

**DANIEL
SCHATZMANN**
daniel.schatzmann@ost.ch



**VALENTINO
DILLER**
valentino.diller@ost.ch



**ROMAN
WEBER**
roman.weber@ost.ch

BACHELOR THESIS – SPRING TERM 2026

gNMIc Operator

Development of a mechanism for target auto-discovery

PROJECT ADVISOR

Severin Dellsperger
severin.dellsperger@ost.ch

CO-ADVISOR

Jan Untersander
jan.untersander@ost.ch

EXTERNAL CO-EXAMINER

John Howard
john.howard@proton.ch

INTERNAL CO-EXAMINER

Prof. Dr. Frieder Loch
frieder.loch@ost.ch

Abstract

With the gRPC Network Management Interface (gNMI) and its collector, the gNMI CLI client (gNMlc), network telemetry is moving from pull-based to streaming-based data collection. The gNMlc Operator brings this technology to Kubernetes, providing declarative configuration management and high availability. While the operator addresses several limitations of standalone gNMlc deployments, target management remains manual and can lead to inconsistencies between the desired and actual system state. In the context of gNMlc, a target is a network device to monitor. This thesis addresses this gap by designing and implementing an automated target discovery mechanism that synchronizes targets with an external source of truth.

To understand real-world practices of network and monitoring engineers, a survey was conducted to identify commonly used sources of truth and operational workflows. Based on these findings, a hybrid solution that combines periodic synchronization with event-driven updates was designed. A provider component periodically retrieves the complete set of targets from the external system to ensure consistency, while an API server receives incremental changes via webhooks. Both mechanisms forward the collected information to a client component, which processes the data and applies the resulting state to the Kubernetes cluster using Custom Resources.

The solution ensures continuously updated target information without manual intervention. It integrates seamlessly with the declarative, reconciled model of the gNMlc Operator and eliminates the need for process restarts, thereby preventing service disruption. Usability tests confirmed that the product is suitable for technical users. Developed in close collaboration with Nokia, it is integrated into the upstream gNMlc Operator project, helping to close the gap between manual and automated target management observed in survey responses. While many use cases can be fulfilled using HTTP polling, the architecture allows for native implementations of other providers such as NetBox or Consul.

Contents

1. Introduction	1
1.1. Problem Statement	1
1.2. Project Goal	1
1.3. Context & Background	2
1.4. Thesis Structure	3
2. Requirement Analysis	4
2.1. Use Case Model	4
2.1.1. Use Case Description	5
2.1.2. Use Case Diagram	9
2.2. Functional Requirements	9
2.3. Non-Functional Requirements	11
3. Research	14
3.1. Market Research	14
3.2. Target Discovery	17
3.2.1. Discovery Mechanisms	18
3.2.2. Summary of Discovery Approaches	18
3.3. Target Lifecycle	19
3.3.1. Current Implementation	20
3.3.2. TargetSource Concept	20
3.3.3. Target Life Cycle Management	21
3.4. Components	22
3.5. Findings	23
4. Design & Implementation	24
4.1. Architecture	24
4.1.1. Context	24
4.1.2. Building Block View	24
4.2. TargetSource Implementation	28
4.2.1. Custom Resource Definition	28
4.2.2. Reconciliation Logic	29
4.2.3. Goroutines & Subprocesses	30
4.2.4. Observability & Status Management	33
4.2.5. Failure Recovery & Consistency	34
4.3. REST API Interface	35
4.3.1. Target Struct	36
4.3.2. Operational Flow of applyTargets	37
4.3.3. Authentication	37
4.4. HTTP Provider	38
4.4.1. Architecture and Execution Model	39
4.4.2. Polling and Synchronization	39
4.4.3. HTTP Communication	39

4.4.4. Response Mapping	40
4.4.5. Pagination	40
4.4.6. Error Handling and Resilience	41
4.5. Testing & Validation	41
4.5.1. Unit Tests	41
4.5.2. Integration Tests	41
4.5.3. Usability Test	42
4.6. gNMIc Operator Documentation	44
5. Validation	45
5.1. Functional Validation	45
5.2. Non-Functional Validation	45
5.2.1. Performance Efficiency	46
5.2.2. Network Efficiency	46
5.2.3. Update Latency	46
5.2.4. Scalability	47
5.2.5. Usability	47
5.3. Limitations	48
6. Conclusion & Outlook	50
6.1. Outcome	50
6.2. Findings	51
6.3. Future Work	51
Bibliography	52
List of External Tools and Resources	56
Glossary	57
List of Abbreviations	59
List of Tables	60
List of figures	61
Code Listings	62
Appendix	63
A. Architectural Decision Records	64
ADR-01: Target Discovery Mechanism	64
ADR-02: Target Identification	65
ADR-03: REST API Framework	66
ADR-04: REST API authentication	67
ADR-05: Error Handling	69
ADR-06: Response Mapping	70
ADR-07: NetBox Pull Interface	71
B. Target Discovery Research	73
B.1. Agent-Based Target Discovery	73
B.2. Pull-Based Target Discovery	74

B.3. Push-Based Target Discovery	74
B.4. Hybrid-Based Target Discovery	75
B.5. Source of Truth Compatibility	75
C. Test Protocols	77
TEST-01: Performance and Scalability Validation	77
TEST-02: Reliability and Recovery Validation	79
TEST-03: Update Latency Validation	82
TEST-04: Network Efficiency Validation	84
D. Operator Documentation Pages	87
D.1. TargetSource	87
D.2. HTTP Provider	89
D.3. Push Mode	101
D.4. NetBox Examples	103
D.4.1. Pull with Export Template	103
D.4.2. Pull with REST API	110
D.4.3. Push Mode with Webhook	115
E. Usability Testing	120
E.1. Guide for Usability Tests	120

1. Introduction

This chapter introduces the context and motivation of the thesis. It presents the gNMI CLI client (**gNMIC**) Operator, which serves as the foundation for the work carried out in this project, and outlines the objectives of the proposed extension. Furthermore, it provides the necessary background on network monitoring and telemetry concepts required for understanding the remainder of the thesis. Finally, it concludes with an overview of the document structure.

1.1. Problem Statement

The **gNMIC** Operator, a project initiated by *Karim Radhouani*, is a Kubernetes operator which addresses some of the operational challenges associated with **gNMIC** in a cloud-native way.

At the start of this thesis, the **gNMIC** Operator was at version v0.1.0 and supported declarative configuration of targets, subscriptions, outputs, and dynamic updates [1]. The project documentation already described the operator's core concepts and resource model [2].

Despite these capabilities, telemetry targets¹ still had to be managed manually. In modern network environments, devices are frequently added, removed, or modified, while inventory information is often maintained in dedicated Sources of Truth (**SoTs**) such as NetBox or Infracore. Keeping telemetry target configurations synchronized with these external systems requires continuous manual effort and increases the risk of configuration inconsistencies. As network infrastructure grows in size and complexity, this approach becomes increasingly difficult to scale and maintain.

The absence of an automated mechanism for discovering and synchronizing targets from external **SoTs** therefore represents a limitation of the current **gNMIC** Operator. Addressing this limitation forms the basis of this thesis.

1.2. Project Goal

Based on the identified limitations of manual target management, this thesis aims to extend the **gNMIC** Operator with automated target discovery (further auto-discovery) capabilities. The goal is to automate the management of telemetry targets by dynamically integrating external **SoTs** into the control plane.

The proposed solution will continuously discover network devices and reconcile them into the operator's custom resources, eliminating the need for manual configuration. This approach aims to simplify the **gNMIC** Operator configuration, lowering the hurdles for a switch from traditional Simple Network Management Protocol (**SNMP**) monitoring to the modern alternative.

Further it improves scalability, reduces configuration errors, and enhances the overall reliability of telemetry collection in large and dynamic network environments.

The product developed in the scope of this thesis shall:

- Extend the **gNMIC** Operator project and remain open-source under the Apache-2.0 license [1].
- Allow for target auto-discovery from one concrete **SoT**, while providing the structure for easy extension with other **SoTs**.

¹In the context of gRPC Network Management Interface (**gNMI**) and this thesis, a target denotes a network device (e.g. router or switch) that exposes telemetry data via **gNMI**[1].

1.3. Context & Background

With the evolution of computer networks, network infrastructure has grown significantly in both size and complexity. Large network infrastructure consist of hundreds or even thousands of routers, switches, and other interconnected devices. Failures within these systems can lead to severe service disruptions or even complete outages, making continuous network visibility essential for reliable operation. To address this need, network operators rely on network telemetry to collect and analyze real-time information about the state and performance of their infrastructure.

Network Monitoring

The term network telemetry is often used as a synonym for network monitoring, as many operators primarily associate it with the collection of network metrics. However, network telemetry also introduces a broader set of techniques and mechanisms for collecting, transmitting, processing, and analyzing data generated by network devices and protocols [3]. Its purpose is not only to monitor the current state of a network, but also to provide detailed and structured insights into network behavior, performance, and health. This is achieved with telemetry data such as metrics, which are quantitative measurements of operational characteristics including interface utilization, latency, packet loss, CPU utilization, and memory consumption.

Network telemetry includes the acquisition of data from routers, switches, servers, and other network components through mechanisms such as polling, streaming subscriptions, event notifications, and flow-based exports [3]. The collected information may include interface statistics, routing states, latency, packet loss, bandwidth utilization, and protocol-specific data. This data can then be processed and analyzed to detect faults, identify anomalies, optimize performance, and support automated network operations.

SNMP - the Golden Standard?

Traditionally, network telemetry has been implemented using the protocol SNMP, standardized in RFC 1157, which enables network management systems to retrieve operational data from network devices through periodic polling and receive asynchronous event notifications via SNMP traps [4]. Due to its simplicity, interoperability, and great vendor support, SNMP has remained the de facto industry standard for network monitoring for decades.

Despite its widespread adoption, the polling-based architecture of SNMP comes with several limitations in modern large-scale environments:

- Inefficient data collection as routers only return as many columns as fit into the package [5].
- Unoptimized data aggregation on routers leading to computational overhead [5].
- *“The more pollers you have, the slower the SNMP response.”* [5].
- Difficulties in scaling efficiently with rapidly growing infrastructure [6].
- Pull-based approach can’t account for router workload.

gRPC Network Management Interface

To address the shortcomings of pull-based methods, modern telemetry approaches increasingly adopt streaming-based models. Protocols such as gNMI, built on top of the gRPC² framework,

²The original and widely known expansion of gRPC is *Google Remote Procedure Call*. The project later adopted gRPC as a standalone name and humorously changes the meaning of the g with each version release.

enable structured and event-based telemetry streaming with improved flexibility and scalability. The advantages of **gNMI** are:

- CPU-friendly subscription-based mode, that allows the router to decide when to push data [7].
- Reliable delivery by using gRPC and Transmission Control Protocol (**TCP**) [8].
- Yet Another Next Generation (**YANG**)-model, schema-based telemetry improves data consistency across vendors [7].

gNMI CLI Client

At the basis, **gNMIC**³ is a command-line client for **gNMI**. It supports the core **gNMI** operations *Capabilities*, *Get*, *Set*, and *Subscribe*. Beyond interactive client usage, it can also run a **gNMI** server, which acts as a telemetry collector and forwards collected data to different output backends [9]. Despite the advantages of **gNMIC**, operational challenges still remain, hampering a wide-spread adoption in the networking community.

The biggest challenges lie within the configuration of **gNMIC**. Configuration files can get huge and updating a network device requires manual changes in **gNMIC**, making the configuration error-prone. On top of that, **gNMIC** isn't highly available. For every config change, the gNMI CLI client needs to be reconciled manually, and during this process, no metrics can be collected, resulting in a loss of data [2].

To deal with these challenges, the **gNMIC Operator**⁴ was brought to life. This is an implementation of **gNMIC** as a Kubernetes operator. Kubernetes solves some of the challenges from **gNMIC**, mainly high availability, a declarative configuration and a constant reconciliation cycle.

However, one big inconvenience remains: The management of targets is tedious since each target has to be created manually, which generates an operational overhead when the network consists of thousands of devices.

1.4. Thesis Structure

The remainder of this thesis is structured as follows.

Chapter 2 derives the functional and non-functional requirements from the identified use cases and defines the expected behavior of the system.

Chapter 3 investigates existing approaches to target discovery, lifecycle management, and system integration. The findings are used to derive the design decisions that guide the implementation.

Chapter 4 presents the architecture, design, and implementation of the developed solution, including the target discovery mechanism, interface implementations, and validation methodology.

Chapter 5 evaluates the implemented solution against the previously defined requirements and discusses its limitations.

Finally, Chapter 6 summarizes the results of the thesis and outlines potential directions for future work.

³<https://github.com/openconfig/gnmic>

⁴<https://github.com/gnmic/operator>

2. Requirement Analysis

This chapter analyzes the system requirements by defining user interactions through use cases and specifying functional requirements derived from them. The requirements follow the use case methodology at the level of Elementary Business Processes (**EBP**).

To assess the requirements, the following scenario is considered:

A company adopts the gNMIc Operator for cloud-native lifecycle management to monitor network devices [2]. While the operator provides scalability and automated distribution of targets, manually defining devices as targets is time-consuming and error-prone. Over time, this leads to inconsistencies between the configured state and the actual network inventory. The system is extended with a discovery mechanism that allows automated synchronization between external inventory sources and the operator.

2.1. Use Case Model

The use case model structures the functional requirements around actor goals and their interactions with the System under Development (**SuD**). Each use case represents a complete unit of work performed by an actor.

The **SuD** is the gNMIc Operator extended with discovery capabilities.

Three actors interact with the system. The **Network Engineer** acts as the primary actor he configures and operates the system. The **Discovery Provider** is a supporting actor that supplies information about network devices. The **Kubernetes Application Programming Interface (API)** is a supporting technical actor that provides services for managing system resources.

The primary objective of the system is to maintain a consistent representation of network devices as targets according to the **SoT**.

Based on the identified actor goals, the use cases listed in Table 1 define the functional scope of the system.

ID	Use Case	Primary Actor	Goal
UC1	SynchronizeTargets	Network Engineer	Maintain consistency between devices provided by the Discovery Provider and their corresponding targets in the system
UC2	ProcessEventBasedTargetUpdate	Discovery Provider	Notify the system about changes to device state
UC3	ConfigureTargetDiscovery	Network Engineer	Define target discovery behavior
UC4	ObserveSystemState	Network Engineer	Obtain insight into system state and behavior

Table 1: Use Case Model

2.1.1. Use Case Description

The following sections describe the identified use cases. Each use case focuses on the goal of the primary actor and describes the interaction with the system while avoiding implementation details.

UC1 SynchronizeTargets

This use case ensures that the system maintains consistency between the Discovery Provider and its internal representation of targets.

Actors

- Primary Actor: Network Engineer
- Supporting Actors: Discovery Provider, Kubernetes API

Goal

- Ensure that discovered devices are accurately reflected as consistent targets within the system.

Preconditions

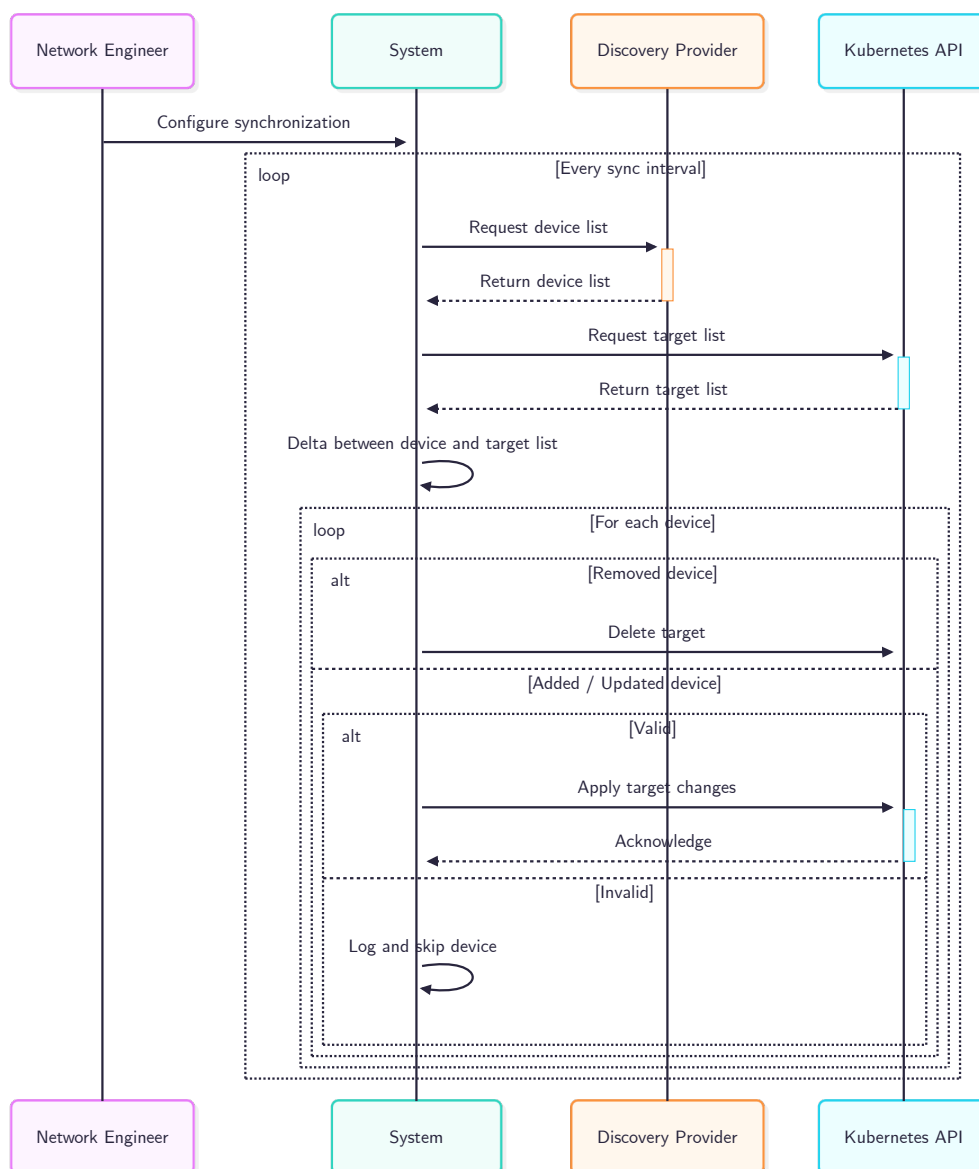


Figure 2: Sequence diagram for UC1

- A Discovery Provider is configured and reachable.

Success Guarantee

- All relevant devices are represented exactly once as targets without inconsistencies.

Main Success Scenario

The main success scenario is illustrated in Figure 2.

Alternative Flows

- If the Discovery Provider is temporarily unavailable, the system postpones the synchronization and retries at a later point in time.
- If individual devices cannot be processed, they are skipped and recorded for later analysis.

UC2 ProcessEventBasedTargetUpdate

This use case describes how the system reacts to updates from the Discovery Provider.

Actors

- Primary Actor: Discovery Provider
- Supporting Actor: Kubernetes API

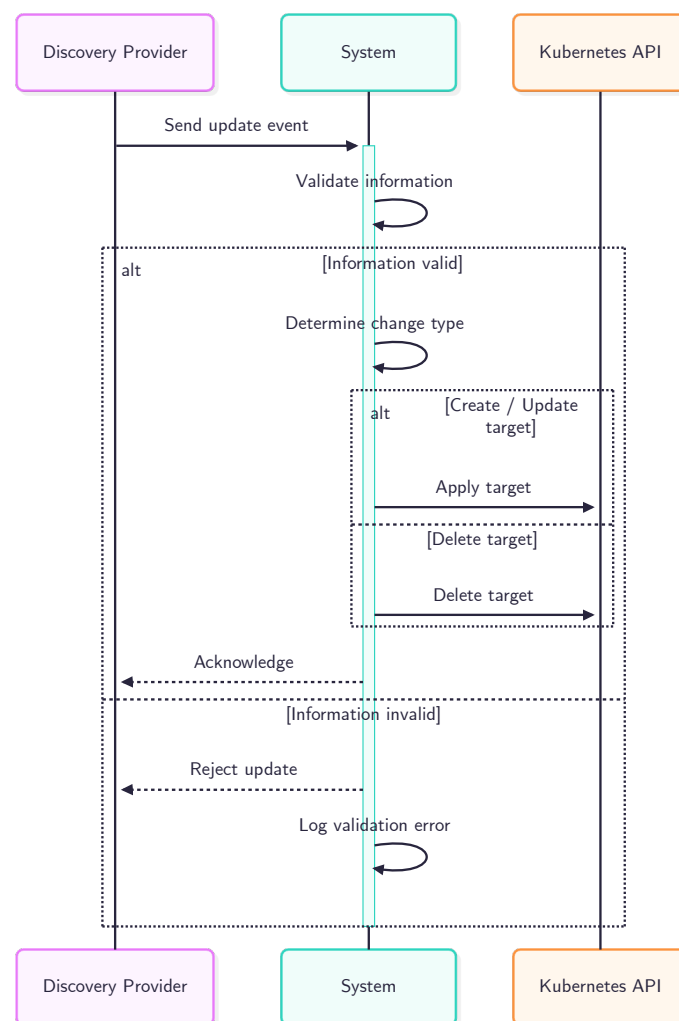


Figure 3: Sequence diagram for UC2

Goal

- Notify the system about changes in device state.

Preconditions

- Update events are provided in a compatible format.

Success Guarantee

- The affected targets reflect the latest device state.

Main Success Scenario

The main success scenario is illustrated in Figure 3.

Alternative Flows

- If events are delayed or missed, inconsistencies are resolved during subsequent synchronization (UC1), provided that synchronization is configured.

UC3 ConfigureTargetDiscovery

This use case describes how the Network Engineer configures how targets are discovered and updated within the system. The configuration determines both periodic synchronization (UC1) and target updates (UC2).

Actors

- Primary Actor: Network Engineer

Goal

- Define target discovery behavior to meet operational needs.

Preconditions

- The system is available for configuration.

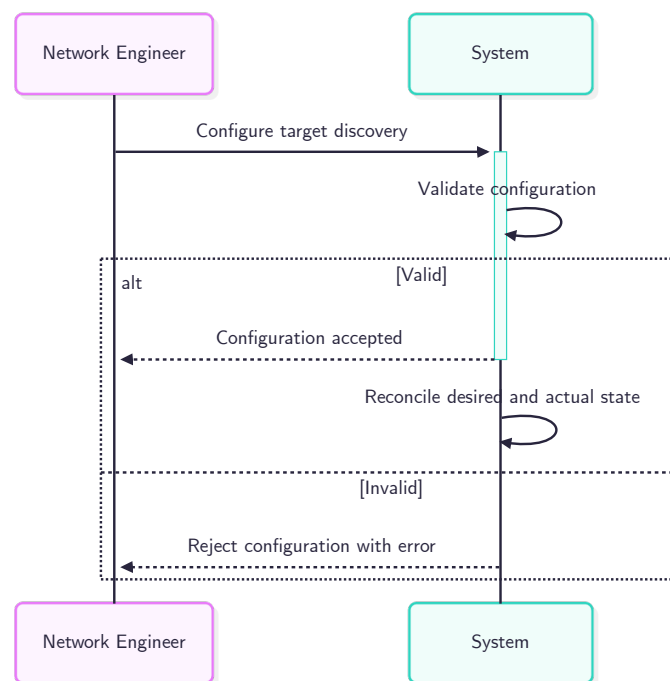


Figure 4: Sequence diagram for UC3

Success Guarantee

- The configuration is valid, persisted, and active.

Main Success Scenarios

The main success scenario is illustrated in Figure 4.

Alternative Flows

- If invalid parameters are detected, the system rejects the configuration and provides feedback.

UC4 ObserveSystemState

This use case describes how the system continuously exposes its internal state so that it can be observed externally.

Actors

- Primary Actor: Network Engineer
- Supporting Actor: Kubernetes **API**

Goal

- Obtain insight into system state and behavior.

Success Guarantee

- Relevant operational information is continuously available.

Preconditions

- The system is operational and target discovery is configured.

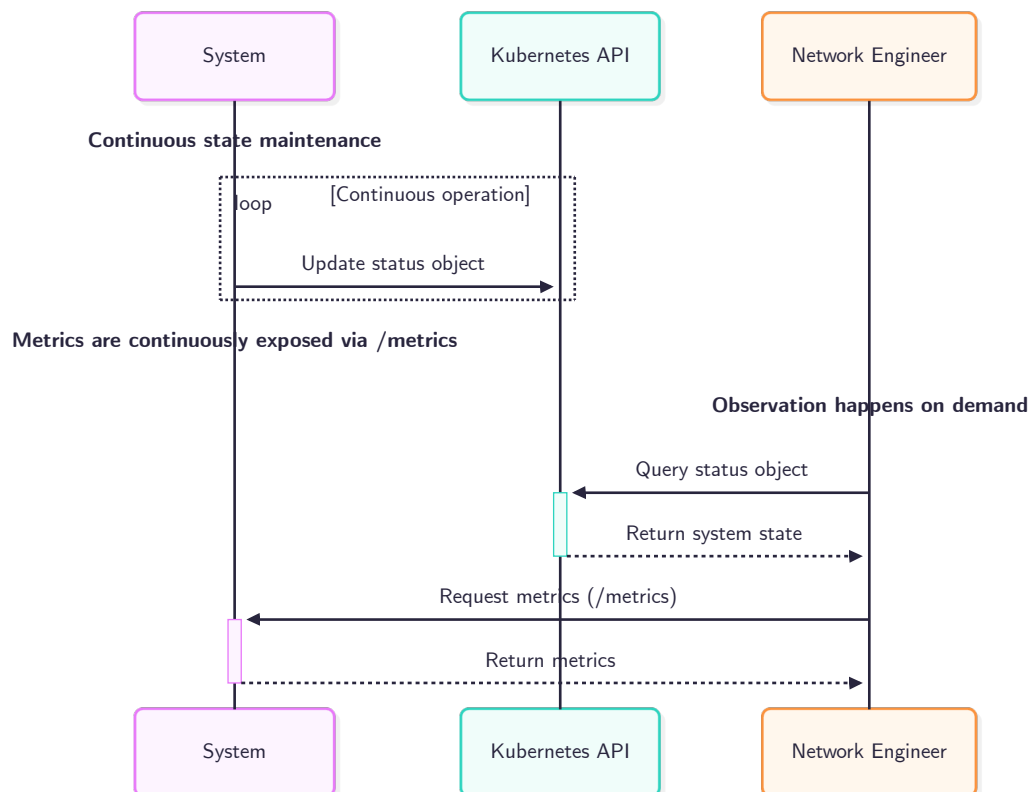


Figure 5: Sequence diagram for UC4

Main Success Scenario

The system continuously maintains status information about its operation. This information is updated and stored in the status fields of Kubernetes custom resources, reflecting the observed system state. In addition, the system exposes metrics that describe its internal behavior via a metrics endpoint.

The main success scenario is illustrated in Figure 5.

Miscellaneous

This approach separates the exposure of system state from direct request-response interactions, allowing the system state to be observed through Kubernetes resources and metrics, which aligns with Kubernetes-native patterns of state observation.

2.1.2. Use Case Diagram

The use case diagram in Figure 6 offers a visual overview of the actors and their interactions with the system to clearly illustrate the functional scope at a glance.

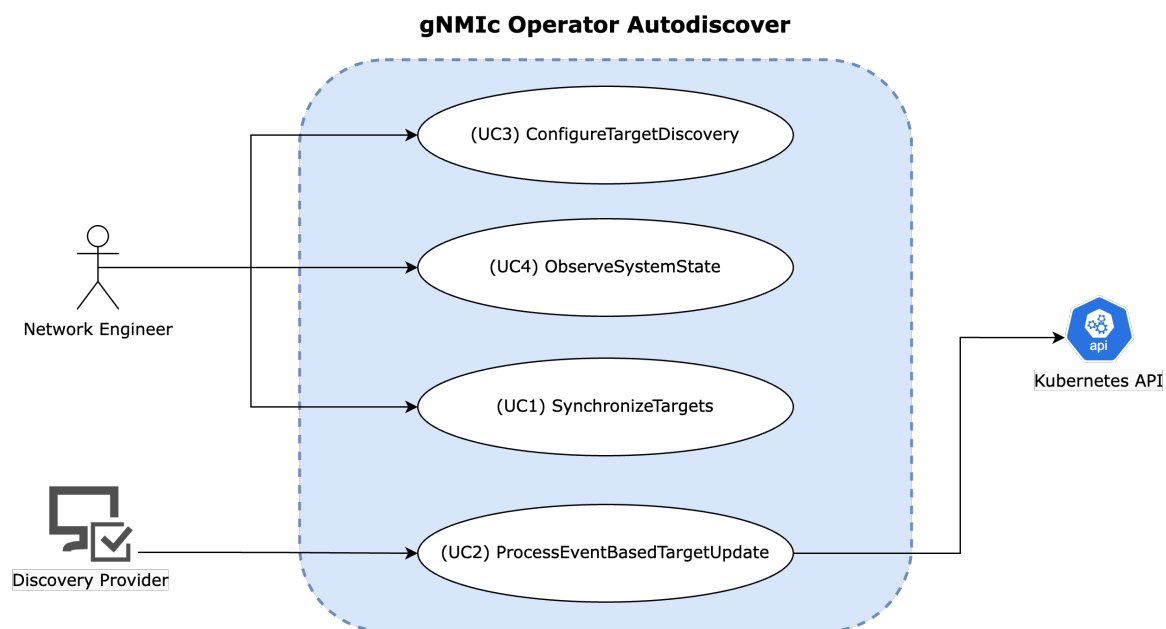


Figure 6: Use case diagram

2.2. Functional Requirements

The functional requirements are derived from the use cases and grouped accordingly. This ensures traceability and consistency between requirements and system behavior. Requirements are assigned a priority level to indicate their relative importance. In the context of this thesis, the levels High and MVP are used, with MVP requirements representing the minimum functionality required for successful project completion.

