

POLITECNICO DI TORINO

MASTER's Degree in CYBERSECURITY



**Politecnico
di Torino**

MASTER's Degree Thesis

**AI-Powered Natural Language Interface
for OpenC2 Firewall Automation**

Supervisors

Prof. Daniele BRINGHENTI

Prof. Fulvio VALENZA

Dott. Matteo REPETTO

Dott. Daniele CANAVESE

Candidate

Alessandro MILANI

MARCH 2026

An AI-Driven Natural Language Interface for OpenC2-Based Firewall Security Automation

Alessandro Milani

Abstract

This thesis describes the design, implementation and validation of an intelligent agentic workflow that acts as an automated bridge between natural language security requirements and the Otupy implementation of the OpenC2 standard. The main objective is to create a robust system for translating human intent into executable firewall policies within the Otupy framework.

The proposed work aims to bridge the gap between high-level natural language requests, which are often ambiguous, and the rigorous, structured input format required by security orchestration tools. The system is specifically designed to manage the translation process, interpreting user inputs and converting them into precise Otupy commands for firewall configuration that are compliant with the target device specifications.

The implementation of the system has been completed in a Python-based application that operates a multi-stage agentic workflow built on LangGraph. First, it performs intent analysis and decomposition, parsing complex requests to isolate atomic security actions. It then executes a generation pipeline, which includes identifying the correct security domain, extracting command components, and ensuring command validity through specialized validation nodes.

To support the core generation logic, a second agent leverage a Graph Retrieval-Augmented Generation system. It is designed to employing a graph-based approach to retrieve relevant information from the Bill of Materials. The most critical function of this component is to enrich the generation process with asset-specific details, ensuring that the resulting Otupy commands are grounded in the actual operational reality of the network infrastructure.

In summary, this thesis presents an exploratory approach to bridging the gap between human security intent and automated policy enforcement. It demonstrates the feasibility of an automated, intelligent process where natural language requirements are dynamically interpreted, infrastructure context is leveraged, and structured commands are generated. The evaluation provides insights into the capabilities and limitations of this approach, establishing a foundation for future research in LLM-based security automation and identifying areas for refinement in translation accuracy and system robustness.

ACKNOWLEDGMENTS

To my family, for their unwavering support and encouragement throughout this journey.

Table of Contents

1	Introduction	1
1.1	The Challenge of Heterogeneous Environments	1
1.2	The Complexity of Manual Configuration	2
1.3	Proposed Approach	2
1.4	Thesis Structure	3
2	Background	5
2.1	OpenC2	5
2.1.1	Design Principles	5
2.1.2	Architecture and Terminology	6
2.1.3	Actuators and Profiles	6
2.1.4	SLPF Profile	7
2.2	Otupy Framework	8
2.2.1	Design Principles	9
2.2.2	Architecture and Components	9
2.2.3	Producer-Consumer Communication Model	10
2.3	Bill of Materials	10
2.3.1	Software Bill of Materials	11
2.3.2	Extended Bill of Materials & CycloneDX	12
2.4	Large Language Models	12
2.4.1	Transformer Architecture	13
2.4.2	Capabilities and Limitations	13
2.4.3	Embeddings	13
2.4.4	Retrieval Augmented Generation	14
2.4.5	LLM-based Workflows	15
2.5	LangGraph Framework	18
2.5.1	Motivation and Design	19
2.5.2	Core Components	19
2.5.3	Compilation and Execution	20
3	Thesis objectives	21
3.1	Translation objective	21
3.2	Validation	22
3.2.1	Validation Mechanisms	22

3.2.2	Evaluation Metrics	22
3.2.3	Test Execution	22
4	Related Works	23
4.1	Access Control Policy Generation	23
4.2	Firewall and Network Security Automation	23
4.3	LLM Fine-Tuning for Security Rule Generation	24
4.4	Research Gap and Positioning	25
5	Agentic Command Generation Framework	27
5.1	Technology Stack	27
5.2	System Workflow	28
5.2.1	Workflow Overview	28
5.2.2	LLM Response Schemas	29
5.3	Execution Pipeline	31
5.3.1	Preprocessing	31
5.3.2	Orchestration	32
5.3.3	Parsing	33
5.3.4	Human-in-the-Loop Clarification	36
5.3.5	Command Assembly and Execution	36
5.3.6	LLM Service Layer	37
6	Graph-based Knowledge Management System	38
6.1	Introduction	38
6.2	Technology Stack	40
6.3	System Design and Data Model	41
6.3.1	Project Structure	41
6.3.2	CTXD Data Model	42
6.3.3	Component Interaction	43
6.4	BOM Parsing and Graph Construction	43
6.4.1	CycloneDX BOM Parsing	44
6.4.2	Knowledge Graph Construction	45
6.4.3	Hybrid Search Index	46
6.4.4	Actuator Discovery Algorithm	48
6.5	API Overview	48
6.6	Security Request Enrichment Pipeline	48
6.6.1	Step 1: Parallel Entity Resolution and Fast Analysis	49
6.6.2	Step 2: Minimal Context Building	50
6.6.3	Step 3: LLM Enrichment	51
6.6.4	Integration with Command Generation Framework	52
7	Testing and Evaluation	54
7.1	Test Scenarios	54
7.1.1	Test Set Design Methodology	54

7.1.2	Infrastructure Coverage	55
7.1.3	Test Categories	55
7.2	Evaluation Framework	60
7.2.1	Semantic Validation	60
7.2.2	Consumer Validation	62
7.2.3	Multi-Command Validation	63
7.2.4	Metrics Collection	63
7.2.5	Report Generation	64
7.2.6	Evaluation Execution	64
7.2.7	Model Configurations	65
7.3	Results Analysis	66
7.3.1	Overall Comparison	66
7.3.2	Per-Category Analysis	67
7.3.3	Component-Level Accuracy	68
7.3.4	Failure Analysis	69
7.3.5	Clarification Behavior	70
7.3.6	Token Efficiency	70
7.3.7	Discussion	71
8	Conclusions	73
8.1	Summary of Results	73
8.2	Prompt Engineering and Its Limitations	74
8.3	Future Work	74
A	Result Analysis Appendix	76
	Bibliography	77
	Dedications	81

List of Figures

2.1	Architecture with native OpenC2 interfaces.	6
2.2	Otupy/OpenC2 architecture showing the Producer-Consumer communication model	10
2.3	Prompt chaining workflow. Tasks are decomposed into sequential steps where each LLM call processes the output of the previous one. Gates represent programmatic checks that validate intermediate results before proceeding.	16
2.4	Routing workflow. An LLM classifier directs inputs to specialized handlers based on input classification. Each handler can have optimized prompts and tools for its specific category.	16
2.5	Parallelization workflow variants. (a) Sectioning breaks tasks into independent subtasks executed concurrently. (b) Voting runs the same task multiple times to aggregate diverse outputs for higher confidence.	17
2.6	Orchestrator-workers workflow. A central orchestrator LLM dynamically breaks down tasks, delegates to worker LLMs, and synthesizes their results. Unlike parallelization, subtasks are not predefined but determined based on the specific input.	17
2.7	Evaluator-optimizer workflow. A generator LLM produces responses while an evaluator LLM provides feedback. The loop continues until the evaluator accepts the solution, analogous to iterative human review processes.	18
5.1	LangGraph workflow structure. Solid arrows indicate direct transitions; dashed arrows represent conditional routing.	28
5.2	LLM response schema hierarchy showing the base class with shared fields (including confidence and reasoning) and derived classes with component-specific fields.	30
5.3	Request decomposition example. A compound request mentioning two services is split into two atomic sub-requests, each processed independently through the scheduler-aggregator loop to produce one OpenC2 command per service.	33
5.4	Command assembly pipeline	36

6.1	Modules structure of the knowledge management system. The API module exposes HTTP endpoints and delegates to the services module, which cover the functionality. Cross-cutting concerns and data definitions are separated into dedicated modules.	42
6.2	CTXD domain model. Service is the central entity; Link and Peer capture inter-service relationships; Consumer identifies the responsible OpenC2 actuator endpoint. Port is an implementation extension. . .	43
6.3	Component interaction diagram. Solid black arrows show the main data flow through the pipeline; purple arrows indicate the build phase where <i>GraphStore</i> populates the search indices; blue dashed arrows show query-time read access from the indices to the components that use them; orange arrows (solid: call, dashed: return) show the <i>RAGService</i> –LLM interaction.	44
6.4	Enrichment pipeline for security requests. Step 1 resolves entity references in parallel using hybrid retrieval (BM25 + semantic similarity via RRF) and LLM-free programmatic analysis. Step 2 builds token-budgeted context containing only cited entities and their actuators. Step 3 invokes the LLM to produce the enriched request with resolved IPs and actuator specifications.	50
A.1	Latency data for each test.	76
A.2	Semantic score data for each test.	76
A.3	Component accuracy data for each test.	76

List of Tables

2.1	SLPF profile: supported actions and their valid targets.	8
5.1	Key Python libraries and their roles in the framework.	27
6.1	Additional Python libraries used in the knowledge management system.	40
6.2	Knowledge management system API endpoints.	49
7.1	Test category distribution.	56
7.2	Distribution of test cases by OpenC2 action.	56
7.3	Tracked components and comparison strategies.	61
7.4	Models evaluated.	65
7.5	Overall evaluation results across five models.	66
7.6	Pass rate (%) by test category.	67
7.7	Component-level exact-match accuracy (%). The N column indicates how many times each component appears in the test set.	68
7.8	Clarification behavior.	70
7.9	Token consumption across the full evaluation (56 tests).	71

Acronyms

GUI	Graphical User Interface.
API	Application Programming Interface.
LLM	Large Language Model.
OpenC2	Open Command and Control.
BOM	Bill of Materials.
RAG	Retrieval-Augmented Generation.
GraphRAG	Graph Retrieval-Augmented Generation.
SLPF	Stateless Packet Filtering.
HTTP	HyperText Transfer Protocol.
HTTPS	HyperText Transfer Protocol Secure.
TCP	Transmission Control Protocol.
UDP	User Datagram Protocol.
SCTP	Stream Control Transmission Protocol.
ICMP	Internet Control Message Protocol.
DNS	Domain Name System.
SSH	Secure Shell.
DDoS	Distributed Denial of Service.

CIDR	Classless Inter-Domain Routing.
JSON	JavaScript Object Notation.
CSV	Comma-Separated Values.
SBOM	Software Bill of Materials.
SPDX	Software Package Data Exchange.
RRF	Reciprocal Rank Fusion.
NLP	Natural Language Processing.
DAG	Directed Acyclic Graph.
NGAC	Next Generation Access Control.
IDS	Intrusion Detection System.
XACML	eXtensible Access Control Markup Language.
LoRA	Low-Rank Adaptation.
PEFT	Parameter-Efficient Fine-Tuning.

Chapter 1

Introduction

The modern cybersecurity landscape is characterized by an escalating volume and sophistication of cyber threats. ENISA’s Threat Landscape 2025 report documented nearly 4,900 incidents over a twelve-month period, highlighting a maturing threat environment characterized by rapid exploitation of vulnerabilities and growing complexity in adversary behaviors [1]. The integration of artificial intelligence into attack campaigns has further amplified this challenge, with AI-enhanced social engineering and automated exploitation techniques becoming increasingly prevalent [1, 2]. To defend against these threats, organizations must implement robust security policies across infrastructure that is increasingly complex and diverse. However, the operationalization of these policies faces two significant hurdles: the inherent difficulty of enforcing consistent security policies across heterogeneous environments, and the complexity present in the process of manually translating high level security intent into executable commands for specific security enforcement points. This thesis proposes a solution to these challenges through the creation of an AI agent capable of translating natural language requests into structured *Otupy* commands, a python library that implements the OpenC2 language, thereby bridging the gap between human intent and machine execution.

1.1 The Challenge of Heterogeneous Environments

Contemporary IT infrastructures are composed of interconnected subsystems, cloud resources, and physical devices that produce and process vast amounts of data [3]. While this technological diversity increases organizational resilience by preventing reliance on a single vendor, it simultaneously fragments the security landscape. Cyber defense systems frequently operate in isolation, utilizing proprietary Application Programming Interfaces (APIs) and distinct configuration languages.

As a result, applying a unified security policy across this heterogeneous environment is operationally expensive and error prone. The complexity is further exacerbated by the dynamic nature of these environments, where devices join and leave the network or change functions over time [3]. Even when local policies are enforced correctly on individual devices, ensuring that the global security posture

matches the organization’s intent remains a difficult validation task. To address this disintegration, the Open Command and Control (OpenC2) language was developed to provide a standardized, vendor agnostic interface for cyber defense [4].

1.2 The Complexity of Manual Configuration

While OpenC2 offers a standardized grammar for command and control, the practical implementation of these commands remains a bottleneck for human operators. The OpenC2 architecture separates the *what* (the action, such as blocking a connection) from the *how* (the specific implementation on a device) [5]. However, manually crafting these commands requires a deep understanding of the abstract notation, serialization formats in use, and the specific profiles required by different security devices in the infrastructure.

Furthermore, the manual process involves significant friction in identifying the correct target and security device that should protect it; a human analyst must typically specify exact infrastructure details—such as IP addresses, server names, or network ranges to create a valid command. If an analyst wishes to *block traffic to the production database*, they must manually look up the database’s IP, identify which security device controls access to it, and then construct the syntactically correct command.

This manual workflow is inefficient and does not scale to meet the velocity of modern cyberattacks. Existing open source implementations often suffer from rigid communication stacks or lack the flexibility to easily integrate new profiles and security enforcement points, complicating the analyst’s ability to respond quickly. Tools like Otupy have emerged to provide a flexible Python framework for OpenC2, decoupling the API from the underlying serialization and transport protocols. However, Otupy remains a programmatic tool; it does not inherently solve the semantic gap between a security analyst’s natural language intent and the strict syntax required for execution.

1.3 Proposed Approach

To address these challenges, this thesis proposes a two-component system that bridges natural language security intent and machine-executable OpenC2 commands.

The first component is an **agentic command generation framework** built on LangGraph, which models the translation process as a multi-stage workflow. When a user submits a natural language request, the system: preprocesses the input, decomposes complex requests into multiple simple sub-requests, parses each sub-request through parallel LLM invocations and assembles the final Otupy commands.

The second component is a **graph-based knowledge management service** that automatically enriches security requests with infrastructure context. Built on an Extended Bill of Materials (XBOM) representation of the organization’s assets. When the user mentions a component of the infrastructure, the system retrieves its IP address, and information about security entities protecting it.

Together, these components enable security analysts to express policies in natural language while the system handles the technical complexity of infrastructure resolution, command syntax, and device-specific configurations.

1.4 Thesis Structure

The remainder of this thesis is organized as follows:

- **Chapter 2, Background:** Introduces the base technologies and concepts required to understand the thesis. It covers the OpenC2 standard and its architecture, the Otupy framework as a flexible Python implementation of OpenC2, Large Language Models and their capabilities for natural language understanding, and knowledge enrichment techniques for providing context to the system.
- **Chapter 3, Thesis Objectives:** Formalizes the goals and defines the system’s design principles. It articulates the primary objective of enabling seamless translation from natural language to executable commands through a multi-stage pipeline (preprocessing, parsing, and execution), and the secondary objective of evaluating system performance across semantic accuracy, component-level precision, and LLM backend comparison.
- **Chapter 4, Related Works:** Reviews existing approaches in the intersection of artificial intelligence and security policy generation. It contextualizes this work within the broader landscape of AI-assisted cybersecurity and discusses relevant prior work in natural language processing for security, policy extraction, and LLM applications to security automation.
- **Chapter 5, Agentic Command Generation Framework:** Presents the core implementation of the system architecture. It describes the LangGraph-based orchestration framework for modeling the translation workflow as a directed graph, the state management mechanism that tracks command components and processing context, the multi-stage processing pipeline (preprocessing, parsing, clarification, and execution), and the integration of LLM-based decision-making for command generation.
- **Chapter 6, Graph-based Knowledge Management System:** Details the infrastructure context retrieval subsystem that enriches security requests with network topology and asset information. It describes the dual-index architecture combining semantic search over Bill of Materials data with knowledge graph queries and the REST API interface that integrates context retrieval into the preprocessing pipeline.
- **Chapter 7, Testing and Evaluation:** Presents the experimental validation methodology. It describes the test scenarios and datasets, the evaluation framework and metrics used to assess system performance across multiple dimensions

(semantic accuracy, component-level precision, and LLM efficiency), and the analysis of results including comparisons between different LLM backends.

- **Chapter 8, Conclusions:** Summarizes the key contributions of the thesis, discusses the implications of the findings, and outlines promising directions for future research including extensions to additional actuator profiles, integration with real-world security operations, and optimization of the LLM-based reasoning process.

Chapter 2

Background

The goal of this chapter is to provide the necessary background information to understand the context and the components used in this thesis. It covers the OpenC2 [6] standard and its Otupy implementation [5]. Then the focus shifts to the AI technologies employed, including Large Language Models and their architectures, LLM-based workflows that permit the orchestration of structured and stateful processes via the LangGraph framework [7], and knowledge enrichment techniques that provide the system with access to external knowledge bases, in this case a graph database representing the Extended Bill of Materials (XBOM) [8] of the network infrastructure.

2.1 OpenC2

Open Command and Control (OpenC2) [6] is a suite of specifications designed to standardize the command and control of cyber defense systems and functions. By providing a common language for machine-to-machine communication, OpenC2 achieves technology agnostic interoperability, enabling security automation and orchestration across a range of cyber security tools regardless of the vendor that developed them or the function they fulfill.

2.1.1 Design Principles

OpenC2 is built upon four core design principles:

- **Technology Agnostic:** It defines abstract, atomic actions independent of specific vendor implementations and enabling interoperability between diverse systems independently from any other aspects.
- **Concise:** The language minimizes overhead, focusing only on essential information required for defense actions.
- **Abstract:** Commands are defined abstractly, decoupled from encoding (how messages are serialized into bytes, e.g., JSON) and transfer (how messages are delivered over the network, e.g., HTTPS).

- **Extensible:** The language is designed to evolve alongside new cyber defense technologies.

2.1.2 Architecture and Terminology

The fundamental unit of communication is the **Command**, which consists of an *Action* (what to do, e.g., *deny*, *query*) and a *Target* (the object of the action, e.g., *ipv4_connection*, *file*). Commands may also include *Arguments* (contextual details like duration) and *Specifiers* (identifying specific actuators).

The OpenC2 architecture follows a Producer-Consumer model where components communicate through a native OpenC2 interface, eliminating the need for middleware. The architecture defines two fundamental roles:

- **Producer:** An entity that generates and sends Commands. The Producer acts as a client in the communication model and is typically integrated with security decision making systems or orchestrators.
- **Consumer:** An entity that receives Commands and executes them via one or more Actuators. The Consumer acts as a server and is typically co-located with or controls managed devices (e.g., firewalls, IDS).

While the specification frequently references *orchestrators* as mission managers that coordinate multiple Producers and Consumers (as clearly understandable by the Figure 2.1), the definition of orchestrator architectures and implementations is explicitly out of scope for OpenC2. An orchestrator conceptually implements the Producer role to issue commands to multiple consumer actuators, but its internal logic and architecture are left to implementation-specific design choices.

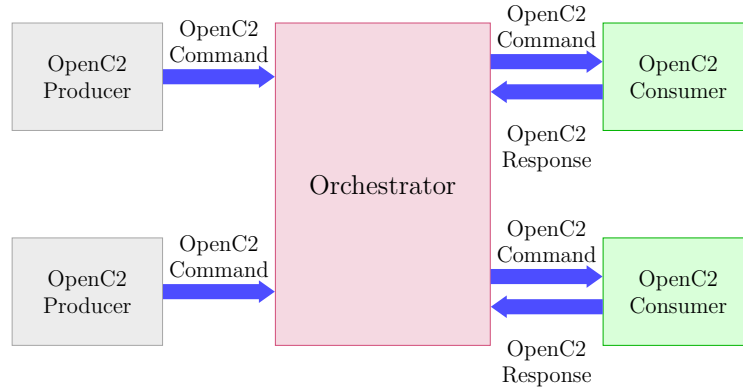


Figure 2.1: Architecture with native OpenC2 interfaces.

2.1.3 Actuators and Profiles

An **Actuator** is the function performed by the Consumer that executes the command [4]. Examples of actuators include stateless packet filtering, stateful packet filtering, or intrusion detection.

To handle the diversity of cyber defense functions without relying on device-specific drivers, OpenC2 defines **Actuator Profiles**. These profiles specify the subset of the OpenC2 language (actions, targets, arguments) relevant to a specific function and provide conformance requirements for enabling interoperability between different technologies.

A primary example is the Stateless Packet Filtering (SLPF) profile. It defines valid commands such as *deny ipv4_connection* and introduces specific arguments like *direction* (ingress/egress) and *insert_rule*. This function-centric approach allows a single profile to cover multiple device types (e.g., host firewalls, cloud firewalls, network ACLs).

2.1.4 SLPF Profile

The *Stateless Packet Filtering* (SLPF) profile [9] is one of the first actuator profiles standardized for OpenC2. It specifies the subset of the OpenC2 language applicable to devices that filter network traffic based on static packet header values (source and destination addresses, ports, and protocol), without considering connection state or application-layer data. Typical consumers implementing this profile include traditional firewalls, and host-based packet filters.

2.1.4.1 Supported Actions and Targets

The SLPF profile defines five actions, each constrained to operate on specific target types:

- **Allow:** Permits traffic that matches specified criteria. Valid targets are *ipv4_net*, *ipv6_net*¹, *ipv4_connection* and *ipv6_connection*².
- **Deny:** Blocks traffic that matches specified criteria. It accepts the same targets as **allow** and additionally supports the *drop_process* argument, which controls whether the device silently drops the packet or sends a notification (e.g., an ICMP unreachable message) to the source.
- **Query:** Retrieves information about the device's capabilities and configuration. The only valid target for this action is *features*, which returns the set of actions, targets, and profiles supported by the consumer.
- **Delete:** Removes an existing filtering rule. The target for this action is exclusively a *rule number*, an integer identifier that uniquely references a previously created rule.
- **Update:** Installs a new configuration file on the device. The target is a *file* containing the updated configuration.

