POLITECNICO DI TORINO

Master degree course in Cybersecurity

Master Degree Thesis

An OpenC2-Based Framework for Infrastructure Discovery and Kernel-Level Security Enforcement

Supervisor
prof. Daniele Bringhenti
prof. Fulvio Valenza
prof. Matteo Repetto

Candidate
Abdeljabbar Ennajadi

Academic Year 2025-2026

Summary

The increasing complexity of modern cloud-native and hybrid environments has led to a fragmented landscape of security tools, complicating the orchestration of policies across diverse infrastructures. To address this challenge, this thesis explores the application of OpenC2 (Open Command and Control)—a standardized language for the command and control of security functions—as a unified interface for both infrastructure discovery and security enforcement.

The research presents two independent but complementary contributions . The first part is concentrated on Context Discovery, developing specialized modules for Azure Kubernetes Service (AKS) and Proxmox VE. This component operates as a standalone service tasked with the automated identification of network topology, node identities, and IP addressing schemes, translating platform-specific metadata into a standardized architectural view.

In the second half I have implemented an OpenC2 Actuator profile for eBPF (extended Berkeley Packet Filter). This profile enables the remote management of high-performance kernel-level security functions, such as dynamic loading and querying of monitoring programs. By leveraging OpenC2 as the common communication protocol, the proposed architecture demonstrates how a centralized orchestrator can manage heterogeneous tasks—from infrastructure inventory to low-level security enforcement—through a single, vendor-agnostic language.

The results show that this modular approach improves the flexibility of security orchestration in software-defined infrastructures.

The framework achieves scalability by separating environment discovery from eBPF execution. The result is a unified approach to security management that works equally well across cloud services and on-premises infrastructure

Contents

Chapter 1

Introduction

This chapter introduces the context and motivations underlying the thesis. It begins by outlining the key challenges in today's cybersecurity landscape, with particular emphasis on the need for effective and interoperable defense solutions. Within this framework, Open Command and Control (OpenC2) is presented as a promising approach, offering a standardized and consistent communication model for cyber defense operations. The chapter then discusses the main motivations of the thesis, highlighting the need for a standardized method to describe network resources within the broader OpenC2 ecosystem. Finally, it provides an overview of the thesis structure, summarizing the content of the following chapters.

1.1 General Context and Motivation

Modern network defense can no longer rely on manual intervention. As distributed infrastructures rapidly mix on-premises, cloud, and edge resources, the sheer velocity of automated threats demands an equally automated response. Yet, building a cohesive defense is notoriously difficult when coordinating heterogeneous systems from different vendors. Siloed security tools simply do not talk to each other. To bridge this gap, the Open Command and Control (OpenC2) protocol introduces a standardized, vendor-agnostic language designed to make cross-platform interoperability possible

1.2 Motivations

MIRANDA is a European research initiative focused on enhancing cyber defense interoperability through automation, orchestration, and standardized control interfaces [1].

The project aims to provide a common framework where different cybersecurity tools (e.g., sensors, firewalls, orchestration platforms) can exchange commands, context, and telemetry data efficiently. This thesis contributes directly to the MIRANDA ecosystem by developing key actuators and context-gathering tools. Specifically, it focuses on implementing a Proxmox VE discovery actuator

and an AKS discovery actuator, both of which will enable the MIRANDA system to understand and interact with these environments effectively. Proxmox VE is an open-source virtualization platform that combines KVM-based virtual machines and LXC containers. It is widely used for managing virtualized infrastructures, making it a critical component to integrate into the MIRANDA ecosystem. By developing a discovery actuator for Proxmox VE, this thesis aims to provide automated insights into the virtualized resources and their relationships within a Proxmox environment, enabling better orchestration and response capabilities in the context of cybersecurity defense. Instead, the AKS discovery actuator will focus on Azure Kubernetes Service (AKS), which is a managed Kubernetes service provided by Microsoft Azure. Kubernetes has become the de facto standard for container orchestration, and AKS is widely adopted for running containerized applications in the cloud. By creating a discovery actuator for AKS, this thesis will enable the MIRANDA system to gather critical information about the Kubernetes clusters, such as node status, pod configurations, and network policies, which are essential for effective cybersecurity management and response in cloud-native environments. The other point of this is to implement an entire actuator, for eBPF that is a powerful technology for monitoring and tracing in Linux environments. By developing an eBPF actuator, this thesis will enable the MIRANDA system to leverage eBPF's capabilities for real-time visibility into system behavior, network traffic, and security events, enhancing the overall situational awareness and response capabilities of the MIRANDA ecosystem.

1.3 Thesis structure

This thesis is structured as follows:

- **Chapter 2** establishes the necessary foundation, reviewing the OpenC2 protocol and the otupy framework used to build the thesis actuators.
- **Chapter 3** outlines the core research objectives and justifies the selection of Proxmox VE and Azure Kubernetes Service (AKS) as our primary discovery targets.
- **Chapter 4** deepens the technical implementation, detailing how the CTXD actuator handles discovery and data modeling inside Proxmox VE.
- **Chapter 5** shifts focus to the cloud, walking through the AKS discovery actuator's architecture and its integration into the broader MIRANDA ecosystem.
- **Chapter 6** introduces the eBPF actuator, demonstrating how it hooks into Linux environments to provide low-level monitoring and tracing.
- **Chapter 7** concludes the thesis with a summary of our contributions, an honest look at current limitations, and a roadmap for future work.

Chapter 2

OpenC2: Standardized Command and Control for Security Systems

2.1 The OpenC2 Protocol

Open Command and Control (OpenC2) is a suite of standardized specifications designed to enable efficient command and control of cybersecurity systems, components, and services. Its primary objective is to provide a common language through which different cyber defense tools can communicate, regardless of vendor or underlying technology. By promoting interoperability, OpenC2 addresses one of the key challenges in modern cybersecurity environments, where heterogeneous systems must cooperate to detect, analyze, and respond to threats in a coordinated manner. OpenC2 operates on a straightforward request–response paradigm. Within this model, a Producer (such as a central security orchestrator) generates a command, which is sent over a secure transport protocol to a Consumer (a managed device, firewall, or virtualized network function) capable of executing the requested action.

Upon receiving the command, the Consumer executes the action—whether that means blocking a malicious IP, isolating a host, or gathering telemetry. The Consumer then transmits a response message back to the Producer detailing the execution status or returning the requested data.

The primary advantage of this architecture is its strict abstraction: OpenC2 separates the intent of a security action from its underlying implementation details. Because commands use a standardized, vendor-agnostic format, a single command can be understood and executed across different platforms. This separation is what allows organizations to unify disparate security tools into a cohesive, fast-reactig defense .

2.1.1 OpenC2 Structure

The OpenC2 language defines two fundamental payload structures that enable communication between different components of a cyber defense system:

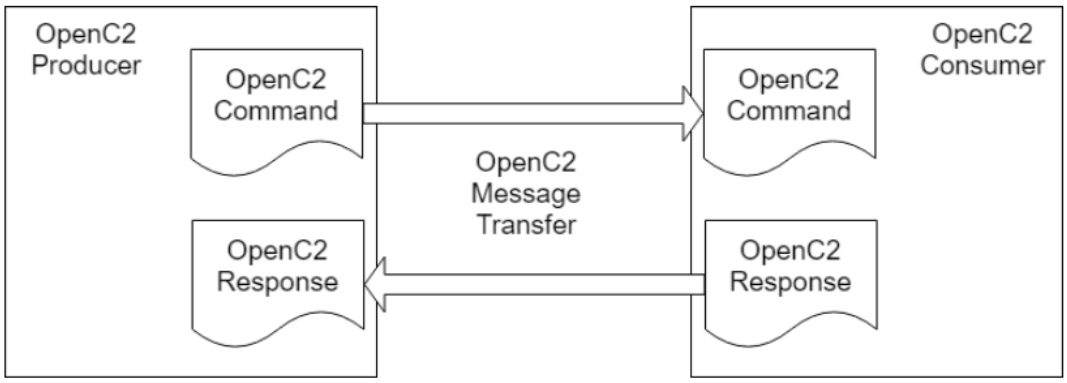

Figure 2.1. OpenC2 overview.

1. **Command**: A Command represents an instruction issued by a system known as the *Producer* to one or more *Consumers*. The Producer is typically a management or orchestration entity that determines the appropriate action to be taken, while the Consumers are the systems, devices, or services responsible for executing that action. Commands encapsulate both the intent (e.g., deny, allow, query) and the target (e.g., IP address, file, process), allowing for clear and structured communication of cybersecurity operations.

2. **Response**: A Response is any information sent back to the Producer as a result of a Command. It provides feedback on the outcome of the requested action, which may include status indicators (such as success or failure), error messages, or additional data explicitly requested by the original Command. Responses play a crucial role in enabling visibility, accountability, and iterative decision-making within automated or semi-automated cyber defense workflows.

In addition to their high-level definition, OpenC2 Commands follow a well-defined internal structure composed of several key elements. A Command typically includes an *action*, which specifies the operation to be performed (e.g., `deny`, `allow`, `query`), and a *target*, which identifies the object of that action, such as an IP address, domain name, file, or process. Optionally, a Command may include an *actuator*, indicating the specific system or class of systems that should execute the command, as well as additional *arguments* that refine or constrain the requested operation. This structured approach allows Commands to be both expressive and flexible, enabling precise control over cyber defense actions while maintaining a consistent syntax across different implementations.

To illustrate, a simple OpenC2 Command could instruct a firewall to block a malicious IP address. In JSON format, such a command might be represented as follows:

```
{
  "action": "deny",
```

```
  "target": {
    "ipv4_addr": "192.168.1.100"
  }
}
```

This example highlights how the action is clearly separated from the target, making the command easily interpretable by any compliant Consumer.

From a conceptual standpoint, OpenC2 is organized into a layered architecture, which promotes modularity, flexibility, and clear separation of concerns. The adoption of this layered model is a deliberate design choice that allows different aspects of the framework to evolve independently, while preserving interoperability across implementations. This approach is similar to other layered models in computer networking, where abstraction simplifies system integration and enhances scalability.

The main layers are described as follows:

- **Secure Transport Layer**: This layer is responsible for providing a reliable and secure communication channel between the Producer and the Consumer. OpenC2 is transport-agnostic, meaning it can operate over a variety of standard communication protocols such as HTTPS, CoAP, and MQTT. The choice of transport depends on the specific deployment scenario, including factors such as latency requirements, resource constraints, and network conditions. Security mechanisms at this layer ensure confidentiality, integrity, and authentication of exchanged messages.

- **Common Content Layer**: This layer defines the overall structure and syntax of OpenC2 Commands and Responses. It establishes a shared set of language constructs and data models used to represent actions, targets, arguments, and other essential elements. By providing a standardized representation, this layer ensures that different systems can correctly interpret and process OpenC2 messages regardless of their internal implementation.

- **Transfer and Encoding Considerations**: In addition to defining the structure of messages, OpenC2 also specifies how these messages are serialized and transmitted. This includes encoding formats such as JSON, as well as message handling conventions. These considerations ensure efficient and consistent communication across heterogeneous environments.

- **Function-Specific Content Layer**: This layer introduces specialized language elements tailored to specific cybersecurity functions, such as network filtering, endpoint protection, or threat intelligence sharing. Each function is associated with an *Actuator Profile*, which defines how a particular class of systems implements the OpenC2 specification. An actuator represents the entity that executes the command, such as a firewall, an endpoint detection system, or an intrusion detection system. Actuator profiles specify supported actions, targets, and expected behaviors, ensuring consistent and interoperable implementations across different platforms.

Together, these layers form a cohesive architecture that supports scalable, interoperable, and extensible cyber defense operations. By clearly separating communication, structure, and function-specific logic, OpenC2 facilitates the integration of diverse security technologies into a unified and coordinated ecosystem.

Despite its advantages, the adoption of OpenC2 also presents certain challenges. These include the need for widespread industry support, the complexity of implementing actuator profiles across diverse systems, and the difficulty of integrating legacy technologies into a standardized framework. Nevertheless, its benefits in terms of interoperability, automation, and scalability make OpenC2 a promising approach for the future of cyber defense.

2.1.2 Comparison with Existing Frameworks

Otupy is compared extensively against other OpenC2 implementations, most notably **Lycan**, the reference implementation by OASIS.

Otupy vs. Lycan

The sources highlight several architectural differences that favor Otupy for scalability and maintainability:

- **Data Modeling:** Lycan derives types from a common ancestor that stores parameters in a flat dictionary, whereas Otupy strictly follows the semantic structures of the OpenC2 grammar .
- **Validation:** Otupy inherits safety controls from existing Python libraries (e.g., `ipaddress`), while Lycan relies on internal `Property` classes that often fail to enforce standard compliance.
- **Flexibility:** Lycan hard-codes JSON serialization, making format changes difficult; Otupy's meta-serializatio makes such transitions smooth.
- **Communication Stack:** Unlike Lycan, which lacks a transport protocol implementation, Otupy provides a complete stack allowing the creation of custom combinations of formats and protocols without internal changes.

2.2 Otupy Framework

Otupy framework is introduced as a flexible, extensible, and portable open-source Python implementation designed to provide an Application Programming Interface (API) for the creation, exchange, and parsing of Open Command and Control (OpenC2) messages [2]. As cybersecurity environments grow increasingly complex, the proliferation of heterogeneous security functions necessitates a vendor-agnostic abstract language like OpenC2 to manage remote command and control. Otupy addresses existing limitations in open-source implementations, which often suffer from

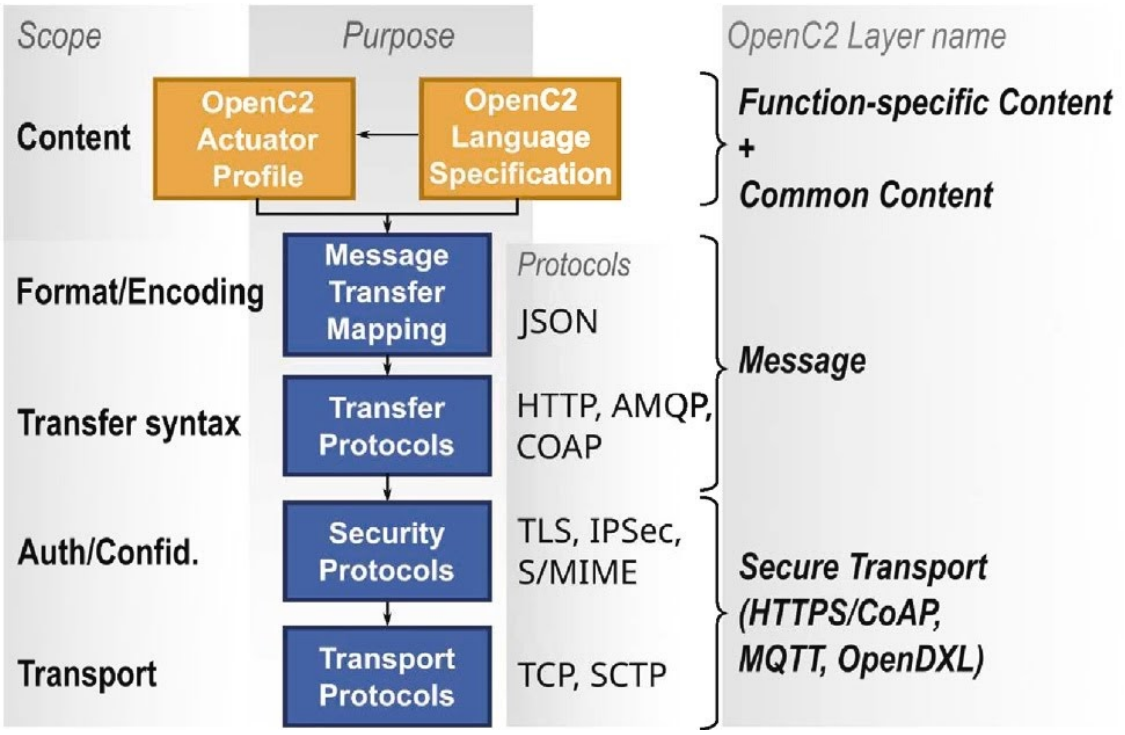

Figure 2.2. Architecture OpenC2

poor engineering that hard-codes serialization formats or lacks adequate transfer protocol implementation.

The *otupy* library is designed to provide a practical implementation of the OpenC2 standard with a focus on modularity and ease of development. Its main objective is to simplify the creation of OpenC2-enabled systems by abstracting low-level details such as message handling, serialization, and transport management.

Using *otupy*, developers can quickly implement both OpenC2 Producers (controllers) and Consumers (Actuators, such as firewalls, IoT devices, or virtualized security services) with minimal boilerplate code. The framework provides ready-to-use components that streamline the development process and ensure compliance with the OpenC2 specification.

2.2.1 Architectural Overview and Workflow

The architecture of Otupy is built upon the OpenC2 client/server model, where a **Producer** (client) sends commands to a **Consumer** (server), which may then return a response. This framework decouples the high-level business logic from the underlying communication syntax, protocol, and encoding.

Core Components and Roles

The framework defines several critical objects to facilitate this interaction:

- **OpenC2 Producer:** Commonly integrated into network security controllers to take decisions and dispatch commands.
- **OpenC2 Consumer:** Resides on an external host or is co-located with managed devices; it dispatches commands to specific actuators.
- **Actuators:** These objects provide the mapping between OpenC2 messages and the proprietary interfaces of security functions, such as `iptables` for firewalling.
- **Managed Devices:** The concrete hardware or software that implements the security functions.

The Otupy library implements all objects described in the OpenC2 Language Specification, including Commands, Responses, and all associated target and data types . The `dispatch()` function within the Consumer object is responsible for selecting one or more actuators to execute a received command .

