

Politecnico di Torino

Master Degree in Cybersecurity

Master Degree Thesis

Cloud Digital Twin for automated policy and vulnerability management

Supervisors
prof. Fulvio Valenza
prof. Daniele Brighenti
prof. Riccardo Sisto
dott. Francesco Pizzato

Candidate
Andrea Gnani

ACADEMIC YEAR 2025–2026

This work is subject to the Creative Commons License

Summary

Kubernetes environments are characterized by their extremely dynamic configurations, where workloads, access rules, and communication policies change throughout the system lifecycle. It's clear that security checks based on static snapshots in this context quickly become obsolete, with the risk of representing an outdated state and failing to quickly detect changes that lead the cluster to unsafe configurations.

This thesis presents the extension of an existing framework for the formal verification of Kubernetes security invariants to a runtime solution based on the paradigm of the Digital Twin. By observing the relevant events and changes exposed by the Kubernetes API Server, a graph-based model is constructed and updated through a normalization phase. This model describes resources, workloads, identities, and security-relevant relationships, with a particular focus on RBAC and Network Policy mechanisms.

The resulting Digital Twin becomes a safe environment for continuous verification of security invariants, orchestrated by the Verification Runner, which integrates the current state of the graph with a set of queries and conducts the analysis with the help of the existing Z3-based SMT solver. Once the checks are complete, the results are modelled into structured information, highlighting the invariants satisfied and the violated properties, then providing useful data for understanding the context.

Additionally, a remediation module for RBAC and Network Policy violations has been developed. Remediation strategies are contained in a catalog from which they are selected based on criteria derived from the context of the violation. Their application is initially evaluated in the virtual scenario of the Digital Twin, and the new candidate configuration is subjected to formal verification. This step allows the Digital Twin's emulation and simulation capabilities to be fully exploited without encountering risks in the real environment and pushing the cluster toward potentially unsafe configurations. A Kubernetes-side dry-run check follows to confirm that the proposed changes are feasible in the real environment and, finally, the application to the cluster requires explicit user confirmation. The fix causes the system to observe and acquire, thanks to the watcher, the new state and verify its compliance with the security invariants. The cycle ends when it is determined that the violation has been effectively resolved.

The prototype is matched with a dashboard-style UI to provide the user with a clear view of the Digital Twin model. This allows all real-time events, the cluster's resource and relationship graph, the status of the invariants, any violations, and proposed remediations to be visible and examinable.

Overall, the aim of this thesis is to lay the foundation for understanding the application and utility of a Digital Twin and to delve deeper into the dynamics of Kubernetes environments. It also demonstrates the feasibility of a continuous monitoring, verification, and remediation cycle, keeping security analysis aligned with the cluster's runtime evolution.

Further, the proof of concept allows us to evaluate the functioning of the proposed solution, while the conclusions and future works discuss the limitations of the current prototype and possible future evolutions towards more scalable and automated verification and remediation.

Contents

Summary 2

List of Figures 9

List of Tables 11

1 Introduction 12

1.0.1 Growth of Cloud-Native Systems . . . 12

1.0.2 Kubernetes Security Complexity . . . 13

1.0.3 Limits of Static Approaches . . . 13

1.0.4 Need for Continuous Assurance . . . 14

1.1 Thesis Objectives . . . 15

1.2 Thesis Description . . . 16

2 Background: Digital Twin 18

2.1 Introduction and Definitions of the Digital Twin . . . 18

2.1.1 Origin of the Term and Historical Development . . . 18

2.1.2 Main Definitions . . . 19

2.1.3 The Concept of Twinning . . . 20

2.2 Taxonomy and Classification of Mirror Systems . . . 20

2.2.1 Levels of Data Integration . . . 20

2.2.2 Types of Digital Twins Across the Lifecycle . . . 21

2.2.3 Maturity Models . . . 22

2.3 Digital Twin Architecture . . . 22

2.3.1 Fundamental Components . . . 22
2.3.2 Layered Models . . . 23
2.3.3 Standardization and Frameworks . . . 23
2.4 Modeling and Simulation Methodologies . . . 24
2.4.1 Physics-Based Modeling . . . 24
2.4.2 Data-Driven Modeling . . . 25
2.4.3 Hybrid and Graph-Based Approaches . . . 25
2.5 Digital Twin in the Product Lifecycle . . . 26
2.5.1 Beginning-of-Life . . . 26
2.5.2 Middle-of-Life . . . 27
2.5.3 End-of-Life . . . 27
2.6 Enabling Technologies . . . 28
2.6.1 IoT and Sensing Technologies . . . 28
2.6.2 Cloud and Edge Computing . . . 29
2.6.3 AI, Machine Learning, and LLMs . . . 29
2.6.4 Blockchain . . . 30
2.7 Main Application Domains . . . 30
2.7.1 Manufacturing and Industry 4.0 . . . 30
2.7.2 Critical Infrastructures and Energy Systems . . . 31
2.7.3 Healthcare and Smart Cities . . . 31
2.7.4 Focus on Cybersecurity . . . 32
2.8 Challenges, Limitations, and Future Directions . . . 32
2.8.1 Implementation Cost and Scalability . . . 32
2.8.2 Data Quality and Interoperability . . . 33
2.8.3 Security and Privacy . . . 33

3 Kubernetes, RBAC and Network Policies 35

3.1 Kubernetes Overview . . . 35
3.1.1 Kubernetes and Container Orchestration . . . 35
3.1.2 Kubernetes Cluster Architecture . . . 37

3.1.3 Declarative Management and Reconciliation . . . 38
3.2 Kubernetes Resources and Abstractions . . . 40
3.2.1 Namespaces . . . 40
3.2.2 Pods and Containers . . . 40
3.2.3 Workload Resources . . . 41
3.2.4 Services . . . 41
3.2.5 Labels and Selectors . . . 41
3.2.6 ConfigMaps, Secrets, and ServiceAccounts . . . 42
3.3 Kubernetes Security Model . . . 42
3.3.1 Main Security Areas . . . 42
3.4 Authorization and Role-Based Access Control . . . 43
3.4.1 RBAC Overview . . . 44
3.4.2 Roles and ClusterRoles . . . 45
3.4.3 RoleBindings and ClusterRoleBindings . . . 46
3.4.4 Effective Permissions . . . 47
3.4.5 RBAC Security Risks . . . 48
3.4.6 RBAC Representation in the Thesis . . . 49
3.5 Kubernetes Network Policies . . . 50
3.5.1 Kubernetes Networking Model . . . 50
3.5.2 Network Policy Overview . . . 51
3.5.3 Pod Selection and Policy Scope . . . 52
3.5.4 Ingress and Egress Rules . . . 53
3.5.5 Network Policy Peers . . . 53
3.5.6 Network Policy Security Risks . . . 54
3.5.7 Network Policies Representation in the Thesis . . . 55

4 Thesis Objectives **56**

5 The Proposed Approach **58**

5.1 Digital Twin for Kubernetes Security . . . 58

5.1.1 Mapping the Kubernetes Cluster to a Digital Twin 59
5.1.2 Synchronization Between the Cluster and the Twin 60
5.2 Security-Oriented Role of the Digital Twin 61
5.3 Architectural Organization of the System 62
5.3.1 Architectural Boundaries and Responsibility Distribution 64
5.4 Security-Relevant Projection of the Kubernetes State 66

6 Implementation of the Runtime Digital Twin 69
6.1 Software Components and Implementation Environment 70
6.2 Kubernetes Event Watcher 71
6.2.1 Resource Configuration, API Integration and Stream Management . 71
6.2.2 Event Normalization 73
6.3 Runtime State Management and Graph Materialization 75
6.3.1 In-Memory Object Store 75
6.3.2 Cluster Parsing and Graph Reconstruction 76
6.3.3 Materialized Digital Twin Files 77
6.4 Runtime Verification and Violation Detection 79
6.4.1 Verification Triggering and Query Set Management 79
6.4.2 Security Invariant Representation 81
6.4.3 Verification Runner and Solver Integration 83
6.4.4 Verification Report and Violation Generation 84
6.4.5 RBAC and Network Policy Query Examples 85

7 Remediation Engine 87
7.1 Objectives and Design Principles 87
7.2 Remediation Engine Architecture and Software Components 88
7.2.1 Remediation Runner and Internal Modules 89
7.2.2 Runtime Files and Data Flow 90
7.3 Violation Context and Remediation Candidate Selection 91
7.3.1 Violation and Invariant Correlation 91

7.3.2 Playbook Catalog and Candidate Filtering . . . 92
7.4 Remediation Strategies for RBAC and NetworkPolicy . . . 93
7.4.1 RBAC Remediation Strategies . . . 94
7.4.2 NetworkPolicy Remediation Strategies . . . 95
7.5 Digital Twin What-If Scenario and Persisted SMT Precheck . . . 97
7.5.1 Temporary Candidate Digital Twin . . . 97
7.5.2 Baseline and Candidate Verification . . . 98
7.5.3 Persisted Precheck and Protection Against Stale Decisions . . . 99
7.6 Safe Application to the Kubernetes Cluster . . . 100
7.6.1 User Confirmation and Kubernetes Server-Side Dry-Run . . . 101
7.6.2 RBAC Apply Handlers . . . 101
7.6.3 NetworkPolicy Apply Handler . . . 102
7.6.4 Watcher Synchronization and Post-Remediation Verification . . . 103
7.7 Auditability and Current Boundaries . . . 104

8 Proof of Concept **106**
8.1 Validation Objectives and Experimental Setup . . . 106
8.1.1 Initial Cluster State and Digital Twin Visualization . . . 107
8.1.2 Runtime Synchronization and Event Propagation . . . 110
8.1.3 RBAC Security Scenario . . . 113
8.1.4 NetworkPolicy Security Scenario . . . 118

9 Conclusions **122**
9.1 Future Works . . . 124

Bibliography **125**

List of Figures

5.1 Architecture of the Kubernetes Digital Twin model. 62

6.1 Runtime verification pipeline. 79

8.1 Initial cluster state displayed in the Overview section of the dashboard. . . 108

8.2 Resource detail panel opened from the Overview graph. 109

8.3 RBAC overlay in the Overview section and detail panel for a selected RBAC relation. 110

8.4 NetworkPolicy overlay in the Overview section and detail panel for a selected network relation. 110

8.5 ADDED Deployment event displayed in the Events section. 111

8.6 Detail panel for the sync-demo Deployment event. 112

8.7 DELETED event generated after removing the sync-demo Deployment. . . 112

8.8 RBAC security invariant in Passing state before the Role modification. . . 114

8.9 MODIFIED event generated by the dangerous-role update. 114

8.10 RBAC security invariant in Failing state after adding the delete verb. . . . 115

8.11 RBAC violation shown in the Detection section. 116

8.12 Digital Twin precheck for the selected RBAC remediation candidate. . . . 117

8.13 Remediation view after resolution of the active RBAC violation. 117

8.14 RBAC security invariant returned to Passing after post-verification. 118

8.15 Cross-namespace connection visible in the Overview during the NetworkPolicy violation. 119

8.16 NetworkPolicy violation shown in the Detection section. 120

8.17 Digital Twin precheck for the selected NetworkPolicy remediation candidate. 121

8.18 Overview after application and post-verification of the NetworkPolicy remediation. 121

List of Tables

6.1 Kubernetes API groups, Python clients, and watched resources. 72
6.2 Materialized Digital Twin files and their main content. 78
6.3 Examples of semantically relevant fields compared for MODIFIED events. . 80
6.4 Fields used to represent security invariants. 82

7.1 Main software components of the Remediation Engine. 90
7.2 Information used to build the violation context. 92
7.3 Remediation playbook catalog. 94

Chapter 1

Introduction

1.0.1 Growth of Cloud-Native Systems

Nowadays, cloud-native systems play a crucial role in modern digital infrastructures, as they are useful to build applications that can scale, change, and recover faster than traditional systems. This shift has changed the way applications are designed and deployed, but it has also introduced novel security-related problems. A cloud-native environment includes several interconnected layers, like containers, orchestration platforms, services, and applications. Each component brings in specific risks, that requires dedicated security controls [2], [3].

Traditional security models were built for slow-changing systems. In those environments, workloads were more stable, network boundaries were easier to define, and communication paths changed less often. Security teams could count on perimeter firewalls, static IP rules, and host-based controls to protect most of the assets [1]. In cloud-native systems, these assumptions no longer hold.

The attack surface increases because many distributed components interact with each other. Microservices exchange data through APIs, workloads start and stop on demand and infrastructure changes thanks to automated tools rather than manual operations [1], [2]. This makes it harder to understand which components communicate, which permissions they have, and which paths an attacker could follow.

Kubernetes perfectly fits into this framework, as it manages distributed workloads and updates the cluster state at runtime. For this reason, security cannot depend only on static checks or fixed network boundaries. It needs mechanisms that can observe the current state of the system and react properly when that state mutates [1], [2].

1.0.2 Kubernetes Security Complexity

Kubernetes is recognized today as one of the main platforms for deploying cloud-native applications. The CNCF (Cloud Native Computing Foundation) reported that production adoption reached 80% in 2024, while overall usage, including pilot environments, exceeded 90% [1]. This wide adoption makes Kubernetes a key element for a lot of modern software infrastructures.

One of the main security challenges in Kubernetes arises from its dynamic behavior. Pods can be created, deleted, moved, or replaced during normal cluster operations [2], [3]. A pod may exist for a short time, receive a temporary network identity, and then disappear or move to another node. This behavior differs from traditional infrastructures, where servers and network addresses were more stable.

Kubernetes also depends on API-driven management and service-to-service communication. Applications often are made of multiple microservices that interact within the cluster. This creates dependency chains that are not always easy to inspect [1], [2]. Each communication path can become a potential attack vector, especially when permissions or network policies are too broad.

The use of Infrastructure as Code and CI/CD (Continuous Integration / Continuous Deployment) pipelines adds another layer of complexity, as they introduce frequent and automated changes across different levels of the cluster configuration.

1.0.3 Limits of Static Approaches

Static security approaches have clear limits in Kubernetes environments. They can detect some configuration issues, but they cannot always follow the flows of changes in the cluster [1], [2]. Many tools analyze single resources or fixed snapshots, without considering how workloads, identities, permissions, and rules interact with each other [2], [3].

For example, a traditional firewall focuses on the network perimeter. This gives limited visibility into pod-to-pod traffic inside the cluster [1], [2]. In Kubernetes, many important communications happen internally, between services that may change over time. The previously mentioned traditional firewall would miss these internal paths.

Static analysis can show that a configuration looks risky, but it cannot always determine if that risk is active at runtime. A permission may appear dangerous in a file, but its real impact depends on the subject that receives it, the resources available, and other enforced policies.

Some of these problems, such as network reachability, access-control policies, and container network configurations, have already been faced with the aid of formal verification

methods [61]. The first that comes to mind is Verikube, which verifies container network policies using a graph-based representation, built to make complex configurations analysis more efficient [60]. Another example, regarding access control, is the framework proposed by Sissodiya et al. [54], that models Kubernetes RBAC and admission policies with first-order logic and exploits an SMT solver to detect configurations where expected security properties are violated. Both cases highlight the same core point: formal models bring out effect and inconsistencies between policies that manual inspections would hardly recognize.

Another problem is the separation between configuration analysis and runtime monitoring. A lot of tools see these activities as independent tasks [2], [3], and this creates partial visibility. The system might be aware of a configuration exists, but not of its runtime behavior. On the other hand, it may observe runtime events without linking them to the structure of roles, bindings, workloads, and network policies. This makes the evaluation of the real security posture of the cluster harder.

1.0.4 Need for Continuous Assurance

Kubernetes' security demands a view of the cluster as a connected system, instead of a set of separate resources. Workloads, identities, permissions, network policies, and runtime events influence each other. A correct security analysis must consider these relations together at any moment [2], [3].

Misconfigurations remain a major source of risk in Kubernetes clusters [2], [3]. Examples include excessive RBAC privileges, weak network isolation, and over-permissive policies. If an attacker compromises one workload, these weaknesses make room for lateral movement across the cluster [1], [2].

Traditional monitoring also struggles with Kubernetes because resources do not always use fixed endpoints. Workloads communicate through service discovery, DNS, and API calls [1]. Since pods and services can change during normal operations, it becomes difficult to maintain a consistent view of identities, permissions, and communication flows.

For these reasons, Kubernetes security needs continuous assurance. This means observing the runtime state of the cluster while also analyzing the structural relations between its resources [3], [4]. A runtime-aware approach can connect events, configurations, permissions, and network behavior in a single model. This supports faster detection of security issues and gives a more accurate view of the current security posture of the cloud-native environment.

1.1 Thesis Objectives

Cloud-native systems have changed the way organizations deploy, manage, and secure applications. In these environments, tenants have little control over deployment and execution monitoring. This inhibits the enforcement of security policies and might permit some unnoticed actions [5]. Traditional security approaches do not address this problem well, because they were designed around fixed physical boundaries that no longer apply in cloud-native infrastructures [6].

Security also becomes more difficult to define during the design phase. Modern software lifecycles depend on automation, on CI/CD pipelines, and on programmable infrastructure [7]. As a consequence, configurations may frequently vary at different system levels. Intent-driven approaches, for example, have been proposed to automate the translation of high-level network isolation requirements into enforceable security configurations in Kubernetes and across distributed cloud environments [6], [67]. This increases the risk of operations going unintentional and makes integrity verification more complex [5]. So, a complete security analysis must take into account the whole stack and the relationships between its components [2]. These relations are relevant because interactions in the infrastructure, orchestration, identities, workload and network policies make the attack surface wider [2].

This thesis addresses these limitations by **extending an existing SMT-based verification framework for Kubernetes security configurations**, focused on **RBAC (Role-Based Access Control)** and **Network Policy**, toward a **runtime Digital Twin**. The current framework verifies cluster security by analyzing static snapshots. This approach is useful, but it does not fully match the behavior of real Kubernetes clusters. In practice, clusters change because of controller actions, scaling operations, deployment updates, and administrative changes.

The goal of this thesis is to design and implement a Digital Twin that mirrors the Kubernetes cluster state and keeps an updated model of its security configuration. The model supports continuous monitoring and formal reasoning over time. When the cluster changes, the Digital Twin updates its internal representation and re-evaluates predefined security invariants. These invariants can include privilege escalation prevention, namespace isolation, and the absence of unintended communication paths.

The thesis also considers **remediation**. When the system detects a violation, it uses information produced by the solver to identify the cause of the problem. Then, it computes corrective actions that aim to restore compliance while limiting changes to the cluster configuration.

Cloud-native systems do not remain fixed after deployment. In a Kubernetes cluster,

services can be created, updated, scaled, moved, or removed as part of normal operation [8]. For this reason, a security analysis based only on static snapshots can become outdated as soon as the cluster changes. The proposed approach treats the cluster as a dynamic system. It collects configuration data and runtime changes and uses them to keep the formal model aligned with the live cluster.

The system also supports **incremental verification**. When a relevant event occurs, the model is updated instead of rebuilding the whole analysis from the beginning. This allows the Digital Twin to preserve an updated view of the cluster's security state. The goal is to check that security properties such as access control, namespace isolation, and communication constraints remain valid during the lifecycle of the Kubernetes environment.

1.2 Thesis Description

Chapter 2: DT Background introduces the concept of Digital Twin. It presents its origin, main definitions and how it has evolved over time. Then it puts the concept into cloud-native systems context and explains how a Digital Twin can monitor and represent distributed environments like a Kubernetes cluster.

Chapter 3: Kubernetes, RBAC and Network Policies deals with Kubernetes basic knowledge, needed for this thesis. It describes main architectural elements and Kubernetes fundamental abstractions. The chapter then focuses on two important security mechanisms: RBAC and Network Policies. It explains how permissions and network communication rules influence the security posture of a cluster, especially in dynamic cloud-native environments.

Chapter 4: Thesis Objective defines the thesis's aim. It exposes the motivation that drove the work and lists the main goals of the proposed approach. The chapter also shows the shift from a static verification to a runtime-aware solution based on a Digital Twin model.

Chapter 5: The Proposed Approach introduces the approach adopted to extend static Kubernetes security verification through a runtime-aware Digital Twin. It explains how the Kubernetes cluster is mapped to its virtual counterpart, how synchronization between the two environments is maintained, and how the Digital Twin supports continuous security verification. The chapter also presents the overall system architecture, describing its main components, their responsibilities, and how they interact with each other.

Chapter 6: Implementation of the Runtime Digital Twin describes the implementation of the system. It presents the technologies used, the integration with Kubernetes, and the mechanisms used to build and update the verification model. It also highlights

how cluster events are collected, normalized, and used to maintain the Digital Twin.

Chapter 7: Remediation Engine introduces the remediation engine. It explains how the system detects violations of security invariants and how it generates corrective actions. The chapter focuses on remediation strategies that restore compliance while reducing the impact on the running system.

Chapter 8: Proof-of-Concept presents the proof of concept used to validate the proposed approach. It describes the experimental setup, the tested scenarios, and the evaluation criteria. It also discusses the results in terms of effectiveness and performance.

Chapter 9: Conclusions and Future Work concludes the thesis. It summarizes the main contributions of the work and discusses possible future developments. These include scalability, stronger automation, and deployment in real Kubernetes environments.

Chapter 2

Background: Digital Twin

2.1 Introduction and Definitions of the Digital Twin

The Digital Twin concept plays an important role in the context of the Industry 4.0 and digital transformation processes. The core idea is creating a digital representation of a real entity, and employing it in observation, analysis, simulation and support for decisions about the represented system [11], [18]. Generally, a Digital Twin is seen as a virtual counterpart of an object, a process or a physical system. This representation is updated over time through data collected by the real system and can support several activities, such as monitoring, prediction, **emulation** and decision-making [9].

The use of Digital Twin does not concern just one sector. This technology finds application in manufacturing, aerospace, healthcare, smart city, energy and other fields characterized by complex systems that require constant supervision. Their goal is facilitating system management, detecting faults, anticipating possible malfunctions and supporting data-based decisions [18]. For this reason, Digital Twins are often associated to the development of cyber-physical systems and data-driven infrastructures.

2.1.1 Origin of the Term and Historical Development

The term Digital Twin is quite recent, but the fundamental idea is way older. Some of the first examples can be traced back to the aerospace sector, with a particular attention to Apollo missions of NASA in the 60's. At that time, engineers employed physical replicas of spacecrafts on Earth to simulate mission's conditions and evaluate potential problems in the middle of flight operations. [9], [11]. These replicas were not digital models yet, but they already expressed the same principle: keeping a real system counterpart to support observation, diagnosis and decisions.

As physical prototypes became more expensive and complex, virtual models turn out to be a useful alternative and complement to material replicas [9]. The modern Digital Twin concept is usually associated to **Michael Grieves**, who introduced it in 2002in the field of PLM (Product Lifecycle Management). Here, the Digital Twin pointed out the link between a physical product and its informative representation across the entire lifecycle [9], [24].

In the following years, similar concepts were proposed under different names, such as Mirrored Spaces Model and Information Mirroring Model [9]. The term Digital Twin was later adopted and formalized in the aerospace sector by John Vickers at NASA in 2010, helping its diffusion in both research and industry [9], [20].

2.1.2 Main Definitions

Although the Digital Twin concept is now widely used, there is still no single definition accepted across all sectors. This depends on the fact that the paradigm is applied in different domains, and each domain focuses on specific aspects. In some cases, the focus is on simulation; in others, it is on synchronization with the real system, lifecycle management, data integration, or decision support [9], [20].

One of the most relevant definitions comes from NASA. In the aerospace field, the Digital Twin is understood as a high-fidelity simulation of a system, capable of integrating physical models, sensor data, operational data, and historical knowledge to reproduce the behavior and evolution of its physical counterpart [9], [20], [31]. This interpretation highlights three central elements: representation accuracy, continuous data-driven updates, and lifecycle monitoring.

From the point of view of manufacturing and PLM, Grieves and Vickers depict the Digital Twin as a set of virtual information constructs representing a physical product. In this perspective, the twin value resides in its informational role: it lets describe the product at different levels, from design data to information gathered from the operational use [9], [18].

A broader definition has been proposed by the Digital Twin Consortium, which considers the DT as a virtual representation of entities and real processes, synchronized at a certain frequency and with a precise fidelity level [18], [24]. This definition is useful because it clarifies that not all Digital Twins have to show the same degree of precision and synchronization. Some scenarios require frequent updates and accurate data, while others may work also with slower updates or at a lower level of detail.

The World Wide Web Consortium instead adopts an infrastructure-oriented perspective. In this case, the Digital Twin is seen as a virtual representation of a device, or a group of

devices, deployed in a cloud or edge environment. In this context, the twin turns into a useful tool to simulate and test possible actions before the effective application to the real system [18].

2.1.3 The Concept of Twinning

The element that tells a Digital Twin and traditional simulation apart is the concept of *twinning*. A simulation can replicate a system's behavior based on pre-defined hypothesis but then, once the model is in place, it is usually disconnected from the real system. On the other hand, a Digital Twin holds a continuous collection with its real counterpart thanks to data streams. These enable virtual representation updates as the real system changes [9], [19].

This synchronization might exploit data coming from sensors, logs, controllers, monitoring systems or IoT devices. Therefore, the virtual model doesn't stay static but keeps evolving in time, following the real system's state. The DT is then able to depict operating conditions, environment variations, components degradation and other factors that influence the behavior of the system [18], [24].

That is why the Digital Twin is often described as a "living" model. Its value does not depend on its representation ability of the system's ongoing state, but on enabled analysis of future or hypothetical scenarios. The twin can be used for testing alternative configurations, studying eventual malfunctions and evaluating the impact of specific changes without exposing the real system to unnecessary risks [9]. In this sense, the Digital Twin works both as an observation tool and as a decision-support tool.

2.2 Taxonomy and Classification of Mirror Systems

Not all digital representations of a real system can be strictly qualified as Digital Twin. The difference lays on the data exchange between the real system and its digital counterpart, the synchronization frequency and whether the virtual model affects the physical environment or not [18], [19], [24]. For this reason, literature identifies different "mirror" system categories, able to reflect a physical reality.

2.2.1 Levels of Data Integration

Although it has received strong critics by Michael Grieves for its conceptual flaws, an adopted classification is based on data integration level between physical space and virtual one. On this criterion, literature identifies Digital Model, Digital Shadow and Digital Twin

[18], [19].

Digital Model is the simplest form of digital representation. The physical object and the virtual model are not connected through an automatic exchange of data, but every change that occurred has to be manually inserted and real system changes do not reflect on digital space if there's no user intervention [11], [18], [19]. So, a Digital Model may be useful for design activities, offline analysis and documentation, but it cannot offer an updated view of the real system's state.

Digital Shadow introduces a major level of integration. Here data flows automatically from physical to digital environment. The virtual model updates itself following real world changes, but the flow stays unidirectional [11], [18], [19]. The digital counterpart can observe and represent the physical entity but cannot send commands or actions backward [18], [27].