Table 2.1 summarizes this mapping between actions and their valid targets, which is a core aspect of the profile: each action restricts the set of acceptable targets,

¹The **_net* targets represent IP addresses or network prefixes in CIDR notation (like 192.168.1.0/24 for IPv4).

²The **_connection* targets represent a 5-tuple consisting of source and destination IP addresses, source and destination ports, and the Layer 4 protocol (like TCP/UDP/SCTP).

ensuring that commands are semantically coherent with the device’s capabilities.

Table 2.1: SLPF profile: supported actions and their valid targets.

Action	Allowed Targets	Comments
allow / deny	ipv4_connection, ipv4_net, ipv6_connection, ipv6_net	Apply network traffic rules
query	features	Returns supported features
delete	slpf:rule_number	Deletes a rule by its ID number
update	file	Loads a new configuration from a file

2.1.4.2 Rule Numbers

When an *allow* or *deny* command is successfully executed, the consumer creates a new filtering rule and assigns it a unique, immutable *rule number*. This identifier is returned to the response and can subsequently be used by the producer to reference or delete that specific rule via the *delete* action. Producers may also request a specific rule number during rule creation through the *insert_rule* argument, which determines the position of the new rule in the rule set and consequently its evaluation priority.

2.1.4.3 Arguments

In addition to rule management, the profile defines several command arguments that refine the behavior of actions:

- **Direction:** Specifies whether the rule applies to *ingress* (incoming) or *egress* (outgoing) traffic.
- **Persistent:** Indicates whether the rule should survive device restarts.
- **Temporal controls:** Standard OpenC2 arguments such as *start_time*, *stop_time*, and *duration* to define time-bounded rules.
- **Response requested:** Controls the level of detail the consumer should include in its response (e.g., *complete*, *ack*, *none*).
- **Drop Process:** For *deny* actions, specifies whether the consumer should silently drop matching packets or send a notification to the source or even a False Ack.

A complete matrix detailing which arguments are valid for each action can be found in the official SLPF profile specification [9].

Understanding this profile is relevant because the system presented in this thesis generates OpenC2 commands targeting SLPF consumers, and needs to follow the constraints defined by the profile.

2.2 Otupy Framework

Otupy is a flexible, extensible, and portable open source Python framework designed to implement the OpenC2 language specification for the remote control of cybersecurity

functions [10]. The framework provides a comprehensive API set that enables developers to create, exchange, and parse OpenC2 messages without dealing directly with the underlying communication syntax, protocols, and encoding details. The motivation behind the Otupy creation arises from limitations in existing OpenC2 implementations, which often hardcode serialization formats and transfer protocols, thereby limiting the flexibility intrinsic to the OpenC2 design.

2.2.1 Design Principles

The core design principle of Otupy is the decoupling of main building blocks from each other, allowing the development of new profiles, serialization formats, and transfer protocols independently without modifications to the core library. This is achieved through three key architectural elements:

- **Modular communication stack:** Otupy cleanly separates transfer protocols from serialization formats, allowing any combination of supported protocols and encodings.
- **Inheritance model:** All data types inherit from base OpenC2 structures (Map, Array, Choice, Record), creating a complete inheritance tree that can be traversed bidirectionally during serialization and deserialization, ensuring consistency and correctness.
- **Meta-serialization:** An internal process converts OpenC2 objects into an intermediate representation of nested key:value pairs (Python dictionaries and lists). This format is agnostic of any specific serialization format and serves as a bridge between the OpenC2 object model and concrete encodings (like JSON, YAML, ...).

2.2.2 Architecture and Components

The Otupy architecture implements all objects defined in the OpenC2 Language Specification, using the same terminology to keep consistency. The framework accepts common data types from networking applications (IP addresses, URLs, domain names), facilitating integration with existing code.

The architecture comprises several key components:

- **Producer and Consumer objects:** Implement the core OpenC2 roles. The Producer creates and sends Commands, while the Consumer receives Commands and coordinates their execution.
- **Encoder interface:** Defines methods for serialization and deserialization. Concrete implementations (JSONEncoder, CBOREncoder, YAMLEncoder, XMLEncoder) translate between the intermediate meta-serialization format and specific encodings.

- **Transfer interface:** Manages network message delivery, adapting OpenC2 Messages to specific communication paradigms. Current implementations include HTTPSTransfer and MQTTTransfer, which embed necessary security services and transport protocols.
- **Actuator interface:** Provides the mapping between OpenC2 Commands and proprietary security function interfaces. Consists of a *run()* method that executes Commands and optionally returns a Response with execution results.

Figure 2.2 illustrates this architecture: the client side integrates business logic with the Producer, which sends Commands through the message transfer layer to the Consumer on the server side. The Consumer dispatches commands to the appropriate Actuators, which in turn interface with the managed devices.

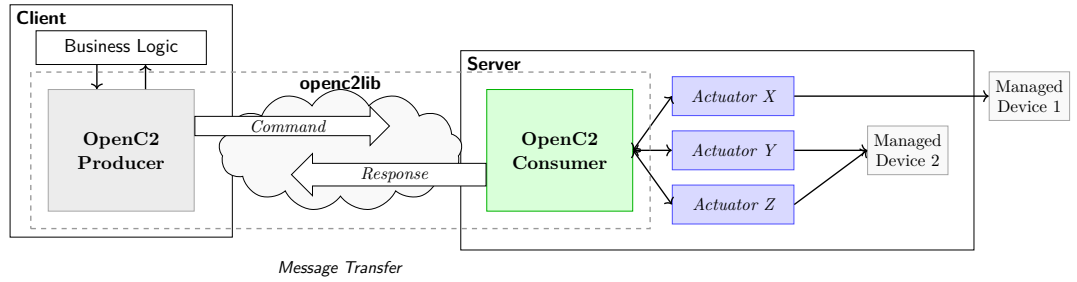


Figure 2.2: Otupy/OpenC2 architecture showing the Producer-Consumer communication model

2.2.3 Producer-Consumer Communication Model

Otupy adopts OpenC2’s client-server model, where Producer (client) and Consumer (server) are logical roles that can be assumed by the same entity (e.g., both may run on the same host).

The Producer is typically integrated with security decision making systems (security controllers, orchestrators and more comprehensive platforms) and handles Command creation, serialization, transmission, Response processing, and error handling. During installation, it is configured with specific Encoder and Transfer implementations that determine communication behavior.

The Consumer acts as the server side component, listening for Commands, deserializing and validating them, dispatching to appropriate Actuators, and returning Responses. It is initialized with an Encoder, Transfer protocol, and list of available Actuators that are managed by the dispatch method and the Command’s Actuator field or default policies.

2.3 Bill of Materials

A *Bill of Materials* (BOM) is a comprehensive inventory that lists all components, parts, and materials required to manufacture or assemble a product [11]. Originating

in traditional manufacturing industries, BOMs serve as foundational documents that specify raw materials, sub-assemblies, intermediate assemblies, and the hierarchical relationships among them.

The fundamental value of a BOM (that also lead to its use in other environments) lies in its ability to provide complete transparency into a product’s composition. By maintaining accurate BOMs, organizations can trace the origin of each component, manage inventory efficiently, ensure compliance with regulatory requirements, and respond quickly when defective parts are identified. This traceability becomes particularly critical when a component failure or vulnerability is discovered, enabling rapid identification of all affected products.

2.3.1 Software Bill of Materials

As software systems have grown increasingly complex, relying on numerous third-party libraries, frameworks, and dependencies, the BOM concept has been adapted to the software domain. A *Software Bill of Materials* (SBOM) is a machine-readable inventory listing all components, libraries, and modules that constitute a software product [12]. Just as a manufacturing BOM enables traceability of physical components, a SBOM provides visibility into the software supply chain, documenting both direct dependencies and transitive relationships.

A SBOM typically contains several essential attributes:

- **Component identification:** Supplier names, component names, and version information for each element.
- **Unique identifiers:** Standardized identifiers such as Common Platform Enumeration, Software Identification tags, or Package URLs for precise component tracking.
- **Dependency relationships:** Documentation of how components relate to each other, including both direct and transitive dependencies.
- **Cryptographic hashes:** Fingerprints that verify component integrity and authenticity.
- **License information:** Legal documentation covering open-source licenses and copyright details.

Two primary standards have emerged for SBOM representation: *SPDX* (Software Package Data Exchange) [13], developed by the Linux Foundation with a focus on license compliance, and *CycloneDX* [8], created by the OWASP Foundation with emphasis on application security and vulnerability management. Both standards support machine-readable formats including JSON and XML, enabling integration with automated security tools and continuous integration pipelines.

2.3.2 Extended Bill of Materials & CycloneDX

The *Extended Bill of Materials* (XBOM) concept, for which the CycloneDX standard represents a widely adopted implementation [8], extends the BOM paradigm beyond traditional software components to encompass the full spectrum of assets within modern digital ecosystems. Recognizing that contemporary systems comprise not only software but also hardware, services, machine learning models, and cryptographic assets, XBOM provides a unified framework for inventorying and managing diverse component types.

CycloneDX defines several specialized BOM types under the XBOM umbrella. Some non-exhaustive examples include:

- **SBOM**: Traditional software component inventories covering applications, libraries, and firmware.
- **HBOM** (Hardware Bill of Materials): Inventories of physical hardware components, integrated circuits, and embedded systems.
- **SaaS**BOM: Documentation of Software-as-a-Service compositions, including external services and API dependencies.
- **ML-BOM**: Inventories specific to machine learning systems, covering models, training data, and algorithmic components.
- **CBOM** (Cryptography Bill of Materials): Documentation of cryptographic assets including algorithms, certificates, and key materials.

The XBOM format can be represented in Serialization formats such as XML and JSON [14], facilitating integration with automated tools for vulnerability scanning, compliance verification, and risk assessment. Each component entry includes detailed metadata encompassing version information, licenses, known vulnerabilities, and provenance data. The format also supports the *Vulnerability Exploitability eXchange* (VEX), which provides contextual information about whether identified vulnerabilities are actually exploitable in a specific deployment context.

2.4 Large Language Models

Large Language Models (LLMs) are neural networks trained on vast corpora of text data, designed for natural language processing, and to be able to perform diverse tasks through in-context learning with minimal examples [15]. By scaling in size, these models demonstrate excellent capabilities not only in the general text generation but also more specific applications like: question answering, complex reasoning and code generation without task specific finetuning. LLMs have emerged as foundational components for a wide range of applications, from conversational assistants to code generation and automated decision-support systems.

2.4.1 Transformer Architecture

Modern LLMs are built upon the *Transformer* architecture, introduced by Vaswani et al. [16]. The key innovation of Transformers is the *multi-head self attention* mechanism, which allows the model to weigh the relevance of different parts of an input sequence when processing each token. Unlike recurrent architectures that process sequences sequentially, Transformers enable parallel computation across all positions, dramatically improving training efficiency on modern hardware.

The general architecture consists of stacked encoder and decoder blocks, each containing multi-head attention layers and feed-forward networks. While encoder only models (e.g., BERT [17]) excel at understanding and classification tasks, decoder only architectures (e.g., GPT [18]) have become dominant for generative applications due to their effectiveness in next token prediction and instruction following.

2.4.2 Capabilities and Limitations

As anticipated, LLMs exhibit several capabilities that make them valuable for complex automation tasks:

- **Natural Language Understanding:** The ability to parse and interpret human language, including ambiguous or context-dependent expressions.
- **Reasoning and Planning:** Capacity to decompose complex problems into steps and maintain coherent reasoning chains, particularly when guided by structured prompts [19].
- **In-Context Learning:** The ability to adapt to new tasks through examples provided in the prompt, without requiring parameter updates or fine-tuning [15].
- **Code Generation:** Proficiency in generating, explaining, and transforming structured outputs including programming code and formal specifications.

However, LLMs also present inherent limitations. They lack access to real time information beyond their training data, may produce plausible but incorrect outputs (hallucinations), and cannot reliably perform precise calculations or access external systems without tool augmentation [20].

2.4.3 Embeddings

Embeddings are dense vector representations that encode semantic meaning in a continuous high-dimensional space [21]. In the context of LLMs, embedding models transform text (words, sentences, or documents) into fixed length vectors where semantically similar content occupies proximate regions of the vector space.

These representations serve as the foundation for vector based retrieval systems, enabling efficient similarity search across large document collections. When a query is embedded alongside a corpus of documents, nearest neighbor algorithms can rapidly identify the most semantically relevant content without requiring exact keyword

matches. Modern embedding models, often derived from Transformer architectures, capture nuanced semantic relationships that enable effective retrieval even when queries and documents use different terminology to express similar concepts [22].

2.4.4 Retrieval Augmented Generation

Large Language Models, despite their general capabilities, face inherent limitations: they lack up to date information beyond their training cutoff, may be unfamiliar with domain specific or niche knowledge, and can generate hallucinations when uncertain [23]. *Retrieval Augmented Generation* (RAG) addresses these challenges by augmenting LLM prompts with relevant information retrieved from external knowledge sources, effectively shifting the burden of context provision from manual prompt engineering to automated search algorithms over curated corpora.

Rather than retraining models with new data, a time consuming and resource intensive process, RAG dynamically enhances prompts with retrieved information snippets from external resources. This approach significantly improves LLM accuracy while reducing hallucinations, as the model grounds its responses in factual, retrieved content rather than relying solely on parametric knowledge.

The effectiveness of RAG is inversely proportional to document noise: increased irrelevant information within retrieved documents significantly degrades performance [23]. The ideal RAG corpus is precisely tailored to the problem domain, providing concise and cogent knowledge representations. This observation has motivated research into structured knowledge forms that minimize noise while maximizing relevance.

2.4.4.1 RAG Architectures

RAG systems can be categorized into three primary architectural variants:

2.4.4.1.1 Vector-based RAG The traditional approach employs dense vector embeddings to represent both documents and queries in a high-dimensional semantic space. Similarity search algorithms (e.g., cosine similarity, approximate nearest neighbors) identify relevant document chunks, which are then concatenated with the user prompt. While computationally efficient and widely adopted, vector-based RAG treats documents as isolated fragments, potentially missing broader contextual relationships and semantic correlations across the corpus [23].

2.4.4.1.2 Graph-based RAG Knowledge Graphs (KGs) provide structured representations of domain knowledge as networks of entities and relationships. Unlike vector embeddings, KGs explicitly encode semantic connections, enabling more precise retrieval of relevant information with minimal noise [23]. Graph based RAG addresses the limitations of vector only approaches by incorporating broader semantics across documents, making it particularly appropriate for structured documents where relationships are already present and can be leveraged. However, subgraph

identification from large KGs remains computationally challenging (NP-hard if done exhaustively).

2.4.4.1.3 Hybrid RAG Contemporary systems increasingly combine vector and graph-based approaches, leveraging the computational efficiency of embeddings with the structural precision of knowledge graphs. Hybrid architectures may cascade retrieval strategies—first attempting structured KG queries, then falling back to vector based document retrieval—or fuse both representations through multi modal encoding mechanisms.

2.4.5 LLM-based Workflows

The integration of Large Language Models into complex systems has evolved beyond simple query response interactions towards sophisticated *LLM-based workflows* [24, 25]. These systems leverage LLMs as specialized reasoning components within structured, multistep processes, enabling the automation of complex tasks that previously required human expertise or rigid rule based systems. Workflows are particularly effective for well-defined tasks by offering consistency, and easier debugging [26]. They are a promising approach in scenarios where the problem structure is understood and can be broken into discrete stages, each handled by specialized LLM calls or deterministic logic. This approach trades the flexibility of autonomous agents for the reliability and maintainability of structured orchestration.

2.4.5.1 Workflow Design Patterns

Effective LLM-based workflows employ several compositional patterns that can be combined to handle complex tasks [25, 26]. These patterns represent building blocks that developers can shape and combine to fit different use cases, always favoring simplicity and adding complexity only when it demonstrably improves outcomes.

2.4.5.1.1 Prompt Chaining Prompt chaining decomposes a task into a sequence of steps, where each LLM call processes the output of the previous one (Figure 2.3). Programmatic checks (gates) can be added at intermediate steps to ensure the process remains on track before proceeding. This pattern is ideal for situations where the task can be cleanly decomposed into fixed subtasks, trading latency for higher accuracy by making each LLM call handle a simpler, more focused subtask. Examples include generating marketing copy and then translating it, or writing a document outline, validating it against criteria, and then producing the full document based on the approved outline [25].

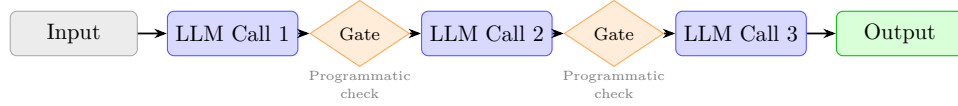


Figure 2.3: Prompt chaining workflow. Tasks are decomposed into sequential steps where each LLM call processes the output of the previous one. Gates represent programmatic checks that validate intermediate results before proceeding.

2.4.5.1.2 Routing Routing classifies an input and directs it to a specialized follow-up task (Figure 2.4). This workflow enables separation of concerns and allows optimization of prompts for specific categories without degrading performance on other input types. Classification can be handled either by an LLM or by traditional classification models. Routing is effective for complex tasks where distinct categories benefit from separate handling for instance, directing customer service queries (general questions, refund requests, technical support) to different downstream processes with specialized prompts and tools, or routing simple questions to smaller, cost efficient models while complex queries utilize more capable models [25].

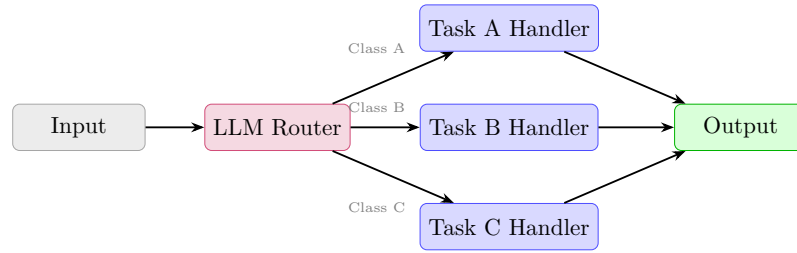


Figure 2.4: Routing workflow. An LLM classifier directs inputs to specialized handlers based on input classification. Each handler can have optimized prompts and tools for its specific category.

2.4.5.1.3 Parallelization Parallelization enables LLMs to work simultaneously on a task, with outputs aggregated programmatically (Figure 2.5). This workflow manifests in two key variations: *sectioning* breaks a task into independent subtasks run in parallel, while *voting* runs the same task multiple times to obtain diverse outputs. Parallelization is effective when subtasks can be executed concurrently for speed, or when multiple perspectives are needed for higher confidence results. For complex tasks with multiple considerations, LLMs generally perform better when each consideration is handled by a separate LLM call, allowing focused attention on each specific aspect. Examples include implementing guardrails where one model processes user queries while another screens for inappropriate content, or reviewing code for vulnerabilities with multiple prompts that flag issues independently [25].

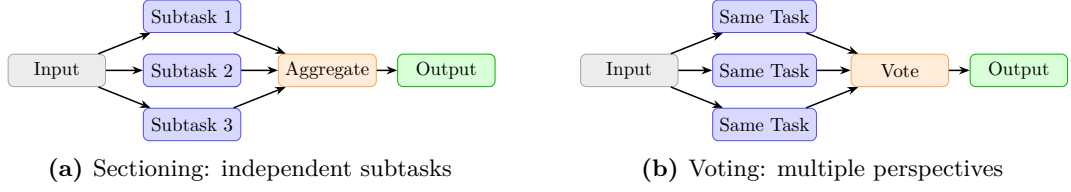


Figure 2.5: Parallelization workflow variants. (a) Sectioning breaks tasks into independent subtasks executed concurrently. (b) Voting runs the same task multiple times to aggregate diverse outputs for higher confidence.

2.4.5.1.4 Orchestrator-Workers In the orchestrator-workers workflow, a central LLM dynamically breaks down tasks, delegates them to worker LLMs, and synthesizes their results (Figure 2.6). While topographically similar to parallelization, the key difference is flexibility: subtasks are not predefined but determined by the orchestrator based on the specific input. This workflow suits complex tasks where the required subtasks cannot be predicted in advance for example, coding products that make complex changes to multiple files, where the number of files and the nature of changes depend on the task, or search tasks involving information gathering from multiple sources [25].

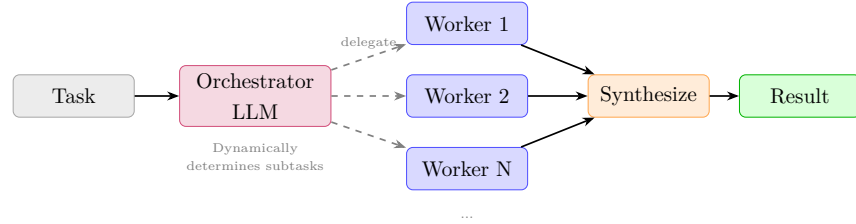


Figure 2.6: Orchestrator-workers workflow. A central orchestrator LLM dynamically breaks down tasks, delegates to worker LLMs, and synthesizes their results. Unlike parallelization, subtasks are not predefined but determined based on the specific input.

2.4.5.1.5 Evaluator-Optimizer In the evaluator-optimizer workflow, one LLM call generates a response while another provides evaluation and feedback in a loop (Figure 2.7). This pattern is effective when clear evaluation criteria exist and iterative refinement provides measurable value. The approach mirrors human iterative processes: a good fit exists when LLM responses can demonstrably improve from articulated feedback, and when the LLM can provide such feedback. Examples include literary translation where nuances might not be captured initially, but an evaluator LLM can provide useful critiques, or complex search tasks requiring multiple rounds of searching and analysis where the evaluator decides whether further searches are warranted [25].