2.2.2 Design Principles: Decoupling and Extensibility

The primary scientific contribution of Otupy is its design principle, which decouples the main building blocks—profiles, serialization formats, and transfer protocols—from the core library . This is achieved through two primary mechanisms: a modular communication stack and an inheritance model coupled with meta-serialization.

Inheritance Model and Meta-serialization

Otupy defines data types using an inheritance tree derived from OpenC2 primitive types (e.g., String, Number) and structures (e.g., Map, Record, Choice) . Meta-serialization converts these structures into an intermediary, nested key-value format that remains agnostic of any specific serialization language.

- **Serialization:** The process starts at the root object and recursively transforms it into key-value pairs until primitive types are reached .
- **Deserialization:** This "reverse process" is more challenging as it requires inferring data types from a bare representation; Otupy leverages the inheritance tree to recursively process meta-serialized data and initialize objects with decoded pairs .

This approach allows users to develop new extensions without modifying the framework's core code .

Modular Communication Stack

The framework features a protocol stack comprising **Encoder** and **Transfer** interfaces . Encoders (e.g., JSON, CBOR, YAML) translate the intermediary meta-serialized representation into common formats, while Transfers (e.g., HTTPS, MQTT) manage the delivery and message structure .

2.2.3 Implementation and API Design

Implemented in Python to ensure portability across common platforms, Otupy leverages reflective programming to handle diverse data structures without prior knowledge .

Mapping to Python Types

Otupy maps OpenC2 structures to native Python types to simplify the developer experience :

Table 2.1. Mapping of OpenC2 Structures into Python Types

OpenC2 Type	**Python Type**	**Notes**
Array	List	Standard list
Choice	Internal	Otupy implementation
Enumerated	aenum.Enum	From aenum library
Map	Dict	Standard dictionary
Record	Object	Attributes declared as private

The API is designed to be intuitive, accepting common networking data types like IP addresses and URLs while remaining fully agnostic of serialization and transfer operations .

2.2.4 Architecture of *otupy*

The *otupy* framework is organized into modular components that mirror the operational flow of OpenC2-based systems. This design enables flexibility and simplifies the integration of new functionalities.

The library also includes base classes for Messages, Commands, Responses, Encoders, and Transfer protocols. By extending these components, developers can concentrate on implementing the operational logic of their actuators, while the framework transparently manages encoding, communication, and message handling.

Data Types and Core Modules

The core modules provide the essential building blocks required to construct OpenC2 applications:

- **Base Classes:** Define the internal representation of Commands, Responses, Arguments, and Actuator profiles, enabling a consistent programming interface.
- **Encoders:** Handle the serialization of Python objects into formats such as JSON, CBOR, or YAML, allowing interoperability with external systems.

- **Transfers:** Implement communication mechanisms (e.g., HTTP, HTTPS, MQTT) used to exchange messages between Producers and Consumers.
- **Profiles:** Provide support for function-specific behavior, including predefined profiles such as SLPF, as well as custom actuator implementations.

Producer and Consumer

- **Producer:** Responsible for creating and sending commands. It manages request identifiers and automatically correlates incoming responses with previously issued commands.
- **Consumer:** Receives incoming commands, validates them against the supported profile, executes the requested operations, and generates appropriate responses by handing off the command to the appropriate actuator.

Messaging Flow

The interaction between Producer and Consumer follows a structured execution pipeline:

1. The Producer constructs a Command object by specifying action, target, and arguments.
2. The Command is serialized using the selected Encoder.
3. The serialized message is transmitted through the configured Transfer protocol.
4. The Consumer receives and decodes the message, then dispatches it to the appropriate actuator logic.
5. A Response is generated, serialized, and returned to the Producer.

2.2.5 Serialization and Deserialization

In *otupy*, serialization and deserialization are fully abstracted. Commands and Responses are represented as Python objects and are automatically converted into the desired format by the selected Encoder.

This abstraction allows developers to avoid manual handling of message formats. Instead, they only need to extend the base classes and implement the `handle_command` method within their actuators. All encoding and decoding operations are managed internally by the framework.

2.2.6 HTTP-Based Communication Example

The framework provides built-in support for HTTP and HTTPS communication through dedicated Transfer classes. Each message includes a unique `request_id`, which is used to match responses with their corresponding requests.

Consumer (Server) Example

Listing 2.1. Consumer Server Example

```
import otupy
from otupy.transfers.http import HTTPTransfer
from otupy.encoders.json_encoder import JSONEncoder
from my_actuators.slpf_actuator import SLPFActuator

actuators = {'slpf': SLPFActuator()}
consumer = otupy.Consumer(
    identifier="consumer.example.net",
    actuators=actuators,
    encoder=JSONEncoder(),
    transfer=HTTPTransfer(host="127.0.0.1", port=8080)
)
consumer.run()
```

Producer (Controller) Example

Listing 2.2. Producer Example

```
import otupy
from otupy.transfers.http import HTTPTransfer
from otupy.encoders.json_encoder import JSONEncoder

producer = otupy.Producer(
    identifier="producer.example.net",
    encoder=JSONEncoder(),
    transfer=HTTPTransfer(host="127.0.0.1", port=8080)
)
```

Sending Commands Example

Listing 2.3. Send OpenC2 Command Example

```
# Define command arguments
args = otupy.Args({'response_requested':
    ↪ otupy.ResponseType.complete})

# Create a query command to check supported features
cmd = otupy.Command(
    action=otupy.Actions.query,
    target=otupy.Features([otupy.Feature.versions,
        ↪ otupy.Feature.profiles]),
    args=args
)

# Sending command
response = producer.send_command(cmd)

# SLPF actuator example
pf_args = otupy.Args({
    'response_requested': otupy.ResponseType.complete,
    'direction': 'ingress'
```

```
})
pf_target = otupy.SLPFSpecifiers({'hostname': 'firewall'})
cmd = otupy.Command(action=otupy.Actions.allow,
    ↪ target="172.19.0.0/24",
                        args=pf_args, actuator=pf_target)

response = producer.send_command(cmd)
```

This design allows new actuators to be integrated with minimal effort, while remaining fully compatible with the OpenC2 communication model.

Chapter 3

Thesis Objective

Modern computing infrastructures are increasingly complex, characterized by a mix of on-premise virtualized environments, public cloud deployments, and deeply instrumented operating systems. The effective management, orchestration, and security of these heterogeneous ecosystems require solid mechanisms for both high-level resource discovery and low-level systemic control. A central problem is that every platform exposes its own proprietary management interface, so situational awareness and active enforcement end up split across tools that do not work together. This thesis addresses this problem by adopting OpenC2 as a single, vendor-agnostic command-and-control language and by extending the *otupy* framework with new actuator profiles that bring different platforms under a common, standardized interface.

The overall design follows a clear separation of concerns. The discovery and enforcement logic is *not* implemented inside the managed platforms: it is built as a platform-independent mechanism based on OpenC2 and its *otupy* reference implementation, and only the platform-specific part is provided as a dedicated actuator. In this sense, the contribution of this work consists of three extensions of the OpenC2 profile ecosystem: two specializations of the CTXD (Context Discovery) profile, one for Proxmox VE and one for Azure Kubernetes Service, and one new actuator profile for eBPF. Each extension translates the proprietary surface of a target platform into the same shared, profile-defined model exchanged over OpenC2, so that infrastructures as different as an on-premise hypervisor cluster, a managed cloud orchestrator and a Linux kernel can be queried and controlled through the same language.

The first contribution of this work focuses on on-premise virtualization through the design and implementation of a CTXD actuator for the Proxmox Virtual Environment (VE). The discovery mechanism itself is universal and defined by the CTXD profile; the contribution lies in integrating the Proxmox REST API as a concrete back-end, so that physical nodes, virtual machines, containers, and networking elements can be enumerated and expressed in the common CTXD data model. The resulting representation is a composite service that captures the hierarchical relationships between physical hosts and their virtualized workloads. The goal is not to duplicate the visibility that mature cloud platforms such as OpenStack already provide through their native APIs; the goal is to expose Proxmox

through the same vendor-agnostic interface used for every other actuator in the framework, so that an orchestrator can consume Proxmox topology without being aware of Proxmox itself.

The second contribution extends the same approach to the public cloud with a CTXD actuator for the Microsoft Azure platform, with specific emphasis on Azure Kubernetes Service (AKS) clusters, their worker nodes, the pods scheduled on them, the containers they host, and the associated network interfaces. As for Proxmox, the discovery semantics are inherited from the shared CTXD profile, while the contribution consists in mapping the Azure-specific access path—based on `az aks command invoke` on top of the Azure REST and `kubectl` APIs—into the same model. Since both actuators produce instances of the same data structures, the orchestrator obtains a uniform, machine-readable view of very different environments and can correlate them without ad-hoc adapters: it is the common interface, rather than any single platform, that is the novelty.

Beyond structural discovery, the thesis addresses the need for low-overhead security and observability through a new OpenC2 actuator profile for eBPF. The profile standardizes the remote management of eBPF programs by exposing their lifecycle—transferring the compiled object, loading and attaching it to a kernel hook, querying the active programs, and safely detaching them—behind a small set of OpenC2 commands, hiding the complexity of the underlying toolchain and Linux distribution. By acting as a Consumer within the OpenC2 communication model, the eBPF agent only executes commands that are checked by the native in-kernel verifier and returns precise execution feedback, so that remote management of deep-system observability does not compromise system integrity. This is the third extension of the OpenC2 profile catalogue contributed by this work, and it complements the two CTXD specializations on the enforcement side.

Taken together, these contributions deliver a single OpenC2-based framework that covers both infrastructure inventory and kernel-level enforcement through the same language. The two CTXD actuators provide situational awareness over on-premise and cloud environments through a common, profile-defined representation; the eBPF actuator profile gives those environments standardized, high-performance enforcement; and the underlying *otupy* machinery makes sure that further platforms or hooks can be added simply by contributing new actuators, without changing the framework itself. The result is a uniform, transparent and extensible base for security orchestration across today's heterogeneous environments.

Chapter 4

CTXD Model

Building on extensive research into context-aware data models, this work refines and expands existing architectural patterns. The solution specifically leverages the CTXD (Context Discover Actuator) profile data model established by Tanzarella et al. in 2024, adapting its core principles to meet the requirements of this study

4.0.1 Overview

The CTXD data model identifies individual components, referred to as Services, and the relationships between them, represented as Links. Each Service corresponds to a distinct resource or application, while Links define how these components are connected and interact within the system. This structure enables a clear and flexible representation of the overall architecture

4.0.2 Core Concepts

- **Service**
  A Service represents a deployable or logical unit within the system. Each Service is uniquely identified by a **Sid (Service Identifier)**, which encapsulates attributes such as name, namespace, domain, and version. This identifier ensures consistent referencing across the entire model.

- **Link**
  A Link defines a relationship or dependency between services. It specifies the owning Service (via its Sid), the type and description of the relationship, and a set of connected entities (Peers).

- **Peer**
  A Peer represents a participant in a Link. Each Peer is identified by a Sid and has a defined role (e.g., provider or consumer), optionally including additional contextual information.

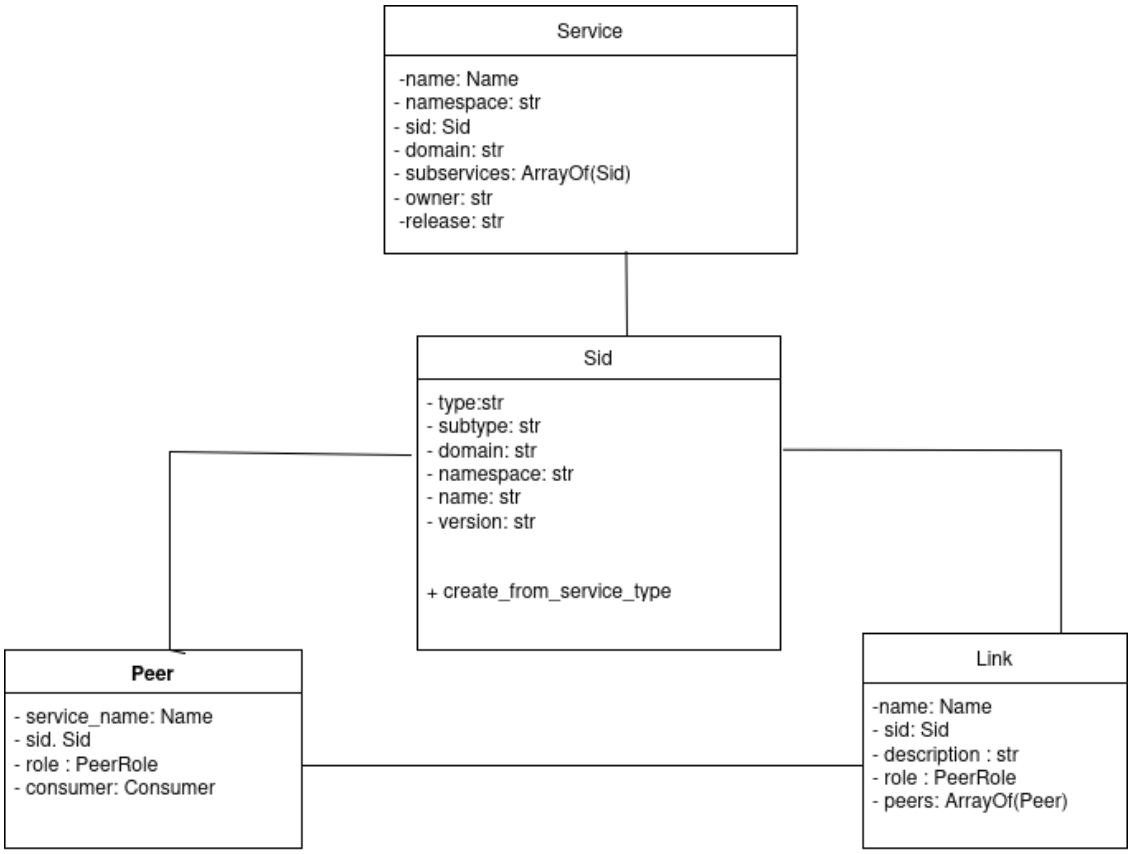

Figure 4.1. CTXD Data Model

4.0.3 Service Relationship Model

Figure 4.1 focuses on the logical relationships between services and is centered around the **CTXDActuator**, which manages collections of Services and Links.

The **Service** entity contains identifying metadata and references a **Sid**, which acts as the primary identification mechanism. A Sid encapsulates attributes such as type, subtype, domain, namespace, and version, and can be generated from a **ServiceType**.

A **Link** represents a relationship owned by a Service, defining its type and description while connecting multiple **Peer** entities. Each Peer identifies another Service (via Sid) and specifies its role within the relationship, such as consumer or provider.

The **ServiceType** component provides a classification mechanism that ensures consistent identity generation and categorization across the system.

4.0.4 Service Type

Figure 4.2 presents a comprehensive model of the execution environment and underlying infrastructure used to represent services in a cloud-based system. It defines

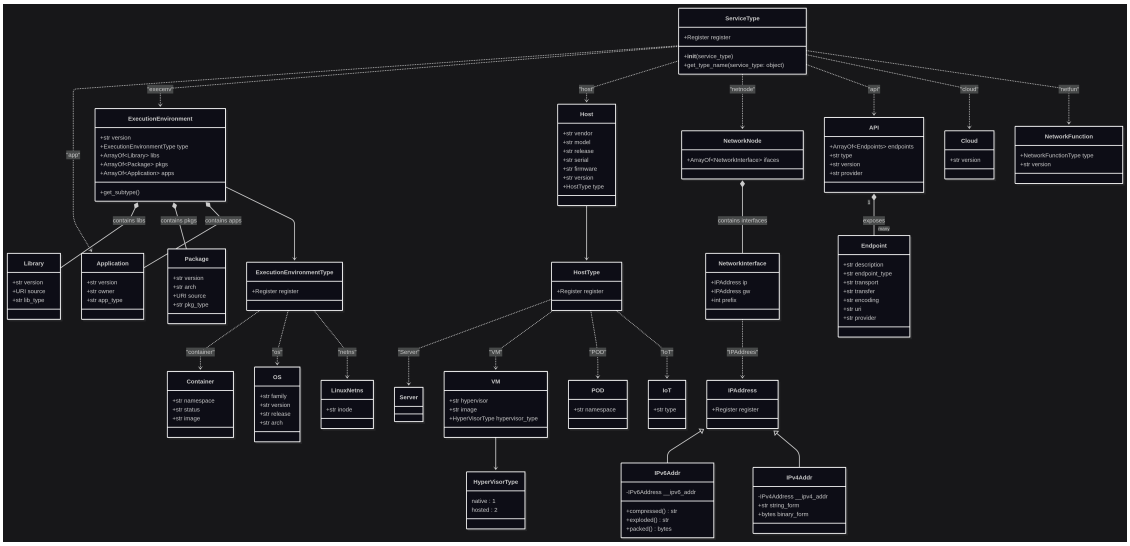

Figure 4.2. Service Type Model

how software components, runtime environments, and physical or virtual resources are structured and interconnected.

At the core of the model is the **ExecutionEnvironment**, which represents the runtime context in which applications are executed. This environment aggregates different software artifacts, including **Applications**, **Packages**, and **Libraries**. Each of these elements carries metadata such as version and source, enabling detailed tracking of deployed software components. The type of execution environment is abstracted through the **ExecutionEnvironmentType**, allowing classification into categories such as containers, operating systems, or network namespaces.

The model further refines execution environments into specific implementations:

- **Container** represents a lightweight, isolated environment that shares the host OS kernel, typically used for microservices and application deployment.
- **OS** represents a full operating system environment, apturing system-level details such as family, version, and architecture.
- **LinuxNetns**, representing Linux network namespaces.