Digital Twin provides the most advanced integration. It is based on an automatic exchange of data in both ways: the virtual model receives information from the physical system and, according to the application, is able to send back decisions, recommendations and control actions [11], [18], [19]. This bidirectional flow enables automation, optimization, monitoring and system-level control [19], [24].

2.2.2 Types of Digital Twins Across the Lifecycle

Digital Twins can also be classified based on the lifecycle stage in which they are used. A system mutates over time: from the design phase, going through operation and finally dismission. The twin can then assume various forms and support different tasks across its lifecycle [19], [27].

The **Digital Twin Prototype** is employed before production or in the design stage. It includes requirements, expected behavior and essential information to come up with the physical product [9], [19]. Its main functions are analysis of support and validation before the real entity is created.

The **Digital Twin Instance** represents a given physical product during its operational phase. Unlike the prototype, it is bonded to an already existing asset, and it's used for monitoring, maintenance, and performance analysis according to data collected while running [9], [11].

The **Digital Twin Aggregate** combines information originating from multiple Digital Twin Instances. This is useful for situations where the focus shifts from monitoring a single element to analyzing a set of systems or interconnected components. It allows for large-scale analysis, system-level optimization and comparison between instances [9], [11].

2.2.3 Maturity Models

Digital Twin systems can also be described through maturity models. These models focus on the level of autonomy, intelligence, and integration reached by the twin [24].

At the lowest level there is the **Pre-Digital Twin**, which consists of a virtual model or simply a prototype. It can be used for simulations and risks analysis, but it's not linked to an active physical counterpart yet [18], [24].

The **Adaptive Digital Twin** introduces more flexible interfaces and visual mechanisms. The peculiarity here is that the system can adapt the way information is shown based on user needs or interactions. This helps in interpreting the state of the system and in a much more conscious decision-making [24].

The **Intelligent or Cognitive Digital Twin** extends these features by integrating AI/ML techniques. The twin now goes beyond mere visualization or data monitoring. It recognizes patterns, supports predictions and contributes to autonomous decisional processes [12], [24].

The **Federated Digital Twin** brings in a more complex structure, where many DTs are connected. This enables data exchange and coordination across domains or organizations. The result is a wider view of heterogeneous and distributed environments [12], [24].

2.3 Digital Twin Architecture

Digital Twin architectures do not rely on a fixed standard. Their structure will likely depend on the application domain, the portrayed system, the required level of synchronization and the kind of analysis the twin must carry on. Despite these differences, many architectures share the same fundamental principle: bonding a real entity to its virtual representation through a data exchange infrastructure. The digital view is then able to monitor the system, perform simulations, analyze behavior and support decisions.

2.3.1 Fundamental Components

A Digital Twin is usually divided into three main components: the physical twin, the virtual twin and the communication infrastructure, i.e. the digital thread [9], [20].

The **physical twin** is the real entity, the real system. It can range from being a product, to an industrial process, a network, a machine, or a complex infrastructure. This component works in the physical environment and generates needed data to describe its

state, behavior and interactions with external conditions [9], [11].

The **virtual twin** is instead the digital representation of the real counterpart. It includes structural and behavioral models, logs, real-time data and analysis features. Depending on the application context, it exploits physical models, data-driven ones or hybrid solutions which combine domain knowledge and data analysis [9].

The last element, the **digital thread**, allows for data exchange between real and virtual system. Collected data from the physical system updates the digital model, while analysis results, predictions and decisions may flow the other way around to be received by the real system [9], [18]. This bridge is essential in order to turn the virtual model into an operative tool, keeping it aligned with the represented system.

2.3.2 Layered Models

Within complex systems, Digital Twin architectures are often arranged into layers. This structure facilitates separation between different system functions, such as data collection, retention, processing, simulation and final applications. In this way, the architecture is easier to manage and adapt to different contexts.

Three-level architectures are frequently employed in Network Digital Twin. They usually embed a physical, network and application layer. The first one represents network elements and resources. Then the network layer manages the data repository. The last one provides operational functions and user services [18], [30].

Four-level architectures offer a more general reference model. Here, the physical level stands for real objects, data level collects information, service level elaborates data and analyzes them, while application level provides visualization, decisions support and useful outputs for users or other systems [11].

Some proposals use more detailed architectures. For example, Six-layer models in manufacturing distinguish between physical devices, controller data sources, local repositories, IoT gateways, cloud repositories, and simulation or emulation environments [18], [31]. This level of detail helps manage data flows, especially when information moves from sensors and controllers to edge or cloud platforms [31].

2.3.3 Standardization and Frameworks

DT application in several fields has led to the definition of various reference architectures and different standardization attempts. However, there's no unique architecture that fits every scenario: each solution has to be adapted to the application domain and to the represented system, thus to the twin goals [18]. Frameworks and standards are

important anyway, because they help define a common language, shared design principles and guidelines to guarantee component-system interoperability.

The ISO 23247 standard suggests a reference framework for Digital Twin in manufacturing. It defines general principles, requirements and architectural components aimed at designing this kind of systems [18]. Even the Digital Twin Consortium contributes to the sector standardization by providing terminology, architectural models, security requirements and interoperability principles [18]. NIST offers guidance as well for trustworthy DT designs, with particular attention to cybersecurity [18].

In industrial environments is often employed the Purdue Model, which describes data and communication stream from devices and control systems to levels of supervisory, cloud, edge or enterprise [21], [23]. Even if not specifically developed for DTs, the Purdue Model remains useful to arrange the cyber-physical architectures and to catch the Information Technology and Operational Technology relation.

2.4 Modeling and Simulation Methodologies

A Digital Twin cannot be associated to a mere visual copy of a physical system. To be useful in a wide range of activities, such as monitoring, diagnosis, prediction and what-if analysis, it must rely on models able to describe both the structure and the behavior of the entity under examination [9], [30].

The model choice depends on the system, the available data, the required accuracy level and the analysis goal. In literature, there is a distinction between main approaches. They may fall under physics-based models, data-driven models or hybrid/graph-based models.

2.4.1 Physics-Based Modeling

Physics-based modeling relies on mathematical descriptions drawn from physical laws and application domain knowledge [11], [22], [24]. Unlike pure data-driven approaches, these models do not extrapolate the system's behavior just from observed data, but they try to frame the expected functioning of the system by defining its components' relationships. This solution works well when the system internal logic is known and can be described through equations, analytical models and constraints.

In many Digital Twin applications, physics-based models describe the expected behavior of the system. They can also simulate how the system reacts under different operating conditions [9], [20]. In electrical and cyber-physical systems, for example, equivalent circuit models, Kalman filters, differential-algebraic equations, and observer-based methods such

as the Luenberger observer can estimate system states and reproduce dynamic behavior [10], [24].

The overall advantage of this approach is interpretability. Since the model is based on explicit physical hypothesis, results are way easier to explain and validate. Yet, this characteristic shows some limits: when dealing with non-linear systems or time-varying ones, coming up with an accurate physical model might require a lot of effort and resources. Moreover, high-fidelity simulations can reach elevated computational costs, making this approach less suitable for some real-time scenarios [11], [24].

2.4.2 Data-Driven Modeling

Data-driven modeling compares historical and real-time data to identify patterns, correlations and trends in system behavior [9], [22], [24]. In contrast to physics-based models, it doesn't begin with a mathematical description of the systems. That's why it fits better when the system is too complex to be described by equations or when there is a big amount of operational data.

Techniques like Artificial Intelligence, Machine and Deep Learning, and Big Data analytics cover a central role in this kind of model [11], [21], [24]. Those tools may be employed to carry out faults and anomaly detection, performance prediction, operational state classification and residual useful life estimation [9], [11], [19], [21], [24].

The main advantage that data-driven approaches bring is the ability to recognize relations that are otherwise hard to manually model. They learn from observed behavior and improve when fed with novel relevant data. Nevertheless, they show some critical issues. They require huge and meaningful datasets, and their performance can get worse in the presence of noisy data. Plus, many learning-based models operate as black-boxes, making it harder to explain why they came up with a certain classification or prediction [11], [21].

2.4.3 Hybrid and Graph-Based Approaches

Hybrid modeling combines previous approaches. The target is preserving explicative ability of physics-based models, integrating data-driven techniques to make the model more versatile, to decrease the modeling effort and to better manage inaccuracy [9], [24], [25]. This solution is useful when building a complete physics-based model would be too complex and expensive, and in the meantime a model based solely on Machine Learning would not be trustworthy enough.

Physics-informed deep learning follows this path. Physical constraints are used as

guides for learning algorithms, so that predictions stay consistent with the known system behavior [11], [24]. Graph Neural Networks are a relevant solution too, because they enable system descriptions through components and relationships, instead of analyzing isolated variables [11], [24].

Graph-based modeling is especially useful in complex systems, in communication networks and in cybersecurity scenarios. Entities are modeled as nodes and relations as edges. This allows for the modeling of dependencies, communication paths, access links, properties relations and attack routes [24], [30]. Attack graphs, for example, help in describing how an attack could move within the system, while Knowledge graphs offer another kind of representation, richer in entities, attributes and relationships [19], [21], [30].

Formal modeling languages and standards also support this purview, making systems descriptions more structured and then simpler to elaborate through automated tools. Automation ML support engineering information exchange and specific languages like Meta-Attack Language can be employed in security scenarios [27], [29]. Other approaches use Finite State Machines, probabilistic graphic models or formal frameworks, such as the Temporal Logic of Actions, with the aim of verifying whether the system's behavior satisfies specific security invariants or requirements [14].

2.5 Digital Twin in the Product Lifecycle

The concept of Digital Twin is strictly connected to system lifecycle management. A twin goes hand in hand with the real entity from its early design phase up to the operational stage and, finally, to dismission [11]. Its role changes according to the considered stage. While designing, it is used to compare alternative solutions and evaluate possible risks before the physical system creation. During operation, it monitors the system, supports optimization and searches for anomalies. In the final stage, it may store historical data for diagnosis, audit and future enhancements [9], [19].

This lifecycle view shows that a Digital Twin is not only a runtime monitoring tool. It is a representation that changes together with the physical asset and supports different tasks depending on the lifecycle phase.

2.5.1 Beginning-of-Life

The Beginning-of-Life phase includes design and pre-manufacturing. At this point, the DT is usually employed as a Digital Twin Prototype [9]. The physical asset might not exist yet, but the virtual model already offers a structured representation, requirements,

expected behavior and components' relationships [28].

From a security perspective, this stage enables the so-called Shift Left approach. Security analysis can already take place during the design, avoiding some vulnerabilities becoming part of the final implementation [21]. Threat modeling, preliminary tests and running simulations can help detect weaknesses, possible information leakage paths and insecure architectural choices before deployment [9], [20], [28].

The employment of a virtual replica reduces experimental risks and costs. Designers and security analysts can run what-if analysis, perform penetration tests and evaluate fault scenarios without even touching the real system. This encourages a secure-by-design approach, where security is taken into account from the early stages of the project, instead of just being added after the deployment [11], [28].

2.5.2 Middle-of-Life

The Middle-of-Life phase starts when the physical asset exists and is operative in the real environment. Now, the DT becomes a true instance bonded to a determined physical space through a data exchange [9]. Its main task is keeping an updated view of the system's state and supporting operational decisions, based on real-time information.

One of the most significant applications of this stage is predictive maintenance. Analyzing data, the Digital Twin is able to spot deviations from the expected behavior and identify early signs of malfunctions or degradation, before they turn into critical failures [9], [18]. This shortens downtimes, improves maintenance planning and increases system reliability.

This phase is also fundamental for runtime cybersecurity. Whilst the Shift Left approach focuses on security when designing, the Shift Right regards security analysis of the system at runtime [18], [21]. Comparing the expected behavior in the twin with the observed one in the real system, the DT can disclose discrepancies that point at faults, erroneous configurations or attacks [24].

Furthermore, the twin may be involved in testing modifications before the application to the real system. Security patches, configuration updates and operational interventions can be evaluated in the virtual environment first. This lowers the risk of introducing errors or causing service interruptions within the real infrastructure [9], [24], [28].

2.5.3 End-of-Life

The End-of-Life phase covers the final stage of the system lifecycle, when the physical asset is retired, replaced, or decommissioned. Even in this phase, the Digital Twin can still

provide value. It can support secure decommissioning and preserve a structured record of the system history, including configurations, events, operational data, and security-relevant information collected over time [11], [16], [28].

This historical memory comes to the aid of post-mortem analysis and digital forensics activities. If an incident occurred in the operational phase, data saved inside the twin can help reconstruct the events timeline, detect root causes and understand how an attack or malfunction spread over the system [19]. Same data are employed for what-if analysis, in order to evaluate what would have been the outcome in the presence of different configurations [19].

Information collected during the lifecycle can also contribute to future system realization. Lessons learned from attacks, errors, failures or performance limits guide the development of brand new Digital Twins and more secure physical systems [11], [19]. The End-of-Life also includes the removal of no more needed virtual components and the release of storage and computation resources when the twin is no longer required [17].

2.6 Enabling Technologies

Digital Twins do not rely on a single technology. They combine several components that support data collection, communication, computation, analysis, and decision support [10]. Together, these technologies allow the Digital Twin to move beyond a static model and remain connected to the operational environment [10].

2.6.1 IoT and Sensing Technologies

IoT technology and **sensing systems** establish data foundations for many DTs. Sensors, wearable devices, industrial controllers, smart meters and other connected devices gather information from the physical environment and make it available for the virtual model [9], [10]. If not for this data flow, the Digital Twin would stand static, not a real-time view of the real system.

The spread of Digital Twin is therefore linked to the development of IoT infrastructures and Industrial IoT [9]. These technologies watch real entities through measurable data, such as pressure, temperature, power consumption, position and network activity. Information collected is then sent through the digital thread, a traceable link between the physical asset and its virtual counterpart [11], [25].

These data streams enable strong synchronization between real and digital models. The twin can then update its state, detect changes and perform analysis on current information, instead of relying on hypothesis formulated at design time [11].

2.6.2 Cloud and Edge Computing

Computing infrastructure plays one of the fundamental roles in a Digital Twin. To effectively carry out its tasks, the digital twin must be able to process big amount of data, perform simulations, preserve historical data and sustain services that call for constant updates. As a consequence, adequate computational resources are mandatory to keep pace with such activities [18], [21].

In this context, **cloud computing** fills a central role, offering scalable elaboration and storage capabilities for every system's needs. Cloud platforms let data and services be centralized, to grant availability, accessibility and operational continuity, even in the presence of multiple and geographically distributed sources [31]. This feature is particularly useful in case the Digital Twin has to integrate data coming from assets placed in different locations or support applications requiring a global view of the monitored system.

Edge computing complements the cloud by moving part of the processing closer to the physical system. This matters when the system needs low latency, fast response, or local decisions [9], [21]. By processing data near sensors or controllers, edge infrastructures reduce communication delays and support faster reactions. Many architectures combine cloud and edge resources: edge nodes handle immediate processing, while cloud platforms support aggregation, large-scale analytics, and long-term storage.

2.6.3 AI, Machine Learning, and LLMs

Artificial Intelligence and **Machine Learning** expand the DT's analysis abilities. They allow the twin to go beyond simple current state representation. Thanks to these technologies, the Digital Twin can support prediction, classification, anomaly detection and optimization [10], [11]. The joint processing of historical and real-time data helps reveal recurring patterns, correlations and behaviors that would be of hard manual detection, making the system management more effective and adaptive [9], [11].

Deep Learning is very useful when the system generates a huge amount of complex and heterogeneous data. At the same time, classical Machine Learning techniques can be employed for classification and regression. These tools help predict possible malfunctions, enhance operational performances and detect deviations from the expected behavior of the system.

A further step is made with Large Language Models, that improve DT-user interaction. These models support natural language queries, output interpretation and complex information consultation [70]. In a nutshell, LLMs can act as user interfaces, making the Digital Twin functionalities more accessible.

2.6.4 Blockchain

The **Blockchain** technology can be a support element for Digital Twins, when reliability, traceability and integrity of information become central [12], [25]. This requirement emerges especially when many organizations or stakeholders collaborate sharing data and distributed resources, calling for authentic and verifiable information exchanges.

Thanks to its decentralized nature and records immutability, blockchain securely stores events, transactions, configuration modifications, system operations and other relevant information for the Digital Twin lifecycle [20], [25]. Each update is then registered in a transparent way and preserves its integrity, offering a reliable history of activities and reducing the risk of tampering or non-authorized modification.

Further contribution is provided by smart contracts. They automate data-sharing rules and access control procedures. Through those mechanisms it is possible to define who can access certain information and under which circumstances [25]. This is particularly useful in distributed environments where sensitive data, intellectual property (i.e. IP) or operational information must be protected without affecting the collaboration between parties [10], [20], [25].

2.7 Main Application Domains

Digital Twin technology first originates in highly specialized contexts like aerospace. As digital technologies evolved and data is increasingly available, this approach progressively spreads in many other sectors: manufacturing, energy, healthcare, smart cities, critical infrastructures and cybersecurity. Although applications differ according to the selected domain, all these fields share the need to understand, monitor and optimize complex systems through a constantly updated digital representation which draws from real-world data.

2.7.1 Manufacturing and Industry 4.0

Manufacturing is one of the main application areas for Digital Twins. In **Industry 4.0**, Digital Twins support cyber-physical production systems, where machines, sensors, control systems, and software components interact with each other [22], [23].

A relevant example is the Digital Twin Shop Floor, which links the physical production environment to its continuously updated virtual representation [9]. This allows to monitor production activities, evaluate different strategies for resource allocation, simulate changes before their real application and support more flexible and adaptive control mechanisms.

Another interesting employment concerns **predictive manufacturing**. By analyzing operational data collected from plants, the DT can detect anomalous signals and potential problems before they turn into actual failures. This shortens downtimes and maintenance costs, while improving production continuity and efficiency [9], [31].

2.7.2 Critical Infrastructures and Energy Systems

Digital Twins are also used for **critical infrastructures** and **energy systems**. In particular, in the energy sector they monitor and manage distributed networks, like microgrids, characterized by multiple sources and bidirectional energy flows [24].

These environments are often built on top of the integration of physical processes, control systems, communication networks and cybersecurity mechanisms. Therefore, the Digital Twin provides a unified representation of the infrastructure, creating a coordinated view for components that are usually separated. In so doing, faults detection, risks assessment and what-if analysis of cyber-physical attacks is easier [18].

Frameworks like ANGEL (Automatic Network Guardian for Electrical Systems), show how physics-based simulations and Digital Twin methods can detect faults and classify affected areas in smart grids [10], [18], [26], supporting strong resilience and continuity of operations.

2.7.3 Healthcare and Smart Cities

In **healthcare**, Digital Twins encourage evolution towards a more personalized medicine through digital representations of patient's physiological traits. These models allow for simulating the effects of treatments, drugs or surgeries, before their real application, becoming a precious tool for doctors with whom compare different options to reduce risks for the patient [9], [11]. Moreover, DTs can contribute to diagnosis, care planning and continuous health monitoring [11], [20].

Smart Cities are another important sector for the growth of Digital Twin. The so-called Urban Digital Twin digitally portrays urban infrastructures, transportation systems, public spots and resource flows, providing advanced support to planning activities and cities management [25], [26]. By integrating sensors' data, public entities and other stakeholders, these systems enable effective decisions for mobility, energy use, emergency response and public safety.

A significant example is the **Digital Twin-as-a-Security-Service**, which applies the DT paradigm to public spaces protection and critical urban infrastructures. Here, the twin combines information coming from the physical environment with knowledge

provided by different involved actors, supporting vulnerability assessment and real-time threat analysis [26]. So, the Digital Twin helps to improve situational awareness and to strengthen prevention capabilities and incident responses.

2.7.4 Focus on Cybersecurity

Cybersecurity is one of the most relevant application areas for Digital Twins. The twin provides a controlled environment where security analysis is performed without exposing the real system to risks [15], [18]. This is useful for critical infrastructures, industrial control systems, and operational technology environments, where direct testing on production systems could be unsafe or unacceptable [11], [22], [29].

A cybersecurity-oriented Digital Twin can be fundamental in studying vulnerabilities, evaluating attack scenarios, and analyzing cascading effects across connected components [13], [18], [22]. It can also support penetration testing, incident response preparation, and assessment of mitigation strategies.

Regarding intrusion detection, the DT provides a reference model of the expected behavior. When the real system differs from this expected behavior, the difference may indicate a misconfiguration, a malfunction or cyberattack [11], [18], [24]. This makes the Digital Twin useful both as a simulation environment and as a runtime monitoring and detection mechanism.

2.8 Challenges, Limitations, and Future Directions

Digital Twin adoption still presents several challenges. These include implementation cost, scalability, data quality, interoperability, security, and privacy. These issues must be overcome to move from experimental prototypes to Digital Twin systems that can support large-scale and long-term use.

2.8.1 Implementation Cost and Scalability

Digital Twin realization may require significant investments in terms of expertise, technological infrastructures, modeling activities and computational resources [9]. In order to be really effective, the twin must represent the physical system with a high degree of detail appropriate to the application's objectives, describing both the structure and the behavior. Reaching this goal demands specific competencies in the reference domain and the employment of advanced tools for data collection and management [9], [31].

One of the most critical aspects when designing a DT is the tradeoff between model

accuracy and model complexity. A detailed representation can provide better analysis' outputs and a better system understanding, but it takes greater needs for data, processing and maintenance. On the contrary, simpler models are easier to implement and manage, but they could not be able to capture all the relevant aspects of the system's behavior [22], [24]. Finding the right equilibrium between fidelity and sustainability is one of the major challenges in Digital Twin design.

Scalability is a further issue, especially when applying the paradigm to extended infrastructure, complex industrial plants or big networks. As the number of monitored components increases, so does the amount of data [30]. In contexts like microgrids, high-frequency data streams could lead to bottlenecks, calling for the adoption of distributed architectures and advanced strategies to manage computational resources [10], [24], [30].

2.8.2 Data Quality and Interoperability

The effectiveness of a Digital Twin depends on the quality of the data used to update and operate it. Missing values, noisy measurements, inconsistent formats, or inaccurate sensors can reduce the reliability of predictions and automated decisions [11], [20]. If the input data is incomplete or corrupted, the twin may produce an incorrect representation of the physical system.

Data heterogeneity is another major issue. Real systems often generate structured, semi-structured, and unstructured data. This data may be stored in different platforms and exchanged through different protocols [20], [24]. This fragmentation makes it harder to build a coherent virtual representation.

Interoperability is therefore a core requirement. Digital Twin components must exchange information through shared formats, interfaces, and semantic models [20], [21]. However, interoperability is not only a technical problem. Organizations may avoid sharing operational data because of privacy, confidentiality, or intellectual property concerns [11], [12], [20].

When data is incomplete or visibility is limited, the Digital Twin may need to rely on approximations. This can reduce its value for optimization, fault diagnosis, security analysis, and automated decision-making.

2.8.3 Security and Privacy

The same features that make Digital Twins effective tools also introduce new security and data protection challenges. Since the twin keeps a continuous information exchange with the physical system, it can become another entry point for attacks [11], [18], [30].

An attacker might aim at data flows, virtual models, storage systems or communication channels between physical and digital environments.

Among the most relevant threats there are Denial of Service and False Data Injection. With DoS the focus is compromising the availability of the DT and its communication infrastructures, inhibiting the correct functioning. Dealing with False Data Injection, the entry of false or tampered information shatters analysis quality and brings to erroneous decisions [11], [22]. When the Digital Twin is involved in control activities, corrupted data propagation can have direct consequences on the behavior of the physical system below [24], [27].

Privacy is also important, especially in healthcare, smart cities, and industrial contexts. Digital Twins may process sensitive, proprietary, or personal data. Weak encryption or poor access control can expose this information to unauthorized actors [11], [18], [20]. The Digital Twin itself can also reveal valuable information about the represented system. If an attacker gains access to the virtual model, they may learn details about topology, configurations, protocols, dependencies, or vulnerabilities [18], [22]. This knowledge can support targeted attacks. For these reasons, Digital Twin systems should follow **security-by-design** and **privacy-by-design** principles. Security should be part of the architecture from the start. It must cover data flows, access control, synchronization, storage, analysis, and lifecycle management [12], [18].

Chapter 3

Kubernetes, RBAC and Network Policies

Kubernetes (also appointed as K8s) has become one of the most powerful platforms for deployment and cloud-native applications management. Its architecture enables workloads creation, update, scaling and replacement through automated processes. This flexibility simplifies resources management but makes cluster security more complex. System's state changes over time and these variations might modify permissions, resources relationships and communication paths.

This chapter introduces the needed Kubernetes concepts to better understand the thesis objectives. First, it presents the main architectural components and fundamental cluster resources. Then, it focuses on two central security mechanisms: RBAC and Network Policies. The former states which subjects are able to perform certain actions on Kubernetes resources, while the latter regulates communication between workloads. At last, the chapter shows the importance of analyzing together those two aspects in dynamic cloud-native contexts.

3.1 Kubernetes Overview

3.1.1 Kubernetes and Container Orchestration

Containers represent a fundamental tool to enclose and execute applications, guaranteeing isolation, portability and a more efficient resource usage. However, when the number of containers increases, their management calls for mechanisms able to minimize downtime and monitor the whole applications' lifecycle. Kubernetes meets this need offering a scalable and extensible environment that supports service discovery and load balancing,

storage orchestration, automated rollouts and rollbacks, self-healing, batch execution, horizontal scaling, IPv4/IPv6 dual-stack and integration with external components [33].

Kubernetes is born from Google experience with previous systems like Borg and Omega. Borg had been developed for large scale management of containers, long -lasting services and batch jobs inside clusters, enhancing resources usage and automating operations, such as scheduling, scaling, updates and faults management. Instead, Omega introduced a more modular architecture, based on a shared state of the cluster and on separated components. Kubernetes relies on many of these principles and makes them available through an open-source platform, even outside Google's infrastructure [34].

Before the diffusion of these technologies, applications were run directly on physical servers. This model made it difficult to define boundaries for resources usage. An application could then keep busy the majority of the capacity and slow down the other applications. A possible solution was the employment of more physical servers, but this approach came with high costs and limited scalability [33].

Virtualization faced part of these problems allowing the execution of multiple Virtual Machines (VMs) in an isolated fashion on the same physical server. Each VM had its own operating system, improving isolation, hardware utilization and infrastructure scalability. Containers have then introduced a lightweight model: differently from VMs, they share the host's OS and require less resources. At the same time, they keep applications separated from the underlying infrastructure, fostering portability between OSs and cloud environments [33].

The success of container is also related to the ease of image creation and distribution, compared to Virtual Machines. They support continuous development, integration and deployment, they keep a better consistency between environments, they improve observability and lead to a more efficient usage of resources. Moreover, they allow for breaking applications into smaller components, called microservices, that can be developed, deployed and managed independently [33].

Yet, the management of a container-based environment does not just consist in their launch. It requires scheduling, monitoring, load balancing, resources allocation, updates, recovery and applications' state checking. Kubernetes makes use of a **declarative model**: the user defines the desired state of the system, while different control processes observe the current state and operate to reduce the gap between the two states. In this sense, Kubernetes doesn't limit to run a pre-defined workflow, but it coordinates many independent control loops. This approach makes the system more extensible, resilient and robust [33], [34].