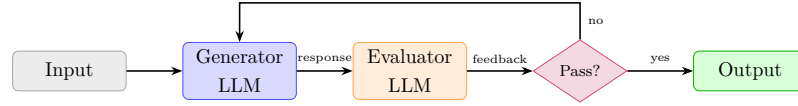


Figure 2.7: Evaluator-optimizer workflow. A generator LLM produces responses while an evaluator LLM provides feedback. The loop continues until the evaluator accepts the solution, analogous to iterative human review processes.

2.4.5.2 Workflow Components

Modern LLM workflows are constructed from three fundamental building blocks [26]:

- **Model:** The LLM serving as the reasoning engine. Model selection follows a pragmatic principle: establish performance baselines with capable models, then optimize for cost and latency by substituting smaller models where acceptable results can be maintained.
- **Tools:** External functions or APIs that extend LLM capabilities. Tools fall into three categories: *data tools* for information retrieval (database queries, document reading), *action tools* for system interaction (sending messages, updating records), and *orchestration tools* where other LLM workflows serve as reusable components.
- **Instructions:** Explicit prompts defining guidelines, constraints, and expected behaviors. Instructions include task descriptions, output format specifications, and guardrails to ensure safe and predictable operation.

The practical value of LLM-based workflows becomes apparent in scenarios requiring complex decision making, handling unstructured data, or managing intricate logic that resists traditional rule-based automation [26]. However, successful implementations consistently favor simple, composable patterns over complex abstractions, as excessive framework complexity can obscure underlying behavior and complicate debugging [25].

2.5 LangGraph Framework

LangGraph [7, 27] is a low-level orchestration framework built on top of LangChain, a modular toolkit providing foundational primitives for prompt engineering, memory handling, document loading, and tool integration in LLM applications. While LangChain offers these building blocks, LangGraph extends the ecosystem to enable stateful, graph-based workflows.

Unlike traditional data orchestration frameworks that operate as directed acyclic graphs (DAGs), LangGraph introduces the capability to create *cyclical graphs*, a critical distinction for agent runtimes where iterative reasoning is required [27].

2.5.1 Motivation and Design

The fundamental motivation behind LangGraph emerges from the need to introduce cycles into the runtime. These cycles use the LLM’s reasoning capabilities to determine what action to take next, effectively running the model in a loop until a satisfactory result is achieved, a paradigm often referred to as *agentic behavior* [27].

However, fully autonomous agent loops often prove too unpredictable for production environments. LangGraph addresses this by enabling *state machines*: controlled loops with human guided structure that balance flexibility with the predictability required for reliable systems [27].

2.5.2 Core Components

LangGraph exposes a focused interface built around three primary concepts: the state graph, nodes, and edges.

2.5.2.1 StateGraph

The *StateGraph* class represents the central graph structure. It is initialized with a *state definition*: a schema that declares all the attributes (with their types, default value and if are optional or not) composing the shared state object, which every node can read and update as the workflow executes. Each attribute in the schema is associated with a *reducer*, a rule that determines how new values produced by a node are merged with the existing state. LangGraph supports two main reducer strategies:

- **Override** (default): the value returned by a node completely replaces the previous one. This is suitable for attributes that represent a single, evolving quantity (e.g., the current query or a status flag).
- **Additive**: the value returned by a node is appended to the existing one, preserving its history. This is useful for attributes that accumulate information over time, such as message logs or lists of executed actions.

2.5.2.2 Nodes

Nodes represent discrete processing steps within the workflow. Each node is a function or LangChain Expression Language (LCEL)³ runnable that accepts the current state as input and returns a dictionary containing updates to apply to specific state attributes. LangGraph includes a special **END** node representing workflow termination ensuring that cycles can eventually conclude [27]. This separation in discrete function nodes also gives the possibility to seamlessly integrate non-LLM logic (e.g., data retrieval, validation, calculations and human input) alongside LLM calls within the same general structure.

³A syntax that use pipe symbols to simplify the design of complex workflows and chains using language models.

2.5.2.3 Edges

Three edge types structure the graph’s control flow:

- **Entry Point:** Designates which node receives the initial input when the graph is invoked.
- **Normal Edges:** Define unconditional transitions where one node always follows another (e.g., always calling the model after tool execution).
- **Conditional Edges:** Use routing functions often LLM powered to dynamically determine the next node based on the current state. These enable branching based on classification results, confidence scores, or any state derived condition.

2.5.3 Compilation and Execution

Once defined, a graph is compiled into a runnable object that exposes the standard LangChain interface:

- **.invoke():** Executes the graph synchronously, processing inputs through the defined nodes and edges until reaching the *END* node and then returning the final state.
- **.stream():** Executes the graph while streaming intermediate chunks of data as the execution progresses. Its main use is to give a better user experience when the graph includes long operations thanks to the incremental output.
- **.astream_log():** Usually used for debugging purposes, include detailed logging of state changes, tools calls and node executions.

Allowing seamless integration with existing LangChain applications. The compiled graph processes inputs by traversing nodes according to defined edges, updating state at each step, until reaching a terminal condition [27].

Chapter 3

Thesis objectives

The core purpose of this thesis is to design and evaluate an AI powered system capable of translating natural language security requests into structured OpenC2 commands executable via the Otypy framework. Modern cybersecurity operations face a dual challenge: enforcing consistent security policies across heterogeneous infrastructure, and translating high level security intent into vendor specific configurations. This work bridges these gaps by combining Large Language Model (LLM) capabilities with the OpenC2 standard, creating an intelligent intermediary between human operators and cyber defense actuators.

The proposed system leverages LangGraph to implement a multistage pipeline operating through distinct phases: a *preprocessing phase* that resolves domain names and enriches requests with infrastructure context via a RAG system; a *parsing phase* that extracts actions, targets, actuators, and arguments; and an *execution phase* that assembles the final command structure.

3.1 Translation objective

The primary objective is to enable seamless translation from natural language to executable OpenC2 commands through an agentic LLM workflow:

1. **Preprocessing:** The system resolves domain names to IP addresses and invokes the RAG service to augment requests with infrastructure context from the network’s Bill of Materials.
2. **Parsing:** The LLM extracts otypy components (action (*deny*, *allow*, ...), target (connections, networks, ...), actuator profile (*iptables*, *openstack* ...), and arguments (direction, duration ...)).
3. **Execution:** Extracted components are assembled into valid Otypy commands ready for dispatch.

The pipeline handles complex requests by decomposing them into atomic sub requests (e.g., a prompt that requires *block ports 22, 80, and 443* generates three rules) and engages in interactive clarification when information is missing or ambiguous.

3.2 Validation

The validation system evaluates the system’s ability to translate natural language into correct OpenC2 commands through a multi-layered approach that checks both semantic correctness and SLPF specification compliance.

3.2.1 Validation Mechanisms

1. **Semantic Validation:** Compares generated commands against expected outputs component-by-component. A single mismatch causes the full the command to fail and also the presence of extra components lead to failure, to evaluate hallucination.
2. **Multi-Command Validation:** For requests generating multiple commands it’s required to match each output to its expected result.

3.2.2 Evaluation Metrics

The system measures performance across three dimensions:

- **Semantic Accuracy:** Overall score (100% threshold for passing), per-component precision (action, target, actuator, arguments), and critical component failure tracking.
- **Operational Efficiency:** Clarification rate (if it’s required human intervention and how many times), execution latency, and token consumption (input, output, reasoning).
- **Model Comparison:** Pass rate, error rate, and efficiency metrics across different LLM backends (local Ollama models and cloud models via OpenRouter).

3.2.3 Test Execution

Each test case runs through the LangGraph application with a set of predefined requests that cover varying levels of technical expertise from precise, domain specific terminology to informal phrasing with less specific terms. Additionally, requests in languages other than English are included to assess multilingual robustness. The system captures generated commands, validates semantic correctness, records performance metrics, and aggregates results into comprehensive reports comparing model performance.

Chapter 4

Related Works

This chapter surveys existing approaches to translating natural language into security policies, positioning the proposed system within this landscape. The automation of security policy generation from natural language has attracted significant research attention across multiple domains including access control policy generation, firewall configuration, and intrusion detection, with existing approaches targeting different output formats and employing varying validation mechanisms. Recent work has also explored whether parameter-efficient fine-tuning of open-source LLMs can improve rule generation accuracy beyond what prompt engineering alone achieves.

4.1 Access Control Policy Generation

Research on translating natural language into access control policies demonstrates the viability of LLM-based approaches for security policy automation. Lawal et al. [28] show that schema-constrained prompting representing policy entities as typed Python dataclass definitions enables LLMs to extract subjects, actions, and resources with high accuracy (99.9% with few-shot examples). Their system validates extracted policies against Next Generation Access Control (NGAC) specifications using a graph database, confirming that structured output schemas significantly improve reliability over unconstrained generation. Martinelli et al. [29] propose extending this approach to XACML and Rego formats for cloud-native environments, though their work remains at the proposal stage.

4.2 Firewall and Network Security Automation

By moving to the firewall context the work done by Taghiyev et al. [30] present the closest existing work to this thesis: a framework for translating natural language into firewall configurations. The system separates policy intent from device-specific syntax through a structured intermediate representation that captures rule components (source, destination, ports, protocol, action) and contextual metadata. Two LLM-based components work in sequence: a *Resolver Agent* maps ambiguous names (e.g., “HR laptops”) to concrete network objects, and an *IR Builder Agent* constructs the

structured rule from resolved entities. Multiple validation layers check correctness before output, including a Safety Gate that blocks overly permissive rules such as unrestricted traffic permissions. The system achieves approximately 85% accuracy on synthetic test cases. Key limitations include: while the IR representation is a high level enough to compile for multiple vendors, at the moment single-vendor output (Palo Alto) is implemented, no iterative refinement for ambiguous requests, and infrastructure context must be manually specified via configuration files.

Gosálvez-White et al. [31] address a complementary scenario with MEDIotWALL, a distributed firewall solution designed for Internet of Medical Things (IoMT) environments. The system adopts a two-tier architecture: low-cost embedded security devices (in this case, Raspberry Pi boards) are deployed inline as per-device firewalls running Suricata IDS and nftables, while a centralized manager coordinates monitoring, analysis, and rule deployment. When Suricata detects suspicious traffic, the system uses an LLM (in the study implemented via OpenRouter API) to analyze the alert and automatically propose nftables rules, which an administrator must review and approve before deployment. The approach achieves detection-to-rule latency between 1.5 and 38 seconds and was validated against TCP SYN floods, port scans, and OS fingerprinting attacks on real medical devices.

4.3 LLM Fine-Tuning for Security Rule Generation

A parallel line of research investigates if fine-tuning can overcome the limitations of prompt engineering for security rule generation. Louro et al. [32] provide a systematic baseline evaluation of six LLMs of different size (from a ChatGPT 3.5 to small local model as Mistral 7B), on their ability to generate iptables firewall and Snort IDS rules from natural language. They design three prompt types of increasing specificity: attack name only, technical description, and explicit instruction. Without fine-tuning, ChatGPT 3.5 achieves the highest accuracy at just 42%, while open-source models score near 0%. Applying Low-Rank Adaptation (LoRA¹) fine-tuning to Mistral 7B raises accuracy to 89%, with the model learning to generalize across prompt structures. Three training strategies are compared: prompt robustness (89%), multi-hop semantic linking from attack names to rules (61%), and syntax extrapolation from generic to attack-specific rules (79%). These results demonstrate that prompt engineering alone is insufficient for reliable security rule generation and that targeted fine-tuning can yield substantial improvements even on small datasets of approximately 100 examples.

Zhang et al. [34] extend this investigation with a comparative study of Parameter-Efficient Fine-Tuning (PEFT)² techniques for nftables and Snort rule generation.

¹LoRA [33] is a parameter-efficient fine-tuning technique that freezes the pre-trained model weights and injects small trainable low-rank matrices into each layer, drastically reducing the number of trainable parameters while preserving model performance.

²PEFT [35] refers to a family of techniques for adapting large pre-trained models to downstream tasks by updating only a small subset of parameters. Common approaches include adapter layers, prompt tuning, and LoRA.

They evaluate prefix tuning, prompt tuning, and LoRA on Mistral-7B, finding that LoRA significantly outperforms the alternatives (63% versus 22% for prompt tuning and 0% for prefix tuning on Snort). Applying LoRA across three 7B-parameter models, Qwen2-7B emerges as the best performer with 84% accuracy on Snort and 99% on nftables, notably outperforming even the larger Qwen3-14B model, which tends to generate verbose explanations rather than concise rules. This finding suggests that task-specific architectural suitability matters more than model scale for structured security rule generation. Both studies converge on LoRA as the most effective PEFT method, though they remain limited to vendor-specific output formats and do not address ambiguity resolution or infrastructure context.

4.4 Research Gap and Positioning

Comparing the surveyed approaches reveals common limitations and complementary gaps. Lawal et al. generate NGAC policies using chain-of-thought prompting with example-based context and Neo4j for validation, while Taghiyev et al. produce typed intermediate representations through schema-bound generation with configuration file context and multi-layer validation. Gosálvez-White et al. demonstrate LLM-driven rule generation from IDS alerts with mandatory human approval, but target a single firewall syntax (nftables) and rely purely on an external API for inference. The fine-tuning approaches of Louro et al. and Zhang et al. show that LoRA can significantly improve rule generation accuracy, yet both produce vendor-specific rules (iptables, Snort, nftables) and assume that the user provides an unambiguous, specified request. Across all five systems, none combines vendor-agnostic output, interactive clarification for ambiguous requests, and automatic infrastructure context enrichment. This thesis fills these gaps through several novel contributions:

- **Vendor-agnostic output:** All analyzed systems target either access control formats (NGAC, XACML) or vendor-specific firewall and IDS syntaxes (Palo Alto, iptables, Snort, nftables). This thesis adopts OpenC2, a standardized, vendor-agnostic command-and-control abstraction that decouples policy intent from device-specific implementation.
- **Prompt engineering with agentic workflows:** Rather than fine-tuning dedicated models for a fixed output format, this thesis employs structured prompt engineering within a multi-node agentic graph. This approach preserves flexibility across LLM providers and avoids the cost and rigidity of model-specific training, while the fine-tuning literature confirms that structured guidance, whether through training data or prompt design, is essential for reliable rule generation.
- **Automatic infrastructure enrichment:** All prior systems require manual context specification, whether through configuration files, pre-defined network objects, or explicit prompt details. This thesis automatically enriches requests

via Graph RAG from Extended Bill of Materials (XBOM) data, resolving actuator locations and network topology without operator intervention.

- **Interactive clarification:** Existing systems either fail silently when facing ambiguity, make assumptions that may not align with user intent, or, in the case of MEDIoT WALL, delegate all decisions to a post-hoc human review. This thesis implements fine-grained clarification that identifies specific ambiguous components and solicits targeted user input before generation.

In summary, this thesis presents a system combining agentic LLM workflows with Graph RAG-based infrastructure enrichment for generating vendor-agnostic OpenC2 security commands, addressing gaps that no single prior work has collectively tackled.

Chapter 5

Agentic Command Generation Framework

The current architecture emerged through iteration. Early attempts with a single-pass LLM approach, where one prompt attempted to extract all command components simultaneously, consistently led the model to confuse the multiple fields' possible values and the correlations between them, a behavior predominantly observed in small and medium-sized models but less common in larger ones. This forced the design toward the multi-stage pipeline described below, where action parsing occurs first and informs subsequent component extraction. Similarly, early prototypes that asked the LLM to produce fully qualified Python expressions for each component suffered from frequent enumeration and namespace errors in the generated code. This motivated the adoption of JSON dictionaries with predefined structures, where the LLM fills in only simple values and a deterministic builder function handles the translation to valid Otupy objects, as detailed in Section 5.2.2.

5.1 Technology Stack

The framework is not built solely around LangGraph and LangChain, already introduced in the Background, but also on another set of Python libraries that together have helped to implement the agentic design patterns. Table 5.1 summarizes the key libraries and their roles in the system.

Table 5.1: Key Python libraries and their roles in the framework.

Library	Role
Pydantic	Data validation for both the workflow state model and structured LLM output schemas
asyncio	For guaranteeing concurrent calls to LLM API for latency optimization
aioconsole	Async-compatible user input for human-in-the-loop clarification without blocking the event loop
dnspython	Permits DNS resolution to translate domain names into IP addresses during preprocessing

5.2 System Workflow

5.2.1 Workflow Overview

The translation pipeline is modeled as a directed graph with ten processing nodes organized into four phases: *preprocessing*, *orchestration*, *parsing*, and *execution*. The design follows a progressive refinement approach: each phase adds a layer of understanding to the user's request, transforming unstructured natural language into increasingly structured representations until a valid command is produced. Figure 5.1 shows the complete graph with both direct and conditional edges.

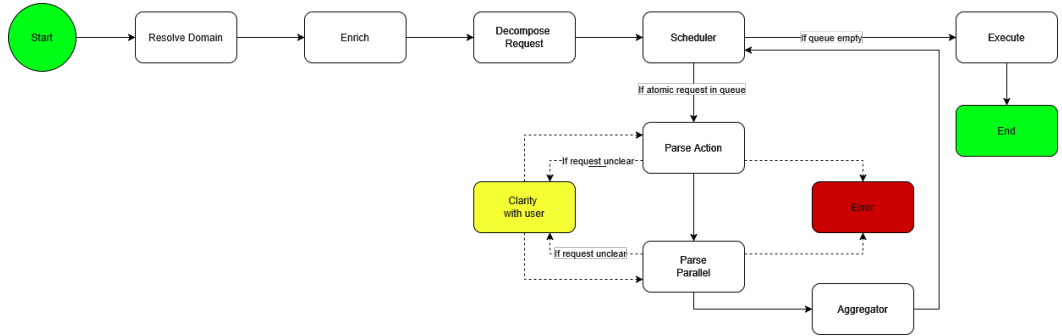


Figure 5.1: LangGraph workflow structure. Solid arrows indicate direct transitions; dashed arrows represent conditional routing.

The ten nodes are organized into four sequential phases, each adding a layer of structure to the raw user request:

- **Preprocessing:** Phase composed of two nodes: the first, *Resolve Domain*, extracts any domain names from the input and resolves them to IP addresses via DNS, ensuring that downstream parsing operates on explicit network addresses rather than symbolic names. Then the *Enrich Request* node forwards the input to the RAG service described in Chapter 6, which augments it with infrastructure context and returns an enriched version of the request ready for parsing.
- **Orchestration:** This group of nodes handles the decomposition and scheduling of sub-requests, starting from the *Decompose Request* node that uses an LLM to analyze the input and, when the request is too complex to map to a single command, splits it into atomic sub-requests each mapping to a single OpenC2 command. Then the *Scheduler* node drives a processing loop by selecting one sub-request at a time and activating the parsing phase. After each iteration, the last *Aggregator* node collects the result and either returns control to the scheduler or, once the queue is exhausted, advances to execution.
- **Parsing:** This phase is split in two macro nodes: the first is *Parse Action*, which have the only scope of identifying the action, since the chosen one determines which target types and argument structures are valid for all subsequent steps. With the action known, it is the turn of the *Parse Parallel* node, that dispatches

three concurrent LLM calls one each for: target, actuator/consumer, and arguments. Both the nodes implement a human-in-the-loop mechanism that when needed route the workflow to the *Clarify With User* node.

- **Execution & error management:** Once all sub-requests have been handled, the *Execute* node assembles the accumulated results into validated Otype command objects and terminates the graph. The *Error* node supports all phases by recording any parsing failures and forwarding them to the aggregator without halting the pipeline, ensuring that one failed sub-request does not prevent the remaining ones from completing.

5.2.1.1 Conditional Routing

The transitions between nodes are governed by conditional edges rather than fixed sequential paths: each edge delegates its routing decision to a function that inspects the shared state flags at runtime. Each parsing node can produce three outcomes: success, clarification needed, or error and the graph routes accordingly. When clarification is needed, the *Clarify With User* node pauses the workflow, collects the user’s response, and returns control to the exact node that originated the request.

5.2.1.2 Shared State Model

Central to the architecture is the State that acts as the single source of truth and persistent data store for the entire workflow. Every node receives the full state, reads the fields it needs, and returns an updated copy, making all internode communication explicit and type-checked. Its fields are organized into four logical groups: *Command Components*, which holds the parsed elements of the OC2 command; *Orchestration*, which manages the sub-request queue and execution state; *Clarification Flow*, which tracks not only if the clarification is required but also information as the calling node and which questions present to the user; and *Token Metrics*, which stores metrics for each LLM call.

5.2.2 LLM Response Schemas

Every LLM call in the framework uses structured output: responses are constrained to conform to a Pydantic schema via LangChain’s structured output mechanism. This approach eliminates manual JSON parsing and guarantees the response structure, ensuring that the application never needs to handle malformed or unexpected output formats.

The schema hierarchy is built on a base class that enables a dual-purpose response pattern: each LLM call returns either a successful parse or a clarification request within the same schema, differentiated by a single boolean flag. The base class also carries confidence and reasoning fields, which are therefore inherited by all derived schemas. Derived classes add only component-specific fields for the particular parsing task at hand, as illustrated in Figure 5.2.

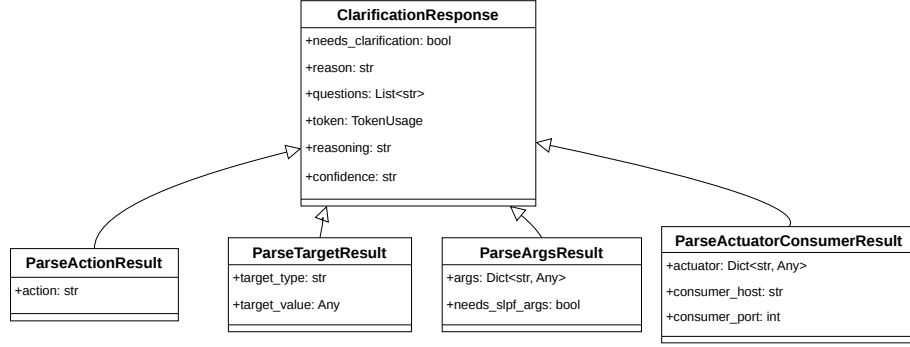


Figure 5.2: LLM response schema hierarchy showing the base class with shared fields (including confidence and reasoning) and derived classes with component-specific fields.