On the infrastructure side, the **Host** entity models the physical or virtual machine that runs the execution environments. It includes attributes such as vendor, model, firmware, and version, and is categorized by **HostType**. Host types include servers, virtual machines, pods, and IoT devices, reflecting the diversity of deployment platforms.

Networking is modeled through the **NetworkNode**, which contains multiple NetworkInterface instances. Each interface is associated with IP configuration details, including IP address, gateway, and prefix. The IPAddress abstraction supports both IPv4Addr and IPv6Addr, providing flexibility for different networking standards.

At a higher level, the model incorporates service exposure and cloud integration components:

- **API**, which groups multiple Endpoint objects. Each endpoint defines communication details such as transport protocol, encoding, URI, and provider.
- **Cloud**, representing the cloud environment in which services are deployed.
- **NetworkFunction**, capturing specialized network roles or functions within the system like router, vpn, nat, firewall or bridge.

Chapter 5

The CTXD Actuator for Proxmox VE

5.1 What is Proxmox VE and what are its main strengths?

Proxmox VE is a platform to run virtual machines and containers. It is based on Debian Linux, and completely open source. It implements two virtualization technologies [3]:

- Kernel-based Virtual Machine (KVM)
- Container-based virtualization (LXC).

Proxmox can be used as a single node, or a cluster of many nodes. All management tasks can be done using a web-based management interface.

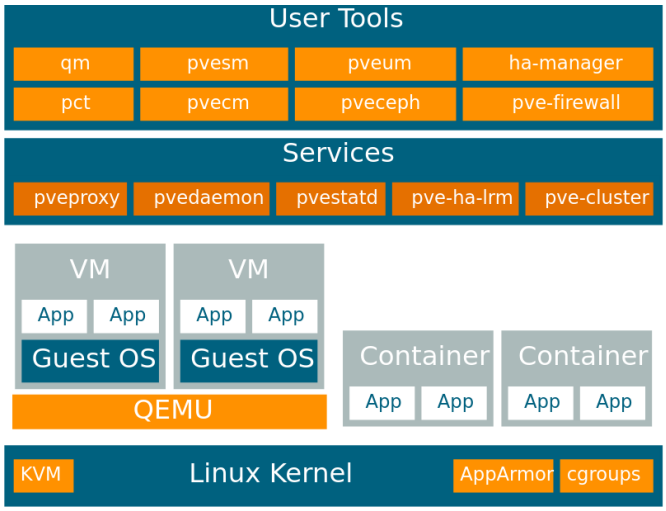

Figure 5.1. Proxmox VE internal architecture

Flexible Networking

Proxmox VE uses a bridged networking model. All VMs can share one bridge as if virtual network cables from each guest were all plugged into the same switch. For connecting VMs to the outside world, bridges are attached to physical network cards and assigned a TCP/IP configuration [3].

Integrated Firewall

The integrated firewall allows you to filter network packets on any VM or Container interface. Common sets of firewall rules can be grouped into security groups [3].

Hyper-converged Infrastructure

A hyper-converged infrastructure (HCI) is especially useful for deployments in which a high infrastructure demand meets a low administration budget, for distributed setups such as remote and branch office environments or for virtual private and public clouds [3].

5.2 Objective

The objective is to design and implement a discovery mechanism for the *Proxmox VE* platform. The goal is to automatically identify and model the different components of a Proxmox infrastructure, including nodes, virtual machines (VMs), containers, and networking elements.

This work aims to represent Proxmox VE as a composite service, where each node acts as a hosting environment for multiple virtual resources. The discovery process focuses on establishing relationships between virtual machines and their underlying physical nodes, as well as identifying network configurations such as bridges.

The developed solution leverages the Proxmox REST API to collect real-time information about the system and build a structured representation of the infrastructure. This representation can then be used for monitoring, analysis, or integration with higher-level management systems.

Ultimately, this work seeks to provide a clear and automated view of a Proxmox environment, enabling better understanding, management, and orchestration of virtualized resources.

5.3 Proxmox Discovery

The discovery process for Proxmox VE is implemented within the `CTXDActuatorProxmox` class. This actuator implements the CTXD profile for a Proxmox cluster, translating physical and virtual resources into a structured graph of *Services* and *Links*.

5.3.1 Connection and Authentication

The discovery starts with the establishment of a session via the `ProxmoxAPI` interface provided by the `proxmoxer` library. The actuator requires a set of credentials — host, username, password, and SSL verification settings — to interact with the Proxmox REST API. Upon a successful `_connect()` call, the actuator is ready to query the cluster's state.

5.4 Implementation and Data Mapping

The integration of Proxmox VE into the CTXD framework requires a semantic mapping of virtualization entities (nodes, VMs, containers, bridges) into the CTXD data model. The implementation is encapsulated in the `CTXDActuatorProxmox` class, which uses the `proxmoxer` library to communicate with the Proxmox REST API.

Table 5.1 summarises the mapping from Proxmox-specific entities to otupy classes.

Proxmox element	**otupy class**	**Mapping context**
Cluster node (physical)	`ExecutionEnvironment`	Type: OS
QEMU virtual machine	`Host`	Type: `HostType(VM)`
LXC container	`Host`	Type: `HostType(VM)`, hypervisor: LXC
Network bridge	`Network`	Type: `NetworkType(EthernetNetwork)`
VM network interfaces	`NetworkNode`	Contains `NetworkInterface` objects
Hosting relationship	`Link`	Type: `LinkType.hosting`
Proxmox cluster (root)	`Cloud`	Aggregates all node sub-services

Table 5.1. Proxmox to CTXD data model mapping

5.4.1 The Discovery Pipeline

The operational logic of the Proxmox actuator is encapsulated in a multi-stage pipeline triggered by the `discover_context()` method:

1. **Node identification:** The pipeline begins by calling `get_cluster_nodes()` to identify all active physical hosts and build a `vmid` → `node` map used by all subsequent link-discovery steps.

2. **Service discovery:** For each identified node the actuator performs a deep scan to categorise VMs (QEMU), containers (LXC), and networking bridges, mapping them to the CTXD service hierarchy.

3. **Link discovery:** Finally the actuator establishes topological relationships: nodes to the cloud root, VMs to their hosting nodes, and VMs and bridges to each other.

5.4.2 Modeling Virtual Resources

Physical nodes. Each Proxmox node is modelled as an `ExecutionEnvironment` with an `OS` sub-type. This recognises the node not merely as hardware but as a platform capable of hosting nested virtual environments. Each node service is cached in `self._node_services` at discovery time, so link-discovery methods can reference the pre-built `Service` objects directly without reconstructing `SId` values.

Listing 5.1. Physical node discovery and CTXD mapping

```
def _discover_proxmox_nodes(self):
    node_sids = ArrayOf(SId)()

    for h in self.nodes:
        node_ee = ExecutionEnvironment(
            name=Hostname(h['node']),
            id=h['node'],
            description="Proxmox physical node",
            type=ExecutionEnvironmentType(OS()),
        )
        node_sid = SId.create_from_service_type(node_ee)
        node_sids.append(node_sid)

        node_svc = Service(
            name=Name(str(h['node'])),
            sid=node_sid,
            type=ServiceType(node_ee),
            subservices=ArrayOf(SId)(),
            owner=self.owner,
            release=None,
        )
        self.services.append(node_svc)
        # cache for link-discovery methods
        self._node_services[h['node']] = node_svc

    proxmox_cloud = Cloud(
        description='Proxmox VE cluster',
        id=None,
        name=self.cloud_name,
        type='proxmox',
    )
    cloud_svc = Service(
        name=Name(self.cloud_name),
        sid=SId.create_from_service_type(proxmox_cloud),
        type=ServiceType(proxmox_cloud),
        subservices=node_sids,   # nodes listed as sub-services
        owner=self.owner,
        release=None,
    )
    self.services.append(cloud_svc)
    self._cloud_service = cloud_svc
```

Full virtualisation (QEMU). Each VM produces two `Service` objects: a `Host/VM` service for the machine itself, and a `NetworkNode` sub-service that carries all network interfaces. When the QEMU Guest Agent is running, layer-3 information (IP addresses and prefixes) is retrieved and stored inside the `NetworkNode`.

This hierarchical nesting demonstrates the CTXD model's capacity for representing composite resources.

Listing 5.2. QEMU VM discovery and CTXD mapping

```
def _discover_proxmox_vms(self):
    for node in self.nodes:
        for vm in self.get_all_vms(node):
            config = self.proxmox_conn.nodes(
                node['node']).qemu(vm['vmid']).config.get()
            ifaces = ArrayOf(NetworkInterface)()

            try:
                vm_details = self.proxmox_conn.nodes(
                    node['node']).qemu(vm['vmid']).agent(
                    "network-get-interfaces").get()
                for iface in vm_details.get("result", []):
                    ips = ArrayOf(IPInfo)()
                    for ip_info in iface.get("ip-addresses",
                        ↪ []):
                        ips.append(IPInfo(
                            ip=ip_info["ip-address"],
                            prefix=ip_info["prefix"],
                            gw=None,
                        ))
                    ifaces.append(NetworkInterface(
                        description=vm.get('name',
                            ↪ str(vm['vmid'])),
                        id=f"{vm['vmid']}.{iface['name']}",
                        iface=iface["name"], ips=ips,
                    ))
            except Exception as e:
                logger.warning("Guest agent unavailable for VM
                    ↪ %s: %s",
                               vm['vmid'], e)

            netnode = NetworkNode(
                name=vm.get('name', str(vm['vmid'])),
                description=f"Network interfaces for QEMU VM
                    ↪ {vm['vmid']}",
                ifaces=ifaces,
            )
            server = Host(
                name=vm.get('name', str(vm['vmid'])),
                id=vm['vmid'],
                description=vm.get('name', ''),
                type=HostType(VM(hypervisor='QEMU',
                                 hypervisor_type="native",
                                 image=config.get('ostype'))),
            )
            name        = Name(server.name)
            netnode_sid = SId.create_from_service_type(netnode)

            vm_service = Service(
                name=name,
                sid=SId.create_from_service_type(server),
                type=ServiceType(server),
                subservices=ArrayOf(SId)([netnode_sid]),
```

```
50                  owner=self.owner, release=None,
51              )
52              self.services.append(vm_service)
53              self.services.append(Service(
54                  name=Name(server.name + ".interfaces"),
55                  sid=netnode_sid,
56                  type=ServiceType(netnode),
57                  subservices=ArrayOf(SId)(),
58                  owner=str(name), release=None,
59              ))
```

Containerisation (LXC). LXC containers are treated as lightweight hosts and share the `Host` base class with QEMU VMs. They are differentiated by their `hypervisor` attribute (set to `LXC`), allowing the system to apply container-specific security or management policies. Each LXC container also receives a `NetworkNode` sub-service, consistent with the VM model. Network interfaces are read from the container's static configuration rather than from a guest agent.

Listing 5.3. LXC container discovery and CTXD mapping

```
1  def _discover_proxmox_lxc(self):
2      for node in self.nodes:
3          for ct in self.get_all_containers(node):
4              vmid   = ct['vmid']
5              config = self.proxmox_conn.nodes(
6                  node['node']).lxc(vmid).config.get()
7              ifaces = ArrayOf(NetworkInterface)()
8
9              for key, value in config.items():
10                 if key.startswith('net') and isinstance(value,
                       ↪ str):
11                     ifaces.append(NetworkInterface(
12                         description=f"LXC {vmid} {key}",
13                         id=f"{vmid}.{key}",
14                         iface=key,
15                         ips=ArrayOf(IPInfo)(),
16                     ))
17
18             server = Host(
19                 name=ct.get('name', str(vmid)),
20                 id=vmid,
21                 description=f"LXC Container: "
22                             f"{config.get('hostname',
                                   ↪ ct.get('name'))}",
23                 type=HostType(VM(hypervisor='LXC',
24                                  hypervisor_type="native",
25                                  image=config.get('ostype',
                                        ↪ 'linux'))),
26             )
27             netnode_sid = SId.create_from_service_type(
28                 NetworkNode(name=server.name, ifaces=ifaces))
29
30             ct_service = Service(
31                 name=Name(server.name),
32                 sid=SId.create_from_service_type(server),
33                 type=ServiceType(server),
```

```
34                  subservices=ArrayOf(SId)([netnode_sid]),
35                  owner=self.owner, release=None,
36              )
37              self.services.append(ct_service)
```

5.4.3 Topological Mapping and Hosting Links

The link-discovery phase creates explicit `Link` objects between the discovered services. Three types of hosting link are established.

Node to cloud. Each physical node is linked to the Proxmox cloud root via a `hosting` link.

VM to node. The `_discover_vms_link_nodes()` method performs a logical join between the discovered `Host/VM` services and the cluster resource map (built once in `discover_context()`). The resulting link uses `LinkType.hosting` with `PeerRole.host` on the physical node side. This is essential for impact analysis: if a node fails, the CTXD framework can instantly traverse these links to identify all affected guests.

Bridge to node. Each Linux bridge is linked to the physical node that owns it.

5.5 Perseo Datacenter

To validate the CTXD data model and the developed actuator, a real-world infrastructure — the **Perseo Datacenter** — was analysed. This section describes the physical and virtual resources visible in the Proxmox management interface.

5.5.1 Infrastructure Overview

As illustrated in Figure 5.2, the Perseo Datacenter runs Proxmox VE 7.4-17 on a single physical node (*perseo*) hosting a variety of critical resources:

- **Virtual Machines (QEMU):** The environment includes diverse instances such as mail servers (*mail1.ge.cnr.it*), web hosts (*vhost150*), and general-purpose machines running various Linux distributions including Ubuntu and Rocky Linux.
- **Containers (LXC):** No active LXC containers are present in the current setup, indicating a focus on full virtualisation for workload deployment.
- **Kubernetes orchestration:** The datacenter hosts a dedicated Kubernetes cluster consisting of four nodes (*kube0* through *kube3*), mapped as individual virtual machines.
- **Storage layer:** The system manages multiple storage pools including *MACCHINE_VIRTUALI*, *datastorebackup*, and *local*. Storage is not modelled in the current CTXD profile — see Section 5.6.6 for a discussion.

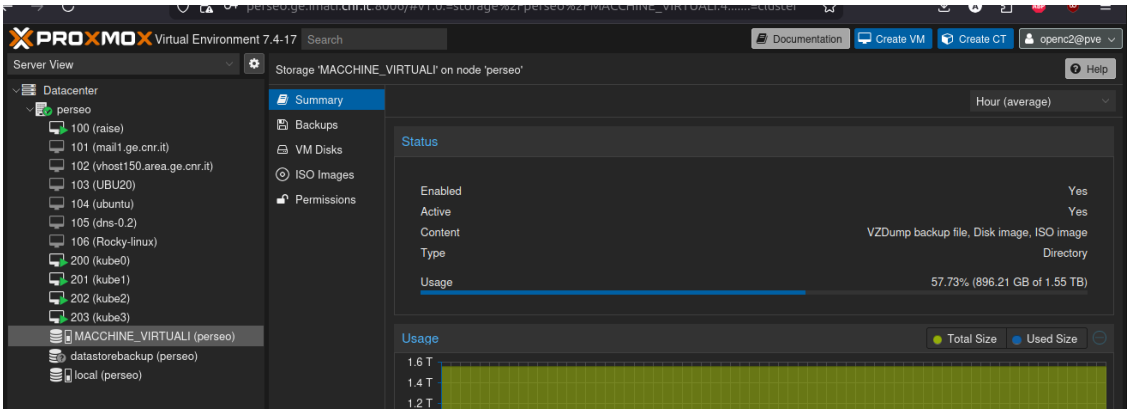

Figure 5.2. Perseo Datacenter: Proxmox VE management interface

5.6 Results

5.6.1 Discovered Nodes

The platform contains a single physical node identified as **perseo**. It serves as the hypervisor hosting all virtual machines. Its characteristics are summarised in Table 5.2.

Name	Description	Type
perseo	Proxmox physical node	OS execution environment

Table 5.2. Discovered physical nodes

5.6.2 Discovered Virtual Machines

The discovery process identified eleven VMs hosted on *perseo*. Each VM is represented as a `Host/VM` service with a unique `SId`, a human readable name, and hypervisor metadata. Table 5.3 lists the discovered virtual machines.

5.6.3 Network Interfaces and Bridges

For each VM a corresponding `NetworkNode` sub-service was discovered. These nodes represent the set of virtual interfaces of each VM, enabling connectivity modelling within the cluster or toward external networks. IP addresses are populated when the QEMU Guest Agent is running; otherwise the interface structure is recorded without layer-3 details.

Four Linux bridges were discovered on the *perseo* node: `vmbr0`, `vmbr1`, `vmbr2`, and `vmbr3`. Each is modelled as a `Network/EthernetNetwork` service and linked both to its hosting node and to the VMs attached to it via `packet_flow` links.

VM name	ID	Hypervisor	Description
raise	100	QEMU	General purpose host
mail1.ge.cnr.it	101	QEMU	Mail server
vhost150.area.ge.cnr.it	102	QEMU	Web host
UBU20	103	QEMU	Ubuntu 20 workload
ubuntu	104	QEMU	Ubuntu workload
dns-0.2	105	QEMU	DNS service
Rocky-linux	106	QEMU	Rocky Linux workload
kube0	200	QEMU	Kubernetes node 0
kube1	201	QEMU	Kubernetes node 1
kube2	202	QEMU	Kubernetes node 2
kube3	203	QEMU	Kubernetes node 3

Table 5.3. Discovered virtual machines hosted on perseo

5.6.4 Discovered Links and Infrastructure Graph

Table 5.4 shows the hosting relationships established between VMs and the physical node.

