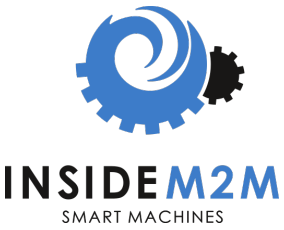
Politecnico di Torino

Master's Degree Thesis



Modernizing infrastructure provisioning for IoT platform

A case study for the M2MGate platform

Supervisors:
Prof. Malnati Giovanni
Kasten Sven
Hellwich Ilja

Candidate:
Borgone Roberto

Academic Year 2025/2026

To my mother Francesca, my father Cosimo, and my brother Andrea: thank you for your trust and support. I will forever remember the unconditional love I have received throughout these years. Being apart is the hardest challenge I face every day, yet I keep you close to my heart. To my friends across Ostuni, Turin, and Berlin. To Ilja for helping me during this work, and to Bethany, with whom I share this exciting life—thank you.

Abstract

The provisioning and management of complex, multi-component IoT platforms present significant operational challenges, particularly when solutions must be tailored to individual customer requirements. These challenges often lead to increased operational costs, slower deployment cycles, and ambiguous team responsibilities. This thesis addresses these issues through a case study of M2MGate, a commercial IoT platform. The objective is to modernize the platform's infrastructure provisioning process by analyzing existing business workflows and identifying key areas for improvement.

The proposed solution implements a fully automated, cloud-native architecture that leverages GitOps principles to manage the system's lifecycle. This approach streamlines deployments, reduces manual intervention, and enforces a clear separation of concerns between development and operations teams.

This work makes five main contributions. First, it provides a detailed analysis of the existing provisioning process, identifying bottlenecks and inefficiencies within the organizational structure. Second, it details the migration of the M2MGate platform and its third-party dependencies to a Kubernetes-based, cloud-native architecture, including partial refactoring of the system codebase. Third, it presents the design and implementation of "Blue", a custom platform solution that abstracts the complexity of underlying infrastructure as code technologies such as Crossplane and OpenTofu, providing a unified interface for the end-to-end provisioning process. Fourth, it facilitates platform adoption by developing a command-line utility and comprehensive documentation. Finally, it evaluates the solution, demonstrating the benefits of the proposed approach. This research demonstrates a practical approach to managing the complexity of large-scale, customizable IoT systems in diverse environments.

Contents

List of Figures

List of Tables

List of Listings

Chapter 1

Introduction

In the contemporary digital landscape, the velocity of software development and the reliability of application deployment are paramount for business success. The traditional approach to infrastructure provisioning, characterized by manual configuration and bespoke setups, has increasingly become a bottleneck. This manual process is time-consuming and error-prone, leading to inconsistencies across environments, a phenomenon often referred to as configuration drift. Such inconsistencies result in unpredictable behavior, making it difficult to guarantee that software will perform as expected when moved from development to production.

Automating the infrastructure provisioning process has emerged as a critical practice to address these challenges. By codifying infrastructure requirements and employing automated tools to create and manage resources, organizations achieve Infrastructure as Code (IaC). This paradigm shift brings the rigor and benefits of software development practices to infrastructure management: version control, code reviews, and automated testing foster a culture of collaboration and accountability.

Automation offers three primary benefits. First, it enhances deployment speed significantly. New environments can be provisioned in minutes rather than days or weeks, allowing development teams to iterate faster and deliver value to customers more quickly. Second, automation ensures predictability and reliability. By defining infrastructure as code, the provisioning process becomes repeatable and consistent. Every deployment is based on version-controlled templates, eliminating configuration drift and ensuring environmental consistency. This minimizes the risk of production failures due to environmental discrepancies. Third, automation reduces operational costs by minimizing manual effort, allowing operations teams to focus on higher-value tasks.

Automating infrastructure provisioning has become a necessity for modern software organizations. It is foundational for building scalable, resilient, and secure systems, enabling businesses to innovate with confidence and respond effectively to evolving market demands. This thesis explores how these principles can be applied to modernize infrastructure provisioning for an IoT platform, demonstrating the tangible benefits of a fully automated, cloud-native approach.

1.1 Problem Statement

This thesis addresses the modernization of the infrastructure provisioning process for M2MGate, a commercial IoT platform. The existing process relies heavily on manual intervention, creating several significant challenges. Each customer deployment requires a tailored setup, and the lack of automation leads to slow and error-prone provisioning cycles. This manual approach increases operational costs and introduces inconsistencies between customer environments, resulting in configuration drift.

Furthermore, the existing workflow lacks clear separation of concerns between development and operations teams, complicating troubleshooting, slowing problem resolution, and hindering overall efficiency. There is a pressing need to re-engineer the provisioning strategy to align with modern DevOps practices, reduce deployment times, and establish a clear, scalable, and cost-effective operational model.

1.2 Contribution

This thesis makes five main contributions to address the challenges outlined in the problem statement:

1. **Analysis of the Existing Provisioning Process:** A comprehensive analysis of current M2MGate provisioning workflows, identifying key bottlenecks, operational inefficiencies, and structural issues that hinder agility and increase costs.

2. **Migration to a Cloud-Native Architecture:** Migration of the M2MGate platform and its dependencies to a Kubernetes-based, cloud-native architecture, including partial refactoring of the system's codebase to ensure compatibility and leverage container orchestration benefits.

3. **Development of a Custom Developer Platform ("Blue"):** Design and implementation of a fully automated infrastructure provisioning workflow using GitLab CI for task automation, Crossplane and OpenTofu for Infrastructure as Code (IaC), and FluxCD for continuous state reconciliation, enabling a GitOps-centric environment management approach. These tools are integrated into a unified platform solution providing a single interface for end-to-end provisioning.

4. **User Experience Enhancements:** Development of a command-line utility for interacting with platform functionalities and comprehensive documentation covering platform usage, releases, and contribution guidelines to facilitate adoption.

5. **Solution Evaluation:** Assessment of the platform's ability to provision environments and deliver the full set of M2MGate features at scale through testing and benchmarking.

These contributions demonstrate a practical approach to modernizing a legacy IoT platform's infrastructure, showcasing the benefits of automation, cloud-native principles, and clear operational methodology.

Chapter 2

Background and Motivation

2.1 M2MGate Overview

M2MGate is an industrial-grade Internet of Things (IoT) platform designed to support machine-to-machine communication across various industries. It serves as a central hub for data aggregation, management, and analysis, facilitating seamless connectivity between devices, sensors, and applications. M2MGate provides a framework for developing end-to-end solutions tailored to specific industry needs through integration with custom software designed for each customer's unique requirements.

2.1.1 Core components

The platform's architecture is composed of several key components that work together to provide robust IoT capabilities:

- **Cascade Service**: Manages the communication protocol. It registers devices and backend services, manages the routing of RPCs and data deliveries between all participants, and distributes metadata about devices, services, and their interactions.
- **Core Service**: Provides persistence for device metadata and historical time series data. Manages device authentication and authorization policies, and provides APIs to interact with the stored data.
- **Distribution Service**: Offers the creation of configuration and firmware updates resources to be distributed to devices in the field.
- **SIM Info Service**: Interacts with the APIs of different mobile network operators to gather information about SIM cards employed for mobile device connectivity, including their status, usage statistics, billing details, etc.
- **Message Adapter Service**: Acts as a middleware between devices and a message broker (e.g. Apache Kafka) to enable 2-way event streaming from and to backend services.
- **Geocoder Service**: Determines the location of devices based on cells or wifi access points information.

- **Crypto Service**: Manages cryptographic keys and provides functions for encryption, decryption, and secure communication.
- **Message Service**: Forwards notifications to 3rd party services and systems: SMS messages, emails, push notifications, etc.
- **Dtools Service**: Provides direct administration and management access to devices in the field, allowing for remote commands and diagnostics.
- **Edge Service (Container Service)**: Extension of M2MGate to deliver and control 3rd party applications on the M2MDevice. Mainly used to control Docker daemons running on the devices.
- **Tenant Service**: Implements logical separation of resources, ensuring data isolation and security across different organizations using the platform.
- **VPNGate Service**: Establishes a secure virtual private network (VPN) connection to devices in the field, enabling encrypted communication over insecure networks.
- **M2MGate Portal**: Web portal that allows users to manage devices, monitor data, and configure platform settings through an intuitive UI
- **License Service**: Manages the billing of the M2MGate platform usage according to the customer's license.

2.1.2 Third-party components

M2MGate is deployed alongside a set of third-party applications, some of which are strict dependencies while others are optional depending on specific needs. These include:

- **Keycloak**: Open-source Identity and Access Management (IAM) system that provides authentication and authorization services.
- **Apache Kafka**: An open-source distributed streaming platform capable of handling high-throughput data streams.
- **Prometheus & Grafana**: Prometheus is a monitoring tool used for collecting, storing, and querying time series data. Grafana provides visualization capabilities, enabling the creation of dashboards to display metrics in an understandable way.
- **MySQL**: Relational database management system (RDBMS) that serves as a central repository for platform data.
- **Logstash, OpenSearch, OpenSearch Dashboards**: Logstash is used for processing logs, while OpenSearch provides full-text search capabilities. OpenSearch Dashboards offers visualization tools to analyze log data.

- **Traefik**: A reverse proxy that handles routing of incoming requests to the appropriate backend services.
- **Portainer**: A web-based management UI for Docker environments, simplifying container management tasks.

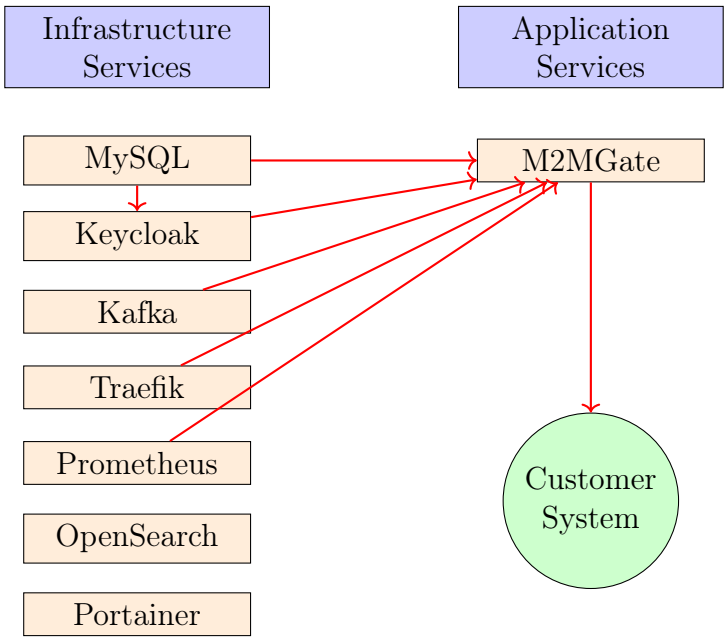


Figure 2.1: Service categories and deployment dependencies

2.1.3 Infrastructure

A typical installation of M2MGate is composed of a cluster with at least three nodes, each node is usually a VPS (Virtual Private Server) with the following specifications:

- **CPU cores**: 16 (x86)
- **RAM**: 64GB DDR5 ECC
- **Disk**: 2TB NVMe SSD
- **Network Bandwidth**: 2.5 Gbps
- **OS**: Debian 12

A set of tools must be present on each node to support the installation and maintenance of M2MGate, among these we can mention:

- **Docker**: Containerization platform that allows running applications in isolated environments. It is configured in "Swarm mode" and used as cluster orchestrator.

- **GlusterFS**: Distributed file system used for shared storage across multiple nodes to ensure data availability and redundancy. Some services require persistent volumes, which are provided by mounting a path of the host filesystem into their container. In order to avoid binding a container to a specific node, GlusterFS is used to provide a shared filesystem that is accessible by all nodes.

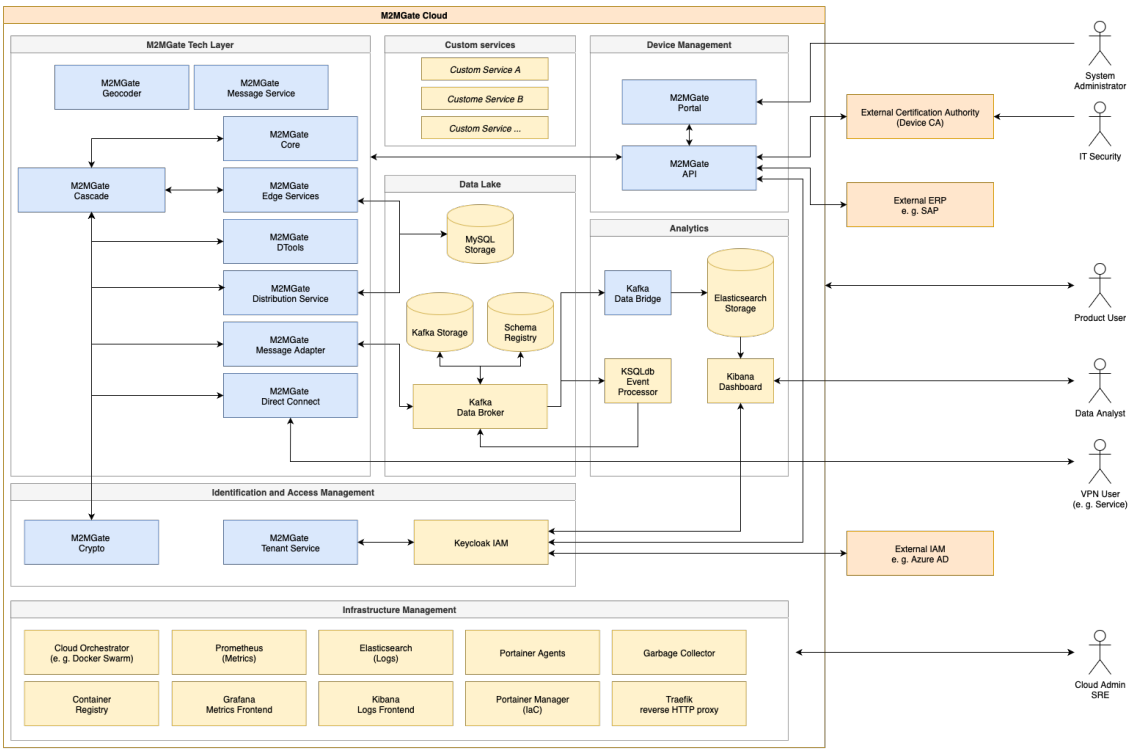


Figure 2.2: M2MGate overview

2.2 M2MGate Provisioning

M2MGate provisioning encompasses the creation of all necessary services to run the platform, including server configuration, Docker Swarm cluster setup, shared storage deployment using GlusterFS, and deployment of all required services. The process is divided into several stages:

- **Server Initialization**: Basic server preparation, including installation of necessary software and packages.
- **Network Configuration**: Configuration of the network to create a private subnet for the cluster. Setup of nodes firewall rules. Creation of DNS names.
- **Cluster Initialization**: Initialization of Docker Swarm and preliminary operations supporting the service stack deployment.
- **Service Deployment**: Deployment of M2MGate services onto the Docker Swarm cluster.

- **Post-Deployment Configuration**: Additional post deployment configuration for full functionality.

Table 2.1 presents a high-level overview of these provisioning stages.

Stage	Task	Type
Server Initialization	Purchase the servers and connect them with a vLAN	Manual
	Create users on each server	Semi-automated
	Configure authorized SSH keys for each user for each server	Semi-automated
	Configure the SSH daemon on each server	Manual
	Install utility tools on each server (e.g. nano, tcpdump, dnsutils, etc.)	Manual
Network Configuration	Setup firewall rules on each server	Semi-automated
	Configure VPN tunnel to access the company private network	Manual
	Create DNS records for each server and a wildcard DNS record pointing to all servers	Manual
Cluster Initialization	Initialize Docker Swarm on manager node and join worker nodes to the swarm	Manual
	Install GlusterFS on each node and initialize a shared volume	Manual
	Create App registration in Microsoft Entra ID for cluster single sign-on	Manual
Service Deployment	Provide initial service configurations and deploy M2MGate services onto the cluster	Automated
Post-Deployment Configuration	Setup replication in MySQL	Manual
	Create users and databases in MySQL	Manual
	Create a realm in Keycloak with the required clients, groups, roles and role mappings	Manual
	Upload of custom Keycloak themes in container volume	Manual
	Configuration of OIDC variables in applications that require them	Manual
	Generation and upload in M2MGate services volumes of TLS material	Manual
	Setup of dns-01 challenge for the Let's Encrypt wildcard certificate issuance	Manual
	Setup alert rules via monit utility and/or grafana	Manual

Table 2.1: High-level overview of M2MGate provisioning process stages and description.

2.3 M2MGate Blueprint

The M2MGate Blueprint is the current infrastructure-as-code solution addressing the Service Deployment stage described in Section 2.2.

2.3.1 Overview and Architecture

The M2MGate Blueprint consists of the following key concepts:

- **GitLab Repository**: A centralized version control system containing all Docker Compose and configuration files necessary for M2MGate provisioning.
- **Parametrization**: Docker Compose files and deployment scripts use parameters to allow customization of the setup according to different environments.
- **GitLab CI/CD Pipeline**: An automated deployment pipeline orchestrated through the `.gitlab-ci.yaml` file, providing continuous integration and deployment capabilities.
- **Environment Configuration Management**: Managing of environment-specific variables, secrets, and configurations through GitLab CI variables.

2.3.2 Deployment Pipeline Analysis

The M2MGate Blueprint deployment pipeline implements a multi-stage deployment process that automates the application of the Docker Compose files to a target environment. The pipeline architecture follows a sequential execution model with five distinct stages:

Pipeline Stages

The deployment pipeline consists of the following stages, executed in sequence:

1. **Service Copy** (`service-copy`): Synchronizes the complete repository content, including Docker Compose files and configuration files, to the target infrastructure.
2. **Network Initialization** (`service-init-networks`): Establishes the necessary Docker overlay networks that enable inter-service communication.
3. **Node Labeling** (`service-init-labels`): Applies role-based labels to the nodes, enabling service placement constraints.
4. **Secret Management** (`service-init-secrets`): Manages the creation and distribution of Docker secrets containing sensitive information such as SSL certificates, database passwords, and API keys required for service operation.
5. **Service Deployment** (`service-update`): Deploys the actual M2MGate services and supporting infrastructure components using Docker stack deployment commands, with environment-specific configuration injection.

Environment-Specific Deployment

The pipeline implements environment management through GitLab's branch-based deployment strategy. Each target environment is represented by a branch following the pattern `system/<domain>`. Pull requests against these branches trigger appropriate deployment workflows.

2.3.3 Environment Provisioning Workflow

Stakeholder Roles and Responsibilities

- **Operations Team**: Maintains the upstream Blueprint repository, manages infrastructure resources, and provides technical oversight for deployment processes.
- **Project Team**: Creates and manages environment-specific forks, configures customer-specific requirements, and executes deployment pipelines.

Workflow Overview

The environment provisioning workflow, illustrated in Figure 2.4, highlights the interactions between stakeholders and the sequence of operations required.

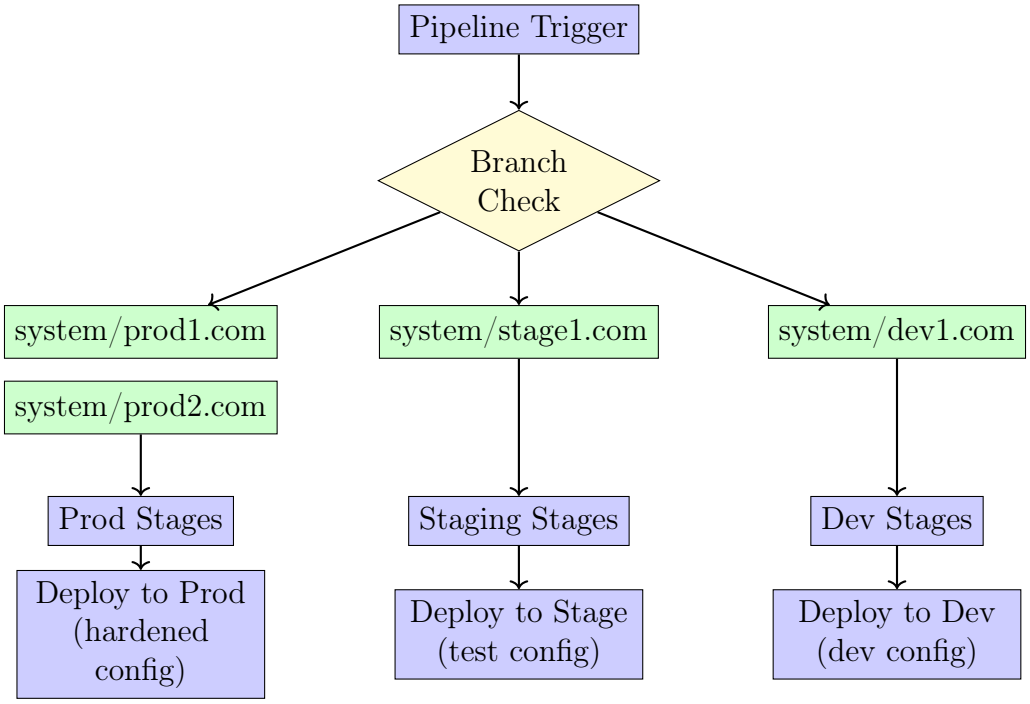


Figure 2.3: Pipeline execution flow based on branch

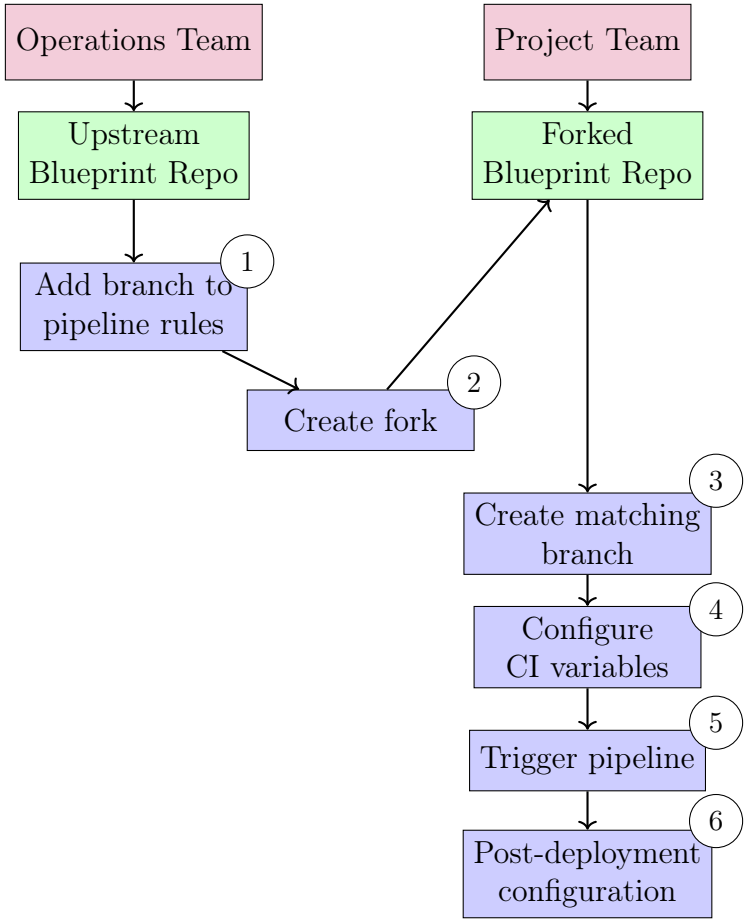


Figure 2.4: Environment provisioning workflow with stakeholder interactions

2.4 Motivation

The current M2MGate provisioning process presents several significant challenges that impede efficiency, reliability, and scalability. These challenges motivate this thesis, which aims to design and implement a more robust and automated solution.