This design eliminates the need for a separate validation pass. The first LLM call itself determines whether it has sufficient information to produce a valid parse; when it does not, it signals ambiguity and supplies a list of questions for the user, which the workflow routes to the clarification node. This unified approach simplifies the control flow considerably, as every parsing node follows the same response-handling pattern regardless of the component being parsed.

The confidence level (high, medium, low) provides a self-assessment of the LLM’s certainty that can drive different behaviors in future iterations. The reasoning field records a summary of the chain of thought that led to the parse, enabling inspection and debugging after the fact without re-invoking the model. For arguments, an additional flag allows the LLM to signal whether the parsed values contain SLPF specific fields, which determines whether the assembler should produce standard or extended arguments.

As visible in the schemas 5.2, the LLM output for complex elements like target and arguments is a JSON dictionary rather than a fully qualified Python expression or a string. Concretely, this means the LLM returns simple string values such as *ingress* or *tcp* inside a predefined key structure, rather than attempting to produce complete Otypy calls like *slpf.Direction.ingress* or *oc2.L4Protocol.tcp*. This choice leads to two main benefits: first, the LLM no longer needs to resolve the correct module namespace for each enumeration member, which during early prototyping led to errors such as incomplete paths (*oc2.complete* instead of *oc2.ResponseType.complete*) and cross-module mis-attributions (*oc2.Direction.ingress* instead of the correct *slpf.Direction.ingress*). Second, because the mapping from simple values to Otypy objects is handled by a dedicated command assembler service (Section 5.3.5), any future changes to the Otypy API naming convention, require updating only the mapping dictionaries, not the LLM prompts that could be more costly to re-engineer and validate.

The transitions between nodes are mainly governed by conditional edges, which delegate the routing decision to the inspection of the state flags. Each parsing node can produce three outcomes: success (proceed to the next stage), clarification needed

(route to the user interaction node), or error (route to the error handler). After clarification, the workflow returns to the exact node that originated the request by consulting the calling node’s identity stored in the state, ensuring that no parsing step is skipped or repeated unnecessarily.

5.3 Execution Pipeline

5.3.1 Preprocessing

The preprocessing phase prepares the raw user input for LLM-based parsing through two enrichment steps. Both steps operate without directly involving the LLM, focusing instead on augmenting the input with concrete, factual information that makes the subsequent parsing more accurate.

5.3.1.1 Domain Resolution

The first node addresses a practical requirement of actual Otupty commands implementation: they operate on explicit network addresses, not domain names. When a user writes “block traffic to example.com”, the downstream parsing nodes need to know the corresponding IP address. To avoid an unnecessary LLM call, domain extraction uses regex pattern matching followed by DNS resolution. This solution solves the problem without involving the LLM but is still not perfect, in fact, if we consider the context of a more discursive user input, like “I want to block traffic to example website”, the regex-based domain extraction would fail to identify “example website” as a domain, and even adding an LLM call to capture the request still presents the need to correctly identify the first level domain name (e.g. like .com, .it etc...). For this reason, given the imperfection of both solutions I decided to implement the one that introduces the less amount of computational overhead and risk of less hallucinations. Another critical limitation of this node is the nature of modern dynamic IP management, where a single domain can resolve to multiple IP addresses that change over time. The current implementation appends all resolved IPs to the user input, that should resolve in the application of a different rule for each address. Future iterations could involve selecting the most relevant IP based on additional context to limit the number of generated rules, but still only a temporary solution given the dynamic nature of modern IP management. In a real-world scenario, this node would likely need to be replaced with a more robust service that maintains an up-to-date mapping of domain names to IP addresses, possibly through integration with the organization’s DNS infrastructure.

5.3.1.2 RAG Enrichment

The second preprocessing step augments the user input with infrastructure context retrieved from the XBOM knowledge base. This implements the RAG integration described in Chapter 6, where a separate service internally uses an LLM to combine the

original request with relevant contextual information about the network environment and returns an enriched and reorganized version of the input.

This implementation it is one of the most important aspect of the framework as give the possibility even to non-expert users to leverage the knowledge base without needing to know how to navigate it, how to interpret it or even know from the start technical details such as IPs, ports and used protocols for each service, in fact the user can simply write a request like *block traffic to the X service*, and the RAG service will automatically enrich it with details about the IP address of the server that is managing the service, the used protocol and the corresponding port and more, so that the parsing phase can directly extract the needed information to produce a correct command.

An important choice here is the implementation of a graceful degradation: if the RAG service is unavailable or returns empty results, the workflow continues with the original input. This ensures that the system remains operational even when the knowledge base infrastructure is offline. The trade-off is that commands generated without enrichment could require heavy human intervention involving manual research of information from the XBOM documentation to produce the correct result if the original input is too generics.

5.3.2 Orchestration

5.3.2.1 Request Decomposition

In practice, security requests often involve more complexity than a single OpenC2 command can capture. A request like *block HTTP and FTP traffic to 10.0.0.1* actually requires two separate commands, one for each protocol. The decomposition node was defined to handle these kinds of situations, using an LLM to analyze the complexity of the incoming request and, when necessary, split it into atomic sub-requests, each of which maps to exactly one OpenC2 command. The decomposition follows a set of rules embedded in the prompt. For example, one command is generated per service or port combination, so a request mentioning both HTTP and FTP yields two sub-requests. When multiple dimensions are involved, such as two services targeting two hosts, the LLM produces the Cartesian product, resulting in $2 \times 2 = 4$ sub-requests. Requests that are already atomic simply pass through as a single-element list.

Figure 5.3 illustrates a concrete decomposition scenario where a two-service request is split and processed through the scheduler loop. Even for this operation it is present a fallback mechanism: if the LLM fails to decompose the request for any reason, the original input is treated as a single sub-request, allowing the workflow to continue. For complex requests, this fallback approach will likely trigger clarification requests and require human intervention to determine how to proceed, but it prevents complete system failure due to decomposition errors.

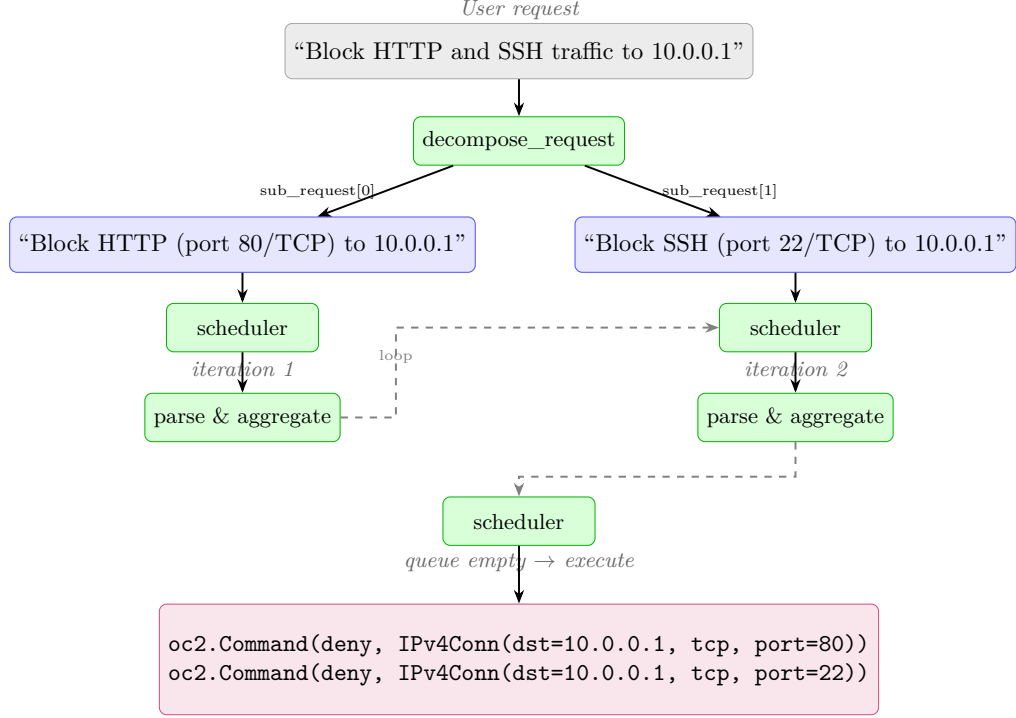


Figure 5.3: Request decomposition example. A compound request mentioning two services is split into two atomic sub-requests, each processed independently through the scheduler–aggregator loop to produce one OpenC2 command per service.

5.3.2.2 Scheduler Aggregator Loop

Once decomposition has produced a queue of sub-requests, the scheduler and aggregator create a processing loop that handles them one at a time. The scheduler pops the next request from the queue, clears the command fields, and activates the parsing phase. After parsing completes whether successfully or with an error, the aggregator collects the result into a running list. The *check_queue* routing function then determines whether more sub-requests remain. If so, the scheduler is invoked again; otherwise, the workflow proceeds to the execution phase.

Even in this case it is present a resilient management of the errors: a parsing failure for one sub-request does not invalidate the others. Failed sub-requests are recorded with an error status, and the scheduler continues processing the remaining items. This means that if a user submits a request that decomposes into five sub-requests and one of them is ambiguous enough to cause a parsing error or fail for other reasons, the system still produces four valid commands rather than failing entirely.

5.3.3 Parsing

The parsing phase is responsible for extracting four semantic components from each sub-request: the action (what operation to perform), the target (what to apply it to), the coupled actuator and consumer (which device should execute it and where), and the arguments (additional parameters such as traffic direction).

5.3.3.1 Component Configuration Pattern

Parsing four different components requires four structurally identical tasks: invoke an LLM with a component-specific prompt, check whether the response requests clarification, extract the relevant field from the result, and store it in the state. Rather than duplicating this logic four times, a configuration-driven approach is used: each component is described by a configuration object that encapsulates its prompt, output schema, and field-extraction logic, while a single generic parsing function operates on any configuration. This eliminates code duplication and keeps the system easily extensible adding a new OpenC2 component requires only a new configuration entry, not a new parsing function.

Since target and argument prompts vary depending on the already-parsed action, they are resolved at runtime through action-specific dictionaries. For example, as already indicated in Section 2.1.4, *query* actions restrict the target to *features* only, while *allow* and *deny* actions accept network connection targets, and *delete* actions expect the ID number of the rule to delete. Selecting the prompt dynamically ensures each parsing step receives only the relevant options and constraints, reducing ambiguity. A fallback to the allow/deny prompt covers any unrecognized action type.

5.3.3.2 Sequential Action Parsing

As I’ve already mentioned, the action component is required to determine the valid values for the other three components, so it is parsed first in a dedicated node. The logic in the node is quite simple, without any unusual implementation choice: the LLM is invoked with an action-specific prompt and the response is checked for clarification. If successful, the parsed action is stored in the state and the workflow proceeds to the next node; if clarification is needed, control is routed to the user interaction node.

5.3.3.3 Parallel Component Parsing

After the action has been identified, the remaining three components target, actuator/consumer, and arguments can be parsed concurrently. Since all three depend only on the action and not on each other, they are dispatched as independent parallel LLM calls. This reduces the number of sequential LLM round-trips from four to two, cutting latency to the duration of the longest of the three calls rather than their sum. An exception applies to *query* actions, whose target and arguments are fixed by the specification, so only one LLM call is needed instead of three. Listing ?? shows the implementation.

When multiple components need clarification, their questions are collected and presented to the user in a single interaction rather than as separate interruptions, prefixed with labels to provide information about which component is being clarified. After the user responds, only the components that needed clarification are reparsed, while successfully parsed components retain their values. One design exception to the one-task-per-call rule is the actuator and consumer pair, which are extracted in a

single LLM call because they are semantically coupled: the consumer endpoint often determines the actuator platform, and together they form a small, well-defined field set that a single prompt can resolve reliably.

5.3.3.4 Prompt Engineering Strategy

The prompts used throughout the parsing phase, totaling over 1,000 lines, encode detailed domain knowledge about the OpenC2/SLPF specification, its interaction with the Otupy framework and a series of examples. Rather than reproducing them in full, this section highlights the key design principles that guide their construction.

5.3.3.4.1 Dual-purpose response format: Every component-specific prompt instructs the LLM to return either a parsed result or a clarification request within the same JSON structure, unified by the *needs_clarification* flag. This avoids the need for separate validation passes and keeps the control flow uniform across all parsing nodes.

5.3.3.4.2 Action-dependent prompts: As described in Section 2.1.4, the SLPF profile restricts which targets and arguments are valid for each action (see Table 2.1). To encode this logic, the framework maintains dedicated prompt dictionaries that map each action to a specialized prompt. For example, the allow/deny prompt describes network connection targets and traffic direction rules, while the delete prompt focuses exclusively on rule number extraction. This means the LLM receives only the relevant options and constraints for the current action, reducing ambiguity and improving parsing accuracy.

5.3.3.4.3 Direction & Consumer inference rules: One of the more nuanced aspects of the prompts is the logic for inferring traffic direction, which follows a priority-based order. Explicit direction keywords in the request (such as “ingress” or “egress”) always take precedence. And same when synonyms are used. When no explicit direction is given, the system infers it from the addressing pattern: if only an internal IP specified as destination with no source, the direction defaults to ingress (traffic arriving at the target); if only a source is specified, it defaults to egress (traffic leaving the source); and if both are present, the direction defaults to egress (traffic flowing from source toward destination). This logic is also kept when the 2 specified IPs are both internal, plus if multiple consumer endpoints are present the one directly connected with the source IP (or otherwise the one nearest to it) is the one to apply the rule to, with the idea of block the soon as possible the malicious traffic before it can propagate through the network. These rules are explicitly encoded in the prompts to guide the LLM’s reasoning and ensure consistent interpretation of direction across different requests.

5.3.3.4.4 Few-shot Learning: Each prompt includes between six and ten examples covering both common and edge cases, demonstrating successful parsing

as well as clarification scenarios. These examples ground the LLM’s behavior in concrete patterns rather than relying solely on abstract rules, which help to improve consistency across different model providers and sizes.

5.3.4 Human-in-the-Loop Clarification

The clarification mechanism implements a reusable interrupt pattern: any parsing node can pause the workflow to request additional information from the user, and processing resumes at the exact point where it was interrupted. This approach ensures that the system never guesses when it lacks sufficient information; instead, it asks a targeted question and uses the answer to produce a more accurate result.

When a parsing node determines it cannot proceed without more information, it records a clarification trigger in the state along with the name of the calling node (used as a return address), a human-readable reason for the ambiguity, and a list of specific questions for the user.

After the user responds, the node appends the question-answer exchange to the original input in a structured format, providing the LLM with full context when the parsing node re-executes and allowing it to incorporate the clarification without losing sight of the original request.

5.3.5 Command Assembly and Execution

5.3.5.1 Command Assembler

The command assembler is a dedicated service, separate from the execution node, whose sole responsibility is to translate the LLM’s JSON output into validated Otypy objects. It uses builder functions that map the simple string values produced by the LLM to their corresponding Otypy enumeration members through predefined dictionaries. Figure 5.4 illustrates this translation pipeline.

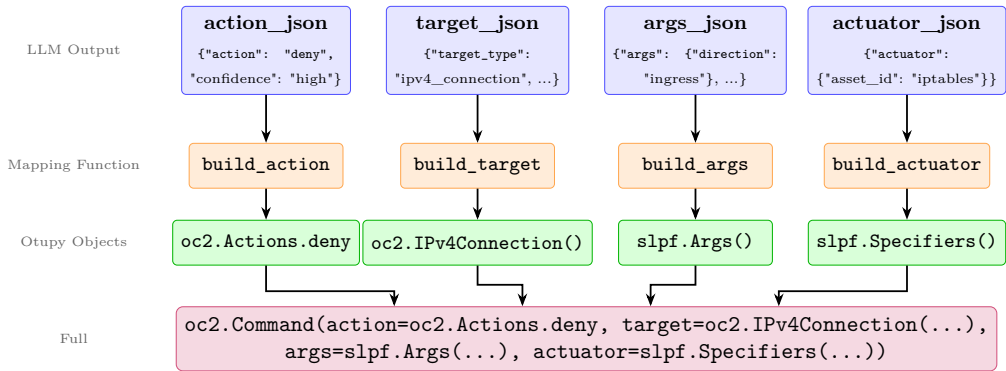


Figure 5.4: Command assembly pipeline

In addition to producing validated command objects, the assembler also generates an evaluation-compatible string representation from the same JSON inputs. This string form is used by the testing framework described in Chapter 7 for validation without needing to execute the command.

Centralizing the translation logic in the assembler has two advantages: first, the LLM no longer needs to resolve module namespaces or enumeration paths it only fills in simple string values; second, any future changes to the Otypy command structure require updating only the assembler’s mapping dictionaries, not the LLM prompts, which are more costly and time consuming to re-engineer and validate.

5.3.5.2 Execution Node

The execution node iterates over the accumulated results, invoking the assembler for each sub-request that was successfully parsed. Both a validated command object and its evaluation-compatible string representation are produced for every successful result. A deliberate fallback handles the case where the Otypy library rejects some values: the node still produces the evaluation string from the raw JSON so that the command is not silently lost, ensuring evaluation metrics can capture the output even when assembly validation fails

For this last node, it is important to note that the actual implementation is limited to printing the generated commands in the console, but in a real scenario this is where the integration with the enforcement infrastructure would happen, with the command sent to the corresponding consumer endpoint.

5.3.6 LLM Service Layer

Keeping the workflow model-agnostic required isolating all provider-specific logic in a dedicated LLM service layer. This layer provides a unified interface for invoking any supported model with structured output, retry logic, and token usage tracking none of these concerns appear in the main workflow nodes. A singleton pattern ensures that only one LLM instance is created throughout the application’s lifetime. Structured output is enforced by constraining every call to a given Pydantic schema, eliminating manual JSON parsing. Transient failures are handled by retry logic with exponential back-off (delays of 1, 2, and 4 seconds across up to three attempts), covering both validation errors and network issues. Model selection is governed entirely by environment variables, allowing switching between a local Ollama instance and a cloud provider without any code changes.

This implementation currently targets two provider backends a local Ollama endpoint and a cloud provider accessible via an OpenAI-compatible API and could in future iterations be extended to support additional proprietary interfaces with minimal configuration changes.

Chapter 6

Graph-based Knowledge Management System

6.1 Introduction

The work presented in this chapter was born as a necessity to make the command generation described in Chapter 5 useful even outside laboratory tests or very structured environments. Concretely, this is the system behind the *Enrich Request* node of the agentic workflow shown in Figure 5.1: during the preprocessing phase, the Enrich node issues an HTTP call to the service described here, which resolves abstract references in the user’s request to concrete infrastructure parameters before any LLM-based parsing begins. Its role is to get requests such as “block SSH traffic to the database server” as input, and then resolve the abstract reference (in this case “database server”) to a concrete IP address, identify the appropriate security actuator capable of enforcing the policy, and determine the network direction relative to that actuator. This chapter presents the *Graph-based Knowledge Management System*: a Retrieval-Augmented Generation (RAG) service that provides this contextual enrichment by combining a structured knowledge graph with keyword-based search over Bill of Materials (BoM) data.

The system parses CycloneDX [8] BoM files annotated with Otupy-specific property extensions [5] into an in-memory directed graph. Each node represents an infrastructure entity (cloud environment, pod, virtual machine, container, or security actuator), and edges encode typed relationships such as hosting hierarchies, control relationships, and network flows. A BM25¹ full-text index built over entity descriptions, in combination with a vector embedding index enables retrieval of relevant entities from natural language queries.

¹BM25 (Best Matching 25) is a probabilistic ranking function widely used in information retrieval. Given a query, it scores each document by weighting term frequency with diminishing returns and penalizing longer documents, producing a relevance-ranked list without requiring any training data or embeddings.

Design Evolution: From Dual-Index to Hybrid Retrieval

The retrieval architecture evolved through four distinct iterations, each addressing limitations of the previous approach.

Stage 1: Direct vectorization of raw BoM documents. The earliest prototype treated each entity’s JSON block in the BoM as a document chunk and embedded it directly into a vector store. This revealed three problems. First, CycloneDX encodes metadata as flat key-value pairs with long, path-like keys (e.g., `otupy:link:5:peer:kubernetes-networkpolicies:consumer:host`). These strings are structurally repetitive and for an embedding model they have the same weight of the value, drowning out the important features. Second, the security-relevant topology is spread across multiple entities: the actuator that protects a pod is listed under its parent cloud environment, not under the pod itself. Cosine similarity² cannot recognize such cross-entity relationships. Third, the dependency section uses opaque BOM reference UUIDs (`urn:cdx:46cc8b09-...`) instead of entity names, making inter-component links invisible to embedding-based retrieval. The result was poor recall on relational queries and context windows filled with irrelevant property strings.

Stage 2: Dual-index (ChromaDB + graph). Recognizing that the raw BoM structure was unsuitable for direct embedding, the next iteration introduced a knowledge graph to parse and restructure the BoM data into domain objects with explicit typed relationships. A ChromaDB³ vector store (using *nomic-embed-text* embeddings via Ollama) was combined with the knowledge graph, merging results through Reciprocal Rank Fusion [36]. While the graph solved the cross-entity traversal problem its function was limited only in case where the user ask directly for an entity name present in the graph while the ChromaDB component, while could give a result even with different terms, its search over all entity attributes, also retrieved loosely related entities. The model then spent excessive tokens reasoning about irrelevant infrastructure details, producing slower responses with lower accuracy and higher cost.

Stage 3: Graph retrieval with BM25 entity resolution. Dropping the separate ChromaDB component in favor of graph-only context retrieval resolved the noise problem. The graph remains the primary retrieval mechanism: once the target entity is identified, topology traversal provides *precise, focused context* only the directly relevant entity, its protection chain, and its actuator endpoints. To overcome the requirement of exact name matches, a BM25 index was built over entity descriptions to provide a lexical retrieval branch alongside the graph. This hybrid approach significantly improved recall for informal queries while maintaining precision. However, BM25’s reliance on exact term overlap still led to missed matches

²Cosine similarity measures how similar two vectors are by computing the cosine of the angle between them. A value of 1 means the vectors are identical in direction (highly similar), 0 means they are unrelated, and -1 means they are opposite.