FR 1 - Synchronization Requirements

Table 2 holds the functional requirements which were derived from UC1.

ID	Requirement	Priority
FR 1.1	The system shall retrieve device information from a Discovery Provider	MVP
FR 1.2	The system shall create targets for newly discovered devices	MVP
FR 1.3	The system shall update existing targets when device data changes	MVP
FR 1.4	The system shall remove targets for devices that are no longer present	MVP
FR 1.5	The system shall prevent duplicate target entries from the same source	High

Table 2: Functional Requirements: Synchronization

FR 2 - Event-Based Update Requirements

Table 3 holds the functional requirements which were derived from UC2.

ID	Requirement	Priority
FR 2.1	The system shall be able to receive update events from a Discovery Provider	MVP
FR 2.2	The system shall process creation, update, and deletion events	MVP

Table 3: Functional Requirements: Event-based Update

FR 3 - Configuration Requirements

Table 4 holds the functional requirements related to UC3, describing what the system must provide in order to fulfill this use case.

ID	Requirement	Priority
FR 3.1	The system shall allow configuration of one concrete <u>SoTs</u> as a Discovery Provider	MVP
FR 3.2	The system shall allow configuration of a synchronization interval	MVP
FR 3.3	The system shall allow limiting queries	MVP
FR 3.4	The system shall allow configuration of the commonest authorization schemas	High
FR 3.5	The system shall support filtering of discovered devices	High
FR 3.6	The system shall support mapping of devices to target profiles	High
FR 3.7	The system shall allow assignment of labels to devices	High

Table 4: Functional Requirements: Configuration

FR 4 - Observability Requirements

Table 5 describes the functional requirements derived from UC4.

ID	Requirement	Priority
FR 4.1	The system shall log discovered and removed devices	MVP
FR 4.2	The system shall provide synchronization status information	MVP
FR 4.3	The system shall provide statistical information about discovery processes	High
FR 4.4	The system shall provide error and warning logs	High

Table 5: Functional Requirements: Observability

2.3. Non-Functional Requirements

The non-functional requirements were derived using the ISO/IEC 25010 quality model as guidance. Due to the scope of this thesis, the requirements focus primarily on performance efficiency, scalability, usability, and operational characteristics relevant to target auto-discovery. The requirements are specified in Table 6 using SMART criteria, which stands for Specific, Measurable, Achievable, Relevant, and Time-bound. These criteria help ensure that the requirements are well-defined and can be effectively implemented and evaluated.

ID	Description	
1	Performance Efficiency	
1.1	Subject Target Auto-Discovery Extension Requirement Performance Behavior Priority High Specific The auto-discovery mechanism must detect and register new or removed network targets without causing noticeable degradation to Kubernetes cluster or network performance. Measurable Operator CPU usage must not exceed 200m on average and memory usage must remain below 256Mi under normal load (≤ 2000 targets). Achievable Research will be conducted to find the best solution in terms of efficiency and reliability. Relevant Ensures stable production operation and efficient resource utilization. Time-bound Validated before production-ready release. Validation Load testing in a controlled Kubernetes cluster with simulated target changes.	
1.2	Subject Operator Runtime Behavior Requirement Controlled Resource Consumption Priority High Specific The operator must avoid unnecessary reconciliation loops and resource intensive computations. Measurable Reconciliation of the TargetSource object must remain below 2 seconds per cycle under normal conditions and must not continuously re-queue without state changes. Achievable Efficient controller-runtime patterns and caching mechanisms are implemented. Relevant Prevents degradation of Kubernetes control-plane performance. Time-bound Benchmarks completed before final evaluation. Validation Monitoring controller metrics and reconciliation latency measurements.	
2	Network Efficiency	

ID	Description
2.1	Subject Auto-discovery Communication Requirement Minimal Network Impact Priority High Specific The auto-discovery process must minimize active probing and avoid excessive gNMI or API requests to network devices. Measurable No more than one discovery-related request per target per configurable interval (default ≥ 60 seconds). Achievable Discovery is primarily event-based using Kubernetes resources or an external source of truth. Relevant Prevents unnecessary load on production network devices. Time-bound Implemented and tested before beta release. Validation Network traffic monitoring during stress tests.
3	Usability
3.1	Subject Target Auto-Discovery Configuration Requirement Ease of Configuration and Documentation Quality Priority High Specific The system must provide clear and comprehensive documentation and enable a Network Engineer to configure target auto-discovery in a simple and intuitive manner. Measurable A Network Engineer must be able to successfully configure a Discovery Provider and enable auto-discovery within ≤ 1 hour using the provided documentation, without external assistance. Achievable Configuration is implemented using Kubernetes-native Custom Resources with consistent schema and documented examples. Relevant Ensures usability of the system and reduces configuration errors in production environments. Time-bound Documentation and usability validated before final thesis submission. Validation Practical usability test where users follow a predefined scenario using the documentation to configure auto-discovery in a test environment.
4	Update Latency
4.1	Subject Target State Synchronization Requirement Bounded Update Propagation Time Priority Medium Specific New, updated, or removed targets must be reflected in the gNMIc operator configuration within a defined maximum delay.

ID	Description
	Measurable Target changes must be propagated and visible in the operator state within ≤ 30 seconds after detection (configurable). Achievable Event-driven reconciliation with configurable re-sync period. Relevant Ensures accurate and up-to-date telemetry coverage. Time-bound Latency validated before thesis submission. Validation End-to-end timing tests measuring detection-to-configuration delay.
5	Scalability
5.1	Subject Auto-Discovery Scaling Capability Requirement Horizontal and Target Scaling Priority High Specific The auto-discovery extension must support large-scale environments with a high number of network targets without significant degradation in performance. Measurable The operator must reliably manage at least 2,000 targets with update latency ≤ 60 seconds and without exceeding 500m CPU and 512Mi memory usage. Achievable Efficient data structures, caching, and optional horizontal scaling of the controller are implemented. Relevant Ensures applicability in enterprise and service provider environments. Time-bound Scalability tests completed before final thesis evaluation. Validation Stress testing with simulated large-scale target inventories and monitoring system metrics.

Table 6: Non-functional requirements

3. Research

The research followed a structured process to ensure a systematic evaluation.

1. **Requirement analysis & creation of basic use case:** Defining functional and non-functional requirements.
2. **Definition of research questions:** The research began by formulating core research questions concerning:
 - Target discovery: How target information is delivered to the **gNMIC** Operator.
 - Target lifecycle: Identification, adding, updating and deleting targets in the **gNMIC** Operator, including reconciliation logic and failure handling.
 - Target authority: Whether the **gNMIC** Operator or a Discovery Provider acts as the authoritative system.
 - Interface requirements: Derived from the chosen target discovery method, including prioritization and implementation planning.
3. **Market Research:** A survey was conducted to gather practical input from industry practitioners, providing real-world insights into operational challenges, use cases, and design constraints.
4. **Evaluation Process:** Each research question underwent a structured evaluation process:
 - Option analysis: investigating possible designs and their technical properties or what currently exists.
 - Assessment of advantages and disadvantages: examining trade-offs, limitations, and operational impacts.
 - Architectural Decision Record (**ADR**) documentation: summarizing the analysis and capturing the rationale behind each decision.
 - Conclusion formulation: providing a concise answer to the research question based on the findings.
5. **Reevaluate requirements and fine-tune to the findings:** Based on the research, check if everything aligns with the proposed use case and make necessary changes.

3.1. Market Research

To better understand the current practices in network telemetry management and inventory integration, a survey was conducted among network engineering professionals. The objective of this survey was to gather information about how network operators currently manage telemetry targets, what systems they use as a **SoT** for network devices, and whether there is a practical need for automated target discovery mechanisms.

The survey was posted in relevant Slack and Discord channels to target the intended audience related to this research.

Survey Structure and Objectives

The first section of the survey focused on respondent background and professional context. Questions regarding the respondent's role, years of experience in networking, and the size of their managed infrastructure were included, as well as the kind of network they manage and if they use Kubernetes in their networks. The purpose of these questions was to determine the relevance and reliability

of the collected responses. Responses from experienced engineers managing large infrastructures provide stronger evidence regarding real-world operational practices.

The second section examined the SoTs for network devices. Respondents were asked which tools or platforms they use to maintain authoritative information about the intent of their network infrastructure. Examples include dedicated network inventory systems, automation repositories, or service management platforms. The goal of this section was to identify potential integration points for the **gNMIC** Operator.

The third section focused on current telemetry practices. Participants were asked which telemetry protocols they currently use, how telemetry targets are defined, and how frequently these targets change within their environment. These questions were intended to identify operational challenges related to telemetry configuration, particularly whether telemetry targets are defined manually or generated through automated systems, supporting the purpose of this thesis.

Finally, the survey included open-ended questions asking respondents about limitations of their current solutions and recommendations for the design of an automated telemetry discovery system. These qualitative questions aimed to capture practical insights from industry professionals that might not be evident from the previous questions.

Respondent Profile

The collected responses indicate that the survey primarily reached experienced network professionals. A significant majority of respondents (22 out of 32) reported having ten or more years of networking experience, while only a small minority had less than five years of experience. It also revealed that most respondents manage large-scale environments. Seventeen participants reported managing more than 1000 network devices, while nine respondents manage between 200 and 1000 devices.

Infrastructure and Platform Adoption

The survey results show that container orchestration platforms are already widely adopted in the networking and infrastructure domain. Seventeen respondents reported using Kubernetes in production environments, while three use it in testing environments and five indicated that they are planning to adopt it in the future. Only seven respondents reported not using Kubernetes at all.

Sources of Truth for Network Devices

One of the most important findings of the survey concerns the systems used as an SoT for network devices. The responses indicate that dedicated network inventory systems are widely used. Other people rely on file-based inventories or custom-built solutions, while only a few reported that they do not maintain a formal SoT, as shown in Figure 7.

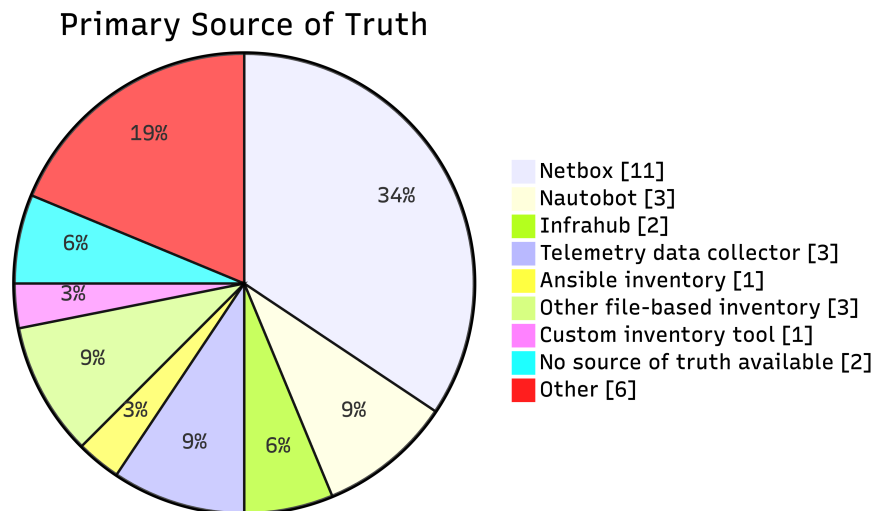


Figure 7: Pie chart showing the primary source of truth according to respondents (n=32).

Telemetry Configuration Practices

Another key focus of the survey was understanding how telemetry targets are currently defined and managed. The results indicate that many organizations still rely on manual configuration processes. Some respondents indicated that telemetry targets are gathered automatically from their inventory systems. However, these solutions seem to be less common and often rely on custom solutions. An overview of the answers is given in Figure 8.

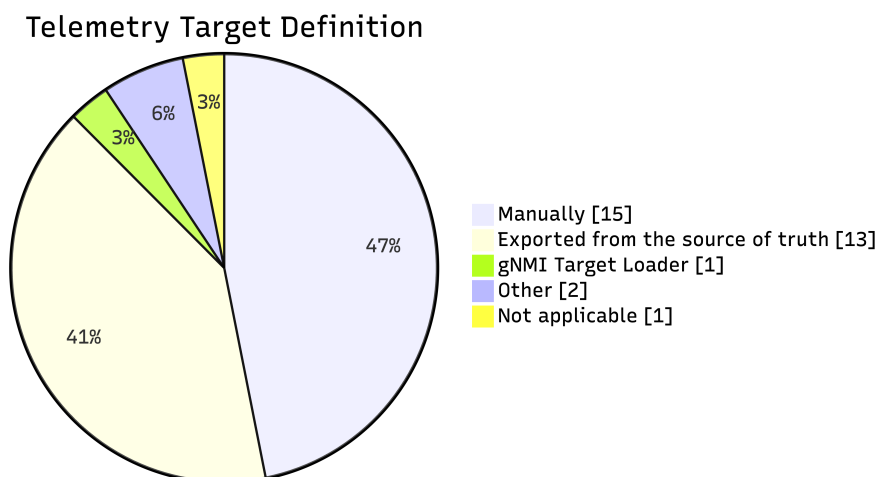


Figure 8: Pie chart showing how targets are defined for telemetry systems (n=32).

The survey also asked how frequently telemetry targets change in the respondents environments. Nearly half of the respondents reported that targets change frequently, while others indicated occasional changes. Only a minority stated that their telemetry targets rarely change.

Preferences for Target Discovery Mechanisms

Participants were also asked how they would prefer telemetry targets to be discovered or generated. The majority indicated that targets should ideally be derived from an authoritative source of truth rather than discovered through network scanning or other mechanisms, which is shown in Figure 9.

Ideal Target Definition Mechanism

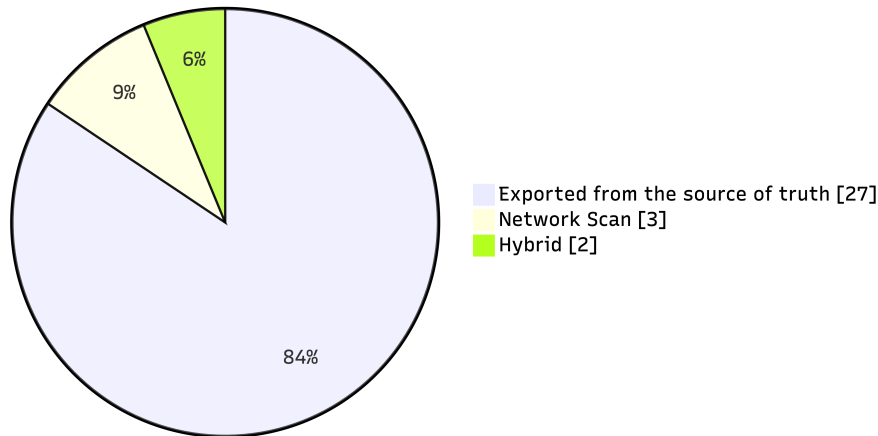


Figure 9: How respondents would want their targets to be defined ideally (n=32).

Qualitative Feedback and Design Implications

The open-ended responses provided additional insights into the practical requirements for a telemetry target discovery system. Several respondents emphasized the importance of tight integration with existing inventory systems, highlighting that telemetry targets should be derived directly from authoritative device records. Other respondents recommended implementing event-driven synchronization mechanisms, such as webhooks or [API](#)-based updates to support timely updates.

Some respondents also noted that they have already implemented custom-built automation pipelines to solve this problem internally. This observation indicates that the problem addressed by this thesis already exists in practice and that engineers often need to develop their own solutions to manage telemetry targets dynamically.

Interpretation of Results

Overall, the survey results demonstrate that network operators frequently manage large infrastructures with dynamic device inventories. While many organizations maintain an authoritative source of truth, telemetry configuration is still often performed manually or through custom automation scripts. This creates operational overhead and increases the risk of configuration inconsistencies.

The strong preference for deriving telemetry targets from an existing source of truth suggests that automated discovery mechanisms integrated with inventory systems would address a real operational need. Furthermore, the widespread adoption of Kubernetes in infrastructure environments supports the viability of implementing such functionality through a Kubernetes-native operator.

These findings provide empirical support for the design goals of the proposed target auto-discovery extension, which aims to automatically synchronize telemetry targets with an authoritative inventory system while leveraging Kubernetes-native reconciliation mechanisms.

3.2. Target Discovery

The main approaches to automatically discover targets are: (1) actively scan the network to detect network devices, and (2) importing the device information from an external [SoT](#). This data will be converted and loaded into the [gNMIC](#) Operator, after which [gNMI](#) monitoring starts.

The target discovery should achieve the following goals:

- Discovery coverage: Discover eligible network devices in a given network.
- Required target attributes: Provides or obtains Internet Protocol (**IP**) address, port for **gNMI** access, Media Access Control (**MAC**) address, credentials, and the state of the target.
- Data consistency: Obtains accurate and complete information of targets.
- Event-based update: Provide the information in a timely manner.
- Data structure: Provide the information in a structured format such as JavaScript Object Notation (**JSON**) or YAML Ain't Markup Language (**YAML**).
- Scalability: Scale well with the network size and the frequency of changes in the network.
- Network impact: Small impact on the network to avoid congestion.

3.2.1. Discovery Mechanisms

Target discovery can be implemented using different mechanisms, each with distinct trade-offs in terms of scalability, timeliness, and operational complexity. The main approaches are summarized below, with the detailed research provided in [Appendix B](#).

Agent-based discovery

Agent-based approaches rely on active or passive scanning within the network to detect devices and extract their attributes. While this enables autonomous discovery, it introduces additional network overhead and operational complexity.

Pull-based discovery

In pull-based approaches, the **gNMIC** Operator periodically retrieves target data from an external **SoT**. This method is simple and reliable, but introduces latency depending on the polling interval and may generate unnecessary requests.

Push-based discovery

Push-based approaches rely on event-driven updates, where the **SoT** actively sends target changes to the operator. This enables low-latency updates but may lead to inconsistencies if events are missed or delayed.

Hybrid-based discovery

Hybrid-based discovery combines pull- and push-based mechanisms to leverage their respective strengths. Push is used for timely updates, while periodic pull operations ensure consistency and completeness. This approach increases implementation complexity, but provides a balanced solution.

3.2.2. Summary of Discovery Approaches

Table 7 gives an overview of the different target discovery mechanism compared to the **gNMIC** Operator goals from Chapter 3.2.

Criterion	Agent-based (active)	Agent-based (passive)	Pull- based	Push- based	Hybrid- based
Discovery coverage	✓ Fulfilled	✓ Partial fulfillment	✓ Fulfilled	✓ Fulfilled	✓ Fulfilled
Required target attributes	✓ Partial fulfillment	✗ Unfulfilled	✓ Fulfilled	✓ Fulfilled	✓ Fulfilled
Data consistency	✓ Partial fulfillment	✓ Partial fulfillment	✓ Fulfilled	✓ Partial fulfillment	✓ Fulfilled
Event-based update	✗ Unfulfilled	✗ Unfulfilled	✗ Unfulfilled	✓ Fulfilled	✓ Fulfilled
Data structure	✓ Fulfilled	✓ Fulfilled	✓ Fulfilled	✓ Fulfilled	✓ Fulfilled
Scalability	✗ Unfulfilled	✓ Partial fulfillment	✓ Partial fulfillment	✓ Fulfilled	✓ Fulfilled
Network impact	✗ Unfulfilled	✓ Fulfilled	✓ Partial fulfillment	✓ Fulfilled	✓ Fulfilled

Table 7: Target Discovery Decision Matrix

The agent-based approaches were dismissed as they don't scale well for bigger networks. Scanning the network with the **gNMIc** Operator generates traffic that is redundant, especially if the **SoT** already does this. Also the community feedback from the survey shows that importing targets from the source of truth is preferred by 90% of people.

The pull-based approach can generate unneeded network traffic and does not support event-based target updates. Using push-based mechanisms include the problem that target updates can get lost in transit and never end up in the **gNMIc** Operator. Thus, **hybrid-based target discovery is chosen**. While adding the complexity of two required integrations, it also brings the benefit of both. The push mechanism is used to obtain event-based updates, while periodically pulling ensures data consistency. This is documented in [ADR-01](#).

The combined implementation of the hybrid-based approach will look as follows.

1. Pull target list at **gNMIc** Operator start.
2. Listen for target updates sent with push.
3. Periodic pulls to ensure no target updates were missed.

3.3. Target Lifecycle

The target lifecycle focuses on the different stages of a Target resource in the Kubernetes cluster, from being discovered, to being created, updated, and deleted, and how error handling is supposed to happen.

3.3.1. Current Implementation

The **gNMIC** Operator is based on a set of Custom Resource Definitions (**CRDs**), which define the different parts of a telemetry infrastructure [10]. It follows the Kubernetes operator pattern, which makes the whole setup declarative and it reconciles itself to adhere to the desired state [11]. An overview of the **CRDs** is given in Figure 10.

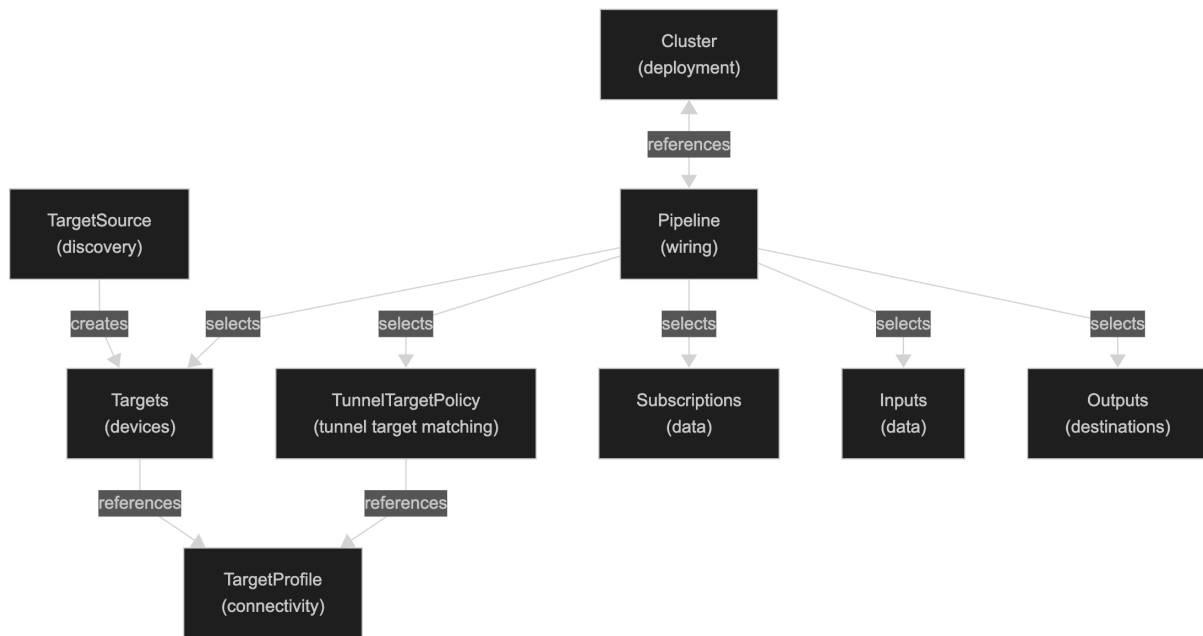


Figure 10: Resource hierarchy of the gNMIC Operator [12].

For the operator to reconcile a target into its configuration and be able to establish a connection to this target, two Kubernetes objects are needed:

- Target: Defines the IP address of the network device and assigns labels
- TargetProfile: References a Kubernetes secret which holds the credentials to access the device.

This object gets referenced by Target.

Paired with a Subscription to specify the **YANG**-path, and the Pipeline to tie all objects together, the telemetry data can be collected.

In the current implementation, Target resources must be created manually. This requires operators to explicitly define each monitored network device within the Kubernetes cluster. While this approach provides full control, it becomes increasingly impractical in environments where network inventories are maintained in external systems or where devices are frequently added or removed.

3.3.2. TargetSource Concept

To enable automated target discovery, the architecture introduces a concept of a TargetSource resource [13]. This resource acts as an abstraction layer between the external **SoT** and the operator.

A TargetSource specifies how the operator should retrieve device information from an external system. The operator then periodically reconciles this information and dynamically generates the corresponding Target resources within Kubernetes.

Essentially, the TargetSource performs three main functions:

- Inventory retrieval from an external system

- Transformation of discovered devices into operator-compatible target definitions
- Synchronization, ensuring the Target resources are always up-to-date according to the external source.

3.3.3. Target Life Cycle Management

Once a TargetSource retrieves the device inventory from an external source, the operator must reconcile this inventory with the existing Target resources in the Kubernetes cluster. This process determines whether targets need to be created, updated, or removed.

The reconciliation logic follows the Kubernetes operator pattern and compares the set of discovered devices with the currently existing Target resources. Based on this comparison, three actions may be performed:

- Creation of new Target resources for previously undiscovered devices
- Updates to existing Target resources when device attributes change
- Removal of stale Target resources that are no longer present in the discovered inventory.

To enable reliable reconciliation, several design considerations must be addressed, including target identification, deletion behavior, and failure handling.

Target Identification

To reconcile discovered devices with existing Target resources, each device must be uniquely identifiable. The discovery backend therefore derives a deterministic identifier from the device information retrieved from the external system. This identifier is used to determine whether a corresponding Target resource already exists within the cluster. If a matching resource is found, the operator updates it if necessary. Otherwise, a new resource is created.

Using a consistent identification mechanism prevents duplicate Target resources from being generated across reconciliation cycles. The goal is to provide a way of identification which is supported by different SoTs. This means that the way of identifying a device should be made based on a basic set of variables available for all kinds of devices.

This is where the gathered information from the market research survey plays a valuable role. With *Netbox* being the SoT for 34% of the respondents, it appears to be an important system to consider. The documentation states that, while device names are optional, they need to be unique within a site [14]. The same goes for *Nautobot* [15], and *Infrahub* supports Display Labels, which can act as identifiers, or provides a default one [16]. These other two SoTs make up for another 16% of SoTs used by the respondents.

Considering this information, the name, or identifier, would provide the necessary information to identify a device within a site. To ensure uniqueness across the Kubernetes cluster, the name gets paired together with a prefix associated with the SoT. To allow for devices with the same name in different tenants, multiple TargetSource objects can be created, with each one covering a specific tenant. According to this, the name might look like: `netbox-router1`. The use of the IP address in the identifier won't be regarded, since this information is subject to change and part of the Kubernetes specifications (specs) of the resource. This is further documented in [ADR-02](#).

CRUD Operations

When a discovered device does not yet exist as a Target resource, the operator creates a new object based on the information provided by the discovery backend. This includes attributes such as the device address, associated labels, and references to the corresponding TargetProfile.

If the device already exists as a resource, the operator compares the discovered attributes with the existing resource definition. If differences are detected, the resource is updated to reflect the latest state of the external inventory. This mechanism ensures that the telemetry configuration remains synchronized with the SoT.

Another design consideration concerns the removal of stale targets. Devices may disappear from the external inventory due to decommissioning, topology changes, or administrative actions. Two approaches can be used to determine when a target should be removed:

- Explicit deletion, where the external system explicitly signals that a device has been removed
- Implicit deletion, where a device no longer appears in the discovered inventory

In the proposed life cycle model, targets that no longer appear in the discovered inventory are considered stale and may be removed during reconciliation. Safeguards must be implemented to avoid unintended deletions caused by temporary inconsistencies in the discovery process.

Failure Handling

The target life cycle must also account for potential failures of the discovery backend. For example, the external SoT may become temporarily unavailable or return incomplete results.

In such cases, the operator must avoid destructive actions based on unreliable data. Instead, previously discovered targets are preserved until a successful discovery cycle can be performed. This prevents valid targets from being removed due to temporary discovery failures.

Since both Kubernetes and the gNMIC Operator implement certain validation and error handling mechanisms, it is a valid approach to only do the necessary error recovery to the system in development in the context of this thesis. Further checks happen at the levels of the Kubernetes API, e.g. when applying targets, or at the operator level, when targets need to be reconciled into the running configuration. This is documented in ADR-05.

3.4. Components

The hybrid approach chosen in Chapter 3.2 introduces complexity as both the pull- and push-based need to be developed. Splitting those requirements into separate interfaces simplifies the development approach. It further brings the required structure and decoupling for efficient coding.

Two more components were identified, the Kubernetes interface that interacts with the cluster and the gNMIC auto-discovery logic, that wires all parts together.

Based on NetBox being the most popular SoT according to survey answers, it was decided to focus development on providing support for NetBox. Support for additional SoT vendors may be considered if time permits. All components are visualized in Figure 11.

All components are visualized in Figure 11.