2.4.1 Inefficiency and Time Consumption

As illustrated in Table 2.1, the provisioning process relies heavily on manual intervention. Critical steps, including server and network configuration, cluster initialization, and post-deployment tasks, are performed manually. This approach is inherently time-consuming, requiring several days to set up a new M2MGate instance. This significant time investment slows project timelines and limits organizational agility in responding to customer needs.

2.4.2 High Susceptibility to Human Error

The complexity and manual nature of the provisioning process make it highly susceptible to human error. Misconfigurations can occur at any stage, from firewall rule setup to Keycloak user role configuration, potentially causing system instability, security vulnerabilities, or complete deployment failures. Troubleshooting is often difficult and time-consuming, requiring deep system expertise and further exacerbating deployment delays. The lack of automation introduces slight, undocumented variations between deployments, resulting in inconsistent environments that are difficult to manage and maintain.

2.4.3 Lack of Comprehensive Documentation and Knowledge Dependency

The provisioning process lacks comprehensive formal documentation. The knowledge required for successful deployment is largely institutional, residing with a small number of experienced employees. This creates a significant bottleneck, as key individual availability directly impacts the ability to provision new instances. It also presents a steep learning curve for new team members and poses substantial business risk if knowledgeable personnel leave the company. The absence of clear documentation makes the process opaque and difficult for most employees to understand or contribute to.

2.4.4 Siloed Knowledge and Inter-Team Dependencies

Organizational silos compound the provisioning challenges by hindering effective collaboration. When a new customer project is initiated, a dedicated project team possessing deep knowledge of customer-specific requirements is assigned. However, this team lacks autonomy to provision or modify the M2MGate environment.

Instead, they rely on a separate operations team to perform deployment and manage infrastructure. This creates a critical dependency: the operations team,

while skilled in infrastructure management, typically lacks insight into unique customer specifications, especially as projects evolve over time. This separation without a shared source of truth (comprehensive documentation or automated processes) leads to communication overhead, potential misunderstandings, and delays. The project team cannot act on their knowledge, and the operations team must work from second-hand information, creating an inefficient and error-prone workflow that undermines clear separation of concerns.

These issues highlight the critical need to re-engineer the M2MGate provisioning process. This thesis addresses these shortcomings by introducing extensive automation, creating a provisioning system that is faster, more reliable, and more accessible to technical staff.

Chapter 3

Methodology

3.1 Technical Foundations

3.1.1 Internal Developer Platforms

This section examines the concepts of internal developer platforms (IDPs) as presented in the CNCF Platform White Paper [1], positions the system described in Chapter 2 within IDP frameworks, and outlines the improvements introduced by this thesis.

Conceptual framework

Internal developer platforms define a set of integrated capabilities, services and tools enabling standardized ways to provide support capabilities to application-specific logic through a consistent interface. For example, a very simple "platform" could be a wiki page with links to standard operating procedures to provision capabilities from providers. A more mature platform, on the other hand, would include self-service APIs, consumable by internal teams via GUI or command line. IDPs are born from the desire of organizations to refocus their delivery teams on their core function: developing applications. By investing in platforms, enterprises can:

- Reduce the cognitive load on product teams and thereby accelerate product development and delivery
- Improve reliability and resiliency of products relying on platform capabilities by dedicating experts to configure and manage them
- Accelerate product development and delivery by reusing and sharing platform tools and knowledge across many teams in an enterprise
- Reduce risk of security, regulatory and functional issues in products and services by governing platform capabilities and the users, tools and processes surrounding them
- Enable cost-effective and productive use of services from public clouds and other managed offerings by enabling delegation of implementations to those providers while maintaining control over user experience

Current state

M2MGate Blueprint, as described in Section 2.3, represents a first step toward an internal platform; however, it lacks key characteristics to be considered a fully functional IDP:

- It streamlines and automates the provisioning stack after Docker Swarm installation. An IDP should provide a cross-cutting layer with consistent user experience across the entire provisioning process.
- A survey revealed that 42.9% of M2MGate Blueprint users identified lack of documentation as a top-three weakness of the current provisioning process. Documentation gaps exist for Server Initialization and Network Configuration stages, while Post-Deployment Configuration is only partially covered. Documentation is central to an IDP.
- 75.0% of users rated their confidence in using the Blueprint to provision a new environment at 3 or fewer points out of 5 (with 53.6% rating it 1 point), indicating room for simplification and user experience improvement. The cognitive load on project teams is too high, as the current solution fails to abstract provisioning complexity behind an easy-to-use interface.
- 75.0% of users rated their confidence in extending M2MGate Blueprint with additional functionalities at 3 or fewer points out of 5 (with 60.7% rating it 1 point). Enabling product teams to manage their own capabilities when necessary is an IDP requirement, as platforms should not limit users. Currently, provisioning additional capabilities requires creating an extra repository (and CI/CD pipeline) or manual deployment via Portainer or Docker CLI. Allowing Blueprint project-specific forks to diverge from upstream complicates maintenance for operation teams, indicating both documentation gaps and organizational issues related to the lack of self-service provisioning.

Improved aspects

This thesis focuses on improving aspects from the Platform Engineering Maturity Model [2] through re-engineering the current provisioning process. Current and expected maturity levels are summarized in Table 3.1, with explanations for each aspect below. The goal is not to achieve perfect maturity, but to increase the overall maturity level of the M2MGate provisioning process.

Aspect	**Current**	**Expected**
Investment	2 - Dedicated team	3 - As product
Adoption	1/2 - Erratic/Extrinsic Push	3 - Intrinsic Pull
Interfaces	1 - Custom processes	3 - Self-service solutions
Operations	1 - By request	1/2 - By request/Centrally tracked
Measurement	1 - Ad hoc	1 - Ad hoc

Table 3.1: Maturity levels of the current and desired provisioning process.

- **Investment:** A dedicated infrastructure team fills gaps within other teams and takes ownership of unowned resources (e.g., customer system monitoring). The goal is providing a user-focused solution requiring the operations team to collect feedback, publish roadmaps and changelogs, and develop solutions tailored to product team needs.
- **Adoption:** While the organization recognizes shared platform value, adoption is fragmented. Users have low motivation to learn the platform, and product teams sometimes maintain their own deployment scripts. Reducing cognitive load on product teams should lead to spontaneous platform adoption.
- **Interfaces:** Varying processes exist to provision different capabilities and services, without interface consistency. Custom processes address immediate individual or team needs through manual intervention. The new solution provides user autonomy and requires minimal maintainer support through streamlined implementation. A narrow range of available solutions leaves users with unique requirements uncertain how to proceed.
- **Operations:** Currently, platforms and capabilities are developed, published, and updated reactively based on ad hoc product team requests. The new solution centralizes organization-wide capability creation while defining lightweight processes for managing project-specific capability lifecycles. Infrastructure as Code is used throughout for easier traceability.
- **Measurement:** The solution currently lacks structured methods to gather platform success data. Usage and satisfaction metrics are gathered informally or through surveys.

3.1.2 GitOps

The proposed solution aligns with GitOps principles [3]. This section demonstrates how design choices satisfy each principle.

1. **Declarative**

   "A system managed by GitOps must have its desired state expressed declaratively."

   The solution creates one GitLab repository containing YAML definitions of specifically designed CRDs that describe the desired environment configuration and its software components. These definitions are human-readable, editable, and purely declarative, specifying *what* state is desired rather than *how* to achieve it.

2. **Versioned and Immutable**

   "Desired state is stored in a way that enforces immutability, versioning and retains a complete version history."

   Adopting Git as state storage aligns with this principle. All changes are tracked through commits, creating an immutable history of applied changes.

3. **Pulled Automatically**

   "Software agents automatically pull the desired state declarations from the source."

   The environment state is managed by FluxCD, an agent running in a control plane cluster that continuously monitors the GitLab repository for changes. The control plane is a continuously available Kubernetes cluster running the platform tools.

4. **Continuously Reconciled**

   "Software agents continuously observe actual system state and attempt to apply the desired state."

   FluxCD ensures the control plane cluster maintains the desired state of user CRDs by comparing current cluster state with desired state stored in GitLab. Discrepancies trigger automatic reconciliation. Environment state, expressed as CRDs in the control plane, is applied to managed clusters by Crossplane controllers.

3.2 Design

This section outlines the design considerations that guided the implementation of the solution described in this thesis. The primary focus is on the choice of tools, architecture, and deployment strategies to achieve seamless end-to-end provisioning of both infrastructure and applications.

3.2.1 Choice of tools

Kubernetes

Kubernetes is an open-source container orchestration platform that automates deployment, scaling, and management of containerized applications [4]. It is the de facto industry standard for managing containerized workloads [5]. Replacing Docker Swarm with Kubernetes is motivated by its superior feature set, community support, and extensive ecosystem. Kubernetes' extensible API enables applications to implement advanced patterns and deeper automation.

Crossplane

Crossplane was selected as the main Infrastructure-as-Code (IaC) tool after comparing alternatives including Pulumi, OpenTofu, and Ansible:

- **Ansible**: Widely used for configuration management, its imperative nature does not align with declarative GitOps principles.
- **Terraform**: A mature IaC tool supporting multiple cloud providers with a large community, rich ecosystem, and strong CI/CD integration [6]. In 2023, OpenTofu was launched as a fork under a more permissive open-source license.

- **Pulumi**: A newer IaC tool supporting multiple languages and infrastructure providers with focus on developer productivity through various SDK options. Like Terraform, it uses internal state representation to plan changes. State must be stored externally for collaboration, using backends like Pulumi Cloud (paid) or S3-compatible storage like Minio [7].

- **Crossplane**: An open-source multi-cloud management tool extending the Kubernetes API to control external cloud resources (GCP, AWS, Azure) using native Kubernetes definitions [8]. It enables custom abstractions through CRDs and function pipelines without complex controller code, managing infrastructure and software as Kubernetes resources. This provides full GitOps compliance but requires a dedicated control plane cluster to orchestrate managed clusters. Cloud provider support, while growing, is not as extensive as Terraform's.

- **OpenTofu**: As a HashiCorp Terraform fork, it inherits all modules and plugins under a more permissive open-source license. It uses the same state management system, well-suited for GitLab CI/CD integration [9].

Crossplane was selected for its full GitOps compliance. The maturity gap with Terraform has been addressed by the Crossplane team through three approaches:

1. **Growing native Providers**: Crossplane contributors are expanding external API integration and broadening the ecosystem through Community Extension Projects [10].

2. **Upjet project**: A code generation framework enabling Crossplane Provider generation from Terraform providers [11].

3. **Provider OpenTofu**: Upbound, the company behind Crossplane, developed the OpenTofu Provider wrapping OpenTofu functionality, enabling use of existing Terraform HCL definitions as Crossplane resources [12].

OpenTofu

OpenTofu is still used for managing the control plane cluster, leveraging GitLab CI/CD integration via the OpenTofu CI/CD Component [13]. OpenTofu is also required for INSIDE M2M's cloud provider, Hetzner Cloud. While Crossplane integrates with many cloud providers, it lacks a native Hetzner Provider. Therefore, the infrastructural components of platform-created environments are implemented as OpenTofu modules applied via the OpenTofu Provider.

GitLab CI/CD pipelines

GitLab CI/CD pipelines are used for automation in various contexts of the proposed solution.

- As mentioned above, one of the purpose they serve is to deploy and manage the infrastructure of the control plane. This choice, opposed to the creation of dedicated software or scripts, allows to operate on the IaC codebase of the control plane right where it resides.
- The GitLab API can be leveraged to trigger automation from different kind of user interfaces (e.g. CLI, GUI). Because of this pipelines are also used as entry point for the automation of the environments provisioning process.
- Automation regarding the software release process as well as the packaging of software artifacts and their versioning are handled by means of CI/CD pipelines.

KCL

The M2MGate stack encompasses a wide range of applications. Producing configurations for these components is non-trivial. The YAML describing the control plane's desired state must be templated to produce different outputs based on user input.

- A popular choice is the Go template language, widely adopted through Helm as a Kubernetes package manager. However, Go templates are not designed for Kubernetes resources and do not guarantee valid YAML output, treating templates as plain text rather than structured YAML [14]. Templating and maintaining the vast set of Kubernetes resources composing the M2MGate platform using Go templates could be prohibitively complex.
- Another common solution is Jsonnet, a data templating language extending JSON with functions and other features. However, Jsonnet lacks schema validation. Third-party tools can generate Jsonnet code from Kubernetes resources for API compliance, but this approach is less straightforward than using a Domain Specific Language (DSL).

KCL is a Domain Specific Language (DSL) for complex configuration in cloud-native environments and a CNCF Sandbox Project [15]. Key features motivating KCL selection for environment customization include:

- **Semantic validation**: ensures code-level configuration validity through features like schema definitions, attribute requirements, type checks, and range constraints.
- **Flexibility**: leverages modern programming language features like conditional statements, loops, functions.
- **Rich tooling**: comes with IDE extensions and robust toolchains (e.g. KCL CLI) that enhance user experience.
- **Configuration Sharing**: configurations can be propagated using package management tools and OCI registries.

- **Integration**: it offers integrations for various SDKs and cloud-native tools.

CUE was also considered but discarded due to less maintained and mature Crossplane integration compared to KCL.

Nushell

Nushell offers the benefits of a rich programming language combined with a full-featured shell, simplifying complex script development [16]. It was selected for writing GitLab CI/CD pipelines and implementing the platform command-line interface.

3.2.2 Architecture

The platform architecture comprises the following principal elements:

- **GitLab instance (self-managed)**: The GitLab server hosting the Blue Platform repository is an active architectural component serving the following purposes:
  - Version control for the platform codebase
  - State storage for the control plane cluster
  - Control plane infrastructure state reconciliation via CI/CD pipelines
  - Documentation hosting
  - Platform capability exposure via REST API
  - Building and publishing platform artifacts including container images, Crossplane Configuration, KCL modules, and OpenTofu modules
- **Control Plane**: The control plane cluster hosts three core controllers:
  - **FluxCD**: Reconciles the desired state of the control plane cluster and managed clusters created by the platform.
  - **Crossplane**: Managed cluster state is described using Crossplane CompositeResources (XRs), which abstract complex Kubernetes object sets. FluxCD pulls CompositeResource definitions while Crossplane controllers process XRs by applying corresponding Compositions. Compositions typically create Crossplane Provider CRDs (CustomResourceDefinitions) in the control plane, which Crossplane Provider Pods use to create actual resources via external APIs.
  - **External Secrets Operator**: Manages secrets by making necessary sensitive information available in the control plane (cloud provider credentials, mail server accounts, GitLab tokens, etc.) by pulling from an external secret store. GitLab CI variables serve as the secret store. ESO also generates random credentials for services deployed in managed clusters (database passwords, admin accounts, etc.).

- **OCI Registry (Harbor)**: Repository for platform artifacts, connecting the GitLab instance publishing artifacts with the control plane cluster consuming them.

Figure 3.1 provides a graphical representation of the architecture, comprehensive of a managed cluster (Project Cluster) and its repository (Project Repository).

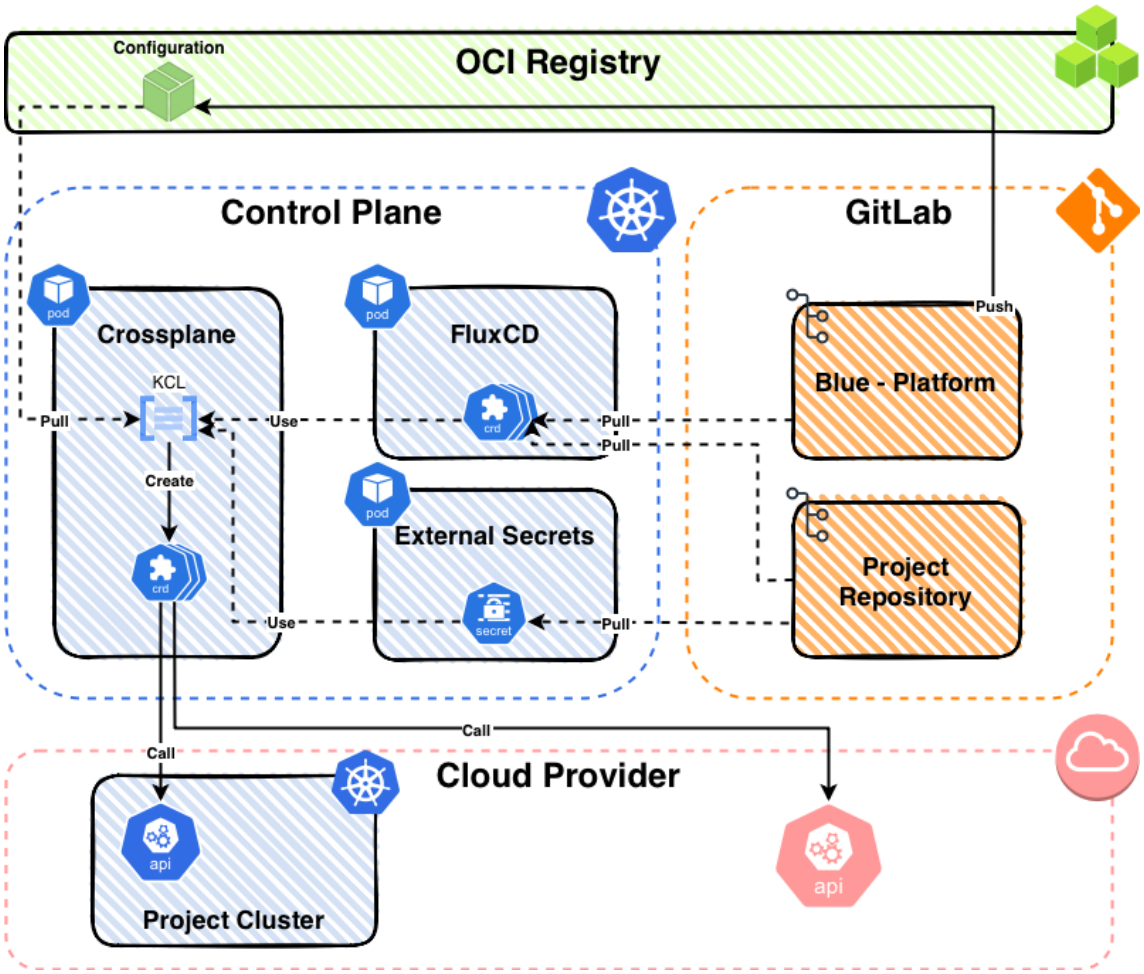


Figure 3.1: M2MGate overview

3.2.3 GitOps Workflows

A key design concept is the adoption of a GitOps approach. The platform goal is not only to provide automated environment provisioning but also to enforce collaboration and security best practices. The solution ensures infrastructure state has the following properties:

- **Easy to understand**: Desired infrastructure state, including server and network configurations, is defined by human-readable code stored in a Git repository, enabling easy inspection.

- **Trusted and auditable**: Infrastructure changes are recorded as version-controlled commits, simplifying change tracking.
- **Easy to manage**: Developers write declarative code describing desired state. Tools handle state achievement, simplifying management and reducing errors.

The platform produces two outputs from environment provisioning:

1. The actual environment comprising cloud resources (virtual machines, storage devices), network configurations, applications, and other resources required for M2MGate operation.
2. A Git repository containing a human-readable, declarative description of resource state, automatically synchronized (by FluxCD and Crossplane controllers) with the actual environment. Codebase changes modify desired infrastructure state.

The following sections detail workflows for environment creation, update, upgrade, and deletion, aligning with GitOps principles to provide seamless user experience. Some steps execute via CI pipelines triggered through a custom CLI, though ultimately all actions occur through Git state changes.

Creation of a new environment

Users initiate environment creation by running the `blue project create` command with a configuration file. This triggers a pipeline creating a new Git repository containing the environment's desired state as an M2MGateCluster CompositeResource definition. A subsequent pipeline job creates Namespace and FluxCD CRD definitions instructing the FluxCD agent to pull the M2MGateCluster from the new repository. These objects are pulled into the control plane, and Crossplane applies the relevant Composition to create the managed cluster and its applications.

Update and upgrade of an existing environment

Users initiate environment updates and upgrades by changing state in the environment's Git repository. When M2MGateCluster CompositeResource definition values are updated, FluxCD detects the change and pulls new values. This triggers Crossplane Composition with different output, updating the managed cluster.

Platform maintainers can evolve M2MGateCluster resource composition definitions over time. The Blue platform is versioned holistically to provide users with a coherent, evolving capability set. Users can upgrade existing environments to the latest environment specification using the same workflow as same-version updates, modifying a specific field in the M2MGateCluster CompositeResource definition to specify the desired Composition version.

