific implementation for its platform is maintained. This fork is not fully compliant with the guidelines of the main kOps project, largely due to the architectural differences between large-scale cloud providers and Elemento's infrastructure. To bridge this gap, another critical component is the Elemento Go plug-in, which serves as a translation layer: it maps the standardized kOps interfaces onto Elemento's proprietary abstractions, thereby enabling functional interoperability despite the underlying heterogeneity.

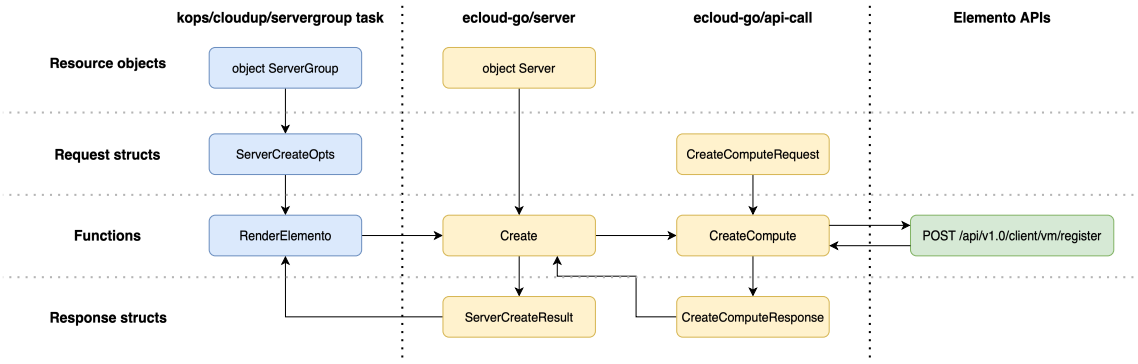

Figure 5.1: Elemento create VM flow

In Figure 5.1, we can see a schema illustrating the interaction between the different system components concerning the Virtual Machine (or Server) object. The execution begins on the left, where the `ServerGroup` task invokes the func-

tion `RenderElemento` within a task (as explained in Section 4.2.1). This function takes as input the `ServerGroup` object and converts the relevant fields into a `ServerCreateOpts` structure, making them compatible with the Elemento Go plug-in (highlighted in yellow). The Elemento Go plug-in then receives the call to the `Create` function and maps the values from the `ServerCreateOpts` fields into the `Server` object. A request is subsequently issued to the Elemento Go API gateway, where the `CreateCompute` function is called. At this stage, the request payload is constructed according to the documented struct definitions and forwarded to the Elemento client API, which authenticates the request and transmits it to the AtomOS server. Once a response is returned to the API gateway, it is validated and marshalled into a `CreateComputeResponse`, which is then passed back to the `Create` function. This function elaborates the response into a `ServerCreateResult` (compatible with kOps) and returns it to the `RenderElemento` function. All these steps and conversions are supported by Go's strict type and struct checks, which significantly reduce the likelihood of errors.

5.2 How I iterated the testing and development

After analysing all the required resources and the overall architecture, I proceeded with the implementation of one resource area at a time, beginning with virtual machines, then moving on to storage, networking, and finally SSH keys, from kOps through to the API gateway. Once each component was implemented, I validated the functionality of the endpoints by executing tests starting from the Elemento Go API gateway toward the AtomOS instance, and then gradually integrating the subsequent layers of the system until i reached kOps. This incremental approach, called Test Driven Development, facilitated more effective debugging and allowed errors to be isolated and resolved step by step.

Test-Driven Development (TDD) is a software development methodology that applies an iterative and incremental approach to programming, driven by a tight feedback loop of writing tests before the corresponding production code. The process, often summarized as Red-Green-Refactor, begins with the developer writing a failing automated test case for a new, small piece of functionality (Red phase). This failure validates the test's purpose and confirms the feature's absence. Next, the developer writes the minimal amount of code necessary to make the test pass (Green phase), ensuring the new code directly addresses the requirement without introducing extraneous functionality. Finally, the developer refactors the code to improve its design and maintainability while ensuring all tests continue to pass (Refactor phase). This cycle, supported by a comprehensive test suite, results in a robust and reliable codebase with high test coverage, serving both as living documentation and as a safety net for future changes. By prioritizing automated testing, TDD shifts the focus from writing large blocks of code followed by testing, to a continuous cycle of verification and refinement, reducing debugging time and promoting cleaner, more modular designs.

5.3 Interesting workflow management of async actions by Hetzner Cloud

While designing the Elemento Go plug-in, I took inspiration from the architecture of the Hetzner Cloud Go library, where I observed a particularly interesting logic for managing asynchronous operations. The Hetzner Cloud Go library provides an elegant mechanism for handling asynchronous operations through its Action system, which functions as a high-level abstraction for long-running cloud tasks that cannot complete immediately. When operations such as server creation, configuration changes, or network management are initiated, the API returns an Action object encapsulating the entire lifecycle of the operation. Each Action includes a unique identifier, current status (e.g., running, success, or error), progress percentage, timing information, and references to affected resources. This design ensures complete visibility into the state of the operation and facilitates robust tracking of cloud resource modifications.

A central feature of this system is the library's polling mechanism, implemented via the `WaitForFunc` method. This method adopts an efficient strategy for monitoring action completion: instead of issuing individual API calls for each action, it batches requests for up to 25 actions at once, polling at configurable intervals (with a default of 500 milliseconds). It further supports real-time updates through callback functions, reducing API overhead while preserving responsiveness. Additionally, the method integrates with Go's context system, enabling timeout management and graceful cancellation of long-running operations. What distinguishes this system is its flexibility and comprehensive error-handling capabilities. The `WaitForFunc` method allows developers to define custom update handlers capable of processing progress information, logging status changes, or embedding domain-specific business logic into the lifecycle of an operation. In cases of failure, the library enriches error reporting by returning `ActionError` objects containing both machine-readable error codes and human-readable descriptions, together with references to the associated Action. This design not only simplifies debugging but also enhances resilience by providing insights into the causes of failed operations.