VM	Host node	Link type
raise	perseo	hosting
mail1.ge.cnr.it	perseo	hosting
vhost150.area.ge.cnr.it	perseo	hosting
UBU20	perseo	hosting
ubuntu	perseo	hosting
dns-0.2	perseo	hosting
Rocky-linux	perseo	hosting
kube0	perseo	hosting
kube1	perseo	hosting
kube2	perseo	hosting
kube3	perseo	hosting

Table 5.4. Hosting links between VMs and the physical node

Figure 5.3 shows the complete infrastructure graph produced by the actuator. Each node type is represented by a distinct border colour: red for the Proxmox cloud root, black for the physical node, blue for VMs, grey for network interface nodes, and black for Linux bridges. Edge styles encode the relationship type: solid lines for hosting links, dashed lines for `packet_flow` links between VMs and bridges.

5.6.5 Latency Measurements

The Proxmox actuator uses a single instrumentation layer, implemented in the module `otupy.actuators.ctxd.metrics`. The module exposes a `MetricsCollector`

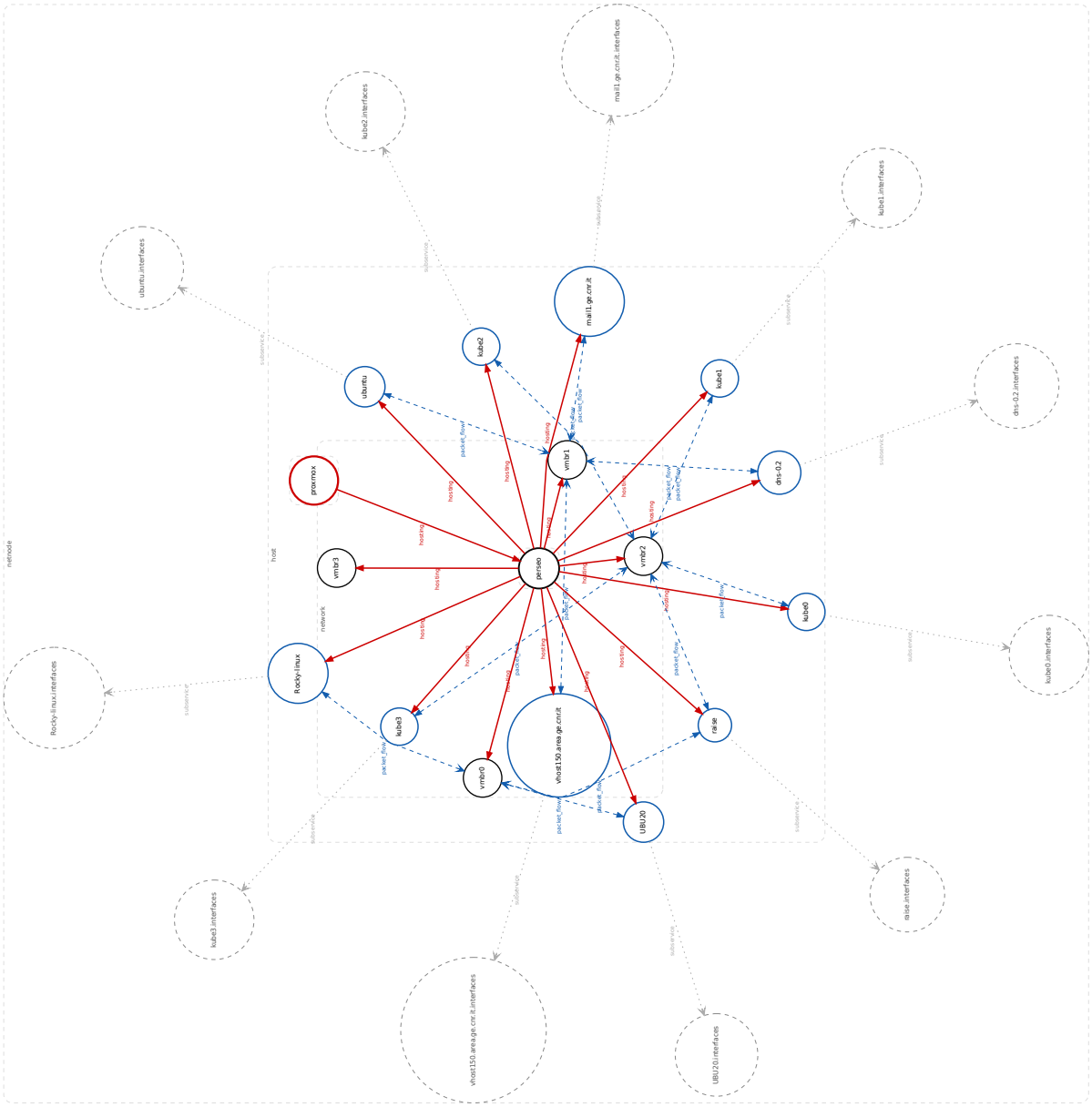

Figure 5.3. CTXD infrastructure graph for the Perseo Datacenter.

class that wraps every remote API call, records its wall-clock duration, success status and payload size, and emits the result on a dedicated logger in a unified line format (`API_CALL ...` for individual calls and `API_SUMMARY ...` for the aggregate emitted at the end of each discovery cycle). The use of the same instrumentation in the two actuators makes their performance figures directly comparable.

In the Proxmox actuator the collector is wired in via the `@measure_latency` decorator, applied to every helper method that issues a request through `proxmoxer`. When the `metrics` block is disabled in the YAML configuration, the decorator

becomes a pass-through and no overhead is incurred.

To collect statistically meaningful measurements, `discover_context()` was invoked 20 times in sequence against the Perseo Datacenter over a local network connection. The resulting dataset contains 80 `API_CALL` records (20 per helper, across the four helpers actually exercised by the discovery pipeline). The remaining decorated helpers (`get_node_networks`, `get_node_storage`, `get_interfaces`, `get_interfaces_vm`) are part of the public surface of the actuator but are not called by `discover_context()` and therefore do not appear in the table. Per-call statistics are reported in Table 5.5. Mean and standard deviation are computed with the sample formula; all times are in milliseconds.

Call	N	Mean [ms]	Std [ms]	Min [ms]	Max [ms]
`get_cluster_nodes`	20	40.4	27.2	21.5	146.6
`get_all_vms`	20	48.1	30.1	25.6	168.0
`get_all_containers`	20	119.6	50.3	28.8	201.5
`get_node_bridges`	20	35.8	15.3	23.4	89.0
Aggregate	**80**	**61.0**	**47.3**	**21.5**	**201.5**

Table 5.5. Per-call latency of the Proxmox REST helpers, aggregated over 20 discovery cycles against the Perseo Datacenter. All times are in milliseconds.

Aggregate behavior. Across all 80 recorded calls the mean per-call latency is $\bar{t} = 61.0\,\text{ms}$ with a standard deviation $\sigma = 47.3\,\text{ms}$; the minimum observed call took 21.5 ms (`get_node_bridges`) and the maximum 201.5 ms (`get_all_containers`). The sum of the four mean latencies is $\sim$244 ms, which bounds the network-bound portion of `discover_context()` well below half a second and makes the overall discovery suitable for use in near-real-time monitoring loops. The total observed wall-clock time of `discover_context()` across the 20 runs averaged approximately two seconds; the difference between this figure and the sum of the four timed calls is accounted for by the additional, untimed REST calls used to fetch the per-VM configuration (`qemu(vmid).config.get()`) and the QEMU Guest Agent interface listings inside the service-discovery loops.

High variance is present in every call. Standard deviations range from 15.3 ms (`get_node_bridges`) to 50.3 ms (`get_all_containers`), with maxima reaching 3–7$\times$ the mean. This variability is characteristic of REST API calls over a shared network to a hypervisor that may be under load: the Proxmox API server handles management requests with lower priority than guest I/O, so response times fluctuate depending on cluster activity. The medians (not shown) sit consistently below the means, confirming that the distribution is right-skewed with occasional outliers rather than uniformly slow.

`get_all_containers` is the most expensive call. With a mean of 119.6 ms, `get_all_containers` is roughly 2–3$\times$ slower than the other three helpers despite returning no items on the validation node (the Perseo Datacenter runs no LXC

containers at the time of the experiment). The cost reflects the fact that the Proxmox API still enumerates the LXC subsystem before producing the empty list. `get_cluster_nodes` (mean 40.4 ms), although it traverses the cluster-wide resource manager, is comparatively fast because the Perseo Datacenter is a single-node deployment: aggregation is therefore trivial. On a multi-node Proxmox cluster the cost of `get_cluster_nodes` would be expected to grow with the number of nodes, whereas the remaining calls would be largely unaffected.

Comparison with the AKS actuator. The figures in Table 5.5 should be read together with those reported for the AKS actuator in Section 6.6.1: both tables are produced by the same instrumentation and the same code path, so they are directly comparable. The two actuators differ in latency by more than two orders of magnitude: the average Proxmox call takes 61 ms, while the average AKS call takes 10.10 s, a ratio of approximately 165. Even the slowest Proxmox call ever observed in the dataset (201.5 ms) is around 25× faster than the *average* AKS call. This gap is not an accident of network conditions: the Proxmox actuator talks directly to the hypervisor's REST API over a local network, whereas the AKS actuator dispatches every call through `az aks command invoke`, which spawns an ephemeral job pod in the cluster for each invocation (Section 6.4.1). The overhead of the AKS access path is therefore intrinsic to the mechanism chosen to support private clusters, and not a property of the discovery logic itself.

5.6.6 Summary

The discovery mechanism successfully:

- Identifies the Proxmox cloud root and registers it as the central `Cloud` service.
- Identifies physical nodes and classifies them as `ExecutionEnvironment/OS` services linked to the cloud root.
- Discovers virtual machines (QEMU) and containers (LXC) with their properties and hypervisor type.
- Creates `NetworkNode` sub-services for VM and container interface sets.
- Discovers Linux bridges and models them as `Network` services.
- Establishes `hosting` links between VMs and the physical node, between bridges and the physical node, and between the physical node and the cloud root.
- Establishes `packet_flow` links between VMs and the bridges they are attached to.

Known limitation — storage. The Proxmox API exposes storage pools (*MACCHINE_VIRTUALI*, *datastorebackup*, *local*) as first-class resources. However, no `Storage` type is currently defined in the otupy CTXD profile. Modelling storage

as a wrong type (e.g. `ExecutionEnvironment`) would be semantically incorrect, so these resources are intentionally excluded. Adding a `Storage` type to the profile is left as future work.

Chapter 6

CTXD Actuator for Azure Kubernetes Service

6.1 Introduction to AZURE

Azure is a cloud computing platform created by Microsoft, providing a global infrastructure for computing, storage, networking, and managed application services. Among the wide catalog of services offered, this work focuses on **Azure Kubernetes Service (AKS)**, a managed Kubernetes offering that abstracts the operational complexity of running a container orchestrator at scale. AKS is particularly relevant for the MIRANDA ecosystem because it represents one of the most common deployment targets for cloud-native workloads and exhibits a layered architecture that benefits from automated context discovery.

6.2 Objective

The objective of this chapter is to design and implement an extension of the CTXD actuator for AKS, in order to automatically identify and model its core components — the cluster itself, the worker nodes, the running pods, their containers, and the associated network interfaces — and to represent them as a composite service compliant with the CTXD model introduced in Chapter 4.

The developed actuator leverages the Azure CLI together with the `kubectl` mediated channel exposed by AKS to collect real-time information about the cluster state. The collected information is translated into structured `Service` and `Link` objects, providing a machine-readable view of the infrastructure that can be consumed by the MIRANDA orchestration layer for monitoring, security policy enforcement, and correlation with other context sources (such as the Proxmox actuator described in Chapter 5).

6.3 AKS Architecture

6.3.1 Control Plane and Data Plane

A Kubernetes cluster is logically organized into two planes: the **control plane** and the **data plane**. The control plane represents the management layer of the cluster. It includes the API Server, which exposes the Kubernetes API and serves all client requests; the etcd database, which stores the cluster state; and the scheduler and controller manager, which ensure that workloads are placed on nodes and that the actual state of the system converges to the declared one.

The key architectural property of AKS, with respect to a self-managed Kubernetes cluster, is that *the control plane is fully managed by Microsoft Azure*. The user neither operates the control-plane virtual machines nor has direct shell access to them: interaction is allowed only through the Kubernetes API. This design choice has important consequences for the discovery process, because it precludes the classical approach of running local agents on master nodes and forces the actuator to operate through authenticated API calls.

The data plane consists of one or more node pools, implemented as Azure Virtual Machine Scale Sets. Each node runs the `kubelet` agent and a container runtime, and is responsible for executing the pods scheduled on it.

6.3.2 Accessing the Cluster

An administrator typically interacts with AKS using the Azure CLI to obtain credentials and then `kubectl` for cluster-level operations:

```
1 az login
2 az aks get-credentials --resource-group <rg> --name <cluster>
3 kubectl get nodes
```

The first command authenticates the user against Microsoft Entra ID; the second retrieves the kubeconfig file and stores it locally; the third issues the actual Kubernetes API request.

This workflow, however, assumes that the workstation has network reachability to the cluster's API server. In production environments, AKS clusters are frequently deployed as *private clusters*, with the API server exposed only on a private endpoint that is not reachable from the public Internet. This is the typical situation in the deployment scenarios considered in this thesis, and motivates the use of the alternative mechanism described in the next section.

6.4 AKS Discovery

The discovery process targets three logical layers of an AKS cluster: the cluster itself, the worker nodes, and the workloads (pods and their containers). Information about each layer is collected by issuing `kubectl` queries on the cluster and parsing their JSON output. The following subsections describe how these queries are dispatched and how their output is validated and consumed.

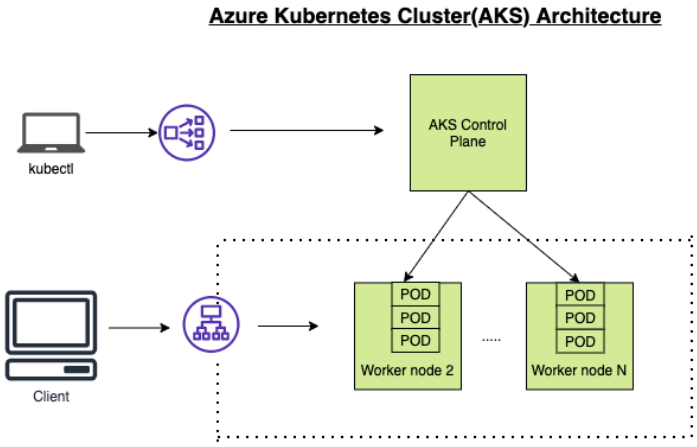

Figure 6.1. AKS architecture: a managed control plane (operated by Azure) and a user-managed data plane composed of node pools.

6.4.1 The `az aks command invoke` Mechanism

The actuator does not rely on a direct connection to the Kubernetes API server. Instead, it submits each `kubectl` command through the `az aks command invoke` primitive of the Azure CLI:

```
az aks command invoke \
    --resource-group <rg> --name <cluster> \
    --command "kubectl get nodes -o json"
```

This design choice is motivated by three considerations:

- **Reachability of private clusters.** The Azure CLI submits the command to the AKS regional control endpoint, which in turn relays it to the cluster through a managed channel. As a consequence, the actuator can operate against private clusters without requiring VPN connectivity, VNet peering, or jump hosts.
- **Unified authentication.** Authentication is performed against Microsoft Entra ID, and authorization is enforced through Azure role-based access control (RBAC). No long-lived Kubernetes service account token has to be issued, stored, or rotated by the actuator.
- **Auditability.** Every invocation is logged by Azure as a control-plane operation, which is desirable in the context of a security orchestration framework such as MIRANDA.

The response returned by `az aks command invoke` is not the raw `kubectl` output: it is wrapped in a textual envelope that contains metadata such as the exit code of the remote command. A typical reply has the following shape:

```
command started at 2025-10-12 14:21:03+00:00, finished at ...
    ↪ exitcode=0
{
  "items": [ ... ],
  ...
}
```

Before parsing, the actuator must therefore strip this envelope. In the current implementation this is done by locating the first { character of the payload and parsing from there:

```
wrapper = json.loads(raw_output[raw_output.find('{'):])
```

For commands that produce non-JSON output (such as `kubectl get ns -o jsonpath=...`), the actuator instead looks for the `exitcode=0\n` marker and consumes everything after it as the actual payload. Output that does not contain a successful exit code or that fails JSON validation is discarded and logged as an error, so that a malformed reply does not propagate to the CTXD layer.

6.4.2 Security Considerations

Because every command is dispatched through the Azure control plane, the security boundary of the actuator coincides with the Azure RBAC boundary of the service principal it uses. In the implementation described in this thesis, the service principal is bound to a custom role granting only the permissions strictly necessary for read operations: `Microsoft.ContainerService/managedClusters/runcommand/action` for issuing the commands, and the corresponding read action for retrieving their results. No write permission is granted on cluster resources, which means that even in the event of a compromise of the actuator host, the attacker would not be able to mutate the cluster state through the credentials in use.

6.4.3 Commands Used by the Actuator

The discovery process is built on a small set of `kubectl` commands, summarized in Table 6.1. The choice of these commands follows the principle of least privilege: only read-only queries are issued, and the output is restricted to the JSON fields that are actually consumed by the mapping logic.

Discovery step	Command
List namespaces	`kubectl get ns -o jsonpath='{.items[*].metadata.name}'`
Inventory nodes	`kubectl get nodes -o json`
Inventory pods per ns	`kubectl get pods -n <ns> -o json`

Table 6.1. `kubectl` commands issued by the AKS actuator through `az aks command invoke`.

It is worth noting that an earlier version of the actuator used `kubectl get pods -A -o json` to retrieve all pods in a single call. This approach was abandoned in favor of a per-namespace iteration, because on large production clusters the

cumulative output exceeded the response size limit imposed by `az aks command invoke`, leading to truncated and unparseable replies. Iterating over namespaces produces smaller payloads at the cost of additional invocations, but yields stable behavior regardless of cluster size.