Deletion of an existing environment

Users initiate environment deletion by running the `blue project delete` command, triggering a pipeline that deletes FluxCD CRDs responsible for reconciliation

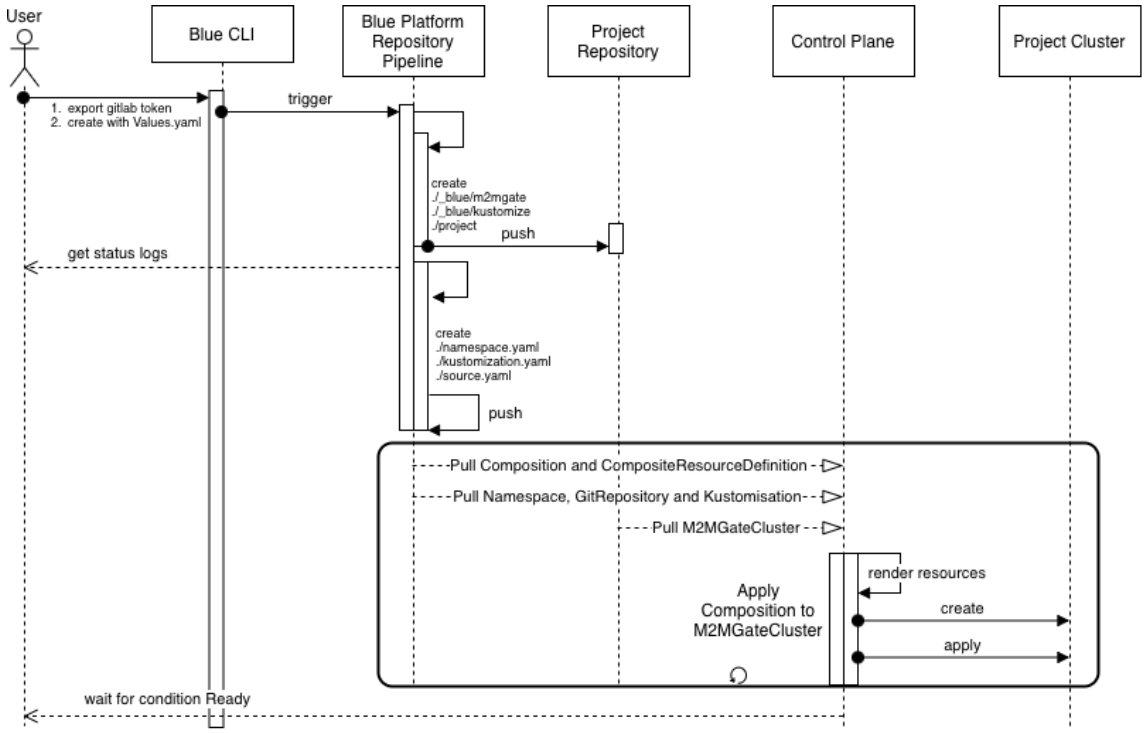


Figure 3.2: Environment creation workflow

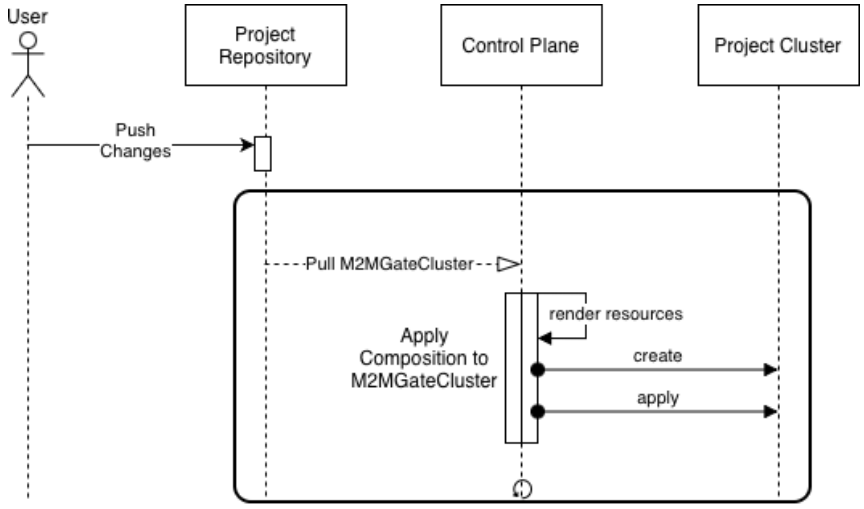


Figure 3.3: Environment update and upgrade workflow

of the M2MGateCluster resource from the control plane. The M2MGateCluster resource itself is then deleted, and Crossplane deletes the managed cluster and its applications. The CLI can optionally delete the Git repository containing environment state.

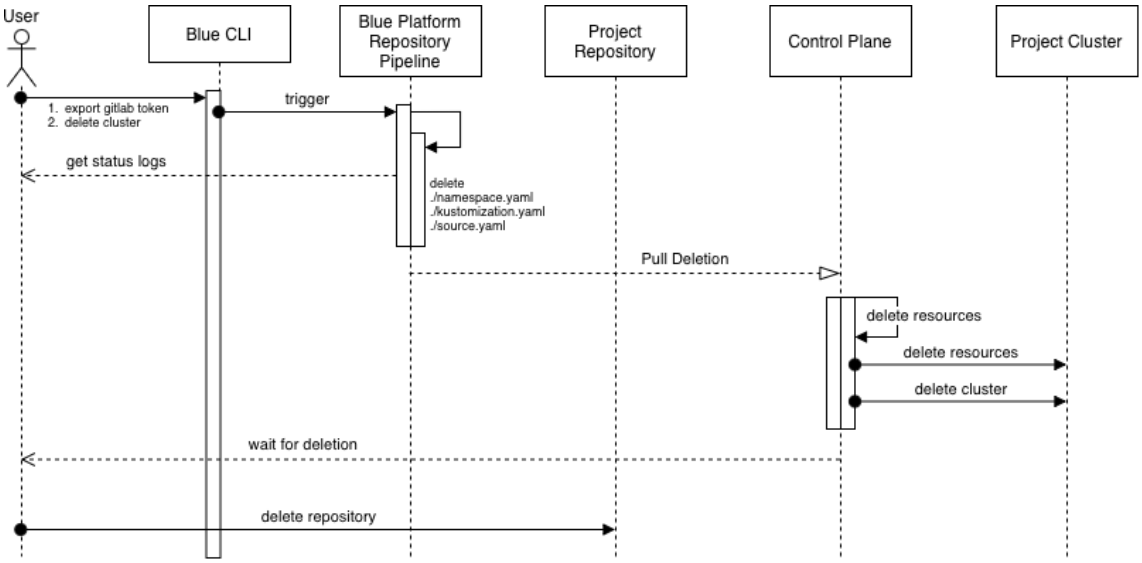


Figure 3.4: Environment deletion workflow

3.2.4 Documentation and user engagement

A key improvement goal is providing complete, user-friendly documentation. Understanding platform functionality and usage is paramount for adoption. Three initiatives address this:

- Contribution guidelines enforcing consistent commit message format, maintaining clean change history and enabling automatic release note and changelog generation.
- An internal communication channel for troubleshooting, support requests, release notifications, and general discussion.
- Complete platform documentation embedded in the codebase, built as static web pages using MkDocs and published on GitLab Pages via CI/CD pipelines. Documentation is built and published for different platform versions using an MkDocs plugin, ensuring users always access documentation relevant to their platform version.

Chapter 4

Implementation

The design presented in the previous chapter resulted from an iterative refinement process. While core concepts and principles such as Kubernetes, GitOps, and end-to-end automation remained consistent, the solution evolved to incorporate new elements as challenges arose during implementation. This chapter describes the implementation, focusing on aspects most relevant to the thesis.

4.1 Control Plane Setup

4.1.1 Infrastructure

As mentioned in Section 3.2.1, the control plane infrastructure was created using OpenTofu as the IaC tool. The following sections present implementation details.

State Backend

OpenTofu requires a storage backend for state data that must be shared among team members and support locking mechanisms for collaboration. GitLab's integrated Terraform state storage was selected as it stores state files securely with automatic encryption at rest and tracks changes through commit history [17]. State is updated each time a user runs OpenTofu commands through the GitLab CI/CD pipeline, implemented using the GitLab-maintained OpenTofu CI/CD Component [13]. Listing 1 shows jobs running the `plan` and `apply` steps when code changes occur. The `apply` step requires manual approval.

Cloud provider - Hetzner Cloud

Hetzner Cloud is a German cloud service provider offering virtual servers and storage. Its API enables programmatic resource management, making it suitable for OpenTofu automation. The provider was selected primarily for cost efficiency and the company's previous experience with its services. Hetzner Cloud hosts both the control plane and managed clusters. Cloud provider support can be extended to deploy managed clusters on other providers, and the company may migrate the control plane cluster to a different provider in the future.

```yaml
include:
  - local: pipeline/gitlab-opentofu/opentofu-component/job-templates.yml
    inputs:
      root_dir: control-plane/infra
      base_os: debian
      post_mr_plan_comment: true
      auto_delete_state: true

plan:
  extends: [.opentofu:plan]
  before_script:
    - export TF_VAR_hcloud_token="${HCLOUD_TOKEN_BLUE}"
    # other variables omitted for brevity
  rules:
    # other rules omitted for brevity
    - if: $CI_COMMIT_BRANCH
      changes:
        - control-plane/infra/**/*

apply:
  extends: [.opentofu:apply]
  rules:
    # other rules omitted for brevity
    - if: $CI_COMMIT_BRANCH == $CI_DEFAULT_BRANCH
      when: manual
      changes:
        - control-plane/infra/**/*
  before_script:
    - export TF_VAR_hcloud_token="${HCLOUD_TOKEN_BLUE}"
    # other variables omitted for brevity
```

Listing 1: GitLab pipeline using OpenTofu.

Kube Hetzner

Hetzner does not offer managed Kubernetes clusters; Kubernetes must be installed on purchased servers. The OpenTofu module `kube-hetzner` [18] was selected to simplify the setup process. This open-source, community-created module sets up K3s clusters on Hetzner Cloud, abstracting the lower-level `hcloud` provider by handling creation of necessary resources including servers, networks, firewall rules, and volumes. The operating system is openSUSE MicroOS, a small, hardened distribution designed for container orchestration [19]. `kube-hetzner` automatically configures control plane and worker nodes while offering customization options for cluster size, plugins, and initially deployed applications. The configuration:

- Configures SSH access to nodes
- Defines network region and server types for control planes and workers
- Establishes high-availability K3s with 3 control plane nodes, 3 workers, an autoscaling pool (up to 5 nodes), and 1 managed load balancer
- Deploys the following operators:
  - Hetzner Cloud Controller Manager
  - Hetzner CSI Driver

  - Kured
  - Traefik
  - Cert-Manager
  - Rancher System Upgrade Controller
  - Cluster Autoscaler
- Enables automatic OS and K3s version updates
- Configures a private Image Registry mirror

```
# variables definitions removed for brevity
# locals block removed for brevity
module "kube-hetzner" {
  hcloud_token = var.hcloud_token

  source  = "kube-hetzner/kube-hetzner/hcloud"
  version = local.kube_hetzner_version

  ssh_public_key  = "ssh-ed25519 AAAA...qOHv"
  ssh_private_key = var.ssh_private_key

  network_region = "eu-central"

  control_plane_nodepools = [
    {
      name        = "control-plane",
      server_type = local.control_plane_type
      location    = local.location
      count       = 3
      labels      = []
      taints      = []
    }
  ]

  agent_nodepools = [
    {
      name        = "agent"
      server_type = local.agent_type
      location    = local.location
      count       = 3
      labels      = []
      taints      = []
    }
  ]
```

Listing 2: Partial configuration for kube-hetzner module (1/2).

The MicroOS node image was created with HashiCorp Packer using the HCL declaration provided by `kube-hetzner` maintainers, modified to include additional tools (`telnet`, `nano`, `git`).

4.1.2 Controllers

FluxCD

FluxCD was selected to implement the continuous cluster reconciliation control loop. It integrates natively with Kubernetes toolkit components, using Kustomize to apply

```
  autoscaler_nodepools = [
    {
      name        = "autoscaled-agent"
      server_type = local.agent_type
      location    = local.location
      min_nodes   = 0
      max_nodes   = 5
    }
  ]

  load_balancer_type     = local.load_balancer_type
  load_balancer_location = local.location
  base_domain = local.base_domain

  # applications configuration removed for brevity

  k3s_registries = <<-EOT
    mirrors:
    registry.example.com:
      endpoint:
      - https://registry.example.com
    configs:
      registry.example.com:
        auth:
          username: robot@blue
          password: ${var.registry_robot_password}
  EOT
}
# providers definitions removed for brevity
```

Listing 3: Partial configuration for kube-hetzner module (2/2).

YAML definitions and encouraging repository structure best practices.

The repository follows the "Repo per environment" approach [20], subdividing resources into distinct folders containing Kustomize sub-folders watched by FluxCD via Kustomization CRDs. A main Kustomization CRDs file instructs FluxCD to pull resources from different folders.

FluxCD pulls three main resource sets into the control plane cluster: External Secrets Operator resources, Crossplane resources, and user project resources. Within these sets, specific deployment order is enforced when required.

Resources required to represent user projects in the control plane:

- **Namespace**: Isolates project resources.
- **GitRepository**: CRD pointing to the user project repository.
- **Kustomization**: CRD pointing to user project Kustomization sub-folders.

Figure 4.1 and Listing 4 demonstrate using Kustomize and FluxCD Kustomization CRDs to separate resource sets and model dependencies between:

1. External Secrets Operator namespace creation
2. External Secrets Operator deployment via HelmRelease resource

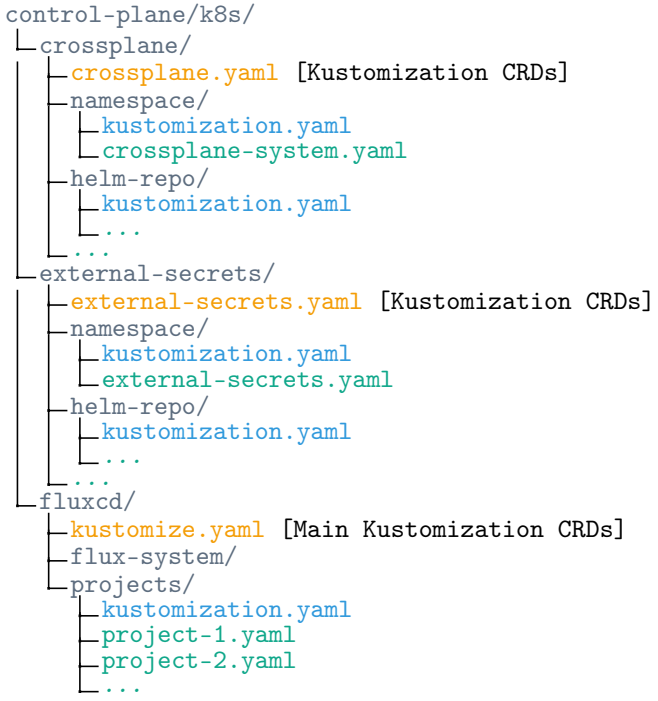


Figure 4.1: Control plane controllers repository structure

```yaml
apiVersion: kustomize.toolkit.fluxcd.io/v1
kind: Kustomization
metadata:
  name: external-secrets-namespace
  namespace: flux-system
spec:
  interval: 10m0s
  sourceRef:
    kind: GitRepository
    name: flux-system
  path: ./control-plane/k8s/external-secrets/namespace
  prune: true
  wait: true
---
apiVersion: kustomize.toolkit.fluxcd.io/v1
kind: Kustomization
metadata:
  name: external-secrets-helm-release
  namespace: external-secrets
spec:
  interval: 10m0s
  dependsOn:
    - name: external-secrets-namespace
  sourceRef:
    kind: GitRepository
    name: flux-system
    namespace: flux-system
  path: ./control-plane/k8s/external-secrets/helm-release
  prune: true
  wait: true
```

Listing 4: Snippet from external-secrets.yaml.

External Secrets

External Secrets Operator (ESO) is a CNCF Sandbox project primarily synchronizing secrets from external APIs into Kubernetes [21]. Additional features make it suitable for complete secrets management solutions. ESO serves three purposes in this work:

- Replicating required secrets across all managed cluster namespaces.
- Generating credentials for services deployed in managed clusters (database passwords, admin accounts, etc.).
- Fetching GitLab CI variables from the GitLab API into Kubernetes secrets. Each managed cluster requires global secrets shared between all clusters and cluster-specific secrets. Global secrets are pulled from a GitLab group containing all project repositories, while cluster-specific secrets are pulled from the project repository itself.

The External Secrets Operator API offers these resources:

- **SecretStore/ClusterSecretStore**: Defines secret sources (external API, local Kubernetes secrets, or random generator).
- **ExternalSecret/ClusterExternalSecret**: Defines secrets to fetch from the secret store and corresponding Kubernetes secrets to create.

Listings 5 and 6 show ESO replicating GitLab API access tokens and OCI Registry credentials for OpenTofu module pulls in each namespace presenting a label named `blue.m2mgate.cloud/project-slug`.

```yaml
apiVersion: external-secrets.io/v1
kind: ClusterSecretStore
metadata:
  name: shared-secrets-store
spec:
  provider:
    kubernetes:
      remoteNamespace: external-secrets
      server:
        caProvider:
          # details removed for brevity
      auth:
        serviceAccount:
          name: shared-secrets-store-service-account
          namespace: external-secrets
```

Listing 5: Replicate secrets in managed clusters namespaces (1/2).

Listing 7 shows SecretStore and ExternalSecret resources for GitLab secrets, generated from KCL code for each managed cluster.

```yaml
apiVersion: external-secrets.io/v1
kind: ClusterExternalSecret
metadata:
  name: shared-secrets
spec:
  externalSecretName: shared-secrets
  namespaceSelectors:
    - matchExpressions:
        - key: blue.m2mgate.cloud/project-slug
          operator: Exists
  externalSecretSpec:
    secretStoreRef:
      name: shared-secrets-store
      kind: ClusterSecretStore
    data:
      - secretKey: gitlab-blue-token
        remoteRef:
          key: gitlab-blue-token
          property: gitlab-blue-token
      - secretKey: .tofurc
        remoteRef:
          key: tofurc
          property: .tofurc
```

Listing 6: Replicate secrets in managed clusters namespaces (2/2).

```yaml
apiVersion: external-secrets.io/v1
kind: SecretStore
metadata:
  name: secret-gitlab-store
  namespace: example-project
spec:
  provider:
    gitlab:
      projectID: "123"
      groupIDs:
        - "572" # group ID of the Blue group
      url: "https://gitlab.example.com"
      auth:
        SecretRef:
          accessToken:
            name: "shared-secrets"
            key: "gitlab-blue-token"
---
apiVersion: external-secrets.io/v1
kind: ExternalSecret
metadata:
  name: secret-external
  namespace: example-project
spec:
  secretStoreRef:
    name: secret-gitlab-store
    kind: SecretStore
  data:
    - secretKey: HCLOUD_TOKEN
      remoteRef:
        key: HCLOUD_TOKEN
    # ... other secrets and target field omitted for brevity
```

Listing 7: Example of SecretStore and ExternalSecret resources for the secrets in GitLab.

Crossplane

Crossplane is gaining momentum in the cloud-native landscape, recently promoted to CNCF Graduated status in November 2025 [22]. It is central to M2MGate provisioning automation, unifying control of external APIs with custom API abstraction creation. This section presents how Crossplane creates a high-level API describing M2MGate as a single entity encompassing infrastructure and software components, and how it translates this entity into actual environments.

Crossplane capabilities are extended using its CRD set:

- **Provider** - Defines a new controller and CRD set enabling external API control. For example, `provider-kubernetes` creates Kubernetes resources in different clusters by defining `Object` kind resources. This thesis employs:
  - `provider-kubernetes`: Deploys and manages arbitrary Kubernetes objects on provisioned clusters.
  - `provider-opentofu`: Runs Terraform code (HCL).
  - `provider-helm`: Deploys and manages Helm Releases on provisioned Kubernetes clusters.
- **CompositeResourceDefinition** - Extends the Kubernetes API with new resource types. Objects of newly created types are called CompositeResources (XRs) in Crossplane terminology. Listing 8 shows the M2MGateCluster XRD.
- **Composition** - Expresses how Crossplane should process an XR through a set of steps. Each step uses a Function receiving the XR, additional user input, and previous step output, producing Kubernetes resources as output. Composition output consists of Provider CRDs creating resources external to the control plane cluster. Listing 9 shows the M2MGateCluster Composition. When a Composition definition changes, Crossplane creates a CompositionRevision, an immutable Composition snapshot. CompositeResources can select specific CompositionRevisions via the XR's `compositionRevisionSelector` field, matching the `version` and `channel` labels from the Composition at CompositionRevision snapshot creation time. This enables smooth XR upgrade paths by matching more recent CompositionRevisions by label when available.
- **Function** - Implements a Composition step. When created, a pod responsible for function execution is created in the control plane and begins listening for inputs. The Crossplane community has implemented Functions creating and manipulating Kubernetes resources using various programming languages. The M2MGateCluster XR Composition uses:
  - `function-kcl`: Creates Kubernetes resources using KCL.
  - `function-auto-ready`: Detects resource readiness.
  - `function-sequencer`: Expresses resource creation order.

```yaml
apiVersion: apiextensions.crossplane.io/v1
kind: CompositeResourceDefinition
metadata:
  name: m2mgateclusters.blue.m2mgate.cloud
spec:
  group: blue.m2mgate.cloud
  names:
    kind: M2MGateCluster
    plural: m2mgateclusters
  scope: Namespaced
  versions:
    - name: v1alpha1
      referenceable: true
      schema:
        openAPIV3Schema:
          description: M2MGateCluster is the Schema for the M2MGateCluster API.
          properties:
            spec:
              description: M2MGateClusterSpec defines the desired state of M2MGateCluster.
              properties:
                baseDomain:
                  description: The base domain for the M2MGate cluster (e.g.,
                  ↪  example.m2mgate.cloud)
                  type: string
                auth:
                  description: Authentication configuration
                  properties:
                    enableInsideM2MAzureAD:
                      description: Enables Single Sign-On with INSIDE M2M Azure AD
                      type: boolean
                  type: object
                # other properties omitted for brevity
```

Listing 8: Snippet of XRD for the M2MGateCluster resource definition.

```yaml
apiVersion: apiextensions.crossplane.io/v1
kind: Composition
metadata:
  name: m2mgateclusters.blue.m2mgate.cloud
  labels:
    version: v0.0.66
    channel: stable
spec:
  compositeTypeRef:
    apiVersion: blue.m2mgate.cloud/v1alpha1
    kind: M2MGateCluster
  mode: Pipeline
  pipeline:
  - step: render-resources
    functionRef:
      name: crossplane-contrib-function-kcl
    credentials:
    # credentials omitted for brevity
    input:
      apiVersion: krm.kcl.dev/v1alpha1
      kind: KCLInput
      spec:
        source: oci://registry.example.com/blue/blue-platform/kcl/modules/m2mgatecluster?tag=v
        ↪  0.0.66
  # other steps omitted for brevity
```

Listing 9: Snippet of Composition for the M2MGateCluster XR.

A set of CompositeResourceDefinitions, related Compositions, and all Providers and Functions implementing them can be packaged as a single artifact called a Configuration, assembled and distributed as an OCI Distribution. Listing 10 shows the Crossplane Configuration definition for the control plane cluster.

```yaml
apiVersion: pkg.crossplane.io/v1
kind: Configuration
metadata:
  name: blue-crossplane-configuration
spec:
  package: registry.example.com/blue/blue-platform/control-plane/k8s/crossplane/configuration:
  ↪ v0.0.66
  revisionHistoryLimit: 100
```

Listing 10: Snippet of Configuration for Crossplane.

4.2 Migration to Kubernetes

Implementing the new solution required migrating existing M2MGate Blueprint Docker Compose definitions to Kubernetes manifests. This process automated a significant portion of previously manual operations, improved system observability, and increased node failure resilience.

As mentioned in Section 3.2.1, Kubernetes resources are rendered from KCL code rather than directly expressed in YAML. However, for clarity, listings in this section are presented in YAML format. Additionally, Kubernetes resources are not directly created in managed clusters but rather in the control plane, wrapped in Crossplane Provider API resources (`provider-kubernetes, provider-opentofu, provider-helm`). Again for clarity, examples present actual resources created in managed clusters whenever possible.

This section presents the migration process, emphasizing system changes.

4.2.1 Cluster Overview

Kubernetes distribution

The platform currently supports only Hetzner as the cloud provider for managed clusters. The same OpenTofu module used for the control plane is reused with minor modifications to provision infrastructure supporting M2MGate installations. Future platform iterations may support other cloud providers, possibly offering Kubernetes as a service. K3s is currently used as the Kubernetes distribution for managed clusters via the `kube-hetzner` module. K3s is a lightweight Kubernetes distribution designed for simple installation, configuration, and operation [23], simplifying setup by bundling features including:

- containerd (CRI)
- Flannel (CNI)

- CoreDNS (cluster DNS)
- Traefik, used also in M2MGate Blueprint (ingress controller)
- Helm controller

Additionally, `kube-hetzner` (Section 4.1.1) enables configuration of useful controllers and tools for managed clusters:

- **Hetzner Cloud Controller Manager** - Manages the Hetzner Cloud API and provides Hetzner Cloud-specific Kubernetes cluster functionality, including creating and managing Hetzner Cloud Load Balancers via Kubernetes resources.
- **Hetzner CSI Driver** - Provides Hetzner Cloud-specific storage functionality to the Kubernetes cluster.
- **Kured** - Manages node reboots post system updates.
- **Rancher System Upgrade Controller** - Initiates Kubernetes and OS updates.
- **Cluster Autoscaler** - Manages Kubernetes cluster autoscaling functionality.