³ChromaDB (<https://www.trychroma.com>) is an open-source embedding database designed for storing and querying vector embeddings with metadata filtering.

when users used different terminology than the BoM descriptions, and it struggled with short queries that lacked distinctive keywords.

Stage 4: Graph retrieval with hybrid BM25 + semantic entity resolution (current). To address the recall limitation of BM25, a local sentence embedding model (*paraphrase-multilingual-MiniLM-L12-v2* from the *sentence-transformers* library) was reintroduced alongside BM25, with results fused via Reciprocal Rank Fusion. As in Stage 3, the graph traversal remains the main body of the retrieval process; BM25 and semantic search together form only the entity resolution layer that identifies the graph entry point. This model, unlike the Stage 2 implementation with ChromaDB + nomic-embed-text, runs entirely on CPU without the need to call any external API or Ollama service.

Another key difference from Stage 2 is that vector search operates only over the curated entity descriptions and name not over raw BoM chunks which eliminates the property-noise problem that made previous vector searches ineffective while keeping the recall advantage.

6.2 Technology Stack

This component reuses several libraries introduced in Chapter 5 LangChain for LLM abstraction, Pydantic for data validation, and *asyncio* for concurrent execution. Table 6.1 lists the additional libraries specific to the knowledge management system.

Table 6.1: Additional Python libraries used in the knowledge management system.

Library	Role
NetworkX	In-memory directed graph for infrastructure topology modeling; provides graph traversal, shortest-path algorithms, and ego-graph extraction
rank-bm25	BM25Okapi lexical scoring over entity descriptions; one branch of the hybrid retrieval pipeline alongside semantic similarity
sentence-transformers	Local multilingual sentence embedding model (<i>paraphrase-multilingual-MiniLM-L12-v2</i>) for semantic similarity search; combined with BM25 via Reciprocal Rank Fusion
FastAPI	REST API framework with automatic OpenAPI documentation, dependency injection, and async request handling
pickle	Binary serialization of the graph and embedding matrix to disk for persistence across service restarts

Among these, the graph library and the embedding model deserve particular attention, as they directly affect retrieval quality and deployment constraints.

NetworkX was chosen over dedicated graph databases like Neo4j because the infrastructure graphs derived from test BoM files are modest in size (typically tens to low hundreds of nodes) and fit comfortably in memory. An in-memory graph avoids the operational overhead of a separate database process while providing sub-millisecond traversal times for the multi-hop queries required by actuator discovery. A dedicated graph database would also introduce another process that needs to be

deployed, configured, and maintained, adding complexity without a clear benefit given the expected graph size and query patterns.

For the semantic branch of the hybrid retrieval pipeline, the *paraphrase-multilingual-MiniLM-L12-v2* [37] model from the **sentence-transformers** library is loaded from disk at startup and encodes entity descriptions into a dense embedding matrix for similarity search. Running locally on the CPU eliminates any external API dependency or hardware vendor lock-in and avoids transmitting infrastructure topology data to remote services. The embedding matrix is persisted alongside the graph so that subsequent service restarts do not require re-encoding all entities.

This particular model was selected for its multilingual capability (supporting 50+ languages) and strong paraphrase detection, which is particularly suited to entity resolution where operators may describe the same infrastructure component using varied terminology and in language different from the one used in the official documentation. With 117M parameters and 12 transformer layers, it offers richer representations than smaller alternatives while still achieving sub-second encoding on CPU for corpora of hundreds of entities, meeting the startup latency requirement without dedicated GPU hardware or cloud dependencies.

FastAPI provides the REST interface, with automatic request validation via Pydantic models, dependency injection for enforcing preconditions (e.g., ensuring the BoM is loaded before queries are accepted), and native async support for concurrent operations and future scalability when the system needs to handle multiple enrichment requests in parallel.

6.3 System Design and Data Model

6.3.1 Project Structure

The system is organized into five modules, each with a clearly delimited responsibility, as shown in Figure 6.1.

From the outside in: the *router* module is the external interface, exposing HTTP endpoints for loading a BoM file, issuing RAG queries, and health monitoring. It delegates all logic, error handling, and state management to the *core* and *services* modules. The *schemas* module defines the data shapes used throughout the system: domain objects, API request formats, and API response structures. The *utils* module provides shared helpers for token counting and text parsing.

The *services* module is where the system’s actual behavior is implemented. Six components each own a distinct step of the pipeline:

- **BOMParser** reads a CycloneDX BoM file and converts it into structured domain objects that the rest of the system can reason over.
- **GraphStore** takes those objects and builds a directed knowledge graph that encodes hosting, control, and dependency relationships between infrastructure

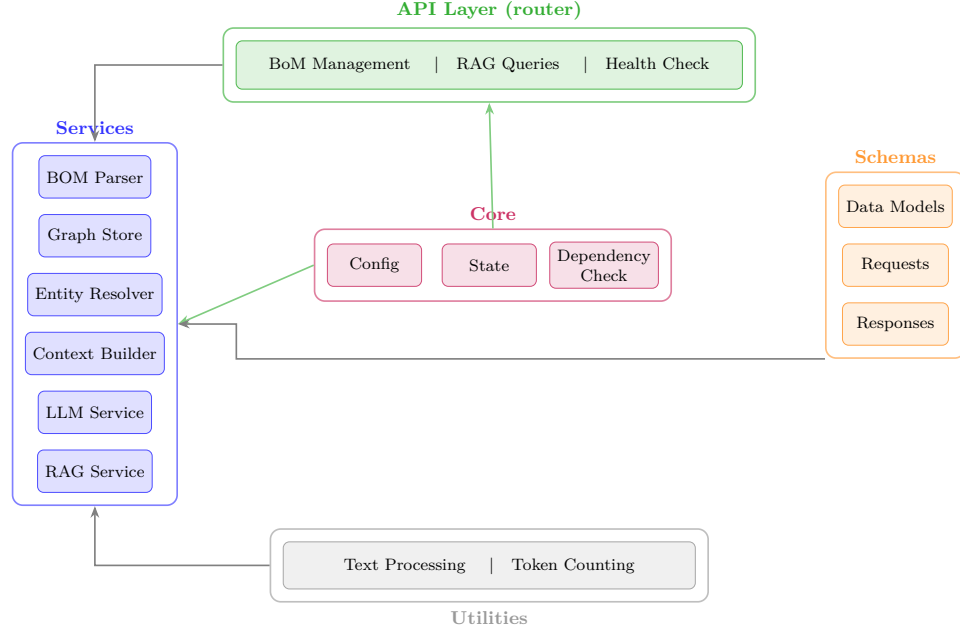


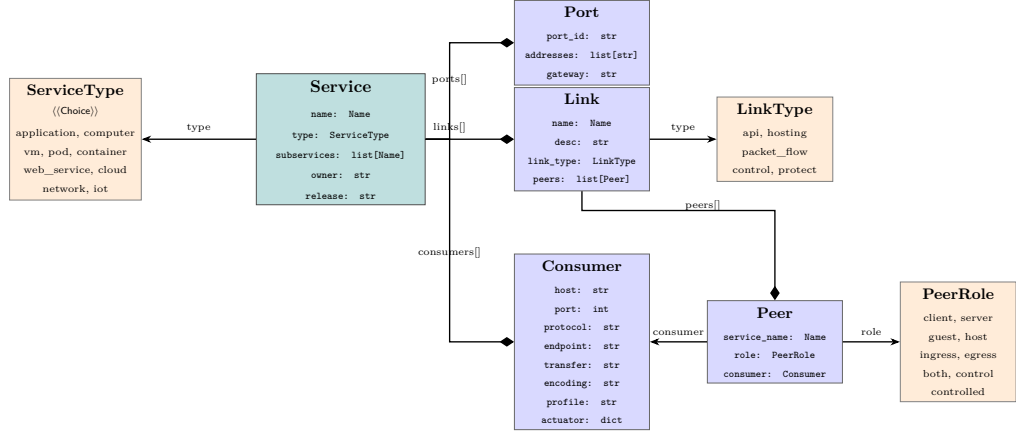
Figure 6.1: Modules structure of the knowledge management system. The API module exposes HTTP endpoints and delegates to the services module, which cover the functionality. Cross-cutting concerns and data definitions are separated into dedicated modules.

entities. It also manages the search indices both lexical and semantic used to retrieve relevant entities from natural language queries.

- **EntityResolver** bridges user language and graph content: given a free-text query, it identifies which graph nodes the user is referring to, even when names are partial or informal.
- **ContextBuilder** selects the most relevant nodes for a given query, scoring each candidate based on how directly it relates to the identified entities and its distance in the graph, then packages the result into a compact, structured context for the LLM.
- **LLMService** handles the language model calls, abstracting over different backends (local or remote) and tracking token usage.
- **RAGService** is the top-level orchestrator: it coordinates all the above components in sequence to either answer a security question or enrich a security request with infrastructure context.

6.3.2 CTXD Data Model

The system’s internal data model present in the Schemas module, Figure 6.1, mirrors the data model defined by the Otupy Context Discovery (x-ctxd) actuator profile [5]. Adopting the same classes ensures that BoM data can be parsed into domain objects without lossy translation. Figure 6.2 shows the resulting classes and their relationships.



Composition (◆): owned collection Association (→): reference

Figure 6.2: CTXD domain model. **Service** is the central entity; **Link** and **Peer** capture inter-service relationships; **Consumer** identifies the responsible OpenC2 actuator endpoint. **Port** is an implementation extension.

The central entity is *Service*, representing any infrastructure resource. Services are connected through typed *Link* objects whose *Peer* endpoints may reference a *Consumer*: the OpenC2 actuator reachable at that point. The *Consumer*’s *profile* field is the key discriminator for the system: an SLPF profile marks an actuator that can enforce firewall rules, while an x-ctxd profile marks a topology-discovery-only endpoint. *Port* is an implementation-level extension that consolidates IP addresses and gateway information to simplify address resolution during query processing.

6.3.3 Component Interaction

Figure 6.3 shows the data flow through the system’s components, from BoM loading through query processing.

The system operates in two phases. During *preprocessing*, the *BOMParser* transforms the CycloneDX JSON into domain objects, which the *GraphStore* converts into a directed graph with a hybrid search index (BM25 lexical and semantic embeddings) over entity descriptions. During *query processing*, the *EntityResolver* identifies relevant entities, the *ContextBuilder* assembles token-budgeted context, and the *RAGService* orchestrates the LLM call to produce the enriched response.

6.4 BOM Parsing and Graph Construction

This section describes everything that occurs before any user query: loading infrastructure data from a CycloneDX BoM file and building the knowledge graph that enables contextual retrieval.

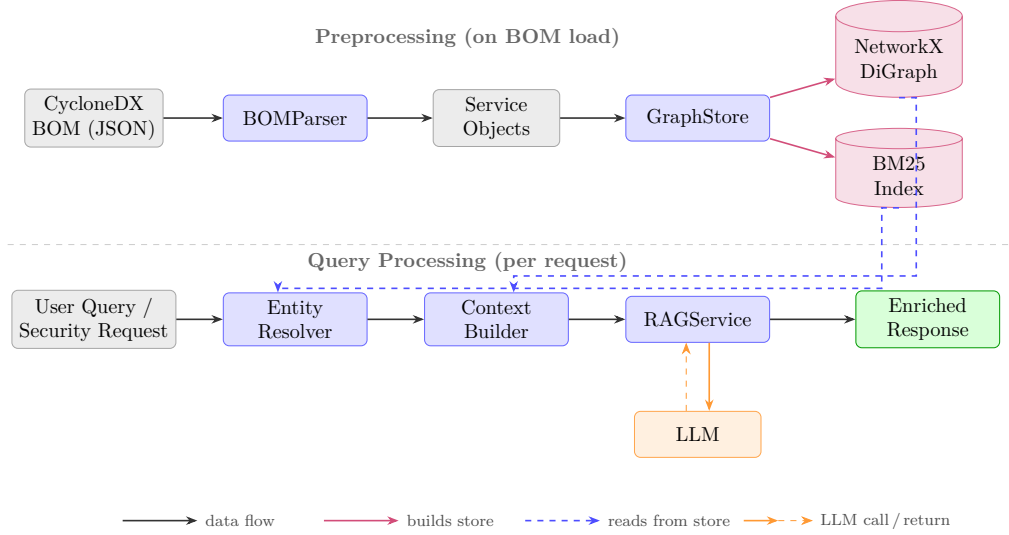


Figure 6.3: Component interaction diagram. Solid black arrows show the main data flow through the pipeline; purple arrows indicate the build phase where *GraphStore* populates the search indices; blue dashed arrows show query-time read access from the indices to the components that use them; orange arrows (solid: call, dashed: return) show the *RAGService*–LLM interaction.

6.4.1 CycloneDX BOM Parsing

Infrastructure topology is encoded in CycloneDX Bill of Materials files with Otupy-specific property extensions. The *BOMParser* class transforms this JSON representation into the domain model objects described in the previous section.

Entity metadata is encoded as flat key-value properties following naming conventions that embed structure. For this project, the following property patterns are the critical ones needed for graph construction and actuator discovery:

- *otupy:type* identifies the service/device type (pod, container, cloud, etc.)
- *otupy:link:{id}:type* and *otupy:link:{id}:desc* define relationship between two entities⁴
- *otupy:link:{id}:peer:service-name* names the connected service
- *otupy:link:{id}:peer:{name}:consumer:** discovers actuator endpoints
- *otupy:*.port:{id}:address* extracts IP addresses from network interfaces

The parser uses regex patterns to extract structured data from flat property keys: port identifiers and attribute names are grouped from *PORT_PATTERN* matches, link definitions are traversed to discover peer relationships, and consumer endpoints are assembled from typed property prefixes.

⁴The *type* field is a machine-readable enumerated value that classifies the nature of the relationship (e.g., *hosting* or *control*), while the *desc* field provides a free-text, human-readable description of the same relationship (e.g., “Kubernetes hosted on kube0”).

The parser iterates through both the *services* and *components* sections of the CycloneDX document, building a flat property dictionary for each entity. Regex patterns extract structured data from the property keys: *PORT_PATTERN* groups port identifiers and attribute names, while link parsing discovers peer relationships and their associated consumer endpoints. Owner information is extracted from type-specific paths (*otupy:cloud:owner*, *otupy:pod:namespace*) to enable organizational context.

6.4.1.1 Parsing Result

The output of the parser is a flat list of *Service* objects, one per entity in the BOM. The transformation is substantial: in the raw CycloneDX document, all information is encoded as a long sequence of flat key-value properties with *otupy:-namespaced* hierarchical keys. A single relationship, for example, requires five or more separate entries such as:

```
{ "name": "otupy:link:0:peer:kube0:consumer:host",    "value": "kube0"  }
{ "name": "otupy:link:0:peer:kube0:consumer:port",    "value": "8080"   }
{ "name": "otupy:link:0:peer:kube0:consumer:profile", "value": "x-ctxd" }
```

After parsing, the same data is a directly accessible, typed object:

```
Consumer(host="kube0", port=8080, profile="x-ctxd", actuator_id="ctxd-kube0")|
```

A similar transformation occurs also for the general representation of each service.

6.4.2 Knowledge Graph Construction

The *GraphStore* class converts parsed domain objects into a NetworkX directed graph. The construction proceeds in five phases:

- **Phase 1:** Adds a node for each service with metadata attributes: service type, description, IP addresses, and owner. A bidirectional mapping between entity names and BOM references enables dependency resolution.
- **Phase 2:** Creates typed directed edges from link definitions. The edge direction is inferred from the combination of link type and peer role: a *hosting* link points from the container to the hosted service, a *control* link points from the controller to the controlled entity, and so on. Consumer actuators are added as separate graph nodes connected to their owner service by a dedicated edge type.
- **Phase 3:** Processes the CycloneDX *dependencies* array, adding dependencies edges between services referenced by their BOM reference identifiers.
- **Phase 4:** Builds the BM25 full text index over entity descriptions for lexical retrieval, described in the next subsection.

- **Phase 5:** Encodes each entity’s combined name and description into a dense vector using the *paraphrase-multilingual-MiniLM-L12-v2* sentence embedding model, building the matrix used for semantic similarity search. The resulting embedding matrix is persisted to disk alongside the graph so that if the service is restarted with the same BoM file, it can load the graph and embedding matrix without re-parsing or re-encoding.

6.4.3 Hybrid Search Index

The hybrid index enables entity discovery when users refer to infrastructure components by informal descriptions rather than exact names. It combines two complementary retrieval methods lexical BM25 and dense semantic embeddings whose ranked lists are fused via Reciprocal Rank Fusion to produce a single ranked result.

6.4.3.1 BM25 Lexical Index

Each document in the corpus combines tokenized description terms with name tokens, filtering stopwords and short tokens to retain only meaningful terms. The resulting BM25 index maps queries to entity names via a parallel name list, enabling score-ranked retrieval. The BM25 index rebuilds in milliseconds when the BoM is reloaded, as it requires only tokenization and inverse document frequency computation.

6.4.3.2 Semantic Embedding Index

Each entity is encoded as a combined “*name: description*” string so the embedding captures both the identifier and its semantics. Because embeddings are normalized at construction time, retrieval reduces to a matrix-vector dot product a fast operation even for hundreds of entities. The resulting embedding matrix is persisted to disk alongside the graph so that service restarts restore it without re-encoding.

6.4.3.3 Hybrid Fusion via Reciprocal Rank Fusion

With 2 different rankings methods in use, to avoid flooding the LLM with too many entities and even double entities, the necessity of a fusion method arises. The chosen method is Reciprocal Rank Fusion (RRF) [36], a simple yet effective technique that combines multiple ranked lists by summing the reciprocal of their ranks, weighted by method-specific coefficients.

The defined RRF implementation assigns each entity a score by summing weighted reciprocal rank contributions from each retrieval method: $\text{score}(d) = \sum_m w_m / (k + \text{rank}_m(d))$, where $k = 60$ is the standard smoothing constant that prevents high scores for top-ranked items from dominating. Entities appearing in only one list still receive a contribution from that list; entities appearing in both receive additive contributions.

The chosen weights $\text{bm25_weight} = 0.4$ and $\text{semantic_weight} = 0.6$ give a slight advantage to semantic similarity, reflecting the idea that operators tend to describe

infrastructure in informal terms (e.g., “the web frontend” for a service named *react-prod-pod*) more often than they quote exact identifiers. BM25 retains a substantial share to ensure that exact term matches always rank highly when users do use precise names. The RRF smoothing constant $k = 60$ is used without modification, as it is insensitive to the relative scales of the two score distributions.

6.4.3.4 Confidence Thresholds and Safe Failure

A critical property of RRF is that it is a *rank-based* fusion method: it normalizes away the absolute quality signal from each retrieval branch. Even when no entity in the graph genuinely matches the user’s query, the least-bad candidate still receives rank 0 and collects the full RRF contribution. In a general information-retrieval setting this is harmless, but in a security enrichment pipeline, returning a wrong entity silently would cause the system to substitute an incorrect IP address or attach an inappropriate actuator, potentially applying a firewall rule to the wrong host.

To address this, the system enforces confidence thresholds at three levels:

1. **Semantic similarity floor** ($\text{min_similarity} = 0.40$). With the *paraphrase-multilingual-MiniLM-L12-v2* model, cosine similarities below 0.40 indicate weak or no direct semantic relationship. This threshold filters out candidates that are only tangentially related to the query, ensuring that the semantic branch contributes only when there is a meaningful match.
2. **BM25 score floor** ($\text{min_bm25_score} = 2.0$). Empirical observation on the test infrastructure showed that BM25 scores between 1.0 and 2.0 consistently correspond to single overlapping common terms: for example, the word “server” alone produced scores of 1.4–1.5 against a variety of entities as is a common word used in service descriptions. Scores above 2.0 instead indicate multi-term overlap (e.g., “web”, “handling”, “requests” matching jointly), which is a much stronger signal of genuine relevance. Setting the floor at 2.0 filters out these single-term false positives while preserving entities with substantive lexical correspondence.
3. **Fused score floor** ($\text{min_fused_score} = 0.006$). After both source-level filters, a safety-net threshold on the RRF fused score catches edge cases where an entity barely passes one source filter but ranks poorly. With $k = 60$, an entity at rank 0 in a single branch receives at most $0.6/60 = 0.01$ (semantic) or $0.4/60 \approx 0.007$ (BM25), so a top-ranked result from either branch alone still passes. However, a BM25-only match drops below 0.006 after rank 6 ($0.4/67 \approx 0.006$), filtering out lower-ranked single-branch candidates that lack cross-method confirmation.

To cover the case where no candidates pass these thresholds, was decided to implement a safe failure mode where the original query is returned. This choice is based on the principle of least surprise: if the system cannot confidently resolve

the query to a specific entity, it should not attempt to guess but instead return the original input, allowing downstream components and the end user to handle it as an unresolved reference rather than risk applying incorrect context.

6.4.4 Actuator Discovery Algorithm

The most critical graph operation for security enrichment is discovering all actuators capable of enforcing a security policy on a given entity. A service may not have a directly attached actuator, but it may be protected indirectly through its hosting hierarchy or via sibling security services sharing a common controller.

The algorithm follows three traversal paths:

- **Path 1 (direct)** checks for actuators attached directly to the entity via consumer edges. This is the most specific protection level, and it's triggered when the actuator has a direct connection or is directly executed on the target entity.
- **Path 2 (host-level)** recursively traverses the hosting hierarchy upward. A pod inherits the protections of its hosting cloud environment: if the cloud has a network security group actuator, that actuator can enforce policies on any service hosted within it.
- **Path 3 (controller-level)** analyzes management relationships. If an entity is controlled by the same controller as a firewall service, that firewall's actuator can enforce policies on the entity through the shared management plane.

6.5 API Overview

The knowledge management system exposes its functionality through a FastAPI application organized into three routers. A singleton *StateManager* holds references to the loaded *BOMParser* and *GraphStore*, ensuring consistent state across requests. A dependency injection function, raises an HTTP 503 error if clients attempt queries before loading infrastructure data.

Table 6.2 summarizes the available API endpoints.