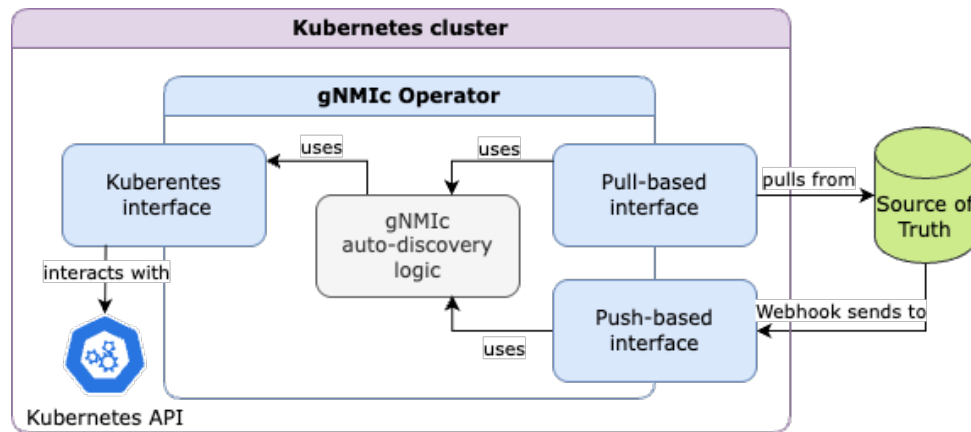


Figure 11: Overview of components evaluated during research.

The following sections describe the components and their technical implementations.

Pull-Based Interface

The pull-based interface retrieves target information from NetBox using its Representational State Transfer (**REST**) **API**, following standard Hypertext Transfer Protocol (**HTTP**) communication patterns. The pull-based interface uses the `net` package from the Go standard library to fetch a list of target from the NetBox **REST API** [17], [18]. This approach aligns with the generic **HTTP**-based loader architecture and the design decision documented in [ADR-07](#).

Push-Based Interface

The push-based interface offers a **REST API** that accepts target updates sent from a webhook by the **SoT**. The **REST API** is defined per the OpenAPI Initiative standard [19]. [ADR-03](#) goes into why the interface is implemented in the gin-gonic framework.

Kubernetes Interface

The **gNMIC** Operator was created using the Kubebuilder SDK, a framework for building Kubernetes **APIs** [20]. The challenge with the Kubernetes interface is to identify how and which parts of the existing code can be used.

gNMIC Auto-Discovery Logic

The **gNMIC** auto-discovery logic is the core feature this thesis implements. Its role is to process targets received via the pull- and push-based interfaces, identify the corresponding device, and initiate the appropriate subsequent steps.

3.5. Findings

The research findings were presented to the thesis advisors during the mid-term evaluation, and their feedback has been incorporated into this chapter. As all research questions have been systematically addressed and the results validated, this marks the completion of the research phase and provides a solid foundation for the subsequent development phase.

4. Design & Implementation

4.1. Architecture

The software architecture is documented according to the arc42 standard [21]. Some parts were omitted, as they are either documented in another chapter or not applicable to this project.

The architecture chapter focuses on the auto-discovery feature, with the full-scale architecture being documented in the [gNMIC Operator docs](#) [11].

4.1.1. Context

Figure 12 shows the context the [gNMIC Operator](#) runs in. Given the flexible nature of Kubernetes, other variants are also valid. For example, the Source of Truth ([SoT](#)) can be run in the Kubernetes cluster or the usage of other tools in the monitoring stack is legitimate.

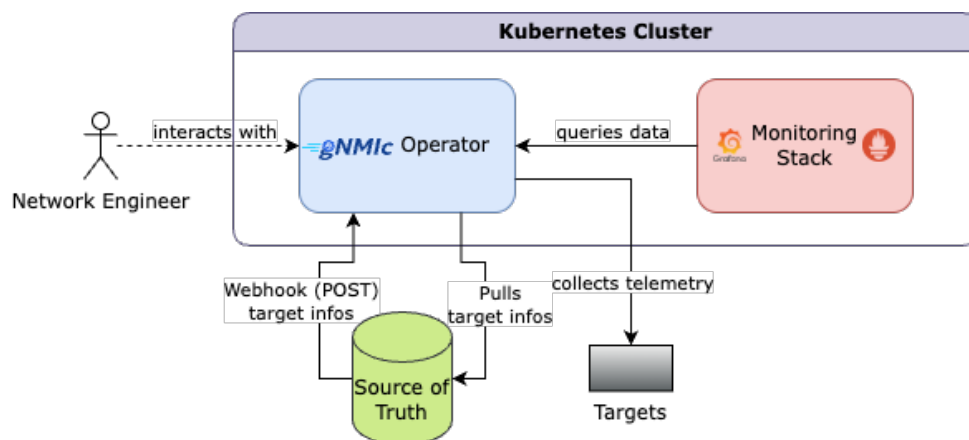


Figure 12: arc42 context view

Table 8 holds descriptions for each of the objects mentioned in the arc42 context view.

Component	Description
Network Engineer	The Network Engineer configures the gNMIC Operator .
Kubernetes Cluster	Container platform on which the gNMIC Operator runs.
gNMIC Operator	The Kubernetes Operator extended in this project.
Monitoring Stack	Stores and visualizes collected telemetry data to monitor network traffic flow and device status. Prometheus and Grafana are used.
Source of Truth	Inventory system for network devices, providing structured device data. NetBox is used as the SoT .
Targets	Network devices from which the gNMIC Operator collects telemetry data. In this project, Nokia SR Linux and Arista cEOS are used as targets.

Table 8: Context diagram components

4.1.2. Building Block View

The building block view shows how the target auto-discovery feature is decomposed into components and how those components interact.

Level 1

Figure 13 gives a broad overview of how the auto-discovery feature is structured. Building blocks are named according to the Go packages in the source code [1].

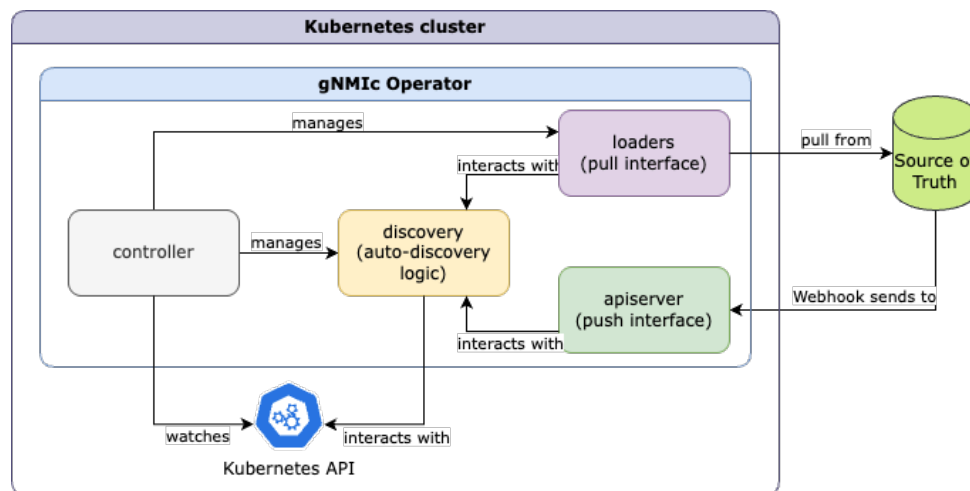


Figure 13: arc42 building block view level 1

Table 9 holds descriptions for each of the objects mentioned in the arc42 building block view of level 1.

Building Block	Description
loaders	Implement provider-specific logic to pull inventory from the <u>SoT</u> .
apiserver	Exposes the <u>REST API</u> used for incoming webhook/post updates.
discovery	Internal auto-discovery controller logic that matches cluster state with desired state and handles incoming target information.
controller	Watches the Kubernetes API for changes and orchestrates the discovery runtime and loaders according to the desired state.

Table 9: arc42 building block view L1 description

Level 2

Figure 14 describes individual components of the auto-discovery feature. The diagram aligns with the code structure from the **gNMIC** Operator repository [1]. Detailed descriptions are given in Table 10.

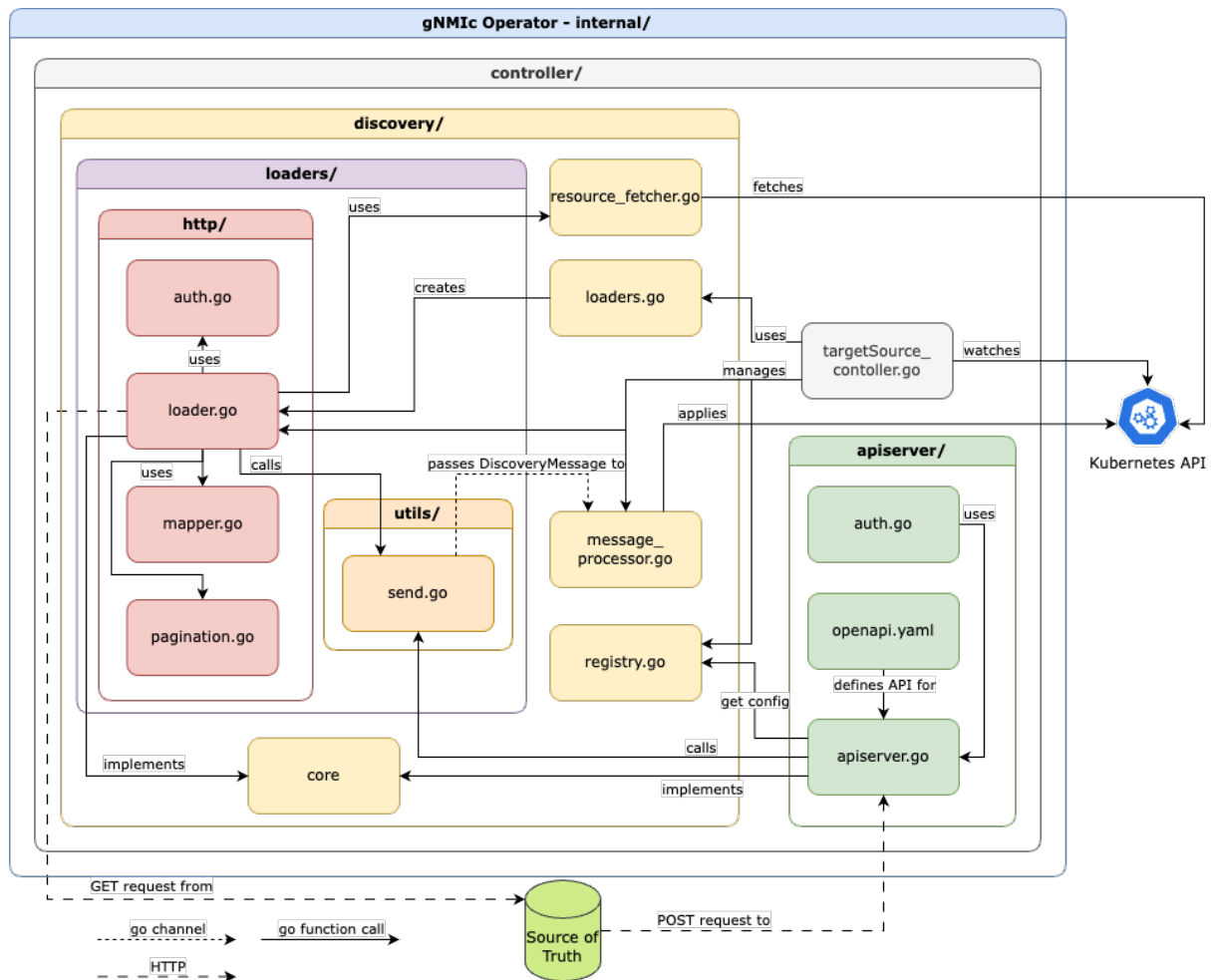


Figure 14: arc42 building block view level 2

Building Block	Description
controller/	Contains the Custom Resource (CR)-specific reconcilers and supporting logic responsible for managing the TargetSource lifecycle, orchestrating loaders and the message processor.
targetSource_controller.go	Watches the Kubernetes API for TargetSource updates. Orchestrates the discovery runtime via the reconcile loop by initializing the loader and message processor and updates runtime information in registry.

Table 10: arc42 building block view L2 controller description

Table 11 through Table 15 provide detailed descriptions of the Go packages.

Building Block	Description
controller/discovery/	Defines loader abstraction, discovery message types, shared runtime configuration structures.
core	Contains helper discovery runtime components used by the TargetSource controller.
message_processor.go	Consumes discovery messages, normalizes the resulting targets, and reconciles them into Kubernetes Target <u>CRs</u> .
resource_fetcher.go	Fetches Kubernetes resources (e.g., ConfigMaps, Secrets) needed by the discovery runtime.
loaders.go	Responsible for mapping TargetSource <u>CR</u> configurations to loader (pull/push) configurations.
registry.go	Maintains the runtime state by associating each TargetSource with its Go channel and shared configuration.

Table 11: arc42 building block view L2 discovery description

Building Block	Description
controller/discovery/loaders/	Discovery providers and supporting components that integrate different <u>SoT</u> implementations.

Table 12: arc42 building block view L2 loaders description

Building Block	Description
controller/discovery/loaders/http/	HTTP loader implementation for pulling inventory data from a <u>SoT</u> via REST API calls.
loader.go	Implements the HTTP loader, which periodically polls the <u>SoT</u> for inventory updates.
auth.go	Authentication for HTTP requests.
mapper.go	Maps the HTTP response data to the discovery message format.
pagination.go	Handles pagination for HTTP responses.

Table 13: arc42 building block view L2 http description

Building Block	Description
controller/discovery/loaders/utils/	Helper functions shared between different loader implementations.
send.go	Provides helpers for sending discovery messages over the go channel to the message processor.

Table 14: arc42 building block view L2 utils description

Building Block	Description
apiserver/	REST API endpoint implementation. Not solely push interface specific, also used by other parts of gNMIC Operator.
apiserver.go	Gin-gonic REST API implementation. Queries registry.go to obtain Go channel information to send discoveryEvent.
openapi.yaml	OpenAPI contract that defines REST API interface.
auth.go	Authentication for applyTargets route.

Table 15: arc42 building block view L2 apiserver description

4.2. TargetSource Implementation

As already introduced by the **gNMIC** Operator, the **SoT** is represented by a TargetSource Custom Resource Definition within Kubernetes. This chapter goes into detail about said **CRD**, the reconciliation logic for each TargetSource instance and the subprocesses managing the target loading & lifecycle.

4.2.1. Custom Resource Definition

The **CRD** is used to define the specific configuration for each **SoT**, as well as some common fields for the targets which get managed by this resource. An example can be found in Code Listing 1.

```

1  apiVersion: operator.gnmic.dev/v1alpha1
2  kind: TargetSource
3  metadata:
4    name: http-ts
5  spec:
6    provider:
7      http:
8        url: https://server.example.local:8443/targets
9        targetPort: 57400
10       targetProfile: profile1
11       targetLabels:
12         key: value

```

Code Listing 1: Example **CR** for a TargetSource object

The sub **spec** provider acts as a “plugin selection”, where a **SoT** gets configured. Depending on the selected provider, different configuration keys get available. Using kubebuilder, rules which validate the **spec** can be defined. This allows for only one provider to be configured, since one instance represents one **SoT**.

The two fields targetPort and targetProfile represent fields passed on to the corresponding targets. If neither of these fields get specified by the external system, these default values are applied to the target. The field targetLabels takes key-value pairs and adds them to the targets.

Kubernetes resources also contain a status section with customizable fields which hold information about the state of the object. In this case, they represent the number of targets managed by this target source and so called *Conditions*, which indicate the state the TargetSource is in (more on that in Chapter 4.2.4).

4.2.2. Reconciliation Logic

Following the Kubernetes operator pattern, each **CRD** has its own controller looking for changes and keeping the actual state up to date with the desired state [22], i.e. the TargetSource controller listens to new and changed resources of its type, and then runs the reconciliation loop.

To separate the tasks of reconciling and loading & applying the targets, the latter two were outsourced, each into a subprocess (see Chapter 4.2.3). This makes the reconciliation loop only responsible for coordinating the loader and message processing subprocesses.

To accomplish this, the high-level process in Figure 15 was defined. The controller listens for new, changed or deleted TargetSource events. Using predicates, existing TargetSources get only reconciled if the Generation (internal Kubernetes variable) changed, specifying a **spec** change.

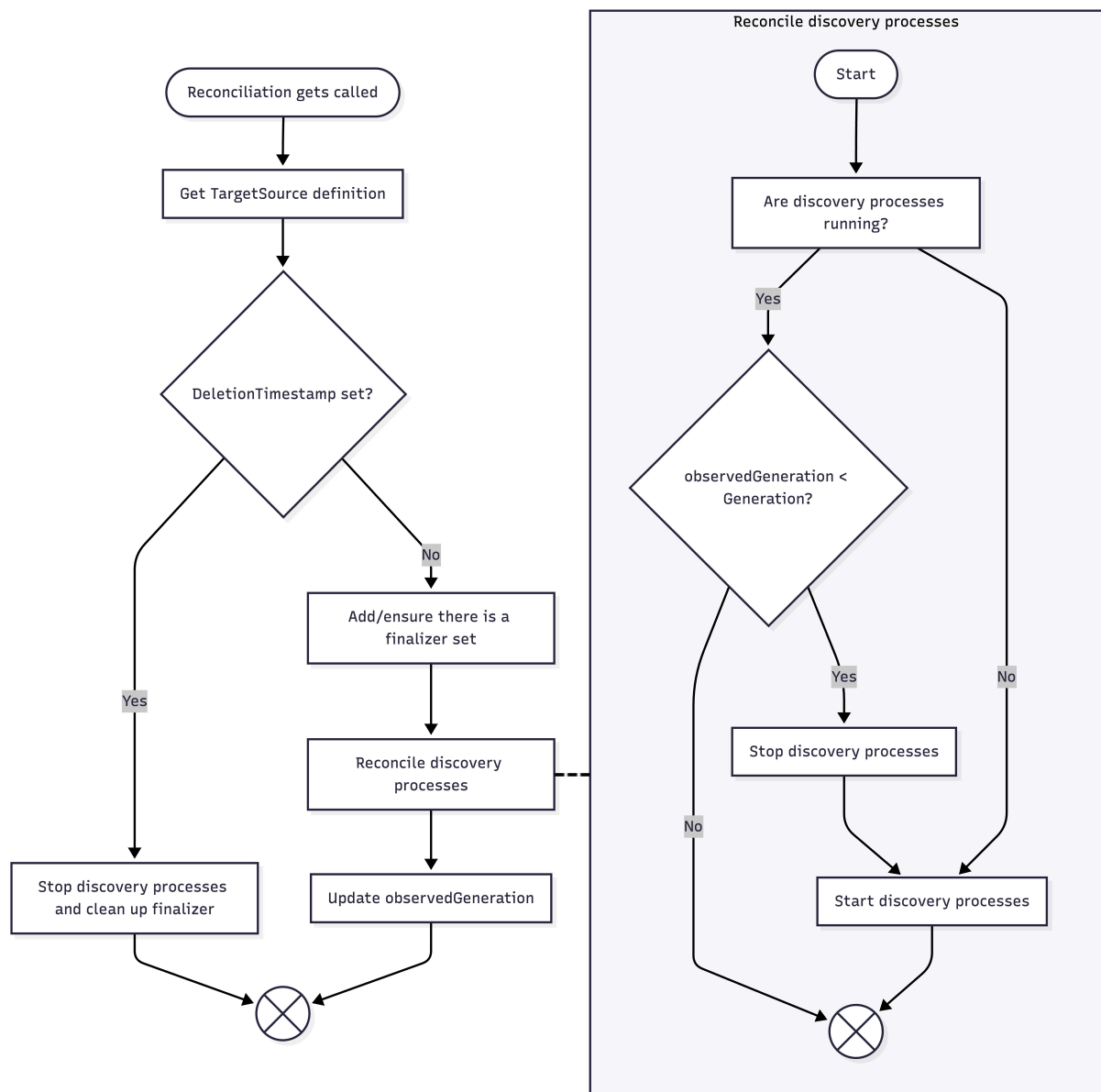


Figure 15: Reconciliation flowchart with the high-level flow on the left, and the *Reconcile discovery processes* sub graph on the right.

4.2.3. Goroutines & Subprocesses

A *goroutine* is the idiomatic way in Golang to make function calls asynchronous and concurrent with the caller function [23]. Using this mechanism, functions which loop periodically, like in this case the function responsible to fetch targets from the SoT, can run in the background until they're no longer needed. Following this approach, two main goroutines are orchestrated by the TargetSource controller:

- TargetLoader: Fetch targets from the SoT
- MessageProcessor: Apply the discovered target events to the Kubernetes cluster.

These tasks were split into two processes to separate concerns; i.e. the TargetLoader only interacts with the external system, while the MessageProcessor only interacts with Kubernetes. But since both of these tasks run asynchronously, they aren't just function calls to each other. Golang addresses this problem by providing *Channels*, representing pipes that connect concurrent goroutines [24]. The API endpoint responsible for event-based updates makes use of the same channel to pass updates to the MessageProcessor.

DiscoveryMessage Channel

The pipe connecting the TargetLoader and the MessageProcessor is a one-way buffered channel of type `[]DiscoveryMessage`. This channel acts as the transport layer for discovery data, decoupling the production and consumption of messages.

The buffering mechanism allows the TargetLoader and the API server to emit discovery payloads without being blocked by the processing speed of the MessageProcessor. Conversely, the MessageProcessor can consume messages asynchronously at its own pace.

Buffering is required because discovery can produce large snapshots (which are split into multiple messages) or bursts of updates faster than the MessageProcessor can reconcile them. The channel buffer temporarily holds these pending messages until they are processed.

However, the buffer size introduces a trade-off: if the buffer is too small, producers may experience back pressure and stall; if it is too large, the system may accumulate a significant in-memory backlog of unprocessed messages. This trade-off is addressed by both the channel buffer size and the message size being configurable runtime parameters.

The `discovery-buffer-size` parameter defines how many discovery messages can be temporarily queued in memory, allowing operators to control the balance between back pressure and in-memory backlog. In addition, the `discovery-chunk-size` parameter limits the number of targets or events contained in a single discovery message. This prevents excessively large payloads and allows large snapshots to be split into smaller, manageable chunks.

DiscoveryMessage Interface

A `DiscoveryMessage` is an interface representing a possible message with target updates discovered by the loaders. A `DiscoveryMessage` can either be a `DiscoverySnapshot`, which is a full list of all existing devices, or a `DiscoveryEvent` representing a single target update (created, updated or deleted).

```

1  type DiscoverySnapshot struct {
2      // omitted for brevity
3      Targets    []DiscoveredTarget
4  }
5
6  type DiscoveryEvent struct {
7      Target DiscoveredTarget
8      Event  EventAction // APPLY or DELETE
9  }
10
11 type DiscoveredTarget struct {
12     Name      string
13     Address   string
14     Port      int32
15     Labels    map[string]string
16     TargetProfile string
17 }

```

Code Listing 2: DiscoveryMessage implementations

Code Listing 2 shows the different types in detail. `DiscoverySnapshot` is a list of `DiscoveredTarget` objects, while `DiscoveryEvent` is just a single `DiscoveredTarget` with either `APPLY` (meaning new or changed device due to idempotency) or `DELETE` as the event. The `DiscoveredTarget` struct was defined to decouple this implementation from the actual Kubernetes [API](#) and to transmit just the necessary information in a serialized way.

TargetLoader

As mentioned, the `TargetLoader` is responsible for fetching information from the [SoT](#), i.e. it represents the implementation of the pull-based mechanism. The architecture supports different [SoT](#) implementations by abstracting them behind an interface called `Loader`. This interface defines a single function `Run(context.Context, out chan<- []DiscoveryMessage) error`, which implements the continuous loop and runs as a goroutine. The `TargetSource` controller does not need to know which `Loader` implementation is needed; it simply calls `NewLoader(...)`, which then returns the correct implementation based on the `TargetSource spec`. The `Loader` then fetches the devices and forwards them through the channel out to the `MessageProcessor`. An implementation of this is described in Chapter 4.4.

MessageProcessor

The `MessageProcessor` implementation is independent from the `Loader` implementation and the same for all `TargetLoaders`. This constantly running function waits for `DiscoveryMessages` arriving in the channel and handles them according to their type. Every change is ultimately represented as a `DiscoveryEvent`, and this flow is illustrated in Figure 16.

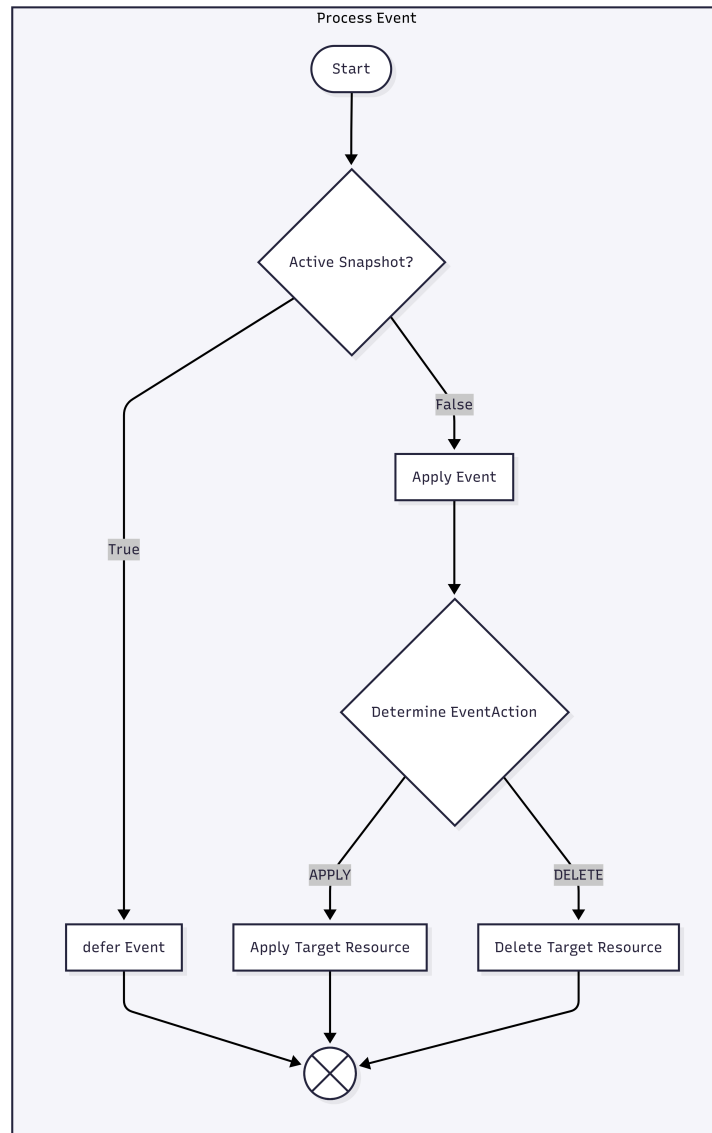


Figure 16: Flowchart of a DiscoveryEvent being applied

Messages going through the channel which already are of the type `DiscoveryEvent` don't need special handling. The `MessageProcessor` only checks if a snapshot is currently being processed, which would lead to the event being deferred to ensure the update doesn't get overwritten by the current snapshot. However, this check is only of relevance when a hybrid configuration is present. When processing a `DiscoverySnapshot`, all the `DiscoveredTargets` present within it need to be converted into `DiscoveryEvents`. By getting all existing `Target` objects inside the Kubernetes cluster and comparing them to the full list of existing devices discovered in the SoT, a list of events gets generated as shown in Figure 17.

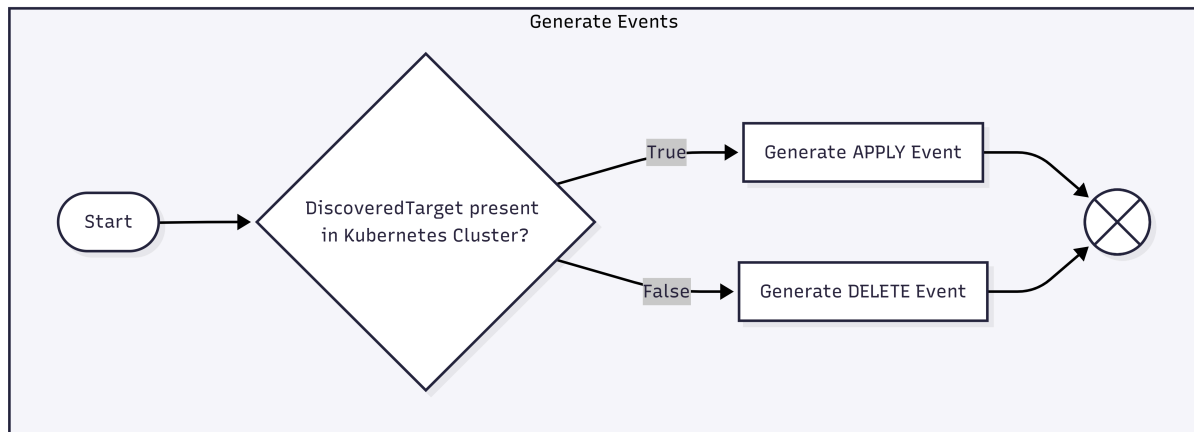


Figure 17: The list of DiscoveredTargets gets compared to the existing targets in the Kubernetes cluster to determine whether a target needs to be applied or deleted.