These controllers provide a more resilient and scalable system compared to M2MGate Blueprint. The setup resembles a managed Kubernetes cluster experience. Pods can be scheduled on different nodes as the Hetzner CSI provides necessary storage functionality without binding containers to specific nodes, and the Cluster Autoscaler ensures automatic cluster scaling based on load. Kured and the Rancher System Upgrade Controller ensure nodes are updated and rebooted safely and automatically, easing operational burden for system administrators.

Listing 11 shows how the `provider-opentofu` Crossplane Provider applies the `kube-hetzner` module. OCI Registry credentials, the `terraform` block, and provider declarations are encapsulated in a ProviderConfig resource from the Crossplane API. Module outputs, such as the newly created cluster's `kubeconfig`, are stored as a "connection secret" Secret resource in the control plane cluster.

```yaml
apiVersion: opentofu.m.upbound.io/v1beta1
kind: Workspace
metadata:
  name: bootstrap-infra-cluster
  namespace: example-m2mgate-cloud
spec:
  forProvider:
    env:
    - name: TF_VAR_hcloud_token
      secretKeyRef:
        key: HCLOUD_TOKEN
        name: secret-external
    # other environment variables omitted for brevity
    module: |

      variable "hcloud_token" {
        description = "Hetzner Cloud API token."
        type        = string
        sensitive   = true
      }

      # other variables omitted for brevity

      output "kubeconfig" {
        description = "Configuration for kubectl"
        value       = module.kube-hetzner.kubeconfig
        sensitive   = true
      }

      module "kube-hetzner" {
        source = "oci://registry.example.com/blue/blue-platform/opentofu/modules/kube-hetzner?
        ↪  tag=v0.0.66"
        hcloud_token = var.hcloud_token
        ssh_private_key = var.ssh_private_key
        registry_robot_password = var.registry_robot_password
        agent_server_type = var.agent_server_type
        base_domain = var.base_domain
        providers = {
          hcloud = hcloud
        }
      }
    source: Inline
    vars:
    # other variables omitted for brevity
  providerConfigRef:
    kind: ProviderConfig
    name: bootstrap-infra-provider-config
  writeConnectionSecretToRef:
    name: bootstrap-infra-connection-secret
```

Listing 11: Opentofu module applied by provider-opentofu.

Storage controllers

Application storage is provided via the Hetzner CSI Driver and NFS CSI Driver:

1. The Hetzner CSI Driver is a Container Storage Interface driver for Hetzner Cloud enabling ReadWriteOnce volumes in Kubernetes [24]. PersistentVolumes backed by Hetzner Cloud Volumes allow pods to be scheduled on different nodes while maintaining access to consistent data, essential for high availability. The Blueprint approach bound containers to specific nodes with mounted local volumes, reducing flexibility and failover potential.

2. ReadWriteMany (RWX) volumes are provided via the NFS CSI Driver, requiring an existing, configured NFSv3 or NFSv4 server. It supports dynamic PersistentVolume provisioning via PersistentVolumeClaims by creating subdirectories under the NFS server [25]. RWX volume capability is required as some M2MGate services share data via filesystem. Previously, this was implemented at node level by mounting container volumes in GlusterFS volumes shared across nodes. Migration to NFS was motivated by GlusterFS loss of momentum and lack of active Red Hat maintenance [26]. Additionally, the NFS CSI Driver simplifies node setup as the NFS server can be a Kubernetes pod within the cluster.

Future platform iterations supporting other cloud providers may simplify the system by relying on native cloud storage solutions.

Listing 12 shows the Release resource created in the control plane cluster for the NFS CSI Driver. The `provider-helm` Crossplane Provider consumes this definition to deploy the NFS CSI Driver via Helm in the managed cluster. Managed cluster API credentials are encapsulated in a ProviderConfig resource.

```yaml
apiVersion: helm.m.crossplane.io/v1beta1
kind: Release
metadata:
  name: bootstrap-k8s-controller-storage-csi-driver-nfs
  namespace: example-m2mgate-cloud
spec:
  forProvider:
    chart:
      name: csi-driver-nfs
      repository: >-
        https://raw.githubusercontent.com/kubernetes-csi/csi-driver-nfs/master/charts
      version: 4.12.0
    namespace: kube-system
    skipCreateNamespace: true
  providerConfigRef:
    kind: ProviderConfig
    name: bootstrap-k8s-provider-config-helm
```

Listing 12: NFS CSI Driver Release resource.

Listings 13, 14, and 15 show PersistentVolumeClaim, Deployment, and StorageClass resources created in the managed cluster for the NFSv4 server. The Service resource is omitted for brevity.

```yaml
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: nfs-server
  namespace: kube-system
spec:
  accessModes:
  - ReadWriteOnce
  resources:
    # ...
  storageClassName: hcloud-volumes
```

Listing 13: NFSv4 server PersistentVolumeClaim resource created in the managed cluster.

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  # ...
spec:
  # ...
  template:
    spec:
      containers:
      - env:
        - name: SHARED_DIRECTORY
          value: /exports
        image: itsthenetwork/nfs-server-alpine:12
        name: nfs-server
        # ports and resources omitted for brevity
        securityContext:
          privileged: true
        volumeMounts:
        - mountPath: /exports
          name: exports
      volumes:
      - name: exports
        persistentVolumeClaim:
          claimName: nfs-server
```

Listing 14: NFSv4 server Deployment resource created in the managed cluster.

```yaml
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  # omitted for brevity
mountOptions:
- nfsvers=4.1
parameters:
  server: nfs-server.kube-system.svc.cluster.local
  share: /
provisioner: nfs.csi.k8s.io
reclaimPolicy: Delete
volumeBindingMode: WaitForFirstConsumer
allowVolumeExpansion: true
```

Listing 15: NFSv4 server StorageClass resource created in the managed cluster.

DNS configuration

A private Hetzner Cloud DNS zone was created to ensure proper DNS resolution for all managed cluster services, managed using the External DNS Operator and Hetzner cert-manager Webhook.

External DNS operator External DNS operator is a Kubernetes operator retrieving resource lists (Services, Ingresses, etc.) from the Kubernetes API to determine desired DNS records for creation [27], in this case via the Hetzner Console API.

Section 4.2.1 described how Hetzner Cloud Controller Manager creates Hetzner Cloud Load Balancers via Kubernetes resources, specifically `LoadBalancer` type Service resources. When a LoadBalancer is created, it receives a public IP address unknown beforehand. The External DNS Operator is therefore configured to create a corresponding DNS record providing the service with a static name.

The External DNS Operator creates the following records for M2MGate stack applications:

1. `A` record for Cascade Service external IP address
2. `A` record for VPNGate Service external IP address
3. `A` record for Traefik Service external IP address binding the base domain name (e.g. `example.m2mgate.cloud`)
4. `A` record for Traefik Service external IP address binding the wildcard domain name (e.g. `*.example.m2mgate.cloud`)

Listing 16 shows External DNS Operator configuration. Hetzner Cloud API credentials are encapsulated in a Secret resource. Hetzner API integration is achieved via a Hetzner-specific webhook. Listing 17 shows how Kubernetes Services can be annotated for External DNS Operator management.

Hetzner cert-manager webhook Hetzner cert-manager webhook creates necessary `TXT` DNS records via the Hetzner API to solve DNS01 challenges for ACME-type cert-manager Issuers [28], automating Traefik wildcard certificate issuance.

Identity Provider

For authentication, the cluster uses Azure Entra ID as the identity provider, enabling single sign-on (SSO) for internal users across all stack services. An Azure Entra ID application registration is created for each new environment, providing credentials for Keycloak to authenticate users. The application registration is created using the OpenTofu module in Listing 19 via the OpenTofu Provider. This implementation choice stems from a late decision to adopt Crossplane for the control plane, after the Azure Entra ID application registration had already been implemented with OpenTofu. Future work includes migrating to the `provider-azuread` Crossplane native Provider.

```yaml
apiVersion: helm.m.crossplane.io/v1beta1
kind: Release
metadata:
  # omitted for brevity
spec:
  forProvider:
    chart:
      name: external-dns
      repository: https://kubernetes-sigs.github.io/external-dns
      version: 1.19.0
    namespace: kube-system
    values:
      policy: sync
      provider:
        name: webhook
        webhook:
          env:
          - name: HETZNER_TOKEN
            valueFrom:
              secretKeyRef:
                key: HETZNERDNS_TOKEN
                name: hetznerdns-token
          image:
            repository: docker.io/hetzner/external-dns-hetzner-webhook
            tag: v0.3.0
      txtOwnerId: example-m2mgate-cloud
  # other fields omitted for brevity
```

Listing 16: External DNS Operator configuration.

```yaml
apiVersion: v1
kind: Service
metadata:
  annotations:
    external-dns.alpha.kubernetes.io/hostname: "*.example.m2mgate.cloud,example.m2mgate.cloud"
  name: traefik
  namespace: traefik
spec:
  type: LoadBalancer
  # other fields omitted for brevity
```

Listing 17: Kubernetes Service with External DNS Operator annotation.

```yaml
apiVersion: helm.m.crossplane.io/v1beta1
kind: Release
metadata:
  # omitted for brevity
spec:
  forProvider:
    chart:
      name: cert-manager-webhook-hetzner
      repository: https://charts.hetzner.cloud
      version: 0.5.0
    namespace: cert-manager
    # other fields omitted for brevity
```

Listing 18: Hetzner cert-manager webhook Release resource.

```
resource "azuread_application_registration" "application" {
  display_name            = var.base_domain
  description             =
  ↪  "Application registration for [${var.base_domain}] OIDC authentication."
  sign_in_audience        = "AzureADMyOrg"
  group_membership_claims = ["All"]
}

resource "time_static" "client_secret_created" {
}

resource "azuread_application_password" "client_secret" {
  application_id = azuread_application_registration.application.id
  display_name   = "Client Secret [${var.base_domain}]"
  end_date       = timeadd(time_static.client_secret_created.rfc3339, "17520h") // 2 years from
  ↪  creation
}

resource "azuread_application_redirect_uris" "redirect_uris" {
  application_id = azuread_application_registration.application.id
  type           = "Web"

  redirect_uris = [
    "https://auth.${var.base_domain}/realms/master/broker/azure/endpoint",
    "https://auth.${var.base_domain}/realms/infrastructure/broker/azure/endpoint",
    "https://auth.${var.base_domain}/realms/m2mgate/broker/azure/endpoint"
  ]
}

resource "azuread_application_optional_claims" "groups_claim" {
  application_id = azuread_application_registration.application.id

  access_token {
    name = "groups"
  }

  id_token {
    name = "groups"
  }

  saml2_token {
    name = "groups"
  }
}
```

Listing 19: Azure Entra ID application registration

4.2.2 Third Party Components

Helm charts and various operators have been leveraged for the deployment of 3rd party services. The following sections present a high-level overview of the most relevant ones.

Kafka

Kafka deployment was handled via the open-source Strimzi project, which simplifies Kafka cluster management through operators managing cluster lifecycle, topics, and users [29]. A 3-node Kafka cluster was deployed in KRaft mode using simple Strimzi CRD abstractions.

MySQL

M2MGate Blueprint relied on two MySQL instances: one primary and one replica. Both instances were manually configured for primary-replica asynchronous replication, with backups using Percona XtraBackup. Primary failure required manual switchover from replica to primary. The new solution deploys a 3-node MySQL cluster using the Percona Operator for MySQL, which automatically handles setup, failover, and backups, providing third-party integrations including certificate management via cert-manager and HAProxy for load balancing [30].

Monitoring

A primary focus for the new solution is improving overall system observability. Prometheus and Grafana, already present in the Blueprint, are deployed through the `kube-prometheus-stack` [31] Helm chart including:

- Prometheus Operator
- Prometheus
- Alertmanager
- Prometheus node-exporter
- Prometheus blackbox-exporter (not used)
- Prometheus Adapter for Kubernetes Metrics APIs
- kube-state-metrics
- Grafana
- Configuration for metrics scraping of Kubernetes components
- A pre-configured set of alerts and dashboards in Grafana

The new Kubernetes setup includes numerous Grafana dashboards and Prometheus rules (alerts) for system monitoring:

- Dashboard count increased from 10 to 46, covering cluster health, node performance, pod metrics, and third-party service application-specific metrics.
- Manual Grafana Alerts and `monit` alerts were replaced by 260 Prometheus rules triggering alerts handled by Alertmanager.

The new alert and dashboard set was sourced from the Prometheus Monitoring Mixin project [32]. Email notifications are configured by default for warning and error alerts, with Microsoft Teams webhook notifications for all alert types.

The log collection system underwent significant changes with Grafana Alloy replacing Logstash. Loki replaced OpenSearch as the logging backend, and Grafana

deprecated OpenSearch Dashboards, now serving as both a metrics dashboarding tool and log management interface. Alloy and Loki were deployed using the `k8s-monitoring` and `loki` Helm charts, both maintained by Grafana Labs.

```yaml
apiVersion: monitoring.coreos.com/v1
kind: PrometheusRule
metadata:
  name: mysql-rules
  namespace: monitoring-system
spec:
  groups:
    - name: mysql.rules
      rules:
        - record: job:mysql_transactions:rate5m
          expr: sum without (command)
          ↪  (rate(mysql_global_status_commands_total{command=~"(commit|rollback)"}[5m]))
    - name: mysql.alerts
      rules:
        - alert: MySQLDown
          expr: mysql_up != 1
          for: 5m
          labels:
            severity: critical
          annotations:
            description: "MySQL {{$labels.job}} on {{$labels.instance}} is not up."
            summary: MySQL not up.
    - name: mysql.galera.alerts
      rules:
        - alert: MySQLGaleraNotReady
          expr: mysql_global_status_wsrep_ready != 1
          for: 5m
          labels:
            severity: warning
          annotations:
            description: "{{$labels.job}} on {{$labels.instance}} is not ready."
            summary: Galera cluster node not ready.
        # other rules omitted for brevity
```

Listing 20: PrometheusRule example with alerts for for MySQL.

4.2.3 Automation

This section presents implementation choices for automating formerly manual post-deployment processes, significantly achieved through Kubernetes patterns (Init Container, Operator, Batch Job) [33].

Create users and databases in MySQL

This task is automated using a Percona Operator for MySQL feature enabling user and database creation via the PerconaXtraDBCluster CRD. Listing 21 shows the PerconaXtraDBCluster resource definition for user and database creation.

```yaml
apiVersion: pxc.percona.com/v1
kind: PerconaXtraDBCluster
metadata:
  # omitted for brevity
spec:
  users:
  - dbs:
    - m2mgate_core
    - m2mgate_messaging
    - m2mgate_geocoder
    - m2mgate_distribution
    - m2mgate_mad
    - m2mgate_tenant
    - m2mgate_vpngate
    grants:
    - ALL
    name: m2mgate
    passwordSecretRef:
      key: m2mgate
      name: mysql-cluster-credentials
    withGrantOption: false
  - dbs:
    - keycloak
    grants:
    - ALL
    name: keycloak
    passwordSecretRef:
      key: keycloak
      name: mysql-cluster-credentials
    withGrantOption: false
  # other fields omitted for brevity
```

Listing 21: PerconaXtraDBCluster resource definition for the creation of the users and databases.

Create realms in Keycloak

Keycloak realm creation uses the official Keycloak Operator, which is in charge of-monitoring `KeycloakRealmImport` resources creation and applying them to running Keycloak instances. Two separate realms were created:

- `infrastructure` - For internal users, containing client applications required by system administrators.
- `m2mgate` - For internal and external users, containing the M2MGate client application and customer-specific clients.

Both realms can use Azure Entra ID as their identity provider with mapping rules converting identity provider claims to realm roles controlling resource access. All realm configuration secrets, including Azure Entra client IDs and secrets, are stored in a separate Kubernetes `Secret` resource.

M2MGate Tenant Service interacts with the Keycloak API to subdivide users into different tenancy scopes, assigning groups and roles. Tenant Service formerly authenticated against Keycloak using OAuth2 authorization code grant type, requiring a pre-configured Keycloak user with password and sufficient privileges. This approach was deemed inadequate, jeopardizing realm crafting process ease for the following reasons:

1. Realm user export is more complex than client and role export, requiring running the Keycloak executable `kc.sh` from a different pod than the one containing the Keycloak instance to export.

2. User passwords are stored as salted hashes present in realm exports. Crafting reproducible exports requires computing these hashes.

Therefore, this step included refactoring M2MGate Tenant Service authentication to use client credentials grant type.

```yaml
apiVersion: k8s.keycloak.org/v2alpha1
kind: KeycloakRealmImport
metadata:
  name: infrastructure-realm
  namespace: keycloak
spec:
  keycloakCRName: keycloak
  placeholders:
    smtp:
      secret:
        name: infrastructure-realm-secrets
        key: smtp
    grafana:
      secret:
        name: infrastructure-realm-secrets
        key: grafana
    azuread-client-id:
      secret:
        name: infrastructure-realm-secrets
        key: azuread-client-id
    azuread-client-secret:
      secret:
        name: infrastructure-realm-secrets
        key: azuread-client-secret
    # more placeholders omitted for brevity
  realm:
    id: 2d10424a-993c-44b2-a57e-458e15386094
    realm: infrastructure
    # rest of the configuration
```

Listing 22: KeycloakRealmImport definition for the infrastructure realm.

Generation and upload in M2MGate services volumes of TLS material

M2MGate supports various remote device types, including legacy, less performant devices, and can be configured with different authentication mechanisms for its communication protocol. The same protocol is used between backend services and between the M2MGate CLI tool and backend. The most common authentication scheme is mutual TLS, where server and client exchange certificates during the handshake phase. This mechanism requires generating and distributing keys and certificates as shown in Table 4.1.

File	Cascade Service	Core Service	Generic Backend Service	M2MGate CLI
CA.key		✓		
CA.crt	✓	✓	✓	✓
cascade.key	✓			
cascade.crt	✓			
service.key		✓	✓	
service.crt		✓	✓	
cli.key				✓
cli.crt				✓

Table 4.1: TLS keys and certificates in the M2MGate system.

Core Service holds the CA certificate (trusted root authority) for signing Certificate Sign Requests (CSRs) from remote devices. Prior provisioning included out-of-band key and certificate generation and manual distribution via Secure Copy Protocol (SCP). Certificate expiration or key compromise required manual regeneration.

The new approach uses `cert-manager` to automate certificate creation, issuance, and renewal, distributing material by placing it in Kubernetes secrets mountable in each M2MGate service pod. M2MGate CLI private and public keys are also generated this way and must be retrieved from the generated secret for CLI use.

Applications consume private and public keys as PKCS12 keystore files. The `SocketAttachmentFactoryImpl` class handling truststore (CA certificate) and keystore (private key and public certificates) loading previously expected a single file containing both elements. The code was refactored to consume two distinct files, following `cert-manager` best practices. Listings 23 and 24 show CRD definitions for the CA `Issuer` and `Certificate`, and how the produced keystore is mounted in Core Service.

```yaml
apiVersion: cert-manager.io/v1
kind: Issuer
metadata:
  name: m2mgate-ca-issuer
  namespace: m2mgate
spec:
  selfSigned: {}
---
apiVersion: cert-manager.io/v1
kind: Certificate
metadata:
  name: m2mgate-ca-certificate
  namespace: m2mgate
spec:
  # details omitted for brevity
  keystores:
    pkcs12:
      create: true
      password: password
  secretName: m2mgate-ca-certificate
```

Listing 23: M2MGate CA Issuer and Certificate.

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: core-service
  namespace: m2mgate
spec:
  # ...
  template:
    # ...
    spec:
      containers:
      - env:
        # other environment variables omitted for brevity
        - name: openssl.config.caP12Pass
          value: password
        image: registry.example.com/m2mgate-sage/sage-core-service:5.13.0
        name: core-service
        # other container spec omitted for brevity
        volumeMounts:
        # other volume mounts omitted for brevity
        - mountPath: /opt/csr-ca
          name: m2mgate-csr-ca
      volumes:
      # other volumes omitted for brevity
      - name: m2mgate-csr-ca
        secret:
          items:
          - key: keystore.p12
            path: keystore.p12
          secretName: m2mgate-ca-certificate
```

Listing 24: Core Service deployment with mounted certificate secrets.

Generation of wildcard certificate for Traefik

Wildcard certificate generation for services behind the Traefik ingress controller uses `cert-manager` and the Hetzner cert-manager webhook. Using the `cert-manager`

API (e.g., ACME-type `Issuer`), a client can verify domain ownership with the Let's Encrypt CA using the ACME protocol to request a certificate. As introduced in Section 4.2.1, the Hetzner cert-manager webhook solves the DNS01 challenge by creating a `TXT` record in the zone for which ownership must be proven, using the Hetzner API. The certificate is then issued by the Let's Encrypt CA and stored in a Kubernetes secret.

```yaml
apiVersion: cert-manager.io/v1
kind: Issuer
metadata:
  name: wildcard-issuer
  namespace: traefik
spec:
  acme:
    email: example@example.com
    privateKeySecretRef:
      name: wildcard-issuer
    server: https://acme-v02.api.letsencrypt.org/directory
    solvers:
    - dns01:
        webhook:
          config:
            tokenSecretKeyRef:
              key: HETZNERDNS_TOKEN
              name: hetznerdns-token
          groupName: acme.hetzner.com
          solverName: hetzner
---
apiVersion: cert-manager.io/v1
kind: Certificate
metadata:
  name: wildcard-certificate
  namespace: traefik
spec:
  dnsNames:
  - example.com
  - "*.example.com"
  issuerRef:
    kind: Issuer
    name: wildcard-issuer
  secretName: wildcard-certificate
  subject:
    organizations:
    - inside-m2m

```

Listing 25: Traefik wildcard certificate Issuer and Certificate resources.

Upload custom Keycloak theme in container volume

Default Keycloak themes do not comply with M2MGate branding requirements, necessitating custom theme development. Keycloak is deployed via the Keycloak Operator. The `Keycloak` CRD offers "unsupported" configuration options allowing direct PodTemplateSpec modifications for the Keycloak pod, enabling volume definition for theme files. Listing 26 shows the Keycloak resource definition for deployment. In a previous design iteration, theme files were fetched from a Git repository using an init container. However, this approach required managed cluster access to the GitLab server. The GitLab server reverse proxy enforces a non-API-configurable

IP whitelist; initially, each new environment required a VPN tunnel configured via Terraform. This solution was deemed unsuitable for security reasons. The current solution offers a helper script, implemented as a platform CLI command, to manually upload theme files to the Keycloak instance from the local filesystem. Future work may explore introducing a CI pipeline in the Keycloak theme project to build OCI artifacts from theme files and push them to the OCI Registry for managed cluster accessibility.

```yaml
apiVersion: k8s.keycloak.org/v2alpha1
kind: Keycloak
metadata:
  name: keycloak
  namespace: keycloak
spec:
  # other fields omitted for brevity
  unsupported:
    podTemplate:
      spec:
        containers:
        - volumeMounts:
          - mountPath: /opt/keycloak/themes
            name: keycloak-themes
        initContainers:
        - args:
          - "-c"
          - chown -R 1000:0 /opt/keycloak/themes
          command:
          - /bin/sh
          image: alpine
          name: keycloak-init
          volumeMounts:
          - mountPath: /opt/keycloak/themes
            name: keycloak-themes
        volumes:
        - name: keycloak-themes
          persistentVolumeClaim:
            claimName: keycloak-themes
```

Listing 26: Keycloak resource definition for the deployment of Keycloak.