5.4 Creating Cluster Manifests and Configurations

The first step to create a cluster is set the environment variables required by kOps. In the Elemento configuration, the variables are:

```
export S3_REGION=eu-south-1
export S3_ENDPOINT=https://s3.eu-south-1.wasabisys.com
export S3_ACCESS_KEY_ID=...
export S3_SECRET_ACCESS_KEY=...
export PROVIDER=elemento
```

These variables are required to authenticate against the S3 bucket hosted by Wasabi, which is used as the state store where kOps maintains its configuration and versioning data.

Now we need to start the Elemento Electros app that is needed for authentication and gateway into the Elemento ecosystem. Then we can run the create cluster command to kOps CLI.

```
kops create cluster --cloud=elemento --name=test.k8s
--state=s3://test-kops --kubernetes-version=1.32.4
--zones=europe --node-count=2 --node-volume-size=64
--control-plane-volume-size=64 --network-cidr 192.168.100.0/24
```

This command generates the configuration for both the cluster control plane and the worker nodes and stores it in the S3 bucket. It generates also manifests for the internal add-ons such as coredns, kops-controller, kubelet-api and cilium CNI. Although kOps supports a wide range of customizations, in this thesis the specification has been limited to a few key elements: the cluster name, the state store bucket, the Kubernetes version, the availability zone, the number of worker nodes, the size of their volumes, the size of the control plane node (which defaults to a single node), and the network address with its netmask. The result of the configuration leads, among others, to the following manifests.
Configuration of the control plane generated by kOps:

```
apiVersion: kops.k8s.io/v1alpha2
kind: InstanceGroup
metadata:
  creationTimestamp: "2025-09-15T08:32:29Z"
  name: control-plane-europe
spec:
  image: ubuntu-24-04
  machineType: argon
  maxSize: 1
  minSize: 1
  role: Master
  rootVolumeSize: 64
  subnets:
  - europe
```

Configuration of the nodes generated by kOps:

```
apiVersion: kops.k8s.io/v1alpha2
kind: InstanceGroup
metadata:
  creationTimestamp: "2025-09-15T08:32:30Z"
  name: nodes-europe
spec:
  image: ubuntu-24-04
  machineType: neon
  maxSize: 2
  minSize: 2
  role: Node
```

```
rootVolumeSize: 64
subnets:
- europe
```

The parameter `machineType` defines the virtual machine flavour, which similarly to other cloud providers, represents a standardized specification of resources.
In Elemento, five flavours are available:

- **Helium** → 1 vCPU, 0.5 GB RAM
- **Neon** → 2 vCPU, 2 GB RAM
- **Argon2** → 4 vCPU, 4 GB RAM
- **Argon** → 6 vCPU, 4 GB RAM
- **Kripton** → 8 vCPU, 8 GB RAM

For the control plane, the **Argon** flavour was selected as the default, since it must handle significant load while being particularly sensitive to latency and network performance. Conversely, the worker nodes were configured with the **Neon** flavour, which allows for finer-grained scalability in response to workload variations. Scaling strategies will be further discussed in the final chapter.

5.5 Installation and startup of the cluster

Once i completed the implementation and fixed all the bugs regarding the CloudUp phase that corresponds to `kops create` command i started debugging the NodeUp side by spawning clusters with `kops update` and seeing the log output of the kOps script. Command executed into kOps CLI:

```
kops update cluster test.k8s --yes --admin --state=s3://test-kops
```

This command provisions the resources required to translate the previously defined specifications into actual infrastructure for running the cluster. The process includes the creation of all necessary virtual machines, storage volumes, and networking components. It is executed autonomously by kOps through the Elemento Go plug-in that I developed. The result can be seen from the Elemento Electros dashboard on Fig. 5.2 and Fig. 5.4.