In conclusion, Figure 18 shows the whole flow from a message being identified as a DiscoverySnapshot up to being handled as a DiscoveryEvent.

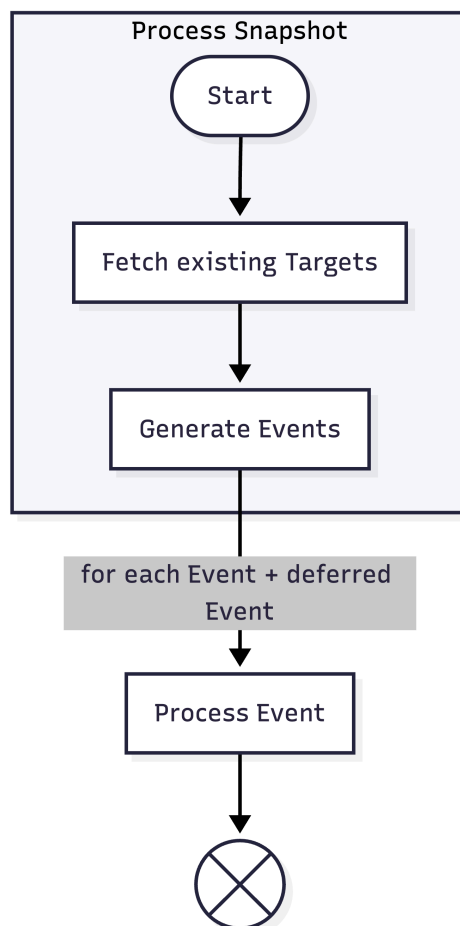


Figure 18: Flowchart of the DiscoverySnapshot being handled accordingly and applied as DiscoveryEvents.

4.2.4. Observability & Status Management

As part of the functional requirements, Table 5 describes that the operational status of a TargetSource should be retrievable. Kubernetes features a standardized way to represent the state

of an object, which are *Conditions* [25]. With these conditions, it is possible to specify the following fields (incomplete list):

- Type: Name of the condition,
- Reason: Machine-readable, domain-based set of reasons,
- Message: Human-readable, customizable message for each specific case [25].

Each time a condition is set, a timestamp of the transition time must be supplied to identify when the state changed. Following this pattern, the types and reasons in Table 16 were defined.

Type	Description
<i>Ready</i>	The synchronization between the gNMIC Operator and the SoT is running (set after each successful synchronization).
<i>Reconciling</i>	Synchronization is currently running.
<i>Degraded</i>	Synchronization is working, but there were some errors during the last execution.
<i>Stalled</i>	Target definitions cannot be retrieved from the external system.

Table 16: TargetSource condition types

If an active condition is set due to an error, the error message gets displayed in the message field.

To ensure that all the processes responsible for updating conditions (namely the TargetLoader and MessageProcessor) apply them in the intended way, the function is abstracted behind the StatusUpdater interface. This method ensures also that the processes which do not have access to the whole Kubernetes client object can only manipulate the status and not interact with the **API** in any other way.

The StatusUpdater interface defines a single function UpdateStatus(context.Context, StatusUpdate) error, which accepts the StatusUpdate struct as defined in Code Listing 3.

```

1  type StatusUpdate struct {
2      Conditions []metav1.Condition
3      TargetsCount *int32
4  }
```

Code Listing 3: The StatusUpdate struct holds all status fields which get updated during the discovery process.

All existing conditions get replaced by the list of conditions supplied in this struct, and the TargetsCount, representing the number of Target objects owned by a TargetSource, gets updated if the corresponding pointer is not nil. The reason this gets passed as a pointer is to differentiate between cases where the count is zero, or the count does not get updated by the current caller (e.g. a synchronization event fails, but the existing targets continue existing).

In general, all external functions called are checked for errors (if applicable) and all errors received get logged in the **gNMIC** Operator logs.

4.2.5. Failure Recovery & Consistency

As this thesis only covers a subset of the whole **gNMIC** Operator, it is important to handle errors specific to target auto-discovery and to not have complete operator outage based on an error in the discovery process.

A key design goal is to prevent temporary failures of the **SoT** from causing unintended modifications to existing Target resources. Discovery snapshots are therefore only used to determine target removals when a synchronization cycle completes successfully. If the external system is unavailable or returns an error, the existing targets remain unchanged and the corresponding TargetSource condition is updated accordingly.

The implementation follows an eventually consistent model. Push-based updates allow changes to be propagated immediately, while periodic synchronizations performed by the TargetLoader ensure that missed updates are reconciled during subsequent discovery cycles. This guarantees that the set of Target resources converges towards the state represented by the external **SoT**.

To avoid inconsistencies caused by repeated processing of the same information, all target changes are applied through the reconciliation logic described in Chapter 4.2.3. Since target identification is deterministic, processing identical discovery data multiple times does not create duplicate Target resources and only results in updates when the desired state differs from the current state.

Due to the operator running on Kubernetes, additional recovery mechanisms such as automatic process restarts are already provided by the platform. In addition to that, validation of variables like target names and addresses are implemented by the **gNMIc** Operator, which is why it would only add redundancy to implement a second layer of validation. This is documented in detail in [ADR-05](#).

4.3. REST API Interface

The push-based method implements a **REST API** interface specified according to the OpenAPI standard [19]. The `openapi.yaml` file serves as the contract that defines the interface. A contract-first approach offers several advantages:

- Contract is the single source of truth, resulting in fewer integration bugs [26].
- Generation of documentation minimizes documentation drift [19].
- Generated code reduces repetitive implementation work [27].

Code Listing 4 defines two **HTTP** endpoints. The first endpoint did exist and was incorporated into the OpenAPI contract, although it is not used by the auto-discovery feature. The second endpoint serves as the interface for incoming **HTTP** requests.

1. `/clusters/:namespace/:name/plan`
2. `/api/v1/:namespace/target-source/:name/applyTargets`

```

1  paths:
2    /clusters/:namespace/:name/plan:
3      get:
4        summary: "Get cluster plan."
5        operationId: "getClusterPlan"
6        responses:
7          '200':
8            description: "ClusterPlan returned"
9    /api/v1/:namespace/target-source/:name/applyTargets:
10     post:
11       summary: "Targets received are applied in gNMIC Operator."
12       operationId: "applyTargets"
13       security:
14         - bearerAuth: []
15       requestBody:
16         required: true
17         content:
18           application/json:
19             schema:
20               $ref: '#/components/schemas/Targets'

```

Code Listing 4: OpenAPI contract **HTTP** paths

The **REST API** interface is implemented using the Gin Web Framework, more details in [ADR-03](#).

4.3.1. Target Struct

The OpenAPI contract defines the body expected by the applyTargets endpoint, which is a list of targets. Each individual target is expected to have the attributes shown in Table 17, a table generated from the OpenAPI contract. The complete interface documentation is part of [Appendix D.3](#) or the **gNMIC** Operator docs [28].

Name	Type	Description	Notes
name	String	Name of the device being monitored	defaults to ""
address	String	IPv4/IPv6 address or host-name	defaults to ""
port	Integer	gNMI port	optional, gets specified by TargetSource if not present
targetProfile	String	TargetProfile to apply to this Target	optional, gets specified by TargetSource if not present
labels	map[string]string	Input of labels as key:value pairs	defaults to nil
operation	String	Either created, updated, or deleted	defaults to ""

Table 17: REST API target attributes

The OpenAPI contract is converted into Go structs with the openapi-codegen command-line tool [29]. The rest of the push-interface code builds around these structs, ensuring consistency across code and documentation.

4.3.2. Operational Flow of applyTargets

Figure 19 illustrates the operational flow of the `applyTargets` endpoint, with descriptions being present in Table 18. If an error occurs at any step, it is logged by the **gNMIc** Operator, an appropriate **HTTP** status code is returned, and the request is rejected.



Figure 19: Operational Flow of `applyTargets`

Step	Description
POST Request received	An HTTP POST request is received by the Gin router.
TargetSource Lookup	Verifies that the TargetSource encoded into the URI exists.
Verify Authentication	If enabled, the request is validated using bearer token authentication and/or a signature. See Chapter 4.3.3.
Bind Payload to Target Struct	Binds the payload to the target struct shown in Chapter 4.3.1. If the payload does not meet the enforced structure, it is rejected.
Create DiscoveryEvent	The payload is converted into the internal <code>DiscoveryEvent</code> data type and forwarded to the message processor.

Table 18: Operational Flow step description

TargetSource Lookup

The values `:namespace` and `:name` as seen in Code Listing 4 are encoded into the URI. These values are extracted from incoming **HTTP** requests using the Gin feature `Bind Uniform Resource Identifier (URI)` and allow identifying to which TargetSource a request belongs to [30]. If the TargetSource exists, it also checks if the push interface is enabled.

4.3.3. Authentication

The `applyTargets` endpoint supports bearer token authentication and signature verification. Both are turned off by default, and can be used individually or in combination. The technical configuration is documented in the **gNMIc** Operator docs [31].

[ADR-04](#) provides the rationale for selecting bearer token authentication.

Bearer Token

The bearer token is passed in the authorization header in the format shown in Code Listing 5.

```
1 Authorization: Bearer <token-goes-here>
```

Code Listing 5: Bearer token format

A Kubernetes secret stores the bearer token. Code Listing 6 shows the TargetSource struct with bearer token authentication enabled, the secret is referenced as a `tokenSecretRef`,

```
1 push:
2   enabled: true
3   auth:
4     bearer:
5       tokenSecretRef:
6         name: gnmic-api-auth # secret name
7         key: bearer-token # secret key
```

Code Listing 6: Targetsource struct with bearer token authentication

Name, key, and namespace are used to obtain the value of the secret. The authentication is successful, if the token from the authorization header is identical to the value of the Kubernetes secret.

Signature

The signature verification obtains the key in the same way as the bearer token. Keyed-Hashing for Message Authentication (**HMAC**) is used to calculate the message authentication codes (MAC)⁵.

1. Get signature secret key.
2. Use sha512(default) or sha256 cipher **spec**.
3. Calculate MAC with cipher **spec**, **HTTP** body and key.
4. Compare MAC to X-Hook-Signature field in authorization header.

The X-Hook-Signature field contains the MAC of calculated by the sender. If the MACs are not equal, then the body is not verified and the request rejected.

Reverse Proxy

HTTP requests must be transmitted over Transport Layer Security (**TLS**). Otherwise, data is sent in plaintext, which makes authentication insecure and effectively useless. Because the **REST API** interface does not accept Hypertext Transfer Protocol Secure (**HTTPS**) requests directly, **TLS** should be terminated at a reverse proxy, which then forwards the request to the **gNMIc** Operator over **HTTP**.

4.4. HTTP Provider

The **HTTP** provider implements the pull-based TargetLoader and retrieves target data from an external **SoT** via **HTTP** requests. The provider expects the **SoT** to return data in JSON format as this aligns with the response mapping mechanism described in this section.

It is one concrete implementation of the abstraction introduced in Section Chapter 4.2.3, where loaders fetch discovery data and forward it to the MessageProcessor.

Within the hybrid discovery model defined in **ADR-01**, the pull-based mechanism fulfills two responsibilities:

- Initial synchronization when a TargetSource is created or the configuration is updated
- Periodic reconciliation to ensure consistency with the external **SoT**

This complements the push-based mechanism, which delivers low-latency updates but does not guarantee completeness.

⁵MAC, based on context, can also mean Media Access Control.

4.4.1. Architecture and Execution Model

The **HTTP** provider implements the Loader interface and runs as a dedicated goroutine managed by the TargetSource controller. The controller instantiates the appropriate loader implementation based on the TargetSource **spec**. Once started, the loader runs continuously in the background, periodically retrieving data from the **SoT** and transforming it into discovery messages. The resulting DiscoveryMessages are emitted through a channel to the MessageProcessor, which is responsible for reconciling the discovered state with the Kubernetes cluster.

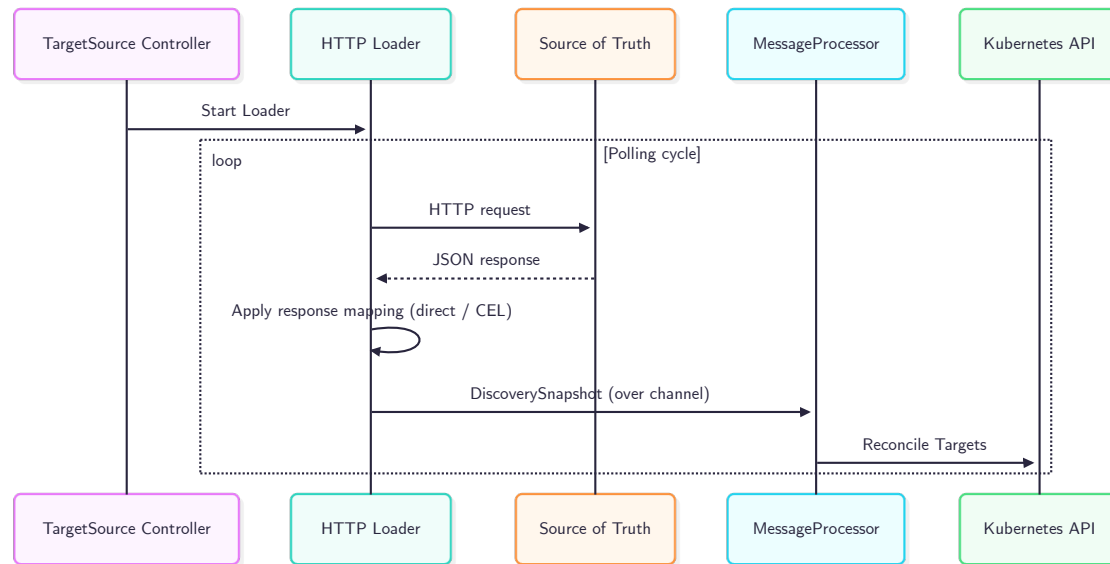


Figure 20: HTTP provider TargetLoader runtime interaction

This design enforces a clear separation of concerns: the **HTTP** provider interacts with the **SoT**, while the MessageProcessor is the only component responsible for applying state changes to Kubernetes resources.

4.4.2. Polling and Synchronization

The **HTTP** provider operates in a periodic polling model consisting of an initial synchronization followed by repeated executions at a configurable interval.

During each polling cycle, the provider sends an **HTTP** request to the configured endpoint, parses the returned JSON response, transforms the data into DiscoveredTarget objects, and emits a DiscoverySnapshot containing the complete discovered state.

Snapshots represent the full set of observed targets and are later processed by the MessageProcessor, which derives individual change events by comparing the snapshot with the current state in the Kubernetes cluster.

4.4.3. HTTP Communication

Requests can be configured using GET and POST **HTTP** methods. Custom headers can be defined, and a request body can be supplied when using POST requests. In addition, timeout values and polling intervals are configurable to control runtime behavior.

Authentication is implemented using Kubernetes secrets. Supported mechanisms include basic authentication as well as token-based authentication. This allows secure integration with protected **APIs** without exposing credentials directly in the configuration.

4.4.4. Response Mapping

The **HTTP** provider transforms JSON responses from the **SoT** into `DiscoveredTarget` objects using a configurable response mapping mechanism.

For each field of a `DiscoveredTarget` (e.g. name, address, labels), the mapping follows a default behavior referred to as *direct mapping*. In this mode, if a JSON field name matches the target field name, the value is copied without modification.

This default behavior allows simple **APIs** with matching schemas to be integrated without additional configuration.

However, real-world **APIs** often differ in structure, naming, or nesting of fields. To support these cases, the mapping can be explicitly defined per field using Common Expression Language (**CEL**) expressions. This allows values to be extracted from nested structures, renamed, transformed, or computed dynamically. The rationale for selecting **CEL** is documented in [ADR-06](#).

This approach enables flexible integration with heterogeneous APIs while avoiding custom parsing logic in the controller.

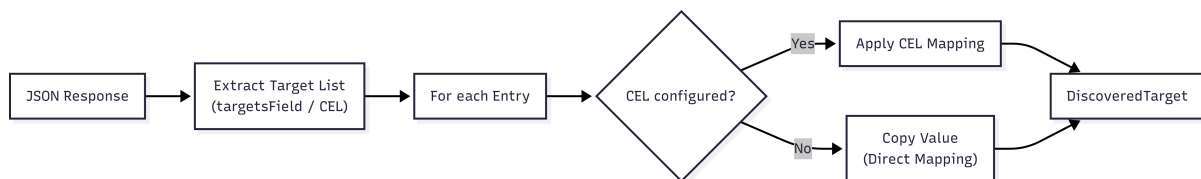


Figure 21: HTTP provider response mapping

To support complex transformations, the following **CEL** extensions are used:

- `ext.Strings`: string manipulation, e.g. splitting Classless Inter-Domain Routing (**CIDR**) notation
- `ext.Math`: numeric operations and comparisons
- `ext.Lists`: working with arrays, including filtering and extraction
- `ext.Sets`: set-based comparisons and membership checks
- `ext.Regex`: pattern matching and validation
- `ext.Bindings`: definition of intermediate variable

4.4.5. Pagination

To ensure complete data retrieval, the **HTTP** provider supports multiple pagination strategies commonly used in **HTTP** APIs. The pagination mechanism was designed to be **API**-agnostic and aligned with established industry standards [32], [33]. The goal is to support different pagination models while maintaining a generic and declarative configuration approach.

Token-based pagination extracts a continuation token from the response body using a **CEL** expression and appends it to subsequent requests. URL-based pagination uses a fully qualified URL returned by the **API** directly for the next request. In addition, link header-based pagination is supported according to RFC 8288, where the next page is referenced via the Link header using the relation `rel="next"` [33].

To ensure robustness, all visited pagination references are tracked within a polling cycle. If a previously visited reference is encountered again, the pagination process is terminated to prevent infinite loops.

4.4.6. Error Handling and Resilience

The **HTTP** provider follows a fault-tolerant design. Request failures do not terminate the loader, allowing the next polling cycle to retry automatically. Invalid objects resulting from mapping errors are skipped but logged for observability. Pagination errors only terminate the current cycle without affecting subsequent executions.

If a synchronization cycle fails, no destructive changes are applied. Previously discovered targets remain unchanged. This behavior aligns with the failure handling strategy described in Chapter 4.2.5 and prevents inconsistencies caused by incomplete data.

4.5. Testing & Validation

4.5.1. Unit Tests

Unit tests are used to verify the behavior of individual components in isolation. Since the auto-discovery feature interacts with multiple external systems, such as the Kubernetes **API** and external **SoTs**, these dependencies were abstracted behind interfaces where possible. This allows implementations to be replaced by mocks during testing and enables the validation of the business logic independently from external systems.

The testing strategy focuses on achieving coverage of all self-contained functions and components. This includes helper functions, target mapping logic, status handling, reconciliation-related utility functions, and other deterministic code paths. By testing these components in isolation, failures can be identified and localized without requiring a running Kubernetes cluster or external **SoT**.

Components whose primary responsibility is interacting with external systems are tested only to the extent possible through mocked interfaces. The correctness of the complete interaction between the auto-discovery feature, Kubernetes, and external systems is verified through integration tests (see Chapter 4.5.2).

4.5.2. Integration Tests

While unit tests validate individual components, integration tests verify the interaction between the auto-discovery feature and the surrounding system. The **gNMIc** Operator already contains an existing integration testing framework, which deploys the operator and related resources inside a Kubernetes test environment. The auto-discovery feature extends these tests with additional resources required for target discovery.

To test the pull-based discovery mechanism, a static HTTP server is deployed as part of the test environment. This server exposes a predefined target inventory which is fetched by the **HTTP** TargetLoader during the synchronization process. A corresponding TargetSource resource is then created and reconciled by the operator.

Additionally, a separate **HTTP** POST request gets sent to the **REST API** endpoint related to the TargetSource, to test the push-based mechanism.

The integration test is considered successful when the TargetSource reaches the Ready condition, indicating that the synchronization process completed successfully and the discovered targets were processed as expected. Verification of the resulting Target resources, their reconciliation into the **gNMIC** configuration, and the subsequent telemetry workflow is already covered by the existing integration tests of the **gNMIC** Operator. Therefore, the additional tests introduced by this thesis focus on validating the integration between the discovery backend and the operator rather than re-testing functionality that is already covered elsewhere.

4.5.3. Usability Test

Given the non-functional requirement 3.1 defined in Chapter 2.3, the target auto-discovery feature must be well documented and simple to configure. The goal therefore is to verify whether the configuration and documentation are understandable for a network engineer.

User Selection

The usability test was conducted with the project advisor and co-advisor. Both participants possess the required expertise in networking, **gNMI**, and Kubernetes, making them well suited to evaluate both the implementation and its usability.

User Story

The network engineer uses NetBox as the **SoT** and aims to automatically load targets into the **gNMIC** Operator.

Success Definition

The usability test is considered successful if:

- Devices defined in NetBox are correctly discovered and created in the **gNMIC** Operator.
- The routers are visible and actively monitored in Grafana.
- Each step of the test procedure produces the expected system behavior.

Conducting Test

Due to the complexity of the development environment⁶, the test environment was prepared in advance on a Linux server.

The test followed the predefined procedure:

1. NetBox is configured to support both push-based (webhook) and pull-based (export template) integration.
2. A TargetSource **CR** is created and applied to configure automatic target discovery in the **gNMIC** Operator.
3. The configuration is verified by checking that targets appear in Kubernetes and that telemetry data is visible in Grafana.
4. A device replacement scenario is executed by creating a new device in NetBox and removing an existing one. The changes are expected to be automatically reflected in the operator and monitoring system.

Any necessary corrections (e.g., IP addresses or metadata) are performed directly in NetBox and should propagate automatically to **gNMIC** Operator.

⁶The development environment requires tools such as Go, Docker, kubectl, Kind, Containerlab, kube-prometheus-stack, NetBox, and **gNMIC** Operator. This setup is not part of the usability test and was preconfigured.

4.5.3.1. Results

The usability test was conducted as an observational study. Participants independently executed the test scenario while verbalizing their thoughts. The author recorded both observed behavior and verbal feedback.

Task Completion

Both participants successfully completed the configuration of the TargetSource resource and verified the integration between NetBox and the **gNMIc** Operator. The measured configuration times of the TargetSource were:

- Advisor: 9 minutes 19 seconds
- Co-advisor: 11 minutes 41 seconds

Both participants achieved the defined success criteria. Devices from NetBox were correctly created as targets in Kubernetes, and telemetry data was successfully displayed in the Grafana dashboard. Additionally, the dynamic update behavior (device creation and removal) functioned as expected.

Observed Issues

The following observations describe challenges encountered by the participants during task execution:

- The structure of the TargetSource configuration was not immediately clear.
- The relationship between the TargetSource and the push- and pull-based mechanisms required additional interpretation.
- While the functionality was understood, the rationale behind defining labels as key-value pairs in the push mechanism was not immediately clear.
- The documentation required repeated navigation, indicating limited structural clarity.

In addition, one unexpected system behavior was observed: the push mechanism was triggered even when it was not explicitly enabled.

Documentation Evaluation

Based on the observed behavior and the feedback provided, the documentation was generally considered correct and suitable for experienced users. However, several improvements are required:

- Missing explanation of push-based configuration, including required fields and request structure.
- Unclear relationship between configuration sections, especially between TargetSource and provider settings.
- Lack of a minimal working example to guide initial setup.
- Minor inconsistencies in naming and insufficient explanation of secret handling.
- Missing troubleshooting guidance (e.g., template formatting issues).

Conclusion

The results show that the feature is functionally correct and usable. However, usability challenges primarily stem from insufficient documentation clarity, structure, and completeness. Addressing these aspects is expected to reduce the learning curve and improve usability for a broader audience.

⁷<https://operator.gnmic.dev/>

4.6. gNMIC Operator Documentation

The **gNMIC** Operator already maintains a documentation page about how to install and configure the environment⁷. As this thesis expands the existing code, the documentation was also updated accordingly [34].

The objective was to integrate into the existing documentation articles for other **CRDs** and give references about the developed **API** and how to use it. Hence, pages for the TargetSource **CRD**, the **HTTP** provider and push mode were written. These pages can be found in [Appendix D.1](#), [Appendix D.2](#), and [Appendix D.3](#), respectively.

Since NetBox was chosen as the primary **SoT** for the development of the auto-discovery feature, an extensive guide describing different NetBox configuration approaches was written and included in [Appendix D.4](#).

5. Validation

This chapter validates the different requirements and elaborates on what is implemented and what limitations are present.

5.1. Functional Validation

Table 19 evaluates the functional requirements defined in Chapter 2.2. Each requirement is rated as:

- *Fulfilled*: All the sub-requirements are fulfilled.
- *Partially fulfilled*: All MVP sub-requirements are fulfilled, but some non-MVP sub-requirements are not fulfilled.
- *Not fulfilled*: At least one MVP sub-requirement is not fulfilled.

Requirement	Rating	Assessment
<i>Synchronization</i>	Fulfilled	All synchronization requirements are fulfilled. The system retrieves device information from the discovery provider, creates targets for newly discovered devices, updates existing targets when device data changes, removes targets for devices that are no longer present, and prevents duplicate targets from the same source.
<i>Event-Based Update</i>	Fulfilled	All event-based update requirements are fulfilled. The system receives update events from the discovery provider and correctly processes creation, update, and deletion events.
<i>Configuration</i>	Fulfilled	All configuration requirements are fulfilled. The system allows configuring NetBox as a discovery provider, setting synchronization intervals, limiting queries, and filtering discovered devices. It also supports extended configuration and mapping of devices to target profiles, as well as label assignment.
<i>Observability</i>	Partially fulfilled	Basic observability is implemented. Discovered and removed devices are logged, and synchronization status information is provided. Statistical information is currently limited to the target count in a TargetSource CR , while broader discovery and process statistics are not implemented.

Table 19: Assessment of the Functional Requirements.

5.2. Non-Functional Validation

The non-functional requirements defined in Chapter 2.3 were validated through a series of dedicated tests. The objective of these tests was to evaluate the performance, scalability, network efficiency, update latency, and usability of the implemented auto-discovery mechanism. All tests were executed in a local Kubernetes environment based on *Kind* using the implementation developed as part of

this thesis. The detailed test protocols can be found in [Appendix E](#) (NFR Usability) & [Appendix C](#) (all other NFRs).

5.2.1. Performance Efficiency

The performance characteristics of the implementation were evaluated using [TEST-01](#). The test measured CPU consumption, memory consumption, and synchronization duration for inventories of increasing size ranging from 10 to 2,000 targets.

The results showed that synchronization completed successfully for all evaluated inventory sizes. Resource consumption increased with the number of managed targets while remaining at a low overall level. For the largest tested inventory containing 2,000 targets, the operator required at most 171 millicores of CPU and 39 MiB of memory. Synchronization completed within five seconds and all Target resources were created successfully.

The measurements indicate efficient resource utilization. Memory consumption remained nearly constant throughout all test scenarios, while CPU consumption increased proportionally with the inventory size. Even for inventories significantly larger than those defined in the requirements, the resource overhead remained low.

Based on the results of [TEST-01](#), NFR 1 (Performance Efficiency) is considered fulfilled.

5.2.2. Network Efficiency

Network efficiency was validated using [TEST-04](#). The objective of this test was to verify that the implementation communicates exclusively with the configured [SoT](#) and does not perform active network discovery or direct communication with discovered devices.

The observations confirmed that synchronization requests were performed only according to the configured synchronization interval. Event-driven updates were processed through the [REST API](#) endpoint without triggering additional synchronization requests. Furthermore, no communication with discovered targets was observed during the discovery process.

These results demonstrate that the implemented solution follows the intended architecture by relying solely on the configured inventory provider and optional event notifications. Consequently, unnecessary network traffic is avoided and no active scanning of network devices is required.

Based on the results of [TEST-04](#), NFR 2 (Network Efficiency) is considered fulfilled.

5.2.3. Update Latency

The responsiveness of the event-driven synchronization mechanism was evaluated using [TEST-03](#). The test measured the time required for changes received through the webhook endpoint to be reflected in the corresponding Target resources.

For each test run, a modification was introduced in the inventory source and propagated through the apiserver interface. The elapsed time between sending the webhook request and observing the updated Target resource was recorded. All measured update times remained well below the defined threshold of 30 seconds.

The results confirm that webhook-based updates provide event-based synchronization between the [SoT](#) and the Kubernetes resources managed by the operator. This allows inventory changes to become effective quickly while avoiding the need for frequent polling operations.

Based on the results of TEST-03, NFR 4 (Update Latency) is considered fulfilled.

5.2.4. Scalability

The scalability of the implementation was evaluated as part of TEST-01. Inventories of increasing size were synchronized in order to assess how resource consumption and synchronization duration evolve as the number of managed targets grows.

The implementation successfully synchronized inventories containing up to 2'000 targets. All Target resources were created correctly and synchronization completed without errors in every scenario. CPU consumption increased approximately linearly with the inventory size, while memory consumption exhibited only moderate growth. Synchronization duration remained low even for the largest inventory, completing within five seconds.

These results indicate that the implemented architecture scales predictably and is capable of managing inventories substantially larger than those required for the minimum viable product. No performance degradation or functional limitations were observed during the evaluation.

Based on the results of TEST-01, NFR 5 (Scalability) is considered fulfilled.

5.2.5. Usability

The usability of the target auto-discovery feature was evaluated based on the nonfunctional requirement 3.1 defined in Chapter 2.3. The requirement states that a network engineer should be able to configure auto-discovery within a given time-frame using the **gNMIc** Operator documentation [35].