A central element of Kubernetes is the **Pod**, the smallest deployable unit and can contain one or more containers. Kubernetes also use labels, selectors, health checks and

reconciliation loops to plan workloads, detect problems and react to cluster changes [33], [34].

Although Kubernetes shares some common features with Platform-as-a-Service (PaaS), it is not a monolithic platform and it does not limit the types of running applications. It's not in charge of source code deployment, it doesn't build applications and it does not provide application services like middleware, database or cache. In the same way, it does not impose a unique logging, monitoring or alerting system, but it lets gather and export metrics towards external tools [33].

3.1.2 Kubernetes Cluster Architecture

K8s cluster is made by a control plane and a set of **worker nodes** that run containerized applications. Worker nodes host Pods and their associated containers, while the **control plane** coordinates cluster operations and keeps an updated vision of the state of resources. In production systems, control plane components may be distributed across different machines and the cluster might include multiple nodes, so that resiliency and availability increase [35].

The control plane takes decisions regarding the whole cluster. The *kube-apiserver* exposes Kubernetes APIs and represents the main access point for control plane. It is designed to scale horizontally, that is creating more istances as demand increases. *etcd* is the key value store used to preserve data and the cluster's state. The *kube-scheduler* assigns new Pods to the available nodes based on requested resources and other scheduling constraints. The *kube-controller-manager* executes control processes, that verify the state of resources and intervene when the current state does not correspond to the desired one. These include, for example, controllers for Node, Job, EndpointSlice and ServiceAccount. The *cloud-controller-manager*, when present, acts as a bridge between the cluster and cloud provider's APIs, and manages resources that depend on the cloud infrastructure [35]

Worker nodes, instead, provide an environment where workloads run. Each node includes the *kubelet*, an agent in charge of monitoring that containers defined inside Pods are actually executed. The *container runtime* manages startup, execution and lifecycle of containers. The *kube-proxy* supports network communication and applies the needed network rules to direct the traffic towards Pods [35].

The control plane also monitors the state and capacity of the nodes. Each of them communicates information about CPU, memory, operating conditions and reachability. Those data help the scheduler in deciding where to place the Pods and allows the cluster to detect faults or availability problems. If a node does not answer, Kubernetes can avoid assigning new workloads to it and initiate the necessary actions to maintain the service.

Furthermore, **labels** and **selectors** can limit the positioning of specific Pods to certain nodes [36].

One of the most interesting aspects of Kubernetes is its architectural flexibility. Components can be mounted, distributed and managed in different ways, adapting the cluster to the context needs, whether it is a small environment or a large-scale production one [35].

3.1.3 Declarative Management and Reconciliation

Dealing with the control plane, it is worth going into detail with the role of **controllers**. A controller is the component that runs the **control loop**, a never-ending observing process aiming at bringing the system's state closer to the desired one. Each controller focuses on a specific aspect of the cluster and tracks at least one kind of Kubernetes resource. These resources carry information about the desired state, which the controller compares with the current state [37].

Based on control loop results, the controller may request the API server for changes or directly intervene on managed systems. It can also update the state of resources to signal, for example, that a task has been accomplished or that a certain condition has been met [37].

An interesting example is the **Job Controller**. A **Job** is a resource that requires one or more running Pods to complete a work. When the Job Controller detects a new Job, it verifies that the correct number of Pods required to complete the work is running in the cluster. To do so, it talks to the API Server and requests the creation or deletion of Pods. Once the work is done, it updates the state of the job to point out its completion [37].

The operation of controllers contributes to the dynamic nature of Kubernetes. The state of the cluster changes as workloads are completed, resources are updated and control loops carry out checks and corrections. Therefore, the system is never in a stable state but keeps moving towards the desired one. The important concept isn't reaching at all costs an immutable condition but applying useful changes that preserve cluster coherence with the requested configuration [37].

Kubernetes makes use of many separated controllers, instead of just one big control loop. This design choice reduces faults impact, because if one controller goes into malfunction then it is non-blocking for the entire system. Different built-in controllers are executed by the *kube-controller-manager*, but Kubernetes leaves to users the possibility of developing their own personal controllers [37].

Starting from this model, the concept of **Operator** arises. An Operator is a software extension made by one or more controllers, often customed, that manage applications or

complex services. It automates tasks which otherwise would require human administrator intervention, such as installation, configuration, update, backup, recovery and scaling. Operators communicate through Kubernetes APIs and check on custom resources. Like controllers, if there is no Operator for a specific application, the user can write its own one [38].

The controller pattern is linked to declarative resource management. In Kubernetes, objects can be created, updated, and deleted through configuration files, usually written in YAML or JSON. These files describe the desired state and can be collected in a directory and applied to the cluster with `kubectl apply`. Before introducing any change, `kubectl diff` shows a preview with which changes will be applied. *kubectl* supports three main management modes: imperative commands, imperative configuration via files, and declarative configuration [39].

In the declarative model, the user does not describe every single operation to be performed. Instead, the user edits the configuration file and lets `kubectl apply` determine whether a resource should be created or updated. During this process, `apply` compares the new configuration with the live state of the object and with the previously applied configuration. This way, it can determine which fields to modify, which to remove, and which changes in the cluster to keep [39].

For example, if the container image is modified in a Deployment file, `kubectl apply` updates that value in the cluster. If a field in the previous configuration is deleted from the file, Kubernetes can also remove it from the live object. A value added by another process and never managed by the file can be preserved. For this reason, it is preferable not to randomly alternate between declarative and imperative management on the same object, because the two modes can produce conflicts [39].

A `deployment.yaml` file can contain the resource type, deployment name, number of replicas, labels, and container image. Once defined, it can be applied with:

```
kubectl apply -f deployment.yaml
```

The live configuration can be viewed with:

```
kubectl get deployment <name> -o yaml
```

To delete a resource, the most direct command is:

```
kubectl delete -f deployment.yaml
```

Alternatively, `kubectl apply` *–prune* can remove objects that no longer appear in the configuration files. This option should be used with caution, however, as an incorrect selection can lead to the deletion of unexpected resources [39].

3.2 Kubernetes Resources and Abstractions

Kubernetes organizes applications through a set of resources and abstractions that describe workloads, communications, configurations and identities. These elements are interconnected and define how applications are executed and managed inside the cluster.

3.2.1 Namespaces

Namespaces isolate groups of cluster resources. These resources must have unique names inside the same namespace, but not necessarily when in different namespaces. They are especially useful in shared environments with many users, teams or projects. Each resource can belong to only one namespace.

Kubernetes presents four already active namespaces: *default*, which allows to have an already initialized and usable namespace; *kube-node-lease*, used for information concerning nodes' state; *kube-public*, readable also from unauthenticated users; and *kube-system*, for objects created by the system. Most of the objects, like Pods, Deployments and Services, belong to a namespace, even though there exist cluster-scoped resources [40].

3.2.2 Pods and Containers

The Pod is the smallest deployable unit in Kubernetes. It includes one or more containers, which share some network and storage resources, and run together on the same node. In many cases a Pod has just one container (it behaves like a wrapper), but it may include others for support, logging and auxiliary activities. Pods are not meant to be permanent. They are created, replaced or deleted during the operation of the cluster. For this reason, Kubernetes usually manages Pods through higher-level resources, such as Deployment or StatefulSet [41].

During its lifecycle a Pod can be in different states, like 'Pending', 'Running', 'Succeeded', 'Failed' or 'Unknown', based on the situation of containers and the hosting node. The *kubelet* monitors containers execution and can restart them, according to the restarting policy. However, if the Pod gets deleted or the node is no longer available, Kubernetes doesn't recover the same Pod, but it creates a brand new one when requested by a higher-level resource [42].

3.2.3 Workload Resources

Workload Resources, such as **Deployment**, **Statefulset**, **DaemonSet** and **Job**, are built-in APIs that allow for a declarative management of workloads and their components.

A Deployment helps manage the workload of a stateless application, where each Pod is interchangeable and replaceable. A StatefulSet, instead, focus on one or more Pods that run the same application code, holding a stable and distinct identity for each Pod. It is useful to create a bridge between Pods and the persistent memory.

A DaemonSet ensures that a certain Pod is executed on every node of the cluster, or on a selected group of them. It is employed for local functions, such as log collection, monitoring or network services.

A Job is used for tasks that need to be completed and then terminated. Kubernetes creates one or more Pods to run the work and checks that the required number of executions complete successfully. It fits well for batch elaborations, migrations, or temporary operations [43].

3.2.4 Services

Because Pods can be created, replaced, or moved during cluster operation, their addresses are not a reliable reference over time. A **Service** provides then a stable access point for a set of Pods.

The Service selects Pods thanks to labels and exposes an endpoint to reach them. It can also distribute traffic among many replicas of the same application. According to the configuration, a Service might be accessible only inside the cluster or exposed to the outside [44].

3.2.5 Labels and Selectors

Labels are key-value pairs associated with Kubernetes objects to identify them, organize them and group them based on criteria dictated by the user. They do not directly modify the resource behavior, but let other components select relevant objects.

Selectors use labels to detect set of resources, for example Pods managed by a Deployment or reached by a Service. This mechanism avoids names or fixed addresses dependency and makes possible a more flexible management of workloads. Labels can also be updated over time, thus changing the membership of objects to different logical groups [45].

3.2.6 ConfigMaps, Secrets, and ServiceAccounts

Kubernetes also provides dedicated resources for application configuration, sensitive data management and workload identity.

A **ConfigMap** is used to separate configuration data from the container code and image. It may contain key-value pairs, configuration files, or other non-sensitive parameters needed by the application. Pods can read this data via environment variables, command line arguments, or files mounted to a volume. This allows you to change the configuration without rebuilding the container image. ConfigMaps are not suitable for passwords or other sensitive information, for which Kubernetes provides Secrets [46].

A **Secret** is used to store confidential data, such as passwords, tokens, keys, or credentials. This way, sensitive information does not need to be directly inserted into container images or application configuration files. Pods can access Secrets through environment variables or through files mounted on a volume. However, Secrets are not automatically encrypted in every cluster configuration, so they must be protected with access controls, encryption at rest, and proper permission management [47].

A **ServiceAccount** represents the identity used by workloads to interact with the Kubernetes API. Unlike human users, it is primarily intended for Pods, automated processes, and applications running in the cluster. Each ServiceAccount belongs to a namespace and can receive permissions via RBAC. When a Pod uses a ServiceAccount, it can authenticate to the API Server and perform only the permitted operations. For this reason, assigning ServiceAccounts with limited privileges is important to reduce the risk of unnecessary access [48].

3.3 Kubernetes Security Model

3.3.1 Main Security Areas

The core point of this thesis is to keep the Kubernetes cluster secure, continuously verifying the observation of the imposed security invariants. In this work the attention is focused on RBAC and Network Policies-related queries, but the CNCF provides also a wide set of good practices to strengthen cluster security.

Confidentiality represents a central feature when protecting the control plane. It can be implemented both for data in transit, using TLS, and for data at rest stored in the Kubernetes control plane. Secrets too represent a useful tool to protect sensitive information, such as passwords, tokens, private keys, which are then used by running applications inside containers during access or authentication phases.

Dealing with worker nodes, one of the main goals is to ensure **isolation**. In this context Network Policies are very important to regulate traffic between Pods and communications between a Pod and the external network. Another relevant mechanism is portrayed by Admission Controllers, which checks specific fields in the Kubernetes API requests and they can validate, modify or even refuse them.

Further practices are **auditing** and **logging**, useful to keep trace of events that happen in the cluster. In this regard, the proposed Digital Twin of this thesis records events in a specific log file and shows them to the user through a dedicated section of the UI (User Interface). In this section, events can be filtered according to both involved resources and to the type of operation (`ADD`, `MODIFY`, `DELETE`).

Among all the different security mechanisms provided by Kubernetes, RBAC and Network Policies play a central role in this project. The former states which subjects can carry out certain operations on specific cluster resources, while the latter checks communication paths between workloads. For this reason, the next sections will dig into the RBAC model and then how Network Policies work [49].

3.4 Authorization and Role-Based Access Control

In Kubernetes, most of the cluster management operations goes through the Kubernetes API server. Workload creation, configuration modification or resource access are then subject to security checks before producing effects on the system's state.

The first check is the authentication, that binds the request to an identity (e.g. a user, a group or a ServiceAccount). The identification does not mean that the operation is allowed, because after the authentication comes the authorization, through which the API server verifies if the identity has all necessary permissions to execute the requested action. An authorization decision can be described with the following relation:

`subject - action - object - scope`

The **subject** represents the identity that generates the request; the **action** is the operation; the **object** points out the resource involved; the **scope** delimits the validity context for the permission (may be a namespace or the whole cluster). The final decision depends on the combination of different attributes, including identities, belonging groups, verbs, API groups, resources and subresources, namespaces and objects' names.

Permissions aren't directly expressed through HTTP verbs. In fact, Kubernetes translates requests directed to its resources into authorization verbs. A `POST` request corresponds to a `create`, while `PUT` and `PATCH` are interpreted as `update` and `pacth`.

The method `DELETE` is associated to `delete`, when it concerns just one object, or to `deletecollection`, when a set is involved.

`GET` requests are distinguished in `get`, `list`, `watch`. The first verb is for a specific resource, the second one is for a collection and then the third for the continuous observation of changes. Speaking of confidentiality, these permissions must be carefully evaluated, because they all allow access to the complete content of returned objects. For example, the list permission on Secrets can expose information contained in all Secrets visible to the subject.

Moreover, Kubernetes tells apart API resource-related requests and non-resource requests. These concern endpoints that do not represent cluster's objects, like some health checking or discovery paths, and are granted based on the requested path and the HTTP method employed.

Beside ordinary verbs, Kubernetes provides specialized operations associated with particularly sensitive capabilities. The `impersonate` verb allows to make requests with another subject's identity, In the scope of RBAC management, `bind` lets assigning specific roles, while `escalate` is for creating or changing roles with privileges which the requestor doesn't have yet. An overly permissive configuration of these operations might pave the way for privilege escalation attacks.

Security evaluation cannot be limited to the explicitly declared permissions. Some authorizations, in fact, produce indirect capabilities. A subject authorized for workload creation or modification could use a more privileged ServiceAccount or gain access to data and credentials available inside the namespace. So, the actual privilege depends also on relationships between workloads, identities Secrets and other cluster's resources.

The API server is crucial for taking decisions, evaluated through one or more authorization modules called *authorizer*. Kubernetes supports different models, including **Node**, **Webhook**, **ABAC (Attribute-Based Access Control)** and **RBAC**. Their characteristics, applicability, and security implications in Kubernetes-based service platforms have also been examined comparatively in the literature [69].

If no authorizer explicitly grants the request, the operation is denied, usually through a `403 Forbidden` response. The authorization must also be separated from the **admission control**: the former checks if the user can perform the operation, the latter analyse the authorized request's content before it is applied [50].

3.4.1 RBAC Overview

Going into a little more detail, one of the most widely used authorization mechanisms in Kubernetes is RBAC. As the name suggests, this model regulates access based on the

permissions associated with roles assigned to users, groups, or ServiceAccounts. Its goal is to establish which actions can be performed on cluster resources, in what scope, and by whom.

Authorizations are managed through the *rbac.authorization.k8s.io* group API, while policies can be defined and configured as regular Kubernetes objects via the Server API. This allows permissions to be described declaratively and integrated into the same process used to manage other cluster resources. RBAC follows a model based on the principle of least privilege, according to which each subject should receive only the permissions necessary to perform their tasks. This allows for limiting unwanted access and reducing the impact of potential errors or compromises. RBAC rules are additive: Kubernetes combines all permissions granted to a subject, while it does not provide explicit *deny* rules.

The effectiveness of RBAC therefore depends on the correct configuration of permissions. Excessively broad permissions, wildcards, or access granted to unnecessary subjects can increase the risk of privilege escalation and resource misuse. For this reason, the analysis of effective permissions represents a central aspect in the evaluation of cluster security [51].

3.4.2 Roles and ClusterRoles

You can create two objects to define a role: **Role** and **ClusterRole**. The Role deals with permissions on resources within a namespace, while in contrast ClusterRole deals with permissions on resources that span the entire cluster (in fact it is itself a non-namespaced resource). An example of how to define a Role might be the following:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
namespace: test-ns
name: pod-view
rules: - apiGroups: [""]
resources: ["pods"]
verbs: ["get", "watch", "list"]
```

With this manifest, the Role grants *read* permissions on the pods of the *test-ns* namespace within the core Kubernetes API group (written as ""). Similarly, the manifest of a ClusterRole can be defined as follows:

```
 apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
name: admin
rules: - apiGroups: ["*"]
```

```
resources: ["*"]
verbs: ["*"]
```

Here, any permission is granted on all resources and within any API group ("*" means any). The main difference is the absence of the namespace field [51].

3.4.3 RoleBindings and ClusterRoleBindings

Once you've defined the actions granted by the role and on which resources, you need to specify who the permissions are associated with. This is where **RoleBinding** and **ClusterRoleBinding** come into play. The former binds a user or ServiceAccount to a Role within a namespace; the latter binds a subject to a ClusterRole across the entire cluster and, consequently, across all namespaces. A RoleBinding can reference a Role defined in the same namespace. However, it can also be referred to a ClusterRole and apply its permissions only within the namespace in which the RoleBinding was created. This allows you to define a set of common permissions at the cluster level and reuse them in different namespaces, without automatically granting access to all resources in the cluster. If, however, the goal is to assign ClusterRole permissions across all namespaces, there is the need to use a ClusterRoleBinding.

```
apiVersion: rbac.authorization.k8s.io/v1
 kind: RoleBinding
 metadata:
   name: pod-view-access
   namespace: test-ns
 subjects:
   - kind: User
     name: Andrew
     apiGroup: rbac.authorization.k8s.io
 roleRef:
   kind: Role
   name: pod-view
   apiGroup: rbac.authorization.k8s.io
```

This RoleBinding example shows how the previously defined Role is bound to the user 'Andrew' within the *test-ns* namespace.

```
apiVersion: rbac.authorization.k8s.io/v1
 kind: ClusterRoleBinding
 metadata:
   name: cluster-admin
 subjects:
```

```
  - kind: Group
    name: system:admins
    apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: ClusterRole
  name: cluster-admin
  apiGroup: rbac.authorization.k8s.io
```

Here, the ClusterRole from before, with full access to the cluster resources, is associated with a group of system users belonging to the 'admins' group.

An important aspect concerns the **roleRef** field, which identifies the Role or ClusterRole associated with the binding. After the creation of the RoleBinding or ClusterRoleBinding, this reference cannot be modified. If you try to change it, the Kubernetes API returns a validation error. To associate the same subjects with a different role, it is therefore necessary to delete the existing binding and create a new one. This choice prevents that a user authorized to modify the list of subjects can also replace the assigned role and grant broader permissions. In this way, it is possible to allow the management of the subjects present in the binding without allowing the modification of the associated permissions. Furthermore, the switch to a different role is considered as the creation of a new relationship between subjects and permissions. Deleting and subsequently recreating the binding therefore requires verifying that all the subjects present must receive the privileges defined by the new role [51].

3.4.4 Effective Permissions

As already said, RBAC permissions are additive, and there are no explicit *deny* rules. The effective permissions for a subject are then the result of all possible permissions received from a binding, automatically combined by Kubernetes. Moreover, effective permissions also depend on multiple elements simultaneously: the subject they apply to, the groups they belong to, Role, ClusterRole, binding, and the namespace they are inserted into. For example, if a ClusterRole exists but is associated with a RoleBinding, permissions are limited at the namespace level.

Verbs, resources, subresources, API groups, and **wildcards** also concretely determine the extent of access. An interesting case is *resourceNames*, a specific field in RBAC rules where you can specify that the permission is limited to individual instances of a resource, and not to all resources of that type. Another thing to keep in mind is that some permissions can be automatically expanded if wildcards are used and new resources or subresources are added.

Then, with the **Aggregated ClusterRoles** feature, a composite role can be constructed without manually entering all the rules within it. Kubernetes selects other ClusterRoles through specific labels and automatically combines their permissions. Consequently, if one of the selected ClusterRoles is updated, the aggregate role's set of permissions also changes. This mechanism enables the reuse and extension of permissions, but requires caution because a change to a source role can affect the privileges assigned to multiple subjects.

Being able to recognize effective permissions by evaluating this entire series of elements and features is very important: reading a single manifest isn't enough; it's necessary to reconstruct all the chain of relationships and calculate the final result in order to be able to robustly assess the system's security risks [51].

3.4.5 RBAC Security Risks

RBAC allows you to limit the operations that users and workloads can perform on cluster resources, but incorrect configuration can pose a risk. The main problem is assigning permissions that are broader than necessary. Users, groups, and ServiceAccounts should only be able to perform actions required by their functions, while in practice, third-party applications and components often receive elevated privileges to simplify installation and management [52], [53].

This risk is particularly evident in add-ons and applications installed in the cluster. If one of its Pods is compromised, an attacker can exploit the permissions of the associated ServiceAccount to interact with the API Server. This allows them to read sensitive information, modify resources, or create new workloads. Multiple permissions, even if individually limited, can also be combined to create a privilege escalation path [53].

Another critical issue arises from the complexity of RBAC relationships. Effective permissions depend on the combination of subjects, groups, roles, bindings, and namespaces. For this reason, reading a single YAML file isn't always sufficient to understand which operations are actually permitted. Small changes can introduce unexpected permissions or expand the access of entities already present in the cluster [54].

To address this problem, RBAC configurations can be translated into a formal model and verified against specific security invariants. A solver can identify cases where an entity obtains unexpected permissions or where multiple rules generate an escalation path. This approach is also central to the Digital Twin proposed in this thesis, which reconstructs the cluster's actual permissions and continuously verifies their compliance with the defined security properties.

3.4.6 RBAC Representation in the Thesis

From the analytical point of view, the RBAC model can be abstracted with the following path:

$$\texttt{Subject} \rightarrow \texttt{Binding} \rightarrow \texttt{Role} \rightarrow \texttt{Permission}$$

This relational structure lends itself to a **graph-based modelling**. In the cluster's Digital Twin, subjects, bindings, roles and permissions are represented as nodes and edges (i.e. relationships), allowing for the reconstruction of the actual privileges and the automated verification of security invariants, such as the absence of excessive access to sensitive resources or unauthorized creation, deletion, and escalation capabilities.

The first point of the path (i.e., subject) needs to be carefully addressed because it can be User, Group, or serviceAccount. It's important to consider group membership in the model, because, as we'll see, a user obtains permissions not only through direct bindings but also inherits them simply by belonging to a group. So, the path can be expanded as:

$$\texttt{Subject} \rightarrow \texttt{Group} \rightarrow \texttt{Binding} \rightarrow \texttt{Role} \rightarrow \texttt{Permission}$$

The model is also able to demonstrate how two seemingly similar relationships can generate very different privileges, simply because in one case the binding is a RoleBinding and in the other a ClusterRoleBinding. Furthermore, permissions are represented not only as simple authorizations, but the Digital Twin treats them as combinations of verbs, resources, API groups, and namespaces (if not cluster-scoped).

A very important task of the DT is also to reconstruct the effective permissions, rather than considering individual manifests (as already specified, RBAC relationships are additive). The model then performs a union operation of the granted permissions, offering the user a view of the effective privileges.

The graph structure produced by the Digital Twin is then used as the basis for **formal verification**. The nodes and relationships extracted from the cluster are transformed into queryable facts, while the **queries** describe the security properties that must be respected. The **solver** then verifies whether there is a path that allows a subject to obtain an unexpected permission or perform an action considered dangerous.

When a query fails, the system not only returns a negative result, but also produces useful information to understand the **violation**. This information may include the subject involved, the crossed binding, the associated role, the detected permission, and the affected resource. This evidence is then displayed in the UI, making the verification results more readable for the user.

A final important aspect concerns dynamic model updating. When an RBAC object is added, modified, or deleted in the cluster, the Digital Twin updates the corresponding relationships and re-run the checks. This allows for ongoing verification that security invariants continue to be respected even after changes to the cluster configuration.

3.5 Kubernetes Network Policies

3.5.1 Kubernetes Networking Model

The Kubernetes networking model was born from the need to allow distributed applications that don't necessarily run on the same node to communicate with each other. In a cluster, workloads can be created, deleted, or moved at any time, so the network must be able to maintain connectivity even in the face of constant changes.

Containers belonging to the same Pod share the same network environment and can communicate directly via *localhost*. Each Pod receives its own **IP address**, which identifies it within the cluster and allows it to reach other Pods without having to know the physical node on which they are running. This makes communication simpler from an application perspective, because the components can be thought of as entities independent of the underlying infrastructure.

The temporary nature of Pods, however, introduces a significant problem. Since a Pod can be recreated with a different address, it's not practical to use it as a stable reference. For this reason, Kubernetes provides Services, which represent a constant access point to a set of Pods. This way, the application continues to be reachable even when its individual instances change.

Cluster networking is therefore not just about packet exchange, but also about the ability to separate an application's logical identity from the physical location where it runs. This principle is particularly useful in dynamic systems, where scalability and fault tolerance require instances to be replaceable without interrupting communications.

Kubernetes defines the general behavior that the network must ensure, but leaves the concrete to plugins compatible with the **CNI (Container Network Interface)**. These plugins handle aspects such as address assignment, interface creation, and connection between Pods distributed across different nodes. The choice of plugin is therefore not purely technical, as it can also influence the security features available in the cluster.

The basic networking model prioritizes connectivity and tends to allow extensive communication between workloads. This choice simplifies application operation but can be too permissive from a security perspective. Once a Pod is compromised, an attacker could

attempt to reach other internal components and move laterally within the cluster [55].

For this reason, connectivity should not be considered merely an operational requirement, but also an element to be controlled. Network Policies introduce precisely this level of control, letting administrators specify which communications must be allowed, and which must be blocked. In the context of the thesis, the network is therefore observed not only as a connection infrastructure, but also as a set of relations that must respect precise security invariants.

3.5.2 Network Policy Overview

Network Policies are the main mechanism used in Kubernetes to control network communications between Pods. Their function is not to create connectivity, which is already provided by the network plugin, but to limit it according to administrator-defined rules. This prevents all cluster workloads from communicating freely with each other.

A Network Policy selects one or more Pods using labels and can regulate incoming traffic (i.e. **ingress**), outgoing traffic (i.e. **egress**), or both. Rules can specify which Pods, namespaces, or IP ranges are allowed to communicate and can also restrict traffic to specific ports and protocols.

Inbound and outbound isolation are evaluated separately. A Pod can therefore be protected only for incoming traffic, only for outgoing traffic, or both. When no policy selects a Pod in a given direction, traffic generally remains allowed. Conversely, when the Pod is isolated, only communications explicitly specified by the applicable policies are permitted.

Rules, similarly to RBAC ones, have **additive** behavior. If multiple Network Policies select the same pod, the allowed communications are combined, and there is no prioritization between the policies. Therefore, the overall effect cannot be assessed by reading a single configuration but requires considering all the rules affecting the workload.

An example is the following policy, which selects pods labeled app: database and allows incoming traffic only from pods labeled app: backend on TCP port 3306.