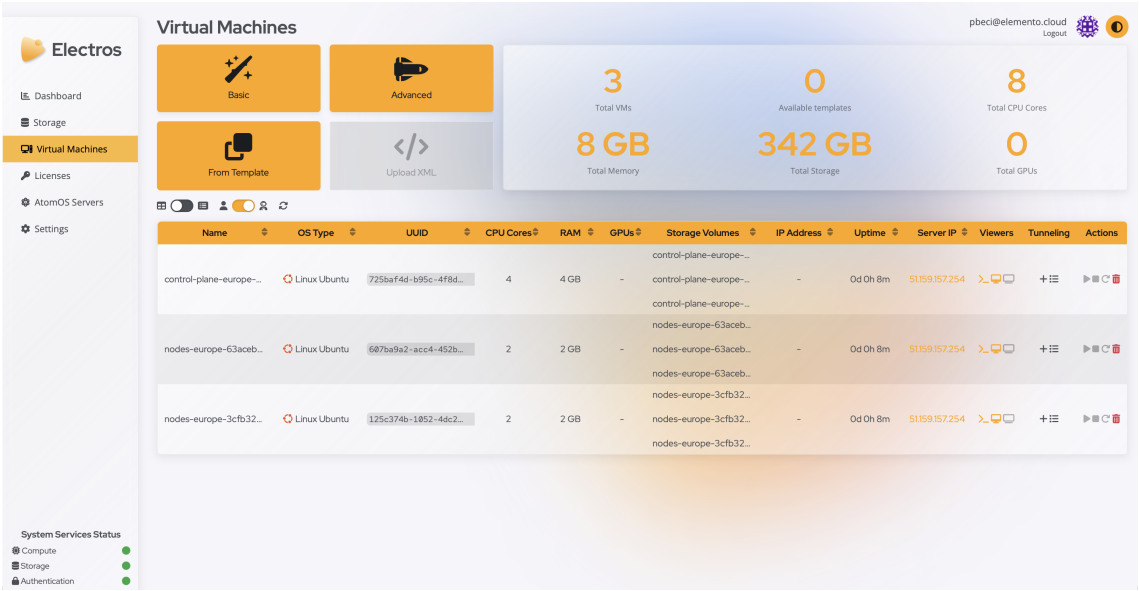

Figure 5.2: Elemento Electros VM list

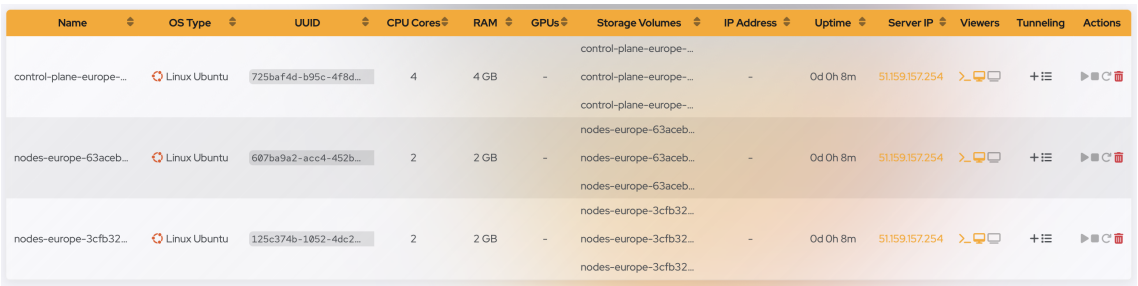

Figure 5.3: Elemento Electros VM list detail

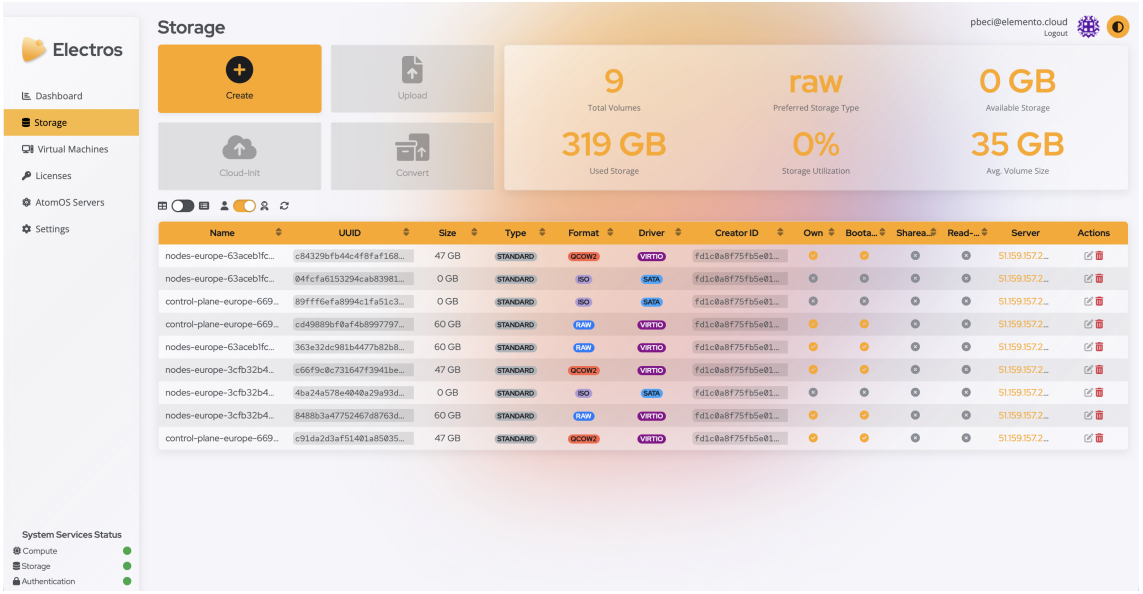

Figure 5.4: Elemento Electros storage list

Figure 5.5: Elemento Electros storage list detail

Each virtual machine is provisioned with three volumes: one containing the operating system boot image, one containing the cloud-init manifest, and one for general storage defined through the kOps CLI.

Cloud-init is the industry-standard, cross-distribution tool for automating the initial configuration of virtual machines (VMs) in cloud environments. Its primary purpose is to transform a generic, pre-built VM image into a customized instance that meets a user's specific requirements during the very first boot. By automating these early-stage configuration tasks, cloud-init eliminates the need for manual intervention, ensuring both consistency and efficiency in large-scale deployments.

Since all the kOps virtual machines must be initialized and execute a startup script, cloud-init is the ideal tool for the task. Below is the configuration used in Elemento VMs to set up SSH keys, users, networking, and the kOps bootstrap script to be executed at startup.

```
#cloud-config
# 1. Create a user, set password, and add to the 'sudo' group
users:
  - name: root
    sudo: ALL=(ALL) NOPASSWD:ALL
    shell: /bin/bash
    lock_passwd: false
    ssh_authorized_keys:
      - ssh-rsa ...
      - ssh-ed25519 ...
ssh_pwauth: true
chpasswd:
  list: |
    root:password
  expire: false

# 2. Configure the network settings
hostname: myhost
network:
  version: 2
  ethernets:
    ens3:
      dhcp4: true`

# 3. Define the bash script content
write_files:
  - path: /home/root/kopsscript.sh
    permissions: "0755"
    owner: root:root
    content: |
      <kOps UserData script>

# 4. Run the bash script
runcmd:
  - [ bash, /home/root/kopsscript.sh ]`
```

Once provisioning is completed, the next step is the actual creation of the cluster. Initially, I encountered several issues because of the cloud-init UserData script executed on each virtual machine. This script includes a hash-based authentication mechanism between itself and the NodeUp binary downloaded from the official kOps release. Since the official release does not include the Elemento modifications, which exist only in a dedicated fork, the authentication process failed. To resolve this, I created a release within the Elemento fork and removed the hash verification from the cloud-init bash script. The code downloaded from this custom release is a binary build of kOps NodeUp, which is responsible for bootstrapping the cluster based on the configuration retrieved from the S3 bucket created earlier. Because the entire startup logic is embedded within the NodeUp binary, no additional external

dependencies or commands are required. This self-contained design represents a significant advantage over other cluster creation systems.