The validation was conducted through a practical usability test, in which two participants independently executed a predefined configuration scenario. Both participants successfully configured the TargetSource resource, integrated NetBox, and verified the resulting targets in the **gNMIc** Operator and Grafana.

The measured configuration times were 9 minutes and 19 seconds, and 11 minutes and 41 seconds, respectively, which is well within the defined threshold of one hour. No external assistance was required during the test. It should be noted that the usability test was conducted using a guided task scenario in which key configuration parameters (e.g., resource names) were explicitly provided. This approach reduced the need for participants to independently explore the documentation and derive configuration details. As a result, the test primarily evaluates the users ability to apply the documentation to a defined use case, rather than their ability to fully understand and derive the configuration from first principles. Consequently, the measured configuration times represent a guided configuration effort and may underestimate the effort required for an initial, unguided setup.

However, the results also revealed several usability limitations. In particular, the structure and relationships within the configuration were not immediately clear, and parts of the documentation required additional interpretation. Missing explanations, especially for the push-based integration, and the absence of a minimal working example increased the initial learning effort.

Based on these findings, the documentation was revised and improved accordingly. The structure was refined, missing explanations were added, and clarity was increased to better support first-time users.

After these improvements, the NFR 4 (Usability) is considered fulfilled. The system can be configured within the defined constraints, and the documentation provides sufficient guidance for successful configuration by users with relevant domain knowledge.

5.3. Limitations

Although the implemented solution fulfills the defined functional and non-functional requirements, several limitations remain. Most of these limitations originate from the project scope, external system dependencies, or validation constraints rather than from deficiencies in the proposed architecture itself.

Discovery Provider Integration

The implemented discovery mechanism is based on a generic **HTTP** provider that can retrieve device information from arbitrary external systems. This design was chosen to maximize flexibility and to provide a foundation for integrating additional **SoTs** in the future.

However, the NetBox integration currently relies on configuration and data preparation on the external side. Depending on the deployment scenario, users may need to define export templates or perform data transformations to expose device information in the format expected by the operator. While this approach avoids implementing provider-specific logic directly within the operator, it places part of the integration effort on the inventory system.

This limitation is primarily a trade-off between flexibility and convenience. The architecture remains open for future native integrations that could reduce the required configuration effort for commonly used inventory systems while preserving compatibility with the generic provider model.

Scalability Evaluation Scope

The scalability validation performed within the scope of this thesis focused on environments containing up to 2'000 targets, as defined by the non-functional requirements. The solution successfully met the specified performance objectives within this range.

While larger environments were not evaluated, the implemented reconciliation logic and data structures do not indicate any obvious non-linear bottlenecks. Resource consumption observed during testing increased approximately proportionally with the number of managed targets. Based on these observations, the solution is expected to remain applicable in larger deployments. Nevertheless, additional testing would be required to confirm this assumption and identify potential limitations that may only become apparent at significantly larger scales.

Dependence on Source of Truth Data Quality

The proposed solution assumes that the configured **SoT** contains accurate and complete device information. Consequently, the quality of the generated targets directly depends on the quality of the imported inventory data.

In more advanced deployments, additional metadata may be required to fully describe a target. For example, environments with multiple authentication profiles may require devices to contain information that allows them to be mapped to the appropriate `TargetProfile`. If such information is not present in the inventory system, administrators may need to extend their existing device records accordingly.

However, this is generally a one-time integration effort and aligns with the broader goal of treating the SoT as the authoritative representation of the network inventory. Once the required metadata is maintained in the inventory system, the synchronization process can operate automatically without further manual intervention.

Event Delivery Reliability

Push-based synchronization mechanisms depend on the reliable delivery of events from the external SoT. Network interruptions, temporary service outages, or webhook delivery failures can result in individual update events being missed.

This risk is mitigated by the hybrid discovery approach implemented in the solution. Event-based updates provide low-latency synchronization, while periodic reconciliation ensures that missed updates are eventually detected and corrected. As a result, temporary inconsistencies may occur, but the system is designed to converge back to the desired state during subsequent synchronization cycles.

Since eventual consistency is an accepted characteristic of many distributed systems and Kubernetes-based controllers, this behavior was considered acceptable for the intended use case.

Security Evaluation Scope

Security considerations within this thesis primarily focused on authentication mechanisms and secure communication with external systems. A comprehensive security assessment of the complete solution was outside the scope of the project.

Topics such as fine-grained authorization models, secret rotation strategies, multi-tenancy, audit logging, and formal threat modeling were not investigated in detail. While the implementation follows established Kubernetes practices, additional security evaluations would be beneficial before deployment in highly regulated or security-sensitive environments.

Validation Environment

The validation activities were conducted in a controlled test environment using Kubernetes clusters, simulated workloads, and representative network inventories. This approach enabled repeatable and measurable evaluation of the defined requirements.

Nevertheless, production environments often introduce factors that are difficult to reproduce in a laboratory setting, such as heterogeneous infrastructure, varying network conditions, organizational processes, and unforeseen operational constraints. Although no significant issues are expected based on the validation results, real-world deployments may reveal additional optimization opportunities that were not observable within the scope of this thesis.

6. Conclusion & Outlook

This chapter draws the conclusion of this thesis by discussing the outcome, summarizing the findings of this thesis and outlining future work to further improve the **gNMIC** Operator.

6.1. Outcome

At the beginning of this project, a substantial part of the work was devoted to understanding how the **gNMIC** Operator functions. Extending an existing project introduces challenges, such as understanding the existing code-base, preserving existing functionality or aligning expectations with the maintainer. Given the complexity of the project environment, significant time was spent on setting up the development environment. This included building a solid understanding of the components like Containerlab or NetBox stack.

The use case addressed by this thesis is straightforward: a discovery mechanism should automatically import targets into the **gNMIC** Operator. How to achieve this goal is more complicated, and work was initiated the research phase, where these research questions were answered:

- Which mechanism should be used to discover targets?
- How should the target lifecycle be handled inside the **gNMIC** Operator?

These questions were answered through a combination of research and a survey among network engineers. The resulting findings were presented and validated at the mid-term presentation.

The chosen architecture is a hybrid approach that combines pull- and push-based interfaces. This provides the advantages of both models: pull mechanisms ensure periodic synchronization of the complete target set, while push mechanisms provide event-driven updates.

The central challenge during development was defining the TargetSource **spec** with kubebuilder and designing reconciliation logic, in which target retrieval and target application are treated as separate processes. The design was revised multiple times in close collaboration with our industry partner. An example of the challenges which arose is the many different ways pagination can be implemented. The push-interface replaced an **API** endpoint implementation, thus required caution not to break existing functionality while extending its features.

The final result is a simple-to-configure and flexible solution. The auto-discovery feature can run with both pull and push enabled, or with either mechanism independently. Giving a network engineer flexibility in using the **gNMIC** Operator. During the thesis, development focused on NetBox as the primary **SoT**, but it was designed to be extensible. The pull interface can integrate with any service that exposes an **HTTP** endpoint returning target information in the required format. The push interface is vendor-independent, as long as the service can send target data according to the OpenAPI contract.

Significant effort was also invested in the **gNMIC** Operator documentation. It now includes articles describing the TargetSource **CRD** and its configuration options, together with detailed step-by-step examples. This documentation effort aims to improve usability and, in the long term, increase acceptance and adoption of the **gNMIC** Operator within the networking community.

6.2. Findings

Implementing a Kubernetes operator taught us a great deal about kubebuilder and the Kubernetes patterns that are needed to build an efficient controller. We learned about the need to separate concerns between target retrieval and target application. This keeps the reconciler efficient, while go channels are used to communicate between the separated Go routines.

Regular code reviews helped us reduce over-engineering, and on several occasions we simplified the architecture as a result. It also became clear that not every situation needs a dedicated recovery path. In some cases, it is enough to log an event. We learned to use Kubernetes to our advantage. Instead of differentiating between target creation and update, we can simply apply changes. Or instead of building a complicated recovery mechanism, the component can be restarted.

We also learned how important the human factor is, steering how users are guided toward better decisions. As an example, we initially considered offering multiple ways to store secrets, but ultimately decided to support only Kubernetes secrets.

Communication and promotion turned out to be important. The intersection of networking and cloud computing remains a niche area within the broader IT landscape, so we had to invest considerable effort in explaining the context, its importance, and practical relevance of this thesis. During discussions, we often had to take (multiple) steps back and provide additional background on gRPC, SNMP, or even on the basic importance of network monitoring.

6.3. Future Work

A straightforward enhancement is to make better use of the vendor-independent interfaces by providing additional step-by-step examples and configurations for other SoT platforms. Based on the conducted market research, Nautobot would be a sensible candidate.

The architecture was designed to be extensible, so additional TargetSource providers can be implemented with significantly less effort. NetBox, for example, offers Go API client that could be used, which would eliminate the need for data manipulation on either side. A new integration must parse target information into the DiscoveredEvent and DiscoveredTarget data structures. These can be passed on to the TargetSource logic developed, reusing all the components developed in this thesis.

The gNMIC Operator will continue to grow and advance as an open-source project. gNMI and the gNMIC Operator aim to revolutionize network monitoring and the work of this thesis should provide some aid in this aspect.

Bibliography

- [1] Karim Radhouani, "gNMIc Operator." Accessed: Mar. 04, 2026. [Online]. Available: <https://github.com/gnmic/operator>
- [2] Karim Radhouani, "Introducing gnmic Operator." Accessed: Feb. 08, 2026. [Online]. Available: <https://operator.gnmic.dev/blog/2026-02-08-introducing-gnmic-operator/>
- [3] Laiba Siddiqui, "What is Network Telemetry?." Accessed: May 12, 2026. [Online]. Available: https://www.splunk.com/en_us/blog/learn/network-telemetry.html
- [4] Network Working Group, "A Simple Network Management Protocol (SNMP)." Accessed: May 18, 2026. [Online]. Available: <https://datatracker.ietf.org/doc/html/rfc1157>
- [5] Shelly Cadora, "The limits of SNMP." Accessed: May 18, 2026. [Online]. Available: <https://blogs.cisco.com/sp/the-limits-of-snmp>
- [6] Kuray Karaaslan, "What is SNMP and Why I hate SNMP? Its Role and Limitations in Network Management." Accessed: May 18, 2026. [Online]. Available: <https://kuray.dev/blog/internet-of-things-iot/what-is-snmp-and-why-i-hate-snmp-its-role-and-limitations-in-network-management>
- [7] gNMI contributors, "gRPC Network Management Interface (gNMI)." Accessed: May 18, 2026. [Online]. Available: <https://openconfig.net/docs/gnmi/gnmi-specification/>
- [8] Karim Radhouani, *Configure Devices and Stream Telemetry with Nokia and Open-Source gNMIc*. Accessed: May 18, 2026. [Online Video]. Available: <https://techfieldday.com/video/configure-devices-and-stream-telemetry-with-nokia-and-open-source-gnmic/>
- [9] gNMIc contributors, "gNMIc Documentation." Accessed: May 18, 2026. [Online]. Available: <https://gnmic.openconfig.net/>
- [10] Karim Radhouani, "Resource Model." Accessed: Mar. 10, 2026. [Online]. Available: <https://operator.gnmic.dev/docs/concepts/resource-model/>
- [11] Karim Radhouani, "Architecture." Accessed: Mar. 10, 2026. [Online]. Available: <https://operator.gnmic.dev/docs/concepts/architecture/>
- [12] Karim Radhouani, "Resource Hierarchy Image." Accessed: Mar. 10, 2026. [Online]. Available: https://operator.gnmic.dev/images/resources_model.svg
- [13] Karim Radhouani, "TargetSource." Accessed: Mar. 10, 2026. [Online]. Available: <https://operator.gnmic.dev/docs/user-guide/targetsource/>
- [14] NetBox Labs, "Devices | NetBox Documentation." Accessed: Mar. 14, 2026. [Online]. Available: <https://netboxlabs.com/docs/netbox/models/dcim/device/>
- [15] Nautobot Authors, "Devices and Cabling - Nautobot Documentation." Accessed: Mar. 14, 2026. [Online]. Available: <https://archive.docs.nautobot.com/projects/core/en/v1.0.3/core-functionality/devices/>
- [16] Infrahub Authors, "Labels | Infrahub Documentation." Accessed: Mar. 14, 2026. [Online]. Available: <https://docs.infrahub.app/topics/labels>

- [17] Go, "net/http Documentation." Accessed: Mar. 26, 2026. [Online]. Available: <https://pkg.go.dev/net/http#example-Get>
- [18] NetBox Labs, "REST API Overview." Accessed: Mar. 12, 2026. [Online]. Available: <https://netboxlabs.com/docs/netbox/integrations/rest-api/>
- [19] OpenAPI Initiative, "Structure of an OpenAPI Description." Accessed: Mar. 19, 2026. [Online]. Available: <https://learn.openapis.org/specification/structure.html>
- [20] Kubebuilder Contributors, "kubebuilder." Accessed: Mar. 26, 2026. [Online]. Available: <https://github.com/kubernetes-sigs/kubebuilder>
- [21] Gernot Starke, "arc42 Documentation." Accessed: Apr. 13, 2026. [Online]. Available: <https://docs.arc42.org/home/>
- [22] Kubernetes Authors, "Operator Pattern | Kubernetes." Accessed: Apr. 22, 2026. [Online]. Available: <https://kubernetes.io/docs/concepts/extend-kubernetes/operator/>
- [23] Mark McGranaghan & Eli Bendersky, "Go by Example: Goroutines." Accessed: June 06, 2026. [Online]. Available: <https://gobyexample.com/goroutines>
- [24] Mark McGranaghan & Eli Bendersky, "Go by Example: Channels." Accessed: June 06, 2026. [Online]. Available: <https://gobyexample.com/channels>
- [25] Kubernetes Authors, "Pod Conditions | Kubernetes." Accessed: June 06, 2026. [Online]. Available: <https://kubernetes.io/docs/concepts/workloads/pods/pod-condition/>
- [26] Rodrigo Venturi, "Contract-First with Spring Boot: A Modern Approach to API Development." Accessed: June 09, 2026. [Online]. Available: <https://medium.com/@rodrigoventuri/123/contract-first-with-spring-boot-a-modern-approach-to-api-development-f1a1635cf95b>
- [27] oapi-codegen contributors, "oapi-codegen." Accessed: Apr. 27, 2026. [Online]. Available: <https://github.com/oapi-codegen/oapi-codegen/#generating-server-side-boilerplate>
- [28] Roman Weber, Daniel Schatzmann, and Valentino Diller, "REST API interface." Accessed: June 09, 2026. [Online]. Available: <https://ba-submission.gnmic-operator2.pages.dev/docs/advanced/rest-api-documentation/>
- [29] "oapi-codegen." Accessed: June 09, 2026. [Online]. Available: <https://github.com/oapi-codegen/oapi-codegen>
- [30] Gin Web Framework contributors, "Bind Uri." Accessed: June 09, 2026. [Online]. Available: <https://gin-gonic.com/en/docs/binding/bind-uri/>
- [31] Roman Weber, Daniel Schatzmann, and Valentino Diller, "REST API interface." Accessed: June 09, 2026. [Online]. Available: <https://ba-submission.gnmic-operator2.pages.dev/docs/user-guide/targetsource/push/>
- [32] Google, "Pagination." Accessed: June 08, 2026. [Online]. Available: <https://google.aip.dev/158>
- [33] Internet Engineering Task Force (IETF), "Web Linking." Accessed: June 08, 2026. [Online]. Available: <https://datatracker.ietf.org/doc/html/rfc8288>

- [34] Roman Weber, Daniel Schatzmann, and Valentino Diller, "REST API interface." Accessed: June 09, 2026. [Online]. Available: <https://ba-submission.gnmic-operator2.pages.dev/docs>
- [35] Karim Radhouani, "gNMIc Operator Documentation." Accessed: June 08, 2026. [Online]. Available: <https://operator.gnmic.dev/docs>
- [36] go-http-routing-benchmark contributors, "Go HTTP Router Benchmark." Accessed: Apr. 27, 2026. [Online]. Available: <https://github.com/gin-gonic/go-http-routing-benchmark>
- [37] HunCoding, "Performance Comparison Top 5 HTTP Routers in Go." Accessed: Apr. 27, 2026. [Online]. Available: <https://huncoding.com/go-http-routers-performance-comparison-benchmark>
- [38] Swagger Authors, "Authentication." Accessed: May 18, 2026. [Online]. Available: https://swagger.io/docs/specification/v3_0/authentication/
- [39] "oapi-codegen." Accessed: June 11, 2026. [Online]. Available: <https://github.com/netbox-community/go-netbox>
- [40] Team Virima, "Active vs passive scanning in IT environments." Accessed: Mar. 06, 2026. [Online]. Available: <https://virima.com/blog/active-vs-passive-scanning-in-it-environments>
- [41] Team Virima, "Active vs passive IT asset discovery: when to use each." Accessed: Mar. 06, 2026. [Online]. Available: <https://virima.com/blog/active-vs-passive-it-asset-discovery-which-one-works-better>
- [42] Tom Olzak, "Passive vs. Active Scanning." Accessed: Aug. 08, 2028. [Online]. Available: <https://www.linkedin.com/pulse/passive-vs-active-scanning-tom-olzak-3oiwc>
- [43] tenable, "What is network scanning?." Accessed: Mar. 06, 2026. [Online]. Available: <https://www.tenable.com/cybersecurity-guide/learn/network-scanning#cluster-top>
- [44] Mark Bowen, "Advantages and disadvantages of active vs. passive scanning in IT and OT environments." Accessed: Apr. 24, 2020. [Online]. Available: <https://www.intelligentcio.com/me/2020/04/24/advantages-and-disadvantages-of-active-vs-passive-scanning-in-it-and-ot-environments/>
- [45] A. Azodi, F. Cheng, and C. Meinel, "Event Driven Network Topology Discovery and Inventory Listing Using REAMS," *Wireless Personal Communications*, vol. 94, no. 3, pp. 415–430, Sept. 2015, doi: [10.1007/s11277-015-3061-3](https://doi.org/10.1007/s11277-015-3061-3).
- [46] Martin Kuehnhausen, "Event Driven Architecture - Push vs Pull." Accessed: Mar. 07, 2026. [Online]. Available: <https://engineering.wellsky.com/post/event-driven-architecture---push-vs-pull>
- [47] System Design Interview Roadmap, "Push vs. Pull Architectures in Real-Time Systems: When to Poll and When to Push." Accessed: Mar. 07, 2026. [Online]. Available: <https://systemdr.substack.com/p/push-vs-pull-architectures-in-real>
- [48] R. V. Jerry Cuomo, "Mobile Design Patterns: Push, Don't Pull." Accessed: Mar. 07, 2026. [Online]. Available: <https://www.redbooks.ibm.com/redpapers/pdfs/redp5072.pdf>

- [49] Nokia, "Nokia Event Driven Automation." Accessed: Mar. 11, 2026. [Online]. Available: <https://docs.eda.dev/25.12/>
- [50] Tyler Garrett, "The Great Debate: Push vs Pull Data Processing Architectures." Accessed: Mar. 12, 2026. [Online]. Available: <https://dev3lop.com/the-great-debate-push-vs-pull-data-processing-architectures/>
- [51] Nautobot, "REST API Overview." Accessed: Mar. 12, 2026. [Online]. Available: <https://docs.nautobot.com/projects/core/en/stable/user-guide/platform-functionality/rest-api/overview/>
- [52] Consul, "Consul API Overview." Accessed: Mar. 12, 2026. [Online]. Available: <https://developer.hashicorp.com/consul/api-docs>
- [53] NetBox Labs, "Webhooks." Accessed: Mar. 12, 2026. [Online]. Available: <https://netboxlabs.com/docs/netbox/integrations/webhooks/>
- [54] Nautobot, "Webhooks." Accessed: Mar. 12, 2026. [Online]. Available: <https://docs.nautobot.com/projects/core/en/stable/user-guide/platform-functionality/webhook/>
- [55] Consul, "Watches overview." Accessed: Mar. 12, 2026. [Online]. Available: <https://developer.hashicorp.com/consul/docs/automate/watch>
- [56] Roman Weber, Daniel Schatzmann, and Valentino Diller, "Usability testing gNMlc Operator." Accessed: June 08, 2026. [Online]. Available: <https://gitlab.ost.ch/ins-stud/sa-ba/ba-fs26-gnmlc/dev-env>

List of External Tools and Resources

Competence	External Tools / Resources
<i>Component Research</i>	Google, DuckDuckGo, ChatGPT, Copilot
<i>Visualization</i>	Excel, draw.io, Mermaid, Microsoft Whiteboard, PowerPoint
<i>Idea Generation</i>	Microsoft Whiteboard, ChatGPT, Copilot
<i>Translation</i>	Google Translate, DeepL, dict.cc, ChatGPT
<i>Coding & Developing</i>	Visual Studio Code, Go docs, ChatGPT, Copilot
<i>Text Creation & Verification</i>	ChatGPT, Copilot
<i>Collaboration</i>	Microsoft 365, GitLab, YouTrack
<i>DevOps</i>	GitLab, GitHub, Kubernetes

Table 20: List of Tools and Resources

Glossary

API Server A software component that exposes an Application Programming Interface (API) and processes incoming requests from clients.

Application Programming Interface (API) A standardized interface that allows software systems to communicate and exchange data.

arc42 A template and methodology for documenting software architectures using a structured set of architecture viewpoints.

Cloud-Native An approach to designing and operating applications that fully leverages cloud computing concepts such as containers, orchestration, and automation.

Controller A Kubernetes component that continuously observes resources and reconciles the actual state of the system with the desired state.

Custom Resource (CR) An instance of a Kubernetes Custom Resource Definition that represents application-specific configuration or state within a cluster.

Custom Resource Definition (CRD) In Kubernetes, an API extension that defines a new resource type, enabling users to create and manage custom objects as part of the cluster.

Event-Driven Architecture A software design pattern in which components communicate through the production and consumption of events.

High Availability A design approach ensuring that applications or infrastructure remain operational even when individual components fail.

JSON A lightweight, text-based data interchange format used to represent structured data and exchange information between systems.

Kubernetes An open-source container orchestration system, originally developed by Google and now maintained by the Cloud Native Computing Foundation.

Operator In Kubernetes, a controller-based application that extends cluster functionality by managing an application's deployment, configuration, and life cycle through custom resources.

Pagination In an HTTP GET request, a technique that splits a large set of results into smaller pages, allowing clients to retrieve data in chunks.

Reconciliation The process by which a Kubernetes controller continuously compares the desired state of a resource with its actual state and applies changes until both match.

Source of Truth (SoT) An authoritative system that stores and maintains the intended state of network devices and related information.

Target A network device, such as a router or switch, from which telemetry data is collected.

Target Discovery The process of automatically identifying and synchronizing network devices that should be monitored by the gNMIc Operator.

Telemetry The automated collection and transmission of operational data from devices or systems for monitoring and analysis purposes.

Webhook An HTTP-based callback mechanism that enables one system to notify another system when a specific event occurs.

YAML A human-readable data serialization format commonly used for configuration files in Kubernetes and cloud-native applications.

List of Abbreviations

ADR	Architectural Decision Record	MAC	Media Access Control
API	Application Programming Interface	REST	Representational State Transfer
CEL	Common Expression Language	SNMP	Simple Network Management Protocol
CIDR	Classless Inter-Domain Routing	SoT	Source of Truth
CR	Custom Resource	SuD	System under Development
CRD	Custom Resource Definition	TCP	Transmission Control Protocol
EBP	Elementary Business Processes	TLS	Transport Layer Security
HMAC	Keyed-Hashing for Message Authentication	URI	Uniform Resource Identifier
HTTP	Hypertext Transfer Protocol	YAML	YAML Ain't Markup Language
HTTPS	Hypertext Transfer Protocol Secure	YANG	Yet Another Next Generation
IP	Internet Protocol	gNMI	gRPC Network Management Interface
IPv4	Internet Protocol Version 4	gNMIc	gNMI CLI client
IPv6	Internet Protocol Version 6	spec	specification
JSON	JavaScript Object Notation		

List of Tables

Table 1	Use Case Model	4
Table 2	Functional Requirements: Synchronization	10
Table 3	Functional Requirements: Event-based Update	10
Table 4	Functional Requirements: Configuration	10
Table 5	Functional Requirements: Observability	10
Table 6	Non-functional requirements	11
Table 7	Target Discovery Decision Matrix	19
Table 8	Context diagram components	24
Table 9	arc42 building block view L1 description	25
Table 10	arc42 building block view L2 controller description	26
Table 11	arc42 building block view L2 discovery description	27
Table 12	arc42 building block view L2 loaders description	27
Table 13	arc42 building block view L2 http description	27
Table 14	arc42 building block view L2 utils description	27
Table 15	arc42 building block view L2 apiserver description	28
Table 16	TargetSource condition types	34
Table 17	REST API target attributes	36
Table 18	Operational Flow step description	37
Table 19	Assessment of the Functional Requirements.	45
Table 20	List of Tools and Resources	56
Table 21	ADR Target Discovery mechanism	64
Table 22	ADR Target Identification	65
Table 23	ADR REST API Framework	66
Table 24	ADR REST API authentication	67
Table 25	ADR Error Handling	69
Table 26	ADR Response Mapping	70
Table 27	ADR NetBox Pull Interface	71

List of figures

Figure 1	Title page server rack picture source	1
Figure 2	Sequence diagram for UC1	5
Figure 3	Sequence diagram for UC2	6
Figure 4	Sequence diagram for UC3	7
Figure 5	Sequence diagram for UC4	8
Figure 6	Use case diagram	9
Figure 7	Pie chart showing the primary source of truth according to respondents (n=32).	16
Figure 8	Pie chart showing how targets are defined for telemetry systems (n=32).	16
Figure 9	How respondents would want their targets to be defined ideally (n=32).	17
Figure 10	Resource hierarchy of the gNMIc Operator [12].	20
Figure 11	Overview of components evaluated during research.	23
Figure 12	arc42 context view	24
Figure 13	arc42 building block view level 1	25
Figure 14	arc42 building block view level 2	26
Figure 15	Reconciliation flowchart with the high-level flow on the left, and the <i>Reconcile</i> <i>discovery processes</i> sub graph on the right.	29
Figure 16	Flowchart of a <code>DiscoveryEvent</code> being applied	32
Figure 17	The list of <code>DiscoveredTargets</code> gets compared to the existing targets in the Kubernetes cluster to determine whether a target needs to be applied or deleted.	33
Figure 18	Flowchart of the <code>DiscoverySnapshot</code> being handled accordingly and applied as <code>DiscoveryEvents</code>	33
Figure 19	Operational Flow of <code>applyTargets</code>	37
Figure 20	HTTP provider <code>TargetLoader</code> runtime interaction	39
Figure 21	HTTP provider response mapping	40

Code Listings

Code Listing 1	Example CR for a TargetSource object	28
Code Listing 2	DiscoveryMessage implementations	31
Code Listing 3	The StatusUpdate struct holds all status fields which get updated during the discovery process.	34
Code Listing 4	OpenAPI contract HTTP paths	36
Code Listing 5	Bearer token format	37
Code Listing 6	Targetsource struct with bearer token authentication	38

Appendix

A. Architectural Decision Records

The **ADRs** in this chapter capture the design decisions and serve as reasoning for the component selection.

ADR-01: Target Discovery Mechanism

Name	Target Discovery Mechanism
Status	Accepted
Initial proposal date	12. March 2026
Last updated	25. March 2026

Table 21: **ADR** Target Discovery mechanism

Context and problem statement

Target discovery refers to the mechanism by which network devices are automatically imported into the **gNMIC** Operator. Given the existence of several implementation approaches, this **ADR** evaluates these alternatives and identifies the most suitable method.

Decision drivers

As this project implements automatic target discovery, the decision on how to achieve this is the most relevant architectural decision.

Considered options

- Agent based scanning - scanning of network.
- Pull-based - **gNMIC** Operator pulls information from **SoT**.
- Push-based - **SoT** pushes target update to **gNMIC** Operator.
- Hybrid-based - combined approach of pull- and push-based methods.

Decision outcome

In the context of selecting the ideal target discovery mechanism for the **gNMIC** Operator, facing the need to import all targets while reducing the network impact to a minimum, we decided to implement a hybrid-based approach where both push- and pull-based methods are used, and against agent based scanning of the network, to achieve a scalable and network-efficient target discovery backend, accepting the hybrid-based complexity as both push- and pull-based implementations are required.

Consequences

The chosen hybrid-based approach introduces complexity and requires a prioritization of which push- and pull-based method to implement first. See Chapter 3.2.2.

Confirmation

This **ADR** is confirmed when the hybrid-based approach is confirmed to be working.

ADR-02: Target Identification

Name	Target Identification
Status	Accepted
Initial proposal date	13. March 2026
Last updated	31. May 2026

Table 22: **ADR** Target Identification

Context and problem statement

Target identification refers to the mechanism by which discovered devices are matched to existing Target resources within the cluster. Given that different Sources of Truth (**SoTs**) provide different device attributes, this **ADR** evaluates possible identification approaches and determines the most suitable method.

Decision drivers

As this project implements target auto-discovery, a reliable identification mechanism is required to prevent duplicate target resources from being created during reconciliation cycles and to ensure compatibility across different **SoTs**.