4.3 Platform API Definition

This section highlights platform API definition and implementation details, specifically presenting the M2MGateCluster CompositeResourceDefinition and related Composition in the control plane cluster and how these resources support environment creation, update, and deletion.

4.3.1 M2MGateCluster CompositeResourceDefinition

As mentioned in Section 4.1.2, Crossplane CompositeResourceDefinitions (XRDs) extend the Kubernetes API with new resource types. This mechanism defines an abstraction for `M2MGateCluster` as a single entity encompassing both infrastructure and software components of an M2MGate installation. Listing 27 shows a minimal

example. When created in the control plane cluster, this resource definition triggers execution of the CompositionRevision matching `channel` and `version` labels, creating the managed cluster and its applications using default values.

```yaml
apiVersion: blue.m2mgate.cloud/v1alpha1
kind: M2MGateCluster
metadata:
  name: m2mgate-cluster
  namespace: example-m2mgate-cloud
  annotations:
    blue.m2mgate.cloud/project-id: "570"
    blue.m2mgate.cloud/project-slug: example-m2mgate-cloud
spec:
  baseDomain: example.m2mgate.cloud
  crossplane:
    compositionRevisionSelector:
      matchLabels:
        channel: stable
        version: v0.0.66
  provider: hetzner
```

Listing 27: M2MGateCluster CompositeResourceDefinition.

Tables B.1, B.2, B.3, B.4, B.5, and B.6 in Appendix B overview available configuration options for the M2MGateCluster resource definition's `spec` field. Capability selection for the abstraction was driven by M2MGate product experience, survey results (Appendix A), and meetings with the M2MGate CORE team. This process led to API design covering the most common M2MGate installation configuration requirements, with expected future extension to support more complex scenarios. The platform API can be extended for any automation task, not only M2MGate environment creation; future CompositeResourceDefinitions may support other use cases (e.g., project repository creation with company best practice scaffolding, plain cluster creation for non-M2MGate applications, etc.).

M2MGateCluster nested CompositeResourceDefinitions

The M2MGateCluster resource encapsulates, in its abstraction, a vast control plane resource set. Generating such a large resource set is uncommon for a CompositeResource; therefore, additional CompositeResourceDefinitions were created and nested in the M2MGateCluster definition. The M2MGateCluster Composition output is a set of CompositeResources processed by secondary Compositions, each responsible for creating a specific resource set.

This mechanism enables resource creation modularity in the Composition and avoids a hard Crossplane limitation encountered during development, forcing pipeline step communication messages to be smaller than 4MB.

The following nested CompositeResourceDefinitions were created:

- `M2MGateClusterSecret`: Creates SecretStore and ExternalSecret resources for GitLab secrets.
- `M2MGateClusterBootstrap`: Creates the Kubernetes cluster and installs various controllers and tools.

- `M2MGateClusterDatabase`: Creates the MySQL cluster and related resources.
- `M2MGateClusterBroker`: Creates the Kafka cluster and related resources.
- `M2MGateClusterAuth`: Creates the Keycloak instance, its realms, and related resources.
- `M2MGateClusterMonitoring`: Creates the monitoring stack and related resources.
- `M2MGateClusterService`: Creates M2MGate services and related resources.

4.3.2 M2MGateCluster Composition

A Composition describes Crossplane actions for processing a CompositeResource. Section 4.1.2 introduced the basic concept: the CompositeResource is piped through steps, each taking previous step output and additional user input as input, until the final result—a Kubernetes resource set—is applied to the control plane cluster. These output resources are mostly Crossplane Provider-defined CRDs creating external, or "managed" resources (in Crossplane terminology) exploiting external APIs. This section describes in detail steps converting an M2MGateCluster resource into a fully functional environment.

Figure C.1 in Appendix C shows the resource tree derived from an M2MGateCluster CompositeResource using default values after Composition application.

Every Composition, both primary and nested, follows the same pattern:

1. Resources that must derive from the CompositeResource are rendered from KCL code via the `function-kcl` Function.
2. If required, resulting resource creation is delayed to follow the sequence defined using the `function-sequencer` Function, ensuring correct creation order.
3. Resources are reported ready to the Crossplane Controller when their `Ready` condition is set to `True` via the `function-auto-ready` Function.

Function KCL

This section expands on the `function-kcl` Function used to render resources from KCL code. The function implementation is a Go application compliant with the KCL KRM (Kubernetes Resource Model) specification [34] [35] offering rich features:

- Pulling KCL modules from OCI registries, Git repositories, or utilizing code written inline in the Composition definition.
- Collecting information from the Kubernetes API for use in KCL code (e.g., CompositeResources and other Kubernetes resources).
- Writing information back to the Kubernetes API (e.g., resulting resource status fields).

The majority of the platform codebase consists of KCL code defining logic governing necessary control plane resource rendering for M2MGate environment creation. The following modules support M2MGateCluster CompositeResource creation:

- `m2mgatecluster`: Main module creating nested CompositeResources from the M2MGateCluster CompositeResource.
- `m2mgateclustersecret`: Defines secret management resources.
- `m2mgateclusterbootstrap`: Defines Kubernetes cluster bootstrap resources.
- `m2mgateclusterdatabase`: Defines MySQL cluster resources.
- `m2mgateclusterbroker`: Defines Kafka cluster resources.
- `m2mgateclusterauth`: Defines Keycloak instance resources.
- `m2mgateclustermonitoring`: Defines monitoring stack resources.
- `m2mgateclusterservice`: Defines M2MGate service resources.
- `common`: Contains cross-module schemas (M2MGateCluster resource schema, Deployment schema, PerconaXtraDBCluster schema, etc.), utility functions, and global variables.

Each module is packaged as a separate OCI artifact and published to the OCI Registry via GitLab CI pipeline. The `function-kcl` Function then pulls these modules for Composition execution. Listing 28 shows a KCL module being pulled from the OCI Registry.

Listing 29 shows KCL code generating `Usage` type resources. `Usage` is a Crossplane CRD expressing resource dependencies. Defining such dependencies when rendering a Composition is crucial; otherwise, CompositeResource deletion may delete resources in incorrect order, leaving dangling control plane cluster objects or worse, leaking external resources (e.g., undeleted cloud provider servers). The example demonstrates KCL code convenience for manifest generation: numerous resources can be expressed concisely while remaining readable and maintainable. Here, a `Usage` is created for each `broker.resources` folder object, expressing namespace dependency. The code creates resources using a schema imported from the `common` module, catching errors at compile time and making the code more concise.

```yaml
apiVersion: apiextensions.crossplane.io/v1
kind: Composition
metadata:
  # ...
spec:
  compositeTypeRef:
    apiVersion: blue.m2mgate.cloud/v1alpha1
    kind: M2MGateCluster
  mode: Pipeline
  pipeline:
  - step: render-resources
    functionRef:
      name: crossplane-contrib-function-kcl
    credentials:
    - name: kcl-registry
      source: Secret
      secretRef:
        namespace: crossplane-system
        name: oci-registry-credentials
    input:
      apiVersion: krm.kcl.dev/v1alpha1
      kind: KCLInput
      spec:
        source:
        ↪  oci://registry.m2mgate.de/blue/blue-platform/kcl/modules/m2mgatecluster?tag=v0.0.66
```

Listing 28: Function KCL usage in the Composition definition.

```
import broker.resources
import common.utils
import common.models.io.crossplane.protection.v1beta1 as protectionv1beta1

_usage_namespace = lambda by: any -> protectionv1beta1.Usage {
    protectionv1beta1.Usage {
        metadata = utils._metadata("broker-usage-ns-" + by.metadata.name)
        spec = {
            replayDeletion = True
            of = {
                apiVersion = "kubernetes.m.crossplane.io/v1alpha1"
                kind = "Object"
                resourceRef = {
                    name = "bootstrap-k8s-namespace-kafka"
                }
            }
            by = {
                apiVersion = by.apiVersion
                kind = by.kind
                resourceRef = {
                    name = by.metadata.name
                }
            }
        }
    }
}

_items = [
    *[_usage_namespace(item) for item in resources._items]
]
```

Listing 29: Snippet of KCL code used to generate resources of type `Usage` in `m2mgateclusterbroker` module.

Function Sequencer

Composition-created resources must be uniquely identified by their names, the `function-sequencer` Function takes YAML-expressed sequences of such names as input, ordering resource creation accordingly. Regular expression patterns can match multiple resources. Listing 30 shows a sequence for the M2MGateCluster Composition. This example shows how resources belonging to:

1. `cluster-secret.*`,
2. `cluster-bootstrap.*`,
3. `cluster-database.*`,
4. `cluster-auth.*`,
5. `cluster-monitoring.*`,
6. `cluster-service.*`

are created in the specified order. Resources belonging to `cluster-broker.*` only depend on `cluster-bootstrap.*` resources and can be created in parallel.

```yaml
apiVersion: apiextensions.crossplane.io/v1
kind: Composition
metadata:
  name: m2mgateclusters.blue.m2mgate.cloud
  labels:
    version: v0.0.66
    channel: stable
spec:
  compositeTypeRef:
    apiVersion: blue.m2mgate.cloud/v1alpha1
    kind: M2MGateCluster
  mode: Pipeline
  pipeline:
  # other steps omitted for brevity
  - step: order-resources
    functionRef:
      name: crossplane-contrib-function-sequencer
    input:
      apiVersion: sequencer.fn.crossplane.io/v1beta1
      kind: Input
      rules:
      - sequence:
        - cluster-secret.*
        - cluster-bootstrap.*
        - cluster-database.*
        - cluster-auth.*
        - cluster-monitoring.*
        - cluster-service.*
      - sequence:
        - cluster-bootstrap.*
        - cluster-broker.*
```

Listing 30: Function Sequencer example.

4.4 Pipeline Automation

GitLab CI pipelines sustain platform development, deployment, and usage. The following sections present main pipeline functions.

4.4.1 Control Plane Infrastructure Synchronization

As described in Section 4.1.1, control plane infrastructure state is stored in GitLab's Terraform state storage and synchronized via CI pipeline. The pipeline performs the following actions:

- `tofu fmt`: Formats the OpenTofu codebase.
- `tofu validate`: Validates configuration files, referring only to configuration without accessing remote services.
- `tofu plan`: Creates an execution plan, previewing OpenTofu's planned infrastructure changes.
- `tofu apply`: Executes actions proposed in the OpenTofu plan.
- `tofu destroy`: Destroys all remote objects managed by the OpenTofu configuration.

Table 4.2 shows the different scenarios and the operations that are performed in each of them.

Scenario	`fmt`	`validate`	`plan`	`apply`	`destroy`
Commit on any branch (infra code changes)	✓	✓	✓		
MR targeting `main` (infra code changes)	✓	✓	✓*		
Manual trigger on any branch	✓	✓	✓		
Commit on `main` (infra code changes)	✓	✓	✓	✓†	
Manual trigger on `main`	✓	✓	✓	✓†	
Manual trigger on `main` with variable `TOFU = destroy`					✓†

* Plan output is posted as a GitLab Merge Request comment for team review.
† Manual approval is required to proceed.

Table 4.2: Control plane infrastructure synchronization scenarios.

4.4.2 Project Management

This section presents how CI pipelines automate user project setup and teardown. In the Blue platform context, a project is a managed cluster and its repository.

Creation

Platform users could directly create `M2MGateCluster` objects in the control plane cluster, which would provision an environment with some minor caveats. However, this would violate GitOps principles and platform design goals. The preferred approach creates a new Git repository containing an M2MGateCluster definition and instructs control plane FluxCD to pull it, enabling environment configuration versioning and change tracking. To facilitate this process, the platform provides a single GitLab pipeline endpoint accepting a JSON payload containing environment configuration and required provisioning secrets. Pipeline scripts perform the following steps:

1. Validate user input against the expected schema using KCL validation capabilities.

2. Create a new GitLab repository for the environment.

3. Extract secrets from user input and store them in a GitLab Variable in the new repository.

4. Generate the new repository folder structure containing the M2MGateCluster definition with user-provided configuration and FluxCD Kustomization resources. `cookiecutter` is used for the purpose as GitLab repository templates are a paid feature.

5. Create a new control plane namespace and necessary GitRepository and Kustomization resources to pull new repository content.

By default, environments are created with the latest stable Composition version and `stable` channel, unless the user explicitly requests `unstable` channel creation.

Listing 31 shows user input validation and manipulation in order to obtain the `cookiecutter` configuration file. This example demonstrates how Nushell offers powerful, flexible data manipulation and complex operation execution while enabling external tool usage, as with the `kcl` and `semver` CLIs.

Deletion

Environment deletion is also triggered via a pipeline deleting Namespace, GitRepository, and Kustomization resources in control plane infrastructure code. Related repository removal is performed in an optional step via CLI command, as the repository might serve as a starting point for a second bootstrap with the same configuration (this feature is not yet implemented).

Tooling

The scripts in this section use external tools encapsulated in a Docker image for pipeline runner use. Listing 32 shows the pipeline runner image Dockerfile.

```
# ...
$project_config | save project_config.json
kcl vet project_config.json pipeline/manage-project/schemas/project_config.k | complete
# ...
$secrets
| reduce --fold $project_config { |secret, acc|
    if $secret.value? != null {
        let updated_acc = $acc | reject $secret.field
        if $secret.substitute? != null {
            $updated_acc | insert $secret.substitute.field $secret.substitute.value
        } else {
            $updated_acc
        }
    } else {
        $acc
    }
}
| reject providerAuth # remove providerAuth field
| if not ($env.PROJECT_DEV? == "true") {
  insert crossplane.compositionRevisionSelector.matchLabels.version (resources/bin/semver get release) # set latest stable composition version
  | insert crossplane.compositionRevisionSelector.matchLabels.channel stable # set stable channel
} else {
  insert crossplane.compositionRevisionSelector.matchLabels.channel unstable # set unstable channel
}
| wrap spec # Wrap in spec object
| insert project_slug $project_slug # Add project slug
| insert project_id $project_id # Add project ID
| to json # Convert to JSON
| save pipeline/manage-project/cookiecutter/cookiecutter.json
# ...
```

Listing 31: Example of the user's input validation and manipulation.

```
FROM kcllang/kcl:v0.11.3
ARG COOKIECUTTER_VERSION=2.6.0
ARG GLAB_VERSION=1.74.0
ARG NUSHELL_VERSION=0.107.0
# Install system dependencies
RUN apt-get update && apt-get install -y --no-install-recommends \
    curl \
    git \
    ca-certificates \
    python3 \
    python3-pip \
    && rm -rf /var/lib/apt/lists/*
# Install binary tools
RUN curl -L "https://github.com/nushell/nushell/releases/download/${NUSHELL_VERSION}/nu-${NUSHELL_VERSION}-x86_64-unknown-linux-gnu.tar.gz" | tar -xz -C /usr/local/bin/ --strip-components=1 \
    && pip3 install --no-cache-dir --break-system-packages cookiecutter==${COOKIECUTTER_VERSION} \
    && curl -L "https://gitlab.com/gitlab-org/cli/-/releases/v${GLAB_VERSION}/downloads/glab_${GLAB_VERSION}_linux_amd64.deb" -o /tmp/glab.deb \
    && dpkg -i /tmp/glab.deb \
    && rm -f /tmp/glab.deb
# ...
```

Listing 32: Dockerfile for the environment management pipeline image.

4.4.3 Version updates and release management

The second principle of GitOps, presented in Section 3.1.2, requires that the desired state is stored in a way that enforces immutability. Considering that parts of this state are stored in OCI artifacts (e.g., KCL modules, Crossplane Configurations, etc.), such artifacts must use immutable tags rather than mutable ones (e.g., `latest`, `unstable`, or `dev`). The importance of immutable tags becomes evident when examining how Kubernetes operators handle resource changes. When a Kubernetes resource references an OCI artifact using a mutable tag, the tag name in the resource definition remains constant even when the underlying artifact changes. Consequently, when applying the Kubernetes manifest (e.g. using FluxCD), the declarative management system detects no changes in the resource specification and does not trigger the operators to pull the updated artifact from the registry. This breaks the GitOps reconciliation loop, as new versions of the artifacts are never fetched despite being available in the registry.

In contrast, immutable tags, such as semantic versions (e.g., `v1.2.3`) or commit SHAs, ensure that each artifact version has a unique identifier. When updating to a new version, the tag reference in the Kubernetes resource definition changes, creating a detectable modification in the desired state. This change triggers the reconciliation process, prompting the operators to pull the new artifact version from the OCI Registry. By enforcing immutability at the artifact level, every commit that produces a new artifact version results in a corresponding change in the control plane configuration, ensuring that the actual state converges to the desired state as prescribed by GitOps principles. In order to achieve this goal, a GitLab CI pipeline has been implemented to update the tags of the OCI artifacts and their corresponding references in the Kubernetes resources whenever a new version of the platform, stable or unstable, is released.

Update

The update process is triggered at every commit to the `main` branch. The pipeline performs the following operations:

1. Create a new tag in the format `main-<Commit SHA (first 8 characters)>`.
2. Update the the KCL code referencing OpenTofu OCI artifacts to the new tag (e.g. `kube-hetzner` and `azuread` modules).
3. Update the version of the KCL modules, in `kcl.mod`, to the new tag.
4. Re-render the API Specification documentation for the M2MGateCluster resource, from its KCL schema.
5. Update the the Crossplane Composition resource `labels.version` to the new tag. `labels.channel` is set to `unstable`.
6. Update the Crossplane Configuration resource to point to the new artifact's tag.

7. Update the pipeline definition to use the new tag for the pipeline runner images.
8. Build and push the KCL modules with the new tag to the OCI Registry.
9. Build and push the Crossplane Configuration with the new tag to the OCI Registry.
10. Build and push the OpenTofu modules with the new tag to the OCI Registry.
11. Build and push the Docker images for the pipeline runner with the new tag to the OCI Registry.
12. Push the changes to the repository.

The FluxCD agent running in the control plane will pull and apply the new definitions, creating a new ConfigurationRevision for Crossplane and a new CompositionRevision for the M2MGateCluster resource. The latter will have the new tag in its `labels.version` field and the `labels.channel` field will be set to `unstable`. By setting the `channel` label to `unstable` in the `compositionRevisionSelector` field of the M2MGateCluster resource without specifying a version, it's possible to obtain an environment using the latest unstable version of the platform as soon as it is available. This is useful for development purposes or for testing new features before they are released to the stable channel.

Listing 33 shows as an example, the part of the script updating the version of the KCL modules and publishing their OCI artifacts to the OCI Registry.

Release

The release process is triggered manually by the platform maintainers by creating a new pipeline with the CI variable `SEMVER` set to `release`, `minor` or `major` depending on the type of release. Most of the code is shared with the update process, with the following differences and additions:

- The new tag is created following the semantic versioning scheme, leveraging the `semver` CLI [36].
- The the Crossplane Composition resource `labels.channel` is set to `stable`.
- A changelog containing the reference to the commits included in the release is generated using the GitLab CLI [37].
- Update version of the platform CLI to the new tag.
- Push the changes to the repository and tag the commit with the new semantic version.
- Create a TAR archive of the platform CLI scripts.
- Create a new release in GitLab, using the GitLab CLI, including release notes and the platform CLI archive.

- Send a notification to the platform's Microsoft Teams channel, including the release notes.

```
# Save root directory
let root_dir = pwd
# Update version in common kcl.mod
cd $"kcl/modules/common"
open kcl.mod
| from toml
| update package.version $version
| to toml
| save --force kcl.mod
cd $root_dir
# Update version in other KCL modules kcl.mod
$kcl_modules | each { |module|
  cd $"kcl/modules/($module)"
  open kcl.mod
  | from toml
  | update package.version $version
  | update dependencies.common.tag $version
  | to toml
  | save --force kcl.mod
  cd $root_dir
}
# Login to OCI registry
kcl registry login -u $env.CI_REGISTRY_USER -p $env.CI_REGISTRY_PASSWORD $env.CI_REGISTRY
# Push common KCL module
cd $"kcl/modules/common"
kcl mod push --force
↪  $"oci://($env.CI_REGISTRY)/blue/blue-platform/kcl/modules/common:($version)"
cd $root_dir
# Push other KCL modules
$kcl_modules | each { |module|
  cd $"kcl/modules/($module)"
  kcl mod update # fetch latest common module
  kcl mod push --force
  ↪  $"oci://($env.CI_REGISTRY)/blue/blue-platform/kcl/modules/($module):($version)"
  cd $root_dir
}
```

Listing 33: Script updating the version of the KCL modules and publishing them to the OCI Registry.

Tooling

Similarly to the environment management pipeline, the version updates and release management pipeline makes use of a Docker image for the pipeline runner to use. In particular Nushell and the following tools were installed:

- `docker-cli`: Docker CLI to build the Docker images.
- `docker-buildx`: Docker Buildx plugin for advanced Docker image building.
- `git`: Git to clone the repository and push the changes.
- `oras`: Oras CLI to authenticate to the OCI Registry and push OpenTofu OCI artifacts.

- `kcl`: KCL CLI to build and push KCL modules to the OCI Registry.
- `crossplane`: Crossplane CLI to build and push Crossplane Configurations to the OCI Registry.
- `glab`: GitLab CLI to generate changelogs and release artifacts.

In order to build the Docker images, the pipeline execution makes use of a container based on the image described above and of a `service` container running a Docker in Docker (DinD) environment. The `docker-cli` running in the custom container is configured to interact with the DinD container via TCP socket. Listing 34 shows how this mechanism is implemented in GitLab CI.

```yaml
update:
  stage: update
  image: ${CI_REGISTRY}/blue/blue-platform/pipeline/manage-version/gitlab-runner:v0.0.66
  services:
  - docker:28.5-dind
  variables:
    DOCKER_HOST: tcp://docker:2376
    DOCKER_TLS_CERTDIR: /certs
    DOCKER_TLS_VERIFY: 1
    DOCKER_CERT_PATH: $DOCKER_TLS_CERTDIR/client
  script: nu pipeline/manage-version/scripts/update.nu
  rules:
  - if: $CI_COMMIT_BRANCH == $CI_DEFAULT_BRANCH
release:
  stage: release
  image: ${CI_REGISTRY}/blue/blue-platform/pipeline/manage-version/gitlab-runner:v0.0.66
  services:
  - docker:28.5-dind
  variables:
    DOCKER_HOST: tcp://docker:2376
    DOCKER_TLS_CERTDIR: /certs
    DOCKER_TLS_VERIFY: 1
    DOCKER_CERT_PATH: $DOCKER_TLS_CERTDIR/client
  script: nu pipeline/manage-version/scripts/release.nu
  rules:
  - if: $SEMVER
    when: manual
```

Listing 34: GitLab CI pipeline for the version updates and release management.