During debugging, I encountered persistent networking issues that prevented the correct initialization of cluster components. To simplify the testing process, I implemented a lightweight demonstration. Instead of relying on the complex kOps NodeUp script to bootstrap the cluster, I developed an Ansible playbook that executes from the AtomOS host machine via SSH commands triggered by kOps.

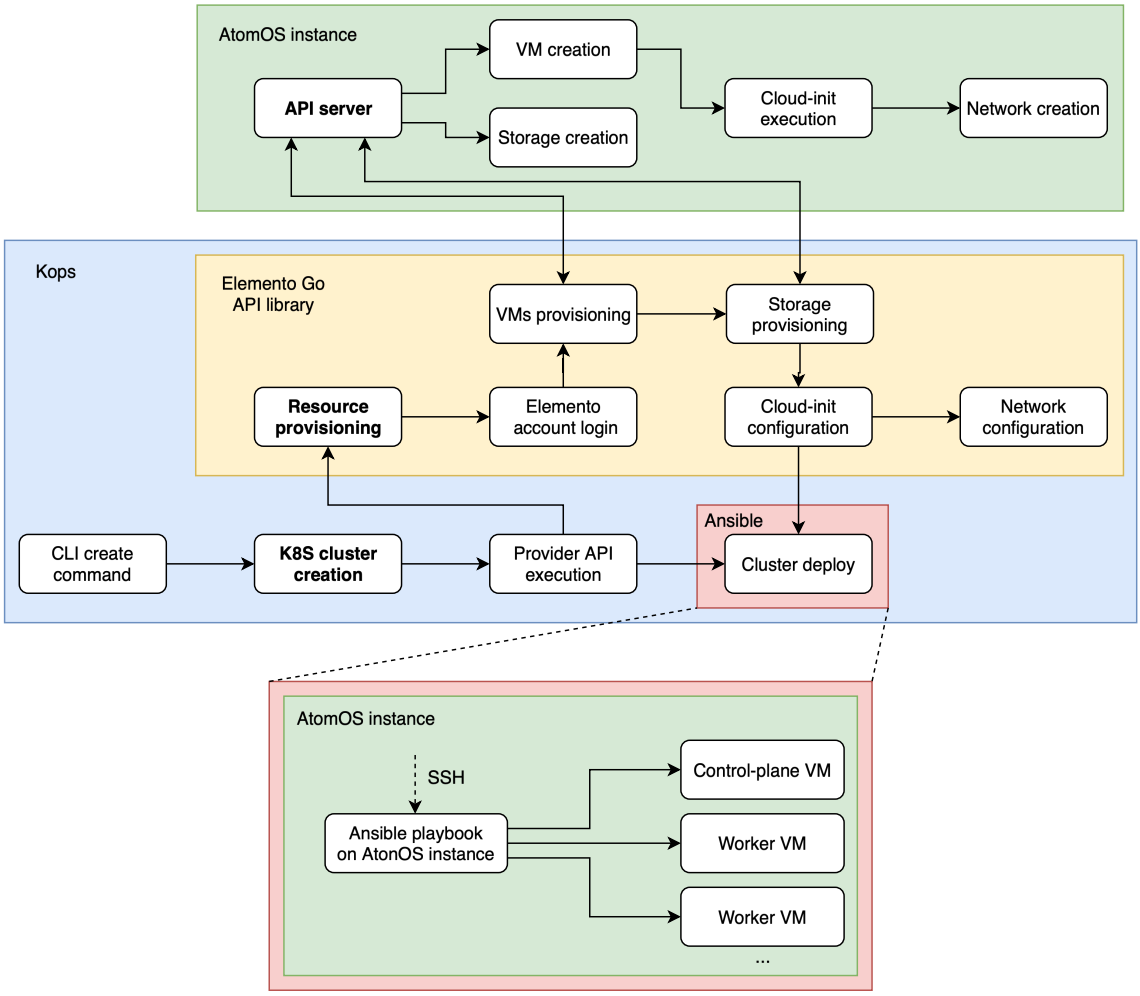

Figure 5.6: Ansible role in the systems architecture

This playbook configures the cluster on all nodes listed in the inventory file and returns the kubeconfig to the kOps client. To connect to the cluster through `kubectl`, SSH access to the AtomOS host must be enabled, as it is required to establish a tunnel on port 6443 in order to reach the Kubernetes API of the cluster.