Considered options

- IP-based identification by using the device management IP address.
- Name-based identification by using the device name only.
- Combined identification by using the TargetSource name together with the device name.

Decision outcome

In the context of selecting a reliable target identification mechanism for the **gNMIc** Operator, facing the need to uniquely identify devices across reconciliation cycles while supporting multiple tenants and **SoTs**, we decided to implement a combined identification approach where the TargetSource name and device name are used, and against identification based solely on device names, IP addresses, or device IDs, to achieve deterministic and unique target identifiers within the Kubernetes cluster, accepting the requirement that device names must be unique within the scope of a TargetSource.

Consequences

The chosen approach requires the device name to remain stable within the originating TargetSource. However, it allows devices with identical names to exist in different tenants or discovery domains by separating them through dedicated TargetSource resources. It also avoids relying on IP addresses, which may change over time.

Confirmation

This **ADR** is confirmed when discovered devices can be consistently reconciled to the same Target resources without creating duplicates and when devices with identical names from different TargetSource resources can coexist successfully.

ADR-03: REST API Framework

Name	REST API Framework
Status	Accepted
Initial proposal date	7. April 2026
Last updated	27. April 2026

Table 23: **ADR REST API** Framework

Context and problem statement

The **gNMIc** Operator had a simple **HTTP** router implemented using the net/http package from the Go standard library. The auto-discover feature requires an **REST API** interface to receive targets sent from a webhook.

This **ADR** aims to identify an appropriate package for implementing the aforementioned **REST API** interface.

Decision drivers

- Compatibility with openAPI standard
- Path parameters/**URI** binding or dynamic endpoints⁸
- Performance
- Code readability
- Actively maintained
- Documentation

Considered options

- net/http
- gin-gonic
- echo

Other **HTTP** routers like Bunrouter or Fiber exist, but were not considered.

Decision outcome

In the context of selecting an **HTTP** router, facing the need of a high-performance interface with openAPI standard capability, we decided for **gin-gonic** due to detailed documentation, improved code readability, support for **URI** binding, and usage experience in the project team, and against net/http or echo, accepting that the other considered options are considered equally valid from a technical perspective⁹.

⁸A route should only be active, when a targetSource exists with push enabled.

⁹All considered options support the openAPI standard [27]. Performance: all implementations should be able to handle the workload [36], [37].

Consequences

- Existing net/http implementation is replaced by gin-gonic.
- gin-gonic is not part of the Go standard library.

Confirmation

The use of gin-gonic is confirmed when: a) the technical implementation works, and b) the performance passes the benchmark tests in this project.

ADR-04: REST API authentication

Name	REST API authentication
Status	Accepted
Initial proposal date	17. May 2026
Last updated	30. May 2026

Table 24: ADR REST API authentication

Context and problem statement

An unauthenticated interface allows any user to tamper with targets monitored by the **gNMIc** Operator. This poses a security risk shows the requirement for authentication.

This ADR aims to identify the fitting authentication method.

Decision drivers

- Authentication should be simple to configure for the network engineer.
- Authentication must be secure.
- The **gNMIc** Operator has no user context.

Considered options

- HTTP authentication schemes
 - Basic
 - Bearer token
- API keys
- OAuth 2.0

All considered options are supported by the openAPI standard [38].

Decision outcome

In the context of selecting an authentication method for the REST API interface, facing the need secure it against unauthenticated requests, we decided to use a **bearer token**, and against all other considered options , to achieve authentication without a user context where the bearer token can be passed in the authentication header, accepting that everyone in possession of bearer token can use the REST API,

Consequences

The bearer token requires TLS encryption to be secure.

Confirmation

This ADR is confirmed when the code implementation has been reviewed and tested.

More Information

In addition to bearer authentication, verification with an X-Hook-Signature has been added.

ADR-05: Error Handling

Name	Error Handling
Status	Accepted
Initial proposal date	13. May 2026
Last updated	21. May 2026

Table 25: ADR Error Handling

Context and problem statement

Target discovery relies on information retrieved from external systems and therefore must account for temporary failures, invalid responses, and other operational issues. Given that multiple components participate in the processing of discovered targets, this ADR evaluates how error handling and logging responsibilities should be distributed.

Decision drivers

As this project focuses on implementing the target discovery backend, error handling and logging should be limited to failures that directly affect the discovery process. Validation and error handling already performed by Kubernetes and the gNMIc Operator should not be duplicated.

Considered options

- Comprehensive validation by validating and logging all discovered data within the discovery backend.
- Minimal validation by only handling and logging failures directly related to discovery.
- Shared validation by distributing validation responsibilities across the discovery backend, Kubernetes, and the gNMIc Operator.

Decision outcome

In the context of implementing a reliable target discovery backend for the gNMIc Operator, facing the need to handle discovery failures while avoiding redundant validation logic, we decided to implement a shared validation approach where the discovery backend only handles and logs errors that directly impact target discovery, and against comprehensive validation within the discovery backend itself, to achieve a simpler and more maintainable implementation with clear responsibility boundaries, accepting that validation and error reporting for downstream processing are delegated to Kubernetes and the gNMIc Operator.

Consequences

The chosen approach reduces implementation complexity and prevents duplication of existing validation mechanisms. Discovery failures, such as unavailable SoTs or incomplete discovery results, are handled and logged by the discovery backend. Validation errors related to Kubernetes resources or runtime configuration are handled by the respective downstream components.

Confirmation

This **ADR** is confirmed when discovery-related failures are correctly handled and logged by the discovery backend, while validation and reconciliation errors continue to be reported by Kubernetes and the **gNMIc** Operator.

ADR-06: Response Mapping

Name	Response Mapping
Status	Accepted
Initial proposal date	21. May 2026
Last updated	2. June 2026

Table 26: **ADR** Response Mapping

Context and problem statement

The **HTTP** provider of the **gNMIc** Operator requires a mechanism to extract and transform data from **API** responses into Target resources.

The loader is intentionally limited to JSON-based **APIs**, as **JSON** is the de facto standard for modern **HTTP APIs**. This reduces implementation complexity and aligns with mapping approaches that operate naturally on JSON-like data structures.

This **ADR** evaluates different approaches to response mapping and identifies the most suitable mechanism.

Decision drivers

- Ability to handle nested and complex **JSON** structures
- Support for conditional logic and transformations
- Expressiveness and flexibility
- Performance
- Ease of use for network engineers
- Integration with Go-based implementation

Considered options

- JSONPath
- JMESPath
- Custom transformation logic (Go code or scripting)
- **CEL**

Decision outcome

In the context of selecting a response mapping approach for the **HTTP** provider, facing the need to transform heterogeneous **JSON** responses from various **SoT** systems, we decided for **CEL** as the primary mapping mechanism, and against JSONPath, JMESPath, and custom transformation logic, to achieve flexible and expressive mappings that support conditional logic, data normalization, and structured field construction, accepting increased complexity compared to simpler extraction mechanisms.

Consequences

- Direct mapping is still used where no transformation is needed (for performance).
- Users must learn **CEL** syntax for complex mappings.
- **CEL** introduces a small runtime evaluation overhead.
- **CEL** enables advanced transformations.
- Compared to JSONPath and JMESPath, **CEL** provides greater flexibility and better integration, while benefiting from active development and support.

Confirmation

This **ADR** is confirmed when:

- The code implementation has been reviewed
- Complex real-world **APIs** can be mapped without external preprocessing

ADR-07: NetBox Pull Interface

Name	NetBox Pull Interface
Status	Accepted
Initial proposal date	6. May 2026
Last updated	11. June 2026

Table 27: **ADR** NetBox Pull Interface

Context and problem statement

The pull-based discovery mechanism of the **gNMIc** Operator requires integration with an external **SoT** to retrieve network device information. NetBox is used as the primary **SoT** in this project and exposes both a **REST API** and a GraphQL interface. In addition, a Go client library for the NetBox API exists [39].

This **ADR** evaluates the available integration options for consuming data from NetBox and determines the most suitable approach for implementing the pull-based discovery interface.

Decision drivers

- Simplicity of integration and implementation effort
- Flexibility to adapt to different **APIs** structures

Considered options

- NetBox REST API
- NetBox GraphQL API
- NetBox Go client library

Decision outcome

In the context of implementing a pull-based integration with NetBox for target discovery, facing the need to integrate a **SoT** in a flexible and simple way, we decided to use the NetBox **REST API** as the interface for the HTTP-based TargetLoader, and against using the GraphQL API or the NetBox Go client library,

to achieve a simple, transparent, and decoupled integration, accepting reduced query flexibility compared to GraphQL and foregoing convenience abstractions provided by the Go client.

Consequences

- The implementation remains simple
- Tight coupling to NetBox-specific libraries is avoided
- JSON responses integrate directly with the response mapping mechanism
- GraphQL-specific capabilities such as selective querying are not utilized
- Some client-side processing may be required to compensate for limited query flexibility

Confirmation

This **ADR** is confirmed when:

- The HTTP loader successfully retrieves and processes data from NetBox using the REST API
- The response mapping mechanism can transform NetBox responses into DiscoveredTarget objects

B. Target Discovery Research

B.1. Agent-Based Target Discovery

The **gNMIc** Operator deploys pod-based agents that identify **gNMI** targets through active or passive network scanning and report the identified targets and associated metadata back to the operator.

Active Scanning

The agents perform network discovery by sending probes to identify devices and extract their associated attributes.

Upsides

- Network visibility: Active scanning detects newly connected devices and disrupted connections, providing a view of the network state [40], [41], [42].
- Target metadata: Active scanning extracts **gNMIc** relevant device attributes, including **IP** and **MAC** addresses as well as exposed ports [40], [42], [43].

Downsides

- Limitations: Some devices may not respond reliably to active scans, potentially causing service disruptions or performance degradation [41], [42], [44].
- Scalability: Active scanning requires explicit configuration of scanning parameters (e.g., **IP** ranges, protocols, credentials) and network adjustments (e.g., opening firewall ports), making it difficult to manage at scale. In large organizations, this process is often time-consuming and requires coordination across multiple teams [40].
- Performance: Active scanning can cause significant network overhead especially in large-scale network environments [40], [41], [44], [45].
- Accuracy: Active scanning may fail to detect certain devices, for example when firewall rules block probes or when network issues occur [40], [42].
- Stale information: Due to the performance impact, scans are often run infrequently, leading to stale information about the network state [45].
- Protocol dependency challenges: Many established Internet Protocol Version 4 (**IPv4**) scanning techniques do not translate effectively to Internet Protocol Version 6 (**IPv6**) and become inefficient due to the significantly larger address space [45].
- Target honesty: Agents depend on targets to accurately and truthfully report their information. Rogue, compromised, or misconfigured devices may provide false data, leading to incorrect discovery results and introducing security risks [45]. Establishing such trust is nontrivial, as it requires mechanisms to authenticate devices, and many devices do not support such capabilities.
- Target metadata: Authentication credentials cannot be discovered through active scanning, they must be known and maintained beforehand.
- Maintenance overhead: Active scanning requires ongoing management and monitoring of distributed agents across the network.
- Redundant scanning infrastructure: When tools rely on separate scanners, the network becomes burdened with numerous scanning processes, increasing overhead and producing divergent discovery results between tools.

Passive Scanning

The agents perform passive discovery by monitoring network traffic or by processing log data to detect devices and extract their attributes [41].

Upsides

- Deployment automation: Passive scanning supports automated deployment processes, since newly appearing devices can be identified automatically through continuous observation of their network communications [40].
- Resource efficiency: Passive scanning does not generate additional network traffic, making it more efficient and less disruptive to the network [40], [41], [42], [43], [44].
- Accuracy: Passive scanning can provide a view of the network state, as it captures real traffic and interactions between devices, rather than relying on periodic scans that may miss transient devices or changes in the network. It can also leverage historical traffic or log data to identify devices that were previously overlooked [40], [41].

Downsides

- Accuracy: Passive scanning can miss devices that produce little or no traffic, as well as those located on network segments that are not part of the monitored scope [42], [43], [44].
- Target metadata: The completeness of metadata obtained through passive scanning is inherently limited. **MAC** addresses are only visible when the monitoring point resides on the same Layer-2 broadcast domain. Protocol capabilities cannot be inferred until a device actually generates corresponding traffic [42]. Therefore, **gNMI** support remains undetected until the device communicates using the protocol. Furthermore, authentication credentials cannot be obtained through passive scanning, they must be known and maintained beforehand.

B.2. Pull-Based Target Discovery

The **gNMIC** Operator periodically pulls newest information from the Source of Truth (**SoT**).

Upsides

- Control consume rate: The **gNMIC** Operator controls the rate at which it events are receives. This allows the consuming service to pull new data when it is ready [46].
- Controlled throttling: If the **gNMIC** Operator struggles under heavy load, the pull rate is decreased reducing load [47].
- Fault tolerance: The **gNMIC** Operator can detect a failed pull and can re-pull [46].
- Simplicity: Pulling is simple to implement by using a Hypertext Transfer Protocol (**HTTP**) call [47].
- Application support: Most applications that serve as an **SoT** support **HTTP** requests.

Downsides

- Scaling: The pull-based approach scales badly in big networks. It increases load on the network and on targets [48].
- Fixed pull interval: Short pull interval leads to unneeded pulls, high interval leads to a high latency.
- Unneeded pulls: Pulling of data that has not changed leads to unnecessary traffic [48].
- Higher latency for changes: New data only arrives on the next pull [47].

B.3. Push-Based Target Discovery

A target update within the **SoT** triggers a push of the update to the **gNMIC** Operator.

Upsides

- Event-based updates: Target updates are immediately pushed by the SoT [47].
- Scaling: Push-based performs better in bigger networks. IBM states that in a mobile banking scenario, the push based design performed approximately 25 times more efficient than a pull-based (query-based) design [48].
- Application support: Webhooks are supported by most applications that serve as an SoT.
- Efficiency: There is less network traffic as packets are only pushed after an update.

Downsides

- Unpredictable load: Push-based messages are not distributed evenly, which can result in unpredictable peaks [47].
- Back-pressure problems: The SoT might push quicker than the gNMIc Operator can handle [47].
- Potential inconsistencies: Updates might get lost, which can lead to target inconsistencies between gNMIc Operator and SoT.

B.4. Hybrid-Based Target Discovery

Hybrid-based does use both pull- and push-based methods to maintain a complete target list. It requires two integrations, but is often used to get the advantage of both methods [47].

Depending on the application, the method of how the hybrid-based approach is implemented varies. Three examples are:

- Amazon CloudWatch: Push for critical alarms, pull for metric queries [47].
- Nokia Event Driven Automation: Uses both push and pull approaches [49].
- Stripe's Webhook system: "Pushes payment events but provides a pull API for reconciliation" [47].

Upsides

- Best of both worlds: Combines strengths of push & pull [47], [50].
- Target update: Event-based target updates with push and guarantee of data consistency using pull [50].
- Polling: Little to no unneeded polling due to higher interval.

Downsides

- Two integrations: Both pull and push target discovery must be implemented.
- Complexity: Logic required to merge targets from push and pull messages.
- Performance: Unpredictable load and back-pressure problems can still occur [47].

B.5. Source of Truth Compatibility

The sources listed in this chapter show that popular Sources of Truth (SoTs) are compatible with the evaluated interfaces.

Pull: The gNMIc Operator queries the REST API endpoint of inventory tools to obtain a complete list of targets. NetBox, Nautobot and Consul¹⁰ all have an REST API endpoint [18], [51], [52].

¹⁰The list of inventory tools is not complete.

Push: Target changes in inventory tools trigger a webhook, which sends a target update as a JSON to a custom REST API endpoint of the gNMIC Operator. NetBox, Nautobot, and Consul support webhooks [53], [54], [55]. The REST API endpoint is developed as part of this thesis.

C. Test Protocols

TEST-01: Performance and Scalability Validation

Test Information

Property	Value
Test ID	TEST-01
Test Name	Performance and Scalability Validation
Related Requirements	NFR 1.1 Performance Efficiency, NFR 5.1 Scalability
Date	2026-06-08
Environment	Kind Kubernetes Cluster

Objective

Validate that the implemented auto-discovery mechanism can synchronize inventories of increasing size while maintaining acceptable resource consumption and synchronization performance.

Test Setup

Infrastructure

- Kubernetes Cluster: Kind
- Kubernetes Version: v1.35.0
- gNMLc Operator Version: Branch ba-submission
- Static Inventory Provider: HTTP-based inventory service running on local machine (not within Kubernetes)
- Metrics Collection: `kubectl top pod` (metrics-server installed)

Test Data

The inventory provider shall be configured to return inventories of varying sizes.

Scenario	Number of Targets
S1	10
S2	50
S3	100
S4	200
S5	500
S6	1000
S7	2000

Test Procedure

For each inventory size:

1. Deploy the inventory dataset.
 2. Ensure the operator is running and idle.
 3. Record baseline CPU and memory consumption.
 4. Create the TargetSource resource.
 5. Allow synchronization to complete.
 6. Record:
 - Peak CPU usage during synchronization
 - Peak memory usage during synchronization
 - Synchronization duration
 7. Verify that all targets were created successfully.
-

Measurements

Targets	CPU Peak (<i>millicores</i>)	Memory Peak (<i>MiB</i>)	Sync Duration (<i>s</i>)
10	5	15	< 1
50	5	19	< 1
100	4	22	< 1
200	16	23	1
500	36	23	1
1000	51	25	3
2000	133	32	5
2000	171	38	5
2000	110	39	5

Functional Verification

Targets	Expected Targets Created	Actual Targets Created	Result
10	10	10	☒
50	50	50	☒
100	100	100	☒
200	200	200	☒
500	500	500	☒
1000	1000	1000	☒
2000	2000	2000	☒

Evaluation Criteria

The test is considered successful if:

- Synchronization completes successfully for all inventory sizes.
- No synchronization errors occur.
- Resource consumption remains within the defined limits.

- Synchronization duration increases proportionally with inventory size.
 - All expected Target resources are created.
-

Result Summary

Observations

Both the CPU and memory usage increase proportionally to the target number.

Conclusion

☒ Passed

Comments: *The scenario with 2000 targets was repeated three times to ensure consistency these values are not based on one single test.*

TEST-02: Reliability and Recovery Validation

Test Information

Property	Value
Test ID	TEST-02
Test Name	Reliability and Recovery Validation
Related Requirements	NFR 3.1 Reliability
Date	2026-06-08
Environment	Kind Kubernetes Cluster

Objective

Validate that the implemented auto-discovery mechanism remains operational under failure conditions and is able to recover automatically without manual intervention.

Test Setup

Infrastructure

- Kubernetes Cluster: Kind
- Kubernetes Version: v1.35.0
- gNMIc Operator Version: Branch ba-submission
- Static Inventory Provider: HTTP-based inventory service running on local machine (not within Kubernetes)

Test Data

A static inventory containing 10 targets was used throughout all scenarios.

Test Procedure

Scenario S1 – Discovery Provider Unavailable

1. Verify successful synchronization of all targets.

2. Stop the inventory provider service.
3. Wait for the next synchronization cycle.
4. Observe the TargetSource status and operator logs.
5. Restart the inventory provider service.
6. Wait for the next synchronization cycle.

Expected Result

- Synchronization fails gracefully.
 - The error is reflected in the TargetSource status (condition Stalled).
 - Existing targets remain unchanged.
 - Synchronization resumes automatically once the provider becomes available again.
-

Scenario S2 – Missed Webhook Event

1. Verify successful synchronization of all targets.
2. Modify a target in the inventory provider.
3. Do not send a webhook event.
4. Verify that the operator state remains unchanged.
5. Wait for the next synchronization cycle.
6. Observe the target state.

Expected Result

- The target state remains unchanged until the next synchronization.
 - The periodic synchronization detects the change.
 - The target state is updated automatically.
-

Scenario S3 – Operator Restart

1. Verify successful synchronization of all targets.
2. Restart the operator pod.
3. Wait until the replacement pod becomes ready.
4. Verify that all targets remain present.
5. Modify the inventory provider.
6. Wait for synchronization.

Expected Result

- The operator restarts successfully.
 - Existing targets remain available.
 - Synchronization resumes automatically.
 - No manual intervention is required.
-

Results

Scenario	Expected Behavior	Observed Behavior	Result
S1	Graceful failure and automatic recovery	As expected	☑
S2	Consistency restored by periodic synchronization	As expected	☑

Scenario	Expected Behavior	Observed Behavior	Result
S3	Automatic recovery after restart	As expected	☒

Evaluation Criteria

The test is considered successful if:

- The operator remains operational throughout all scenarios.
- No targets are unintentionally deleted.
- Synchronization resumes automatically after failures.
- Consistency between inventory and Target resources is restored.

Result Summary

Observations

Scenario 1: The TargetSource shows the correct status condition Stalled and keeps target definitions.

```
> kubectl describe targetsource http-ts
Name:          http-ts
Namespace:     default
# --- omitted for brevity
Kind:          TargetSource
# ---
Spec:
  Provider:
    Http:
      Interval:  1m0s
      Method:    GET
      Timeout:   30s
      URL:       http://10.22.197.226:5500/10-targets.json
  Target Profile: default
Status:
  Conditions:
    Last Transition Time:  2026-06-08T22:51:29Z
    Message:              HTTP endpoint not available
    Reason:               SyncFailed
    Status:               True
    Type:                 Stalled
  Last Sync:              2026-06-08T22:50:31Z
  Observed Generation:    1
  Targets Count:          10
Events:                  <none>
```

Conclusion

☒ Passed

Comments: *CURL request used in scenario 2 to simulate the webhook (localhost:8081 is the port forwarding to the api service):*

```
curl -X POST http://localhost:8081/api/v1/default/target-source/http-ts/applyTargets \
```

```
-H "Content-Type: application/json" \  
-d '[  
  {  
    "address": "192.168.1.10:57400",  
    "name": "router-01",  
    "operation": „created“,  
    "profile": "default",  
    "labels": [  
      {  
        "Key": "tags",  
        "Value": "lab,core"  
      }  
    ]  
  }  
'
```

TEST-03: Update Latency Validation

Test Information

Property	Value
Test ID	TEST-03
Test Name	Update Latency Validation
Related Requirements	NFR 4.1 Update Latency
Date	2026-06-08
Environment	Kind Kubernetes Cluster

Objective

Validate that changes received through the event-based interface are propagated to the corresponding Target resources within the required time limit.

Test Setup

Infrastructure

- Kubernetes Cluster: Kind
- Kubernetes Version: v1.35.0
- gNMIc Operator Version: Branch ba-submission
- Static Inventory Provider: HTTP-based inventory service running on local machine (not within Kubernetes)
- Webhook Endpoint: Enabled

Test Data

A static inventory containing ten targets was used. Changes were applied to an existing target and propagated through the webhook endpoint.

Test Procedure

For each test run:

1. Verify that the targets are synchronized and present in Kubernetes.
 2. Using a CURL POST request, simulate a webhook being sent to the endpoint.
 3. Record the timestamp immediately before sending the webhook request.
 4. Send the webhook request.
 5. Observe the corresponding Target resource.
 6. Record the timestamp when the Target resource reflects the change.
 7. Calculate the update latency.
 8. Repeat the test multiple times.
-

Measurements

Run	Webhook Sent	Target Updated	Latency (s)
1	2026-06-09T03:47:55Z	2026-06-09T03:47:55Z	< 1
2	2026-06-09T03:48:12Z	2026-06-09T03:48:12Z	< 1
3	2026-06-09T03:48:21Z	2026-06-09T03:48:21Z	< 1
4	2026-06-09T03:48:33Z	2026-06-09T03:48:33Z	< 1
5	2026-06-09T03:48:49Z	2026-06-09T03:48:49Z	< 1

Functional Verification

Run	Expected Result	Actual Result	Result
1	Target updated correctly	As expected	☒
2	Target updated correctly	As expected	☒
3	Target updated correctly	As expected	☒
4	Target updated correctly	As expected	☒
5	Target updated correctly	As expected	☒

Evaluation Criteria

The test is considered successful if:

- All updates are propagated successfully.
 - No webhook requests are lost.
 - The measured latency remains below the defined threshold.
 - The resulting Target resource reflects the expected state.
-

Result Summary

Observations

Logs for scenario 1:

```
[GIN] 2026/06/09 - 03:47:55 | 200 | 304.75µs | 127.0.0.1 | POST "/api/v1/default/target-source/http/applyTargets"
{"level":"info","ts":"2026-06-09T03:47:55Z","msg":"Received discovery event","component":"message-processor","targetsource":"http","namespace":"default","event":"APPLY","target":"router-01"}
{"level":"info","ts":"2026-06-09T03:47:55Z","msg":"applied target object","component":"message-processor","targetsource":"http","namespace":"default","name":"http-router-01"}
```

Conclusion

☑ Passed

TEST-04: Network Efficiency Validation

Test Information

Property	Value
Test ID	TEST-04
Test Name	Network Efficiency Validation
Related Requirements	NFR 2.1 Network Efficiency
Date	2026-06-08
Environment	Kind Kubernetes Cluster

Objective

Validate that the implemented auto-discovery mechanism minimizes network traffic by communicating exclusively with the configured inventory source and processing event-based updates without performing active network discovery.

Test Setup

Infrastructure

- Kubernetes Cluster: Kind
- Kubernetes Version: v1.35.0
- gNMIc Operator Version: Branch ba-submission
- Static Inventory Provider: HTTP-based inventory service running on local machine (not within Kubernetes)
- Webhook Endpoint: Enabled

Test Data

A static inventory containing 10 targets was used.

Test Procedure

Scenario S1 – Periodic Synchronization

1. Configure a synchronization interval of 60 seconds.
2. Start the inventory provider.

3. Start the operator.
4. Observe incoming requests on the inventory provider.
5. Record the number of requests during a five-minute period.

Expected Result

- The operator performs one inventory request per synchronization interval.
 - No additional requests are generated.
-

Scenario S2 – Event-Based Update

1. Verify successful synchronization.
2. Modify a target in the inventory provider.
3. Send a webhook request.
4. Observe operator and provider logs.

Expected Result

- The update is processed through the webhook endpoint.
 - No additional inventory synchronization is triggered.
-

Scenario S3 – Communication Scope

1. Synchronize the inventory.
2. Observe network activity and operator logs.
3. Verify communication targets.

Expected Result

- The operator communicates only with the inventory provider and webhook endpoint.
 - No direct communication with discovered targets occurs.
 - No active network scanning is performed.
-

Results

Scenario	Expected Behavior	Observed Behavior	Result
S1	One request per synchronization interval	As expected	☒
S2	Update processed through webhook	As expected	☒
S3	Communication limited to configured sources	As expected	☒

Evaluation Criteria

The test is considered successful if:

- Synchronization requests occur only according to the configured interval.
 - Event-driven updates are processed through the webhook endpoint.
 - No communication with discovered targets is observed.
 - No active network discovery is performed.
-

Result Summary

Observations

The communication only happens when necessary, i.e. once per synchronization interval or once per webhook event.

Conclusion

☒ Passed

D. Operator Documentation Pages

D.1. TargetSource

The TargetSource resource enables dynamic discovery of network devices from external sources. The operator automatically creates, updates, and deletes Target resources based on discovered devices.

Basic Configuration

```
apiVersion: operator.gnmic.dev/v1alpha1
kind: TargetSource
metadata:
  name: targetsource-1
spec:
  provider:
    # Configure one of the supported providers
  targetPort: 57400
  targetProfile: default
  targetLabels:
    source: inventory
```

The supported TargetSource providers are documented on the [TargetSource Provider](#) page.

Spec Fields

Field	Type	Required	Description
provider	object	Yes	Provider-specific discovery configuration. Exactly one provider must be configured
targetPort	int32	No	Default port used when the discovered target does not provide a port
targetProfile	string	No	Reference to default TargetProfile applied to all discovered targets if no profile was discovered
targetLabels	map[string]string	No	Labels added to all discovered targets

Label Inheritance

Each generated Target receives an ownership label identifying the originating TargetSource:

```
operator.gnmic.dev/targetsource: targetsource-1
```

This label is automatically managed by the operator and is used to:

- Identify targets owned by a specific TargetSource
- Determine which targets should be updated or deleted during reconciliation

The operator.gnmic.dev/targetsource label is reserved and always takes precedence over any provider-supplied labels.

TargetSource Labels

Additional labels can be applied to all generated targets using spec.targetLabels:

```
apiVersion: operator.gnmic.dev/v1alpha1
kind: TargetSource
metadata:
```

```
name: targetsource-1
spec:
  provider:
    http:
      url: http://targetsource-1:8080/targets
  targetLabels:
    datacenter: dc-a
    environment: production
```

All targets discovered from this source will have:

- datacenter: dc-a
- environment: production

This enables Pipelines to select targets using label selectors.

Labels from Discovery Providers

Discovery providers may return additional labels for each target. These labels are applied directly to the generated Target resource.

The `gnmic_operator_` label prefix is reserved for operator-specific behavior. Labels using this prefix are interpreted by the operator and are not applied directly to the generated Target resource.