The */bom/** endpoints support graph exploration and debugging. The */rag/answer* endpoint answers general infrastructure questions using a full context mode with neighbor expansion and relevance scoring. The */rag/enrich* endpoint, described in detail in the following section, serves as the primary integration point with the command generation framework from Chapter 5.

6.6 Security Request Enrichment Pipeline

When the command generation framework's *enrich_request* node issues an HTTP POST to */rag/enrich*, the system executes a three-step pipeline: parallel entity reso-

Table 6.2: Knowledge management system API endpoints.

Endpoint	Method	Purpose
<code>/bom/load</code>	POST	Load BoM file (upload or default), parse it, and build the knowledge graph with BM25 index
<code>/bom/stats</code>	GET	Return parsing statistics: service count, dependency count, graph size
<code>/bom/graph/stats</code>	GET	Return graph statistics: node and edge counts by type
<code>/bom/graph/entity/{name}</code>	GET	Return the full protection chain for a specific entity
<code>/bom/graph/path/{src}/{tgt}</code>	GET	Find the shortest path between two entities
<code>/rag/answer</code>	POST	Answer questions using graph-based RAG with BOM context
<code>/rag/enrich</code>	POST	Enrich security requests with resolved IPs and actuators
<code>/health</code>	GET	Service health check

lution, minimal context building, and LLM-based enrichment. Figure 6.4 illustrates the complete flow.

6.6.1 Step 1: Parallel Entity Resolution and Fast Analysis

The first step identifies which infrastructure entities the user’s request refers to. Two independent analyses execute concurrently, minimizing latency.

Branch A Hybrid Description Resolution submits the user’s natural language description to the hybrid retrieval pipeline, which fuses BM25 and semantic similarity scores via Reciprocal Rank Fusion and returns up to five best-matching entities ($top_k=5$). The confidence thresholds described in Section 6.4.3 already filter out weakly related candidates, so a higher top_k improves recall without flooding the context with noise. (The `/rag/answer` endpoint uses the same top_k but with neighbor expansion, as its goal is to provide a comprehensive answer rather than precise enrichment data.)

Branch B Fast Analysis performs LLM-free entity extraction using four complementary techniques applied in sequence:

1. **IP address extraction:** compiled regex identifies IPv4 and IPv6 literals present in the query.
2. **Protocol extraction:** a pattern matches protocol names (TCP, UDP, ICMP, HTTP, etc.).
3. **Exact name matching:** graph entity names are matched using word-boundary-aware regex, sorted longest-first to prevent substring collisions (e.g., “pod1” must not match “pod10”).

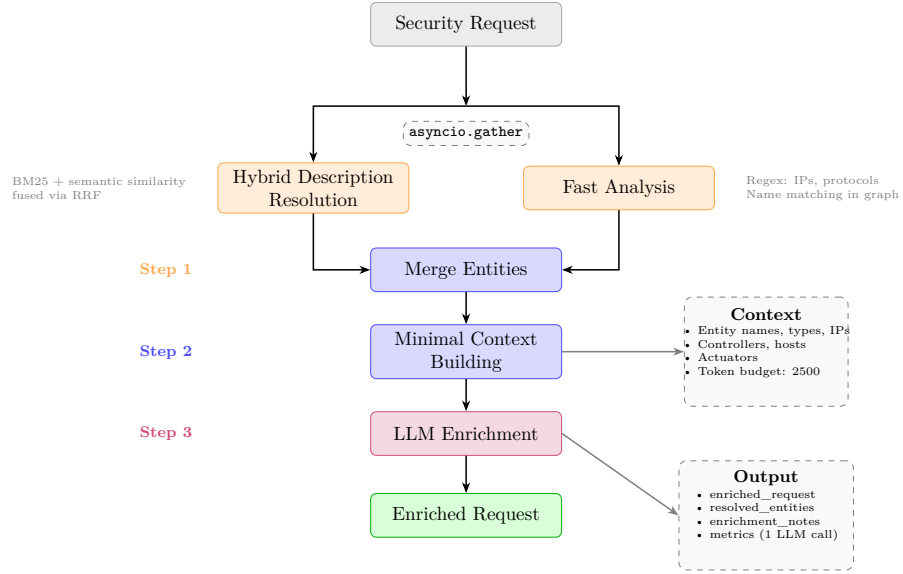


Figure 6.4: Enrichment pipeline for security requests. Step 1 resolves entity references in parallel using hybrid retrieval (BM25 + semantic similarity via RRF) and LLM-free programmatic analysis. Step 2 builds token-budgeted context containing only cited entities and their actuators. Step 3 invokes the LLM to produce the enriched request with resolved IPs and actuator specifications.

After both branches complete, results are merged with priority: entities from the fast analysis receive a score of 100.0 (highest priority), while hybrid-resolved entities carry their RRF fused scores. Deduplication ensures each entity appears only once.

6.6.2 Step 2: Minimal Context Building

The context builder assembles a structured JSON representation of the identified entities, optimized for the enrichment use case. Unlike the full context mode used for question answering (which expands neighbors and applies relevance scoring), the enrichment context includes *only the cited entities and their actuators* without any neighbor expansion. This design choice ensures the context remains concise and focused on actionable information, fitting within the token budget while providing the LLM with the details needed for enrichment.

For each entity, the builder retrieves its protection chain from the graph and constructs a compact dictionary containing:

- **name** and **type**: entity identifier and service type
- **IPs**: IP addresses extracted from the `otupy:x:port:x:address` properties
- **description**: for matching when users refer to entities by description
- **controllers**: entities that manage this service
- **actuators**: up to two security consumers with `actuator_id`, `profile`, `host`, `port`, and `relation` (direct, host-level, or controller-level)

The token budget (2500 tokens, estimated at 4 characters per token) ensures the context fits within a strict 4096-token LLM context window alongside the prompt template and response. Entities are added incrementally; once the budget would be exceeded, the remaining entities are skipped.

The needs of a strict token budget for the enrichment data born as a requirement to ensure that the total amount of tokens used by main prompt + enrichment data + user request does not exceed the context window of local/smaller LLMs.

Full Context Mode for Question Answering

The presence of a full knowledge graph opens the possibility of a more general question-answering capability beyond the enrichment use case. The `/rag/answer` endpoint exposes this as a tangential feature: it is mentioned here because the graph representation makes it possible without additional infrastructure, but it falls outside the scope of this thesis and has not undergone systematic optimization or evaluation.

In practice, the mode works by scoring candidate entities based on how directly they match the query (exact name matches rank highest, BM25 description matches contribute proportionally, and co-hosted entities receive a bonus), then penalizing candidates by their hop distance from the matched nodes. Only entities above a relevance threshold are included. Neighbor expansion depth is adapted to graph density: denser graphs warrant shallower traversal to avoid flooding the context, while sparser graphs allow deeper exploration. The result is a richer but less focused context than the enrichment mode, better suited to open-ended infrastructure questions than to precise policy generation.

6.6.3 Step 3: LLM Enrichment

The final step invokes the LLM with a structured prompt containing the original security request and the graph context. The prompt encodes domain-specific rules for translating informal security language into precise, actionable specifications.

6.6.3.1 Enrichment Prompt Design

The enrichment prompt follows a minimal-transformation philosophy: it instructs the LLM to substitute entity names with concrete network parameters while leaving every other word in the original request untouched. The rules are:

- **Name resolution.** Each entity name found in the request is replaced with the format `<first_ip>/32 (entity_name)` (using `/128` for IPv6 addresses). User-provided IP literals are preserved as-is. If an entity appears in the request but is absent from the context, the original name is kept unchanged.
- **Actuator appending.** The best available actuator is appended once at the end of the enriched request: `using <profile> actuator (consumer-<id>: asset_id=<id>, host=<host>, port=<port>)`. The selection of the best one follows two criteria: first, correlation with the identified entity type (e.g., if the

target is a Kubernetes pod, a Kubernetes-native actuator is preferred); second, when the environment is ambiguous or multiple entities are involved, actuators directly attached to the target take priority, followed by host-level and then controller-level. If no actuator is available for any resolved entity, this clause is omitted entirely.

- **Strict text preservation.** All other reference, numbers, ports, and in general details from the original request are kept exactly as givens. The prompt explicitly prohibits rephrasing.

This peculiar choice, even if it's sound counterintuitive, it's fundamental to preserve original data as much as possible, that otherwise could be misinterpreted or removed by the LLM during the rephrasing. This problem is particularly relevant when the user request delete or update actions, that may contain file names or rule ID that are not strictly correlated with the enrichment process so LLM, by trying to follow the prompt, leave out these information, resulting in an unenforceable command.

- **Output format.** The LLM returns, for practical parsing reasons, a JSON object with the single field *enriched_request* containing the fully enriched request text, without extra explanation or information.

```
{"enriched_request": "<substituted request text>"}
```

A concrete example illustrates the expected transformation:

```
Input:  "block TCP traffic to frontend-pod1 on port 80"
Output: "block TCP traffic to 10.244.1.10/32 (frontend-pod1)
        on port 80 using slpf actuator
        (consumer-np: asset_id=np, host=127.0.0.1, port=8080)"
```

Several worked examples are embedded directly in the prompt to handle edge cases: two entities where only one has an actuator (the actuator of the resolved entity is used), a user-supplied CIDR that must pass through unchanged, and an entity matched via description similarity rather than exact name.

6.6.4 Integration with Command Generation Framework

The command generation framework (Chapter 5) invokes this pipeline during its preprocessing phase. The *enrich_request* node issues an HTTP POST to */rag/enrich* with the user's natural language security request:

```
POST /rag/enrich
{
  "security_request": "Block SSH traffic to the database server",
  "k": 2
}
```

The response provides the enriched request with resolved entities and actuator specifications:

```
{
  "original_request": "Block SSH traffic to the database server",
  "analysis": {
    "identified_entities": ["db-server"],
    "reasoning": "Found entity 'db-server' in query"
  },
  "retrieved_sources": ["db-server"],
  "enriched_request": "Block SSH traffic to 10.50.2.15/32
    (db-server) using slpf actuator
    (consumer-np: asset_id=np, host=127.0.0.1, port=8080)",
  "resolved_entities": {},
  "enrichment_notes": "",
  "metrics": {
    "tokens": {"input_tokens": 312, "output_tokens": 47,
      "reasoning_tokens": 0, "total_tokens": 359},
    "duration_ms": 1840.5,
    "llm_call_count": 1
  }
}
```

The enriched request replaces the original *user_input* in the workflow state, providing the subsequent parsing nodes with concrete network parameters, an explicit traffic direction, and actuator specifications. If the RAG service is unavailable, the preprocessing node logs a warning and continues with the original request, implementing graceful degradation. This separation of concerns enables independent updates to infrastructure knowledge (by refreshing the BoM) without modifying the command generation logic.

As said, of all the fields returned by the enrichment endpoint, only the *enriched_request* is used by the command generation workflow; the other fields are included for observability and debugging purposes but do not affect the enrichment process itself.

Chapter 7

Testing and Evaluation

In Chapter 3 I have already introduced the evaluation schema for the proposed system, that works along three dimensions: accuracy, operational efficiency, and model comparison. This chapter describes the practical realization of this evaluation schema. Section 7.1 presents the composition of the test set, designed to progressively evolve the used inputs, from straightforward technical requests to non-technical more ambiguous ones. Section 7.2 details the automated evaluation framework that validates generated commands, aggregates metrics, and produces comparative reports across multiple LLM backends. Section 7.3 presents the results of the evaluation across five models. The test set is aligned with an infrastructure Bill of Materials created ad hoc and consumed by the RAG service described in Chapter 6, ensuring that every test case operates on the same data.

7.1 Test Scenarios

7.1.1 Test Set Design Methodology

The evaluation relies on a curated set of **56 test cases**, each defined as a JSON object with five fields:

- **ID**: A unique identifier (e.g., TC001) that encodes the category and sequence of the test case.
- **Description**: A brief summary of the test case’s intent and key challenge.
- **Input**: The natural language prompt that simulates a real-world operator request, designed to test specific capabilities of the system.
- **Expected**: The manually verified Otupy command string (or array of commands for multi-command cases) that represents the correct output for the given input, along with the expected consumer endpoint.
 - **Command(s)**: The expected OpenC2 command(s) in Otupy syntax, encoding the action, target, actuator, and arguments that the system should generate.
 - **Consumer**: An object specifying the expected consumer endpoint with *host* (that could be the IP or domain name) and *port* fields, validating that

the system selects the correct destination for the command.

- **Tags:** A list of categorical tags (e.g., “description-based”, “non-technical language”, “multilingual”) that facilitate filtering and analysis by test category.

Every entity referenced (directly or indirectly) in the test inputs exists in the BoM file (*bom.json*) used by the RAG service, so that the evaluation simulates realistic retrieval scenarios.

The expected outputs are manually verified Otupty command strings encoding every component. Both sides of the comparison (expected and generated) are parsed by the same regex-based extractor, guaranteeing consistent treatment of formatting differences.

The test set tries to follow a principle of *progressive difficulty*. The first group of cases uses pieces of the entity descriptions taken from the BoM and standard networking terminology. Subsequent groups introduce non-technical business language that demands intent interpretation, action verb synonyms, advanced arguments, multilingual inputs, and special OpenC2 actions beyond the allow/deny pair.

7.1.2 Infrastructure Coverage

All test cases target entities drawn from a single BoM that models a heterogeneous infrastructure spanning three environments, each managed by a distinct security actuator and reachable through a dedicated consumer endpoint.

- **Azure Cloud:** Four virtual machines (web server, API gateway, database server, Redis cache) grouped under the *azure* actuator. The consumer endpoint is an Azure Network Security Group at *azure-nsg.eastus.azure.com:443*.
- **Kubernetes Cluster:** Five pods (two frontend, two backends, one database) managed via Kubernetes NetworkPolicies (*np* actuator). The consumer endpoint is a local controller at *127.0.0.1:8080*.
- **OpenStack Legacy:** Four virtual machines (two application servers, an Oracle database, a SAP integration server) protected by OpenStack security groups (*openstack* actuator). The consumer endpoint is *openstack.legacy.local:8080*.

Several test cases exercise cross-environment references for instance like the Test Case TC014 that targets a Redis cache server on Azure from a specific subnet.

7.1.3 Test Categories

The 56 test cases are divided in six categories, each one focused on testing a specific ability of the system. Table 7.1 provides a first overview of the different categories.

Two test cases (TC021, TC049) require the system to generate multiple commands from a single request; TC021 also belongs to the non-technical language category, as it combines request decomposition with informal business terminology.

Table 7.1: Test category distribution.

Category	IDs	Count	Key Challenge
Description-based	TC001–TC020	20	Semantic entity retrieval via BoM descriptions
Non-technical language	TC021–TC034	14	Business terminology without protocol details
Action synonyms	TC037–TC044	8	Verb variation (cut off, kill, halt, grant...)
Advanced SLPF	TC046–TC050	5	Extra Args: Duration, drop_process, multi-command, ...
Multilingual (Italian)	TC035–TC036, TC045	3	Non-English inputs
Special actions	TC051–TC056	6	actions: <i>query</i> , <i>delete</i> , <i>update</i>

Table 7.2: Distribution of test cases by OpenC2 action.

Action	Count	Notes
<i>deny</i>	31	Including <i>drop_process</i> variants and bidirectional blocks
<i>allow</i>	19	Including subnet-filtered and cross-pod rules
<i>query</i>	2	Feature and profile listing
<i>delete</i>	2	Firewall rule removal by rule number
<i>update</i>	2	Configuration file update

7.1.3.1 Description-Based Entity Resolution (TC001–TC020)

These 20 test cases form the largest group and the baseline for semantic retrieval evaluation. Inputs reference entities by their BoM description rather than by name, using characteristics such as software stack, role, and operational details. The system must perform semantic retrieval through the hybrid BM25 + dense search described in Chapter 6 to resolve the correct entity. A representative example:

TC003 *Block Redis port to distributed cache cluster via description*

Listing 7.1: TC003: description-based entity resolution.

```

1 # Input
2 "Block Redis traffic on port 6379 to the distributed
3  cache cluster used for session management and product catalog
   caching"
4
5 # Expected output
6 oc2.Command(
7     action=oc2.Actions.deny,
8     target=oc2.IPv4Connection(dst_addr=oc2.IPv4Net("\
10.1.1.15\")),
9     protocol=oc2.L4Protocol.tcp, dst_port=oc2.Port(6379)),
10    actuator=slpf.Specifiers({'asset_id': 'azure'}),
11    args=slpf.Args({'direction': slpf.Direction.ingress,
12                   'response_requested': oc2.ResponseType.complete})
13 ),

```

14

```
15 # Consumer: azure-nsg.eastus.azure.com:443
```

The input never mentions *azure-db-vm2*; instead, it describes the entity using characteristics from the BoM description (Presence of Redis server, use for session management and product catalog caching). The system must resolve this to *10.1.1.15*, select the correct actuator (*azure*), infer the protocol (TCP) and determine the direction (ingress).

7.1.3.2 Non-Technical Language (TC021–TC034)

These 14 test cases simulate requests from operators without deep networking knowledge. Inputs use business terminology (“online shop”, “financial reports”, “login sessions”) and frequently omit protocol or port details entirely:

TC021 *Take down the online shop (multi-command)*

Listing 7.2: TC021: non-technical language (multi-command).

```
1 # Input
2 "Take down the online shop, nobody should be able
3 to reach it"
4
5 # Expected output [0]: first frontend pod
6 oc2.Command(
7     action=oc2.Actions.deny,
8     target=oc2.IPv4Net("10.244.1.10"),
9     actuator=slpf.Specifiers({'asset_id': 'kubernetes'}),
10    args=slpf.Args({
11        'direction': slpf.Direction.ingress,
12        'response_requested': oc2.ResponseType.complete
13    })
14 )
15
16 # Expected output [1]: second frontend pod
17 oc2.Command(
18     action=oc2.Actions.deny,
19     target=oc2.IPv4Net("10.244.1.11"),
20     actuator=slpf.Specifiers({'asset_id': 'kubernetes'}),
21    args=slpf.Args({
22        'direction': slpf.Direction.ingress,
23        'response_requested': oc2.ResponseType.complete
24    })
25 )
26 # Consumer: 127.0.0.1:8080
```

This test case combines two challenges: the system must interpret “online shop” as the two frontend pods (*frontend-pod1*, *frontend-pod2*) and generate one command per pod and the “Take down” as a *deny* action. Without any reference to protocols or particular function as a webpage, file hosting or some other service, the preferred target

is an *IPv4Net* (network-level block) rather than an *IPv4Connection* (connection-level block), and the system must infer the direction (ingress) from the concept of “nobody should be able to reach it”.

7.1.3.3 Action Synonyms (TC037–TC044)

These eight test cases use informal verb synonyms for the standard *deny/allow* actions “cut off”, “kill”, “halt”, “drop”, “enable”, “open up”, “grant”, “let through” to test whether the model can correctly map colloquial language to OpenC2 actions:

TC037 *Cut off SSH to NGINX web proxy*

Listing 7.3: TC037: action synonym test case.

```

1 # Input
2 "Cut off SSH access to the NGINX web proxy in Azure"
3
4 # Expected output
5 oc2.Command(
6     action=oc2.Actions.deny,
7     target=oc2.IPv4Connection(
8         dst_addr=oc2.IPv4Net("10.1.1.10"),
9         protocol=oc2.L4Protocol.tcp,
10        dst_port=oc2.Port(22)
11    ),
12    actuator=slpf.Specifiers({'asset_id': 'azure'}),
13    args=slpf.Args({
14        'direction': slpf.Direction.ingress,
15        'response_requested': oc2.ResponseType.complete
16    })
17 )
18 # Consumer: azure-nsg.eastus.azure.com:443

```

The model must recognize that “cut off” is semantically equivalent to *deny* and resolve “NGINX web proxy in Azure” to the correct entity/machine that it’s running it.

7.1.3.4 Advanced Arguments + direct reference (TC046–TC050)

These five test cases exercise less straightforward arguments and direct entity references that bypass the retrieval step. The model must correctly interpret the advanced arguments and incorporate them into the generated command:

- **TC046:** *duration=7200000* a temporary two-hour block, specified in milliseconds.
- **TC047:** *drop_process=reject* the firewall must send a TCP RST or ICMP unreachable.
- **TC048:** *drop_process=false_ack* combined with *response_requested=none*.
- **TC049:** Multi-command generation, blocking SSH to two VMs from one request.

- **TC050:** *protocol=sctp* with *src_port* filtering a rare protocol and uncommon parameter.

TC046 *Temporary block with duration*

Listing 7.4: TC046: advanced SLPF argument (duration).

```

1 # Input
2 "Block HTTP traffic to azure-web-vm1 for 2 hours"
3
4 # Expected output
5 oc2.Command(
6     action=oc2.Actions.deny,
7     target=oc2.IPv4Connection(
8         dst_addr=oc2.IPv4Net("10.1.1.10"),
9         protocol=oc2.L4Protocol.tcp,
10        dst_port=oc2.Port(80)
11    ),
12    actuator=slpf.Specifiers({'asset_id': 'azure'}),
13    args=slpf.Args({
14        'direction': slpf.Direction.ingress,
15        'duration': oc2.Duration(7200000),
16        'response_requested': oc2.ResponseType.complete
17    })
18 )
19 # Consumer: azure-nsg.eastus.azure.com:443

```

The model must interpret “for 2 hours” as a duration argument and convert it to milliseconds (7 200 000 ms).