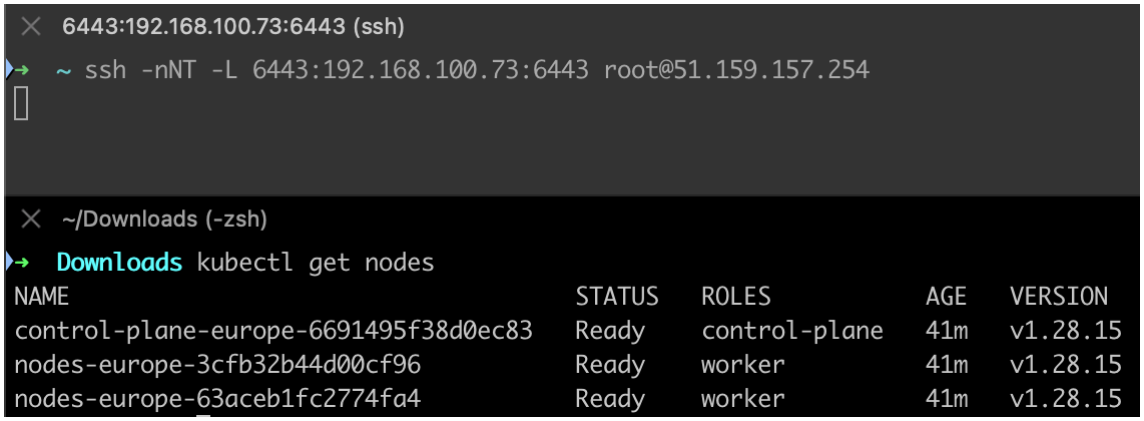

Figure 5.7: Cluster nodes list from external host

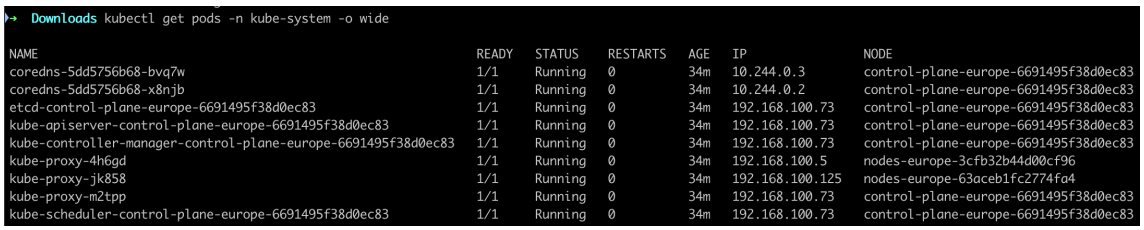

Figure 5.8: Cluster pods list from external host

Where the command:
`ssh -nNT -L 6443:192.168.100.73:6443 root@51.159.157.254`
establishes a secure tunnel that forwards any connection made to port 6443 on the local machine through an SSH connection to the remote server at `51.159.157.254`. From there, the traffic is transparently redirected to the private address `192.168.100.73` on port 6443. This mechanism is commonly employed to access services running within a private network that are not directly reachable from the public internet. In this context, the public-facing AtomOS server at `51.159.157.254` functions as a secure jump host or gateway, enabling controlled access to the Kubernetes API endpoint.

Chapter 6

Work evaluation

6.1 Testing the Kubernetes provisioning through kOps

In this section, I will evaluate the performance of the kOps library in comparison with other cloud providers offering the same or similar services. The evaluation focuses on three key aspects: provisioning speed, ease of setup, and cost. The data used for this comparison is drawn from an article which, although not very recent, provides exactly the information required for this analysis [14].

6.1.1 Provisioning speed comparison

All the provisioning benchmarks presented in the article [14] were carried out using Terraform. For this thesis, I compare those results with the provisioning process performed through kOps integrated with AtomOS. It is important to note, however, that Elemento does not currently support a high-availability (HA) control plane, which may slightly affect the accuracy of the comparison.

The measurements in Fig.6.1 highlight the efficient provisioning speed achieved by the Elemento-kOps system. Most of the total time is consumed by the installation of the Kubernetes cluster on the virtual machines and the subsequent reboot steps required to apply the installed packages. Further research could identify opportunities to streamline or optimize these phases, thereby reducing the overall provisioning time.

Provisioning speed across various cluster sizes

I conducted several tests to measure the time required to provision clusters of varying sizes, with the goal of identifying potential scalability trends. The experiments were performed on clusters ranging from 3 to 10 nodes, as hardware limitations prevented testing larger configurations. The test was conducted on an AtomOS host machine equipped with 16 CPU cores, 64 GB of RAM, and 1 TB of storage capacity.

The results shown in Fig.6.2 indicate that the increase in provisioning time remains minimal as the number of nodes grows. This behaviour is largely due to the parallel execution model adopted by both kOps and Ansible, which allows the

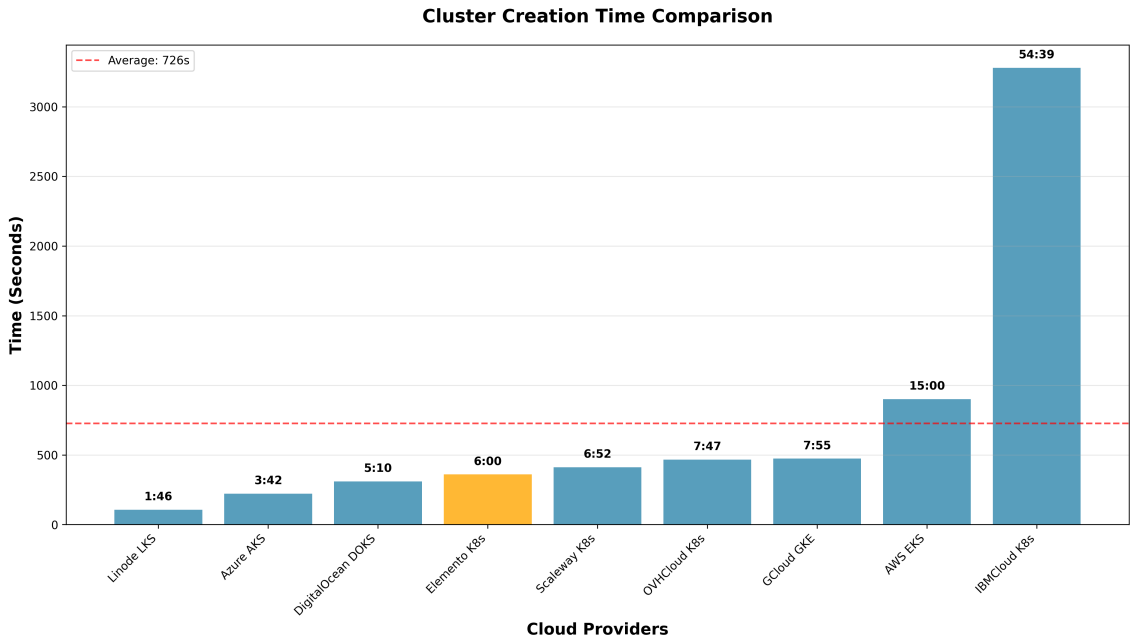

Figure 6.1: Cluster creation time comparison based on 2023 data - source [14]