Supported operator labels:

Label	Description
<code>gnmic_operator_target_profile</code>	Overrides the TargetProfile configured in the TargetSource

Label Precedence

If the same label key is defined in multiple places, labels are applied in the following order (highest precedence first):

1. TargetSource ownership label (`operator.gnmic.dev/targetsource`)
2. Labels from TargetSource.spec.targetLabels
3. Labels returned by the discovery provider

Status

The TargetSource status shows discovery state:

```
status:
  status: Synced
  observedGeneration: 1
  targetsCount: 42
  lastSync: "2024-01-15T10:30:00Z"
```

Field	Description
<code>status</code>	Current sync status (Synced, Error, Pending)
<code>observedGeneration</code>	Generation of the spec last processed by the controller
<code>targetsCount</code>	Number of targets discovered
<code>lastSync</code>	Timestamp of last successful sync

Lifecycle

Target Creation

When a TargetSource discovers a new device:

1. A new Target resource is created
2. The TargetProfile referenced in spec.targetProfile is assigned
3. Labels from spec.targetLabels are applied
4. The TargetSource is set as the owner reference

Target Updates

On each discovery cycle, existing Target resources are reconciled with the latest discovered state:

1. The corresponding Target resource is updated and overwritten
2. Clusters consuming the target are reconciled automatically

Manual changes to Target resources managed by a TargetSource are overwritten on every reconciliation cycle.

Target Deletion

When a device is no longer returned by the discovery provider:

1. The corresponding Target resource is deleted
2. Clusters automatically stop using the target

TargetSource Deletion

When a TargetSource is deleted:

1. All Target resources owned by it are deleted via owner references
2. Clusters are reconciled and remove the deleted targets

D.2. HTTP Provider

Basic Configuration

```
apiVersion: operator.gnmic.dev/v1alpha1
kind: TargetSource
metadata:
  name: targetsource-1
spec:
  provider:
    http:
      url: http://inventory-service:8080/targets
      authentication:
        token:
          scheme: Bearer
          tokenSecretRef:
            name: inventory-token
            key: token
      # Enable push mode
      push:
        enabled: true
      targetPort: 57400
```

```
targetProfile: default
targetLabels:
  source: inventory
```

HTTP Spec Fields

Field	Type	Required	Default	Description
url	string	No	-	HTTP endpoint used to pull targets. Required unless push.enabled is enabled
method	string	No	GET	HTTP method used for requests
headers	map[string]string	No	-	HTTP headers to include in requests
body	string	No	-	Request body for POST requests
authentication	object	No	-	Authentication configuration for the HTTP endpoint
interval	duration	No	30m	Polling interval used to refresh targets
timeout	duration	No	30s	Timeout for HTTP requests
tls	object	No	-	Client TLS configuration for HTTPS endpoints
pagination	object	No	-	Pagination configuration for parsing HTTP responses
mapping	object	No	-	Response mapping configuration for JSON responses
push	object	No	-	Push-based update configuration

Pull Mode

The HTTP provider supports pull-based target discovery by periodically querying a remote HTTP endpoint that returns target data in JSON format.

```
spec:
  provider:
    http:
      url: http://inventory-service:8080/targets
```

In pull mode, the operator sends HTTP requests to the configured url at a fixed interval and updates targets based on the response. The push.enabled field is optional when pull mode is enabled, but can still be used for accepting incoming webhook notifications.

How Pull Mode Works

1. The operator sends an HTTP request to the configured url
2. The response is parsed (either directly or via mapping)
3. Targets are created, updated, or removed based on the returned data
4. This process repeats according to the configured interval

Authentication

The HTTP provider supports authenticated requests to the inventory endpoint.

Exactly one authentication method can be configured.

Basic Authentication

Credentials are referenced from a [Kubernetes Secret](#).

```
spec:
  provider:
    http:
      url: https://inventory.example.com/targets
      authentication:
        basic:
          credentialSecretRef:
            name: inventory-credentials
            key: username
```

Token Authentication

Token authentication is configured using a [Kubernetes Secret](#) reference.

```
spec:
  provider:
    http:
      url: https://inventory.example.com/targets
      authentication:
        token:
          scheme: Bearer
          tokenSecretRef:
            name: inventory-token
            key: token
```

TLS

TLS settings can be configured for HTTPS endpoints.

```
spec:
  provider:
    http:
      url: https://inventory.example.com/targets
      tls:
        insecureSkipVerify: false
        caBundleRef:
          name: inventory-ca
          key: ca.crt
```

TLS Fields

Field	Type	Required	Description
<code>insecureSkipVerify</code>	bool	No	Skip verification of the server certificate. Defaults to false
<code>caBundleRef</code>	object	No	Reference to a Kubernetes ConfigMap containing a PEM-encoded CA bundle

Pagination

Pagination enables the operator to retrieve complete result sets from APIs that return data in multiple pages. The operator automatically follows pagination until no further pages are available.

```
spec:
  provider:
    http:
      url: https://inventory.example.com/devices
      pagination:
        nextField: "self.next"
```

Pagination Fields

Field	Type	Required	Description
nextField	string	No	CEL expression used to extract the next page reference from the response
requestParam	string	No	Query parameter used when the extracted value is a token

The nextField value may either contain:

- A full URL for the next request
- A pagination token appended as a query parameter to the original URL

How Pagination Works

The operator handles the following pagination patterns:

1. Link Header Pagination

If the API provides a Link response header with rel="next", the operator will automatically follow it.

Example response header:

Link: <https://api.example.com/devices?page=2>; rel="next"

Behavior:

```
Request 1: GET /devices?page=1
Request 2: GET /devices?page=2
Request 3: GET /devices?page=3
...
```

2. URL-Based Pagination

If the response contains a full URL in the body (e.g. "next": "https://..."), it will be used directly.

Example response:

```
{
  "devices": [...],
  "next": "https://inventory.example.com/devices?offset=50"
}
```

3. Token-Based Pagination

If the response contains a pagination token, the operator appends it as a query parameter.

Example:

```
pagination:
  nextField: "self.next_token"
  requestParam: "page_token"
```

Example:

```
GET /devices
-> "next_token": "abc123"
GET /devices?page_token=abc123
```

CEL-Based Extraction

The nextField is evaluated as a CEL expression using:

- self -> entire JSON response

Example:

```
pagination:
  nextField: "self['@odata.nextLink']"
```

This allows extracting values from nested or special keys.

Response Processing

The HTTP provider supports two response processing modes:

- **Default response format:** The endpoint returns a JSON array of target objects.
- **Response mapping:** Custom JSON structures are mapped to target fields using CEL expressions.

If mapping is configured, the custom mapping rules are used. Otherwise, the response itself must be a JSON array.

Default Response Format

If mapping is not configured, the endpoint must return a JSON array of objects with the following structure:

Field	Type	Required	Description
name	string	Yes	Name of the generated Target resource
address	string	Yes	Device address (FQDN or IP address)
port	int32	No	Port used for gNMI connections. If omitted, spec.targetPort is used
labels	map[string]string	No	Labels added to the generated Target resource
targetProfile	string	No	Reference to a TargetProfile. If omitted, spec.targetProfile is used

Example response:

```
[
  {
    "name": "spine1",
    "address": "spine1.local",
    "port": 57400,
    "labels": {
      "role": "spine"
    },
    "targetProfile": "spine-profile"
  },
  {
```

```

    "name": "leaf1",
    "address": "leaf1.local",
    "port": 57400,
    "labels": {
      "role": "leaf"
    }
  },
  {
    "name": "leaf2",
    "address": "leaf2.local",
    "port": 57400,
    "labels": {
      "role": "leaf"
    }
  }
]

```

Response Mapping via CEL

When your inventory API's JSON structure differs from the default format, use CEL (Common Expression Language) mapping to extract target fields.

```

spec:
  provider:
    http:
      url: https://inventory.example.com/devices
      mapping:
        targetsField: "self.results"
        name: "item.hostname"
        address: "item.management.ip"
        port: "item.gnmi.port"
        targetProfile: "item.profile"
        labels: "{ 'role': item.metadata.role, 'site': item.metadata.site }"

```

Understanding targetsField

The targetsField expression tells the operator where to find the list of target objects in your API response. It's particularly important when your API wraps the target list in a data structure.

When to use targetsField:

- Your API returns {"results": [...]} -> use "self.results"
- Your API returns {"data": {"devices": [...]]} -> use "self.data.devices"
- Your API returns a plain array [...] -> omit targetsField (default behavior)

Example scenarios:

Custom API response example 1:

```

{
  "count": 42,
  "next": "https://...",
  "results": [
    { "id": 1, "name": "device1", "primary_ip": "10.0.0.1" },
    { "id": 2, "name": "device2", "primary_ip": "10.0.0.2" }
  ]
}

```

Usage: targetsField: "self.results"

Custom API response example 2:

```
{
  "status": "success",
  "data": {
    "timestamp": "2024-01-01T00:00:00Z",
    "devices": [
      {"name": "router1", "mgmt_ip": "192.168.1.1"},
      {"name": "router2", "mgmt_ip": "192.168.1.2"}
    ]
  }
}
```

Usage: targetsField: "self.data.devices"

Mapping Fields

Field	Type	Required	Description
targetsField	string	No	CEL expression selecting the target list from the response. If omitted, assumes response is a direct JSON array
name	string	No	CEL expression for the target name
address	string	No	CEL expression for the target address
port	string	No	CEL expression for the target port
labels	string	No	CEL expression returning a map of labels
targetProfile	string	No	CEL expression for the target profile

CEL Variables

The mapping expressions support the following variables:

- item: the current target object being processed
- self: the complete unprocessed response from the HTTP endpoint

Performance: CEL vs Direct Mapping

Understanding the performance implications helps optimize your configurations:

Direct Mapping (No CEL) - *Fastest*

- Used when your API response matches the default structure exactly
- No expression compilation or evaluation overhead
- Suitable for high-frequency polling (e.g., every minute)
- Example: API returns [{"name": "...", "address": "..."}]

CEL Mapping - *Slight overhead*

- CEL expressions are compiled once at startup (not per request)
- Evaluation is performed per target object during each poll cycle
- At high scale (10,000+ targets), consider the interval between polls

Best practices:

- Use direct mapping if your API already returns the correct structure

- For large result sets, increase the interval
- Combine CEL and direct mapping for efficiency (see hybrid mapping below)
- Use CEL extensions (see reference table below) to reduce complexity and improve readability

CEL Extensions

The operator includes a set of standard CEL extensions from the official [CEL Go library](#) to enable more advanced expressions.

These [extensions](#) expand CEL with additional capabilities commonly needed when transforming API responses:

Extension	Purpose
Strings	String manipulation such as splitting values, case conversion, and extracting parts of text (e.g. parsing hostnames or IPs)
Math	Numeric operations and comparisons (e.g. calculations, min/max, type conversions)
Lists	Working with arrays (e.g. indexing, filtering, joining values)
Sets	Set-style operations such as membership checks and comparisons
Regex	Pattern matching and validation using regular expressions
Bindings	Defining intermediate variables to simplify complex expressions

Examples:

mapping:

```
# Extract site from hostname
labels: |
{
  'site': item.name.split('-')[0]
}

# Conditional profile
targetProfile: "item.type == 'edge' ? 'edge' : 'core'"

# Pattern-based classification
labels: |
{
  'role': item.name.matches('^spine') ? 'spine' : 'leaf'
}
```

Combining CEL and Direct Mapping (Hybrid Approach)

You don't need to map all fields with CEL. The operator supports mixing CEL expressions and direct field lookups for maximum efficiency:

Scenario	Behavior	Use Case
name, address use CEL; others omitted	Extracts mapped fields via CEL; looks for port, labels, targetProfile directly in item JSON	Simple API where only some fields need transformation

Scenario	Behavior	Use Case
Only labels uses CEL	Other fields use direct mapping; labels constructed from CEL expression	API returns correct name, address, port but custom labels need extraction
Only address uses CEL	Direct mapping for other fields; only address requires transformation	Most fields match API exactly except address requires CIDR parsing or format conversion
All fields use CEL	Complete transformation via expressions	API structure completely different from expected format

This hybrid approach optimizes performance by only compiling and evaluating CEL where needed.

Example - Partial CEL mapping (only transform what needs transforming):

```
mapping:
  # Use CEL only when you need to transform a field
  name: "item.hostname"
  address: "item.primary_ip4 != null ? item.primary_ip4.split('/')[0] :
item.primary_ip6.split('/')[0]" # CEL: parse CIDR

  # Fields that already exist should be omitted
  # Port already exists as "port" field in item
  # port: item.port <- omit this

  # Use CEL for structured or derived values
  labels: |
    {
      "site": item.site.name,
      "role": item.device_role.name
    }

  # targetProfile can also be omitted if already present or not needed
```

In this example, only address and labels use CEL expressions; name, port, and targetProfile use direct field lookups for efficiency.

Using YAML | for Complex CEL Expressions

When writing more complex CEL expressions, it is recommended to use YAML's pipe (|) literal block instead of inline strings.

This is especially useful for expressions that span multiple lines or contain nested logic.

Labels example:

```
mapping:
  labels: |
    {
      "site": item.site.name,
      "rack": item.rack != null ? item.rack.name : "",
      "role": item.role != null ? item.role : "unknown",
      "tags": item.tags.join(',')
    }
```

Why use | instead of quoted strings:

- **Readability:** Multi-line expressions are easier to understand
- **Maintainability:** Complex CEL expressions don't require escaping
- **YAML best practice:** Literal blocks handle special characters naturally

Push Mode

The HTTP provider supports webhook-based target updates via `spec.provider.http.push`.

```
spec:
  provider:
    http:
      push:
        enabled: true
```

When `push.enabled` is true, the operator accepts incoming webhook notifications and can update targets without polling a remote endpoint. The `url` field is optional when push mode is enabled, but can still be used for polling and fallback behavior.

See [Push mode](#) for more details.

Recommended Production Settings

When deploying HTTP TargetSource providers in production networks, follow these guidelines to ensure reliable and efficient target discovery:

Polling Configuration

Scenario	Setting	Rationale
Small environment (< 100 targets)	<code>interval: 5m</code>	Frequent updates without excessive load
Medium environment (100-500 targets)	<code>interval: 10m</code>	Balance between freshness and API load
Large environment (500-2000 targets)	<code>interval: 15m</code>	Reduce API polling overhead
Very large environment (2000+ targets)	<code>interval: 30m</code>	Minimize impact on inventory system
High-frequency changes	Use push mode with interval	Enables updates via push while periodic polling ensures completeness and consistency

Timeout Configuration:

```
timeout: 30s # Allows for network latency
```

If timeouts consistently occur, increase interval instead of timeout (don't poll faster)

Authentication & Security

Always use TLS in production:

```
tls:
  insecureSkipVerify: false # Never skip verification in production
  caBundleRef:
```

```
name: inventory-ca-bundle
key: ca.crt
```

For authenticated APIs:

- Store credentials in Kubernetes Secrets
- Rotate credentials periodically
- Use token-based auth when possible (simpler secret rotation)

Example:

```
authentication:
  token:
    scheme: Bearer
    tokenSecretRef:
      name: inventory-api-token
      key: token
```

Pagination & Large Result Sets

Configuration for APIs returning large result sets:

```
pagination:
  nextField: next # Always configure pagination if your API supports it

interval: 30m # Increase interval for large datasets (reduces cumulative API
load)
timeout: 60s # Increase only if individual requests are slow or responses are
large
```

Pagination splits large datasets into multiple smaller HTTP requests. This improves reliability and reduces the likelihood of timeouts compared to fetching a single large response.

Optimization strategies:

- Request API filtering (if supported) to reduce result set size (e.g. ?limit=1000 or ?status=active)
- If the API does not support pagination or filtering increase the timeout
- Consider webhook push mode for frequently-changing inventories (if API supports it)

Mapping Optimization

Use hybrid CEL and direct mapping for performance:

```
# EFFICIENT - Only CEL-transform what needs it
mapping:
  #
  name: "item.hostname" # CEL expression
  # port: (OMITTED) # Direct: exists as "port" in item

  # Only these need transformation -> use CEL
  address: "item.primary_ip.split('/')[0]" # CEL: parse CIDR
  labels: | # CEL: construct from nested fields
    {'site': item.site.name}
```

Avoid unnecessary CEL complexity:

```
# GOOD - Simple expressions
mapping:
  address: "item.management_ip"
```

```
port: "int(item.gnmi_port)"
```

AVOID - Nested ternary logic (hard to debug)

```
mapping:
```

```
  name: "item.has_override ? item.override_name : (item.hostname != '' ?
item.hostname : 'default-' + string(item.id))"
```

CEL expression best practices:

- Compile expressions once at startup (not per request), so complexity is paid only once
- Use ext.Bindings for repeated expressions to avoid redundant evaluation
- Test CEL expressions thoroughly; they're compiled but errors only appear during evaluation
- Keep expressions under 200 characters for maintainability

Example Production Configuration

```
apiVersion: gnmic.openconfig.net/v1alpha1
```

```
kind: TargetSource
```

```
metadata:
```

```
  name: production-inventory
```

```
spec:
```

```
  provider:
```

```
    http:
```

```
      # Security
```

```
      url: https://inventory.prod.example.com/api/dcim/devices/?limit=100
```

```
      tls:
```

```
        insecureSkipVerify: false
```

```
        caBundleRef:
```

```
          name: netbox-ca
```

```
          key: ca.crt
```

```
      # Authentication
```

```
      authentication:
```

```
        token:
```

```
          scheme: Bearer
```

```
          tokenSecretRef:
```

```
            name: api-token
```

```
            key: token
```

```
      # Timing
```

```
      interval: 15m # Balanced update frequency
```

```
      timeout: 30s # Allow for network latency
```

```
      # Pagination
```

```
      pagination:
```

```
        nextField: next
```

```
      # Mapping for fields
```

```
      mapping:
```

```
        targetsField: "self.results"
```

```
        #name: "item.name" -> already handled with fallback direct mapping
```

```
        address: "item.primary_ip4 != null ? item.primary_ip4.split('/')[0] :
item.primary_ip6.split('/')[0]"
```

```
        port: "item.custom_fields.gnmi_port"
```

```
        labels: "{\n          'site': item.site.name,\n          'role':
```

```
item.device_role.name,\n          'status': item.status.value\n        }"
```

```
        targetProfile: "item.custom_fields.gnmi_profile"
```

```
# Global settings
targetPort: 9339
targetProfile: default-profile
```

This configuration ensures:

- Secure HTTPS communication with certificate validation
- API authentication with token-based credentials
- Balanced polling interval for stable environments
- Proper pagination handling for large device inventories
- Rich label extraction from custom fields
- Fallback to defaults when fields are missing

D.3. Push Mode

Basic configuration

This CR enables the push interface with no authentication.

```
apiVersion: operator.gnmic.dev/v1alpha1
kind: TargetSource
metadata:
  name: targetsource-1
spec:
  provider:
    http: # can be changed to a different TargetSourceProvider
  push:
    enabled: true
```

http is currently the only TargetSourceProvider implemented, once others are added they can be used instead. Push mode is not coupled to a specific TargetSourceProvider implementation.

Spec Fields

Field	Type	Required	Default	Description
push	object	No	-	Push interface config
enabled	bool	Yes	False	Whether the push interface is active
auth	object	No	-	Bearer token authentication
signature	object	No	-	HTTP body verification using HMAC
algorithm	string	No	sha512	Algorithm for signature verification (sha256 or sha512)

Address

The REST API endpoint runs on `http://cluster-address:8082/api/v1/:namespace/target-source/:name/applyTargets`.

- `cluster-address`: Address of your cluster.
- `:namespace`: Namespace the TargetSource is created in.

- `:name`: Name of the TargetSource.

See [Push mode with webhook](#) for an example on how to configure the URL.

Cluster Address

The cluster address depends on where the API is accessed from.

- Use `http://<server-fqdn>:8082/` when accessing the API from outside the cluster.
- Use `http://localhost:8082/` for local development (requires port-forwarding).
- Use `gnmic-controller-manager-api.gnmic-system.svc.cluster.local` when NetBox (or another source of truth) runs in the same cluster.
- If you use a reverse proxy, run `kubectl get service -n <gnmic-controller-namespace>` and use the returned service address and port in your proxy configuration.

REST API

Refer to the [REST API documentation](#) for the expected request schema and payload format. Any system or script capable of sending HTTP POST requests can integrate with this interface.

Security

The API supports Bearer Token authentication and X-Hook-Signature, both are optional and **turned off by default**. They are enabled by adding them to the specification. They can also be used in combination.

An example configuration of both is documented in the [Netbox webhook](#) example.

Bearer Authentication

Bearer authentication compares a token stored in Kubernetes with the one sent in the HTTP header. The Kubernetes secret is referenced as `tokenSecretRef`.

```
apiVersion: operator.gnmic.dev/v1alpha1
kind: TargetSource
metadata:
  name: targetsource-1
spec:
  provider:
    http:
      push:
        enabled: true
        auth:
          bearer:
            tokenSecretRef:
              name: gnmic-api-auth # secret name
              key: bearer-token # secret key
```

This requires the [creation](#) of an Opaque Kubernetes secret:

- Must be in the same namespace the gNMIc controller runs in.
- `name`: refers to the secret name
- `key`: key of the secret

- Example: `kubectl create secret generic gnmic-api-auth --from-literal=bearer-token=YOUR_SECRET_TOKEN`

Authorization Header

HTTP request must contain the Bearer token in the header in the format:

Authorization: Bearer YOUR_SECRET_TOKEN

Signature

Signature verification requires an Opaque Kubernetes secret that stores the shared key (see Bearer Authentication). For each request, the HMAC generated from the request body and shared key must be provided in the X-Hook-Signature header.

```
spec:
  provider:
    http:
      push:
        enabled: true
        auth:
          signature:
            algorithm: sha512
            secretRef:
              name: gnmic-signature
              key: signature
```

Reverse Proxy

In order to have a secure setup, the HTTP post requests must be sent using TLS. The REST API interface does not support HTTPS, at least not directly. It is recommended to terminate the TLS connection at the reverse proxy and forward a HTTP request to the gNMIc Operator.

D.4. NetBox Examples

D.4.1. Pull with Export Template

This guide shows how to use **NetBox Export Templates** with the HTTP provider to discover and sync targets.

Export Templates offer powerful filtering, transformation, and formatting directly in NetBox, reducing the load on the operator.

Overview

An **Export Template** is a Jinja2 template that:

1. **Queries** NetBox's internal database (devices, interfaces, etc.)
2. **Filters** results based on custom criteria
3. **Transforms** data into your desired output format
4. **Returns** the formatted output via REST API endpoint

When used with gNMIc's HTTP provider, the operator fetches the rendered JSON template and parses the result with no further transformation needed by the gNMIc Operator.

Prerequisites

- A running Kubernetes cluster with gNMIC Operator installed
 - `kubectl` access to your cluster
 - A reachable NetBox instance with permissions to create Export Templates
 - A NetBox API token
 - Familiarity with Jinja2 templates
-

Step 1: Create a NetBox API Token and Store It Securely

Step 1a: Create the API Token in NetBox

Create a dedicated API token in NetBox for gNMIC Operator access.

1. Log in to NetBox.
2. Open your user profile or go to **User > API Tokens**.
3. Click **Add** or **Add token**.
4. Enter a descriptive name such as gNMIC Operator.
5. Grant the minimum permissions required for read-only device discovery.
6. Copy the token value and store it safely; NetBox will not show it again.

Step 1b: Store the Token in a Kubernetes Secret

Create a [Kubernetes Secret](#) containing the token.

```
# Substitute YOUR_NETBOX_API_TOKEN with your actual token
# Bearer Token Format (v2): nbt_<key>.<token>
kubectl create secret generic netbox-api-token \
  --from-literal=token=YOUR_NETBOX_API_TOKEN \
  -n gnmic-system
```

Verify the Secret was created:

```
kubectl get secret netbox-api-token -n gnmic-system -o yaml
```

Step 2: Create an Export Template in NetBox

Log in to your NetBox instance and navigate to **Customization > Export Templates**.

Step 2a: Create a New Template

Click **Add Export Template** and fill in the details:

Field	Value	Notes
Name	gNMIC Device Export	Descriptive name for your template
Content Type	dcim > device	Export template applies to Device objects
Template Code	(see below)	Jinja2 template
File Extension	json	Output format
Mime Type	application/json	Correct MIME type for JSON

Step 2b: Template Code Example

The following Export Templates only work for devices that have a primary IPv4 address set in NetBox. If `primary_ip4` is missing, the expression returns `''`, so those devices will not yield a valid target address. For NetBox data model details, see the [NetBox Devices Data Model](#) documentation.

See the HTTP provider's "Default Response Format" section for the expected JSON structure:
[HTTP provider docs](#)

Basic Template (All Devices)

```
[
  {% for device in queryset %}
  {
    "name": "{{ device.name }}",
    "address": "{{ device.primary_ip4.address.ip }}",
    "labels": {
      "site": "{{ device.site.name }}",
      "role": "{{ device.role.name }}",
      "region": "{{ device.site.region.name }}",
      "type": "{{ device.device_type.model }}"
    }
  }
  {% if not loop.last %}
  {% endfor %}
]
```

Advanced Template (Filtered by Status and Role)

```
[
  {% for device in queryset.filter(status='active', role__name__in=['leaf', 'spine']) %}
  {
    "name": "{{ device.name }}",
    "address": "{{ device.primary_ip4.address.ip }}",
    "labels": {
      "site": "{{ device.site.name }}",
      "role": "{{ device.role.name }}",
      "region": "{{ device.site.region.name }}",
      "model": "{{ device.device_type.model }}",
      "serial": "{{ device.serial }}",
      "asset_tag": "{{ device.asset_tag }}"
    }
  }
  {% if not loop.last %}
  {% endfor %}
]
```

Key template elements:

- `queryset`: The filtered set of devices (all unless you add `.filter()`)
- `device.name`: Device hostname
- `device.primary_ip4.address.ip`: Primary IPv4 address
- `device.site.name`, `device.device_role.name`: NetBox relationships (site, role, etc.)
- `loop.last`: Jinja2 loop variable to avoid trailing comma on last item

Step 2c: Save and Access the Template

Once saved, NetBox exposes the template via:

`http://netbox.example.com:8000/api/dcim/devices/?export=gNMIC+Device+Export`

Or fetch it directly:

```
# Replace with your NetBox URL and template name
# Substitute YOUR_NETBOX_API_TOKEN with your actual token
# Bearer Token Format (v2): nbt_<key>.<token>
curl -H "Authorization: Bearer YOUR_NETBOX_API_TOKEN" \
  "http://netbox.example.com:8000/api/dcim/devices/?export=gNMIC%20Device%20Export"
```

The response should be a JSON array of targets ready for the gNMIC Operator.

Sample JSON output produced by the basic export template:

```
[
  {
    "name": "edge-rtr-01.dcl.example.com",
    "address": "203.0.113.1",
    "labels": {
      "site": "DC1",
      "role": "edge",
      "region": "eu-central-1",
      "type": "router"
    }
  },
]
```

Ensure the response is valid JSON and contains no hidden or invalid characters, otherwise the gNMIC Operator will fail to parse it.

If you instead return a JSON object with a nested array, add a mapping section such as `targetsField: "self.targets"` to the TargetSource CR.

Step 3: Create a TargetProfile

Define how discovered targets should be configured. The TargetProfile points to a [Kubernetes Secret](#) containing device credentials, such as username/password or client certificates.

Create a credentials Secret first, then reference it from the TargetProfile.

```
# Replace YOUR_DEVICE_USERNAME and YOUR_DEVICE_PASSWORD with your corresponding
# default device username and password
apiVersion: v1
kind: Secret
metadata:
  name: device-credentials
  namespace: gnmic-system
type: Opaque
stringData:
  username: YOUR_DEVICE_USERNAME
  password: YOUR_DEVICE_PASSWORD
apiVersion: operator.gnmic.dev/v1alpha1
kind: TargetProfile
```

```

metadata:
  name: netbox-device
  namespace: gnmic-system
spec:
  credentialsRef: device-credentials
  timeout: 10s

```

For more TargetProfile options and credential handling, see the operator documentation for TargetProfile.