6.5 Mapping AKS Resources to the CTXD Model

The actuator translates the Kubernetes entities returned by the queries above into instances of the CTXD data model. Table 6.2 summarizes the mapping; the rest of this section describes each rule in detail.

AKS element	CTXD class	Mapping context
AKS cluster	`Cloud`	type = AKS
Kubernetes node	`ExecutionEnvironment`	ExecutionEnvironmentType(OS)
`kubelet`	`Application`	app_type = kubelet
Pod	`Host`	HostType(Pod)
Container	`ExecutionEnvironment`	ExecutionEnvironmentType(Container)
CNI status	`NetworkNode + Port`	per-pod
Node → kubelet	`Link`	LinkType.hosting
kubelet → pod	`Link`	LinkType.controlling

Table 6.2. Mapping between AKS resources and CTXD classes.

6.5.1 From AKS Resources to CTXD Services

AKS cluster as a Cloud service. The cluster is modeled as a single `Cloud` object that acts as the root of the discovered topology. AKS is classified as a *managed container service* (a Platform-as-a-Service offering): the consumer of the cluster receives a Kubernetes API as a service, while the underlying machines, control-plane components, and operational tasks are handled by the provider.

```
aks = Cloud(description='Azure cloud', id=None, name="AKS",
    ↪ type='AKS')

self.services.append(Service(
    name=Name(aks.name),
    sid=SId.create_from_service_type(aks),
    type=ServiceType(aks),
    subservices=aks_subservices
))
```

Kubernetes nodes as execution environments. Each entry returned by `kubectl get nodes -o json` is mapped to an `ExecutionEnvironment` representing a Kubernetes node, with type OS. System-level metadata exposed by Kubernetes (`nodeInfo.operatingSystem`, `architecture`, `kernelVersion`, `containerRuntimeVersion`) is preserved and carried into the model, so that downstream consumers can reason about the underlying platform of each node.

```
node = ExecutionEnvironment(
    name=n.metadata.name,
    id=n.metadata.uid,
    version=n.status.nodeInfo.kernelVersion,
    description="AKS node " +
        ↪ n.status.nodeInfo.containerRuntimeVersion,
    type=ExecutionEnvironmentType(
        OS(family=n.status.nodeInfo.operatingSystem,
           arch=n.status.nodeInfo.architecture,
           version=n.status.nodeInfo.kernelVersion)
    )
)
```

Kubelet as an application. On each node, the actuator instantiates one `Application` of type `kubelet`. This is the in-cluster process responsible for managing the lifecycle of pods scheduled on the node, and it is the entity that, in the CTXD model, "controls" the pods (see 6.5.3). Kubelet instances are added as *subservices* of the AKS `Cloud` object.

Pods as hosts and containers as execution environments. Each pod is modeled as a `Host` of type `Pod`, parametrized by its namespace. Containers running inside the pod — including init containers and regular containers — are modeled as nested `ExecutionEnvironment` instances of type `Container`, recording the image reference and the current state (`running`, `waiting`, `terminated`). Each container is added as a subservice of the enclosing pod.

Pod networking. Network connectivity is recovered from the `k8s.v1.cni.cncf.io/network-status` annotation, which is populated by the CNI plugin and lists the interfaces attached to the pod with their addresses. The actuator parses this annotation and builds a `NetworkNode` containing one `Port` per interface, with the associated `IPInfo` entries. The `NetworkNode` is exposed as a subservice of the pod, so that the IP-level view of a workload is addressable as an independent service in the model.

6.5.2 Service Identification across Namespaces

In Kubernetes, the same pod name may be reused across different namespaces. A pure name-based identifier would therefore be ambiguous. To avoid collisions, the actuator generates each `SId` by combining the CTXD `type` and `subtype` with the Kubernetes namespace:

```
SId.create_from_service_type(exe_env, namespace=namespace)
```

This guarantees globally unique identifiers within the cluster and preserves the namespace as a first-class organizational dimension in the CTXD graph.

6.5.3 Relationships through Links

Structural mapping alone is not enough to capture the operational semantics of a Kubernetes cluster. Two kinds of relationships are modeled explicitly through `Link` objects.

Hosting links (node → kubelet). For each kubelet `Application` added as a subservice of the AKS cluster, a hosting link is created between the kubelet (guest) and the node on which it runs (host). The link uses `LinkType.hosting` and the appropriate `PeerRole` values:

```
peer = Peer(service_name=node.name, sid=node,
            role=PeerRole.host,
            consumer=self.get_consumer(sid=node))

self.links.append(Link(
    name=sub.name, sid=sub,
    description=sub.name + " hosted on " + str(node),
    role=PeerRole.guest,
    link_type=LinkType.hosting,
    peers=ArrayOf(Peer)([peer])
))
```

Controlling links (kubelet → pod). For each pod, the actuator identifies the kubelet that manages it (using the `spec.nodeName` of the pod, which is matched against the kubelets indexed by node name) and creates a controlling link. The kubelet plays the `control` role; the pod plays the `controlled` role:

```
peer = Peer(service_name=kubelet_sid.name,
            sid=kubelet_sid,
            role=PeerRole.control,
            consumer=self.get_consumer(sid=kubelet_sid))

self.links.append(Link(
    name=pod.name, sid=pod.sid,
    description="Kubelet controls pod " + str(pod.name),
    link_type=LinkType.controlling,
    role=PeerRole.controlled,
    peers=ArrayOf(Peer)([peer])
))
```

Handling missing nodes. A subtle edge case occurs when a pod refers, through `spec.nodeName`, to a node that was not returned by `kubectl get nodes` (this can happen, for instance, on pods that are still being scheduled or whose node has just been removed from the cluster). In this situation the actuator synthesizes a placeholder `SId` for the node, with type `ExecutionEnvironment/OS` and the original name, and uses it as the controlling peer. This guarantees that the resulting graph is well-formed even when the cluster state is transiently inconsistent.

6.5.4 Actuator Workflow

The actuator extends the base `CTXDActuator` and exposes a single entry point, `discover_context()`, which orchestrates the entire process:

```
def discover_context(self):
    self.discover_services()
    self.discover_links()
```

`discover_services()` is itself decomposed into two phases. The first phase, `_discover_AKS_cloud()`, queries the nodes and instantiates the cluster, the node execution environments and the kubelet applications. The second phase `_discover_AKS_pods()`, iterates over the namespaces, retrieves all pods, and produces the pod hosts, container execution environments and pod network nodes. The two phases are intentionally separated, so that the node index built by the first phase can be reused when binding pods to their kubelets in the second one. Finally, `discover_links()` consumes the services produced above and generates the hosting and controlling links described in Section 6.5.3.

6.6 Discovery Results

The actuator has been executed against a production AKS cluster used to host the MIRANDA platform. Figure 6.2 shows the complete graph produced by a single invocation of `discover_context()`: each rectangular node is a CTXD service, red directed edges are controlling links (kubelet → pod), and blue edges are hosting links (node → kubelet). The figures reported in the rest of this section are extracted from that execution.

Nodes. The discovery returns seven worker nodes, belonging to two node pools: one system-purpose node (`aks-system-35229103-vmss000009`) and six user-purpose nodes of the `d4ds` pool (`aks-d4ds-76825604-vmss00001n` through `...vmss00001s`). Each of them is mapped to an `ExecutionEnvironment` of type `OS` and to a corresponding `kubelet` application exposed as a subservice of the cluster.

Namespaces and pods. The cluster is partitioned into twelve namespaces, summarized in Table 6.3 together with the number of pods discovered in each one. The largest namespace is `mir-prod`, which hosts the application workloads of the MIRANDA platform; `kube-system` is the second largest and groups the system components managed by AKS (CNS, CSI drivers, kube-proxy, CoreDNS, metrics-server, konnectivity, and the related daemonsets). The remaining namespaces host supporting services typical of a cloud-native deployment: GitOps controllers (`flux-system`), service mesh and tracing (`istio-system`, `kiali`), monitoring (`prometheus`), logging (`fluentd`, `elastic-system`), certificate management (`cert-manager`), resource recommendation (`goldilocks`), and operator dashboards (`portainer`). The `aks-command` namespace contains the ephemeral job pods spawned by `az aks command invoke` itself, which the actuator discovers like any other workload.

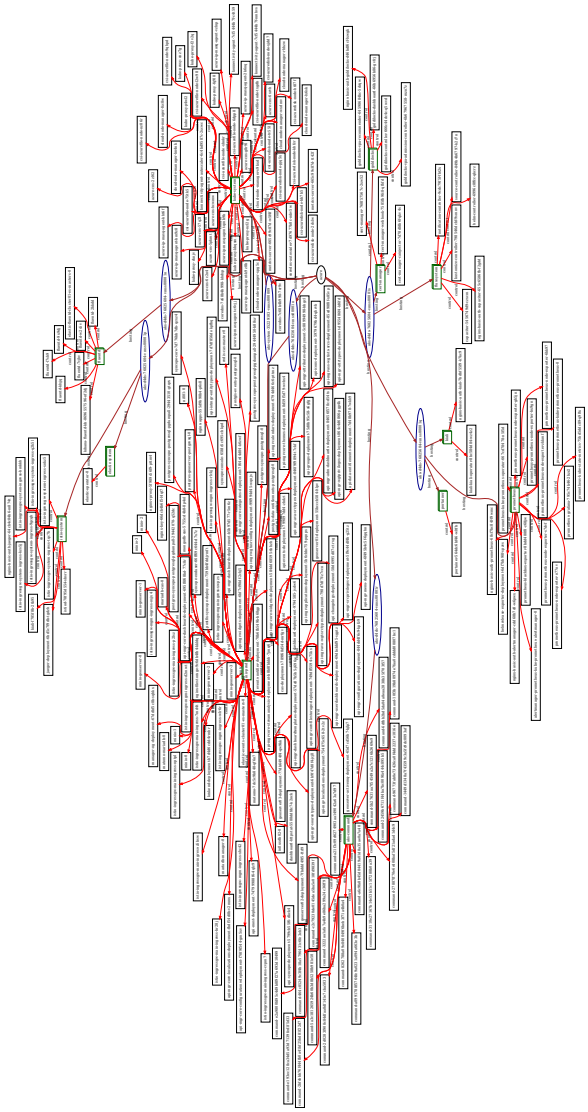

Figure 6.2. Discovery graph of the validation AKS cluster. Rectangles are CTXD services, red edges are controlling links and blue edges are hosting links.

Links. For each of the seven nodes, the actuator produces one hosting link between the node and the kubelet that runs on it. For each of the 210 discovered pods, it produces one controlling link between the kubelet of the hosting node and the pod itself. The resulting graph therefore contains seven hosting links and 210 controlling links, which is consistent with the edge structure visible in Figure 6.2.

Containers and networking. On top of the pods, the actuator produces one `ExecutionEnvironment` per init container and per regular container, together with one `NetworkNode` per pod. The `NetworkNode` is populated from the `k8s.v1.cni.cncf.io/network-status` annotation when this is set by the CNI plugin; otherwise it is still produced, but with an empty interface list. Containers and network nodes are not shown in Figure 6.2 to keep the diagram readable, but they are present in the underlying service set returned by the actuator.

Namespace	Pods
mir-prod	89
kube-system	50
aks-command	20
prometheus	15
istio-system	10
fluentd	8
flux-system	6
goldilocks	5
cert-manager	3
kiali	2
elastic-system	1
portainer	1
Total	**210**

Table 6.3. Pods discovered per namespace on the validation cluster.

Operational behavior. Across the discovery, the actuator handled correctly the case of pods whose `spec.nodeName` could not be resolved against the node inventory (for instance ephemeral pods of `aks-command` created while `kubectl get nodes` was being parsed): in such cases a placeholder `SId` was synthesized for the missing node, as described in Section 6.5.3, so that the corresponding controlling links could still be created and the graph remained well-formed.

The model produced is not a static snapshot. Each invocation of `discover_context()` regenerates the graph from the current state of the cluster, providing the MIRANDA orchestrator with an up-to-date view of the AKS environment that can be correlated with information from other actuators (notably Proxmox, Chapter 5) and consumed by the eBPF profile described in Chapter 7 for policy enforcement.

6.6.1 Performance of the Discovery Process

The actuator is instrumented through the shared module `otupy.actuators.ctxd.metrics`, which records the wall-clock duration of every `az aks command invoke` call and emits one `API_CALL` line per call together with an `API_SUMMARY` block at the end of each discovery cycle. The same module is used by the Proxmox actuator (Section 5.6.5), so the figures in this section and those reported for Proxmox are directly comparable.

To collect statistically meaningful measurements, `discover_context()` was invoked 20 times in sequence against the validation cluster (`miranda-prod-aks`). The resulting dataset contains 355 `API_CALL` records, with 19 or 20 samples per kubectl command (a few cycles toward the end of the experiment were truncated before the last few namespaces could be queried, hence the four labels with $n = 19$).

Table 6.4 summarises per-call statistics across the 20 runs. Mean and standard deviation are reported with the sample formula (denominator $n - 1$); all times are in seconds.

Call	N	Mean [s]	Std [s]	Min [s]	Max [s]
get_nodes	20	10.527	1.794	8.866	15.571
get_namespaces	20	10.239	2.939	7.392	20.066
get_pods:aks-command	20	11.529	2.582	8.601	17.676
get_pods:cert-manager	20	9.813	1.441	7.869	14.160
get_pods:default	20	10.036	4.278	7.636	27.828
get_pods:elastic-system	20	9.553	1.439	7.777	13.616
get_pods:fluentd	20	9.995	2.225	8.078	16.983
get_pods:flux-system	20	8.913	0.894	8.017	11.700
get_pods:goldilocks	20	8.877	1.292	7.213	12.945
get_pods:istio-system	20	9.880	2.238	7.989	16.592
get_pods:kiali	20	9.577	4.416	7.530	27.935
get_pods:kube-node-lease	20	8.469	1.146	5.128	11.032
get_pods:kube-public	20	9.977	3.347	7.429	23.196
get_pods:kube-system	19	12.115	3.120	8.899	20.729
get_pods:metallb-system	19	9.001	1.217	7.461	11.366
get_pods:mir-prod	19	13.232	4.919	8.643	24.838
get_pods:portainer	19	9.713	3.157	7.726	21.442
get_pods:prometheus	19	10.579	2.797	8.391	18.359
Aggregate	**355**	**10.101**	**2.933**	**5.128**	**27.935**

Table 6.4. Per-call duration of `az aks command invoke`, aggregated over 20 discovery cycles. All times are in seconds. Mean and standard deviation are computed with the sample formula.

Aggregate behavior. Across all 355 recorded calls, the mean per-call latency is $\bar{t} = 10.10\,\text{s}$ with a standard deviation $\sigma = 2.93\,\text{s}$. The fastest observed call took 5.13 s and the slowest 27.94 s, a 5.4× ratio that confirms the distribution is right-skewed with occasional large outliers rather than a tight Gaussian. A single discovery cycle therefore takes between approximately two and four minutes, dominated by the time spent in the ∼18 remote invocations.

Latency is dominated by per-invocation overhead. The most informative subset of Table 6.4 is formed by the empty namespaces (`default`, `goldilocks`, `kiali`, `kube-node-lease`, `kube-public`, `metallb-system`, `portainer`), whose response is a 218-byte JSON envelope. Across the seven labels and 20 runs, 138 such empty-payload calls were recorded with a mean of 9.38 s. This provides a direct measurement of the per-invocation overhead introduced by `az aks command invoke`: each call spawns an ephemeral job pod in the cluster's `aks-command` namespace, waits for it to be scheduled and started, executes `kubectl` inside the pod, and finally collects the output through the Azure regional control endpoint. The Pearson correlation between response size and elapsed time across the full dataset is $r = 0.35$ – weak positive – confirming that the payload-dependent component is small compared to the fixed cost. The two heaviest namespaces in the cluster, `kube-system` (524 KB) and `mir-prod` (524 KB), are visibly slower than the average (12.1 s and 13.2 s respectively) but still within 40% of the empty-namespace floor.

The `aks-command` outlier. With a mean of 11.53 s, `get_pods -n aks-command` is consistently among the slowest queries of each cycle. This is consistent with the fact that the `aks-command` namespace contains the ephemeral job pods spawned by `az aks command invoke` itself: at the moment the query is issued several of those pods are in the process of being created or terminated, which inflates both the payload size and the API server response time. The phenomenon is intrinsic to the chosen access mechanism and cannot be mitigated at the client side.

Implications for the design choices. The measurements quantify the trade-off introduced in Section 6.4.1, when the actuator switched from a single `kubectl get pods -A` call to per-namespace iteration. On the validation cluster this strategy issues 16 pod-listing calls instead of one, at a roughly constant cost of ∼10 s each. Aggregating nodes and namespaces, the total discovery time is dominated by this fixed cost and grows linearly with the number of namespaces rather than with the number of pods. Reducing the number of invocations is therefore the only effective optimization avenue: either by widening the response-size limit (an Azure-side change), by relaying queries through a long-lived in-cluster agent that bypasses the per-call pod-spawn overhead, or by caching discovery results between invocations when up-to-the-second freshness is not required. These alternatives are left as future work; in the current deployment a discovery cycle of approximately three minutes is acceptable, since the orchestrator triggers `discover_context()` on demand and not as part of a tight monitoring loop.

Summary. The validation confirms that the AKS actuator:

- discovers the cluster topology (cluster, nodes, kubelets) from the output of `kubectl get nodes`;
- enumerates pods on a per-namespace basis, recovering both their containers and their network interfaces;
- produces hosting links between nodes and kubelets, and controlling links between kubelets and pods;
- gracefully handles pods that reference nodes not present in the node inventory, producing a well-formed graph even when the cluster state is transiently inconsistent.