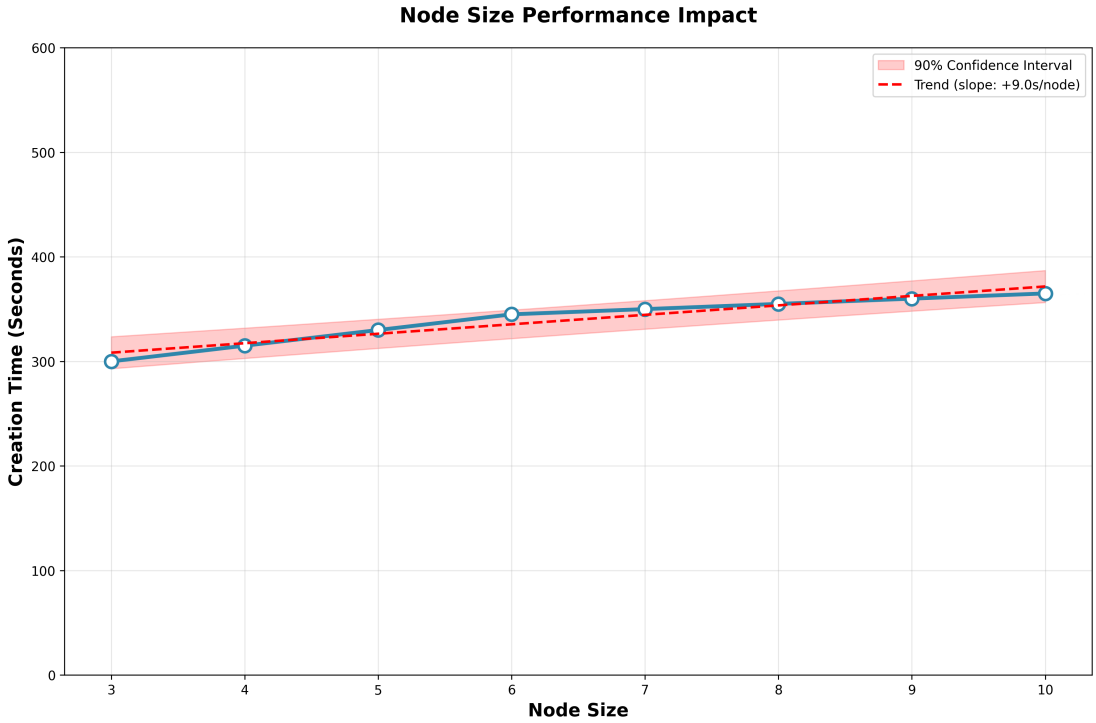

Figure 6.2: Cluster creation time comparison based on dimension

provisioning tasks to run concurrently across multiple nodes. Consequently, the total execution time is primarily influenced by hardware performance and available network bandwidth, rather than the cluster size itself. The only component that operates in a single-threaded manner is the Elemento API server, which accounts for the slight increase in provisioning time observed when adding individual nodes.

6.1.2 Complexity compared to other provisioning methods

In any case, creating a cluster in the cloud requires first setting up an account with the chosen cloud provider. Since the user experience and perceived complexity of this process can vary significantly and are often subjective, this factor will not be considered in the evaluation.

There are several approaches to provisioning a Kubernetes cluster across different cloud providers. The most common methods include using the provider's dashboard, public APIs, Terraform, or external tools such as kOps (which itself operates through public APIs). While all these approaches provide a degree of automation, each requires a learning curve to be used effectively.

Among these, the most challenging is undoubtedly the direct use of public APIs, due to their inherent complexity and the often limited clarity of the documentation. Terraform offers a more structured approach but introduces its own domain-specific syntax that must be learned. Provider dashboards are generally simpler to use, but their functionality and user experience vary significantly between providers, making them highly subjective.

External tools like kOps, by contrast, provide a higher level of abstraction compared to the aforementioned solutions. They typically generate an optimized configuration that adheres to both security standards and industry best practices. This enables the creation of production-grade clusters that include all essential components without requiring developers to directly manipulate Kubernetes configuration files or deployment manifests.

A distinctive feature of the Elemento system is its ability to operate seamlessly in both cloud and on-premise environments. This dual capability makes it particularly valuable for organizations that maintain hybrid infrastructures or prefer to keep sensitive workloads on-premise. Compared to the traditional manual startup process that requires extensive configuration files, Elemento provides a significant simplification. By leveraging kOps, the complexity for developers is reduced dramatically: instead of writing and maintaining hundreds of lines of configuration, they can achieve the same result with just two commands on the terminal.

This not only accelerates the provisioning process but also minimizes the risk of human error, ultimately improving reliability and lowering the entry barrier to Kubernetes adoption, which is often driven by ease of use and is particularly important for smaller teams with limited operational expertise.

6.1.3 Operating cost of the cluster

Cost is a crucial factor to consider when managing Kubernetes clusters, as it can vary significantly across providers. Some providers apply fixed pricing models, others charge specifically for the managed control plane, and a few offer the control plane free of charge. For the purpose of a fair comparison, we consider a cluster composed of 10 virtual machines, each with 2 vCPUs, 8 GB of RAM, and 100 GB of storage. These machines represent the worker and control plane nodes of the cluster. The monthly cost under a 12-month commitment is illustrated in Fig.6.3. The 12-month commitment is highlighted since many providers offer substantial discounts for long-term usage.