```
apiVersion: networking.k8s.io/v1
 kind: NetworkPolicy
 metadata:
   name: allow-backend-to-database
   namespace: test-ns
 spec:
   podSelector:
     matchLabels:
```

```
      app: database
  policyTypes:
    - Ingress
  ingress:
    - from:
        - podSelector:
            matchLabels:
              app: backend
      ports:
        - protocol: TCP
          port: 3306
```

In this case, *podSelector* identifies the 'database' pods to which the policy applies, while the *from* field defines which workloads can send traffic. Finally, the *ports* section limits communication to the port used by the database service.

A common approach is to first apply a **default deny** policy and then add only the necessary communications. This reduces the possibility of lateral movement in the cluster and applies the principle of **least privilege** to the network as well [56].

The actual application of Network Policies, however, depends on the CNI plugin used. In the Digital Twin developed in the thesis, these policies are represented together with the Pods, the namespaces and the related labels, to reconstruct the communications actually allowed and verify security invariants such as isolation between workloads or limited access to specific ports.

3.5.3 Pod Selection and Policy Scope

The effectiveness of a Network Policy depends first and foremost on the ability to correctly identify the workloads to which it must be applied. Kubernetes does not link policies to individual Pods by name or IP address, but uses labels instead. This choice is consistent with the dynamic nature of the cluster, where Pods can be deleted and recreated while maintaining the same application function.

The *podSelector* field therefore defines the set of Pods protected by the policy. The selection remains limited to the namespace in which the Network Policy was created. This means that a policy does not automatically affect Pods located in other namespaces, even when they share the same labels.

An empty selector, however, takes on a broader meaning, as it includes all Pods in the namespace. This configuration is particularly useful when introducing a general level of isolation and subsequently authorizing only certain communications. From a security

perspective, labels therefore become an essential part of the policy: an out-of-date or inconsistent labelling can result in a Pod not being included in the expected rules [56].

In the Digital Twin, the set of Pods selected by a Network Policy isn't represented as a simple manifest field, but as a relationship between policies, labels, namespaces, and actual Pods. This allows you to verify which workloads are actually selected even after changes to the cluster configuration.

3.5.4 Ingress and Egress Rules

After identifying the affected Pods, it's necessary to determine which communications must be allowed. Ingress rules affect the traffic received from the selected workloads, while egress rules control the connections initiated by them. The two directions should not be considered a single authorization, as they can be configured and evaluated independently.

For example, an application can accept traffic only from a specific frontend while simultaneously communicating with a database or external service. The rules must therefore describe not only who can communicate, but also in which direction, on which port, and via which protocol.

When both Pods involved are isolated, the connection must be compatible with the policies applied at both ends. The source Pod must be authorized to send traffic, and the recipient Pod must be allowed to receive it. A single authorization is therefore not sufficient.

Policies applicable to the same workload all contribute to the final result. Kubernetes does not see them as ordered rules, but combines the allowed flows. Therefore, evaluating a single Network Policy may not be sufficient to determine if communication is possible. This feature makes a global representation useful, capable of simultaneously comparing ingress, egress and relationships between workloads [56].

3.5.5 Network Policy Peers

Allowed sources and destinations are represented by peers. A **peer** does not necessarily identify a single resource, but can describe a set of Pods, a group of namespaces, or an external network. The choice of peer type therefore determines the level of precision of the rule.

The *podSelector* allows you to authorize workloads identified by labels. The *namespaceSelector*, on the other hand, introduces a selection based on namespace labels. The two criteria can be combined to describe, for example, only Pods with a specific function that belong to namespaces classified as trusted.

However, it is important to pay attention to the manifest structure. Placing the *podSelector* and *namespaceSelector* in the same peer means applying both conditions simultaneously. Placing them in separate peers, however, broadens the rule, because alternative sets of sources or destinations are considered. A seemingly minor syntactic difference can therefore significantly change the scope of allowed communication.

The *ipBlock* field is used when the rule must refer to an address range, generally outside the cluster. The **CIDR (Classless Inter-Domain Routing)** notation allows you to define the allowed network, while the except field allows you to remove any subranges. This approach is less suitable for directly identifying Pods, since their addresses are not necessarily stable [56].

In the Digital Twin, peers are transformed into relationships between endpoints and resource sets. This makes it possible to determine whether a policy authorizes communication between two Pods, between different namespaces, or to an external network. Verification is then not limited to the policy's syntactic correctness, but also considers the effect that selectors have on the current cluster topology.

3.5.6 Network Policy Security Risks

Network policies are an important tool for limiting communications between workloads, but their effectiveness depends on how accurately they are defined and applied. A policy that is absent, incomplete, or built on incorrect selectors can leave open connections that the administrator believed to be blocked. This problem is particularly relevant in complex clusters or multi-cluster environments, where pods, labels, and namespaces change over time, making it difficult to manually evaluate the overall effect of the rules and so calling for security requirements to be translated into consistent policies and coordinated across several Kubernetes domains [68].

An overly permissive configuration can encourage lateral movement. After compromising a pod, an attacker could exploit the remaining communications to reach databases, internal services, or other sensitive workloads. The risk therefore arises not only from a total lack of isolation, but also from the presence of policies that only cover a portion of the necessary flows or that authorize larger sets of resources than intended [57].

Implementations such as **Calico** and **Cilium** offer more advanced controls than standard network policies, such as *deny* rules, priorities, global policies, or checks at higher application levels [58], [59]. These features increase protection capacity but also introduce greater complexity. The combination of Kubernetes policies and plugin-specific rules can make it difficult to determine which traffic is allowed or blocked.

That is why it's not enough to verify that Network Policy objects are present in the

cluster. It's necessary to reconstruct the actual authorized flows, considering selectors, namespaces, ports, protocols, and traffic direction [6]. For the Digital Twin developed in the thesis, this reconstruction allows for continuous verification that possible communications comply with the defined security invariants and that a configuration change doesn't introduce new, unwanted access paths.

3.5.7 Network Policies Representation in the Thesis

From an analytical perspective, Network Policies can be represented as relationships between selected workloads and the communications they are authorized to establish. In the DT, Pods, namespaces, labels, ports, protocols, and network policies are translated into nodes and edges of the graph. This allows us to reconstruct not only the rule declared in the manifest, but also the set of connections actually allowed in the current cluster configuration. A possible abstraction of the model is the following:

$$\texttt{Policy} \rightarrow \texttt{Selected Pod} \rightarrow \texttt{Peer} \rightarrow \texttt{Port/Protocol} \rightarrow \texttt{Direction}$$

The selected Pod represents the workload to which the policy applies, while the peer identifies the possible source or destination of the traffic. The direction distinguishes between ingress and egress communications. Since the selection of Pods and namespaces depends on the labels, the model also includes the relationships between resources and labels, to determine which workloads are involved in each rule.

The Digital Twin combines all policies applicable to the same Pod, respecting the additive behavior of Kubernetes Network Policies. It can therefore verify whether communication between two workloads is possible by considering both the egress rules of the source Pod and the ingress rules of the recipient Pod. The same representation also allows for modeling communications to external networks, for example when a rule refers to an IP range or a specific port and protocol.

The **graph** is finally used as the basis for formal verification. Queries can check properties such as isolation between namespaces, access to sensitive workloads, DNS connectivity, or the exclusive use of authorized ports [6]. When an invariant is violated, the system produces evidence relating to the Pods involved, the direction of the traffic, the applied policy, and the detected connection. These results are displayed in the UI and updated whenever Pods, labels, namespaces, or Network Policies are added, modified, or deleted in the cluster.

Chapter 4

Thesis Objectives

The starting point for this thesis is an already existing framework designed to verify security invariants related to the configuration of a Kubernetes cluster. The system translates information regarding resources, RBAC permissions, and network policies into a formal model, which is then used to execute specific security queries. This allows for verification, for example, of whether a subject has excessive permissions, whether a specific operation is permitted on a protected resource, or whether unexpected network communications exist between different workloads.

The original framework bases verification on **static snapshots** of the Kubernetes configuration. The cluster's state is collected at a specific point in time, transformed into a parsable representation, and then provided to the verification engine. This approach allows for a consistent description of the observed configuration, but presents a clear limitation when applied to a dynamic environment like Kubernetes. The verification result is valid only with respect to the state represented by the snapshot and does not guarantee that the same security properties will continue to be respected after a subsequent modification of the cluster.

Kubernetes is a constantly evolving system. Users can create, modify, or delete resources through the Server API, while controllers, schedulers, and automated operators constantly intervene to keep the system's real state aligned with the desired one. A Deployment can be updated, a Pod can be deleted and recreated, a new replica can be started, or a label can be modified. Similarly, an administrator can change the rules of a Role, add a subject to a RoleBinding, or introduce a new Network Policy.

As a result, a correctly executed verification can become invalid immediately after creating, modifying, or deleting a resource. A cluster that appears secure when the snapshot is generated could subsequently enter a non-compliant state, undetected by the static framework until the next snapshot is taken and analyzed. Snapshots therefore introduce potential **time gaps**, during which the real system may be in a risky state,

while the model used for verification continues to represent a previous state.

The duration of these intervals depends on the frequency with which configurations are collected and analyzed. Increasing the snapshot frequency can reduce the problem, but it doesn't completely eliminate it. Furthermore, repeatedly rebuilding and verifying the entire model can introduce significant computational costs, especially when the change affects only a limited portion of the cluster. An update to a single RBAC rule, for example, doesn't necessarily mean that every network relationship or system resource must be re-built from scratch. The main gap to be filled is therefore the transition from a series of isolated checks to a form of continuous, runtime security assurance. The goal is no longer simply to verify that a given configuration is secure at a specific point in time, but to ensure that invariants continue to be evaluated throughout the entire operation of the cluster. Security is considered a property to be maintained over time, not the result of an occasional check. This shift first requires a representation capable of evolving alongside the real system. The verification model must receive information about cluster changes, update the resources and relationships involved, and provide a consistent state on which to rerun analyses. Events generated by Kubernetes become key elements for understanding when the representation is no longer aligned with the cluster and needs to be updated. Analyzing the literature, the concept of a Digital Twin has emerged as a possible solution to support this transition. A Digital Twin allows maintaining a digital representation of a real system, updated through a continuous flow of information from the observed environment. Applied to Kubernetes, this paradigm allows associating the real cluster with a virtual model that follows its evolution and maintains an up-to-date view of resources, identities, permissions, and communication relationships. Activities that require observation of the internal behavior of applications or containers are outside the scope of this work. The system does not perform image vulnerability scanning, malware detection, comprehensive application log analysis, or monitoring of instructions executed by processes. It does not directly address the protection of the underlying cloud infrastructure, physical nodes, or provider components. Having defined the objectives of the thesis, the next chapter presents the proposed approach and its overall architecture, while the following ones will delve into the verification, remediation, and dashboard visualization processes, respectively.

Chapter 5

The Proposed Approach

This chapter presents the approach developed to extend static security verification in Kubernetes into a runtime-aware solution based on a Digital Twin. Having defined the thesis objectives in previous chapters, the focus here shifts to the concrete structure of the proposed solution: how the Kubernetes cluster is mapped to its virtual counterpart, the mechanisms used to keep the two environments synchronized, and how the Digital Twin enables continuous security verification capable of supporting remediation actions.

The discussion begins with the role the Digital Twin plays in the context of Kubernetes security, along with the principles followed to maintain the model's consistency with the cluster's state—which, by nature, is constantly evolving. It then describes the system's overall architecture, the responsibilities of its key components, and the security-relevant information represented by the model. Implementation details regarding individual modules, as well as the specific verification and remediation mechanisms, are deferred to subsequent chapters.

5.1 Digital Twin for Kubernetes Security

The Digital Twin represents the reference paradigm adopted in this work to overcome the limitations of an exclusively static analysis of the Kubernetes cluster. Although the most well-known and established applications of Digital Twins are primarily in industrial and manufacturing environments, such as the aerospace sector, the concept can also be adapted to complex and dynamic software systems. In particular, Kubernetes offers characteristics that make the use of a constantly updated digital representation useful.

In this context, the Digital Twin is not used to reproduce a physical object, but rather to represent the operational and security status of a cloud-native infrastructure. The Kubernetes cluster constitutes the real system being observed, while the twin maintains a

virtual counterpart capable of tracking the most significant variations. This representation allows information that is distributed across different resources and difficult to evaluate separately, to be brought together in a single model.

The concepts introduced in the previous chapters are now interconnected. The twin becomes a tool at the service of the cluster: it observes its evolution, maintains a consistent representation of the current state, and provides the necessary basis for analyzing compliance with security invariants.

5.1.1 Mapping the Kubernetes Cluster to a Digital Twin

Recalling the concepts introduced in Chapter 2, even in the context of this work, it is possible to identify the three fundamental elements that characterize a Digital Twin: the real environment, the virtual environment, and the digital thread.

The **real environment** coincides with the running Kubernetes cluster. It contains resources, configurations, and relationships that can be created or modified either directly by the user, for example, through *kubectl*, or automatically by control plane components. The observed state consists not only of individual objects present in the cluster, but also of the properties that determine their behavior, such as namespaces, labels, selectors, references to other resources, RBAC rules, and network policies.

The **virtual environment**, on the other hand, is the digital representation of the real cluster. In the proposed system, this representation takes the form of a **graph-based model**, in which Kubernetes entities are considered interconnected elements. So, the Digital Twin does not simply reproduce a list of existing resources, but reconstructs the overall structure of the cluster and how the different objects are connected to each other. The use of an intermediate abstraction for security configurations has also been investigated in virtualized network environments, where heterogeneous and technology-dependent rules are translated into a common representation suitable for analysis and automated processing [64].

This feature is particularly important for security verification purposes. The presence of a resource, taken individually, is not always sufficient to determine whether the cluster configuration is secure. The true meaning of an object often depends on the relationships it maintains with other elements. A Pod, for example, can belong to a specific workload, use a specific ServiceAccount, and be selected by one or more Network Policies. Similarly, an entity's permissions do not depend solely on its identity, but on the chain of relationships that connects it to bindings, roles, and permissions.

However, the main focus of this thesis is on the relationships within the two security domains analyzed: RBAC and Network Policies. In the case of RBAC, the representation

must allow us to reconstruct the path that starts from a subject and arrives at the effective permissions through bindings and roles:

`Subject` → `Binding` → `Role` → `Permission`

Instead, in the case of Network Policies the model must connect a policy to the selected Pods, the authorized peers and the characteristics of the permitted communication:

`NetworkPolicy` → `Selected Pod` → `Peer` → `Allowed Communication`

These relationships allow us to move beyond a view based on isolated objects and highlight the security consequences of their joint configuration. These include permissions or unwanted communications.

5.1.2 Synchronization Between the Cluster and the Twin

The third fundamental element of the model is the **digital thread**, which forms the link between the Kubernetes cluster and its virtual counterpart. Its role is particularly important, as it enables the exchange of information necessary to keep the real and virtual environments synchronized. In one direction, changes observed in the cluster are transferred to the Digital Twin and used to update the model. In the other direction, analyses performed on the twin can support decisions and possible interventions on the real configuration, as occurs in the subsequent remediation phases.

Synchronization is a necessary feature in this work, as the cluster continuously generates changes during its operation. Resources can be created, modified, or deleted. Consequently, the Digital Twin cannot be built once and considered valid for the entire duration of its execution. To continue to represent the cluster reliably, it must receive an updated flow of information on changes that occur over time. Events of interest can primarily be traced back to the `ADDED`, `MODIFIED`, and `DELETED` operations.

Since not every change produced by Kubernetes carries the same weight from a security standpoint, synchronization also has to take into account the semantic impact of the observed events. Purely operational updates, which leave the properties represented by the Digital Twin untouched, need to be distinguished from changes that can actually alter the security state of the cluster.

Another reason why synchronization between the real and virtual environments is important is the **re-verification** of security invariants. After a major update to the Digital Twin, the properties associated with the modified resources and relationships must be re-evaluated. This allows you to verify if the latest creation, modification, or deletion

has caused one or more invariants to move from a safe state to a violated state. Timely synchronization makes this process nearly **real-time**, preventing the cluster from being exposed to incorrect security configurations for too long.

5.2 Security-Oriented Role of the Digital Twin

In the proposed system, the Digital Twin plays a role specifically designed for cluster security. The representation maintained by the twin serves as the foundation for monitoring and verification activities, focusing particularly on the two areas addressed in this thesis: RBAC permissions and communications governed by Network Policies.

The cluster state represented in this way is utilized by the formal verification framework to evaluate security invariants and identify non-compliant configurations. The information within the model also aids in interpreting detected violations and serves as a starting point for considering potential corrective actions; thus, the Digital Twin acts as an intermediary between cluster observation and the subsequent verification and remediation phases, which are examined in detail in dedicated chapters.

System's results are presented via a dashboard, which serves as the primary interface for interacting with the Digital Twin. This dashboard aggregates and organizes data regarding the cluster state, observed events, security invariants, detected violations, and corresponding remediation options, thereby providing the user with a unified view of the results from the various stages of the process.

5.3 Architectural Organization of the System

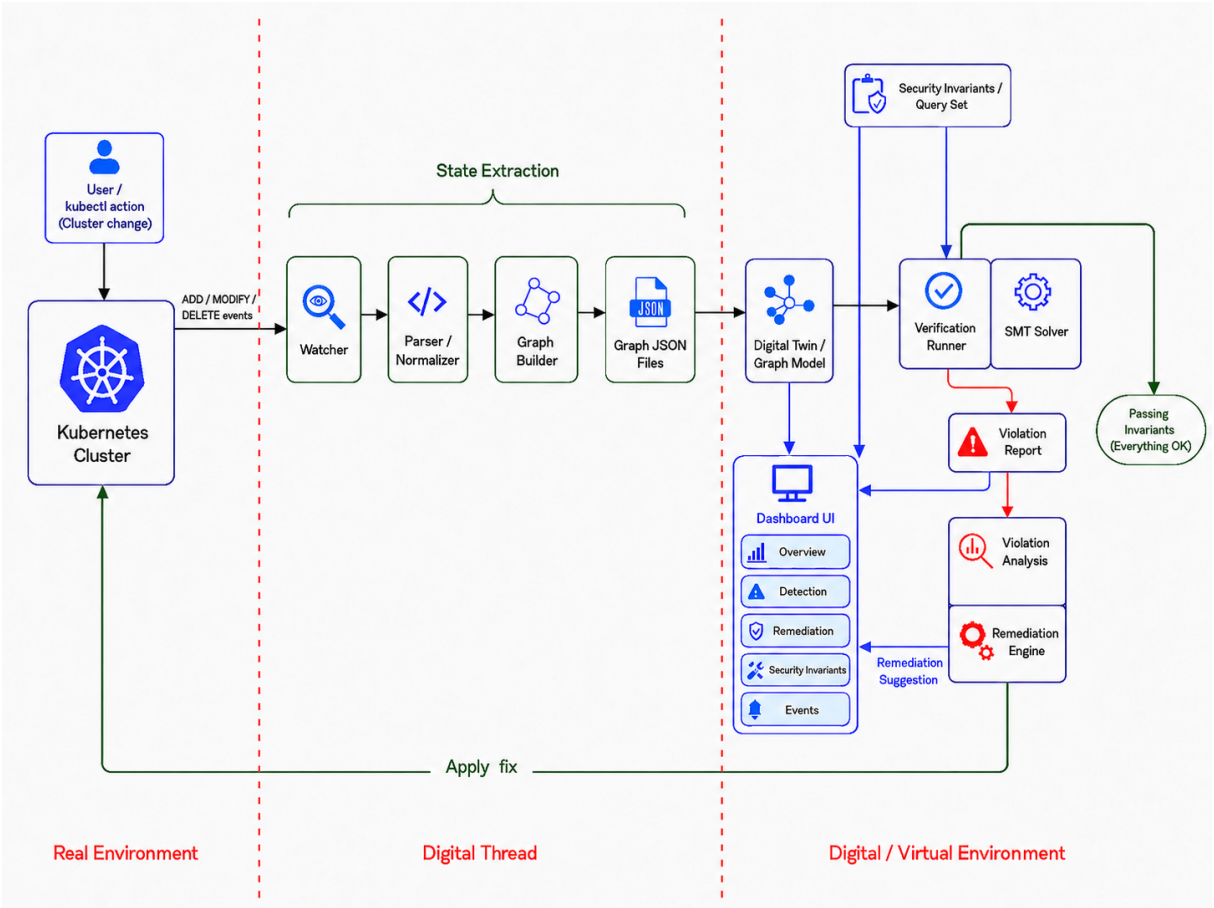


Figure 5.1: Architecture of the Kubernetes Digital Twin model.

The project's architecture is designed as a **pipeline** in which each module falls into one of the three macro-environments already introduced: real environment, digital thread, and virtual environment. This division allows for the placement of the different components within the system, but the pipeline does not follow a purely linear path. The process takes on a **cyclical** structure when the results produced in the digital environment are used to intervene on the real cluster and the new state is subsequently observed. This characteristic stems from the use of a model based on the Digital Twin.

The pipeline starts from the state of the Kubernetes cluster. The information collected from the real environment is not sent directly to the verification engine, but undergoes a series of transformations. Events are acquired, standardized, and used to maintain an updated representation of the observed resources. From this state, a structured model is then built, suitable for both automatic analysis and visualization.

Each step prepares the data for the next. A single Kubernetes event describes only one change that occurred in the cluster. To assess the security configuration, it is necessary to

reconstruct the overall state of the resources and the relationships between them. The modeling process then transforms a sequence of events into a coherent representation of the current system.

The Digital Twin occupies the central position of the architecture. It serves as a collection point for processed information on the cluster's state and organizes it into a security-oriented graph model. It not only represents the existence of Kubernetes resources, but also includes the relationships between workloads, namespaces, identities, RBAC permissions, and network policies.

This position makes the Digital Twin the bridge between the observation process and the analysis phases. The model produced from the cluster is used as the basis for verifying security invariants and identifying potential violations. The information displayed in the dashboard also derives from the DT graph or from the results generated by the components that use it. The architecture can then be traced back to four main functions: observation, modeling, verification, and remediation.

Figure 5.1 shows the overall structure of the system. Following the figure from left to right, the process originates in the Kubernetes API Server, which exposes the cluster's state and events associated with resources. The watcher intercepts relevant changes, while the normalization layer converts heterogeneous events into a stable internal form. This information updates the runtime state maintained by the system and is then processed by the graph builder to materialize the resulting Digital Twin into a set of JSON files.

The resulting graph is provided to the verification process along with the set of security invariants. The verification runner coordinates the execution of queries through the SMT solver and collects the resulting outputs. Unsatisfied properties are transformed into structured violations, including the information needed for their interpretation and for subsequent system phases, and put into a violation report.

The pipeline continues with the dashboard and the remediation engine. The dashboard allows the operator to consult the graph, events, the state of invariants, and active violations. The remediation engine uses the information associated with a violation to carry out a violation analysis and select possible corrective actions.

The flow returns to the cluster when a remediation passes the required checks and is approved by the user. The change is applied to the live environment, generating new Kubernetes events. The watcher observes the new state, the digital twin is updated, and the invariants are re-evaluated. Closing the loop allows verification that the intervention has effectively removed the violation and that the digital model is once again aligned with the cluster.

5.3.1 Architectural Boundaries and Responsibility Distribution

Along the pipeline, data changes shape and meaning. Transformations are not only related to the two environments in which it may be located or the communication channel, but also based on the module receiving the data and the task it must perform. For this reason, architectural boundaries coincide not only with the three macro-environments, but they also appear between individual levels of the pipeline.

The first separation concerns the source of truth and the derived representation. The Kubernetes cluster is the **authoritative source** of the real state: it contains the resources actually present, and the configurations applied by the control plane, users, and controllers. The Digital Twin does not replace this source and cannot be considered autonomously authoritative. It maintains a representation built from data observed through the Kubernetes API Server and reprocessed during the state extraction phase. This distinction becomes particularly relevant when evaluating remediation. A change introduced in a virtual scenario describes a possible future state, but does not modify the cluster or update the main Digital Twin. The actual state changes only after the fix is applied to the Kubernetes API Server and the subsequent update produced by the watcher.

A second boundary separates raw data from **normalized events**. The watcher receives events from different APIs and resources, each with its own structure and specific fields. This data is strictly tied to the native Kubernetes format and is not yet suitable for direct use by the rest of the pipeline. The *parser/normalizer* converts it into a more stable representation, in which the resource identity, operation type, namespace, labels, and necessary references are exposed according to a common schema. In this step, the scope changes to that of the individual modules: the *watcher* handles the observation, while the *normalizer* makes the information usable regardless of the type of resource that generated the event. This separation prevents the graph updater from directly depending on the differences between the various Kubernetes APIs.

The next boundary concerns solely the digital environment. Here, the concept shifts from what data is to how it is used. For the DT graph, the normalized data collected in JSON files is used for modeling, while for the subsequent step those JSON files are used to conduct reasoning in the verification phase: they are loaded by the verification layer to build the **Knowledge Base** used by the solver.

All this allowed us to employ the already existing *SMT solver*. The original framework was designed to load a serialized representation of the cluster and verify queries against RBAC and Network Policies. The new architecture does not embed the solver in the watcher and does not rewrite its internal logic. Instead, the Digital Twin produces data compatible with the format expected by the Knowledge Base, while the verification runner

manages the connection between the runtime model and the formal engine. In this way, the previous work remains separated from the new event-driven part and can continue to perform its task without knowing how the watcher, dashboard, or remediation works.

A further separation exists between analysis and actuation. It has already been investigated in virtualized networks, where security services can first be defined at the functional level and only later mapped to specific implementations and deployment choices [65]. Here, the solver evaluates invariants and produces results, but does not have the ability to modify the cluster. Even a correctly identified violation does not automatically trigger an intervention. Application on the cluster is a separate phase, preceded by specific checks and user confirmation. This separation reduces the risk that an error in the queries, in the interpretation of the results, or in the generation of the candidate remediations will directly result in a modification of the real environment.

The quintessential separation is found in the UI. The virtual environment's modules act as a **backend** that processes and makes fetchable the graph, events, verification results, and violations. The dashboard, on the other hand, acts as a **frontend** and accesses this information via the **Node API**. It does not read Kubernetes objects directly and does not build the model autonomously. During remediation, it only sends the identifiers and the confirmation required by the workflow; patches, manifests, and operation plans remain on server-side. Therefore, the UI does not have a direct channel for applying arbitrary changes to the cluster.

The distribution of responsibilities reduces dependencies between components. The watcher can change the way it observes Kubernetes without requiring changes to the solver, as long as it continues to produce events in the expected format. The graph model can be extended with new relationships, and similarly, the interface can change its presentation mode without interfering with formal verification. This **modular structure** makes it easier to detect errors and test individual phases, because each component receives a defined input and produces a recognizable output.

The boundaries described also introduce greater control over the data path. The cluster is kept apart from simulations, reasoning remains separated from the application of fixes, and the UI is detached from the logic that manages Kubernetes changes. In addition to improving maintainability, this design limits the number of components authorized to act on the real system and makes it clearer at what point in the pipeline a piece of information is observed, transformed, interpreted, or used to make a decision.

5.4 Security-Relevant Projection of the Kubernetes State

The Kubernetes cluster contains a much larger amount of information than necessary for the objectives of this work. In addition to security configurations, the API Server exposes data related to scheduling, container status, pod health, operational metrics, controllers, endpoints, and other aspects of cluster operation. A complete replication of all these elements would make the model more cumbersome, without providing any tangible benefit in verifying the properties considered in this thesis. The Digital Twin, so, adopts a targeted projection of the Kubernetes state, built around the information needed to represent RBAC, network policies, and structural relationships between resources.

The scope of the collection does not derive solely from the availability of data in the API Server. It is primarily driven by the type of queries the verification framework must be able to execute. The verification layer requires a representation composed of resources, subjects, roles, grants, bindings, object references, labels, group memberships, and network policies. The *watcher* and *graph builder* must therefore produce data compatible with this structure so that the solver can reconstruct the knowledge base and evaluate security invariants.

For workloads, the mere presence of the Pod is not enough. Information is needed to correctly place it in the cluster topology and reconstruct its relationships with other resources. The namespace defines the object's scope, while labels and selectors establish memberships, associations, and logical groups. Owner references allow tracing the individual instance back to the workload that generated it, such as a Deployment, a StatefulSet, or a DaemonSet. The ServiceAccount associated with the Pod connects the workload to the RBAC part of the model.

The information collected for RBAC is more complex, because the actual permissions depend on the combination of different objects. The model must know the subjects involved, like Users, Groups, and ServiceAccounts, along with Roles, ClusterRoles, RoleBindings, and ClusterRoleBindings. Rules describe the granted permissions, while *roleRef* links each binding to the corresponding role. The scope determines whether the permission applies within a namespace or across the entire cluster. The representation must also take into account the aggregations between ClusterRoles and user group memberships, otherwise the privilege calculation is incomplete.

The *graph builder* materializes these relationships through distinct structures. `Bind_edges` connect subjects and roles, `grant_edges` associate roles with permitted actions, `aggregate_edges` represent inheritance between ClusterRoles, while `contains_edges` and `subject_group_map` reconstruct hierarchical relationships and

group membership, respectively. The presence of separate files is not simply a serialization choice. Each relationship describes a specific step in calculating the effective permission and is then loaded into the knowledge base used by the solver.

The network policies projection requires a different set of attributes. The system must know which Pods are selected by a policy, which peers are allowed, the traffic direction, ports, protocols, and the presence of ipBlocks. Pod and namespace selectors play a central role, because the policy does not refer to stable addresses or specific instance names, but to sets of resources identified through labels. To reconstruct the actual effect of a rule, it is therefore necessary to compare the policy configuration with the current state of the Pods and namespaces.

The Network Policy normalization module supports podSelector, namespaceSelector, a combination of the two, ipBlock, ingress, egress, protocols, ports, port ranges, and the semantics of `allow-all` and `deny-all`. The policy is also identified through its namespace, preventing identical selectors in different namespaces from being merged into a single entity. This detail is necessary because the scope of a Network Policy is always namespaced, and the same set of labels can have different meanings in two separate parts of the cluster.

Not all updates produced by Kubernetes have the same weight in the model. The control plane frequently modifies state fields to record operational conditions, controller progress, or resource lifecycle information. Treating every change as a security change would lead to unnecessary reconstructions and checks. For `MODIFIED` events, the system then compares fields that can alter the meaning of the queries: Pod labels and owner references, ServiceAccount, namespace labels, Service selectors and ports, RBAC rules, subjects, *roleRef*, workload selectors and Network Policy specifications.

This distinction allows us to separate operational changes from modifications that could alter a permission or communication path. An update to a Pod's status, for example, does not necessarily alter its RBAC or NetworkPolicy semantics. A change to its labels, however, can cause it to enter or exit the scope of a policy. Similarly, a change to the subject list of a RoleBinding can assign new privileges, while an update with no impact on the rules or roleRef does not require the same attention.

Secrets are treated with a specific criterion. For the Digital Twin, it can be useful to know that a Secret exists, the namespace it belongs to, and the relationships bonding it to RBAC workloads or permissions. The content of the Secret, however, is not essential to verify the invariants considered. The *data* and *stringData* fields are then removed before events are saved or forwarded. In this way, the model preserves structural information without propagating passwords, tokens, keys, or other sensitive values along the pipeline.

A different limitation concerns **Users** and **Groups**. ServiceAccounts are Kubernetes resources and can be observed directly through the API. Users and groups, on the other

hand, often rely on external authentication systems and are not exposed as normal cluster objects. They can appear in bindings, but Kubernetes does not necessarily provide a complete list of known identities or their memberships. To address this, the system offers an optional `identities.json` file, used to provide users, groups, and associations that cannot be derived from observed streams. Without this file, the model continues to operate with an empty set of users and groups, while ServiceAccounts remain available via the API.

Subresources also require a distinction. Elements like **pods/exec**, **pods/log**, or **deployments/scale** are relevant to RBAC because a role can grant specific operations on these endpoints. However, they do not have an independent lifecycle that can be observed via a watch stream. Therefore, they are modeled as authorization targets, when necessary, but not as standalone objects collected by the watcher. Their presence in the Digital Twin derives from permission semantics and not from a dedicated event.

The result of all these choices is a representation that is smaller than the real cluster, yet rich enough for formal reasoning. The projection preserves the objects and attributes that influence the invariants, eliminates not required data, and integrates information that Kubernetes doesn't directly expose. This makes the model more manageable and reduces the noise generated by operational updates. It also improves the readability of the graph structure, because the shown relationships correspond to dependencies or security effects.

The following three chapters will finally dig into the technical details behind all the modules shown in the architecture, offering a clear view of how they work, how they interact and how they had been implemented.

Chapter 6

Implementation of the Runtime Digital Twin

We now move from the architectural organization of the Digital Twin and the responsibilities assigned to the various system blocks to the concrete implementation of the first part of the workflow, following the path that originates in the Kubernetes cluster and ends with the generation of the violation report. The focus is on the software modules, the formats used for data exchange, and the mechanisms through which the observed state is transformed into a model suitable for formal verification.

The main entry points for this part of the pipeline are `watcher.py`, `graph_updater.py`, and `verification_runner.py`, alongside modules dedicated to resource configuration, event normalization, graph construction, and output management. The Verification Runner combines the current Digital Twin with the set of invariants and coordinates query execution through the external SMT solver. The results are organized in `verification_report.json`, while the unsatisfied properties are converted into structured objects and collected in `violations.json`.

The implementation contribution described in this chapter consists of introducing a runtime layer around the existing verification framework. The formal engine retains the task of building the knowledge base and performing SMT reasoning; the new components instead allow it to be fed with an updated representation of the cluster and integrated into the continuous workflow of the digital twin.

6.1 Software Components and Implementation Environment

The software components developed and integrated into the project can be traced back to the main phases of the runtime pipeline. The first group concerns the **acquisition** and **normalization** of Kubernetes events and includes `watcher_config.py`, `watcher.py`, `event_models.py`, and `event_normalizer.py`. These modules define the resources to be observed, configure access to their respective APIs, receive events generated by the cluster, and convert them into a common internal representation.

The next phase is dedicated to maintaining the **runtime state** and **building** the Digital Twin. The `graph_updater.py` module maintains the latest known version of the observed objects and coordinates the process leading to the reconstruction of the **ClusterGraph**. The parsing logic is primarily handled by `serializer.py`, which extracts resources, namespaces, ownership relationships, labels, ServiceAccounts, and RBAC configurations. NetworkPolicy processing is completed by `serializer_netpolicy.py`, which is responsible for converting selectors, ingress and egress rules, peers, protocols, and ports into the format used by the model. Finally, `graph_core.py` writes the files that materialize the current state of the Digital Twin.

The modules dedicated to cluster observation and graph construction comprehensively implement the **State Extraction** phase introduced in the previous chapter. Individual events received by the watcher are not directly used by the solver, as each describes only a limited variation. The system applies them to the current runtime state and subsequently reconstructs a consistent representation of the entire set of security-relevant resources and relationships.

The **verification** phase is coordinated by `verification_runner.py`, which **orchestrates** the activities between the materialized Digital Twin and the pre-existing formal framework. The solver remains a separate component from the *watcher* and the *graph updater*.

The primary language used for the backend is **Python**. This choice maintains compatibility with the official Kubernetes client and allows for direct integration with the SMT framework modules, which are also developed in the same language. Python also facilitates the management of objects returned by the Kubernetes APIs and the manipulation of documents used to exchange data between the different stages of the pipeline.

Dependencies are declared in the `requirements.txt` file. The *kubernetes>=29.0.0* package provides the Python client used to connect to the cluster, query the APIs, and open the watch streams used by watcher.py. *PyYAML>=6.0.0* provides the functionality

needed to read and produce **YAML** documents, a format commonly used for Kubernetes manifests and configurations. The *z3-solver>=4.13.0* dependency makes Z3 available within the Python environment.

JSON and **JSONL** formats are used for data exchange and persistence. The normalized event stream is recorded in `cluster_events.jsonl`, where each line contains an independent JSON document. This organization is suitable for an updated append log, since the arrival of a new event requires the addition of a single line rather than rewriting the entire content. Regular JSON files are used to represent complete structures. This category includes the materialized state of the graph, the set of invariants, the report produced by the verification, and the list of any violations. Some files constitute the output of one phase and become the input for the next. Together, they form an explicit and inspectable data contract between the *graph updater*, *verification runner*, *dashboard*, and *remediation engine*.

6.2 Kubernetes Event Watcher

Going into detail about the architecture modules, we begin with the *watcher*, essential for an event-driven, runtime model. It serves as a bridge between the real environment and the DT. It's no coincidence that it's the starting module for the digital thread and the state extraction phase.

6.2.1 Resource Configuration, API Integration and Stream Management

The *watcher*'s very first task is to connect to the Kubernetes API Server and, through watch requests, consume the streams produced by it, rather than having to periodically query the cluster with traditional API requests, which terminate the connection once satisfied. When a resource is created, modified, or deleted in the cluster, the API sends an event to the relevant stream, consisting for example of the following:

```
{
  "type": "MODIFIED",
  "object": {
    "apiVersion": "v1",
    "kind": "Pod",
    "metadata": {
      "name": "backend",
      "namespace": "app"
    }
```

```
  }
}
```

The 'object' contains not only the modified field, but also the updated representation of the resource, including the metadata and sections required by the relevant Kubernetes schema.

Specifically, the *watcher* configuration is included in a support file called `watcher_config.py`, which defines the top-level resources for which streams will be opened: Pod, Service, ServiceAccount, ConfigMap, Secret, Namespace, Node, Deployment, ReplicaSet, StatefulSet, DaemonSet, Role, RoleBinding, ClusterRole, ClusterRoleBinding, and Network Policy. Sub-resources such as pods/log, pods/exec, and deployments/scale are modeled as RBAC targets if required, but are not considered independent resources by the watcher, so events involving them are considered non-relevant. It maps each resource type to the correct Kubernetes API using structures such as:

```
"pods": (
    client.CoreV1Api,
    "list_pod_for_all_namespaces"
)
```

Here, the K8s Python client class and the corresponding watch/list method name are specified. The same mechanism, which applies to other resources, is summarized in the following table:

API group	**Python client**	**Resources**
`core/v1`	`CoreV1Api`	Pod, Service, ServiceAccount, ConfigMap, Secret, Namespace, Node
`apps/v1`	`AppsV1Api`	Deployment, ReplicaSet, StatefulSet, DaemonSet
`rbac.authorization.k8s.io/v1`	`RbacAuthorizationV1Api`	Role, RoleBinding, ClusterRole, ClusterRoleBinding
`networking.k8s.io/v1`	`NetworkingV1Api`	NetworkPolicy

Table 6.1: Kubernetes API groups, Python clients, and watched resources.

Another feature of the configuration file is the distinction between namespaced and cluster-scoped resources, which will later be used during event normalization to determine the canonical key.

`watcher_config.py` works in conjunction with `k8s_metadata.py`. The latter contains the general properties of supported resources, including their kind, API group, and whether they are namespaced or cluster-wide in scope. The `build_watch_specs()` function combines this metadata with the API class and method defined in the configuration, producing a list of **WatchResourceSpec** objects. This structure avoids having to include a sequence of conditions specific to each object type in `watcher.py`. The *watcher* can apply the same logic to all resources, dynamically retrieving the class and method specified in the resource specification.

As a result, `watcher.py` can use the official Kubernetes Python Clients to open a watch stream for each resource type. Specifically, its tasks include supporting local *kubeconfig* and in-cluster configuration, resuming streams from the latest resource version when possible, managing `Kubernetes '410' Gone` by resetting the resource version, and reconnecting after stream failures. Its most important role, however, is appending events to the JSONL file called `cluster_events.jsonl`. This file serves as an event log and is calibrated to record the last 3,000 events in a structure called a deque: if the maximum capacity is reached, inserting the last line deletes the oldest record. Access to the file is protected by a lock, preventing different streams from writing to the same output simultaneously.

6.2.2 Event Normalization

The final step in the observation and acquisition phase is to normalize the collected events (using the event_`normalizer.py` file, part of the Parser/Normalizer module), making them suitable for model building. The main reason this step is necessary is that the information coming from the various Kubernetes APIs doesn't have exactly the same formats. Thus, regardless of the source resource, all events are transformed into the same **NormalizedK8sEvent** structure. This structure is unique and, taking the example of a pod modification, would look like this:

```
{
  "event_type": "MODIFIED",
  "resource_type": "pods",
  "kind": "Pod",
  "api_version": "v1",
  "name": "backend",
  "namespace": "app",
  "resource_key": "Pod@app#backend",
  "uid": "9f32e2b7-2b1d-4f40-a4b6-01d82f5b1280",
  "resource_version": "12346",
  "labels": {
    "app": "backend",
```

```
      "tier": "api"
    },
    "owner_references": [
      {
        "apiVersion": "apps/v1",
        "kind": "ReplicaSet",
        "name": "backend-7d8f5b9c6",
        "uid": "1b2c3d4e-..."
      }
    ],
    "raw_object": {
      "apiVersion": "v1",
      "kind": "Pod",
      "metadata": {
        "name": "backend",
        "namespace": "app",
        "uid": "9f32e2b7-2b1d-4f40-a4b6-01d82f5b1280",
        "resourceVersion": "12346",
        "labels": {
          "app": "backend",
          "tier": "api"
        }
      },
      "spec": {
        "serviceAccountName": "backend-sa"
      },
      "status": {
        "phase": "Running"
      }
    },
    "observed_at": "2026-07-21T09:42:18.421367+00:00"
  }
```


The most interesting field generated is undoubtedly *resource_key*, which uniquely identifies the resource in the digital environment. It is created using the object's kind, its namespace, and its name, separated by '@' and '#', respectively. In the case of a cluster-scoped resource, the namespace name is simply replaced by '*'.

The list of the various canonical event types has been specified in the `event_models.py` file:

- `ADDED`, creates a resource;

- `MODIFED`, updates one or more fields of an object;
- `DELETE`, removes a resource from the cluster;
- `BOOKMARK` and `ERROR`, handle stream management.

We can think of this as the configuration file, which defines how a normalized event should be made, while `event_normalizer.py` takes the raw Kubernetes event and actually transforms it into the canonical format by: converting the Kubernetes client object into a serializable dictionary, reading metadata, kind and apiVersion, extracting uid and resourceVersion, normalizing *label* and *ownerReference*s, building the *resource_key* and removing sensitive Secret values.

6.3 Runtime State Management and Graph Materialization

Once the NormalizedK8sEvent is ready, it must be delivered to the *graph builder* architectural module, which is responsible for reflecting the event on the internal state of the Digital Twin.

To ensure a runtime and constantly updated representation of the cluster, the entire event history will not be considered, but only the latest version of each distinct resource. This way, each new event modifies only the affected portion of the state, making the update incremental with respect to the current representation of the cluster.

6.3.1 In-Memory Object Store

The entry point for the entire module is `graph_updater.py`, which implements one of the system's most important classes: **`IncrementalGraphUpdater`**. After receiving normalized events, it maintains a dictionary structure in memory in which the resources' unique resource keys are associated with their most recent representations.

Events are received by the `handle_event()` method, which uses the resource_key as an index to retrieve the version, if already present, and then applies the operation specified by the event_type semantics: `ADDED` generates a new entry for the resource, `MODIFIED` replaces the representation associated with the resource key of the object in question, and `DELETED` causes the record to be removed from the dictionary.

An interesting feature at this stage is concurrency management: since events may arrive at the same time from different watch streams, access to the store is protected by an

RLock, which allows access to shared data to be synchronized between multiple threads. Furthermore, each modification sets a '_dirty' flag to signal that the state in memory has changed. A worker daemon then periodically checks the flag (with a debounce value of one second) and, if necessary, initiates the DT materialization update.

The current snapshot of the objects maintained in the store will be the input for the next graph parsing and reconstruction phase (or possibly construction if this is the initial system startup).

6.3.2 Cluster Parsing and Graph Reconstruction

The parsing stage uses the `serializer.py` module, specifically the `K8sClusterParser` class, which receives store objects and initiates a kind-based interpretation, considering the data required to represent the cluster, such as namespace, ServiceAccount, labels, ownership relationships, RBAC elements, and Network Policy.

The output of this operation is a translation of the cluster objects into explicit entities and relationships within a ClusterGraph. This is the internal structure of the Digital Twin that centralizes resources, subjects, roles, and relationships indicating dependencies, communications, and authorizations in a single model. With this representation, the cluster becomes semantically queriable for the subsequent verification phase. More specifically, the resources take the form of nodes in the graph, and the various relationships become its edges. One example is *ownerReferences*, which allow the reconstruction of connections between workloads such as Pods, ReplicaSets, or Deployments.

An even more relevant case is certainly that of RBAC. Consider, for example, the following Role:

```
kind: Role
metadata:
  name: pod-reader
  namespace: app
rules:
- apiGroups: [""]
  resources: ["pods"]
  verbs: ["get", "list"]
```

This will be represented in the graph as a node with the key 'Role@app#pod-reader', while the associated rule will become a grant edge toward the target (specified here as all pods in the 'app' namespace). The system is implemented to clearly distinguish the relationship between the role and the targets of interest from that of the RoleBinding,

which will then assign the privilege to a subject. This preserves the semantics of RBAC configurations in Kubernetes.

To manage network policies, a second serialization module called `serializer_netpolicy.py` was needed. This is because K8s Network Policies can often contain nested structures based on *podSelector*, *namespaceSelector*, *ipBlock*, *ingress* or *egress* direction, *protocols*, and *ports*. To be able to evaluate network policies during the verification stage, they have been reworked to obtain a more regular logical representation. For example, the selector:

```
podSelector:
   matchLabels:
      app: backend
```

Viene convertito dal file nel formato canonico seguente:

```
{
  "op": "IN",
  "key": "app",
  "values": ["backend"]
}
```

Another interesting aspect is that the parser supports combinations of Pods and namespace selectors, IP blocks, port intervals, allow-all, or deny-all. Furthermore, by combining the namespace with the normalized selector, it is possible to prevent two distinct network policies with the same selector but belonging to different namespaces from being considered the same entity. If within the same namespace, policies regarding the same set of Pods fall under the additivity property of network policies in K8s.

For all relationships that cannot be resolved when the first object is parsed because the referenced resource will be encountered later, the graph construction phase includes a final phase where all dependencies that required a complete view of the state are resolved.

6.3.3 Materialized Digital Twin Files

Once the ClusterGraph has been built, the model can be converted into a persistent representation using the serialization logic implemented in `graph_core.py` using the `save_graph_to_files()` method. This method allows you to save the graph information processed so far in a dedicated output directory. This will include a set of JSON files separated by category and content, as shown in the table:

Category	File	Main content
Entities	`resources.json`	Kubernetes resources and resource buckets
Entities	`subjects.json`	Users, Groups and ServiceAccounts
Entities	`roles.json`	Roles and ClusterRoles
RBAC relations	`grant_edges.json`	Permissions from roles to resource targets
RBAC relations	`bind_edges.json`	Associations between subjects and roles
RBAC relations	`aggregate_edges.json`	ClusterRole aggregation relationships
Structural relations	`contains_edges.json`	Containment between buckets and resources
Structural relations	`reference_edges.json`	Ownership and reference relationships
Indexes	`label_resources.json`	Mapping between labels and resources
Indexes	`subject_group_map.json`	Subject and group memberships
Networking	`network_policies.json`	Normalized NetworkPolicy rules

Table 6.2: Materialized Digital Twin files and their main content.

The model division into multiple files stems from previous work, but it's also well-suited to this project because it allows the information to be used separately from subsequent components in the pipeline. It's important to note that these aren't just copies of Kubernetes manifests, but rather data produced by normalization, parsing, and serialization operations. For example, consider an element in the `grant_edges.json` file, which describes the privilege associated with a role:

```
{
  "role": "Role@app#pod-reader",
  "target": "Pod@app#*",
  "verbs": ["get", "list"],
  "resource_names": []
}
```

As mentioned earlier, these files were not developed from scratch, but were already part of a pre-existing verification framework, where they were used as input for the SMT Solver module (which was also incorporated into this system). Ultimately, this reiterates the purpose of this thesis: integrating a DT model into an already functioning verification system and providing it with runtime behavior, adapting to its input and output formats.

The difference now lies in the lifecycle of these files. Previously, they were the result of parsing performed on static snapshots or objects acquired in batch mode. Now, with the help of the watcher and the runtime state, they materialize the current state of the DT and, when the ClusterGraph is rebuilt following new events, they are updated and aligned with the most recent cluster configuration.

6.4 Runtime Verification and Violation Detection

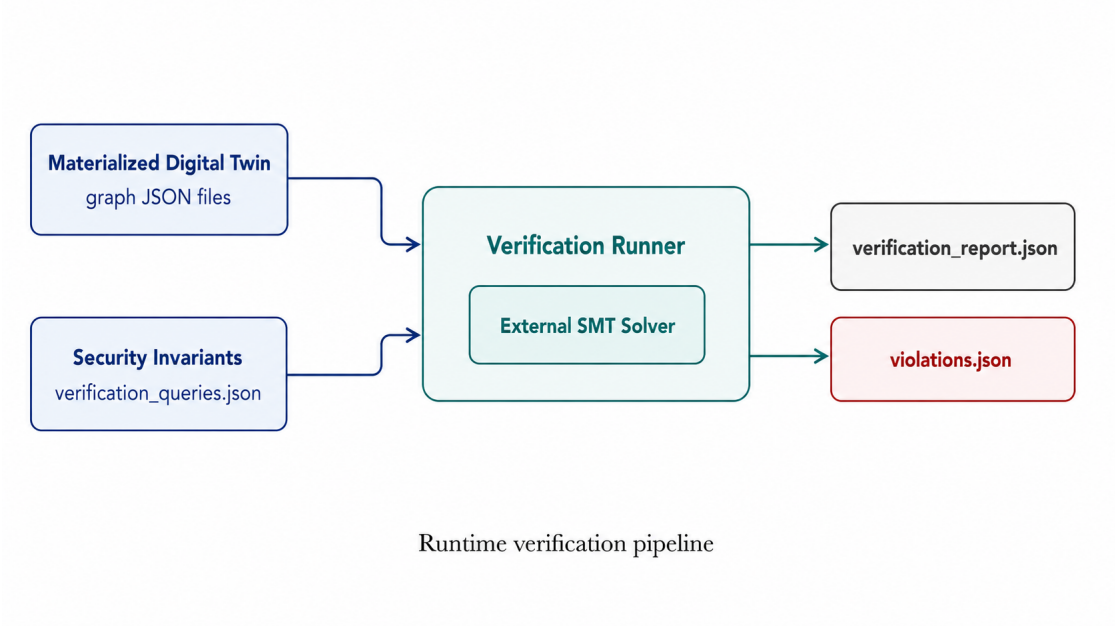


Figure 6.1: Runtime verification pipeline.

Materializing the Digital Twin is certainly the core of this thesis, but as stated from the outset, this model is a means to an end: runtime verification. As shown in Figure 6.1, the JSON files representing the graph, along with the set of security invariants, feed the formal verification module. The former is the object of the analysis, and the latter includes the conditions to be verified.

The central element of this phase will be the *verification runner*, whose task is to coordinate and integrate the pre-existing solver framework into the runtime context of this thesis.

The verification process will then output two JSON reporting files to track the complete execution outcome and isolate the violated invariants, which will then be passed on to the next remediation phase.

6.4.1 Verification Triggering and Query Set Management

Analyzing the verification management details, it's worth mentioning that automatic execution of the SMT verification must be enabled via

```
--enable-verification
```

Because it is managed by the *graph updater*, it can function perfectly well as a module for updating the Digital Twin based on cluster changes. Verification triggering must

therefore be explicitly configured at startup. This choice stems from the ability to separate graph construction and update from the solver, if necessary. The system can therefore be configured to operate only in observation mode.

When explicitly configured, verification is not triggered by any event. For example, the numerous updates produced by the cluster regarding the operational status of resources do not affect RBAC or Network Policy relationships; therefore, they become irrelevant to the project's objectives because they would trigger unnecessary verifications. The system will only consider events with certain types: Namespace, Pod, Service, NetworkPolicy, ServiceAccount, Role, ClusterRole, RoleBinding, ClusterRoleBinding, Deployment, ReplicaSet, StatefulSet, and DaemonSet. When events involve ConfigMap, Node, or Secret, the model is updated, but not verified.

Considering relevant resources, an `ADDED` or `DELETED` event will always trigger a new verification, since an object has been introduced or removed from the cluster, which can impact invariants. However, for `MODIFIED` events, the situation is different: before replacing the resource in the store, the *graph updater* has access to the old and updated versions and can evaluate whether the modified fields impact security.

Resource kind	**Compared fields**
`Pod`	labels, owner references, ServiceAccount
`Namespace`	labels
`Service`	selector and ports
`NetworkPolicy`	policy specification
`Role` / `ClusterRole`	rules
`RoleBinding` / `ClusterRoleBinding`	subjects and `roleRef`

Table 6.3: Examples of semantically relevant fields compared for MODIFIED events.

The table shows examples of resources and their fields that would trigger a new check if modified. For example, if a Pod goes from 'Pending' to 'Running', permissions or relationships in the DT do not change. Conversely, changing the *serviceAccountName* from 'default' to ''privileged-sa' can change the workload's privileges. Considering a Network Policy, changing a label could include or exclude a Pod from the rule's scope.

However, when a semantic change is detected, the solver is not activated immediately. An internal flag records the time of the last relevant change and allows the check to be started only after the debounce interval expires (the default value is three seconds). This serves to consolidate a potentially close sequence of changes into a single execution, avoiding a cascading series of checks.

Another trigger for the solver, in addition to cluster changes, is modifying the `verification_queries.json` file, which contains all the security invariants to be monitored. If a new query is entered, the security state must necessarily be re-evaluated.

6.4.2 Security Invariant Representation

Security invariants belong to the `verification_queries.json` module. Each is modeled to have its own formal query, along with other metadata for identification, processing, and interpretation, just like the following example:

```
{
  "queries": [
    {
      "id": "dangerous_role_can_delete_test_pod",
      "description": "Detect whether a Role grants delete permission on the
    protected Pod.",
      "query": "list (*) from [[Role(Role@default#dangerous-role)] has
    Permission([delete], Pod@default#protected-pod)]",
      "violation_when": "non_empty",
      "severity": "high",
      "category": "RBAC",
      "message": "The Role can delete the protected Pod.",
      "user_defined": false
    }
  ]
}
```

Here, for example, the query checks for the presence of the delete permission on the protected Pod within the indicated Role.

Field	Technical role
`id`	Stable identifier used across queries, results and violations
`description`	Human-readable explanation of the property being checked
`query`	Expression written in the solver DSL
`violation_when`	Rule used to interpret the solver result
`severity`	Impact level assigned to the invariant failure
`category`	Security domain, mainly `RBAC` or `NetworkPolicy`
`message`	Text associated with the detected violation
`user_defined`	Indicates whether the invariant was created through the UI

Table 6.4: Fields used to represent security invariants.

The table helps shed light on the various metadata present in the invariants. The most important field is undoubtedly *query*, which is only stored at this stage. If the *id* field is empty, the *verification runner* assigns one based on the query's position in the file.

The *violation_when* field defines the interpretation of the results collected by the query. Its values can be 'non-empty', 'empty', 'true', 'false', and 'never'. Looking more closely, 'non-empty' returns a violation if the query returns at least one result, while 'empty' works the opposite. 'True' and 'false' are for interpreting boolean results, and 'never' is used for informational and debugging queries that do not result in violations. The default value if the field is empty is 'non-empty'.

Severity and *category* are used to classify the result and support the detection and remediation phases (this metadata is also present in the *verification runner*'s output files). The first has values ranging from 'low' to 'critical', through 'medium' and 'high' (there's also an info value for debug queries), while the second separates 'RBAC' checks from 'NetworkPolicy' checks.

One last interesting data is *user_defined,* a boolean value indicating whether the invariant was written by the user using the dedicated section in the dashboard. The only reason this field was included is that user-defined invariants can be deleted directly from the dashboard. Those entered, for example by an administrator, directly in the `verification_queries.json` file should not be deletable. If the user were to incorrectly compile a new invariant, it would be evaluated and assigned an error status, without generating any violations.

6.4.3 Verification Runner and Solver Integration

Finally, the analysis reaches the heart of this section: the *verification runner*. It acts as an orchestrator for the SMT solver, which is more of a submodule here. The runner receives the previously materialized graph, along with the security invariants, and uses the elements inherent in the external solver to evaluate them. The solver modules are maintained in a separate directory, indicated to the runner via the *solver_dir* parameter. Its role here is to coordinate between an existing framework and an event-driven, runtime mechanism.

The actual verification is always preceded by an input validation phase, to verify that the graph directory exists and that all the required JSON files are present. This operation is performed by the `validate_graph_dir()` function, which compares the directory with the *REQUIRED_GRAPH_FILES* list and aborts the process if at least one file is missing. This avoids the risk of initializing the solver's Knowledge Base with an incomplete representation of the cluster.

After validation, `add_solver_to_path()` inserts both the watcher and solver module directories into the Python import paths. The runner can thus dynamically load `Z3KubernetesKB`, `parse_query`, and `QueryExecutor`, while maintaining their implementations separated. The **Knowledge Base** is accessed through `kb.py`, which exposes a stable import point for the `Z3KubernetesKB` class. Within a single run, the Knowledge Base is built only once:

```
kb = Z3KubernetesKB(graph_dir)
executor = QueryExecutor(kb)
```

Then the invariants are evaluated sequentially on the same model. This eliminates the need to reconstruct the entire representation for each element of the query set. The `run_verification()` loop extracts the identifier and DSL string from each rule, passes it to the parser, and delivers the result to the `QueryExecutor`.

An important observation is that the query field does not contain Z3 code. The string is first passed to `parse_query()`, which parses its syntax and constructs an internal object containing the search root, predicates, and any logical operators. The parser recognizes entities such as Subject, Role, and Endpoint, along with predicates such as Permission and Connection.

Next, `QueryExecutor` comes into play, distinguishing between predicate queries, 'get' operations, and 'list' queries. Formula construction is handled by `Z3QueryBuilder`, while the actual checks are performed by the solver associated with the Knowledge Base. For queries that search for resources or relationships, the output can be returned in text and tabular form, or as an empty set. The runner also records the time spent building and

executing the query; these values will then be included in the final report.

6.4.4 Verification Report and Violation Generation

Once the queries have been executed, the *verification runner* reshapes the outputs into a readable and persistent format and stores them in two separate files: `verification_report.json`, which includes the results of the entire execution, and `violations.json`, which simply lists the violations. To be able to trace the evaluated invariants, the two modules maintain the identifier, description, DSL query, execution status, raw solver output, processing times, and, if applicable, the error. The general structure of the **verification report** is as follows:

```
{
  "status": "completed",
  "generated_at": "2026-06-28T09:00:00+00:00",
  "graph_dir": "...",
  "query_file": "...",
  "total_time": 0.75,
  "results": [],
  "violations": []
}
```

Particularly important are the fields *graph_dir* and *query_file*, which refer to the inputs; *results*, which contains all records related to query evaluation; and *violations*, which contains records of violations found that will be reflected in `violations_json`.

A *results* field element can take this form:

```
{
  "query_id": "role_can_delete_protected_pod",
  "description": "Detect whether the Role can delete the protected Pod.",
  "query": "list (*) from [[Role(Role@security#dangerous-role)] has
   Permission([delete], Pod@security#protected-pod)]",
  "status": "ok",
  "raw_result": "Role@security#dangerous-role",
  "build_time": 0.1,
  "exec_time": 0.01,
  "error": null
}
```

query_id maintains the connection to the source invariant, *description* and *query* use the same values as those in the `verification_queries.json` file, *raw_result* contains

the solver's response, and *status* can be set to 'ok' if parsing and execution were successful (it may still generate a violation) or 'error' if parsing or execution resulted in a technical issue. *build_time* and *exec_time* contain the time required to generate the executable representation of the query and the time required to evaluate it, respectively. These two values are useful for demonstrating the level of realtime achieved by the system and for analysis aimed at future developments.

The runner's next operation is to apply the *violation_when* clause to the results. This allows the solver's output to be given an operational meaning. If the condition is true, the runner constructs a violation object, which is saved in `violations.json`.

```
{
  "status": "completed",
  "source": "verification_runner",
  "generated_at": "...",
  "graph_dir": "...",
  "total_violations": 1,
  "violations": [
    {
      "id": "vio-role_can_delete_protected_pod-0",
      "query_id": "role_can_delete_protected_pod",
      "category": "RBAC",
      "severity": "high",
      "message": "The Role grants delete permission on the protected Pod.",
      "evidence": "Role@security#dangerous-role",
      "status": "open"
    }
  ]
}
```

The violation structure also incorporates some metadata already encountered in security invariants, such as *category*, *severity*, and *message*. *Evidence* contains the element or elements that led to the query's failure, and the *status* set to 'open' indicates that it has not yet been resolved.

6.4.5 RBAC and Network Policy Query Examples

It's useful to consider two examples related to the domains analyzed in this thesis: RBAC and Network Policy. In both cases, the query is executed on the model built from the Digital Twin, and the result is then interpreted through the *violation_when* field.

The first example concerns the presence of a dangerous permission within a Role, so

the query is expressed as follows:

```
list (*) from [[Role(Role@security#dangerous-role)]
has Permission([delete], Pod@security#protected-pod)]
```

The root of the search is the *Role* identified by *Role@security#dangerous-role*. The *Permission* predicate instead checks whether, among the rules associated with that role, the *delete* verb appears applied to the *protected-pod* Pod in the *security* namespace. If the query returns at least one match and the invariant is configured with *violation_when: "non_empty"*, the result is considered a violation.

This formulation evaluates the Role definition itself, not the actual privileges of a subject. Deleting a RoleBinding, for example, can prevent a user or ServiceAccount from receiving that role, but does not remove the permission contained in its rules. To resolve such a violation, it would then be necessary to modify the Role, removing or reducing the grant authorizing the Pod deletion.

The second example case concerns verifying connectivity between workloads:

```
list (*) from [[Endpoint(Pod@app#backend)]
has Connection(Pod@client#client, 80, TCP, INGRESS)]
```

The key element here is *Endpoint(Pod@app#backend)*, which is the destination Pod. The Connection predicate checks whether the client Pod, located in the client namespace, can reach the backend on port 80, using TCP protocol, in the INGRESS direction.

The solver reconstructs the overall behavior by considering the rules applicable to the source and destination. For ingress communication, therefore, the policy protecting the receiving Pod comes into play; at the same time, if the source is isolated in egress, the rules authorizing outbound traffic must also be considered. When no policy applies to a given direction, that side is treated as permitted according to Kubernetes semantics.

Chapter 7

Remediation Engine

After observing the real cluster, reconstructing its state, evaluating it against invariants, and identifying any violations, the workflow reaches the remediation phase. When a property is not satisfied, the generated report collects all the significant information needed to reconstruct the cause of the problem. This data then becomes the input for the *Remediation Engine*.

The module initially performs an investigative function, aiming to identify possible changes that can restore the cluster to a safe state, in which all invariants are respected. For the purposes of this thesis, the *remediation engine* is exclusively responsible for correcting incorrect configurations related to RBAC and Network Policies.

When a possible corrective action is identified, it is not applied directly to the real environment but instead the Digital Twin is used to run a sort of simulation and evaluate the effects of the change within a safe and controlled environment. If the simulation produces a positive outcome, the fix can be proposed to the user to correct the real cluster.

The chapter then analyses in detail the operations carried out by the remediation module.

7.1 Objectives and Design Principles

As a first step, the module must analyze the results of the verification phase, contained in the `violations.json` file, and identify a concrete and verifiable fix. If you want to use a formal script to model this behavior, you can indicate the current state of the Digital Twin with G and the possible corrective action with R, then represent the candidate state as G' = R(G).

The strategy is successful if, after re-executing the invariants on the candidate state G',

the Digital Twin's state is safe again. It goes without saying that the proposed remediation must not introduce new violations.

Previous research on firewall policy management has shown how formal representations can support both anomaly analysis and the generation of corrected configurations, while preserving the legitimate behavior of rules not involved in the detected problem [62].

So, the most important criterion for this section is certainly the **minimal modification**: the proposed state, while satisfying the invariants, must remain as similar as possible to the previous state. This avoids excessively modifying the cluster configuration and compromising legitimate permissions associated with other resources. An intervention on a Role should only eliminate the verbs identified during the analysis, leaving the other rules unchanged. Similarly, in the case of a binding, the system removes the affected entity without changing the *roleRef* or deleting the entire resource.

Furthermore, when the precheck phase (verifying the candidate state) is initiated, the system creates a temporary copy of the graph, applies the change to it, and compares the results before and after the intervention. This avoids actions that could compromise the current representation. The precheck addresses the need to monitor the effects of the remediation on the Digital Twin, while the subsequent phase, the dry run, ensures that the change is supported by the Kubernetes API Server.

Assisted approaches to policy anomaly resolution preserve the operator's role in selecting among the proposed corrective actions, rather than leaving the entire decision process to an automatic mechanism [63]. In fact, a key aspect of this workflow phase is user involvement in the process. Once the precheck and dry-run are successfully completed, applying the fix to the actual configuration requires **manual confirmation**.

The general tendency is to adopt a conservative approach, in which the remediation is discarded if the target cannot be correctly identified, if the precheck produces errors, or if new violations appear.

7.2 Remediation Engine Architecture and Software Components

Analyzing the architecture in detail, the *remediation engine* is located between the *solver/verification runner* and the Kubernetes cluster. Its inputs are the violation report, security invariants, and materialized JSON files from the DT, with the aim of developing a countermeasure for the detected issue.

The implementation is separated into different components: `remediation_runner.py` is the module's entry point, while all the application logic is found in the remediation

package. All internal modules are used to reconstruct the context of the violation, select a **playbook**, prepare a strategy, create the candidate state, and initiate the precheck.

The three phases of the project architecture in this section are therefore analyzing the violation and selecting a candidate plan, simulating the strategy applied to the Digital Twin and re-executing the queries, and performing a dry run with the subsequent application of the fix to the real cluster.

This framework separates the abstract strategy from the concrete implementation. In fact, the playbook indicates the general plan (e.g. reduce privileges to a role), but it is the internal modules that carry out the action (e.g. identify the Role, the verbs to remove and the files on which to run the simulation).

7.2.1 Remediation Runner and Internal Modules

`remediation_runner.py` orchestrates the remediation process, providing various operations including catalog listing, candidate search, prechecking, and the actual application. The division of actions helps avoid a monolithic remediation scenario, which could modify the actual cluster too roughly, reconfiguring it to a violated state. Depending on the requested command, the necessary modules are initialized to return a clear and structured result, which can also be accessed from the dashboard.

An important first step is to reconstruct the problem that caused the invariant to fail. This is handled by `context.py`, which retrieves the violation in question in the violations JSON and extracts its *query_id* to identify the original invariant in `verification_quries.json`. The collected data helps construct the **ViolationContext** object.

Management of the remediation catalog is handled by the `catalog.py` file, which loads and validates the **templates** in `remediation_playbooks.json`. The templates are differentiated by category (RBAC and NetworkPolicy) and severity, as well as by the proposed strategy and the handlers to be used during the simulation and application phases.

The main structures exchanged between the modules are defined in `models.py`, including **RemediationTemplate**, **ViolationContext**, **RemediationOperationPlan**, **PrecheckResult**, and **ApplyResult**. The use of explicit models clarifies the information produced in each phase and limits the passage of unvalidated data between components.

To create the candidate state, `scenario.py` comes into play, where the `WhatIfScenario` class copies the Digital Twin to a temporary location and assigns the scenario an id. The **handlers** apply this id to the previous copy, taking into account that the RBAC and Network Policy operations are in two separate handlers (`rbac.py` and `network_policy.py`).

The output of the handlers will be an object called RemediationOperationPlan, which includes the action target, the action itself, and the proposed changes.

To verify the scenario and simulate its effects, `precheck.py` executes the invariants on the current state, then calls the candidate state handler and starts a second check on the modified copy. The results are condensed into the PrecheckResult object, which is used by the last module in this phase, `apply.py`. This reloads the persisted precheck (checking that it is still valid) and starts the server-side dry-run. When the Kubernetes API Server accepts the request, the fix is approved for application to the real cluster.

Component	**Main responsibility**
`remediation_runner.py`	Coordinates the remediation workflow
`context.py`	Correlates the violation with the corresponding invariant
`catalog.py`	Loads, validates, and selects remediation playbooks
`models.py`	Defines the shared data structures
`scenario.py`	Creates and manages the candidate Digital Twin state
`precheck.py`	Verifies and compares the current and candidate states
`handlers/`	Simulates the proposed configuration changes
`apply.py`	Performs the server-side dry-run and applies the change to the cluster

Table 7.1: Main software components of the Remediation Engine.

7.2.2 Runtime Files and Data Flow

An interesting thing to keep in mind is that the catalog does not contain pre-built Kubernetes changes; instead, the fix is processed at runtime based on the violation and the current state of the Digital Twin.

For the simulation, the data employed is the materialized graph files. For example, RBAC strategies use `roles.json`, `grant_edges.json`, `bind_edges.json`, and `subject.json`, while NetworkPolicy strategies focus more on `network_policies.json` and `resources.json`.

When the precheck starts, the Digital Twin directory is copied into a `dt-scenarios` folder. In this copy, the handler modifies only the necessary files. The *verification runner* is then executed using the scenario as the new `graph_dir`, producing a report and a set of violations related to the candidate state.

The resulting object, already mentioned in the previous subsection, is PrecheckResult, which contains the verification outcome, the selected playbook, the plan, and the results of two proposed scenarios. Once a precheck is completed, it is saved in a directory so it

can be retrieved in the next phase without having to rebuild the remediation again.

Then, upon application request, the backend resumes the plan associated with the precheck already executed and checks that there have been no changes to the Digital Twin. If the *watcher* has updated the graph after the simulation, the precheck must be considered obsolete. Once these checks are passed, a dry-run is initiated on the proposed change, and finally the application is launched via the Kubernetes Python client. The operation is recorded in the audit log with the resources involved and the result. The new configuration is then observed by the watcher and, thanks to the *graph updater*, materialized in the graph and subjected to reverification.

7.3 Violation Context and Remediation Candidate Selection

When `violations.json` is generated, the violations it contains and the information they convey are insufficient to convert them into a strategy for modifying the cluster configuration.

The crucial point here is to recreate the context of the problem that led to the violation and reconduct it to the invariant that was violated.

`Violation` → `ViolationContext` → `Candidate Playbooks`

By creating the ViolationContext from the violation, the runtime result is enriched with information contained in the original query definition. Then, using the context, the list of possible strategies can be narrowed down.

This step does not lead to any changes, but it suggests the direction to move in, which strategies to consider and which to discard based on the results of the verification phase.

7.3.1 Violation and Invariant Correlation

The context creation begins with the selected violation. The system uses its identifier to search for it in `violations.json` and, once found, retrieves crucial information, such as the associated query, category, severity, message, and any evidence collected during the verification.

When the initial query is found in `verification_queries.json`, merging it with the violation helps build the ViolationContext, which will include the *violation_id*, *query_id*, *category*, *severity*, *message*, *evidence*, the complete *query*, and the *violation_when* condition.

The evidence alone is not enough to identify the remediation target; it also requires analysis of the query and the Digital Twin materialized files. If any information is missing or incomplete (e.g., the query id), the system will not create an approximation of the remediation but will return an error.

Information	**Source**	**Purpose**
`violation_id`	`violations.json`	Identifies the detected violation
`query_id`	`violations.json`	Links the violation to the corresponding invariant
Category and severity	Violation report or query definition	Support remediation candidate filtering
Message and evidence	`violations.json`	Describe the concrete verification result
Complete query	`verification_queries.json`	Preserves the semantics of the security property
`violation_when`	`verification_queries.json`	Defines when the query result is considered a violation

Table 7.2: Information used to build the violation context.

7.3.2 Playbook Catalog and Candidate Filtering

Once the ViolationContext has been built, the system queries the catalog defined in `remediation_playbooks.json`, which contains the set of corrective actions supported by the framework and establishes their operational scope: changes are not freely generated but are derived from predefined strategies linked to known handlers.

Each catalog entry becomes a RemediationTemplate object, with an identifier, category, severity, title, description, general strategy, and references to the handlers used in the simulation and during application on the cluster. At this level, the template only specifies the type of intervention (e.g., reducing a role's privileges, restricting incoming traffic to a workload) without specifying which resource to modify or which fields to change.

Before use, `remediation_playbooks.json` is validated, verifying that the JSON structure is correct, that the required fields are present, that no identifiers are duplicated, and that the category and severity fall within the permitted values ('RBAC' and 'NetworkPolicy' for categories; 'low', 'medium', 'high', and 'critical' for severities). Information violations do not generate automatic actions.

The catalog contains sixteen playbooks: 2 categories $\times$ 4 severity levels $\times$ 2 alternatives $= 16$. The initial filter is then based on category and severity, and returns a limited set of candidates compatible with the detected issue. However, these may not be truly

applicable, as the resource may not exist in the Digital Twin, the target may not be precisely identifiable, or the query may not contain enough information for a safe modification.

Hence, the distinction between RemediationTemplate and RemediationOperationPlan is that the former is reusable for multiple violations of the same type, while the latter is specific to a single execution and contains the resources involved and the intervention parameters. For example, a template might propose a reduction of RBAC privileges; the operational plan should specify the Role involved, the namespace, the rule to be modified, and the verbs to be removed.

7.4 Remediation Strategies for RBAC and NetworkPolicy

As previously mentioned, the catalog currently includes 16 playbooks, eight of which are RBAC-related and eight of Network Policy. Each of these categories can have up to four severity levels, and two template variants are available for each category-severity combination. These currently represent basic strategies that are then tailored to individual violations by handlers. The architecture allows for the catalog to be expanded in the future to include new remediation scenarios, provided the template, simulation handler, and application handler are defined.

Category	Severity	Template ID	Remediation strategy
RBAC	Low	`rbac-low-remove-unnecessary-permission`	Remove unnecessary RBAC permission
RBAC	Low	`rbac-low-remove-unnecessary-subject`	Remove unnecessary subject assignment
RBAC	Medium	`rbac-medium-restrict-role-scope`	Restrict RBAC rule scope
RBAC	Medium	`rbac-medium-restrict-role-assignment`	Restrict RBAC role assignment
RBAC	High	`rbac-high-remove-dangerous-permissions`	Remove dangerous RBAC permissions
RBAC	High	`rbac-high-remove-privileged-binding`	Remove privileged RBAC binding
RBAC	Critical	`rbac-critical-replace-admin-with-least-privilege`	Replace administrative RBAC access with least privilege
RBAC	Critical	`rbac-critical-revoke-privileged-assignment`	Revoke critical privileged RBAC assignment
NetworkPolicy	Low	`network-low-restrict-port-protocol`	Restrict NetworkPolicy port and protocol
NetworkPolicy	Low	`network-low-narrow-peer-cidr`	Narrow NetworkPolicy peer or CIDR
NetworkPolicy	Medium	`network-medium-restrict-ingress`	Restrict ingress traffic
NetworkPolicy	Medium	`network-medium-restrict-egress`	Restrict egress traffic
NetworkPolicy	High	`network-high-block-cross-namespace`	Block cross-namespace traffic
NetworkPolicy	High	`network-high-add-workload-isolation`	Add workload isolation
NetworkPolicy	Critical	`network-critical-apply-default-deny`	Apply default deny
NetworkPolicy	Critical	`network-critical-quarantine-workload`	Quarantine workload

Table 7.3: Remediation playbook catalog.

7.4.1 RBAC Remediation Strategies

In the current model, RBAC strategies focus on two different elements: the permissions defined by a role and the role's binding to a subject. These two cases arise because most RBAC violations are due to excessive permissions in a Role/ClusterRole or incorrect assignment of the role to a subject that shouldn't have it.

For the first scenario, the `rbac.modify_role_permissions` handler was implemented, which modifies the `grant_edges.json` file when necessary. The handler is able to identify the role and target of interest via the ViolationContext and remove the verbs specified in the remediation. The mode field in the template structure determines the behavior of the fix. Some examples are:

- *remove_matched_verbs* removes verbs related to the violation;
- *remove_matching_grants* restricts the rules compatible with the query result.
- *remove_dangerous_verbs*, used for high severity, removes sensitive verbs such as create, update, patch, delete, bind, escalate, and impersonate.

- *least_privilege* eliminates dangerous grants in critical violations to reduce administrative privileges.

The second scenario uses the `rbac.modify_role_assignment` handler, which manages the `bind_edges.json` file. The goal here is to remove relationships linking roles to subjects who may not hold them, leaving the *roleRef*, other subjects linked to the binding, and assignments unrelated to the violation unchanged.

Below is an example of an RBAC remediation template:

```
{
  "id": "rbac-high-remove-dangerous-permissions",
  "category": "RBAC",
  "severity": "high",
  "title": "Remove dangerous RBAC permissions",
  "description": "Remove dangerous verbs from the involved role.",
  "strategy": "Delete dangerous verbs from matching grant edges.",
  "simulation_handler": "rbac.modify_role_permissions",
  "cluster_handler": "rbac.patch_role",
  "operation": {
    "mode": "remove_dangerous_verbs"
  },
  "warnings": []
}
```

It does not contain yet the role name and verbs, because these are data that are defined by the handlers at runtime, considering the template, the violation, the starting query and the materialized files of the DT.

7.4.2 NetworkPolicy Remediation Strategies

NetworkPolicy strategies affect the normalized representation of network rules contained in `network_policies.json`. Templates in this category use the `network_policy.restrict` handler, modifying the mode field to manage its behavior.

To construct the remediation, the handler extracts key elements from the query, such as the affected Pod, the communication direction (Ingress or Egress), a possible peer, and finally the port and protocol. It is important that the target is identified with sufficient precision: when the query refers to a generic endpoint, an attempt is made to complete it using the evidence collected during the breach. If this fails to identify a specific workload, the transformation is simply aborted.

Strategies associated with a low severity restrict individual aspects of communication.

This is the case with *narrow_port*, which affects the port and protocol, and *narrow_peer,* which restricts the set of allowed peers (or CIDR ranges, as appropriate). For medium-severity violations, *isolate_direction* was introduced, which provides isolation limited to the direction affected by the event.

For high-severity violations, the strategies must address more complex scenarios. *Restrict_matching_rules* removes compatible rules that are too permissive. Violations classified as critical have two dedicated strategies, *default_deny_both* and *quarantine*, both designed to act simultaneously on Ingress and Egress. The first simply introduces a default deny configuration for the workload, while the second goes beyond this, providing stronger bidirectional isolation, designed for cases where the workload must be temporarily disconnected from the rest of the cluster.

From an implementative point of view, the solver's model follows an order in which any overly permissive allow rules are removed first, and only then is the rule generated for the specific scenario added. This choice is not random, but takes into account the additive nature of NetworkPolicies in Kubernetes. Therefore, adding a new restrictive policy alone is not enough to cancel communications already permitted by other policies applicable to the same Pod. An example of a NetworkPolicy template:

```
{
  "id": "network-medium-restrict-ingress",
  "category": "NetworkPolicy",
  "severity": "medium",
  "title": "Restrict ingress traffic",
  "description": "Apply simulated ingress isolation to the involved workload
    .",
  "strategy": "Remove broad ingress rules and add generated isolation.",
  "simulation_handler": "network_policy.restrict",
  "cluster_handler": "network_policy.apply_manifest",
  "operation": {
    "mode": "isolate_direction",
    "expected_direction": "Ingress"
  },
  "warnings": [
    "Kubernetes NetworkPolicy is additive."
  ]
}
```

Like RBAC, the template only includes the generic strategy. When processing the specific violation, parameters such as the namespace, Pod, and selectors will be derived.

7.5 Digital Twin What-If Scenario and Persisted SMT Precheck

Finding the right strategy in the catalog and modeling it in the specific fix is only the first step to effectively resolving the violation. The remediation plan must be validated, checking its effects on the entire model. It should resolve the violation, but without heavily reconfiguring existing relationships, causing other invariants to be violated.

In this phase, the Digital Twin provides a risk-free simulation and what-if analysis environment [32]. Thanks to it, the current state of the model remains unchanged, while the handler associated with the selected playbook works on a copy of the graph, and the resulting configuration is the one on which the SMT verification is initiated.

To be considered valid, the fix result must be compared with the one of the initial state verification. The goal is to find a candidate state that, unlike the current one, resolves the violation and does not introduce new violations. If it meets these conditions, the result is stored in a persistent record (which also includes an expiration time and a fingerprint) to facilitate the next phase of the remediation module.

This section is divided into three operations: construction of the candidate state, comparison of the latter with the initial one and storage of the result for the next step.

7.5.1 Temporary Candidate Digital Twin

After the strategy has been built from the template, the RemediationOperationPlan is obtained, but to observe its effects on the model, it must be applied to a copy of the current state.

Referring to the formal relationship introduced at the beginning of this chapter, the candidate state is expressed as G' = R(G), where G is the initial, authoritative state of the DT and R is the transformation resulting from the fix.

The module in which the scenario management is implemented is `scenario.py`, where the WhatIfScenario class creates an id and a subdirectory within the larger directory containing the various scenarios generated as they occur. The materialized Digital Twin files will be copied into each of these subdirectories, which will represent a candidate state, to maintain (aside from strategy changes) the model organization expected by the verification module. If all the artifacts were not preserved, the solver would not be able to reconstruct the Knowledge Base it needs. The scenario uniquely identified by the id will be used as the new *graph_dir* when the next verification is launched.

Once all necessary files have been copied, the strategy simulation handler modifies

the target resources of the RemediationOperationPlan. Currently, the model (considering only RBAC and NetworkPolicy categories) will only modify `grant_edges.json`, `bind_edges.json`, and `network_policies.json`. The other JSON files are still carried over to the new scenario, since, for example, `roles.json`, `subjects.json`, or `resources.json` are useful for identifying the target, even though they are not modified.

An important detail regarding NetworkPolicies is that new rules resulting from changes by the candidate scenario are marked with the *dt_scenario_generated* field, to better distinguish them from the policies observed in the real cluster.

Creating a separate subdirectory for each scenario helps then to independently evaluate multiple playbooks starting from the same initial state. Currently, there is only one playbook pair for each type of violation detected, so there will be two subdirectories associated with a given current state. However, the catalog may be expanded over time by adding more specific templates that increasingly adhere to the minimal modification principle.

By default, the scenario is deleted once the precheck is completed, but the *–keep-scenario* option has been implemented to preserve the candidate state for manual comparisons of the initial files with the modified ones, for solver error analysis or for checking handler behavior during testing. It's useful to keep in mind that the retained candidate state in this case is different from the persisted precheck, which represents a validated version of the precheck, including comparison results and data useful for the actual application phase.

7.5.2 Baseline and Candidate Verification

The **precheck** consists of two distinct checks by the *verification runner*: the first is the ordinary one, in which the subject is the authoritative Digital Twin, which becomes the baseline; the second execution uses the candidate state directory as the *graph_dir*. The only real difference is this, because the set of security invariants, the solver, and the result interpretation logic remain the same in both cases.

A key concept is that a new check must be conducted for the baseline, different from the first one triggered by the watcher and which generated the violation in `violations.json`. This happens because in the time frame between the very first check and the remediation request, there may have been some new observation from the watcher that reconfigures the model and deletes the violation from the baseline. In this case, obviously, the precheck would not make sense.

Instead, in a standard case, the results of the two verification processes are collected in

objects called **VerificationSnapshots**, which are simply summary views of the verification outcome containing information such as detected violations and query results. It is precisely this formatting using uniform structures and criteria that allows the comparison of the baseline and candidate state results. For example, to identify the target violation, the id in `violations.json` is not used alone, because two subsequent checks may yield two different values for that field. So, to recognize the same security property even when the index assigned to the violation is regenerated, the system uses a combination of *query_id*, *category*, and *severity*.

After the comparison, the key requirement is for the violation to be present in the baseline but not in the candidate state. If, by pure chance, other violations disappear, it would be irrelevant; only the deletion of the one associated with the query on which the remediation is built can determine the success of the comparison. Furthermore, if the fix simulation introduces new violations (absent in the baseline), the precheck is considered failed.

$$V_{\text{new}} = V(G') \setminus V(G)$$

The system is configured so that the precheck is discarded when V_{new} contains high or critical violations. It is not inconceivable that a strategy may modify relationships to resolve its problem, introducing new (perhaps more serious) risks elsewhere in the model.

A curious case at this stage is when some candidates are identified as inapplicable even before the comparison. This occurs, for example, with a specific NetworkPolicy template: it is designed for violations involving Ingress traffic but initiating a precheck with this remediation for a violation involving Egress traffic returns the `REMEDIATION_NOT_APPLICABLE` code (different from a technical error by the solver or an ineffective remediation).

The analysis concludes with the creation of PrecheckResult, an object that contains the general outcome of the comparison, the selected playbook, the operational plan resulting from the handler, two snapshots, a notification of the disappearance of the violation of interest, errors, and any new violations (high or critical).

7.5.3 Persisted Precheck and Protection Against Stale Decisions

The PrecheckResult is not used yet for application to the real cluster but is saved server-side in a directory containing all remediation runs, where each precheck has its own JSON file in the `precheck-<id>.json` form. Before switching to the dry-run, the system must verify that the precheck is valid and not expired, consistent with the DT, and not consumed.

The JSON record has a lifetime of 900 seconds (i.e. 15 minutes), configurable with the *–precheck-ttl-seconds* option. Once this interval expires, the `PRECHECK_EXPIRED` code will be returned, and the backend will request a new simulation. This is useful for cases where the cluster changes configuration after the output of a precheck has been recorded, making the previous state unreliable.

Another technique to avoid these situations is to associate the precheck with a SHA-256 fingerprint from the authoritative DT, calculated on the model's materialized files that can influence remediation (i.e., `resources.json`, `network_policies.json`, `roles.json`, `grant_edges.json`, `bind_edges.json`, and `subjects.json`). Upon apply, the fingerprint is recalculated on the current state and then compared with the saved state:

$$H(G_{\text{precheck}}) = H(G_{\text{apply}})$$

If the two hashes match, then the fix still refers to the same state on which the precheck was constructed and verified. Otherwise, the precheck is considered obsolete and the `PRECHECK_STALE` code is returned. It is important for the fingerprint check to be based on all files that materialize the DT, not just those modified by the remediation (e.g., just `grant_edges.json`), because even seemingly insignificant changes could alter the fix result.

The final piece of information regarding prechecks concerns their use: since a precheck already consumed cannot be reused, the `PRECHECK_ALREADY_APPLIED` code serves as a warning if the rule is violated.

7.6 Safe Application to the Kubernetes Cluster

Once the precheck phase is completed, we move on to the final steps of the *remediation engine* module. This step must be separated from the precheck because the objectives are different: previously, it was a semantic check on the model, while now it's a matter of determining whether the Kubernetes API Server can accept the changes.

Applying to the live cluster will require manual user confirmation to retrieve the persisted plan from the backend and deploy it in the live environment, checking its validity through a **Kubernetes dry-run**.

If successful, the fix will be applied to the cluster, and the remediation will be considered effective when the target violation disappears with the very last post-verification operation.

7.6.1 User Confirmation and Kubernetes Server-Side Dry-Run

The entire project is based on the Digital Twin, which offers a secure simulation environment where to conduct a formal analysis of the cluster without actually affecting it. For this reason, at this point in the workflow, explicit confirmation is required from the user before applying the remediation to the real cluster. The user must be aware that this confirmation will cause changes not only to the virtual environment, but also to actual resources, bindings, roles, verb lists, or network policies. In the future, it may also be possible to implement a sort of login for confirmation, so that only users with the appropriate credentials are enabled to make changes to the cluster.

Anyway, authorization is transmitted from the dashboard to the backend using the persisted precheck id. The server will retrieve the validated RemediationOperationPlan, verifying that the precheck meets the requirements of the previous paragraph (e.g., not expired and unused).

An important step in starting the real operation is initializing the Kubernetes Python client, whose configuration is first looked up in the local *kubeconfig* or, otherwise, in the in-cluster environment. If neither method is valid, access to the Kubernetes API Server will be denied and the `NO_KUBERNETES_CONFIG` error will be returned.

Next, the patch or manifest related to the remediation is subjected to a server-side dry-run, executed by the Kubernetes API Server. This phase is crucial for determining if the plan can be satisfied in the real world, not just in the digital one; it helps to uncover any errors related to the resource's schema, invalid fields, permissions, admission controllers, or conflicts with the live version of the object.

It's worth noting that none of the various steps can independently determine the success of the remediation. In fact, the precheck alone points out that the fix works on the digital model but doesn't mean that the real cluster will accept the request; or the dry-run verifies that the change can be applied to the real environment but does not guarantee the elimination of the violation.

Only the sequential completion of all steps determines the success of the remediation phase. If the dry-run fails, the backend aborts the process and does not send the validated plan.

7.6.2 RBAC Apply Handlers

Once the dry run has passed, remediation can be applied to the Kubernetes cluster. For the RBAC category, two types of handlers have been implemented, similar to Simulation handlers, but this time they are of the Apply type. Here too, the distinction follows the two

types of interventions contemplated for RBAC violations: modifying the verbs associated with a role or revising the subjects list of the RoleBinding or ClusterRoleBinding. So, the Apply handlers for RBAC are:

- `rbac.patch_role`, used in cases where the rules of a Role or ClusterRole caused the violation. An important task is to retrieve the live version of the object via the Kubernetes API Server to avoid mistakenly operating on the copy used in the simulated scenario. A trivial example is a Role that grants a list of permissions `[get, list, delete]` on Pods, and the violation concerns the ability to delete. The patch will simply remove delete and keep the other verbs, while everything else (API groups, resources, resourceNames) will be ignored to comply with the minimal change principle. The fix also relies on the *resourceVersion* of the live object, which allows the API server to determine if there have been changes in the real cluster during the patch construction. In this case, the handler would avoid overwriting an updated version of the object (for example, by other users or the admin). Considering instead rules with wildcards (e.g., `verbs:["*"]`), the automatic verb reduction does not have a unique meaning. Therefore, risking creating overly broad patches, it was preferred to block operations when they are not sufficiently defined.

- `rbac.patch_binding`, deals with subjects illegitimately bound to a role via RoleBinding or ClusterRoleBinding. For this handler a reference value is also considered, in this case *roleRef*, to understand if the version of the relationship under examination in the live cluster corresponds to the one marked in the fix and to remove only that assignment, while keeping all the others.

7.6.3 NetworkPolicy Apply Handler

Alternatively, Network Policy remediations are managed during application by the `network_policy.apply_manifest` handler. The fields affected by the new manifest are the namespace, the selector of the involved Pods, and the traffic direction with the ingress/egress sections. These are important when implementing remediations aimed at isolating a workload. The proposed example shows how, through an empty list in the Ingress section, a remediation blocks incoming communications to a Pod (here called the backend):

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: dt-remediation-backend-ingress-<hash>
  namespace: app
  labels:
    app.kubernetes.io/managed-by: digital-twin-remediation
```

```
spec:
  podSelector:
    matchLabels:
      app: backend
  policyTypes:
    - Ingress
  ingress: []
```

The most interesting field to analyze is undoubtedly the resource name, which is constructed deterministically, considering the data present in the fix and respecting the DNS-1123 format. Furthermore, it is no longer than 63 characters and includes a hash to permanently preserve the policy's identity regarding strategy and target. Labels and annotations are then used to differentiate it from policies derived from users or admins.

A critical step is converting the DT's normalized selector into a *podSelector*, because the apply is then executed only if the target is unambiguous. Anything containing empty selectors, wildcards, or sets with multiple values is rejected, as it could unintentionally extend the number of pods considered. The `UNSAFE_NETWORK_POLICY_SELECTOR` code is used to alert the user to these cases.

Another special case is when a policy already has the same deterministic name. If it is marked as managed by the *remediation engine*, the live version is read, the *resourceVersion* is preserved, and the new manifest is replaced, updating the patch without duplicating the policy. If, instead, the policy with the same name is not managed by the system, it is left untouched because the handler cannot assume that a user's resource can be manipulated at will. In this scenario, `NETWORK_POLICY_NAME_CONFLICT` is returned.

The Network Policy application must then be careful, with precise target, selector, and object properties, to prevent fixes for specific communications from extending beyond the boundaries verified in the precheck.

7.6.4 Watcher Synchronization and Post-Remediation Verification

The remediation phase is complete when the apply events are detected by the *watcher*'s normal observation behavior. Typically, fixes related to updating a role's verb list or a binding's subjects generate a `MODIFIED` observable event; similarly, events generated by Network Policy patches are `ADDED` (because this type of remediation requires adding policies that further isolate communications), with some `MODIFIED` events if the new policy replaces an already managed one.

The object detected by the *watcher* is converted through the usual workflow into a

NormalizedK8sEvent, from which the *graph updater* extracts the canonical key to insert or replace the element in the memory store. The model is then rematerialized, and it is now that the dt-output files (the json files that materialize the Digital Twin) are essentially updated. Typically, all events triggered by remediations are considered relevant and therefore trigger the verification runner for a new verification phase (which in this case is better called **post-verification**).

A technique used to avoid interpreting output from before remediation as post-verification is to save the creation timestamp of the `violations.json` file and, after modifying it, wait for another file with a later timestamp (the wait is set to sixty seconds with a check every two seconds).

In the report produced by post-verification, the violation of interest will be searched using the *query_id*, *category*, and *severity* combination, as explained in the previous chapter. If the search yields no matches, the outcome will be 'resolved'; otherwise, it will be 'still_detected'. 'verification_timeout', 'verification_error', and 'pending' are also considered, for cases where the system was unable to complete post-verification with a valid outcome within the expected wait interval.

7.7 Auditability and Current Boundaries

The system requires each remediation apply to be logged in a file called `remediation_audit.jsonl`, so that a separate entry for each operation performed can be maintained in chronological order. The record includes the precheck and apply ids, as well as the playbook used, the violated query, the resources involved, and the results of the dry-run and cluster modification. There is also the possibility to include the fix and a representation of the resource's original state, providing not just a simple log but a record that links the simulation, authorization, and actual operation.

Although the project aims for a high level of automation, there are still some limitations; first and foremost, the fact that live modifications only consider Role, ClusterRole, RoleBinding, ClusterRoleBinding, and NetworkPolicy. But going into detail, as already noted, some RBAC changes that include wildcards aren't handled when the permission to be removed can't be precisely defined, or some NetworkPolicy strategies don't initiate the apply if the selector can't be converted without increasing the number of affected Pods.

Transient states represent a further limit that may arise while applying a change. In fact, security policies reconfigurations are not always instantaneous and the transition from initial to final state can lead to unknown behavior, absent in both stable states [66]. The designed workflow verifies the candidate scenario before the apply and the post-synchronization state observed by the watcher, but it does not take into account

every transient configuration through the whole update.

Another consideration is that there's no automatic rollback if post-verification leaves the violation active. This comes from the fact that restoring the previous configuration is tricky in a Kubernetes cluster, where legitimate additions, modifications, or deletions unrelated to the fix may have occurred in the meantime.

Chapter 8

Proof of Concept

After viewing the architecture and implementation of the various modules, with particular attention to the verification and remediation ones, we can now analyze the prototype's **Proof of Concept** in a controlled Kubernetes environment, highlighting the different sections that make up the UI and what they allow us to examine.

We will have the opportunity to verify that all the modules previously presented integrate seamlessly to receive the expected inputs, process them consistently, and deliver outputs in the desired format. This allows us to validate the flow between the real environment, the Digital Twin in the virtual environment, and security functions.

The tests presented introduce deliberate changes to the cluster, ensuring they are detected by the *watcher* and the entire resulting pipeline is activated, gradually displaying the data in the dashboard, which consumes the information produced at runtime without creating a separate model.

The purpose of this chapter is functional validation rather than an extensive performance benchmark: we will focus on the correct behavior of the prototype and the consistency of the various phases.

8.1 Validation Objectives and Experimental Setup

The Kubernetes environment where the validation phase was conducted has been configured locally using **Docker Desktop**, an application that allows you to generate and modify clusters for experimental activities. An important feature for this thesis is the ability to introduce controlled configuration changes by simulating a real environment that can be queried via the Kubernetes API Server, without necessarily involving real production infrastructure.

When the prototype is executed, all components related to graph updates, particularly the *watcher*, remain connected to the active cluster. The tests have been performed in live mode and not on previously enriched static datasets. As explained in detail in the implementation chapters, the observed data is normalized and passed to the update logic, which, in addition to storing the last observed state of the resources in memory, materializes the Digital Twin via the JSON files in the dt-output directory. This will be useful in subsequent phases and in the dashboard backend, to demonstrate how the data displayed in the UI effectively reflects the observed behavior of the cluster.

In all tests, the external SMT solver was configured to perform verification automatically. When a change affects properties relevant to the security model, the new graph state can be used to reevaluate queries. The system also distinguishes semantically significant updates from normal operational variations, avoiding unnecessary activation of the verification process in the presence of changes that do not impact the information considered by the model. To allow manual interaction with the prototype, a web dashboard has been built. The **React frontend** receives data through the local API in the `dt-api.mjs` file, avoiding drawing directly from the materialized Digital Twin files. It exposes data relating to the graph, invariants, events, checks, and violations, ensuring a constantly updated view of the cluster throughout its evolution.

Whenever a change affects properties under verification, the result must be consistent with the new cluster structure and accessible from the interface. If corrective action is required, the prototype distinguishes between the candidate configuration and the actual cluster application, thus avoiding alterations to the cluster's authoritative state during simple DT analyses.

8.1.1 Initial Cluster State and Digital Twin Visualization

The Proof of Concept begins by observing the initial state of the local cluster proposed by Docker Desktop, which contains both system and application resources. Starting the *watcher*, and then `graph_updater.py`, allows for an initial acquisition of the cluster configuration and the construction of its representation in the dt-output directory with the materialized files: `resources.json`, `subjects.json`, `roles.json`, `bind_edges.json`, `grant_edges.json`, `reference_edges.json`, `label_resources.json`, and `network_policies.json`. The appearance of these files confirms that the prototype has been bootstrapped successfully, and within them, the various resources will be uniquely named using the convention explained in previous chapters: *Kind@namespace#name*.

By launching the dashboard from the terminal and navigating to the first section, called **Overview**, you can access the cluster baseline view: both namespaced and cluster-scoped resources are present. The presence of system and application namespaces highlights the

fact that the Digital Twin not only describes the prototype's test workloads but also the Kubernetes context. Focusing on application namespaces, the Overview has been structured to highlight the workload hierarchy, as in the case of Pods, which are traced back to their respective Deployment, again traced back to the Namespace. The hierarchies allow us to verify that the prototype has reconstructed the ownerReferences and containment relationships of the original real cluster (Figure 8.1).

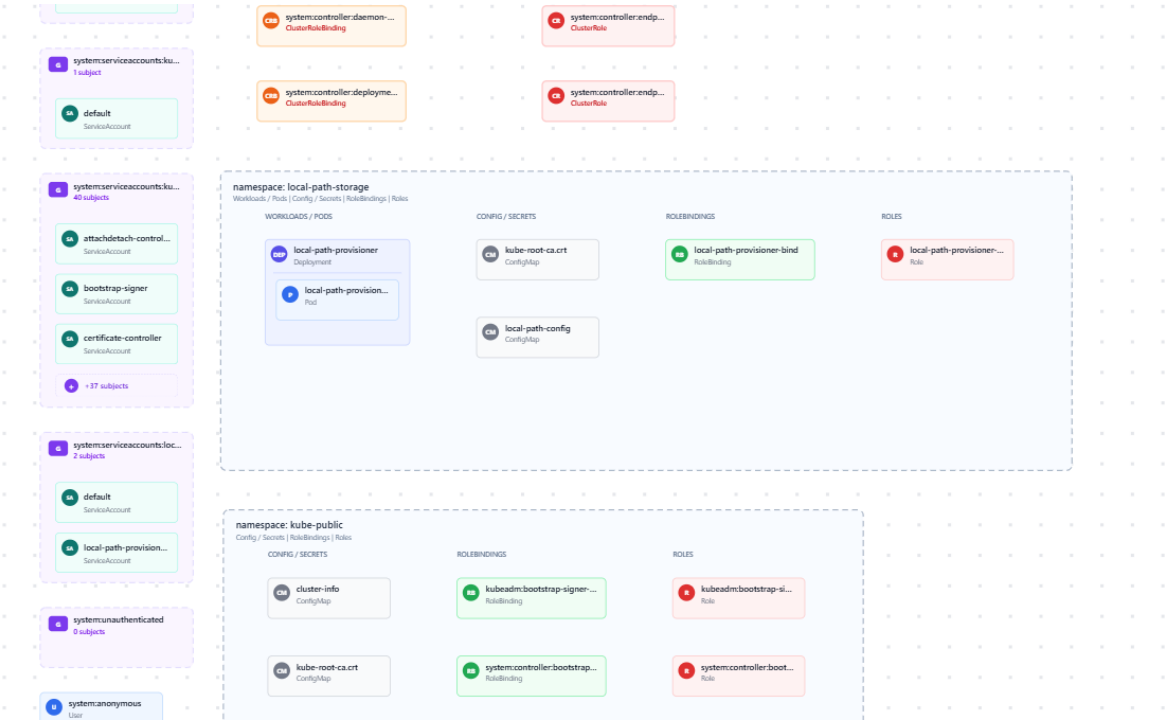


Figure 8.1: Initial cluster state displayed in the Overview section of the dashboard.

Remaining in the Overview section, it is possible to individually select a resource by clicking on it to access a panel containing the specifics of that object. This feature is very useful for enriching the information deducible from the cluster view alone and for concretely verifying the correspondence between the real cluster, the materialized representation and the user interface (Figure 8.2).

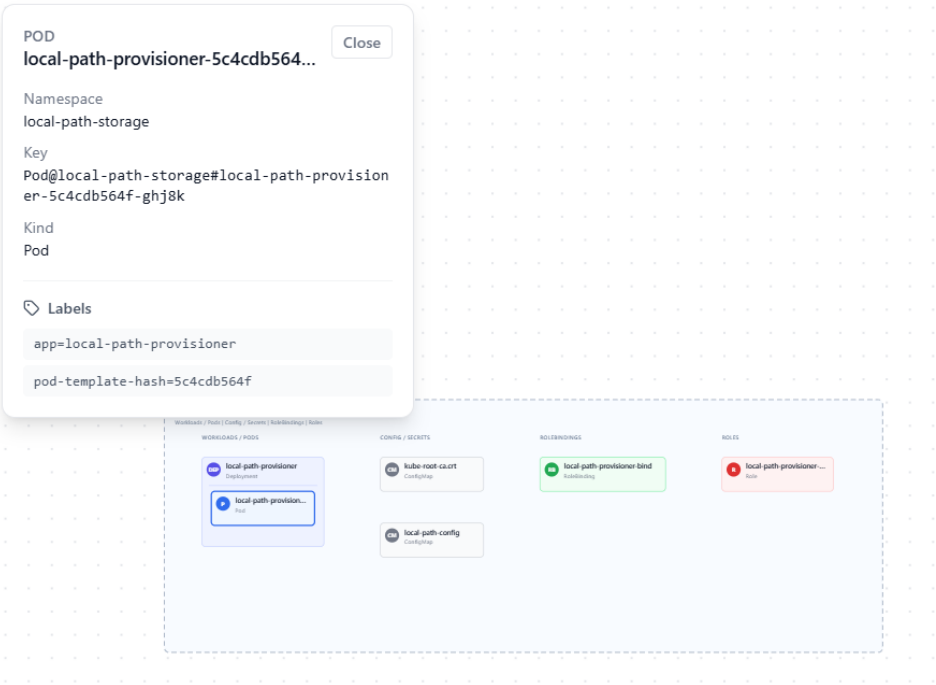


Figure 8.2: Resource detail panel opened from the Overview graph.

This view also includes buttons to activate the security mechanism layers, namely RBAC and NetworkPolicy. With the RBAC overlay, the cluster baseline can be read considering privileges: the connections between subjects, bindings, Role/ClusterRole, and grant targets are activated. For greater readability, the RBAC relations (links between subjects, bindings, and roles) have been distinguished in green from the Role->target permissions connections in red. Here, too, it can be seen that various pieces of information stored in separate files in the Digital Twin are merged into a single view, allowing the user to follow the logical path from an identity to a permission. Selecting the relevant elements and connections also opens detail panels that provide information on the binding, the associated role, the affected subjects, and the actual grants, offering a more direct understanding of the relationships already materialized in the Digital Twin. Similarly, you can activate the NetworkPolicy overlay, this time displaying information about the workloads' ingress/egress protections (with badges indicating `ALLOWED`, `RESTRICTED` or `ISOLATED`) and the network relationships resulting from the policies. Once more, selecting a badge or network relationship brings up a panel detailing the policies involved, the applied selectors, the peers, the direction, the protocol, and the ports considered (Figures 8.3 and 8.4).

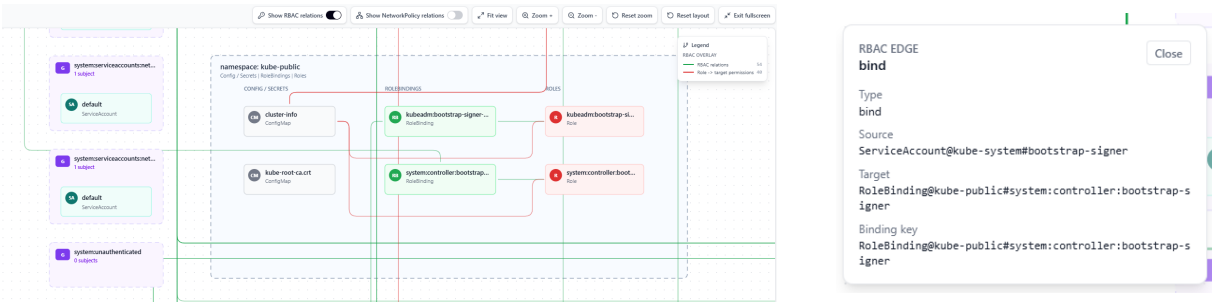


Figure 8.3: RBAC overlay in the Overview section and detail panel for a selected RBAC relation.