The resulting model is not a static snapshot: each invocation of `discover_context()` regenerates the graph from the live state of the cluster, providing the MIRANDA orchestrator with an up-to-date view of the AKS environment that can be correlated with information from other actuators (notably Proxmox, Chapter 5) and consumed by the eBPF profile described in Chapter 7 for policy enforcement.

Chapter 7

The eBPF Profile

7.1 Introduction to eBPF (Extended Berkeley Packet Filter)

eBPF is a revolutionary technology that enables sandboxed programs to run safely inside the Linux kernel. The operating system then guarantees safety and execution efficiency as if natively compiled with the aid of a Just-In-Time (JIT) compiler and verification engine. Today, eBPF is used extensively to drive a wide variety of use cases: Providing high-performance networking and load-balancing in modern data centers and cloud native environments, extracting fine-grained security observability data at low overhead, helping application developers trace applications, providing insights for performance troubleshooting, preventive application and container runtime security enforcement, and much more.

[4] This allows for deep system inspection and control without modifying the kernel source code or loading unstable kernel modules.

7.1.1 A bit of history of eBPF

BPF originally stood for Berkeley Packet Filter and was designed to provide a highly efficient means of packet filtering, surpassing traditional methods that required packets to traverse from kernel space to user space for processing. This innovative approach allowed for direct packet analysis and decision-making within the kernel, thereby significantly reducing latency and improving overall system throughput. eBPF is backward-compatible with traditional BPF, meaning it can handle all the tasks BPF was originally designed for, including:

- Packet Filtering: eBPF can still be used to filter network packets as BPF does, making it a seamless upgrade for tools like tcpdump.
- Raw Socket Processing: Applications using BPF for direct socket access can also use eBPF without modification.

However, eBPF extends these capabilities significantly, allowing for a much broader range of use cases that traditional BPF couldn't support, such as:

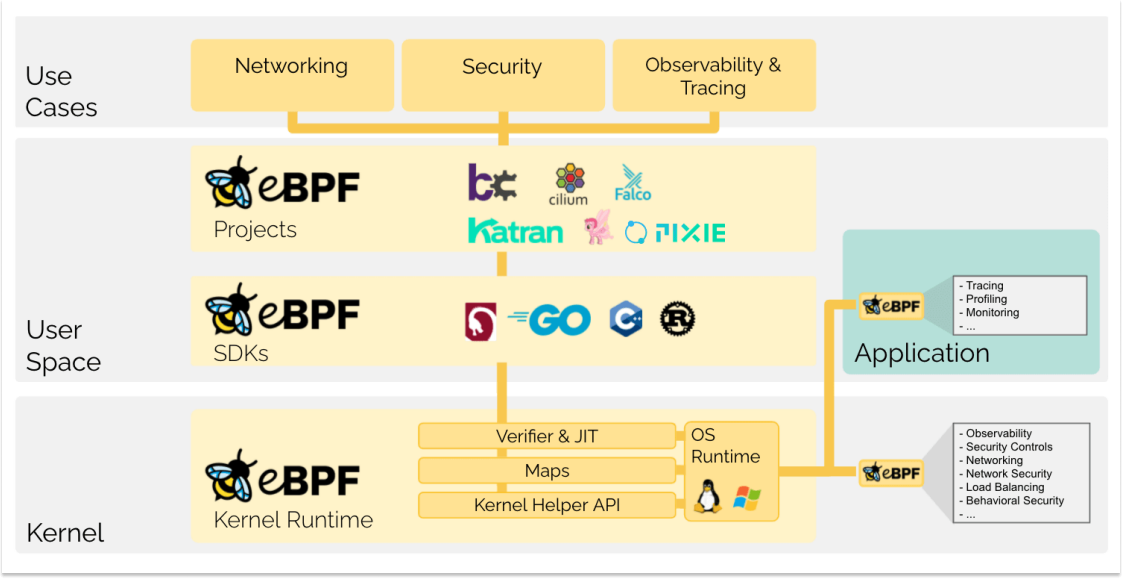

Figure 7.1. eBPF overview.

- Dynamic Event Attachment: eBPF can attach programs to various kernel hooks (e.g., system calls, kprobes, tracepoints, and more).
- Programmable Logic: eBPF allows for more complex logic in programs compared to BPF, which was limited in computational power.
- Enhanced Observability: Tools like bpftrace use eBPF to extract rich telemetry from the kernel, going far beyond basic packet inspection.

7.1.2 The complexity of writing an eBPF program

It is hard to write an eBPF program because this requires to be able to write low level C and be familiar with the Kernel Space. But today there are several libraries that are helpful to develop an eBPF program and tools like Cilium, bcc, or bpftrace provide an abstraction on top of eBPF and do not require writing programs directly but instead offer the ability to specify intent-based definitions which are then implemented with eBPF.

7.1.3 Architecture and Operating Principles

eBPF programs are written, like said before, in a restricted C, compiled into eBPF bytecode, and then loaded into the kernel. Before loading, a **Verifier** statically analyzes the bytecode to ensure it is safe—for example, it checks that the program will always terminate and will not access arbitrary memory. This safety guarantee is paramount.

As the program is loaded into the Linux kernel, it passes through two steps before being attached to the requested hook:

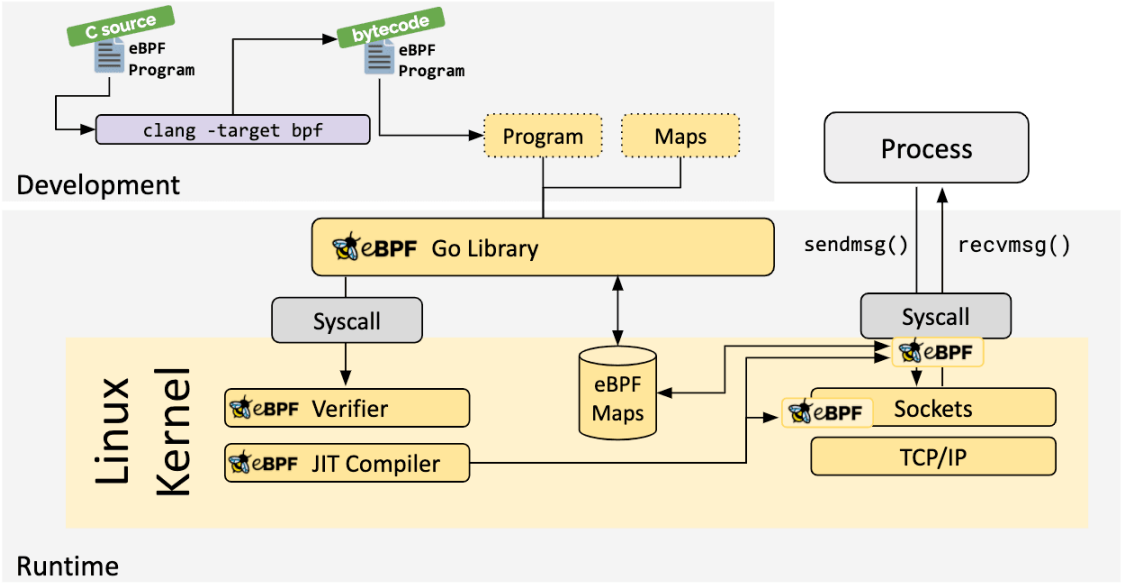

Figure 7.2. eBPF architecture.

Verification

The verification step ensures that the eBPF program is safe to run.

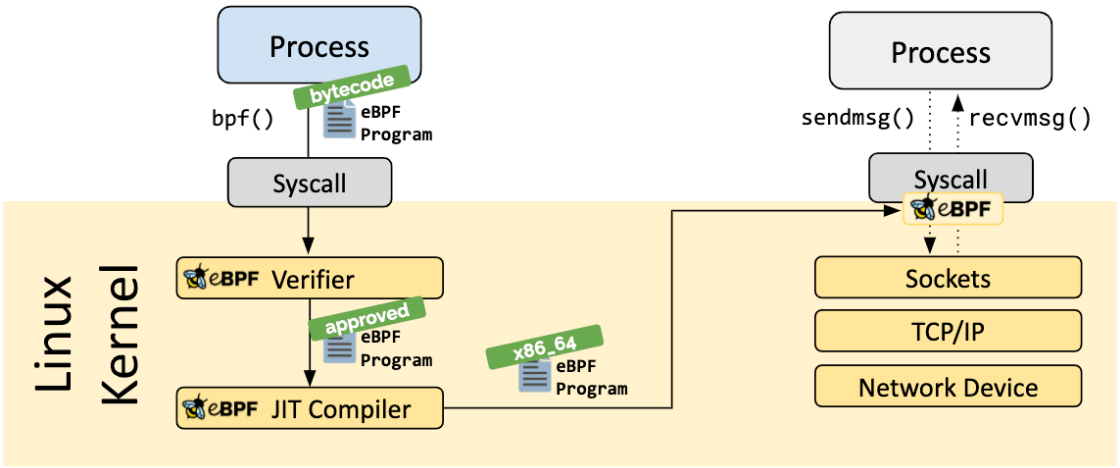

Figure 7.3. eBPF loader.

It validates that the program meets several conditions, for example:

- The process loading the eBPF program holds the required capabilities (privileges). Unless unprivileged eBPF is enabled, only privileged processes can load eBPF programs.
- The program does not crash or otherwise harm the system.

- The program always runs to completion (i.e. the program does not sit in a loop forever, holding up further processing).

Compilation

The Just-in-Time (JIT) compilation step translates the generic bytecode of the program into the machine specific instruction set to optimize execution speed of the program. This makes eBPF programs run as efficiently as natively compiled kernel code or as code loaded as a kernel module.

7.1.4 eBPF's effects on Linux Kernel

The main purpose of the Linux kernel is to abstract the hardware or virtual hardware and provide a consistent API (system calls) allowing for applications to run and share the resources. In order to achieve this, a wide set of subsystems and layers are maintained to distribute these responsibilities. Each subsystem typically allows for some level of configuration to account for different needs of users. If a desired behavior cannot be configured, a kernel change is required, historically, leaving two options:

- Native Support
  - Change kernel source code and convince the Linux kernel community that the change is required.
  - Wait several years for the new kernel version to become a commodity.
- Kernel Module
  - Write a kernel module
  - Fix it up regularly, as every kernel release may break it
  - Risk corrupting your Linux kernel due to lack of security boundaries

With eBPF, a new option is available that allows for reprogramming the behavior of the Linux kernel without requiring changes to kernel source code or loading a kernel module

7.1.5 Use Cases and Security Implications

eBPF is widely used for high-performance networking, observability (telemetry), and security. In the MIRANDA context, the goal is to use eBPF as a dynamic security actuator. An OpenC2 command could instruct the service developed in this thesis to load an eBPF program that, for example, monitors specific system calls, blocks traffic from a malicious IP at the kernel level, or gathers low-level telemetry on a suspicious process.

7.2 Objectives

The eBPF Actuator Profile aims to enable the remote configuration, deployment, and control of eBPF-based security and observability functions. It is primarily conceived as a standardized mechanism for managing high-performance, kernel-level security components through OpenC2 commands. By providing a unified method to attach and manage eBPF programs remotely, the eBPF profile facilitates low-overhead interactions with the operating system kernel, allowing consistent control of modern, natively instrumented systems through a common interface. The primary objective of the eBPF profile is to abstract the complexities of the eBPF lifecycle such as loading compiled bytecode, configuring eBPF maps, attaching programs to specific kernel hooks (e.g., XDP, TC, kprobes, and tracepoints), and safely detaching them—regardless of the underlying Linux distribution or specific eBPF toolchain implementation.

7.3 Operational Workflow

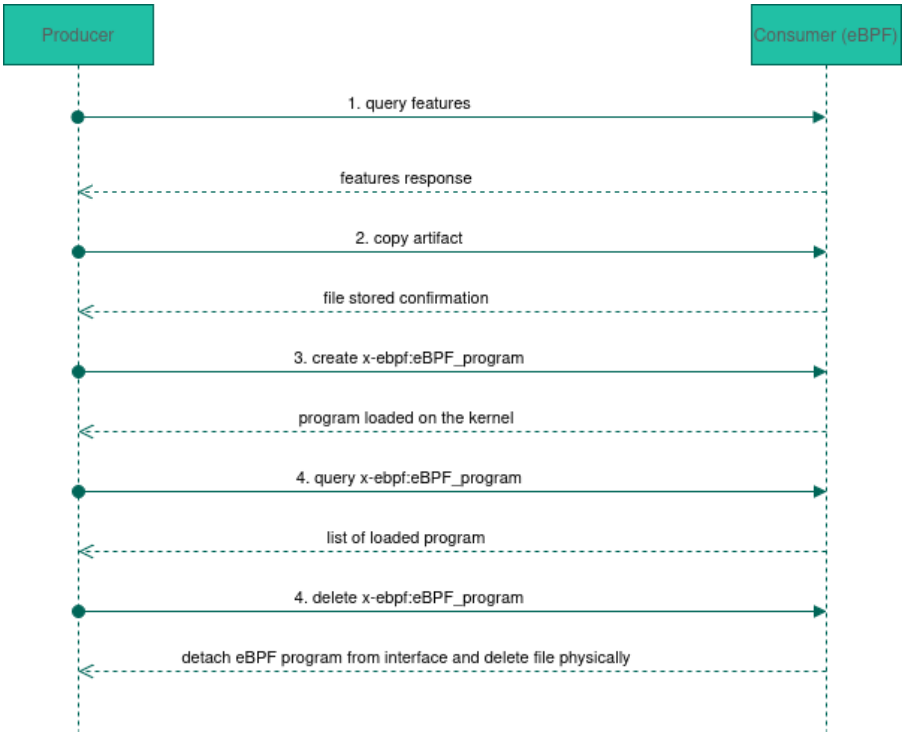

Figure 7.4. eBPF workflow.

Figure 7.4 illustrates the end-to-end lifecycle for managing an eBPF program through a producer–consumer interaction model. The workflow is structured as a sequence of controlled operations that progressively move from capability discovery to program deployment and eventual cleanup.

Initially, the *Producer* queries the *Consumer* (eBPF-enabled system) to discover supported features and capabilities, this part is not mandatory but can be useful to

ensures compatibility and allows the producer to adapt its behavior based on the target environment. Once the required capabilities are confirmed, the necessary artifact (e.g., compiled eBPF object file) is transferred to the consumer.

After the artifact is successfully stored, the producer issues a command to `create` the eBPF program into the kernel. At this stage, the program becomes active and can be triggered by relevant kernel events depending on its attachment point (e.g., XDP, tracepoints, kprobes). The producer may then query the system to retrieve a list of loaded programs, enabling monitoring and verification of successful deployment.

Finally, when the program is no longer needed, the producer issues a delete operation. This results in detaching the eBPF program from its hook point and removing any associated resources (e.g., pinned objects or file references). This step ensures proper cleanup and prevents resource leakage within the kernel.

This workflow can been extended associating a specific user space program thanks to another important work the `RCLI` profile that handles the lifecycle of user space programs. Thus, the eBPF profile must not been seen as disconnected piece but as part of larger system.

Step	Action	Target	Purpose
1	query	features	Discover supported capabilities
2	copy	artifact	Upload required eBPF object file
3	create	x-ebpf:eBPF_program	Create and attach eBPF program to the kernel
4	query	x-ebpf:eBPF_program	Retrieve list of loaded programs for monitoring
6	delete	x-ebpf:files	Remove stored artifacts and release resources

Table 7.1. Mapping of eBPF workflow steps to abstract command actions.

7.4 Command Components

This section describes the OpenC2 Command elements relevant to the eBPF Profile: Actions, Targets, Arguments, and optional Actuator Specifiers. While the eBPF Profile reuses Actions from the Language Specification, it introduces new Targets, data types, and a profile-specific Argument.

7.4.1 Actions

The eBPF Profile uses the following Actions:

ID	Name	Description
3	query	Request information about loaded eBPF programs, maps, or capabilities.
19	create	Attach an eBPF program to a specific kernel hook to begin execution.
20	delete	Unload an eBPF program from the kernel or remove an eBPF map.
28	copy	Transfer a file to the consumer

Table 7.2. Actions Used in the eBPF Profile

7.4.2 Targets

The eBPF Profile introduces new Targets

Name	Type	Purpose
x-ebpf:eBPF_program	Record	OpenC2-compliant target record for loading,removing or querying an eBPF TC program file.
Features	Record	OpenC2-compliant target record for retrieve the Actuator's characteristics
Artifact	Artifact	OpenC2-compliant target record for managing artifact

Table 7.3. Targets Used in the eBPF Profile

7.4.3 Arguments

The eBPF profile extends the base `Args` specification with additional fields required for program configuration and execution. These parameters define how the eBPF program is attached to kernel hooks, which network interfaces it operates on, and optional runtime constraints such as map dependencies.

Name	Type	Cardinality	Purpose
direction	Direction	0..1	Defines the traffic direction (e.g., ingress or egress) for packet processing or monitoring.
attach_type	AttachType	0..1	Specifies the kernel hook type where the eBPF program is attached.
interfaces	Interfaces	0..1	List of network interfaces where the eBPF program is active.
maps	ArrayOf(String)	0..*	Optional list of eBPF map names used by the program for state storage or communication.
maps_required	Boolean	0..1	Indicates whether the presence of maps is required for query actions

Table 7.4. Arguments used in the eBPF profile (Otupy extension).

7.4.4 Specifiers

There is a single specifier that determines if this specific Actuator should execute the command; may be null.

Name	Purpose
asset_id	check if the Actuator can run the command

Table 7.5. Specifiers Used in the eBPF Profile

7.5 Command Matrix

Target	Query	Create	Delete	Copy
x-ebpf:eBPF_program	X	X	X	
Features	X			
artifact				X

Table 7.6. Allowed Action-Target Pairs in the eBPF Profile

This matrix defines the valid interactions between OpenC2 Actions and the eBPF-specific Targets supported by this profile. This structured mapping ensures that kernel-level operations—such as loading bytecode or querying active hooks—are executed through authorized command combinations.