7.1.3.5 Multilingual Italian (TC035–TC036, TC045)

Three test cases present inputs in Italian to assess multilingual robustness (that should be provided by the use of *paraphrase-multilingual-MiniLM-L12-v2* embedding model as introduced in presentation of the RAG used libraries 6.2). The expected outputs are identical to their English counterparts, validating that the LLM can extract the same structured components regardless of the input language:

TC035 *Block checkout page (Italian)*

Listing 7.5: TC035: Italian language test case.

```

1 # Input
2 "Blocca tutte le connessioni verso la pagina dove
3 i clienti completano il pagamento con carta di credito"
4
5 # Expected output
6 oc2.Command(
7     action=oc2.Actions.deny,
8     target=oc2.IPv4Net("10.244.1.10"),
9     actuator=slpf.Specifiers({'asset_id': 'kubernetes'}),

```

```
10  args=slpf.Args({
11      'direction': slpf.Direction.ingress,
12      'response_requested': oc2.ResponseType.complete
13  })
14  )
15  # Consumer: 127.0.0.1:8080
```

7.1.3.6 Special Actions (TC051–TC056)

The final six test cases target OpenC2 actions beyond the *allow/deny* pair: *query* (feature and profile listing), *delete* (rule removal by number), and *update* (configuration file update). These actions use different target types (*Features*, *RuleNumber*, *File*) that are distinct from the connection and network targets used by firewall rules:

TC051 *Query profiles on Kubernetes cluster*

Listing 7.6: TC051: special action (query).

```
1  # Input
2  "List the security profiles supported by the
3  Kubernetes cluster"
4
5  # Expected output
6  oc2.Command(
7      action=oc2.Actions.query,
8      target=oc2.Features([
9          oc2.Feature.profiles
10     ]),
11     actuator=slpf.Specifiers({'asset_id': 'kubernetes'}),
12     args=slpf.Args({
13         'response_requested': oc2.ResponseType.complete
14     })
15 )
16 # Consumer: 127.0.0.1:8080
```

7.2 Evaluation Framework

This evaluation framework, as the full project, is model-agnostic: it accepts any LLM backend (local or cloud) and applies the same validation logic to all outputs. The evaluation dimensions and metrics introduced in Chapter 3 are implemented through four components: a semantic validator, a metrics aggregator, a report generator, and an evaluation runner that orchestrates the full pipeline.

7.2.1 Semantic Validation

The semantic validator is the core of the evaluation framework. It compares a generated Otupy command string against the expected output on a component-by-component basis and produces a score between 0.0 and 1.0.

7.2.1.1 Component Extraction

Both the expected and generated command strings are parsed by the same set of regex patterns, each isolating a single component from the Otupy syntax. Two additional components (*consumer_host* and *consumer_port*) are extracted from the consumer string via dedicated patterns, bringing the total to 16 tracked components. Table 7.3 lists all components with their comparison strategy.

Table 7.3: Tracked components and comparison strategies.

Component	Comparison	Description
<i>action</i>	Exact	OpenC2 action (deny, allow, query, delete, update)
<i>target_type</i>	Exact	Otupy target class (IPv4Connection, IPv4Net, Features, ...)
<i>src_addr</i>	IP	Source IP address or CIDR block
<i>dst_addr</i>	IP	Destination IP address or CIDR block
<i>protocol</i>	Exact	Layer-4 protocol (tcp, udp, sctp, icmp)
<i>src_port</i>	Exact	Source port number
<i>dst_port</i>	Exact	Destination port number
<i>actuator</i>	Exact	Actuator asset identifier
<i>direction</i>	Exact	Traffic direction (ingress, egress)
<i>duration_ms</i>	Exact	Rule duration in milliseconds
<i>start_time_offset_s</i>	Exact	Start time offset in seconds
<i>stop_time_offset_s</i>	Exact	Stop time offset in seconds
<i>drop_process</i>	Exact	Drop strategy (none, reject, false_ack)
<i>response_requested</i>	Exact	Response type (complete, none, ack)
<i>consumer_host</i>	Exact	Consumer endpoint hostname or IP
<i>consumer_port</i>	Exact	Consumer endpoint port number

Applying the same extraction logic to both sides eliminates discrepancies caused by whitespace, quoting style, or argument ordering, ensuring that the comparison reflects semantic equivalence rather than syntactic formatting.

7.2.1.2 Presence-Based Equal Weighting

Only components that have non-*None* values in the expected output participate in the score calculation. Each participating component contributes equally: the base score is the arithmetic mean of the individual component scores. This means that a test case expecting six components weights each at approximately 16.7%, while one expecting ten components weights each at 10%. The approach avoids manually tuned weights and ensures that every specified component matters proportionally.

7.2.1.3 Comparison Strategies

Two comparison strategies handle the different component types:

1. **Exact match** (default): The component scores 1.0 if the generated value matches the expected value exactly, and 0.0 otherwise. This applies to categorical fields like *action*, *protocol*, and *direction* where any deviation is significant.

2. **IP address comparison:** Works similarly to the Exact match, but includes a normalization mechanism that treats single-host IP addresses with or without the */32* mask as equivalent (same mechanism for IPv6 with the */128* mask). For example, *10.0.0.1/32* and *10.0.0.1* are considered a match.

7.2.1.4 Critical Components and Failure Capping

The *action* and *target_type* components are designated as *critical*. So if the selected action is not the one expected, the *critical_failed* flag is set and the overall score is capped at 0.50, regardless of how well other components match. This reflects the fact that selecting the wrong action (e.g., *allow* instead of *deny*) or the wrong target type makes the command fundamentally incorrect from a security perspective: no combination of correct addresses or ports can compensate.

7.2.1.5 Extra Component Penalty

After computing the base score, the validator checks for components present in the generated output but absent from the expected output. Each extra component incurs a penalty of -0.20 , subtracted from the base score (floored at 0.0). This mechanism penalizes *hallucinated fields*: for instance, if the model invents a *src_port* filter that was not specified in the request, the extra component degrades the score even if all other components are correct.

7.2.1.6 Pass Threshold

The semantic **pass threshold is set to 1.0** (100%). A test case passes only when every expected component matches exactly and no extra components are present. This strict criterion was chosen because security commands must be precise: a firewall rule with a wrong port could block legitimate traffic, while a missing direction argument could leave an attack vector open.

7.2.2 Consumer Validation

Beyond the command itself, the system must select the correct consumer endpoint to which the command is dispatched. The validator extracts *host* and *port* from the generated consumer string using dedicated regex patterns and compares them against the expected values. If both match, the score is 1.0; if only one matches, the score is 0.5; if neither matches, the score is 0.0. The *consumer_host* and *consumer_port* fields are added to the component-level tracking so they participate in per-component accuracy reporting alongside the command components.

7.2.3 Multi-Command Validation

For test cases where the expected output contains multiple commands (TC021, TC049), the validator uses a *greedy matching algorithm*¹. This handles the fact that the system may produce commands in any order. Unmatched generated commands receive a score of 0.0.

7.2.4 Metrics Collection

The evaluation framework collects metrics along the three dimensions defined in Chapter 3, implemented through dedicated data structures that aggregate per-test results into model-level statistics.

7.2.4.1 Semantic Accuracy

Semantic accuracy is tracked at two levels. At the *test-case level*, each test receives a semantic score between 0.0 and 1.0 and a binary pass/fail label (pass requires a score of exactly 1.0). At the *component level*, a dedicated data structure records, for each of the summary components, the counts of exact matches, partial matches, mismatches, and missing values across all test cases, plus the average score and component-level accuracy.

7.2.4.2 Operational Efficiency

Three aspects of operational efficiency are measured:

- **Clarification metrics:** The clarification rate (fraction of test cases where the system requested human interaction), the average number of questions per clarification event, and the distribution of questions by component. A lower clarification rate indicates that the system can handle inputs autonomously, which is desirable for production deployment (but still a possible problem if the system proceeds without having retrieved enough information to generate a complete command).
- **Latency metrics:** Mean, median, standard deviation, minimum, maximum, and configurable percentiles (e.g., p95) of execution time per test case are recorded. Clarification wait time, the interval during which the system is waiting for human input, is subtracted from the measured execution time, so that only processing time is reported.

This metric is useful to reflect the responsiveness of each LLM but, with the current implementation, with the use of OpenRouter API, the latency is dominated by API response times and the different availability and capabilities of each provider’s API, so latency comparisons should be interpreted with

¹Each generated command is compared against all remaining expected commands, matched to the one that yields the highest semantic score, and that expected command is then removed from the pool.

caution and in the context of the specific deployment scenario. To get a more fair comparison should be implemented a local evaluation with the same models running on same hardware or use API provided directly by the model provider without the OpenRouter layer.

- **Token consumption:** Input tokens, output tokens, reasoning tokens (for models that expose chain-of-thought), and total tokens are tracked per test case. Each LLM invocation within the graph (decomposition, action parsing, parallel parsing) contributes its token count as a *(node_name, count)* tuple, which is summed at the test-case level and aggregated across the full evaluation.

7.2.4.3 Model Comparison

When multiple models are evaluated, the metrics aggregator produces a comparison table with per-model rankings across pass rate, average semantic score, clarification rate, average latency, and error rate. For each metric, the best-performing model is identified. This enables direct comparison between models of different sizes, architectures, and providers.

7.2.5 Report Generation

The report generator produces four output artifacts, all timestamped for traceability:

1. **Detailed CSV** (one per model): One row per test case, with columns for test ID, input text, semantic score, pass/fail status, execution time, enrichment duration, clarification count, questions asked, generated commands, consumer validation, individual scores for all 16 components, extra component data, and token counts (input, output, reasoning, total).
2. **Summary CSV**: One row per model, with only the aggregated metrics including pass rate, average semantic score, per-component accuracies for the summary components, clarification rate, latency statistics, error count, and error rate.
3. **Comparison CSV** (generated when two or more models are evaluated during the same run): One row per metric (pass rate, average semantic score, clarification rate, average latency, error rate), one column per model, plus a column identifying the best model for each metric.
4. **Full JSON report**: A structured document containing the complete evaluation data, including per-test-case semantic breakdowns with component-level scores, suitable for programmatic analysis and visualization.

7.2.6 Evaluation Execution

The evaluation is invoked via the command-line interface, which supports several execution modes:

- `python main.py -evaluate -local`: Evaluates all configured local Ollama models sequentially.
- `python main.py -evaluate -cloud`: Evaluates all configured cloud models via OpenRouter.
- `python main.py -evaluate -all`: Runs the full evaluation across both local and cloud models.
- `python main.py -evaluate -model <name> -type <ollama/openrouter>`: Evaluates a single specific model.
- `python main.py -evaluate -local -test TC001 TC002`: Runs only the specified test cases.

For each model, the runner sets the appropriate environment variables, initializes a fresh LangGraph instance, and executes each test case sequentially with the initial state provided with the natural language input for the test case in execution. Results are collected into typed data structures and passed to the metrics aggregator and report generator.

The model connected to the RAG pipeline⁶ is defined separately in its configuration file. This choice, in a production environment, should help to set a less demanding (or in general a more suited one for the kind of operation) model for the RAG pipeline and a more capable one for the command generation, to optimize latency and cost while maintaining accuracy. But for this evaluation, the same model is used for both steps to isolate the LLM ability across the full pipeline.

7.2.7 Model Configurations

Table 7.4 lists the models evaluated in this work. All models are accessed through the OpenRouter API and run with the same system prompts and pipeline parameters, isolating the LLM as the only variable across evaluations.

Table 7.4: Models evaluated.

Model	Provider	Notes
Grok-4 Fast	xAI (OpenRouter)	Reasoning-capable model with chain-of-thought
DeepSeek v3.2	DeepSeek (OpenRouter)	General-purpose large language model
Trinity Large Preview	Arcee AI (OpenRouter)	Highest average semantic score
GLM-4 32B	Zhipu AI (OpenRouter)	32B parameter model
Minstral 8B	Mistral AI (OpenRouter)	Smallest and fastest model evaluated

The selection spans different provider ecosystems and model sizes, from the compact Minstral 8B to reasoning-augmented architectures such as Grok-4 Fast. All models are configured identically, enabling fair comparison of their ability to translate natural language into OpenC2 commands.

Although an Ollama integration is implemented and the original plan included local models, all models are ultimately accessed through OpenRouter for the evaluation. This choice avoids hardware-related limitations and ensures full use of the project’s

parallelization capabilities, which would not be feasible with local models due to resource constraints.

7.3 Results Analysis

This section presents the evaluation results across the five models listed in Table 7.4, covering overall accuracy, per-category performance, component-level analysis, failure patterns, operational efficiency, and a comparative discussion.

7.3.1 Overall Comparison

Table 7.5 summarizes the headline metrics for each model. All five models completed the full 56-test evaluation without execution errors.

Table 7.5: Overall evaluation results across five models.

Model	Passed	Pass Rate	Avg Score	Mean Lat.	Med. Lat.	P95 Lat.
Grok-4 Fast	43/56	76.8%	0.872	23.3 s	22.3 s	36.3 s
DeepSeek v3.2	42/56	75.0%	0.922	32.1 s	29.6 s	52.5 s
Trinity Large	40/56	71.4%	0.923	81.9 s	56.7 s	174.7 s
GLM-4 32B	39/56	69.6%	0.908	13.0 s	11.7 s	22.6 s
Ministral 8B 2512	37/56	66.1%	0.899	4.9 s	4.0 s	8.7 s

Grok-4 Fast achieves the highest pass rate (76.8%), while Trinity Large and DeepSeek v3.2 share the highest average semantic scores (≈ 0.92). Notably, a higher average score does not always correlate with a higher pass rate: for instance, Trinity Large ranks third in pass rate despite having the highest average score. This means that, on average, about 92% of each generated command’s components are correct, but only 71% of the commands are fully correct and since a perfect score is required to flag a test as passed, a single wrong field is enough to push the overall score below the passing threshold. Ministral 8B, the smallest model, delivers the lowest latency (4.9s mean), which is $4.7\times$ faster than Grok-4 and $16.7\times$ faster than Trinity Large, albeit at a lower pass rate (66.1%).

An important observation should be given to the latency, as this extreme difference between models is caused mainly by the different availability and capabilities of each endpoint available on the OpenRouter Platform. For instance, the Mistral model at the execution, had available only one endpoint that provides a latency to the request of around 0.12 seconds with a Throughput of 190 token/s, while for Trinity Large where present two execution endpoints, one with a latency of around 0.72s and 80 token/s and another with a latency of around 0.8s and 6 token/s, so the latency difference is not only caused by the model capabilities and reasoning times (if presents) but also by the provider’s API performance, which should be taken into account when interpreting the results.

More in-depth data about the latency could be found in the Appendix A.1.

7.3.2 Per-Category Analysis

Table 7.6 breaks down the pass rate by test category. The categories are ordered by overall difficulty, from the highest average pass rate to the lowest.

Table 7.6: Pass rate (%) by test category.

Category	Grok-4	DeepSeek	Trinity	GLM-4	Ministral
Special actions (6)	100	100	67	100	83
Description-based (20)	95	80	90	90	85
Adv. SLPF (5)	80	80	60	80	20
Action synonyms (8)	75	75	75	62	62
Multilingual (3)	33	33	67	0	100
Non-technical (14)	50	64	50	43	43

7.3.2.1 Special actions.

Three models (Grok-4, DeepSeek, GLM-4) achieve 100% on the *query/delete/update* test cases, demonstrating that these actions are well handled. Trinity Large and Ministral 8B each fail one or two special-action tests, primarily on target-type selection (*Features* vs. *RuleNumber*).

7.3.2.2 Description-based.

The strongest category overall, with all models scoring 80% or above. Grok-4 leads at 95%, failing only on TC016 (inter-pod HTTPS routing). The high scores confirm that the RAG-based entity resolution described in Chapter 6 reliably maps descriptive references to the correct BoM entities.

7.3.2.3 Advanced SLPF.

Grok-4, DeepSeek, and GLM-4 all reach 80%, while Ministral 8B scores only 20%: it fails on correctly defining *drop_process*, *duration*, and *src_port* parameters. This suggests that smaller models struggle with more complex arguments and requests.

7.3.2.4 Action synonyms.

Three models tie at 75%, while GLM-4 and Ministral 8B reach 62%. The main failure is TC041 (“halt all incoming requests”), which all five models fail. Four of them misassign the destination address in the target, while trinity large also fails to correctly interpret the protocol. These results indicate that models can handle common action synonyms and the failures are more related to the correct extraction of data from the RAG and the correct enrichment of the command rather than the correct interpretation of the action verb.

7.3.2.5 Multilingual.

Performance is highly model-dependent: Ministral 8B passes all three Italian tests (100%), while GLM-4 fails all of them (0%). Trinity Large passes two out of three, and both Grok-4 and DeepSeek pass only one. The results suggest as first instance that *paraphrase-multilingual-MiniLM-L12-v2* as embedding model is capable enough to handle the multilingual retrieval, but the LLM’s ability to interpret the retrieved information and integrated them with the Italian text is more variable across models and not seem to be coherent with model size or general performance.

7.3.2.6 Non-technical language.

This is the hardest category across all models, with pass rates ranging from 43% (GLM-4, Ministral 8B) to 64% (DeepSeek v3.2). The absence of protocol-level details forces the model to infer intent from business terminology alone, which proves challenging for all five models. DeepSeek v3.2 performs best, possibly due to its parameter size of 685B and its training data, which may include more business-oriented language.

7.3.3 Component-Level Accuracy

Table 7.7 reports the exact-match accuracy for each component across all models. Only components that appear in at least one expected output are included.

Table 7.7: Component-level exact-match accuracy (%). The *N* column indicates how many times each component appears in the test set.

Component	N	Grok-4	DeepSeek	Trinity	GLM-4	Ministral
<i>action</i>	58	100.0	100.0	100.0	100.0	100.0
<i>target_type</i>	58	87.3	96.4	98.2	96.4	98.2
<i>protocol</i>	39	97.4	100.0	94.9	100.0	100.0
<i>response_requested</i>	58	100.0	100.0	100.0	100.0	98.2
<i>actuator</i>	58	96.5	96.6	91.4	96.5	93.0
<i>direction</i>	52	86.3	92.3	92.3	92.2	96.1
<i>consumer_host</i>	56	96.4	96.4	91.1	96.4	92.9
<i>consumer_port</i>	56	96.4	96.4	91.1	96.4	92.9
<i>dst_addr</i>	35	91.4	91.4	85.7	80.0	94.3
<i>dst_port</i>	37	97.3	83.8	89.2	89.2	89.2
<i>src_addr</i>	10	100.0	100.0	90.0	90.0	80.0
<i>src_port</i>	1	100.0	100.0	100.0	100.0	0.0
<i>drop_process</i>	2	100.0	100.0	100.0	100.0	0.0
<i>duration_ms</i>	1	100.0	100.0	100.0	100.0	0.0

Some patterns emerge:

- **Action classification is fully reliable:** All five models achieve 100% on the *action* component, confirming that the mapping from natural language to OpenC2 actions (*deny*, *allow*, *query*, *delete*, *update*) is robust across architectures.

- **Address resolution is the primary error source:** *dst_addr* accuracy ranges from 80.0% (GLM-4) to 94.3% (Ministral), making it the weakest standard component. Errors here trace back to the RAG retrieval step: when the entity description is ambiguous or matches multiple candidates, the wrong IP address may be selected.
- **Ministral 8B lacks specialized parameter handling:** It scores 0% on *src_port*, *drop_process*, and *duration_ms*. These parameters appear in only 1 to 2 test cases each, but they represent SLPF features that the smallest model consistently fails to generate, suggesting that these less common constructs require more model capacity, targeted training or different prompting strategies to be handled correctly.
- **Direction inference varies:** Grok-4 achieves only 86.3% on *direction* the lowest among the five models despite having the highest pass rate overall. An inspection reveals that its direction errors tend to occur alongside correct generation of all other components, indicating that direction inference may be a specific weak point for this model, possibly due to ambiguous phrasing in the input or retrieval results that lead to confusion between *ingress*, *egress* and the split of an action in more than one command with different directions when the direction **both** should be used.

7.3.4 Failure Analysis

To better understand the failure modes, some of the most failed test cases were manually reviewed. Three test cases (TC022, TC030, TC033) were failed by all five models, and they share the pattern of combining non-technical language with entity descriptions that require semantic disambiguation.

- **TC022** “*Let the accountants access the system where they run their financial reports*”. The phrase “financial reports” maps to the OpenStack SAP middleware server description, but the indirection through a user role (“accountants”) and a business activity (“financial reports”) seems enough to retrieve the information from the RAG but too vague to define a correct port and protocol, leading all models to fail the test case.
- **TC030** “*Fully isolate the backup server for the procurement middleware*”. The combination of “backup server” with “procurement middleware”, very generic terms that could be used in multiple descriptions, to retrieve multiple elements. Then the Rag pipeline seems able to correctly enrich the request with the correct IP address but not to correctly define the consumer and actuator fields, which are the ones that cause the failure for the majority of the models.
- **TC033** “*Restore access to the login and authentication gateway*”. The synonym “restore” for *allow* combined with “login and authentication gateway” that, based on the LLM, lead again the rag pipeline to complete fail the enrichment

of the request or to provide the wrong element as are used too generic terms used in multiple descriptions. In this case the failure is even more severe as also the destination address is wrong for all models, which means that the retrieval process is not able to correctly disambiguate the retrieved information and to provide the correct one to the LLM, which then fails to generate a correct command.

An additional four test cases (TC021, TC023, TC029, TC043) are failed by four out of five models. These share the pattern of combining non-technical or synonym-based language with entity descriptions that require semantic disambiguation.

7.3.5 Clarification Behavior

Table 7.8 reports the clarification metrics for each model.

Table 7.8: Clarification behavior.

Model	Tests w/ Clar.	Total Questions	Observations
GLM-4 32B	5	24	Most questions per event (4.8 avg)
Minstral 8B	2	12	6 questions per event when triggered
DeepSeek v3.2	4	11	2.8 questions per event
Grok-4 Fast	3	7	2.3 questions per event
Trinity Large	1	3	Near-autonomous operation

GLM-4 requests clarification most frequently (5 tests, 24 total questions), while Trinity Large operates nearly autonomously (1 test, 3 questions). Clarification frequency does not correlate positively with pass rate: the two models with the fewest clarifications (Trinity Large and Grok-4) have the first and third highest pass rates, respectively. This suggests that, in this evaluation setting, models that attempt to resolve ambiguity internally rather than requesting human input achieve comparable or better results, while also maintaining a smoother automated workflow. However, the small number of clarification events limits the statistical significance of this observation, and further testing with more cases designed to trigger clarifications would be needed to draw stronger conclusions about the relationship between clarification behavior and accuracy.