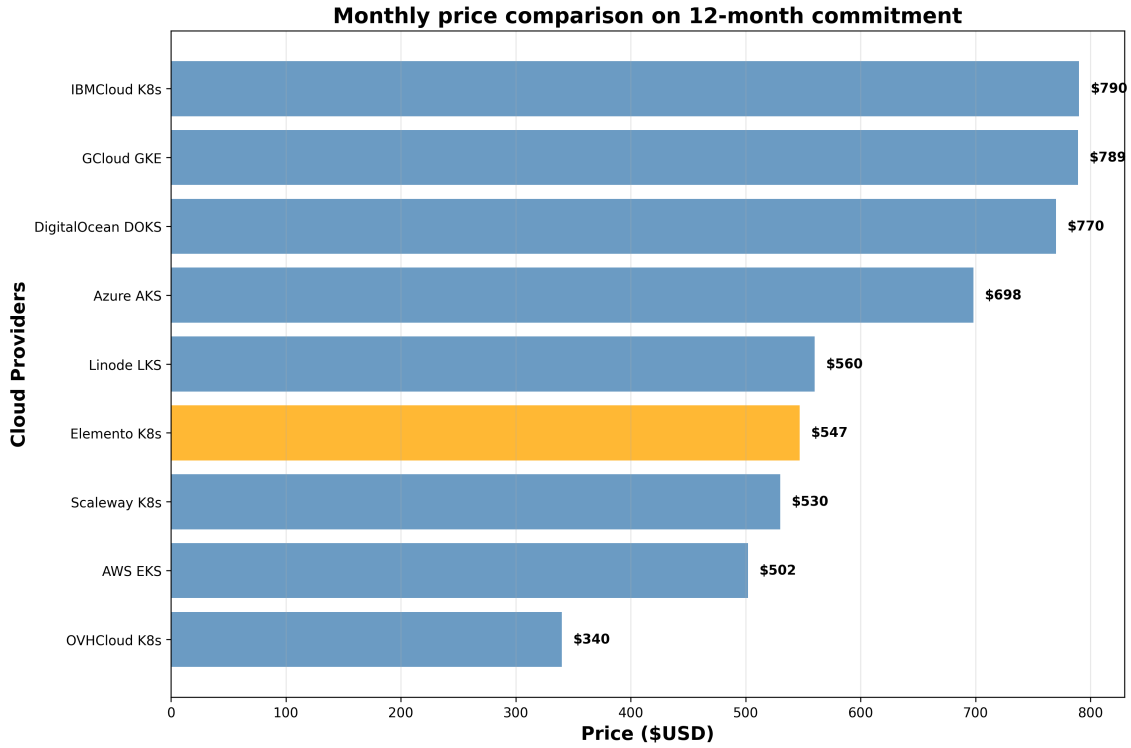

Figure 6.3: K8s clusters pricing comparison based on 2023 data - source [14]

The Elemento option price is calculated based on a managed cloud configuration. A significant portion of the cost is attributable to the "managed" component, which reflects the inclusion of monthly man-hours required for system maintenance, upgrades, and client support. This service element ensures that the infrastructure remains up to date and operational, while also reducing the operational burden on the client.

The Elemento pricing model can be further optimized through economies of scale. By aggregating multiple client orders, the system can take advantage of bulk discounts offered by cloud providers, which are typically unavailable to individual clients.

For an on-premise solution, the cost structure would differ, as it involves a one-time payment. In this case, the estimated price would be approximately €5,543 for a Dell PowerEdge R6615 server that matches the required specifications.

6.2 How reliable is it?

Reliability is one of the most important aspects to consider when working with infrastructure. Ensuring compatibility in the cloud ecosystem is often challenging due to the variety of standards, distributions, and proprietary implementation logics.

To address operating system distribution issues, I chose a default image for the scope of this thesis: Ubuntu 22.04 LTS. The translation of implementation logic was carried out to make the kOps system compatible with the Elemento framework, although some aspects, such as SSH key management, are still mocked, since they are not yet supported by Elemento. Regarding Kubernetes, I tested version 1.32.4, one of the most recent releases available at the time of writing. Due to time constraints, the system has not undergone extensive testing across different VM flavors and specification combinations. This limitation is acknowledged and highlighted as part of the future development work required to evolve the solution into a production-ready system.

Chapter 7

Conclusions

7.1 Outcome of the thesis

This thesis presents a proof of concept that serves as a solid foundation for the development of a future product. It demonstrates both the potential and the challenges involved in creating and managing a fully functional Kubernetes cluster. The company intends to further extend and refine the kOps codebase, integrating it into its service offerings. In addition, several promising future improvements and features have been identified, which could enhance the system and contribute unique value to the Elemento implementation. This work highlights a concrete path toward simplifying Kubernetes adoption and enabling more accessible hybrid cloud management for a broader range of organizations.
The source code developed for this thesis is publicly available in the following GitHub repositories:
Elemento kOps fork:
`https://github.com/Elemento-Modular-Cloud/kops`
Elemento Go plug-in library:
`https://github.com/Elemento-Modular-Cloud/tesi-paolobeci`

7.2 Future improvements

7.2.1 Extend to multi-AtomOS server scenario

As of now, the system has been created and tested on a single AtomOS server, both in cloud and on-premise environments. A valuable future expansion would be the implementation of a multi-server architecture, which would enhance redundancy and increase resilience against service disruptions in a specific region or provider. Such an extension could allow the system to provision and manage multiple control planes, with each AtomOS node hosting its own control plane. These control planes would then be interconnected through encrypted VPN tunnels, ensuring both security and reliability of inter-node communication. The main challenge in this scenario would likely be the latency and stability of the inter-server connections, since Kubernetes is highly sensitive to network performance.