This matrix describes the combinations of OpenC2 actions and targets that are permitted within the eBPF profile. Each row represents a specific eBPF management record, while each column corresponds to a standard OpenC2 action. An "X" indicates that the specific action is valid for the corresponding target; empty cells denote unsupported operations.

For example, the Query action is utilized to retrieve the status of currently active programs via the `x-ebpf:eBPF_program` record, whereas the `CREATE` and `DELETE` actions are strictly mapped to the create and remove records respectively. This mapping outlines the core control operations available to security producers when interacting with the eBPF actuator, providing a concise overview of the profile's operational capabilities.

7.5.1 Features

The features target lets a controller (Producer) query an eBPF-enabled endpoint (Consumer) to learn what capabilities it supports. This includes the available eBPF programs, hooks, and any interface-specific extensions exposed by the endpoint. The purpose is to enable capability discovery and interoperability between the controller and the eBPF-based system.

For an eBPF-based actuator profile, the feature set can be extended to expose the allow-listed operations or program interactions (for example, permitted eBPF program attachments, maps, or management commands) that can be invoked remotely. This gives Producers controlled visibility into what actions are supported and how their parameters must be structured, helping ensure secure and predictable remote interaction with the eBPF environment.

7.5.2 Artifact

Type: Artifact (Record) The `artifact` target is employed together with the copy action to facilitate the transfer of files to the actuator host. Artifacts may be represented either as binary payloads or as URI-based references for remote access. Following the transfer, the actuator persists the file in its local database and associates it with the identity of the producing entity responsible for the operation.

7.5.3 x-ebpf:eBPF_Program

Type: Record

The **x-ebpf:eBPF_program** target enables remote management of eBPF programs attached to network interfaces through the `tc` subsystem. It allows a Producer to `create`, `delete`, and `query` eBPF programs on the Consumer using standardized OpenC2 commands. Each `eBPF_program` entry may include a `file` attribute of type `ProgramFile` that represents the eBPF program to be managed. When the `file` field is absent, the command applies to all eBPF programs currently active on the Consumer.

Data Model

The **x-ebpf:eBPF_program** target is formally defined as an OpenC2 `Record` type. Table 7.7 describes its fields, types, cardinality, and purpose.

Name	Type	Card.	Description
file	ProgramFile	0..1	The eBPF program file (compiled `.o` or source `.c`) to be loaded on the Consumer. Must be absent for `query` commands targeting all active programs.

Table 7.7. Fields of the `x-ebpf:eBPF_program` target record.

It is important to note that execution parameters such as `direction`, `attach_type`, and `interfaces` are not part of this target record. Following the OpenC2 language design principles, these parameters are passed as `Args` within the command, maintaining a clear separation between *what* the command targets and *how* the operation should be performed

Create

The create action attaches an allowlisted eBPF program to one or more network interfaces. The program to be loaded must be present in the filesystem of the Consumer, otherwise it has to be transferred via the copy action. A persistent database is used to maintain information about loaded eBPF programs and their attachment configuration. The database stores each hook instance together with its associated program metadata, enabling tracking, verification, and lifecycle management of deployed eBPF programs. When invoked, the actuator ensures that the clsact qdisc is present on the specified interface and then attaches the program using the `tc filter add` mechanism.

Delete

The remove action detaches one or more previously loaded eBPF programs from the specified interfaces. The actuator identifies the relevant filter entry by inspecting the currently attached filters and removing the matching program using its filter preference identifier. Before performing the detachment, the actuator verifies the presence of the corresponding entry in the internal database. If the program is found, its metadata is retrieved from the database, including attachment parameters such as interface, section, and attach type. This information is used to correctly locate and remove the associated filter entry. After successful removal from the system, the corresponding record is also deleted from the database to ensure consistency between the runtime state and the stored metadata.

Query

The `query` action retrieves the set of eBPF programs currently registered in the internal database for the requesting Producer.

Filtering is achieved through a combination of two OpenC2 command components:

- **Target** (`x-ebpf:eBPF_program`): the `file` field of the target provides file-level filters, specifically `file_name`, `file_path`, and `Section`.
- **Args**: the arguments provide attachment-level filters, specifically `direction`, `attach_type`, and `interfaces`.

If no filters are specified in either the target or the arguments, the actuator returns all programs associated with the requesting Producer. If filters are provided, only matching entries are returned.

The response may include metadata such as the interface name, direction, object file, section, and program name. Optionally, if the argument `maps_required` is set to `true`, the actuator also resolves the eBPF maps currently loaded in the kernel via `bpftool` and includes them in the response.

The following examples illustrate the two cases:

Unfiltered query Returns all programs owned by the Producer:

```
{
  "action": "query",
  "target": {
    "x-ebpf:eBPF_program": {}
  }
}
```

Filtered query Returns only programs matching the specified file and attachment criteria:

```
{
  "action": "query",
  "target": {
    "x-ebpf:eBPF_program": {
      "file": {
        "file_name": "monitor.o",
        "file_path": "/opt/ebpf/",
        "Section": "tc"
      }
    }
  },
  "args": {
    "x-ebpf": {
      "direction": {"Name": "ingress"},
      "attach_type": {"Name": "tc"},
      "interfaces": {"Names": ["eth0"]}
```

```
17         }
18     }
19 }
```

7.6 Response

Table 7.8. Fields Composing an OpenC2 Response in the eBPF Profile

Name	Type	Name Type	Card.	Description
status	Status-Code		1	The status code indicating the outcome of the command.
status_text	String		1	Human-readable description of the response status.
results	Results		0..1	The data returned by the actuator in response to the command.

7.6.1 Response Status Code

The eBPF profile implements standard OpenC2 response codes. Each code reflects the actuator's interpretation of the requested action outcome.

Table 7.9. eBPF Response Status Codes and Their Descriptions

Code	Status	Description
200	OK	Command executed successfully.
400	Bad Request	Invalid syntax or unsupported parameters.
404	Not Found	Requested eBPF program or actuator not found.
500	Internal Error	Unexpected actuator or system failure.
501	Not Implemented	Action or target not supported by this profile.

7.6.2 Common Results

Table 7.10 lists the standard result fields returned by the actuator in response to a `query features` command. These fields are defined in the OpenC2 Language Specification and are shared across all actuator profiles

Name	Type	Card.	Description
versions	Version	0..*	List of OpenC2 language versions supported by this actuator.
profiles	ArrayOf(Nsid)	0..1	List of profiles supported by this actuator (e.g., `x-ebpf`).
pairs	Action-Targets	0..1	List of valid target types per supported action.

Table 7.10. Common OpenC2 Result Fields

7.6.3 Specific Results

The eBPF profile extends the standard OpenC2 results to report actuator-specific information such as eBPF program, maps, and available commands. These results are returned in the results object of an OpenC2 Response.

Name	Type	Card.
program_status	ArrayOf(ProgramStatus)	0..*

Table 7.11. Field Specific Results

7.7 Data Model

This section defines the formal structure of the information entities used within the eBPF Profile. It translates abstract operational requirements into concrete data structures, specifying the types, cardinalities, and relationships of each field. By providing a standardized representation of programs, maps, and execution contexts, this model ensures interoperability and consistent state management across different OpenC2 implementations.

7.7.1 ProgramFile

The `ProgramFile` type is a `Record` representing the abstraction of an eBPF program file, whether in source (`.c`) or compiled (`.o`, `.ebpf`) format. Beyond basic file metadata, this record includes the `Section` (SEC) information, which is critical for identifying the program's entry point and its attachment type within the Linux kernel.

Name	Type	Card.	Description
file_name	String	1	The name of the eBPF program file. Must have a valid extension: `.c`, `.o`, `.bpf`, or `.ebpf`.
file_path	String	1	The filesystem path where the program file is located on the Consumer.
section	String	0..1	The eBPF section name (SEC macro), identifying the program entry point and kernel attachment type (e.g., `tc`, `xdp`). Auto-detected from source files if not provided.
isUri	Boolean	0..1	Indicates whether `file_path` is a URI reference rather than a local filesystem path. Defaults to `false`.

Table 7.12. Fields of the `ProgramFile` record.

7.7.2 Direction

The `Direction` type is an `Enumerated` representing the packet flow direction in eBPF programs, particularly in TC (Traffic Control) and XDP contexts. It restricts values to a controlled set of directions, ensuring semantic correctness when defining packet processing behavior.

Value	ID	Description
`ingress`	1	Apply rules to incoming traffic only.
`egress`	2	Apply rules to outgoing traffic only.
`both`	3	Apply rules to all traffic.

Table 7.13. Allowed values of the `Direction` enumeration.

7.7.3 AttachType

The `AttachType` type is an `Enumerated` representing the eBPF attachment type. It defines where an eBPF program can be attached inside the Linux networking stack. This abstraction ensures that only valid kernel attachment points are used when deploying eBPF programs.

Value	ID	Description
`tc`	1	Attach to Traffic Control (TC).

Table 7.14. Allowed values of the `AttachType` enumeration.

7.7.4 Interfaces

The `Interfaces` type is a `Record` representing a collection of network interfaces used for attaching eBPF programs. It validates interface names according to Linux naming conventions (e.g., `eth0`, `enp0s31f6`, `wlp3s0`).

Name	Type	Card.	Description
Names	ArrayOf(String)	0..*	List of network interface names on which the eBPF program is active. Each name is validated against Linux naming conventions (alphanumeric, `.`, `:`, `-`).

Table 7.15. Fields of the `Interfaces` record.

7.7.5 MapeBPF

The `MapeBPF` type is a `Record` representing an eBPF map definition inside a program. It encapsulates both the map name and its kernel-assigned identifier, providing a structured abstraction for interacting with eBPF map objects in a type-safe manner.

7.7.6 ProgramStatus

The `ProgramStatus` type is a `Results` record representing the runtime status of an eBPF program within the OpenC2-based execution framework. It aggregates all

Name	Type	Card.	Description
name	String	1	The name of the eBPF map as defined in the program source.
id	String	1	The kernel-assigned identifier of the map, retrieved via `bpftool map show`.

Table 7.16. Fields of the `MapeBPF` record.

relevant execution metadata, including the program source file, attachment type, execution direction, network interfaces, and associated eBPF maps. This structure provides a unified view of a deployed eBPF program and its runtime context.

Name	Type	Card.	Description
program	ProgramFile	0..1	The eBPF program file metadata, including file name, path, and kernel section.
direction	Direction	0..1	The traffic direction (`ingress`, `egress`, or `both`) of the attached program.
hook_point	AttachType	0..1	The kernel attachment point (e.g., `tc`).
interfaces	Interfaces	0..1	The list of network interfaces on which the program is active.
maps	ArrayOf(MapeBPF)	0..*	The eBPF maps associated with the program, populated only if `maps_required` is set to `true` in the query arguments.

Table 7.17. Fields of the `ProgramStatus` record.

7.8 Implementation Details

This section describes the implementation of the TC actuator, focusing on its execution model, internal structure, and security mechanisms. The implementation follows a modular design centered on secure command execution and controlled interaction with Linux traffic control (TC) utilities.

Execution Abstraction Layer

Command execution is abstracted through a base interface (`BaseCommandExecutor`), which defines a unified method for running system commands. Figure 7.5 illustrates the class hierarchy.

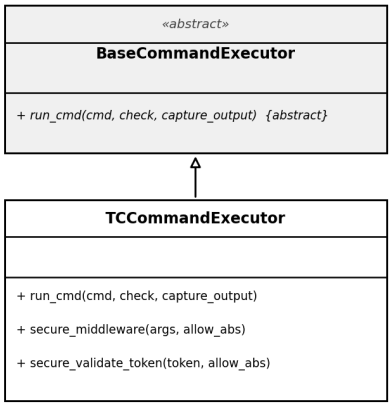

Figure 7.5. Class hierarchy of the Execution Abstraction Layer.

- **Executor abstraction** — `BaseCommandExecutor` is an abstract class that defines the single mandatory interface method `run_cmd()`, enforcing a consistent contract for all concrete implementations.
- **Concrete implementation** — `TCCommandExecutor` extends the base class with a security middleware layer applied before every command execution, supporting the `tc` attachment type defined by the current profile.

This design eliminates reliance on shell parsing and significantly reduces the risk of command injection vulnerabilities. The abstraction layer is structured to accommodate future concrete executors without modifying the base interface.

7.8.1 TCActuator Management

The management of TC-based eBPF programs is implemented through an actuator. The `TCActuator` class acts as an OpenC2-compliant execution layer that maps high-level commands to concrete Traffic Control operations.

- **Command dispatching** — Incoming OpenC2 commands are validated and routed based on their action type (`create`, `query`, `delete`, `copy`). Each action is delegated to a dedicated handler module, ensuring separation of concerns between control logic and execution logic.
- **Program deployment** — eBPF programs are deployed through the `create` and `copy` actions. The `copy` action transfers the program file to the Consumer's local filesystem. The `create` action then handles validation and TC-based attachment. If the provided artifact is a source file (`.c`), the actuator automatically invokes `clang -O2 -target bpf` to compile it into eBPF bytecode (`.o`) before loading. If the file is already compiled, the compilation step is skipped and the bytecode is loaded directly into the kernel via `tc filter add`.
- **Program removal** — Active TC filters are removed via the `delete` action, which identifies running programs through validated identifiers and safely triggers their detachment.
- **Program querying** — The `query` action retrieves the current system state by inspecting active TC configurations and returning structured representations of deployed eBPF programs.

7.8.2 Interface Management and TC Preconditioning

The `InterfaceManager` class provides a utility layer for managing network interfaces prior to eBPF program attachment. Its primary role is to ensure that target interfaces are in a valid and TC-ready state before any program deployment occurs, thereby preventing runtime failures during filter installation.

  - **Interface discovery** — The `list_up` method retrieves all network interfaces currently in the *UP* state by querying system configuration via the `ip` utility. This ensures that only active interfaces are considered for eBPF attachment.
  - **TC preconditioning** — The `ensure_clsact` method guarantees the presence of the `clsact` queuing discipline on a given interface. Since `clsact` is required to attach ingress and egress TC filters without interfering with normal packet forwarding, its presence is a prerequisite for safe program deployment.
  - **Safe attachment guarantee** — By verifying and, if necessary, installing the `clsact` qdisc before program loading, the system ensures that eBPF programs can be attached consistently across different interface states and system configurations.

This preconditioning layer is essential to decouple eBPF program deployment from system-level network state variability. It ensures that the execution environment satisfies all kernel-level requirements before invoking TC-based attachment operations, thereby reducing failure rates and improving deployment robustness.

Secure Command Construction and Execution Model

The `TCCommandExecutor` implements a multi-layer validation pipeline applied to every command before execution. The validation logic is encapsulated in the `secure_validate_token()` and `secure_middleware()` methods described in Section 7.8.

- **Token-level validation** — Each token in the command argument list is validated by `secure_validate_token()` before execution. The method rejects tokens containing forbidden characters (`;`, `|`, `&`, `$`, `>`, `<`, backticks) and unsafe patterns such as directory traversal sequences (`..`) and whitespace injection.
- **Path constraints** — The `allow_abs` flag in `secure_validate_token()` optionally disallows absolute paths (starting with `/` or `~`). All execution paths are resolved exclusively from the internal database after artifact ingestion, ensuring no remote URI is ever passed to the system executor.
- **Constrained command surface** — Only predefined system utilities (`tc`, `ip`, `clang`) are invoked by the actuator. Commands are executed via `subprocess.run()` without `shell=True`, fully bypassing shell interpretation.

7.8.3 Residual Risks and Limitations

Despite the applied safeguards, some limitations remain:

- **Dependence on system utilities** — The implementation relies on external tools (`tc`, `ip`), whose behavior may vary across environments.
- **Parsing fragility** — Output parsing from CLI tools may break if command output formats change.
- **Privilege requirements** — eBPF program loading requires elevated privileges, which must be controlled at the deployment level.

Chapter 8

Test and Validation

This chapter validates the prototype implementation of the eBPF OpenC2 profile built on top of the `otupy` library. The validation is carried out along two complementary axes. The first is a *functional* verification, which confirms that each supported OpenC2 action is correctly constructed, transmitted, executed on the target, and acknowledged. The second is a *quantitative* characterisation, which measures the execution time of each command under repeated, steady-state conditions and isolates the overhead introduced by the OpenC2 abstraction layer with respect to the equivalent native operation.

8.1 Objectives

The validation pursues three goals:

- verify the correct end-to-end behaviour of the four implemented actions — `query`, `create`, `copy` and `delete` — applied to the full lifecycle of an eBPF program;
- measure the execution time (latency) of each command, reporting both central tendency (mean, median) and dispersion (variance, standard deviation, range);
- quantify the overhead introduced by the OpenC2 API stack — message encoding, HTTP transport, parsing and command dispatch — by comparing the latency of an OpenC2 command against that of the corresponding native kernel/`tc` operation.

8.2 Test Environment

All tests were executed on a single host, where the OpenC2 *Producer* and the target *Consumer* run locally. Communication between the two is performed over HTTP on `127.0.0.1:8080`, using the JSON encoder provided by `otupy`. The eBPF program is attached to a network interface through the Traffic Control (TC) hook.

Logging was enabled throughout to trace command construction, transmission, and response handling.

Table 8.1. Test environment.