7.3.6 Token Efficiency

Table ?? summarizes token consumption across the full 56-test evaluation.

GLM-4 is the most token-efficient (896 K total), while Grok-4 Fast consumes the most (1 547 K), driven by 259 K reasoning tokens from its chain-of-thought process. Despite this overhead, Grok-4’s reasoning tokens appear to contribute to its leading pass rate. Minstral 8B and DeepSeek v3.2 consume more tokens than GLM-4 and Trinity Large, given their higher rate of clarification requests that add to the input and output token counts.

Table 7.9: Token consumption across the full evaluation (56 tests).

Model	Input	Output	Reasoning	Total
GLM-4 32B	869 K	27 K		896 K
Trinity Large	944 K	53 K		996 K
Ministral 8B	1045 K	61 K		1106 K
DeepSeek v3.2	1136 K	50 K		1185 K
Grok-4 Fast	1260 K	29 K	259 K	1547 K

In conclusion, given the combination of long predefined prompts and the relatively limited number of tokens required to generate each command, the discrepancy in total token consumption is mainly caused by two factors: the varying number and type of clarifications requested by each model (asking multiple questions to generate a single element leads to minor token usage than asking one question per element, as less LLM calls are triggered), which increase both input and output token counts, and the reasoning tokens used by Grok-4 Fast, the only model employing a chain-of-thought process. Therefore, token efficiency is correlated to model’s specific behavior in terms of clarifications and reasoning.

7.3.7 Discussion

The evaluation reveals that no single model dominates across all dimensions. The key trade-offs can be summarized along three axes:

7.3.7.1 Accuracy vs. latency.

Grok-4 Fast offers the best pass rate (76.8%) with moderate latency (23.3s mean). Ministral 8B provides the fastest responses (4.9s mean) at a 10.7 percentage point accuracy cost. For latency-sensitive deployments and considering the analyzed models, Ministral represents a viable trade-off; for accuracy-critical scenarios, Grok-4 or DeepSeek may be preferable despite their higher latency.

7.3.7.2 Pass rate vs. average score.

Trinity Large and DeepSeek v3.2 have the highest average semantic scores (≈ 0.92) but rank third and second in pass rate, respectively. This indicates that their failures, while fewer, tend to produce partially correct commands that score above zero but below the 1.0 threshold. In a production setting where partial correctness is unacceptable for security commands, pass rate is the more relevant metric. However, both could be considered as base for a more fine-tuned implementation of the system to compensate for the specific weaknesses of each model.

7.3.7.3 Autonomy vs. accuracy.

Models that request fewer clarifications (Trinity, Grok-4) tend to perform as well as or better than those that request more (GLM-4, DeepSeek). While clarifications

could theoretically improve accuracy by resolving ambiguity, in practice they add latency and interrupt the automated workflow. The results suggest that, for this task, internal disambiguation by a capable model is preferable to interactive clarification that should be reserved for cases of complete failure or as a last check before executing a command that the system is not confident about.

7.3.7.4 Category-specific strengths.

Each model exhibits distinct strengths: Ministral 8B is the only model to pass all Italian tests; Grok-4 leads on description-based retrieval (95%) and not allow/deny actions (100%); DeepSeek v3.2 handles non-technical language best (64%). A production deployment could leverage these complementary strengths through model routing based on input classification.

7.3.7.5 Limitations.

The evaluation has several limitations. First, the test set, while covering six kinds of user request and three infrastructure environments, contains 56 test cases sufficient for comparative analysis but not for statistically robust confidence intervals. Second, the single-run evaluation does not capture model stochasticity; repeated runs with temperature variation could provide a more complete accuracy picture. Finally, the presence of universally failed test cases highlight that highly abstract business language and rare protocol combinations remain open challenges for current LLM architectures.

Chapter 8

Conclusions

To summarize, this thesis set out to design, implement, and evaluate an AI-driven system capable of translating natural language security requirements into executable OpenC2 commands via the Otypy library. The proposed solution comprises two tightly integrated components: a multi-stage agentic command generation framework built on LangGraph (Chapter 5), which models the translation process as a directed graph of ten processing nodes organized into four phases, and a graph-based knowledge management system (Chapter 6), which enriches user requests with infrastructure context retrieved from an Extended Bill of Materials through hybrid BM25 and semantic search. The system was evaluated across five LLM backends of varying size and architecture on a set of 56 test cases spanning six categories (Chapter 7). The best-performing model achieved a 76.8% pass rate, demonstrating the feasibility of the approach while exposing systematic limitations that motivate the discussion that follows. This chapter summarizes the key results, examines the fundamental constraints of the prompt-engineering-based strategy adopted, and outlines directions for future research.

8.1 Summary of Results

The five models achieved pass rates between 66.1% (Ministral 8B) and 76.8% (Grok-4 Fast), with average semantic scores from 0.872 to 0.923 (Table 7.5). Action classification reached 100% accuracy across all models, confirming the robustness of the mapping from natural language to OpenC2 actions. Core fields such as protocol, response type, and actuator profile consistently exceeded 90% accuracy, indicating that the combination of action-dependent prompts, structured output schemas, and few-shot examples provides effective guidance for well-defined command fields. Pure description-based entity resolution was the strongest category (80 to 95% pass rate, Table 7.6), validating the hybrid retrieval architecture. In contrast, non-technical language proved the hardest category (43 to 64% pass rate), with several test cases failed by every model, all sharing the pattern of combining abstract business terminology with entity descriptions that require multi-hop semantic reasoning and disambiguation among similar candidates.

No single model dominated across all evaluation dimensions. Grok-4 Fast offered the highest pass rate at moderate latency (23.3s mean), while Ministral 8B delivered the fastest responses (4.9s mean) at a 10.7 percentage point accuracy cost. The destination address component (*dst_addr*, 80 to 94.3% accuracy) emerged as the primary error source across all models, with errors traceable to the RAG retrieval step: when entity descriptions are ambiguous or match multiple candidates, the wrong IP address may be selected. Additionally, Ministral 8B scored 0% on some parameters (*src_port*, *drop_process*, *duration_ms*), demonstrating that smaller models lack the capacity to handle constructs that appear infrequently in general pre-training data.

8.2 Prompt Engineering and Its Limitations

The prompt engineering strategy, composed of over 1,000 lines of domain-specific instructions with action-dependent routing, few-shot examples, and Pydantic-enforced output schemas, proved sufficient for well-structured technical requests: 100% action classification and up to 95% pass rate on description-based queries demonstrate that careful prompt design, combined with agentic decomposition, can handle a significant portion of the translation task. However, the 43 to 64% pass rate on non-technical language and the universally failed test cases reveal a systematic boundary: the necessary mappings between business concepts and technical entities are absent from the models' pre-training data. Similarly, Ministral8B's inability to correctly define parameters like *drop_process* and *duration_ms* shows that when a concept is underrepresented in the training corpus, no prompt can inject the missing knowledge. These findings align with related work: both Louro et al.[32] and Zhang et al.[34] showed that LoRA fine-tuning[33] significantly outperforms prompt-based approaches for security rule generation.

The results of this thesis, strongly suggest that prompt engineering alone is not a sufficient strategy for reliable security rule generation from natural language where technical language is lacking. While the graph workflow, structured output enforcement, and RAG-based enrichment provide a strong and reusable scaffolding, the LLM at the core would benefit from domain-specific fine-tuning that prompt engineering cannot substitute.

8.3 Future Work

The most impactful next step would be to fine-tune a compact model (7 to 14B parameters) using LoRA on a dataset of natural-language-to-OpenC2 command pairs. The existing evaluation infrastructure (Chapter 7) could serve as the validation set, while a larger training corpus could be constructed by augmenting these cases with paraphrases at varying levels of technicality, with a focus on the non-technical business language that currently represents the hardest category. The resulting fine-tuned model could be deployed on-edge without cloud API access, addressing both operational cost and data privacy concerns. The system's model-agnostic service

layer already supports seamless model switching, so integrating a fine-tuned model would require no architectural changes.

To address the retrieval problems, two complementary improvements could be tested to reduce the destination address errors that represent the main bottleneck: enriching entity descriptions in the Bill of Materials with business-context information, running services, and used ports to expand the similarity surface available to both BM25 and the semantic embedding model; and equipping the retrieval agent with an iterative loop capability to reformulate queries and cross-reference partial results until a confidence threshold is met. Further directions include expanding the test set for statistical robustness, replacing static prompt sections with direct references to Otupy documentation for easier maintenance, and validating the system within a real-world Security Operations Center workflow.

Appendix A

Result Analysis Appendix

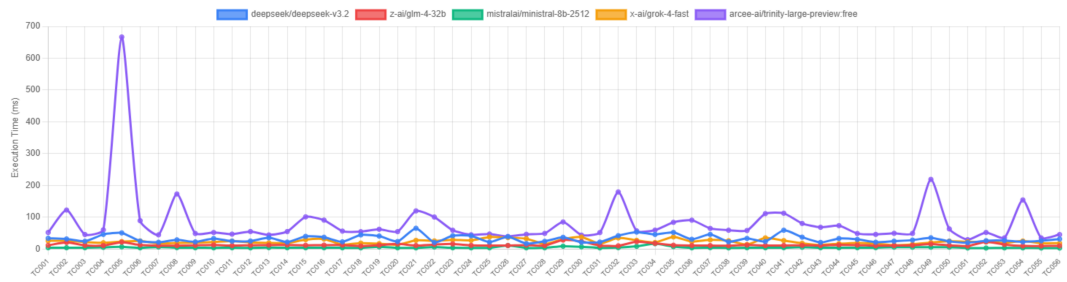


Figure A.1: Latency data for each test.

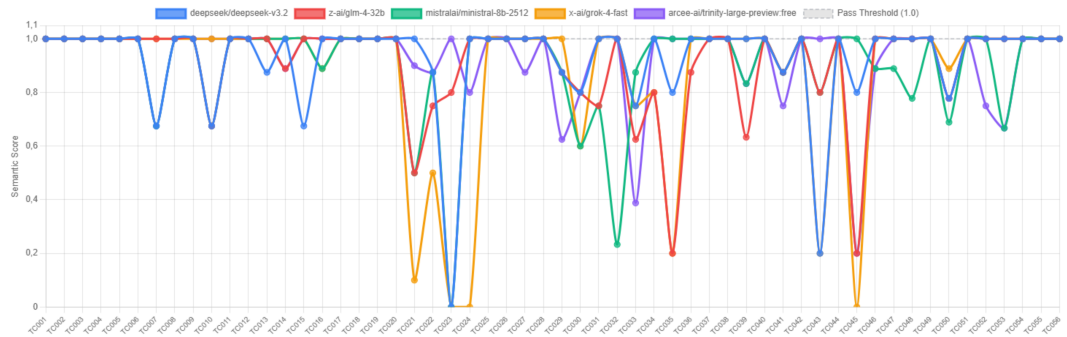


Figure A.2: Semantic score data for each test.

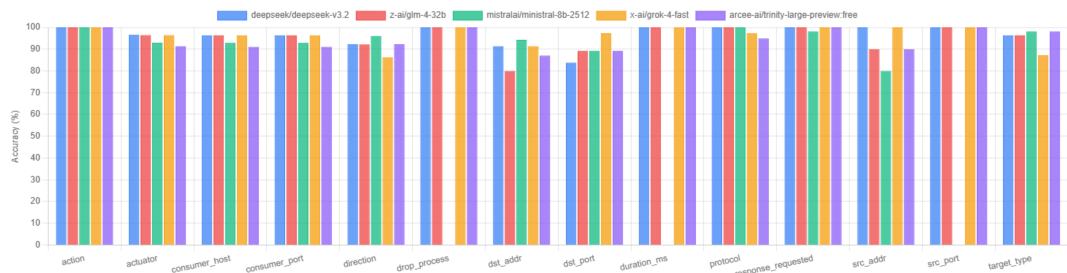


Figure A.3: Component accuracy data for each test.

Bibliography

- [1] ENISA. *ENISA Threat Landscape 2025*. Technical Report. European Union Agency for Cybersecurity, Oct. 2025. URL: <https://www.enisa.europa.eu/topics/threat-landscape> (cit. on p. 1).
- [2] Siddhant Srinivas, Brandon Kirk, Julissa Zendejas, Michael Espino, Matthew Boskovich, Abdul Bari, Khalil Dajani, and Nabeel Alzahrani. “AI-Augmented SOC: A Survey of LLMs and Agents for Security Automation”. In: *Journal of Cybersecurity and Privacy* (2025). DOI: 10.3390/jcp5040095. URL: <https://doi.org/10.3390/jcp5040095> (cit. on p. 1).
- [3] Amato Flora, Casola Valentina, Cozzolino Giovanni, De Benedictis Alessandra, and Moscato Francesco. “Exploiting Workflow Languages and Semantics for Validation of Security Policies in IoT Composite Services”. In: *IEEE Internet of Things Journal* 7.5 (2020), pp. 4655–4665. DOI: 10.1109/JIOT.2019.2960316 (cit. on p. 1).
- [4] Vasileios Mavroeidis and Joe Brule. “A nonproprietary language for the command and control of cyber defenses – OpenC2”. In: *Computers & Security* (2020). DOI: 10.1016/j.cose.2020.101999 (cit. on pp. 2, 6).
- [5] Matteo Repetto. “Otupy: A flexible, portable, and extensible framework for remote control of security functions”. In: *Computers & Security* 158 (2025). DOI: 10.1016/j.cose.2025.104597 (cit. on pp. 2, 5, 38, 42).
- [6] OpenC2 Committee. *OpenC2 Specification*. <https://openc2.org/specifications> (cit. on p. 5).
- [7] LangGraph Developers. *LangGraph: Building Agentic Applications with LLMs*. <https://docs.langchain.com/oss/python/langgraph/overview/> (cit. on pp. 5, 18).
- [8] CycloneDX Community. *CycloneDX Software Bill of Materials Standard*. <https://cyclonedx.org/>. 2024 (cit. on pp. 5, 11, 12, 38).
- [9] OASIS Open. *Open Command and Control (OpenC2) Profile for Stateless Packet Filtering Version 1.0*. <https://docs.oasis-open.org/openc2/oc2slpf/v1.0/oc2slpf-v1.0.html>. 2019 (cit. on pp. 7, 8).
- [10] Otupy Developers. *Otupy: OpenC2 Framework*. <https://github.com/mattereppe/otupy/> (cit. on p. 9).

- [11] Association for Supply Chain Management. *ASCM Supply Chain Dictionary*. <https://learn.ascm.org/>. 2024 (cit. on p. 10).
- [12] IBM. *What is an SBOM? Software Bill of Materials Explained*. <https://www.ibm.com/think/topics/sbom>. 2024 (cit. on p. 11).
- [13] Linux Foundation. *SPDX: Software Package Data Exchange*. <https://spdx.dev/>. 2024 (cit. on p. 11).
- [14] OWASP Foundation. *OWASP CycloneDX Authoritative Guide to SBOMs*. 2025. URL: https://cyclonedx.org/guides/OWASP_CycloneDX-Authoritative-Guide-to-SBOM-en.pdf (cit. on p. 12).
- [15] Tom B. Brown et al. “Language Models are Few-Shot Learners”. In: *CoRR* abs/2005.14165 (2020). arXiv: 2005.14165. URL: <https://arxiv.org/abs/2005.14165> (cit. on pp. 12, 13).
- [16] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. “Attention Is All You Need”. In: *CoRR* abs/1706.03762 (2017). arXiv: 1706.03762. URL: <http://arxiv.org/abs/1706.03762> (cit. on p. 13).
- [17] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. “BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding”. In: *CoRR* abs/1810.04805 (2018). arXiv: 1810.04805. URL: <http://arxiv.org/abs/1810.04805> (cit. on p. 13).
- [18] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. “Language models are unsupervised multitask learners”. In: *OpenAI blog* (2019). URL: https://cdn.openai.com/better-language-models/language_models_are_unsupervised_multitask_learners.pdf (cit. on p. 13).
- [19] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Ed H. Chi, Quoc Le, and Denny Zhou. “Chain of Thought Prompting Elicits Reasoning in Large Language Models”. In: *CoRR* abs/2201.11903 (2022). arXiv: 2201.11903. URL: <https://arxiv.org/abs/2201.11903> (cit. on p. 13).
- [20] Wayne Xin Zhao et al. “A Survey of Large Language Models”. In: (2026). arXiv: 2303.18223 [cs.CL]. URL: <https://arxiv.org/abs/2303.18223> (cit. on p. 13).
- [21] Deepak Suresh Asudani, Naresh Kumar Nagwani, and Pradeep Singh. “Impact of word embedding models on text analytics in deep learning environment: a review”. In: *Artificial Intelligence Review* 56.9 (Sept. 2023), pp. 10345–10425. DOI: 10.1007/s10462-023-10419-1 (cit. on p. 13).
- [22] Nils Reimers and Iryna Gurevych. “Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks”. In: 2019. arXiv: 1908.10084 [cs.CL]. URL: <https://arxiv.org/abs/1908.10084> (cit. on p. 14).

- [23] Tyler Thomas Procko and Omar Ochoa. “Graph Retrieval-Augmented Generation for Large Language Models: A Survey”. In: *2024 Conference on AI, Science, Engineering, and Technology (AIxSET)*. IEEE, 2024, pp. 166–169. DOI: 10.1109/AIxSET62544.2024.00030 (cit. on p. 14).
- [24] Aditi Singh, Abul Ehtesham, Saket Kumar, and Tala Talaei Khoei. “Enhancing AI Systems with Agentic Workflows Patterns in Large Language Model”. In: *2024 IEEE World AI IoT Congress (AIIoT)*. IEEE, 2024, pp. 527–533. DOI: 10.1109/AIIoT61789.2024.10578990 (cit. on p. 15).
- [25] Erik Schluntz and Barry Zhang. *Building Effective AI Agents*. Anthropic Engineering Blog. Dec. 2024. URL: <https://www.anthropic.com/engineering/building-effective-agents> (cit. on pp. 15–18).
- [26] OpenAI. *A Practical Guide to Building Agents*. Business Guide. Apr. 2025. URL: <https://cdn.openai.com/business-guides-and-resources/a-practical-guide-to-building-agents.pdf> (cit. on pp. 15, 18).
- [27] LangChain. *LangGraph*. LangChain Blog. Jan. 2024. URL: <https://www.blog.langchain.com/langgraph/> (cit. on pp. 18–20).
- [28] Sani Lawal, Xiang Zhao, Antony Rios, Ram Krishnan, and David Ferraiolo. “Translating Natural Language Specifications into Access Control Policies by Leveraging Large Language Models”. In: *2024 IEEE 6th International Conference on Trust, Privacy and Security in Intelligent Systems and Applications (TPS-ISA)*. Washington, DC, USA: IEEE, Oct. 2024, pp. 361–370. DOI: 10.1109/TPS-ISA62245.2024.00048 (cit. on p. 23).
- [29] Fabio Martinelli, Francesco Mercaldo, Luca Petrillo, and Antonella Santone. “Security Policy Generation and Verification through Large Language Models: A Proposal”. In: *Proceedings of the Fourteenth ACM Conference on Data and Application Security and Privacy*. CODASPY ’24. New York, NY, USA: ACM, June 2024, pp. 195–197. ISBN: 9798400704215. DOI: 10.1145/3626232.3658635. URL: <https://doi.org/10.1145/3626232.3658635> (cit. on p. 23).
- [30] Fagan Taghiyev et al. “Natural Language Interface for Firewall Configuration”. In: *arXiv preprint arXiv:2512.10789* (Dec. 2025). URL: <https://arxiv.org/abs/2512.10789> (cit. on p. 23).
- [31] Irene Gosálvez-White, Nerea Rodríguez-Martín, Nicolás Barajas-García, Carmen Mena-Gallardo, and Pedro García-Teodoro. “MEDIotWALL: Securing Smart Healthcare Environments Through IoT Firewalls”. In: *Sensors* 25.19 (2025), p. 6235. DOI: 10.3390/s25196235 (cit. on p. 24).
- [32] Bernardo Louro, Raquel Abreu, Joana Cabral Costa, João B F Sequeiros, and Pedro Ricardo M Inácio. “Analysis of the Capability and Training of Chat Bots in the Generation of Rules for Firewall or Intrusion Detection Systems”. In: *Proceedings of the 19th International Conference on Availability, Reliability and Security*. ACM, 2024. DOI: 10.1145/3664476.3670902 (cit. on pp. 24, 74).

- [33] Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. “LoRA: Low-Rank Adaptation of Large Language Models”. In: *International Conference on Learning Representations (ICLR)*. 2022. URL: <https://arxiv.org/abs/2106.09685> (cit. on pp. 24, 74).
- [34] Chi Zhang, Muhammad Shadi Hajar, Harsha Kalutarage, and Lankeshwara Munasinghe. “Parameter-Efficient Fine-Tuning of LLMs for Intrusion Detection and Firewall Rule Generation: A Comparative Study”. In: *Security and Trust Management (STM) 2025*. Springer, 2025 (cit. on pp. 24, 74).
- [35] Chi Zhang, Muhammad Shadi Hajar, Harsha Kalutarage, and Lankeshwara Munasinghe. “Parameter-Efficient Fine-Tuning of LLMs for Intrusion Detection and Firewall Rule Generation: A Comparative Study”. In: (2026). Ed. by Sokratis Katsikas and Cristina Alcaraz, pp. 83–98. DOI: 10.1007/978-3-032-06155-3_5 (cit. on p. 24).
- [36] Gordon V. Cormack, Charles L. A. Clarke, and Stefan Buettcher. “Reciprocal Rank Fusion Outperforms Condorcet and Individual Rank Learning Methods”. In: *Proceedings of the 32nd International ACM SIGIR Conference on Research and Development in Information Retrieval*. ACM, 2009, pp. 758–759. DOI: 10.1145/1571941.1572114 (cit. on pp. 39, 46).
- [37] sentence-transformers. *paraphrase-multilingual-MiniLM-L12-v2*. <https://huggingface.co/sentence-transformers/paraphrase-multilingual-MiniLM-L12-v2>. 2021 (cit. on p. 41).

Dedications

Dedications for everyone