Step 4: Create a TargetSource Using Export Template

Create a TargetSource that references your NetBox export template endpoint:

```

apiVersion: operator.gnmic.dev/v1alpha1
kind: TargetSource
metadata:
  name: netbox-export-source
  namespace: gnmic-system
spec:
  targetPort: 57400
  targetProfile: netbox-device
  targetLabels:
    inventory: netbox
    sync-source: export-template
  provider:
    http:
      url: "http://netbox.example.com:8000/api/dcim/devices/?export=gNMIC%20Device%20Export"
      method: GET
      interval: 30m
      timeout: 30s
      authentication:
        token:
          scheme: Token
          tokenSecretRef:
            name: netbox-api-token
            key: token

```

Step 5: Verify Target Discovery

Once the TargetSource is deployed, verify that targets are being discovered:

```

# List discovered targets
kubectl get targets -n gnmic-system

# Check TargetSource status and sync details
kubectl describe targets netbox-export-source -n gnmic-system

```

Successful sync shows:

- status.status: "success" (or similar)
 - status.targetsCount: number of devices
 - status.lastSync: recent timestamp
-

Example: Complete Setup

Here's a full example combining all components:

```
---
# Secret for NetBox API token
apiVersion: v1
kind: Secret
metadata:
  name: netbox-api-token
  namespace: gnmic-system
type: Opaque
data:
  # base64-encoded token (echo -n "YOUR_TOKEN" | base64)
  token: YOUR_BASE64_ENCODED_TOKEN

---
# Secret for Target Credential
apiVersion: v1
kind: Secret
metadata:
  name: device-credentials
  namespace: gnmic-system
type: Opaque
stringData:
  username: YOUR_DEVICE_USERNAME
  password: YOUR_DEVICE_PASSWORD

---
# TargetProfile
apiVersion: operator.gnmic.dev/v1alpha1
kind: TargetProfile
metadata:
  name: netbox-device
  namespace: gnmic-system
spec:
  credentialsRef: device-credentials
  timeout: 10s

---
# TargetSource using Export Template
apiVersion: operator.gnmic.dev/v1alpha1
kind: TargetSource
metadata:
  name: netbox-export-source
  namespace: gnmic-system
spec:
  targetPort: 57400
  targetProfile: netbox-device
  targetLabels:
    inventory: netbox
    sync-source: export-template
  provider:
    http:
      url: "http://netbox.example.com:8000/api/dcim/devices/?export=gNMIC%20Device%20Export"
```

```
method: GET
interval: 30m
timeout: 30s
authentication:
  token:
    scheme: Token
    tokenSecretRef:
      name: netbox-api-token
      key: token
```

Advantages of Export Templates

- **Powerful Filtering:** Filter devices by site, status, role, tags, etc. directly in NetBox
 - **Reduced Operator Load:** NetBox handles data transformation; operator just fetches JSON
 - **Reusability:** One template can serve multiple consumers
 - **Maintainability:** Update discovery logic in NetBox without changing Kubernetes manifests
 - **Performance:** Avoids REST API pagination for large inventories
-

Limitations & Considerations

1. Reverse Proxy and URL Path Rewriting

If NetBox is behind a reverse proxy with URL path rewriting:

- **Issue:** The export template endpoint uses query parameters that may not survive proxy transformation.
- **Solution:**
 - Ensure the proxy preserves query strings exactly.
 - Test the export URL directly:

```
curl -H "Authorization: Token YOUR_TOKEN" \
  "http://netbox.example.com:8000/api/dcim/devices/?export=gNMIc%20Device%20Export"
```

- If the proxy blocks or modifies parameters, consider using a direct NetBox endpoint without proxying.

2. Large Inventory Rendering

- Very large device counts can cause NetBox to take time rendering the template.
- **Solution:**
 - Use `.filter()` in your template to limit results.
 - Create separate export templates for different device groups (e.g., by site or role).

3. Complex Jinja2 Logic

- NetBox's Jinja2 sandbox restricts some Python functions for security.
 - **Solution:** Keep templates simple and use NetBox's built-in filters and objects. Test the URL with curl or similar before deploying.
-

Template Troubleshooting

Missing Targets in Kubernetes

- **Check:** Are all required fields populated in NetBox? (e.g., `primary_ip4` may be `None` if not set)
- **Solution:** Add conditional checks:

```
{% if device.primary_ip4 %}
  "address": "{{ device.primary_ip4.address.ip }}"
{% endif %}
```

Authorization Fails

If you get a 403 error:

- Verify the token is valid and not expired.
- Ensure the API token is enabled.

D.4.2. Pull with REST API

This guide shows how to configure the HTTP provider to discover targets from NetBox using its REST API.

The REST API approach is direct and straightforward — query NetBox's standard API endpoints to retrieve devices that match your criteria.

Prerequisites

- A running Kubernetes cluster with gNMIc Operator installed
- `kubectl` access to your cluster
- A reachable NetBox instance (inside or outside the cluster)
- A NetBox API token

Overview

The HTTP TargetSource loader performs these steps:

1. **Fetch** JSON device data from a NetBox REST API endpoint (`/api/dcim/devices/`)
2. **Transform** each device record into a gNMIc target using CEL expressions
3. **Create** or **update** Target resources in Kubernetes with the extracted data

Step 1: Create a NetBox API Token and Store It Securely

Step 1a: Create the API Token in NetBox

Create a dedicated API token in NetBox for gNMIc Operator access.

1. Log in to NetBox.
2. Open your user profile or go to **User > API Tokens**.
3. Click **Add** or **Add token**.
4. Enter a descriptive name such as gNMIc Operator.
5. Grant the minimum permissions required for read-only device discovery.
6. Copy the token value and store it safely; NetBox will not show it again.

Step 1b: Store the Token in a Kubernetes Secret

Create a Kubernetes Secret containing the token so it is not embedded in manifests.

```
# Substitute YOUR_NETBOX_API_TOKEN with your actual token
# Bearer Token Format (v2): nbt_<key>.<token>
kubectl create secret generic netbox-api-token \
  --from-literal=token=YOUR_NETBOX_API_TOKEN \
  -n gnmic-system
```

Verify the Secret was created:

```
kubectl get secret netbox-api-token -n gnmic-system -o yaml
```

Step 2: Create a TargetProfile

Define how discovered targets should be configured. The TargetProfile points to a Kubernetes Secret containing device credentials, such as username/password or client certificates.

Create a credentials Secret first, then reference it from the profile.

```
# Replace YOUR_DEVICE_USERNAME and YOUR_DEVICE_PASSWORD with your corresponding
# default device username and password
apiVersion: v1
kind: Secret
metadata:
  name: device-credentials
  namespace: gnmic-system
type: Opaque
stringData:
  username: YOUR_DEVICE_USERNAME
  password: YOUR_DEVICE_PASSWORD
apiVersion: operator.gnmic.dev/v1alpha1
kind: TargetProfile
metadata:
  name: netbox-device
  namespace: gnmic-system
spec:
  credentialsRef: device-credentials
  timeout: 10s
```

For more TargetProfile options and credential handling, see the operator documentation for TargetProfile.

Step 3: Create a TargetSource Using REST API

The following TargetSource queries NetBox's REST API to discover devices:

```
apiVersion: operator.gnmic.dev/v1alpha1
kind: TargetSource
metadata:
  name: netbox-rest-source
  namespace: gnmic-system
spec:
  targetPort: 57400
  targetProfile: netbox-device
  targetLabels:
```

```

inventory: netbox
sync-source: rest-api
provider:
  http:
    url: "http://netbox.example.com:8000/api/dcim/devices/?limit=1000"
    method: GET
    interval: 5m
    timeout: 30s
    authentication:
      token:
        scheme: Bearer
        tokenSecretRef:
          name: netbox-api-token
          key: token
    pagination:
      nextField: "next"
    mapping:
      targetsField: "self.results"
      address: "item.primary_ip4 != null ? item.primary_ip4.address.split('/')[0] :
...
      labels: |
        {
          "site": item.site.name,
          "role": item.device_role.name,
          "model": item.device_type.model,
          "status": item.status.value
        }

```

This mapping only works for devices that have a primary IPv4 address set in NetBox. If `primary_ip4` is missing, the expression returns `''`, so those devices will not yield a valid target address. For NetBox API details, see the [NetBox REST API](#) documentation.

The HTTP loader supports `targetsField` and individual CEL expressions for `name`, `address`, `port`, `labels`, and `targetProfile`. See the HTTP Provider docs “Response Mapping via CEL” section for more details: [HTTP provider docs](#)

Use `self` for the full response and `item` for each candidate object.

Step 4: Apply and Verify Target Discovery

Deploy the `TargetSource` and check that targets are being discovered and synced:

```

# List discovered targets
kubectl apply -f /path/to/targetsource.yaml -n gnmic-system

# List discovered targets
kubectl get targets -n gnmic-system

# Check TargetSource status
kubectl describe targetsource netbox-rest-source -n gnmic-system

```

Look for:

- `status.status`: “success” (or similar)

- `status.targetsCount`: number of discovered devices
 - `status.lastSync`: recent timestamp
-

Example: Complete Setup

Here's a complete example combining all resources:

```
---
# Secret for NetBox API token
apiVersion: v1
kind: Secret
metadata:
  name: netbox-api-token
  namespace: gnmic-system
type: Opaque
data:
  # base64-encoded token (echo -n "YOUR_TOKEN" | base64)
  token: YOUR_BASE64_ENCODED_TOKEN

---
# Secret for Target Credential
apiVersion: v1
kind: Secret
metadata:
  name: device-credentials
  namespace: gnmic-system
type: Opaque
stringData:
  username: YOUR_DEVICE_USERNAME
  password: YOUR_DEVICE_PASSWORD

---
# TargetProfile
apiVersion: operator.gnmic.dev/v1alpha1
kind: TargetProfile
metadata:
  name: netbox-device
  namespace: gnmic-system
spec:
  credentialsRef: device-credentials
  timeout: 10s

---
# TargetSource with REST API
apiVersion: operator.gnmic.dev/v1alpha1
kind: TargetSource
metadata:
  name: netbox-rest-source
  namespace: gnmic-system
spec:
  targetPort: 57400
  targetProfile: netbox-device
  targetLabels:
    inventory: netbox
    sync-source: rest-api
  provider:
```

```

http:
  url: "http://netbox.example.com:8000/api/dcim/devices/?limit=1000"
  method: GET
  interval: 5m
  timeout: 30s
  authentication:
    token:
      scheme: Bearer
      tokenSecretRef:
        name: netbox-api-token
        key: token
  pagination:
    nextField: "next"
  mapping:
    targetsField: "self.results"
    address: "item.primary_ip4 != null ? item.primary_ip4.address.split('/')[0] :
...
    labels: |
      {
        "site": item.site.name,
        "role": item.device_role.name,
        "model": item.device_type.model,
        "status": item.status.value
      }

```

Performance Considerations & Limitations

REST API Query Limits

- **Query Size:** The example uses `limit=1000`. Adjust based on your NetBox instance's pagination settings and response size limits.
- **Response Timeout:** Large device lists can take time. Set appropriate timeouts in your TargetSource.

Reverse Proxy Considerations

If NetBox is behind a reverse proxy:

- **Base URL:** Ensure the reverse proxy correctly handles the `/api/dcim/devices/` path.
- **Authentication:** Some proxies may require additional headers; verify with your proxy and NetBox admin.
- **HTTPS:** If using HTTPS, ensure certificates are trusted by the operator or else use the `tls` setting.

Large Inventories

For inventories with thousands of devices:

- Consider using **Export Templates** (see [NetBox Export Templates]({{< relref "../Export Template" >}})) for better filtering and performance.
 - Implement filtering in the REST API URL (e.g., `?site=us-west&status=active`).
-

Security Considerations

Token and Credentials

- **Never** embed plaintext tokens or credentials in manifests or YAML files.
- Always store tokens in Kubernetes Secrets.
- Restrict RBAC permissions on the Secret to only necessary service accounts.

HTTPS and Certificates

If connecting to NetBox via HTTPS:

- Ensure cluster DNS resolves the hostname correctly.
 - Mount CA certificates if using self-signed certificates.
 - Verify the operator's HTTP client configuration for certificate validation.
-

Troubleshooting

Show TargetSource Errors

```
kubectl describe targetsources netbox-rest-source -n gnmic-system
```

Targets Not Appearing

- Check that the TargetProfile exists and is correctly referenced.
- Verify labels and addresses are being extracted correctly from the NetBox response.
- Review operator logs for parsing errors:

```
kubectl logs -l app=gnmic-operator -n gnmic-operator-system
```

Rate Limiting or Timeouts

Increase the sync interval in your TargetSource or adjust timeouts:

```
spec:
  provider:
    http:
      interval: 1h
      timeout: 1m
```

D.4.3. Push Mode with Webhook

This example walks through configuring a webhook in NetBox to push real-time target updates to the gNMIc Operator. It covers the configuration in the gNMIc Operator (Step 1-3), and the configuration within Netbox (step 4).

1. Create Targetprofile
2. Create Kubernetes Secrets
3. Apply TargetSource
4. Netbox setup a: Configure Webhook b: Create Event Rule
5. Verification

At the end, the logs will show the incoming POST requests and the targets updates can be verified with `kubectl get targets`.

Prerequisites

- Kubernetes cluster with gNMIc Operator installed

- `kubectl` access to your cluster
 - Running NetBox instance
 - Network connectivity from NetBox to the gNMIc Operator API endpoint
-

1. Create TargetProfile

Define how discovered targets should be configured. The `TargetProfile` contains device credentials, such as username/password or client certificates. These are either defined inline strings or stored in a [Kubernetes Secret](#).

```
# Replace YOUR_DEVICE_USERNAME and YOUR_DEVICE_PASSWORD with your corresponding
# default device username and password
apiVersion: v1
kind: Secret
metadata:
  name: device-credentials
  namespace: gnmic-system
type: Opaque
stringData:
  username: YOUR_DEVICE_USERNAME
  password: YOUR_DEVICE_PASSWORD
```

When using a secret, create a credentials Secret first, then reference it from the profile.

```
apiVersion: operator.gnmic.dev/v1alpha1
kind: TargetProfile
metadata:
  name: netbox-device
  namespace: gnmic-system
spec:
  credentialsRef: device-credentials
  timeout: 10s
```

For more `TargetProfile` options and credential handling, see the operator documentation for `TargetProfile`.

2. Create Kubernetes Secrets

Bearer authentication and signature verification both require Kubernetes secrets. Ensure that the secrets:

- Are created in the same namespace as the `TargetSource` (`gnmic-system` in this example).
- Use name and key values that match the `TargetSource` spec.

```
kubectl create secret generic gnmic-api-auth --from-literal=bearer-
token=YOUR_SECRET_TOKEN -n gnmic-system
kubectl create secret generic gnmic-signature --from-
literal=signature=YOUR_SECRET_SIGNATURE -n gnmic-system
```

3. Apply TargetSource

The `TargetSource` has the following settings configured:

- `spec.provider.http.push.enabled` must be set to `true`, otherwise updates are rejected.

- Bearer authentication and signature verification are enabled, referencing to the secrets created in step 2.

```
# netbox.yaml
apiVersion: operator.gnmic.dev/v1alpha1
kind: TargetSource
metadata:
  name: netbox
  namespace: gnmic-system
spec:
  targetPort: 57400
  targetProfile: netbox-device
  targetLabels:
    inventory: netbox
    sync-source: rest-api
  provider:
    http:
      push:
        enabled: true
        auth:
          bearer:
            tokenSecretRef:
              name: gnmic-api-auth
              key: bearer-token
          signature:
            secretRef:
              name: gnmic-signature
              key: signature
```

Namespace is gnmic-system, the name of the TargetSource is netbox. These values will be in the URL in step 4.

4. Netbox Setup

Next, configure a webhook in NetBox. The webhook is triggered by device events (for example, updates) and sends an HTTP POST request to the gNMIc Operator.

Configure Webhook

In NetBox, go to Operations > Webhooks and create a webhook with the following settings:

- **Name:** gNMIc Operator push
- **URL:** `http://gnmic-controller-manager-api.gnmic-system.svc.cluster.local:8082/api/v1/gnmic-system/target-source/netbox/applyTargets`
 - URL contains the namespace gnmic-system and TargetSource name netbox. See section address in [Push Mode](#) for more details on URL construction.
 - gnmic-controller-manager-api.gnmic-system.svc.cluster.local is only reachable if Netbox is inside the cluster.
 - The address may instead be `http://localhost:8082/` or `http://servername:8082/`.
- **HTTP method:** POST
- **HTTP content type:** application/json
- **Additional headers:** Authorization: Bearer YOUR_SECRET_TOKEN

- *Body Template:*

```
[
  {
    "name": "{{ data.name }}",
    "address": "{{ data.primary_ip4.address.split('/')[0] }}",
    "operation": "{{ event }}",
    "targetProfile": "{{ data.custom_fields.target_profile | default('', true) }}",
    "port": {{ data.custom_fields.gnmic_port | default(57400, true) }},
    "labels": [
      {"vendor": "{{ data.device_type.manufacturer.name }}" }
    ]
  }
]
```

- *Secret:* YOUR_SECRET_SIGNATURE

- *SSL Verification:* true

Create Event Rule

The webhook requires a trigger, configured as an event rule under Operations > Event Rules.

- *Name:* gNMIc Operator push target change
- *Object types:* DCIM > Device
- *Event types:* Object Created, Object Updated and Object Deleted
- *Action type:* Webhook
- *Webhook:* gNMIc Operator push

5. Verification

Updating a device in NetBox should now trigger the webhook. Verify this with the following commands:

```
kubectl get targets
kubectl get targets <targetname> -o yaml
```

Check logs of incoming POST requests:

```
kubectl logs -n gnmic-system deploy/gnmic-controller-manager -f
```

Every incoming POST request is logged, including rejected requests. If no POST requests appear in the logs, the webhook request is not reaching the gNMIc Operator.

Example: Complete Setup

Here's a complete example combining all resources:

```
---
# Secret for Target Credential
apiVersion: v1
kind: Secret
metadata:
  name: device-credentials
  namespace: gnmic-system
type: Opaque
stringData:
  username: YOUR_DEVICE_USERNAME
```

```
password: YOUR_DEVICE_PASSWORD

---
# TargetProfile
apiVersion: operator.gnmic.dev/v1alpha1
kind: TargetProfile
metadata:
  name: netbox-device
  namespace: gnmic-system
spec:
  credentialsRef: device-credentials
  timeout: 10s
---
# Apply Targetsource
apiVersion: operator.gnmic.dev/v1alpha1
kind: TargetSource
metadata:
  name: netbox
  namespace: gnmic-system
spec:
  targetPort: 57400
  targetProfile: netbox-device
  targetLabels:
    inventory: netbox
    sync-source: rest-api
  provider:
    http:
      push:
        enabled: true
      auth:
        bearer:
          tokenSecretRef:
            name: gnmic-api-auth
            key: bearer-token
        signature:
          secretRef:
            name: gnmic-signature
            key: signature
```

E. Usability Testing

E.1. Guide for Usability Tests

The following pages present a printed version of the guide used during usability testing [56].

Usability testing gNMlc Operator

This document describes the usability testing process for the auto-discovery feature of the gNMlc Operator.

Requirements

You are expected to have basic knowledge of using Kubernetes, Netbox, and Grafana.

Introduction

Given the complexity of the testing environment, it is prepared on a Linux server with the required tools already installed. These include:

- Go, Docker, kubectl, kind, Containerlab, gNMlc CLI client, make and helm.
- Kind cluster set up and running
- Prometheus and Grafana deployed
- Containerlab running
- Netbox including router data prepared
- All gNMlc Operator CRDs not related to the TargetSource are already applied

For details on the setup or how to reset the testing environment, see [usability-testing-setup.md](#).

User Story

You are a network engineer responsible for operating and monitoring a fleet of routers across your infrastructure.

All devices, including their IP addresses and metadata, are maintained in NetBox, which acts as your single source of truth.

You are using the gNMlc Operator deployed in Kubernetes to collect telemetry and monitor the health of these devices via gNMI. To reduce manual effort and avoid configuration errors, you want the operator to automatically discover and manage targets based on the data stored in NetBox.

Test Rundown

The test roughly consists of the following steps:

1. Configuration in Netbox for both pull- and push-based integration.
2. Configuration of target auto-discovery.
3. Perform an end-of-life (EOL) device replacement
4. Provide feedback and answer open questions

Success Definition

The usability test is considered successful, when the devices from Netbox are applied correctly in the gNMlc Operator. The routers in Netbox can be found in Grafana. Each step resulted in the expected behavior.

Connecting to the VM

The following Linux VM has been prepared: `srbsci-152.ost.ch`. Your SSH keys are added to the VM, to connect use: `ssh ins@srbsci-152.ost.ch`. It is recommended to use `remote-ssh` in Visual Studio Code.

It should be obvious, but a VPN connection is required if you are not in the OST network.

Help and References

This guide is designed to be as clear and easy to follow as possible. However, due to the complexity of the gNMIc Operator setup, additional documentation may be helpful: <https://docs-targetsource-userguide.gnmic-operator2.pages.dev/docs/>.

In case of any problems, feel free to reach out to Roman.

Testing

This is where the interesting part starts, let us set up the auto-discovery feature :smile:.

Port Forwarding

As the services run in a kind cluster, port forwardings are required to make the sites below accessible. Use the command `k-forward-start` in the console, this allows accessing:

- Grafana: <http://srbsci-152.ost.ch:8080/>
- Netbox: <http://srbsci-152.ost.ch:8081/>
- Also the forwarding of port 8082 for the push interface is started.

Log into Grafana

Grafana: <http://srbsci-152.ost.ch:8080/>

- Username: `admin`
- Password: `kubectl get secret prometheus-grafana -o jsonpath="{.data.admin-password}" -n monitoring | base64 --decode` In Grafana, navigate to Starred > "Routers - Target Health". This dashboard displays each router along with the timestamp of the most recent data received. You will need this Dashboard later on.

1. Netbox

The following steps will be done within Netbox <http://srbsci-152.ost.ch:8081/>

Open Netbox with the link provided above. Username is `admin`, password is `netbox`. Netbox already has devices and interfaces configured.

Push-based config

The push-based interface requires a webhook and an event rule. The event rules triggers when a device is updated, and uses the webhook to send a POST request.

Webhook

Configure the Webhook under Operations > Webhooks and create a new Webhook with the following settings:

- *Name*: `GNMIc operator push`

- **URL:** `http://gnmic-controller-manager-api.gnmic-system.svc.cluster.local:8082/api/v1/default/target-source/netbox/applyTargets`
 - The API endpoint listens to `/api/v1/:namespace/target-source/:name/applyTargets`.
 - `:namespace` refers to the Kubernetes namespace the gNMIc operator runs in.
 - `:name` is the name of the TargetSource, NetBox in this case.
- **HTTP method:** POST
- **HTTP content type:** application/json
- **SSL Verification:** true
- **Additional headers:**
 - To obtain the bearer token, use `kubectl get secret -n gnmic-system gnmic-api-auth -o jsonpath="{.data.bearer-token}" | base64 --decode`.

Authorization: Bearer 3rIQ6Y40...

- **Body Template:**

```
[
  {
    "name": "{{ data.name }}",
    "address": "{{ data.primary_ip4.address.split('/')[0] }}",
    "operation": "{{ event }}",
    "targetProfile": "{{ data.custom_fields.target_profile | default('', true) }}",
    "port": {{ data.custom_fields.gnmic_port | default(57400, true) }},
    "labels": [
      {
        "key": "vendor",
        "value": "{{ data.device_type.manufacturer.name }}"
      }
    ]
  }
]
```

Event Rule

To trigger the webhook, an event rule is used.

- **Name:** GNMIc operator push target change
- **Object types:** DCIM > Device
- **Event types:** "Object Created", "Object Updated", "Object Deleted"
- **Action type:** Webhook
- **Webhook:** GNMIc operator push

Pull-based config

The pull-based integration will be configured with an export template. A key is required that allows accessing Netbox.

Export template

Navigate to Customization > Export Templates and create the following template:

- **Name:** GNMIc_operator_pull
- **Object types:** DCIM > Device.
- **As attachment:** false
- **Template code:**

```
[{% for device in queryset %}
{
  "name": "{{ device.name }}",
  "address": "{% if device.primary_ip %}{{ device.primary_ip.address.ip }}{% endif
%}",
  "labels": {
    "vendor": "{{ device.device_type.manufacturer.name }}"
  }{% if device.cf.gnmic_port %},
  "port": {{ device.cf.gnmic_port }}{% endif %}
}{% if not loop.last %},{% endif %}
{% endfor %}]
```

API key

Navigate to the account profile in the top right, go the API Tokens and create a new API Token. **Take note of the “Token” before creating it**, after clicking on “Create” note the Authorization Header in the “Example Usage” this shows the format of the token.

API Token Format: nbt_<Key>.<Token>

2. gNMIc Operator

The following steps will be done on the CLI of `srbsci-152.ost.ch`. In the next step, we'll configure for the gNMIc Operator :partying_face:.

Change to the folder `~/operator` and ensure that the branch `demo/auto-discover-feature` is checked out:

```
cd ~/operator
git checkout demo/auto-discover-feature
```

TargetSource

Info: If you are unable to successfully configure the TargetSource, you can refer to an example configuration located at

`~/operator/lab/dev/resources/targetsources/targetSourceDemo.yaml`

1. Please start a timer
2. Configure the TargetSource CR according to the requirements below using the operator documentation: <https://docs-targetsource-userguide.gnmic-operator2.pages.dev/docs/user-guide/targetsource/>:
 - Name: netbox

- Namespace: default
 - Configure "default" as the default target profile to use
 - Netbox endpoint to pull targets from: http://srbsci-152:8081/api/dcim/devices/?export=gNMIC_operator_pull
 - Pull the targets every 12 hours
 - Create a Kubernetes Secret from the previously obtained bearer token and use it to authenticate with NetBox: `kubectl create secret generic netbox-api-token --from-literal=<secret_selector_key>=<bearer_token_from_netbox>`
3. Apply the TargetSource CR with `kubectl apply -f </path/to/targetsource.yaml>`
 4. Verify that the targets from Netbox (Devices > Devices) were created in Kubernetes as shown in verification.
 5. Stop the time.

Verification:

- There should be shown the following devices leaf1, leaf2, spine1
- The Grafana dashbaord Routers - Target Health the routers leaf1, leaf2, spine1 should continuously receive data (show as green).

```
ins@srbsci-152:~/operator/lab/dev$ kubectl get targets
```

NAME	ADDRESS	PROFILE	CLUSTERS	STATE
netbox-leaf1	172.18.0.5:57400	default	1	READY
netbox-leaf2	172.18.0.6:57400	default	1	READY
netbox-spine1	172.18.0.3:57400	default	1	READY

Target Last Seen			Target Age (Live)		
Time	Router	Last Seen ↓	Time	Router	Age (s) ↑
2026-05-28 12:28:01.021	default/netbox-spine1	2026-05-28 12:28:00	2026-05-28 12:28:01.021	default/netbox-spine1	1.02 s
2026-05-28 12:28:01.021	default/netbox-leaf1	2026-05-28 12:28:00	2026-05-28 12:28:01.021	default/netbox-leaf2	1.02 s
2026-05-28 12:28:01.021	default/netbox-leaf2	2026-05-28 12:28:00	2026-05-28 12:28:01.021	default/netbox-leaf1	1.02 s

3. End-of-life device replacement

The srlinux router leaf1 is end of life and should be replaced by an Arista router. This will be done in two steps:

- Creation of new Arista router in Netbox
- Delete leaf1 router
- *Technically both routers are running, as can be seen in by running `clab inspect` in `/lab/dev`.*

Create Arista router

Navigate to Devices > Devices and add a new device

- Name: ceos1
- Device role: Router
- Device type: Arista > ixr-d2l
- Site: Lab

Navigate to Devices > Interfaces, create a new interface and assign it to ceos1:

- Device: ceos1
- Name: mgmt0
- Type: Virtual

Navigate to IPAM > IP Addresses > IP Addresses, create a new IP Address:

- Use `clab inspect` in `~/operator/lab/dev` to get correct IP.
- Add this IP to the "Address" field (with mask) e.g. `172.18.0.4/32`
- Under Assignment:
 - Assign it to the `mgmt0` interface of `ceos1`
 - Set the IP as the Primary IPv4 address for `ceos1`.
- Create

Once the required attributes name and IP are configured, the device will be created in the gNMIc Operator via the Webhook.

Verification

Once again, verify that the `ceos1` target is created.

```
ins@srbsci-152:~/operator/lab/dev$ kubectl get targets
```

NAME	ADDRESS	PROFILE	CLUSTERS	STATE
netbox-ceos1	172.18.0.4:57400	default	1	DEGRADED
netbox-leaf1	172.18.0.5:57400	default	1	READY
netbox-leaf2	172.18.0.6:57400	default	1	READY
netbox-spine1	172.18.0.3:57400	default	1	READY

It can be seen, that `ceos1` is in state `DEGRADED` and also not showing in Grafana :worried:. That is because the Arista router has different credentials and a different GNMI port. For this, adjust the custom device fields `gnmic_port` and `target_profile` for the `ceos1` router in Netbox with these values

- Gnmic Port: 6030
- Target Profile: eos

This will change the state to `READY` and show all targets as green in the Grafana dashboard.

Target Last Seen			Target Age (Live)		
Time	Router	Last Seen ↓	Time	Router	Age (s) ↑
2026-05-28 12:37:44.471	default/netbox-leaf1	2026-05-28 12:37:30	2026-05-28 12:37:44.471	default/netbox-spine1	14.5 s
2026-05-28 12:37:44.471	default/netbox-spine1	2026-05-28 12:37:30	2026-05-28 12:37:44.471	default/netbox-leaf2	14.5 s
2026-05-28 12:37:44.471	default/netbox-leaf2	2026-05-28 12:37:30	2026-05-28 12:37:44.471	default/netbox-ceos1	14.5 s
2026-05-28 12:37:44.471	default/netbox-ceos1	2026-05-28 12:37:30	2026-05-28 12:37:44.471	default/netbox-leaf1	14.5 s

Delete leaf1

Now that the Arista router is monitored correctly, the end-of-life router `leaf1` can be removed from NetBox. This change can be verified in two ways:

- In Grafana, the status of leaf1 will turn yellow after approximately 30 seconds and red after about one minute, indicating that no new telemetry data is being received.
- In the CLI with `kubectl get targets`, the device will no longer appear in the list of configured gNMI targets.

```
ins@srbsci-152:~/operator/lab/dev$ kubectl get targets
NAME          ADDRESS          PROFILE  CLUSTERS  STATE
netbox-ceos1  172.18.0.6:57400 default
netbox-leaf2  172.18.0.3:57400 default
netbox-spine1 172.18.0.4:57400 default
ins@srbsci-152:~/operator/lab/dev$
```

4. Feedback

This concludes the usability testing. We would appreciate some written feedback and thoughts on:

- How much time did it take to configure the TargetSource?
- Was the [documentation](#) understandable?
- Pull and push Interface configuration
- Anything else you want to say

Thanks for helping us with usability testing!