4.4.4 Documentation

The platform documentation is expressed as a set of Markdown files in the `docs` folder of the platform's repository. In the context of a release, these files are compiled in a set of HTML static pages using MkDocs [38] and the `mike` plugin [39]. The compiled pages are placed in a new folder in a dedicated branch of the repository (e.g. `gl-pages`) and published on GitLab Pages. The `mike` plugin can create aliases for the different versions of the documentation, in this case the `latest` alias is set to the latest stable version of the platform at every release. A permalink to the latest version of the documentation is also created allowing user redirection when navigating the base URL of the documentation. The script used to build the documentation is listed in Listing 35, it is an adaptation of a script found on

StackOverflow [40]. The script uses a custom container based on the official Python image and installing the necessary packages to build the documentation such as `mkdocs`, `mike`, `mkdocs-material` etc. The configuration file for MkDocs is also reported in Listing 36.

```yaml
publish_docs:
  stage: publish_docs
  image: ${CI_REGISTRY}/blue/blue-platform/pipeline/publish-docs/gitlab-runner:v0.0.66
  variables:
    PAGES_BRANCH: gl-pages
    HTTPS_REMOTE: https://git:${GITLAB_BLUE_TOKEN}@git.example.com/blue/blue-platform.git
  script:
  - # ...
  - mike deploy --deploy-prefix public -r $HTTPS_REMOTE -p -b $PAGES_BRANCH -u $CI_COMMIT_TAG
  ↪  latest
  - mike set-default --deploy-prefix public -r $HTTPS_REMOTE -p -b $PAGES_BRANCH latest
  pages: true
  rules:
  - if: $CI_COMMIT_TAG
```

Listing 35: Script building the documentation.

```yaml
site_name: Blue Documentation
site_url: https://gitlab-pages.example.com/blue/blue-platform/
repo_url: https://git.example.com/blue/blue-platform.git
repo_name: Blue - Platform
site_dir: public
remote_branch: gl-pages
theme:
  name: material
  palette:
    # omitted for brevity
  features:
    - navigation.tabs
    - navigation.indexes
    - navigation.expand
    - content.code.copy
  logo: pictures/blue-logo.png
  favicon: pictures/blue-logo.png
markdown_extensions:
  - attr_list
  - pymdownx.highlight
  - pymdownx.superfences
  - pymdownx.details
  - admonition
  - pymdownx.snippets:
      base_path: docs
plugins:
  - search:
      lang: en
  - macros:
      include_dir: docs/snippets
extra:
  version:
    provider: mike
nav:
  # omitted for brevity
```

Listing 36: MkDocs configuration file.

4.5 CLI Implementation

The platform CLI is implemented as a set of scripts written in Nushell. This implementation choice has greatly simplified the development, while still offering rich functionalities, when compared to a more traditional approach using a programming language like Python or Go. The downside of this approach is that the CLI cannot be shipped as a single binary, but rather as a set of scripts which require the Nushell interpreter to be installed on the target system. Such compromise has been accepted due to the fact that the CLI has to rely on third-party tools in any case (e.g. `hcloud` CLI), unless integrating third-party functionalities by means of SDKs and custom built API clients.

It is also worth mentioning that in the future it might be considered the possibility to develop other types of interfaces to the platform, such as a web GUI, for this reason the effort for the development of the CLI has been kept to a minimum at this stage.

4.5.1 Installation Script

An installation script written in Zsh, POSIX-compatible shell default on macOS, has been implemented to ease the installation of the platform CLI and its dependencies. The script, which must be fetched from the platform's repository, essentially performs three simple operations:

1. Installing all the third-party dependencies required by the platform CLI.
2. Fetching the platform CLI scripts from the repository a placing them in the user's home directory.
3. Update the user's PATH environment variable to include the platform CLI scripts directory.

```
brew install -q hcloud > /dev/null 2>&1
# other dependencies omitted for brevity
mkdir -p $HOME/.blue
git clone -q git@git.m2mgate.de:blue/blue-platform.git /tmp/blue-platform
CURRENT_DIR=$(pwd)
cd /tmp/blue-platform
git checkout -q $BLUE_VERSION
mv cli/src/* $HOME/.blue
mv cli/blue-update $HOME/.blue
cd $CURRENT_DIR
rm -rf /tmp/blue-platform
echo -e "\n\n# Blue CLI path settings" >> $HOME/.zshrc
echo 'export PATH="$PATH:$HOME/.blue"' >> $HOME/.zshrc
echo "# End of Blue CLI path settings" >> $HOME/.zshrc
source $HOME/.zshrc
```

Listing 37: Installation script for the platform CLI.

The list of dependencies installed by the script is the following:

- `packer` - CLI tool from Hashicorp to build machine images from templates [41]. It is used to build MicroOS images for the managed cluster nodes.

- `nushell` - shell environment for the CLI.

- `glab`: GitLab CLI to interact with the GitLab server API.

- `kubectl`: Kubernetes CLI to interact with the Kubernetes API.

- `gum`: CLI tool which offers commands for user interface elements. It is used to prompt the user for input and format some of the CLI output.

- `kcl`: CLI tool to interact with the KCL language. It is used to validate the user's input against the expected schema.

- `crossplane`: CLI tool to interact with Crossplane running in the control plane cluster. It is used to fetch information about the deployed M2MGateCluster resources.

- `hcloud`: CLI tool to interact with the Hetzner Cloud API. It is used to clean up some resources created by the platform.

4.5.2 Authentication Module

The platform CLI provides a set of commands to interact with the platform's services. In order to do so, the CLI needs to authenticate. In particular it needs authentication against the GitLab server and the Kubernetes API of the control plane cluster. In this phase of the platform development it has not been implemented a proper fine grained authorization mechanism, it has been decided that a user with access to the platform's repository has granted access to the full set of functionalities of the platform. The authentication script is a wrapper for the authentication command of the GitLab CLI. The user is prompted to enter a GitLab API token. The token must have the `api` scope on the `Blue` group, containing the platform's repository and all the projects managed by the platform. Such token can be a personal access token or a group access token. Once the user is authenticated against the GitLab server, the CLI will retrieve the `kubeconfig` file of the control plane from a GitLab variable stored in the platform's repository.

Listing 38 shows the login subcommand implementation. The comments and `@example` notations are used by Nushell to generate the documentation for the command, automatically added to the command help prompt.

```
# Authenticate with Blue Platform
@example "interactive mode" { blue auth login }
@example "non-interactive mode" { blue auth login --token <gitlab_api_token> }
def "main auth login" [
  --token (-T): string # The GitLab API token to authenticate with
] {

  mut auth = $token

  # If no token is provided, prompt the user for it
  if $auth == null {
    $auth = (gum input --padding "1 0" --placeholder "Enter your GitLab API token" --password)
  }

  # Authenticate with glab and verify status
  $auth | glab auth login --hostname git.m2mgate.de --api-protocol https --git-protocol ssh
  ↪  --stdin
  let result = $in | glab auth status --hostname git.m2mgate.de | complete

  if $result.exit_code != 0 {
    print error "Authentication failed. Is the token correct?" $result.stderr
    exit $result.exit_code
  }

  # Set the default host for glab
  glab config set --global host git.m2mgate.de | complete

  # Get the control plane kubeconfig from the Blue Platform repository
  let control_plane_kubeconfig = (glab variable get KUBECONFIG --repo blue/blue-platform) |
  ↪  from yaml

  open $"($env.HOME)/.kube/config"
  | from yaml
  | update clusters { where { $in.name != "blue"} }
  | update contexts { where { $in.name != "blue"} }
  | update users { where { $in.name != "blue"} }
  | merge deep $control_plane_kubeconfig --strategy=append
  | to yaml
  | save --force $"($env.HOME)/.kube/config"

  print success $"Authenticated with Blue Platform"
}
```

Listing 38: Login subcommand implementation.

4.5.3 Project Module

The CLI implements commands to help the user manage projects. In particular it provides a command to create a new project, delete an existing project, get the `kubeconfig` file of a project cluster and upload a Keycloak theme to the project's Keycloak instance.

Create

The `project create` command accepts a YAML file as input which schema is very similar to the one of the `M2MGateCluster` resource and defined using KCL. Other parameters include the possibility to skip confirmation prompts, skip the creation of snapshots using Packer, specify version of the platform configuration to use or if the unstable channel should be used. The command performs the following operations:

1. Verifies that the user is authenticated.
2. Determines the platform version to use, if not specified the latest stable version is used.
3. Fetches the input schema from the platform repository.
4. Validate the user's input against the expected schema.
5. Checks if a project with the same identifier already exists.
6. Prompts the user for confirmation to proceed with the creation.
7. If the cloud provider is Hetzner (the only supported one at this stage), it creates a new snapshot of the MicroOS image using Packer.
8. Retrieves a pipeline trigger token from the platform repository.
9. Triggers the pipeline to create the project presented in section 4.4.2.
10. Returns the pipeline logs to the user.
11. Waits for the M2MGateCluster resource to be created in the control plane and for its status to be ready.
12. Retrieves the credentials of the project cluster and adds them to the user's `kubeconfig` file.
13. Prints a success message to the user.

Listing 39 shows a snippet (pseudo-code) of the implementation of the `project create` command.

```
verify authentication blue
# ...
kcl vet $values_path /tmp/blue-platform/pipeline/manage-project/schemas/project_config.k --format yaml | complete
# ...
let pipeline_id = http post https://git.m2mgate.de/api/v4/projects/459/trigger/pipeline --content-type "application/json" {
  token: $trigger_token
  ref: $ref
  variables: { PROJECT_CONFIG: ($values_content | to json --raw), PROJECT_DEV: ($dev | into string) }
} | get id
glab ci trace "setup_project" --pipeline-id $pipeline_id --repo blue/blue-platform
glab ci trace "create_project" --pipeline-id $pipeline_id --repo blue/blue-platform
# ...
let elapsed_time = timeit {
  try {
    kubectl wait --context blue -n $project_slug --for=condition=Ready m2mgatecluster m2mgate-cluster --timeout=1h
  } catch {
    exit 1
  }
}
```

Listing 39: Snippet of the implementation of the `project create` command.

Delete

The `project delete` command accepts the identifier of the project to delete as input. Optionally the user can specify if confirmation prompts should be skipped and if the project repository should be deleted. The command performs the following operations:

1. Verifies that the user is authenticated.
2. Checks if the project exists.
3. Prompts the user for confirmation to proceed with the deletion.
4. Retrieves the pipeline trigger token from the platform repository.
5. Triggers the pipeline to delete the project presented in section 4.4.2.
6. Returns the pipeline logs to the user.
7. Waits for the M2MGateCluster resource to be deleted in the control plane.
8. Removes the project's credentials from the user's `kubeconfig` file.
9. If the project repository should be deleted, it deletes the repository from the GitLab server using the GitLab CLI.
10. Prints a success message to the user.

The project is deleted from the GitLab server using the GitLab CLI.

Get Kubeconfig

The `project getkubeconfig` command accepts the identifier of the project and adds to the `kubeconfig` file of the user the credentials of the project cluster. The project credentials are retrieved from a Kubernetes secret in the control plane cluster using the Kubernetes CLI.

Upload Keycloak Theme

The `project uploadtheme` command accepts the identifier of the project and the path to the Keycloak theme folder as input. The theme folder is uploaded to the project's Keycloak instance using the Kubernetes CLI, interacting with the project's cluster API. The functionality is implemented as follows:

1. Verification of user authentication (GitLab and control plane).
2. If not specified, the user is prompted to enter the path to the theme folder.
3. Verify that the path to the theme folder exists.
4. Verification of the authentication to the project's cluster.

5. Removal of the existing theme folder with the same name from the project's Keycloak instance.
6. Creation of a debug container attached to the project's Keycloak pod.
7. Uploads the theme folder to the project's Keycloak main container, using the debug container.
8. Deletes the Keycloak pod.
9. Waits for the new Keycloak pod to be ready.
10. Prints a success message to the user.

The usage of a debug container to upload the theme folder is necessary because the main container running in the Keycloak pod does not have the `tar` binary installed, which is required by the `kubectl cp` command. The debug container accesses the main container filesystem at the a path mounted on a volume as shown in Section 4.2.3 by sharing the same process Linux namespace and copying the files in the `/proc/1/root` filesystem subtree. Listing 40 shows a snippet (pseudo-code) of the implementation.

```
kubectl exec --context $project_slug -n keycloak keycloak-0 --container keycloak -- rm -rf /opt/keycloak/themes/($theme_name)
kubectl debug --context $project_slug -n keycloak keycloak-0 --image=busybox --container theme-copy --profile general --target=keycloak -- sleep infinity
kubectl cp --context $project_slug $theme_path $"keycloak/keycloak-0:/proc/1/root/opt/keycloak/themes/($theme_name)" -c theme-copy
kubectl delete --context $project_slug -n keycloak pod keycloak-0 | complete
kubectl wait --context $project_slug -n keycloak --for=condition=Ready pod keycloak-0 --timeout=5m
```

Listing 40: Snippet of the implementation of the `project uploadtheme` command.

4.5.4 M2MGateCluster Module

This module offers one command to allow users to inspect M2MGateCluster resources created in the control plane cluster. The functionalities of this module might be moved to the Project Module in the future. The command offers two types of output:

- One returns the YAML representation of the M2MGateCluster resource and its events, consulting the Kubernetes API. Similar to the `kubectl describe` command.
- The other returns the resources tree deriving from the composition of the M2MGateCluster resource and their synchronization/readiness status. Similar to the `crossplane beta trace` command. The output of this command resembles the one shown in Appendix C.

Chapter 5

Evaluation

5.1 Platform Capabilities and Operational Validation

The platform's core functionalities have been validated through extensive manual testing, focusing on the three fundamental cluster lifecycle operations: environment creation, updates, and deletion. This section presents the operational characteristics observed during testing and discusses the current state of quality assurance for the platform.

5.1.1 Functional Testing Methodology

Due to the nature of infrastructure-as-code platforms and the complexity of Kubernetes cluster orchestration, traditional software testing methodologies are not directly applicable. The validation of platform capabilities required manual testing procedures that encompassed the entire lifecycle of managed clusters. Comprehensive automated testing remains an area for future development, as it necessitates specialized testing frameworks and techniques distinct from conventional software testing approaches.

The development of automated tests for this platform presents several unique challenges:

- **KCL Code Generation Validation:** Verifying the correctness of dynamically generated KCL compositions requires specialized assertion mechanisms capable of parsing and validating infrastructure definitions.

- **Deployment Correctness Verification:** Ensuring that Crossplane successfully provisions all specified resources in the target cloud infrastructure demands integration with multiple APIs and state reconciliation checks.

- **Multi-Cluster State Validation:** Testing must account for the distributed nature of the system, where state exists across the control plane, managed clusters, and external infrastructure providers.

- **Time-Dependent Operations:** Many platform operations are inherently asynchronous, requiring test frameworks capable of handling long-running reconciliation loops and eventual consistency.

Given these complexities, the design and implementation of a comprehensive automated testing suite represents a substantial engineering effort that extends beyond the scope of the initial platform development.

5.1.2 Lifecycle Operations Performance

Manual testing of the platform's core operations yielded the performance characteristics summarized in Table 5.1.

Operation	**Metric**	**Value/Status**
Environment Creation	Time to completion	30–60 minutes
Environment Creation	Success rate	100% (manual testing)
Environment Updates	State propagation time	<5 minutes
Environment Deletion	Resource cleanup	Complete and correct

Table 5.1: Platform lifecycle operations performance characteristics

Environment Creation. The provisioning of a complete M2MGate cluster environment, including cloud infrastructure provisioning via Crossplane, Kubernetes cluster initialization, and application service deployment, consistently completed within 30 to 60 minutes. This timeframe encompasses:

- Infrastructure resource provisioning (compute instances, networking, storage)
- Kubernetes control plane initialization and worker node joining
- Core service deployment (CNI, CSI, ingress controllers)
- Database deployment (MySQL) and initialization (∼15min)
- Keycloak deployment and initialization (∼10min)
- M2MGate application stack deployment and configuration

The observed duration aligns with expectations for cloud infrastructure provisioning and represents a significant improvement over the manual provisioning process documented in Chapter 2, which required several days of engineering effort.

Environment Updates. Modifications to managed environments, whether triggered by changes to the project repository or updates to Crossplane compositions, propagate to target clusters in under five minutes. The platform successfully maintained configuration drift prevention throughout testing, ensuring that manual modifications to managed clusters were automatically corrected.

Environment Deletion. The platform's cleanup mechanisms functioned correctly in all test scenarios, with complete removal of provisioned cloud resources and no observed orphaned infrastructure. The deletion process properly handles dependency ordering, ensuring that Kubernetes resources are removed before their underlying infrastructure. One exception was observed for nodes in the auto-scaling pool. When created they have no representation in the state stored in the control plane and are not deleted automatically. The platform CLI must run a cleanup pass using the `hcloud` tool to delete them.

5.1.3 Production Trial and Adoption

Following the completion of initial development and validation, the platform entered a trial phase within the organization. Development teams have begun utilizing the platform to provision ephemeral development environments for testing new M2MGate features and conducting integration testing. This trial phase serves multiple purposes:

- Validation of the platform under real-world usage patterns
- Identification of user experience issues and workflow inefficiencies
- Assessment of platform reliability under diverse workload conditions
- Gathering feedback for future enhancements and feature prioritization

Early adoption feedback has been positive, with teams reporting significant reductions in environment setup time and improved consistency across development environments.

5.1.4 Identified Limitations and Future Work

Control Plane Scalability. During the trial phase, preliminary observations suggest a potential resource bottleneck in the control plane cluster that may limit the number of environments that can be concurrently managed. In particular the pod running `function-kcl` has proven to be the most resource intensive component. This can possibly be improved by optimizing the KCL code, although issues on GitHub [42] suggest that it might be a limitation of the `function-kcl` software itself. The current control plane configuration has not been stress-tested under high-load conditions with numerous concurrent managed clusters.

Automated Testing Infrastructure. As discussed in Section 5.1.1, the development of comprehensive automated testing capabilities represents a significant area for future work. Priority areas for test automation include:

- Validation of generated KCL compositions against schema specifications
- End-to-end provisioning tests using temporary cloud infrastructure
- Regression testing for composition updates and versioning
- Performance benchmarking of cluster provisioning operations

5.2 Performance Benchmarking

To validate the successful migration of the M2MGate platform from Docker Swarm to Kubernetes, a comprehensive set of performance benchmarks were conducted on a test deployment. The primary objectives of this evaluation were to:

- Assess system responsiveness under high-load conditions
- Verify functional parity with the legacy Swarm deployment
- Identify performance bottlenecks and resource constraints
- Validate scalability characteristics in the Kubernetes environment

All tests were conducted using the `stream-sim` device simulation tool, created for the purpose of these tests and deployed on an existing Swarm infrastructure for development, to simulate realistic device behavior and generate controlled load patterns.

5.2.1 Test Infrastructure

The Kubernetes cluster used for testing comprised three worker nodes, each equipped with 16GB of RAM and 8 CPU cores. Table 5.2 presents the common simulation parameters used across all test scenarios.

Parameter	**Value**
Total simulated devices	50,000
Simulation instances	10
Devices per instance	5,000
Thread pool size	10 threads per instance

Table 5.2: Device simulation configuration parameters

5.2.2 Connectivity Responsiveness Tests

These tests evaluated the system's ability to handle large-scale, simultaneous device connections and assess the time required to onboard and correctly display connected devices in both the monitoring dashboards and the web portal.

Test Scenario 1: Hourly Device Information (DIGeneral) Updates

The first connectivity test simulated a scenario with minimal device activity, where each device sends infrequent status updates. Table 5.3 details the test parameters.

The results, shown in Table 5.4, demonstrate that while the cascade component rapidly detected all connections, the portal exhibited significant latency in reflecting the system state.

Parameter	Value
Initial device count	0 (cold start)
Device info delivery rate	1 per hour
Ping interval	300 seconds
Idle timeout threshold	10 pings

Table 5.3: Connectivity test 1 parameters

Component	Metric	Result
Cascade	Connection detection time	∼1 minute
Cascade	Devices detected	50,000 (100%)
Portal	Device display time	∼30 minutes
Portal	Devices displayed	50,000 (100%)
Logs	Error status	No errors

Table 5.4: Connectivity test 1 results

Test Scenario 2: Per-Minute Device Information (DIGeneral) Updates

The second test increased the device information delivery rate by a factor of 60, creating a more intensive load on the system. Table 5.5 shows the modified parameters.

Parameter	Value
Initial device count	0 (cold start)
Device info delivery rate	1 per minute
Ping interval	300 seconds
Idle timeout threshold	10 pings

Table 5.5: Connectivity test 2 parameters

As illustrated in Table 5.6, this increased load exposed a critical bottleneck: the `distribution-service` exhausted its database connection pool and became unresponsive. Furthermore, even after the simulation terminated, the portal continued to show an increasing device count, suggesting a significant backlog in event processing.

Component	Metric	Result
Cascade	Connection detection time	∼1 minute
Cascade	Devices detected	50,000 (100%)
Portal	Device display time	>20 minutes
Portal	Device count accuracy	Incorrect
Distribution-service	Status	Unresponsive (DB pool exhaustion)
Post-simulation	Cascade disconnection	Correct (0 devices)
Post-simulation	Portal behavior	Anomalous (count increasing)

Table 5.6: Connectivity test 2 results

5.2.3 Message Handling Tests

These tests assessed the system's capacity to handle high message throughput from the Message Adapter Service (MAD) messages generated by a large device fleet.

Test Scenario: High-Frequency MAD Messages

Table 5.7 presents the test configuration, designed to simulate realistic high-traffic conditions with a gradual device connection process.

Parameter	Value
Initial device count	0 (cold start)
Device connection delay	500 ms between devices
Device info delivery rate	1 per hour
Ping interval	300 seconds
MAD messages per device	60 per minute
MAD message type	TEXT
MAD message size	0–1 KB (variable)
Expected peak throughput	50,000 msg/sec (worst case: 50 MB/sec)

Table 5.7: Message handling test parameters

The results in Table 5.8 reveal a critical scalability constraint: the `mad-service`, despite having access to substantial resources (600m CPU, 2GB RAM), could only sustain approximately 1,340 messages per second before resource exhaustion. This corresponds to supporting only 5,000–6,000 actively messaging devices, representing 10–12% of the target 50,000 device capacity.

Phase	Observation	Value/Status
Initial phase	System response	Handled correctly
Mid-test	Kafka throughput	Plateaued at ∼1.34K msg/sec
Mid-test	mad-service resources	Approaching limits
Late-test	mad-service status	Hit resource limits
Late-test	Kafka throughput	Declining
Resource Analysis		
mad-service allocation	CPU	600m
mad-service allocation	Memory	2GB
Supported device count	Before saturation	5,000–6,000 devices

Table 5.8: Message handling test results

5.2.4 Job Creation and Distribution Tests

This test evaluated the system's capability to create and execute jobs that distribute resources to large device fleets, a common operational requirement for firmware updates and configuration management.

Test Scenario: Large-Scale Job Distribution

Table 5.9 describes the job distribution test setup.