Component	Value
CPU	`Intel(R) Core(TM) i7-8750H CPU @ 2.20GHz`
Memory	`15Gi`
Operating system	`Ubuntu 24.04.4 LTS`
Kernel	`6.17.0-29-generic`
Python	`3.12.3` (virtual environment)
Transport	HTTP, `127.0.0.1:8080`
Network interface	`lo` (loopback)

8.3 eBPF Program Under Test

The validation uses a single, representative eBPF program drawn from the `eBPF_scripts` collection [5]. The program is *not* loaded from C source at run time: it is distributed and handled as a *precompiled object*, i.e. an ELF file (`tc_fl_kern.o`) that contains the verified eBPF bytecode produced from the C source (`tc_fl_kern.c`) by the LLVM/Clang BPF backend. The distinction is relevant for the profile, because the artifact that the `copy` action transfers and the `create` action loads is the object file, not the source.

The program implements per-flow statistics indexed by the IPv6 Flow Label: it is attached at the TC ingress hook and stores its counters in a single BPF map. Its main characteristics are summarised in Table 8.2.

Table 8.2. eBPF program used for the tests.

Property	Value
Object file	`tc_fl_kern.o` (ELF, eBPF bytecode)
Source	`tc_fl_kern.c` (C)
Toolchain	Clang/LLVM, target `bpf`
Object size	`15224 bytes`
Program section	`tc_flowlabel_stats`
Map	`fl_stats`
Hook / direction	TC, ingress
Interface	`lo`

8.4 Functional Test Cases

Each action was exercised against the eBPF program described above. The expected status for every command is HTTP/OpenC2 `200 (OK)`.

8.4.1 Query

Retrieves information from the target. Two query forms are exercised: `query_features` (independent of any file, returning supported versions, profiles and command pairs) and `query` on an eBPF program target with `maps_required=False`, which lists the currently attached programs.

Expected result. The system returns the supported features and the configuration/status details of the attached programs.

8.4.2 Copy

Transfers the compiled object to the remote storage location.

- **Artifact:** binary content of the eBPF object file;
- **Arguments:** destination storage path;
- integrity is preserved through an MD5 hash carried with the artifact.

Expected result. The file is transferred and stored, with hash verification confirming integrity.

8.4.3 Create

Loads and attaches the eBPF program object to the interface.

- **Target:** eBPF program file (`tc_fl_kern.o`, section `tc_flowlabel_stats`);
- **Arguments:** direction, attach type, interface, and map definitions (`fl_stats`).

Expected result. The program is loaded and attached to the specified interface at the TC ingress hook.

8.4.4 Delete

Removes the eBPF program from the system.

- **Target:** eBPF program;
- **Arguments:** direction, attach type, interface.

Expected result. The program is detached and the corresponding TC filter, stored file and database record are removed, returning the system to a clean state.

8.5 Functional Results

All four actions executed successfully and returned the expected `200 (OK)` status across all repetitions (Section 8.6). The logging output confirmed correct command construction, transmission, and response handling, demonstrating:

- correct implementation of the OpenC2 command structure for the eBPF profile;
- proper communication over the HTTP transport;
- accurate handling of the complete eBPF program lifecycle (transfer, load/attach, query, detach/remove).

8.6 Performance Evaluation

8.6.1 Methodology

Execution times were measured with a dedicated test runner that issues each command from the Producer and times the complete round trip — encoding, HTTP transmission, server-side execution, and response parsing — using a high-resolution monotonic clock (`time.perf_counter`). Latency is reported in milliseconds.

A naive per-command loop is not viable here, because the actions are *stateful* and interdependent: `delete` removes the TC filter, the stored file and the database record, so issuing the same command repeatedly in isolation would either fail or operate on an inconsistent state. To guarantee that every command is always exercised in its valid state, the runner executes a complete *lifecycle chain* per iteration:

$$\texttt{copy} \rightarrow \texttt{create} \rightarrow \texttt{query} \rightarrow \texttt{delete}.$$

Each iteration therefore starts and ends in a clean state, and each of the four commands is measured exactly once per iteration under valid conditions (all expected to return `200`). The `query_features` command, which requires no file and is idempotent, is measured independently in the same iteration.

The measurement protocol is as follows:

- a *warm-up* phase of W iterations is run first and discarded, to remove cold-start effects (process caches, first-touch allocations, connection setup);
- N further iterations are then recorded;
- for each command the following statistics are computed: mean, median, variance, standard deviation, minimum and maximum latency, together with the count of successful (200) responses;
- optionally, a *native baseline* phase (Section 8.6.3) measures the equivalent operations performed directly through the `tc` command, to isolate the OpenC2 overhead;

- results are exported to `test_results.json` and `test_results.csv`.

The configuration used here is $N = 30$ recorded iterations and $W = 5$ warm-up iterations.

8.6.2 Execution Time Results

Table 8.3 reports the per-command latency statistics over the $N = 30$ recorded iterations (warm-up discarded). The values are taken directly from `test_results.csv`; the variance is omitted from the table, as it is simply the square of the reported standard deviation.

Table 8.3. Per-command latency statistics over $N = 30$ iterations (warm-up discarded). All times in milliseconds.

Command	OK	Mean	Median	Std. dev.	Min	Max
`query_features`	30/30	17.676	16.866	3.537	12.715	31.262
`copy`	30/30	135.421	127.074	36.193	93.571	276.659
`create`	30/30	40.386	35.892	14.335	31.092	100.473
`query` (programs)	30/30	17.743	16.443	5.135	12.081	38.841
`delete`	30/30	44.288	42.859	4.961	37.225	59.248

The two query operations are the cheapest, both averaging about 17.7 ms, which is consistent with their read-only nature: they do not modify kernel state. The `create` and `delete` commands are more expensive, reflecting the privileged kernel work and the consumer-side state management they trigger. `delete` (mean 44.3 ms, median 42.9 ms) is the more stable of the two, while `create` (mean 40.4 ms, median 35.9 ms) shows a markedly higher dispersion ($\sigma = 14.3$ ms) driven by a few outliers reaching up to 100 ms; for this command the median is the more representative figure. Even by the median, `delete` remains heavier than `create`, which is explained by its additional cleanup steps (removing the stored file and the database record in addition to detaching the filter).

The `copy` action is by far the most expensive (mean 135.4 ms) and the most variable ($\sigma = 36.2$ ms, range 94–277 ms). This is consistent with its nature: unlike the other commands, `copy` carries the entire binary object as a Base64-encoded payload inside the JSON message, so its cost is dominated by serialisation, transport and disk I/O of the file content rather than by a fixed protocol overhead. Its variance is therefore driven by I/O scheduling and is not representative of the lightweight control commands.

8.6.3 OpenC2 Overhead Analysis

To separate the cost of the OpenC2 abstraction from the cost of the underlying kernel operation, the runner additionally measures a *native baseline* that performs

the equivalent operations directly through the `tc` command, bypassing OpenC2 entirely. Because the actions are stateful, the baseline mirrors the OpenC2 chain as a native lifecycle: it attaches the program (`tc filter add ...bpf obj`), lists the attached filters (`tc filter show`), and detaches it (`tc filter del`), timing each step over the same $N = 30$ iterations after warm-up. This phase requires root privileges, since attaching and detaching TC filters needs `CAP_NET_ADMIN`.

Each control command is then compared against its native counterpart. The OpenC2 overhead is defined as

$$\Delta_{\text{abs}} = \bar{t}_{\text{OpenC2}} - \bar{t}_{\text{native}}, \qquad \Delta_{\text{rel}} = \frac{\bar{t}_{\text{OpenC2}}}{\bar{t}_{\text{native}}},$$

where $\bar{t}$ denotes the mean latency. The `copy` action is deliberately excluded from this comparison: it has no native `tc` equivalent — it is a file transfer rather than a kernel operation — and its cost scales with the artifact size instead of representing a fixed protocol overhead. The results are reported in Table 8.4.

Table 8.4. OpenC2 overhead relative to the equivalent native `tc` operation (mean latency, milliseconds).

Operation	**Native (`tc`)**	**OpenC2**	**Overhead Δ_{abs}**	**Factor**
`create` (load + attach)	2.841	40.386	37.545	$\approx 14\times$
`query` (list programs)	1.876	17.743	15.867	$\approx 9\times$
`delete` (detach)	1.863	44.288	42.425	$\approx 24\times$

The most striking result is that the native `tc` operations are uniformly inexpensive — all below 3 ms, with sub-millisecond dispersion — so the kernel work itself accounts for only a small fraction of each OpenC2 command. The overwhelming majority of the measured latency is therefore introduced above the kernel, by the OpenC2 protocol stack and by the consumer-side bookkeeping.

The `query` path provides the cleanest estimate of the pure protocol overhead, because it performs essentially no consumer-side state management: its ≈ 15.9 ms overhead reflects JSON encoding/decoding, the HTTP request/response round trip, and command parsing and dispatch. The `create` and `delete` commands carry this same protocol cost but add further consumer-side work — creating or removing the database record, file handling, and filter management — which accounts for their larger overheads of 37.5 ms and 42.4 ms. Taking the query overhead as the protocol baseline, the additional consumer-side bookkeeping can be estimated at roughly 22 ms for `create` and 27 ms for `delete`, with `delete` again the heavier of the two because it performs more cleanup steps.

In relative terms the OpenC2 commands are 9 to 24 times slower than the bare `tc` operations. This ratio is inflated precisely because the native operations are so cheap (a denominator of 1–3 ms): a fixed protocol cost of around 16 ms dwarfs a kernel operation of 2 ms. In absolute terms, however, the overhead remains on the order of tens of milliseconds per command, which is entirely acceptable for command-and-control traffic: such traffic is event-driven and infrequent, not throughput-bound, and the cost buys a standardised, transport-agnostic and interoperable interface to the eBPF program lifecycle.

8.7 Limitations

The evaluation was conducted in a controlled, single-host environment and does not yet cover:

- distributed or remote deployments, where network latency would dominate the transport component;
- high-load or stress testing with concurrent commands;
- systematic error and exception-handling scenarios (e.g. malformed targets, verifier rejections, missing files);
- a wider set of eBPF programs of different sizes and complexity, which would allow correlating `create` latency with program size and verifier workload.

A further caveat concerns the native baseline: it loads the program through iproute2's own loader and excludes one-off qdisc set-up from the timed region, so it provides a close but not byte-identical reference for the consumer's `create` path. The reported overhead for `create` should therefore be read as an approximation rather than an exact figure.

8.8 Conclusion

The functional tests confirm that the prototype correctly implements the four OpenC2 actions over the complete eBPF program lifecycle, with all commands returning `200 (OK)` across all 30 iterations. The quantitative evaluation shows per-command latencies ranging from about 17.7 ms for the query operations to 135 ms for the payload-carrying `copy`. The native baseline reveals that the underlying `tc` operations all complete in under 3 ms, so essentially all of the measured latency is contributed by the OpenC2 layer and the consumer-side bookkeeping: a roughly 16 ms fixed protocol cost, plus an additional 20–27 ms of state management for the `create` and `delete` commands. While these overheads are large in relative terms, they remain small in absolute terms and well within the requirements of event-driven command-and-control traffic. Together, these results demonstrate both the correctness and the acceptable, measurable cost of managing eBPF programs through OpenC2.

Chapter 9

Conclusion

Modern computing infrastructures are inherently heterogeneous: they combine on-premise virtualization, public-cloud orchestration platforms, and deeply instrumented operating-system kernels, each with its own management interface and proprietary tooling. This fragmentation is one of the main obstacles to automated cyber defence, because situational awareness and active enforcement end up scattered across tools that do not interoperate. Within the context of the MIRANDA project, this thesis addresses this problem by adopting a single, standardised and vendor-agnostic language — OpenC2, realised through the otupy framework — and applying it coherently across the entire spectrum, from high-level infrastructure discovery down to low-level, kernel-resident security enforcement. To this end, the work delivered three concrete components that together demonstrate the feasibility of this approach

9.1 Summary of Contributions

The thesis produced three independent but complementary artefacts, all built on the same OpenC2 foundation:

- **A CTXD discovery actuator for Proxmox VE.** Building on the CTXD data model, the actuator queries the Proxmox REST API and translates physical nodes, QEMU virtual machines, LXC containers, and network bridges into a structured graph of `Service` and `Link` objects. It was validated against a real infrastructure, the Perseo Datacenter, where it correctly reconstructed the full topology of the node, its eleven virtual machines, and their network bridges.
- **A CTXD discovery actuator for Azure Kubernetes Service.** The actuator recovers the cluster, its worker nodes, the kubelets, the pods and their containers, and the per-pod network interfaces, dispatching every query through the `az aks command invoke` primitive. This design choice allows the actuator to operate against private clusters without VPNs or jump hosts, while inheriting Azure RBAC for authentication and auditability. It was validated against the production MIRANDA cluster, where it discovered seven

nodes across twelve namespaces and modelled 210 pods, generating the corresponding hosting and controlling links.

- **An OpenC2 actuator profile for eBPF.** A new actuator profile was designed and implemented to abstract the eBPF program lifecycle — transferring the compiled object (`copy`), loading and attaching it to the TC hook (`create`), enumerating active programs (`query`), and safely detaching them (`delete`) — behind a small set of standardised OpenC2 commands. The implementation emphasises safe execution, replacing shell invocation with a validated, constrained command surface, and maintains a persistent record of the loaded programs for consistent lifecycle management.

9.2 Discussion of Results

Because the two discovery actuators share a single instrumentation layer, their performance figures are directly comparable, and the comparison is itself informative. The Proxmox actuator averages roughly 61 ms per API call, whereas the AKS actuator averages about 10.1 s per call — a difference of more than two orders of magnitude. This gap is not a property of the discovery logic but of the access mechanism: Proxmox is queried directly over a local REST endpoint, while every AKS query spawns an ephemeral job pod through the Azure control plane. A practical consequence, quantified in Chapter 6, is that AKS discovery time grows with the number of namespaces rather than with the number of pods, which points clearly at where any future optimisation must act.

The eBPF profile was subjected to both functional and quantitative validation. All four actions returned the expected `200 (OK)` status across all thirty recorded iterations, confirming the correctness of the command structure, the HTTP transport, and the full program lifecycle. The performance evaluation revealed that the underlying native `tc` operations all complete in under 3 ms, so that almost the entire measured latency is contributed by the OpenC2 layer and the consumer-side bookkeeping: a roughly 16 ms fixed protocol cost, augmented by tens of milliseconds of state management for the stateful commands. In relative terms this overhead is large, but in absolute terms it remains on the order of tens of milliseconds per command, which is entirely acceptable for command-and-control traffic that is event-driven and infrequent rather than throughput-bound.

Beyond the individual figures, the central result of the thesis is qualitative: the same OpenC2 language, implemented through the same framework, proved expressive enough to span infrastructure inventory on two very different platforms and kernel-level enforcement on a third. The strict separation that OpenC2 enforces between the *intent* of an operation and its platform-specific *implementation* is precisely what made this breadth possible.

9.3 Limitations

The work has a some limitations, which were discussed in detail in the respective chapters and are summarised here:

- The CTXD profile does not yet define a `Storage` type, so the storage pools exposed by Proxmox are intentionally excluded from the model rather than mapped to a semantically incorrect type.
- The per-invocation overhead of the AKS actuator is intrinsic to the `az aks command invoke` access path and cannot be mitigated on the client side; discovery is produced on demand as a snapshot rather than as a continuous stream.
- The eBPF profile currently supports only the TC attachment point; other hooks such as XDP, kprobes, and tracepoints are described in the data model but not yet implemented.
- The eBPF validation was carried out in a controlled, single-host environment with a single test program, and did not include distributed deployments, high-load or stress testing, or systematic error-handling scenarios.
- The implementation depends on external system utilities (`tc`, `ip`) and on parsing their textual output, which is fragile across versions, and it requires elevated privileges to load eBPF programs.

9.4 Future Work

Several directions naturally extend the work presented here:

- **Enriching the CTXD model**, in particular by introducing a first-class `Storage` type and a more detailed network model, so that the discovery output covers resources that are currently omitted.
- **Reducing the AKS access overhead**, for example through a long-lived in-cluster relay agent that avoids the per-call pod-spawn cost, through result caching when up-to-the-second freshness is not required, or by widening the response-size limit on the Azure side.
- **Broadening the eBPF profile**, by implementing additional attachment types (XDP, kprobes, tracepoints), validating it against a wider set of programs of different sizes and complexity, and hardening it with distributed deployment, stress testing, and fault injection. A tighter integration with the RCLI profile would also allow the coordinated management of the associated user-space programs.

9.5 Final Remarks

Taken as a whole, this thesis demonstrates that standardised, vendor-agnostic security orchestration through OpenC2 is not only conceptually attractive but also practically achievable across markedly different layers of a modern infrastructure. By implementing discovery actuators for an on-premise virtualization platform and a managed cloud service, and an enforcement profile for the operating-system kernel, the work shows that a single command-and-control language can unify situational awareness and active defence. The modular design inherited from `otupy` ensures that the resulting building blocks are reusable: new actuators and profiles can be added without altering the core framework. In contributing these components to the MIRANDA ecosystem, the thesis takes a concrete step towards the broader goal of automated, interoperable, and transparent cyber defence over heterogeneous cloud and on-premise environments.

Bibliography

[1] MIRANDA, "MIRANDA project", https://www.mirandaproject.eu/, 2024, Accessed: October 25, 2025

[2] R. Matteo, "Otupy: A flexible, portable, and extensible framework for remote control of security functions", https://www.sciencedirect.com/science/article/pii/S016740482500286X

[3] P. S. S. GmbH, "Proxmox ve administration guide", https://pve.proxmox.com/pve-docs/pve-admin-guide.pdf, December 12, 2025

[4] eBPF Foundation, "ebpf - introduction, tutorials & community", https://ebpf.io/, 2024, Accessed: October 25, 2025

[5] E. ABDELJABBAR, "ebpf test tc", https://github.com/mattereppe/otupy/blob/2025_ENNAJADI_ABDELJABBAR/examples/ebpf-test.py, 2026