7.2.2 Auto-scaling

Another possible future improvement is the implementation of virtual machine auto-scaling based on workload. By establishing a direct integration between the Kubernetes APIs and the AtomOS APIs, it would be possible to allow Kubernetes controllers to dynamically manage the underlying infrastructure. In this way, the cluster could automatically scale out by provisioning additional VMs when workloads increase, and scale in by releasing resources during low demand periods. Such functionality would not only improve resource utilization and cost efficiency, but also bring the Elemento system closer to the level of automation offered by large-scale cloud providers. It would enable organizations to react more effectively to fluctuating workloads while maintaining high availability and performance.

7.2.3 Observability

To further simplify system administration, the integration of an observability tool should be considered as part of the Elemento system. A common choice would be a Grafana dashboard, which could provide real-time telemetry on VM resource usage, network activity, and overall cluster health. Such a dashboard would not only give administrators greater visibility into the system but could also work in conjunction with the auto-scaling mechanism. The same measurement points used to trigger scaling events could also serve as monitoring indicators, thereby reducing duplication of effort and ensuring consistency between monitoring and scaling logic. This integration would allow sysadmins to both oversee system performance and trust that the infrastructure can automatically adapt to workload changes, improving operational efficiency and reducing the risk of undetected bottlenecks.

Bibliography

[1] Fernando Almeida, Oliveira José, and Cruz José. *Open Standards And Open Source: Enabling Interoperability*. 2011. URL: https://www.researchgate.net/publication/49612109_Open_Standards_And_Open_Source_Enabling_Interoperability (visited on 09/18/2025).

[2] Kubernetes authors. *Docs: Installing kubeadm*. 2025. URL: https://kubernetes.io/docs/reference/setup-tools/kubeadm/ (visited on 07/30/2025).

[3] AWS. *What is debugging?* 2025. URL: https://aws.amazon.com/what-is/debugging/ (visited on 09/06/2025).

[4] Kelsey Hightower Bredan Burns Joe Beda. *Kubernetes up & running: Dive into the Future of Infrastructure*. Oreilly, 2019, pp. 1–100.

[5] CNCF. *Governance overview*. 2025. URL: https://contribute.cncf.io/maintainers/governance/overview/ (visited on 09/18/2025).

[6] Community. *Kubernetes Documentation*. 2024. URL: https://kubernetes.io/docs/home/ (visited on 06/25/2025).

[7] Delve community. *GitHub repository*. 2025. URL: https://github.com/go-delve/delve (visited on 09/06/2025).

[8] Company. *Elemento*. 2025. URL: https://www.elemento.cloud/en (visited on 06/11/2025).

[9] Company. *Elemento AtomOS*. 2025. URL: https://www.elemento.cloud/en/technology/atomos (visited on 06/11/2025).

[10] Company. *Elemento Cloud Netowrk*. 2025. URL: https://www.elemento.cloud/en/technology/cloud-network (visited on 06/11/2025).

[11] Docker Company. *What is docker?* 2025. URL: https://www.docker.com/resources/what-container/ (visited on 06/25/2025).

[12] Apache Software Foundation. *Apache License Version 2.0*. 2025. URL: https://www.apache.org/licenses/LICENSE-2.0 (visited on 09/18/2025).

[13] The Linux Foundation. *CNCF landscape*. 2025. URL: https://landscape.cncf.io (visited on 06/28/2025).

[14] Elliot Graebert. *Comparing the Top Eight Managed Kubernetes Providers*. 2023. URL: https://medium.com/@elliotgraebert/comparing-the-top-eight-managed-kubernetes-providers-2ae39662391b (visited on 09/22/2025).

[15] Red Hat. *What is an SDK?* 2020. URL: https://www.redhat.com/en/topics/cloud-native-apps/what-is-SDK (visited on 06/28/2025).

[16] John Arundel Justin Domingus. *Cloud Native DevOps with Kubernetes: Building, Deploying and Scaling Modern Applications in the Cloud*. Oreilly, 2022, pp. 1–100.

[17] Alhosban A.; Pesingu S.; Kalyanam K. "CVL: A Cloud Vendor Lock-In Prediction Framework". In: *Mathematics* 12.387 (2024), pp. 1–10.

[18] kOps. *Docs: Commands and arguments*. 2025. URL: https://kops.sigs.k8s.io/getting_started/commands/ (visited on 07/31/2025).

[19] kOps. *Docs: How it works*. 2025. URL: https://kops.sigs.k8s.io/contributing/how_it_works/ (visited on 06/28/2025).

[20] Kubespray. *Repository: How it works*. 2025. URL: https://github.com/kubernetes-sigs/kubespray (visited on 07/30/2025).

[21] Mateo Mojica. *Navigating large codebases. Tips and Tricks.* 2022. URL: https://mateo-mojica.medium.com/navigating-large-codebases-tips-and-tricks-153901231093 (visited on 09/06/2025).

[22] Alexander Potasnick. *AKS vs EKS vs GKE: Managed Kubernetes services compared*. 2023. URL: https://www.pluralsight.com/resources/blog/cloud/aks-vs-eks-vs-gke-managed-kubernetes-services-compared#:~:text=GKE%20has%20the%20most%20available,31 (visited on 07/30/2025).

[23] Pulumi. *Docs: Terraform vs pulumi?* 2025. URL: https://www.pulumi.com/docs/iac/concepts/vs/terraform/#:~:text=Terraform (visited on 07/30/2025).

[24] Rancher. *Docs: What is Rancher?* 2025. URL: https://ranchermanager.docs.rancher.com (visited on 07/30/2025).

[25] Kubernetes SIGs. *Docs: What is Kubernetes Cluster API*. 2025. URL: https://cluster-api.sigs.k8s.io/#:~:text=Cluster%20API%20is%20a%20Kubernetes,and%20operating%20multiple%20Kubernetes%20clusters (visited on 07/30/2025).