Parameter	Value
Connected devices	50,000
Filter configuration	All 50,000 devices
Resource type	FILE
Resource size	∼1 KB
Device info delivery rate	1 per hour
Ping interval	300 seconds
Database configuration	Clustered (Percona XtraDB)

Table 5.9: Job creation test parameters

The results presented in Table 5.10 demonstrate that while the system successfully created and began executing the job, the process exhibited two notable characteristics: (1) only a subset of devices from the filter were initially added to the job (5,000 out of 50,000), and (2) the completion rate, while steady, would

Metric	Time	Value/Status
Job creation	–	Successful
Device assignment delay	–	Several minutes
Devices added to job	–	5,000 (10% of filter)
Filter size	–	50,000 devices
Completion rate	+30 min	25% (1,250/5,000 tasks)
Completion rate	+1h 23min	59.2% (2,960/5,000 tasks)
Test status	+1h 23min	Interrupted
Projected completion	–	Likely successful
Overall system stability	Throughout	No critical failures

Table 5.10: Job creation test results

require several hours to complete all tasks. The test was terminated after 83 minutes with nearly 60% completion, suggesting that full job execution would require approximately 2–3 hours for 5,000 devices.

5.2.5 Key Findings and Analysis

The comprehensive benchmark suite revealed several important insights regarding the Kubernetes-based deployment:

Performance Parity. The Kubernetes environment demonstrated comparable performance to the legacy Docker Swarm deployments, with no inherent performance penalties attributable to the orchestration platform itself. Identical issues observed in both environments confirmed that the identified bottlenecks are application-level rather than infrastructure-related.

Identified Bottlenecks. Two primary performance constraints were identified:

1. **Database Connection Pool Saturation:** High-frequency device information deliveries (1 per minute per device) caused the `distribution-service` to exhaust its database connection pool, rendering it unresponsive. This issue persisted even when testing with a single-instance MySQL database configured with 700 maximum connections, ruling out clustered database configurations as the root cause. Must be said that these rates are not realistic for a production environment, but are used for testing purposes to simulate a high-load scenario.

2. **Message Processing Capacity:** The `mad-service` exhibited linear resource consumption growth with message throughput, reaching its 2GB memory and

600m CPU core allocation limits at approximately 1,340 messages per second. This translates to a practical limit of 5,000–6,000 devices generating one message per second, significantly below production fleet requirements.

Functional Verification. Despite the performance limitations, all core M2MGate functionalities operated correctly in the Kubernetes environment, including:

- Debug tooling via `dtools-service`
- Topic and schema management via `mad-service`
- Resource and job management via `distribution-service`
- User and tenant administration via `tenant-service`
- VPN creation and management via `vpngate-service`

Limitations. VPN connectivity testing with large device fleets was not possible due to limitations in the simulation software.

Chapter 6

Conclusion

This thesis addressed the modernization of infrastructure provisioning for M2MGate, a commercial IoT platform, by transitioning from a manual, error-prone process to a fully automated, cloud-native solution. The work demonstrated that systematic application of modern DevOps principles and cloud-native technologies can transform legacy infrastructure management into a streamlined, scalable, and maintainable system.

6.1 Impact and Benefits

The implementation of the Blue platform delivers tangible benefits across multiple dimensions:

Operational Efficiency. Provisioning time decreased from several days to under one hour, dramatically accelerating project initialization and customer onboarding. The automated approach eliminates manual configuration steps and associated human error, improving reliability and reducing troubleshooting time. Organizations can now provision development, testing, and production environments rapidly, enabling more agile development practices and faster iteration cycles.

Consistency and Reliability. GitOps-based state management ensures that all environments are provisioned from version-controlled, declarative definitions, eliminating configuration drift. Every deployment follows identical procedures, producing predictable outcomes regardless of who initiates the provisioning. This consistency simplifies operations, reduces cognitive load, and enables confident deployment across development, staging, and production environments.

Improved Collaboration. The platform establishes clear boundaries between platform capabilities and application-specific customizations, enabling better separation of concerns between operations and project teams. Project teams gain autonomy to manage their environments through declarative configuration, while operations teams maintain control over platform evolution and infrastructure standards. The shared Git-based workflow provides a natural collaboration mechanism with built-in review processes and change tracking.

Knowledge Democratization. Comprehensive documentation and self-service capabilities reduce dependency on individual experts, distributing knowledge across the organization. New team members can quickly become productive, and the barrier to entry for provisioning and managing environments is significantly lowered. The platform itself serves as executable documentation, codifying best practices and institutional knowledge in maintainable infrastructure code.

Platform Engineering Maturity. As analyzed through the Platform Engineering Maturity Model framework, the solution substantially increased the organization's maturity across multiple dimensions. The platform represents a transition from ad-hoc, reactive infrastructure management to a product-oriented approach with defined roadmaps, user feedback mechanisms, and continuous improvement processes.

6.2 Lessons Learned

The development and deployment of the Blue platform provided valuable insights into the practical challenges of building internal developer platforms:

Complexity Management. While the platform successfully abstracts infrastructure complexity for end users, the underlying implementation necessarily embodies significant technical sophistication. The integration of multiple cloud-native tools (Crossplane, FluxCD, KCL, OpenTofu) required deep understanding of each component's operational characteristics and interaction patterns. Future platform engineering efforts should budget substantial time for tool integration and edge case handling.

Testing Challenges. The lack of comprehensive automated testing emerged as a significant limitation. Infrastructure-as-code platforms present unique testing challenges that extend beyond traditional software testing methodologies and remain areas for future development.

Documentation as First-Class Concern. The emphasis on documentation proved essential for platform adoption. Technical capabilities alone are insufficient; users require clear guidance, examples, and troubleshooting resources. Embedding documentation in the codebase and versioning it alongside platform releases ensured consistency and relevance.

Iterative Refinement. The platform evolved significantly during implementation as real-world requirements revealed gaps in initial designs. An iterative approach with continuous feedback from early adopters enabled course corrections and feature prioritization based on actual usage patterns rather than theoretical needs.

6.3 Future Work and Platform Evolution

While the Blue platform successfully addresses the core provisioning challenges identified in this thesis, several areas for future enhancement and expansion have been identified. These improvements will further increase the platform's capabilities, security posture, and flexibility.

6.3.1 Security and Access Control

The current implementation lacks fine-grained authorization mechanisms for platform capabilities. Future work will focus on implementing Role-Based Access Control (RBAC) policies that protect the control plane from unauthorized modifications and enable more granular authorization for different organizational roles. Admission policies will be developed to validate resource definitions before they are applied to managed clusters, preventing misconfigurations and enforcing organizational standards. These security enhancements will enable the platform to support larger organizational structures with varying levels of trust and responsibility, ensuring that users can only access and modify resources appropriate to their role.

6.3.2 Multi-Cloud Support Expansion

The current implementation focuses primarily on Hetzner Cloud as the infrastructure provider. Future development will expand support to additional cloud providers, including major public cloud platforms such as Amazon Web Services (AWS), Google Cloud Platform (GCP), and Microsoft Azure. This expansion will leverage Crossplane's multi-cloud capabilities and the extensive provider ecosystem to enable customers to deploy M2MGate environments on their preferred infrastructure.

6.3.3 Graphical User Interface Development

While the command-line interface provides efficient access to platform capabilities for technical users, a graphical user interface would lower barriers to adoption and improve the user experience for less technically oriented stakeholders. The development of a web-based GUI using Backstage, a leading open-source platform for building developer portals, is planned as a future enhancement.

6.3.4 Platform Extensibility

A fundamental requirement for platform adoption by diverse project teams is the ability to deploy project-specific resources beyond the standard M2MGate stack. To address this need, three new custom resources are currently under development:

- **ResourceDeployment.** This resource provides a wrapper around the Object resource of `provider-kubernetes`, enabling users to deploy arbitrary Kubernetes objects to their managed clusters.

- **HelmDeployment.** Building on the same principles, HelmDeployment wraps the `provider-helm` Release resource to enable deployment of arbitrary Helm charts to managed clusters.
- **SecretDeployment.** Creates an ExternalSecret resource to fetch secrets from GitLab variables and a `provider-kubernetes` Object resource (wrapping a Secret resource) patched with the retrieved values, enabling users to securely deploy secrets to their managed clusters.

These extensibility features represent a critical step toward platform maturity, transforming Blue from a specialized M2MGate provisioning tool into a general-purpose infrastructure platform capable of supporting diverse workloads. In the future M2MGateCluster resource itself might be further modularized to support more complex use cases. Furthermore, other kind of processes other than environment provisioning might be supported.

6.3.5 Community and Collaboration

Sustained platform evolution within the organization requires expanding the contributor base. As project teams adopt the platform and develop domain-specific expertise, mechanisms must be established to channel their contributions back into the core platform. Governance models, contribution guidelines, and review processes will need refinement as the platform transitions from a single-maintainer project to a collaborative effort.

6.4 Closing Remarks

Platform engineering is fundamentally about people as much as technology. The true measure of success will be sustained adoption, continuous improvement based on user feedback, and evolution of organizational practices to fully leverage the platform's capabilities. As more team members engage with the platform, contribute enhancements, and integrate it into their workflows, the initial investment in automation and abstraction will multiply in value.

The journey from manual, error-prone provisioning to automated, self-service infrastructure represents significant progress, but it is only the beginning. The platform must continue to evolve, adapting to new cloud providers, accommodating emerging technologies, and responding to changing organizational needs. With ongoing development, expanding the contributor community, and maintaining focus on user needs, the Blue platform can serve as a cornerstone of infrastructure operations for years to come.

Appendix A

Deployment Modernization Survey

This appendix provides an overview of some of the questions in the Deployment Modernization Survey which was used to gather data for this dissertation.

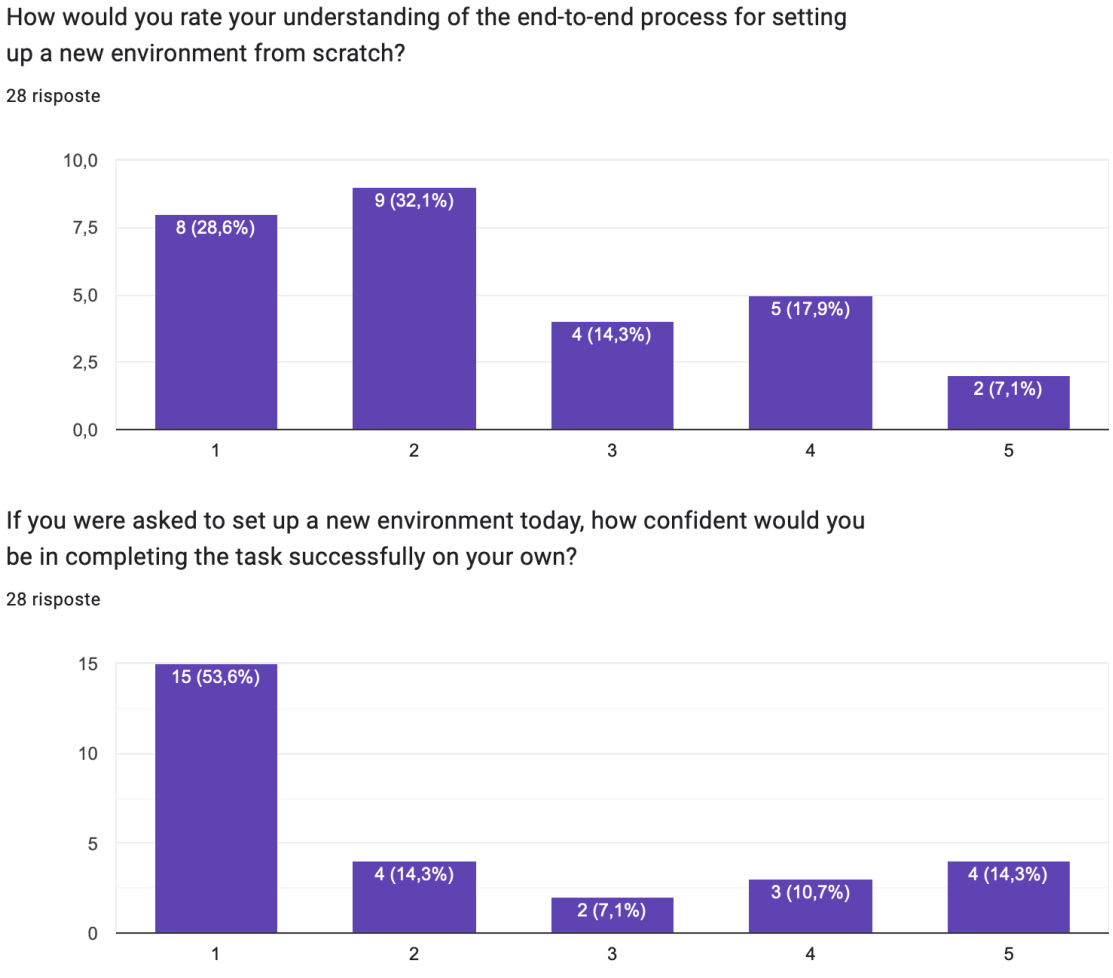
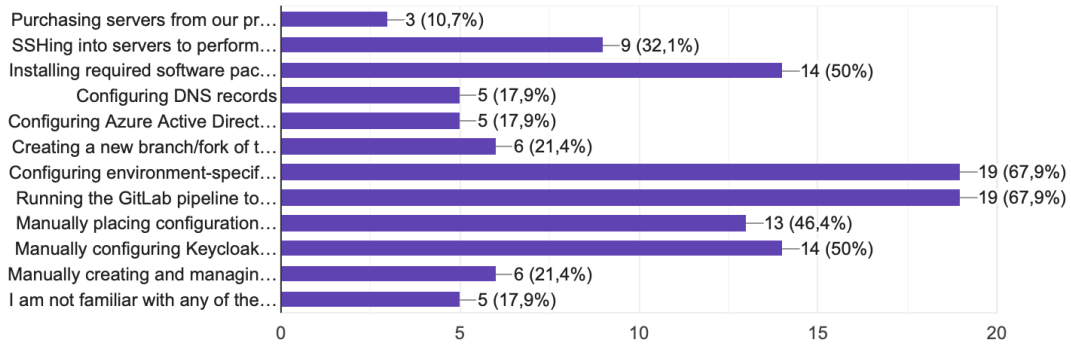


Figure A.1: Deployment Modernization Survey Questions (1/5)

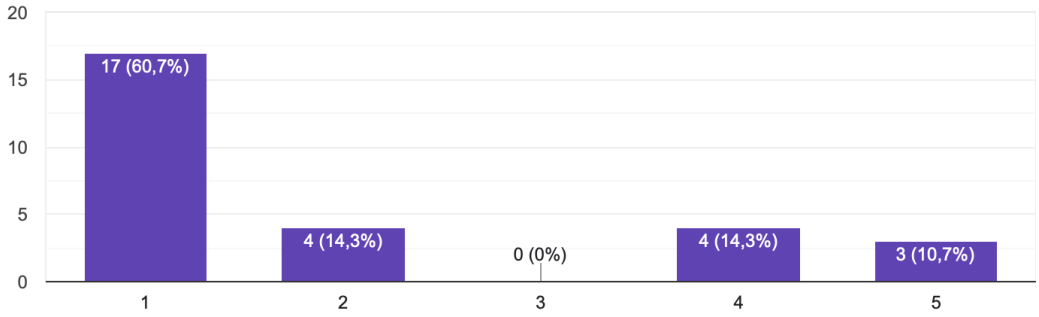
Please check the steps of the current deployment process you are familiar with:

28 risposte



If you were asked to make a change to the the **M2MGate Blueprint** (e.g. adding an Apache server and making it available upon authentication to the internet) and rollout the change in an existing environment, how confident would you be in completing the task successfully on your own?

28 risposte



What do you consider the most significant weak points or challenges in our current deployment process? (Select up to three)

28 risposte

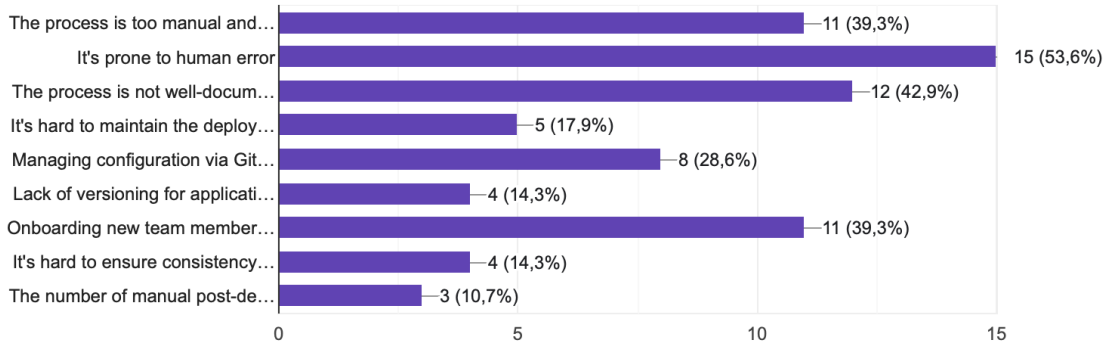
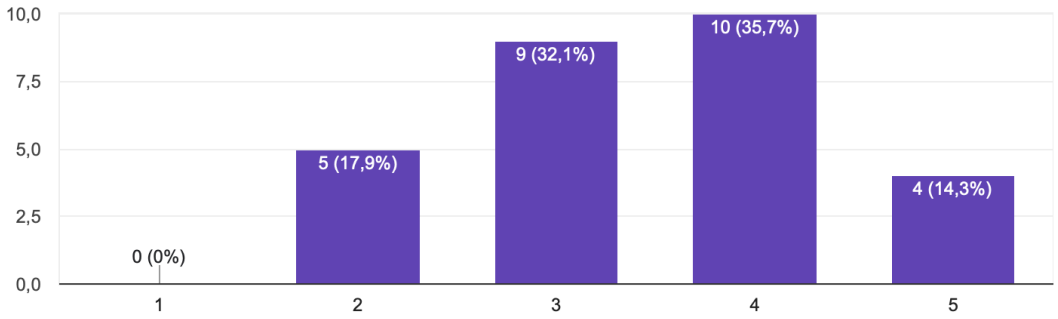


Figure A.2: Deployment Modernization Survey Questions (2/5)

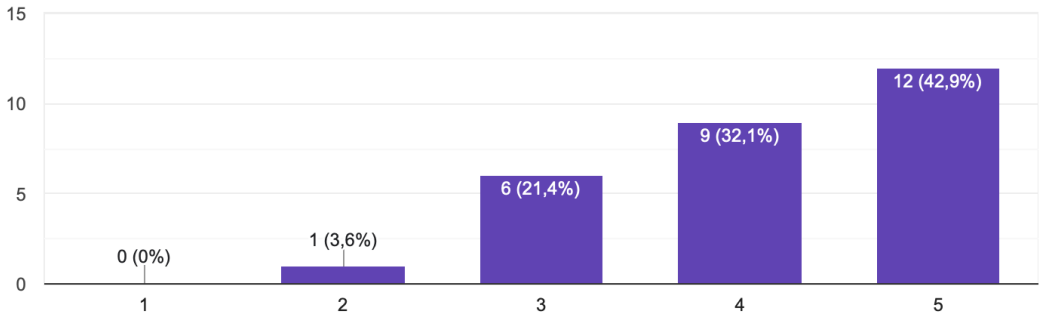
On a scale of 1 to 5, how difficult do you find it to debug deployment issues with the current system?

28 risposte



How valuable would an automated tool that handles the end-to-end deployment process be for you and your team?

28 risposte



Which of the following potential benefits of the new tool would be most important to you? (Select up to three)

28 risposte

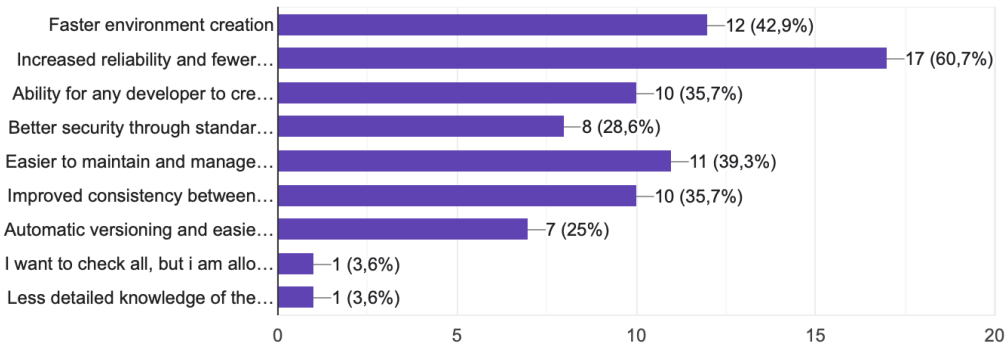
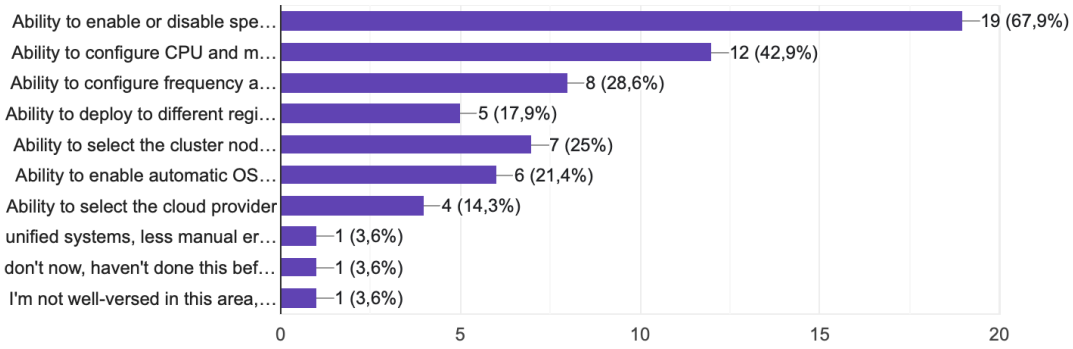


Figure A.3: Deployment Modernization Survey Questions (3/5)

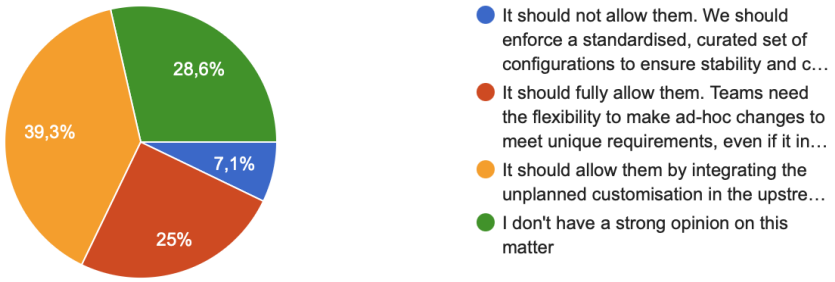
When creating a new environment, which of the following customizations are most important to you? (Select all that apply)

28 risposte



Should the new tool allow for unplanned customisations to the environment and application stack?

28 risposte



When it comes to customizing the application stack, there's often a trade-off between simplicity and flexibility. Which of the following statements best reflects your opinion?

28 risposte

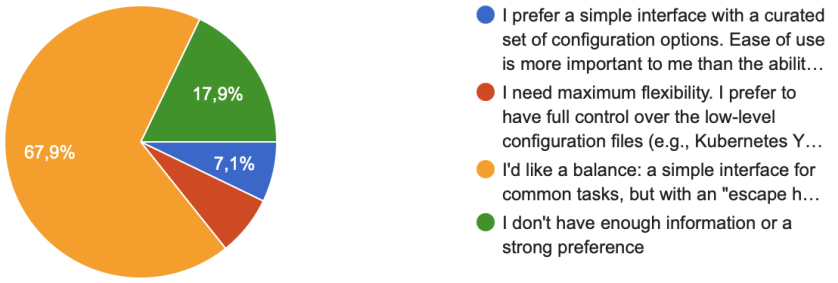
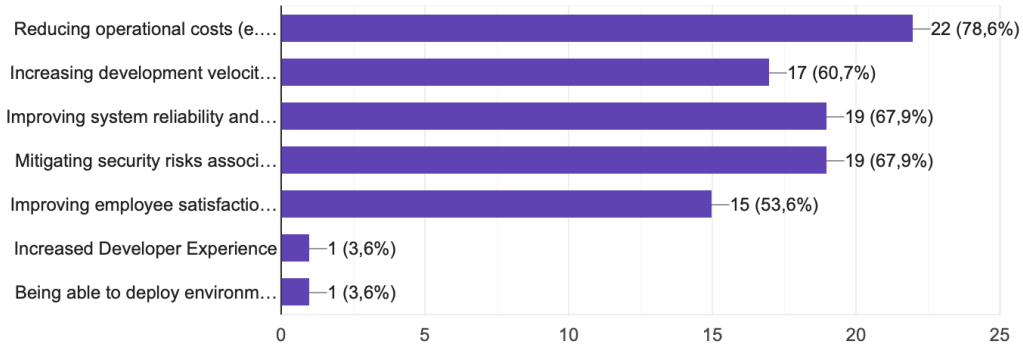


Figure A.4: Deployment Modernization Survey Questions (4/5)

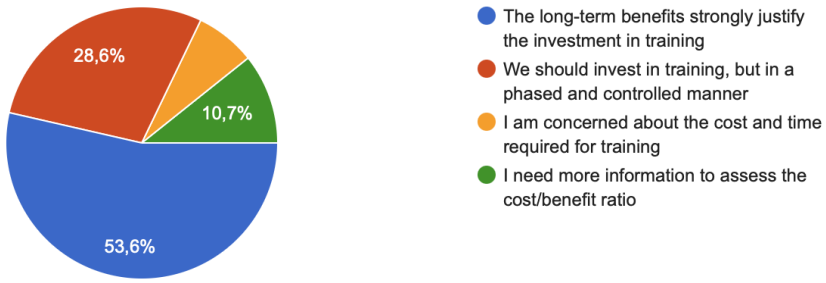
From your perspective, what is the primary business driver for automating our deployment processes? (Select all that apply)

28 risposte



The proposed solution involves adopting new technologies like Kubernetes and Terraform. What are your thoughts on the cost of training personnel versus the long-term benefits of automation?

28 risposte



How do you foresee this automation impacting your team's budget and resource allocation? (Select all that apply)

28 risposte

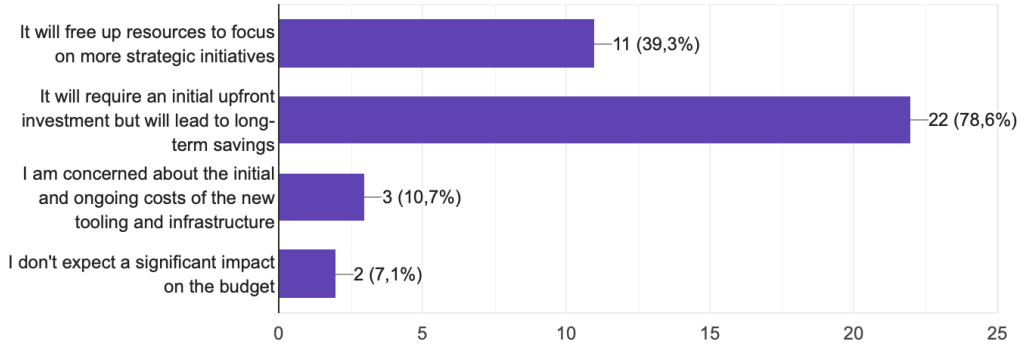


Figure A.5: Deployment Modernization Survey Questions (5/5)

Appendix B

M2MGateCluster Specification

This appendix provides a detailed specification of the M2MGateCluster resource definition, some of the fields or their values are omitted for brevity.

M2MGateCluster resource `spec` fields

Fields marked with * are required.

Field	Type	Description	Default
`baseDomain` *	string	The base domain for the cluster (e.g., example.m2mgate.cloud).	`null`
`provider` *	string	Identifies the cloud provider.	`"hetzner"`
`nodeSize`	string	Specifies the size of worker nodes: medium (16GB RAM), large (32GB RAM).	`"medium"`
`auth`	object	Authentication configuration.	`{}`
`broker`	object	Message broker configuration.	`{}`
`database`	object	Database configuration.	`{}`
`m2mgate`	object	M2MGate services configuration.	`{}`
`monitoring`	object	Monitoring stack configuration.	`{}`

Table B.1: M2MGateCluster resource `spec` fields.

M2MGateCluster `spec.auth` fields

Field	Type	Description	Default
`enableInsideM2MAzureAD`	bool	Enables Single Sign-On with INSIDE M2M Azure AD.	`true`

Table B.2: M2MGateCluster `spec.auth` fields.

M2MGateCluster `spec.broker` fields

Notation `"-"` is used to omit values.

Field	Type	Description	Default
`enabled`	bool	Enables broker deployment. MAD service requires this to be enabled.	`true`
`kafka.storageSizeGB`	int	Size in GiB for Kafka storage.	`10`
`kafka.resources`	object	Resources requests and limits for Kafka nodes.	–

Table B.3: M2MGateCluster `spec.broker` fields.

M2MGateCluster `spec.database` fields

Notation `"-"` is used to omit values.

Field	Type	Description	Default
`mysql.storageSizeGB`	int	Size in GiB for MySQL storage.	`100`
`mysql.resources`	object	Resources requests and limits for MySQL nodes.	–
`backup.schedule`	array	Backup schedule configuration.	–

Table B.4: M2MGateCluster `spec.database` fields.

M2MGateCluster `spec.monitoring` fields

Notation `"-"` is used to omit values.

Field	Type	Description	Default
`enabled`	bool	Enables monitoring stack.	`true`
`grafana.exposeDashboard`	bool	Exposes Grafana dashboard to the internet.	`false`
`alertManager.alertsMailList`	string	Comma-separated list of email addresses for alerts.	–
`alertManager.enableMSTeamsAlerts`	bool	Enables alerts delivery to Microsoft Teams.	`false`
`loki.storageSizeGB`	int	Size in GiB for Loki storage.	`10`

Table B.5: M2MGateCluster `spec.monitoring` fields.

M2MGateCluster `spec.m2mgate` services

All services support `env`, `resources`, and `tagRef` fields. Notation `"-"` is used to omit values.

Service	Enabled	Notable Configuration Options	Default
`cascade`	always	`connectionType.devices`: type of connection for devices [Plain, TLS, mTLS].	`mTLS`
`coreService`	always		–
`portal`	`true`	`enableGoogleAPI`, `urlSubdomain` for custom subdomain configuration.	`false`, `""`
`containerService`	`true`		–
`cryptoService`	`true`		–
`distributionService`	`true`		–
`dtoolsService`	`true`		–
`geocoderService`	`true`	`enableGoogleAPI`: enables Google Geocoding API integration.	`false`
`madService`	`true`		–
`messageService`	`true`	`enableEsendex`: enables Esendex SMS provider integration.	`false`
`simInfoService`	`true`		–
`tenantService`	`true`		–
`vpngateService`	`true`	`enablePPTP`, `numberOfVPNs`: configures VPN instances [0-5].	`false`, `2`
`licenseStatusService`	`false`	`mailRecipients`: email addresses for license status notifications.	–
`placeholderService`	`false`	`serviceKey`: service key for the placeholder service.	`"4096"`

Table B.6: M2MGateCluster `spec.m2mgate` services.

Appendix C

M2MGateCluster Composition

This appendix provides a more detailed view of the output resources obtained using the default values for the M2MGateCluster Composition.

```
M2MGateCluster m2mgate-cluster
├─M2MGateClusterAuth cluster-auth
│ ├─Object auth-k8s-keycloak
│ ├─Object auth-k8s-keycloak-ingress
│ ├─Object auth-k8s-keycloak-secret
│ ├─Object auth-k8s-keycloak-pvc
│ ├─Object auth-k8s-realm-infrastructure
│ ├─Object auth-k8s-realm-m2mgate
│ ├─ProviderConfig auth-infra-provider-config
│ ├─Workspace auth-infra-azuread
│ └─Usage auth-usage-* (42 resources)
├─M2MGateClusterBootstrap cluster-bootstrap
│ ├─Release bootstrap-k8s-controller-dns-acme-webhook
│ ├─Release bootstrap-k8s-controller-dns-external-dns
│ ├─Release bootstrap-k8s-controller-operator-kafka-release
│ ├─Release bootstrap-k8s-controller-reloader-release
│ ├─Release bootstrap-k8s-controller-storage-csi-driver-nfs
│ ├─Object bootstrap-k8s-controller-cert-manager-dashboard
│ ├─Object bootstrap-k8s-controller-operator-keycloak-*
│ ├─Object bootstrap-k8s-controller-operator-mysql-*
│ ├─Object bootstrap-k8s-controller-storage-nfs-*
│ ├─Object bootstrap-k8s-controller-traefik-*
│ ├─Object bootstrap-k8s-namespace-* (6 namespaces)
│ ├─ProviderConfig bootstrap-k8s-provider-config-helm
│ ├─ProviderConfig bootstrap-k8s-provider-config-kubernetes
│ ├─Workspace bootstrap-infra-cluster
│ └─Usage bootstrap-usage-* (4 resources)
├─M2MGateClusterBroker cluster-broker
│ ├─Object broker-dashboard-kafka-*
│ ├─Object broker-kafka
│ ├─Object broker-kafka-node-pool
│ ├─Object broker-redpanda-*
│ ├─Object broker-schema-registry-*
│ └─Usage broker-usage-* (12 resources)
├─M2MGateClusterDatabase cluster-database
│ ├─Object database-dashboard-mysql-*
│ ├─Object database-mysql-cluster
│ ├─Object database-mysql-credentials
│ └─Usage database-usage-* (7 resources)
├─M2MGateClusterMonitoring cluster-monitoring
│ ├─Release monitoring-logs
│ ├─Release monitoring-logs-loki
│ ├─Release monitoring-metrics
│ ├─Object monitoring-dashboard-loki-*
│ ├─Object monitoring-monitor-*
│ ├─Object monitoring-rule-*
│ └─Usage monitoring-usage-* (23 resources)
├─M2MGateClusterSecret cluster-secret
│ ├─ExternalSecret secret-external
│ ├─ExternalSecret secret-generated
│ └─SecretStore secret-gitlab-store
├─M2MGateClusterService cluster-service
│ ├─Object service-cascade-* (StatefulSets, PVCs)
│ ├─Object service-certificate-*
│ ├─Object service-core-* (Config, Deployment, Ingress)
│ ├─Object service-crypto-*
│ ├─Object service-distribution-*
│ ├─Object service-dtools-*
│ ├─Object service-geocoder-*
│ ├─Object service-mad-*
│ ├─Object service-message-*
│ ├─Object service-portal-*
│ ├─Object service-sim-info-*
│ ├─Object service-tenant-*
│ ├─Object service-vpngate-*
│ ├─NopResource service-*-nop
│ └─Usage service-usage-* (84 resources)
├─NopResource cluster-auth-nop
└─NopResource cluster-database-nop
```

Figure C.1: M2MGateCluster Composition: Hierarchical structure of resources

Bibliography

[1] CNCF TAG App Delivery. *CNCF Platforms White Paper*. Tech. rep. White paper on platform engineering for cloud-native computing. Cloud Native Computing Foundation, 2024. URL: https://tag-app-delivery.cncf.io/whitepapers/platforms/.

[2] CNCF TAG App Delivery. *Platform Engineering Maturity Model*. Referenced in CNCF Platforms White Paper. URL: https://tag-app-delivery.cncf.io/.

[3] *GitOps Principles and Glossary*. https://github.com/open-gitops/documents. Accessed: 2025-07-29. Release version: 1.0.0. Authors: https://github.com/open-gitops/documents/blob/main/reference/authors-v1.0.0.md. 2021.

[4] Cloud Native Computing Foundation. *Kubernetes*. https://kubernetes.io/. Container orchestration platform.

[5] Cloud Native Computing Foundation. *CNCF Annual Survey 2024*. Tech. rep. Accessed: August 13, 2025. Cloud Native Computing Foundation, Dec. 2024. URL: https://www.cncf.io/wp-content/uploads/2025/04/cncf_annual_survey24_031225a.pdf (visited on 08/13/2025).

[6] HashiCorp. *What is Terraform?* https://developer.hashicorp.com/terraform/intro. Accessed: 2025-08-13. HashiCorp, 2024. URL: https://developer.hashicorp.com/terraform/intro.

[7] Pulumi. *Managing state & backend options*. Pulumi Corporation. 2024. URL: https://www.pulumi.com/docs/iac/concepts/state-and-backends/ (visited on 08/13/2025).

[8] Crossplane Authors. *What's Crossplane?* Official documentation for Crossplane, a control plane framework for platform engineering. The Linux Foundation. 2025. URL: https://docs.crossplane.io/latest/whats-crossplane/ (visited on 11/24/2025).

[9] GitLab Inc. *Infrastructure as Code with OpenTofu and GitLab*. https://docs.gitlab.com/user/infrastructure/iac/. Accessed: January 30, 2026. GitLab Inc., 2024.

[10] Crossplane Authors. *Community Extension Projects*. Version 2.1. Documentation for community-maintained Crossplane providers, functions, configurations, and utilities. The Linux Foundation. 2025. URL: https://docs.crossplane.io/v2.1/learn/community-extension-projects/ (visited on 11/24/2025).

[11] The Crossplane Authors. *What is Upjet?* Documentation for Upjet, a framework to build Crossplane providers with code generation pipeline and generic reconciler implementation. The Linux Foundation. 2023. URL: https://github.com/crossplane/upjet/blob/main/docs/README.md.

[12] Upbound. *provider-opentofu. OpenTofu Crossplane Provider*. Version 1.0.1. A Crossplane provider that can run Terraform code (HCL) using the OpenTofu fork. 2025. URL: https://github.com/upbound/provider-opentofu (visited on 11/24/2025).

[13] GitLab Components. *OpenTofu CI/CD Catalog project*. Version Latest. GitLab CI/CD component for OpenTofu infrastructure automation. 2024. URL: https://gitlab.com/components/opentofu.

[14] Jacco Taal. *Why Helm's Design is Flawed*. Blog post critiquing Helm's templating system and design decisions in Kubernetes. DevOps.dev. June 2023. URL: https://blog.devops.dev/why-helms-design-is-flawed-a66c07c2e9a1 (visited on 11/24/2025).

[15] KCL Authors. *Introduction to KCL*. Official documentation introducing KCL, an open-source configuration and policy language. Last updated March 30, 2025. Cloud Native Computing Foundation. 2025. URL: https://www.kcl-lang.io/docs/user_docs/getting-started/intro (visited on 11/24/2025).

[16] The Nushell Project. *The Nushell Book*. Official documentation for Nushell, a cross-platform shell and modern programming language that brings the Unix philosophy of shells to modern development. 2025. URL: https://www.nushell.sh/book/.

[17] *GitLab-managed Terraform/OpenTofu state. Managing infrastructure state files across teams*. Official GitLab documentation covering secure state file management, versioning, access control, and CI/CD integration for Terraform and OpenTofu. GitLab Inc. 2024. URL: https://docs.gitlab.com/user/infrastructure/iac/terraform_state/ (visited on 08/15/2025).

[18] kube-hetzner. *terraform-hcloud-kube-hetzner. Optimized and Maintenance-free Kubernetes on Hetzner Cloud in one command!* Version 2.18.1. 2025. URL: https://github.com/kube-hetzner/terraform-hcloud-kube-hetzner (visited on 08/15/2025).

[19] openSUSE Contributors. *Portal:MicroOS - openSUSE Wiki*. Modern Linux operating system designed for container hosts and edge devices. openSUSE Project. Sept. 2024. URL: https://en.opensuse.org/Portal:MicroOS.

[20] The FluxCD Authors. *Ways of structuring your repositories*. FluxCD Documentation. Last modified 2024-05-13. Accessed January 30, 2026. 2024. URL: https://fluxcd.io/flux/guides/repository-structure/.

[21] The external-secrets Authors. *External Secrets Operator*. Kubernetes operator that integrates external secret management systems. The Linux Foundation. 2025. URL: https://external-secrets.io/latest/.

[22] Cloud Native Computing Foundation. *Cloud Native Computing Foundation Announces Graduation of Crossplane*. Official announcement of Crossplane's graduation within the CNCF project ecosystem. Cloud Native Computing Foundation. Nov. 2025. URL: https://www.cncf.io/announcements/2025/11/06/cloud-native-computing-foundation-announces-graduation-of-crossplane/ (visited on 11/28/2025).

[23] K3s Project Authors. *K3s - Lightweight Kubernetes*. Lightweight Kubernetes distribution documentation. The Linux Foundation. 2025. URL: https://docs.k3s.io (visited on 08/07/2025).

[24] Hetzner Cloud. *csi-driver. Kubernetes Container Storage Interface driver for Hetzner Cloud Volumes*. Version 2.18.2. CSI driver enabling ReadWriteOnce volumes in Kubernetes 1.19 or newer. 2025. URL: https://github.com/hetznercloud/csi-driver (visited on 12/02/2025).

[25] kubernetes-csi. *csi-driver-nfs. NFS CSI driver for Kubernetes*. Version 4.12.1. Container Storage Interface driver for NFS server on Linux nodes, supporting Kubernetes 1.21+. 2025. URL: https://github.com/kubernetes-csi/csi-driver-nfs (visited on 12/02/2025).

[26] bitchecker. *Is the project still alive?* GitHub Issue #4298, gluster/glusterfs repository. Discussion regarding the future of GlusterFS following Red Hat Gluster Storage EOL announcement. Jan. 2024. URL: https://github.com/gluster/glusterfs/issues/4298 (visited on 12/02/2025).

[27] kubernetes-sigs. *external-dns. Configure external DNS servers dynamically from Kubernetes resources*. Version 0.20.0. Kubernetes controller that synchronizes exposed Kubernetes Services and Ingresses with DNS providers. Kubernetes Special Interest Groups, 2025. URL: https://github.com/kubernetes-sigs/external-dns (visited on 12/02/2025).

[28] Hetzner. *cert-manager-webhook-hetzner. cert-manager ACME webhook for Hetzner*. Version 0.6.2. Webhook that creates DNS entries in Hetzner DNS API to solve DNS01 challenges for cert-manager. 2025. URL: https://github.com/hetzner/cert-manager-webhook-hetzner (visited on 12/02/2025).

[29] Strimzi Authors. *Strimzi Documentation*. https://strimzi.io/documentation/. Documentation for Strimzi 0.47.0, which supports Kafka Bridge 0.32.0. The Linux Foundation, 2025. URL: https://strimzi.io/documentation/ (visited on 08/11/2025).

[30] Percona LLC. *Percona Operator for MySQL Based on Percona XtraDB Cluster*. Version 1.17.0. Percona LLC. 2025. URL: https://docs.percona.com/percona-operator-for-mysql/pxc/ (visited on 08/11/2025).

[31] Prometheus Operator Contributors. *kube-prometheus*. GitHub repository. Use Prometheus to monitor Kubernetes and applications running on Kubernetes. Prometheus Operator, 2025. URL: https://github.com/prometheus-operator/kube-prometheus.

[32] *Prometheus Monitoring Mixins*. A collection of Grafana dashboards and Prometheus rules and alerts for monitoring various systems. Monitoring Mixins Community. 2024. URL: https://monitoring.mixins.dev.

[33] Bilgin Ibryam and Roland Huß. *Kubernetes Patterns*. Intermediate to advanced level. Sebastopol, CA: O'Reilly Media, Inc., Apr. 2019, p. 266. ISBN: 9781492050278. URL: https://www.oreilly.com/library/view/kubernetes-patterns/9781492050278/.

[34] Crossplane Contributors. *function-kcl. Crossplane Composition Functions using KCL Programming Language*. Version 0.11.6. A Crossplane composition function that allows using KCL to define and compose Kubernetes resources, supporting features like extra resources, connection details, conditions, and events. Crossplane Contrib, 2025. URL: https://github.com/crossplane-contrib/function-kcl (visited on 12/04/2025).

[35] KCL Authors. *krm-kcl. Kubernetes Resource Model KCL Specification and Integrations*. Version 0.12.1. KRM KCL function SDK containing a KRM KCL spec and interpreter to run KCL codes to generate, mutate or validate Kubernetes resources. Integrates with Kubectl, Kustomize, Helm, Helmfile, Crossplane, KPT, and more. Cloud Native Computing Foundation, 2025. URL: https://github.com/kcl-lang/krm-kcl (visited on 12/04/2025).

[36] maykonlsf. *semver-cli. An easy to use CLI tool to manage your project versions according to Semantic Versioning*. Version 1.1.2. CLI tool for semantic versioning management supporting alpha, beta, release candidate, and release version workflows. 2025. URL: https://github.com/maykonlsf/semver-cli (visited on 12/09/2025).

[37] GitLab. *glab. A GitLab CLI tool bringing GitLab to your command line*. Official GitLab CLI tool for interacting with GitLab from the command line, supporting merge requests, issues, pipelines, and more. 2025. URL: https://gitlab.com/gitlab-org/cli (visited on 12/09/2025).

[38] Tom Christie and MkDocs Team. *MkDocs. Project documentation with Markdown*. A fast, simple static site generator geared towards building project documentation. Documentation source files are written in Markdown and configured with a single YAML file. 2025. URL: https://www.mkdocs.org (visited on 12/09/2025).

[39] Jim Porter. *mike. Manage multiple versions of your MkDocs-powered documentation via Git*. Version 2.1.3. Python utility for deploying multiple versions of MkDocs documentation to a Git branch, suitable for hosting on GitHub Pages. 2024. URL: https://github.com/jimporter/mike (visited on 12/09/2025).

[40] Joy Jedidja Ndjama. *Auto deploy mkdocs with versionning using mike on gitlab pages*. Stack Overflow. Question and answer on deploying versioned MkDocs documentation using mike on GitLab Pages with CI/CD. June 2022. URL: https://stackoverflow.com/questions/72467521/auto-deploy-mkdocs-with-versionning-using-mike-on-gitlab-pages (visited on 12/16/2025).

[41] HashiCorp. *Packer - Build Automated Machine Images*. Official documentation for Packer, a tool that creates identical machine images for multiple platforms from a single source template. HashiCorp. 2025. URL: https://developer.hashicorp.com/packer (visited on 12/10/2025).

[42] patpicos. *cannot run Function - rpc error: code = DeadlineExceeded desc = context deadline exceeded*. GitHub Issue #211, crossplane-contrib/function-kcl repository. Bug report regarding context deadline exceeded errors when running Crossplane KCL functions with many resources under management. Dec. 2024. URL: https://github.com/crossplane-contrib/function-kcl/issues/211 (visited on 12/12/2025).