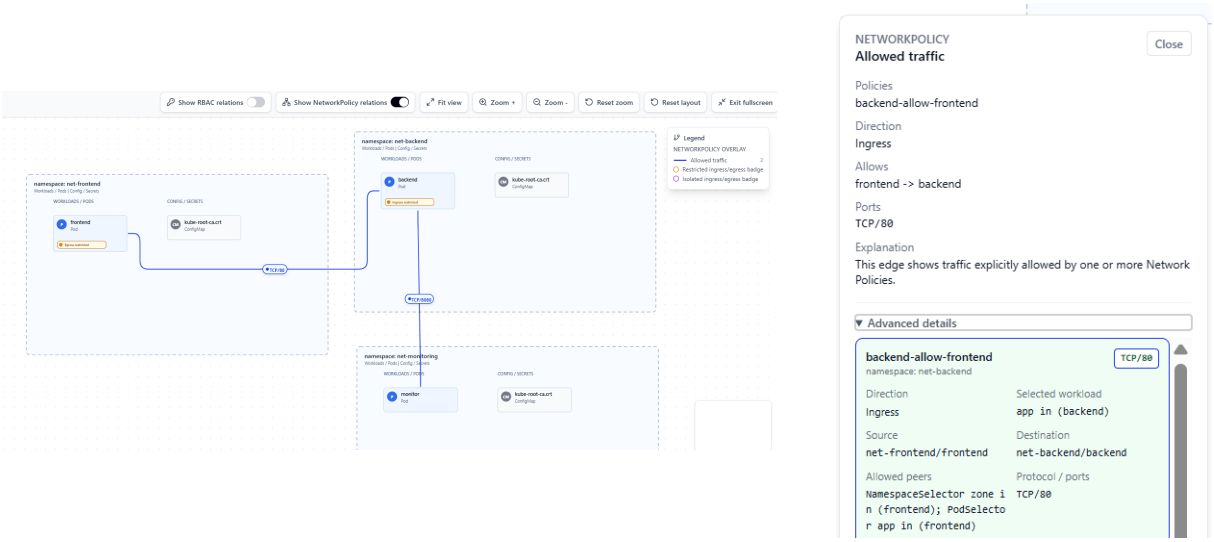


Figure 8.4: NetworkPolicy overlay in the Overview section and detail panel for a selected network relation.

At this point, we have defined the cluster baseline, which will be the starting point for the changes introduced by the Proof of Concept.

8.1.2 Runtime Synchronization and Event Propagation

The next step is to analyze how the system responds to the introduction of a significant change. We'll begin by evaluating the effects of a simple change, with no specific impact on RBAC or NetworkPolicy (we'll discuss these in the following subsections). The proposed example involves introducing a new Deployment in the default namespace using a simple kubectl command launched from the terminal:

```
kubectl create deployment sync-demo --image=nginx -n default
```

This will produce a new Deployment object in the Kubernetes API Server. Since it's included in the list of resources monitored by the *watcher*, its creation will be intercepted

by the relevant watch stream and converted into a normalized internal event, including information such as the event type (in this case, `ADDED`), the object's kind, the namespace, the name, and the canonical key (which, according to the construction method, will be *Deployment@default#sync-demo*).

Once normalized, the event is recorded in `cluster_events.jsonl` and stored in JSON format for up to 3,000 events. To ensure the *watcher* has correctly captured the event, simply find the relevant entry in this file, which in our case will take the following form:

```
{
"event_type": "ADDED",
"kind": "Deployment",
"name": "sync-demo",
"namespace": "default",
"resource_key": "Deployment@default#sync-demo",
"resource_version": "...",
"observed_at": "..."
}
```

For convenience, the fields just shown are not all those that make up the JSON record: missing fields include *api_version*, *label*, and *owner_references*.

The received event will also modify the runtime state maintained by `graph_updater.py`, which uses an indexed list with *resource_keys* that stores the latest version of each object. In the case of this `ADDED` event, a new entry will be created in this map, and the parser and serializer will then reproduce the updated ClusterGraph and materialize the files in the dt-output directory. By consulting the dashboard, you can check out the event logging in the **Events** section. Here, you can view the last 100 events in chronological order and filter them both by event type (`ADDED`, `MODIFIED`, `DELETED`) and resource type (Figure 8.5).

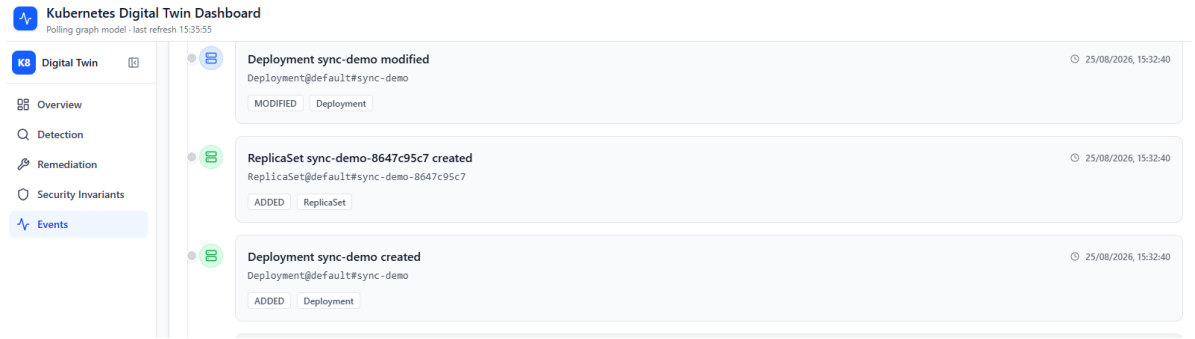


Figure 8.5: ADDED Deployment event displayed in the Events section.

Clicking on the event of interest, it is possible to view its details (Figure 8.6).

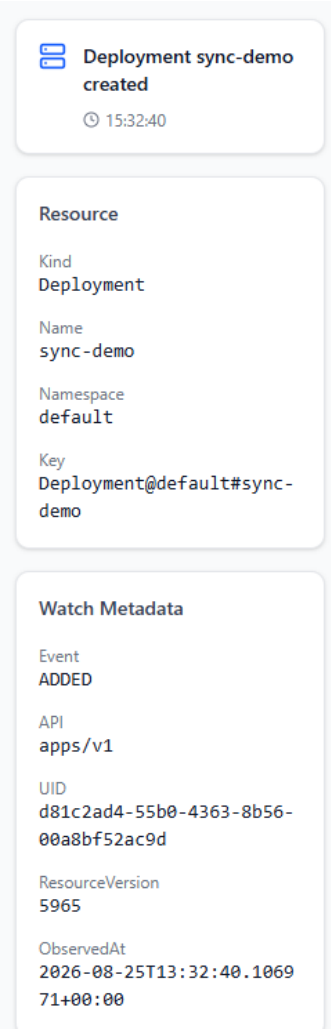


Figure 8.6: Detail panel for the sync-demo Deployment event.

At the same time, the subsequent poll will also update the Overview section, showing the new Deployment in the default namespace.

To verify another type of event and complete this simple test, simply delete the object with another terminal command:

```
kubectl delete deployment sync-demo -n default
```

Now the *watcher* will detect a `DELETED` event, and the graph updater will delete the resource from the internal state, resulting in a new Digital Twin materialization that will no longer contain the Deployment, as well as the Overview. The Events section and `cluster_events.jsonl` will contain a new `DELETED` entry (Figure 8.7).

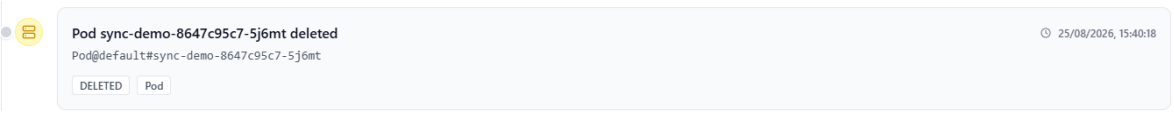


Figure 8.7: DELETED event generated after removing the sync-demo Deployment.

8.1.3 RBAC Security Scenario

A more significant test for the goals of this thesis has been conducted on one of the two security mechanisms, RBAC, in which the modification alters the representation of the Digital Twin and also violates a verified security invariant. There will be a namespaced Role called *dangerous-role* (with canonical key Role@security#dangerous-role) to which a delete permission is added, enabling a prohibited operation on a protected Pod (with canonical key *Pod@security#protected-pod*) in the same namespace. The original version of the Role is as follows:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: dangerous-role
  namespace: security
rules:
  - apiGroups: [""]
  resources: ["pods"]
  verbs:
    - get
    - list
```

The property that will be violated verifies that the Role does not have delete permissions on the Pod and is contained in `verification_queries.json` with this form:

```
{
"id": "role_can_delete_protected_pod",
"description": "Detect whether the Role can delete the protected Pod.", "
    query": "list (*) from [[Role(Role@security#dangerous-role)] has
    Permission([delete], Pod@security#protected-pod)]",
"violation_when": "non_empty",
"severity": "high",
"category": "RBAC",
"message": "The Role grants delete permission on the protected Pod."
}
```

Before the role change, the *violation_when* condition does not yet detect any elements that would trigger the violation, so the invariant will be present in the property list of the **Security Invariants** section of the UI with the status `Passing` (Figure 8.8).

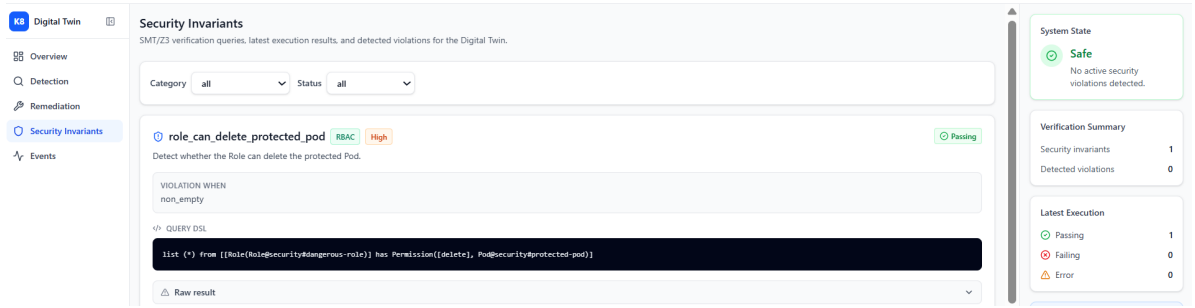


Figure 8.8: RBAC security invariant in Passing state before the Role modification.

The modification operation is performed using a JSON Patch command from the terminal, so as not to modify other fields of the object:

```
kubectl patch role dangerous-role `
-n security `
--type='json' `
-p='[{"op": "add", "path": "/rules/0/verbs/-", "value": "delete"}]'
```

This causes the role's verbs field to become:

```
verbs:
  - get
  - list
  - delete
```

The RBAC resource watch stream intercepts the change, which is normalized and labeled as `MODIFIED`. As with the previous generic case, the event is added to `cluster_events.jsonl`, becomes observable in the Events section, and will contain fields such as the event type and resource type (here, Role), the namespace, the name, and the canonical key (Figure 8.9).

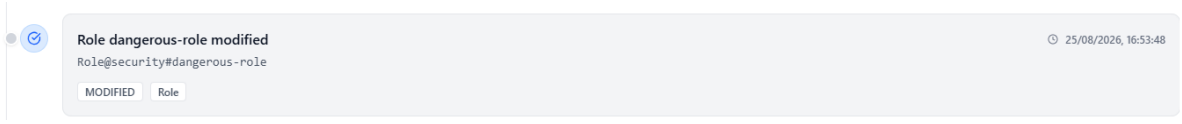


Figure 8.9: MODIFIED event generated by the dangerous-role update.

There will also be updates to the graph's materialized files, specifically `roles.json` and `grant_edges.json` (which contains the authorization relationships generated by the updated rule), and to the runtime store managed by `graph_updater.py`, where the previous version of the object is replaced by the newly received one, which materializes the current state of the DT.

Now, this test allows us to activate operations that weren't triggered with the previous one: changing a role's verbs is considered semantically relevant and so leads to a new

check of the security properties. After the three-second debounce (to ensure no further changes are arriving in the cluster and then verify everything together), the v*erification runner* is invoked, which uses the updated Digital Twin and the queries to initialize the knowledge base needed by the external solver.

The queries are then re-executed, and the one shown previously (`role_can_delete_protected_pod`) will find a match in the *dangerous-role* role, because it now contains the delete verb on the Pods in the namespace. The most relevant result produced in this phase is the record stored in the violation report in `violations.json`:

```
{
"id": " vio-role_can_delete_protected_pod-0 ",
"query_id": "role_can_delete_protected_pod",
"category": "RBAC",
"severity": "high",
"message": "The Role grants delete permission on the protected Pod.", "
    evidence": "...",
"status": "open"
}
```

At this point, the invariant entry can be consulted again in the Security Invariants section of the dashboard, but this time with the `Failing` status. In the **Detection** section, the violation will be visible, labeled with category 'RBAC' and severity 'high'. It will also contain other information derived from the `violations.json` entry, including the message and the evidence returned by the verification phase (Figures 8.10 and 8.11).

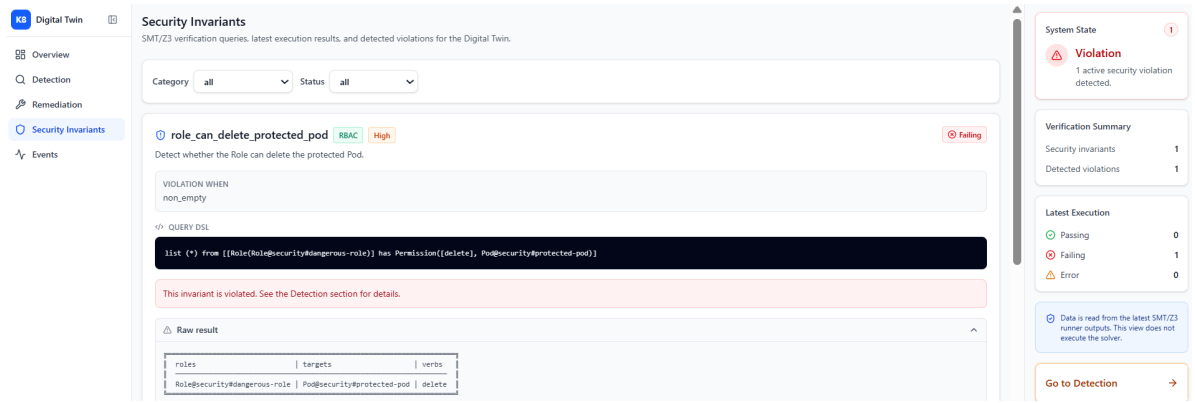


Figure 8.10: RBAC security invariant in Failing state after adding the delete verb.

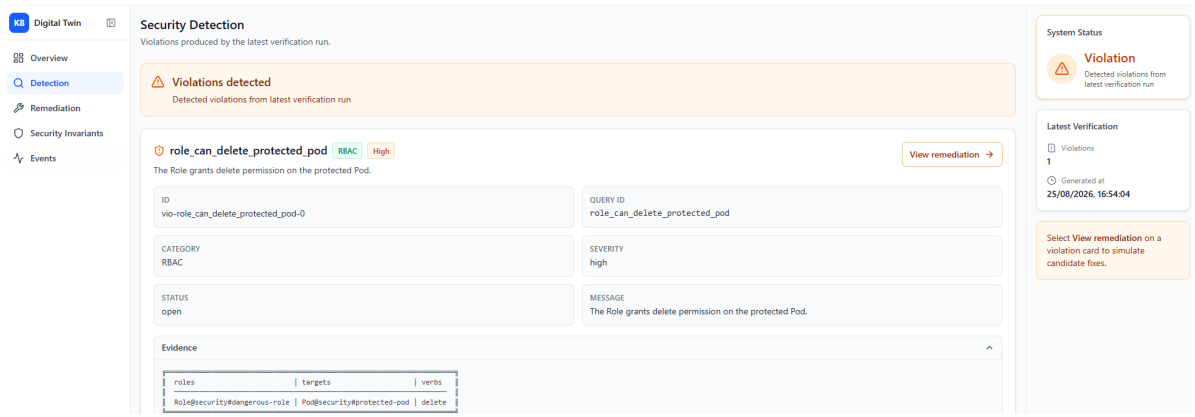


Figure 8.11: RBAC violation shown in the Detection section.

From the Detection page, you can then switch to the **Remediation** section via a button, which becomes visible when the violations section is no longer empty. The backend then retrieves the current violation and associates it with the query that generated it. The catalog is filtered based on category and severity, and for the given case, an intervention is proposed on the Role. In this case, the chosen strategy aims to remove the delete verb from *dangerous-role*, using the `rbac.patch_role` handler.

Starting from the authoritative graph (G), the *remediation engine* generates a copy of it in the dt-scenarios directory and then applies the playbook's remediation to it. The resulting candidate (G') will be a graph in which the *delete* verb is no longer present for the *dangerous-role* role.

The precheck phase ensures that the target violation is present in the baseline state and absent in the candidate state, and that no new violations have been introduced. The positive result is saved server-side in `precheck-<id>.json` in the dt-remediation-runs directory (Figure 8.12).

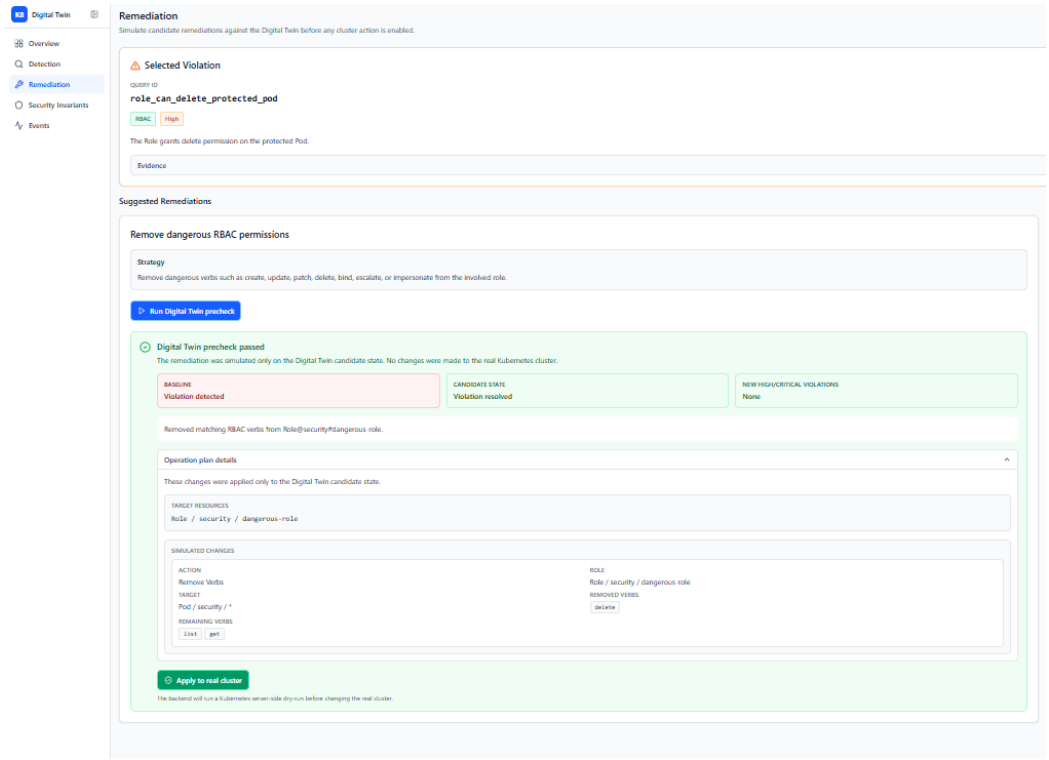


Figure 8.12: Digital Twin precheck for the selected RBAC remediation candidate.

At this point, the interface enables the button to apply the fix to the real cluster, requiring manual confirmation from the user. The backend, before any modification, will check that the precheck has not expired, has not already been used, and that it refers to the same state as the DT (this check is done by recalculating the fingerprint on the materialized graph files).

Modifying the Role verbs generates an additional `MODIFIED` event, observed by the *watcher* and triggering another verification process (post-verification). After remediation, the query will no longer fail, and the target violation will not appear anymore in the violation report. Successful application of the fix is visible in several spots of the interface: the Events section displays a new `MODIFIED` event related to the Role, the violation disappears from Detection, and the property is once again labeled as `Passing` in the Security Invariants section (Figures 8.13 and 8.14).

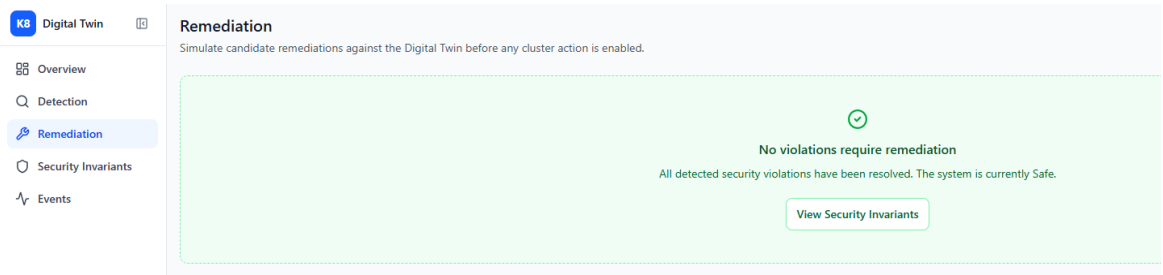


Figure 8.13: Remediation view after resolution of the active RBAC violation.

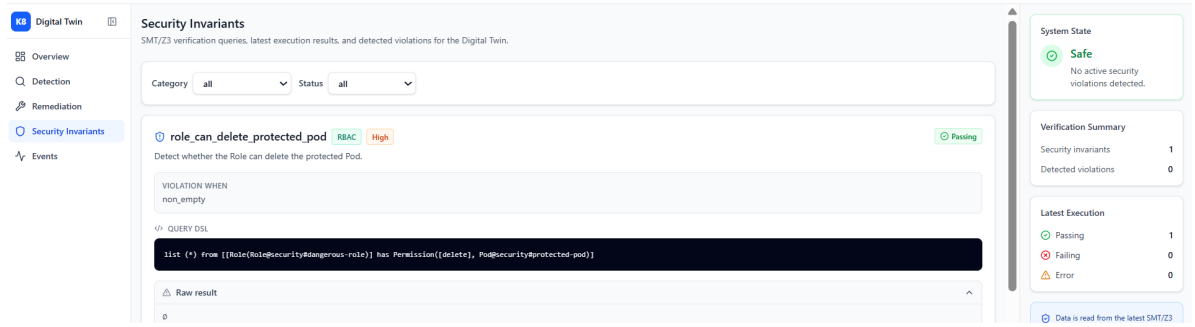


Figure 8.14: RBAC security invariant returned to Passing after post-verification.

In case of tests like this one where violations are generated, the overall system status can be consulted in the Overview, Detection, and Security Invariants sections, where it is marked as `Safe` or `Violation` (they both can be seen in previous images).

8.1.4 NetworkPolicy Security Scenario

The second security mechanism considered in this work is Network Policies, and for this reason, it's worth conducting a test on it as well. Here, reachability between two workloads from different namespaces is checked: we'll have a Pod called *backend* in the *app* namespace and a *client* Pod in the *client* namespace, and we'll evaluate the latter's ability to reach the former through TCP traffic on port 80. Both workloads will have distinct labels, useful for Digital Twin and for Kubernetes itself, to identify the endpoints involved in the policies. For the *backend*, we'll have the following configuration:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: backend
  namespace: app
spec:
  selector:
    matchLabels:
      app: backend
  template:
    metadata:
      labels:
        app: backend
    spec:
      containers:
        - name: backend
        image: nginx
```

client will have a similar form, but the label will be *app:client*, and in general, the canonical keys for the two endpoints will be *Pod@app#backend* and *Pod@client#client*. The invariant considered for this test is:

```
{
"id": "cross_namespace_ingress_backend_80",
"description": "Detect whether the external client can reach the backend
    over TCP/80.",
"query": "list (*) from [[Endpoint(Pod@app#backend)] has Connection(
    Pod@client#client, 80, TCP, INGRESS)]",
"violation_when": "non_empty",
"severity": "medium",
"category": "NetworkPolicy",
"message": "The backend accepts cross-namespace ingress traffic over TCP
    /80."
}
```

This queries a connectivity relationship that specifies source and destination, port, protocol, and direction. The solver's goal now is to verify whether communication is allowed, considering the NetworkPolicies affecting the endpoints. The *backend* does not have an ingress restriction preventing incoming communications from *client*, so the DT deems the *client*-to-*backend* connection possible over TCP/80 (this can also be verified by enabling the NetworkPolicy overlay from the Overview), and the *violation_when* condition triggers a violation (Figure 8.15).

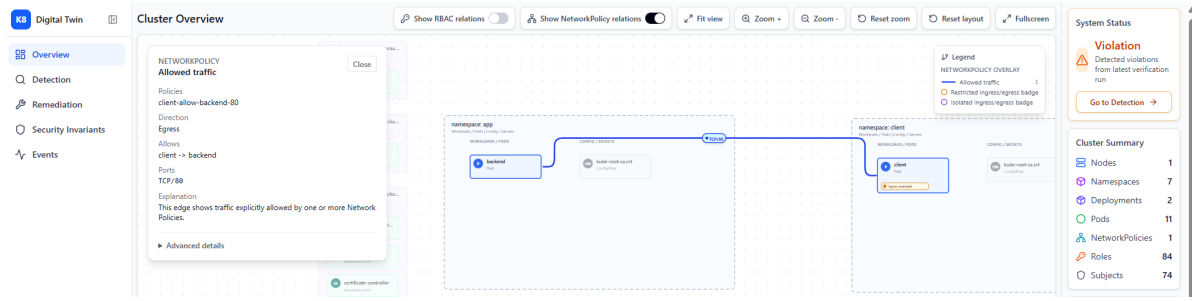


Figure 8.15: Cross-namespace connection visible in the Overview during the NetworkPolicy violation.

It's important to remember that each policy is identified by its namespace in addition to its canonical selector, precisely to prevent two policies with identical selectors but in different namespaces from being mistakenly aggregated into the same model element.

As with RBAC, the violation appears on the Detection page with its *query_id* and, this time, with category 'NetworkPolicy' and severity 'medium', in addition to the usual message and evidence (Figure 8.16).



Figure 8.16: NetworkPolicy violation shown in the Detection section.

At this point of the test, the details are almost identical to the previous RBAC case, so we'll focus directly on the remediation phase, where a strategy associated with the `network_policy.apply_manifest` handler is chosen. The materialized file affected by the changes will now be `network_policies.json`. The candidate fix proposes an ingress-type NetworkPolicy that selects only Pods with the *app:backend* label and is aggregated considering the additive property of NetworkPolicy:

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: dt-remediation-backend-ingress-<hash>
  namespace: app
  labels:
    app.kubernetes.io/managed-by: digital-twin-remediation
spec:
  podSelector:
    matchLabels:
      app: backend
  policyTypes:
    - Ingress
  ingress: []
```

An important feature to keep in mind when choosing templates for NetworkPolicy remediations is the direction: in this case, the query concerns an ingress connection to the backend, so a strategy that is not compatible with the identified traffic type is rejected with the `REMEDIATION_NOT_APPLICABLE` code, immediately excluding a useless precheck that would fail. Following the same RBAC workflow, the candidate status is verified, with the difference that now the focus is on the *cross_namespace_ingress_backend_80* query, which must no longer return the original connection (Figure 8.17).

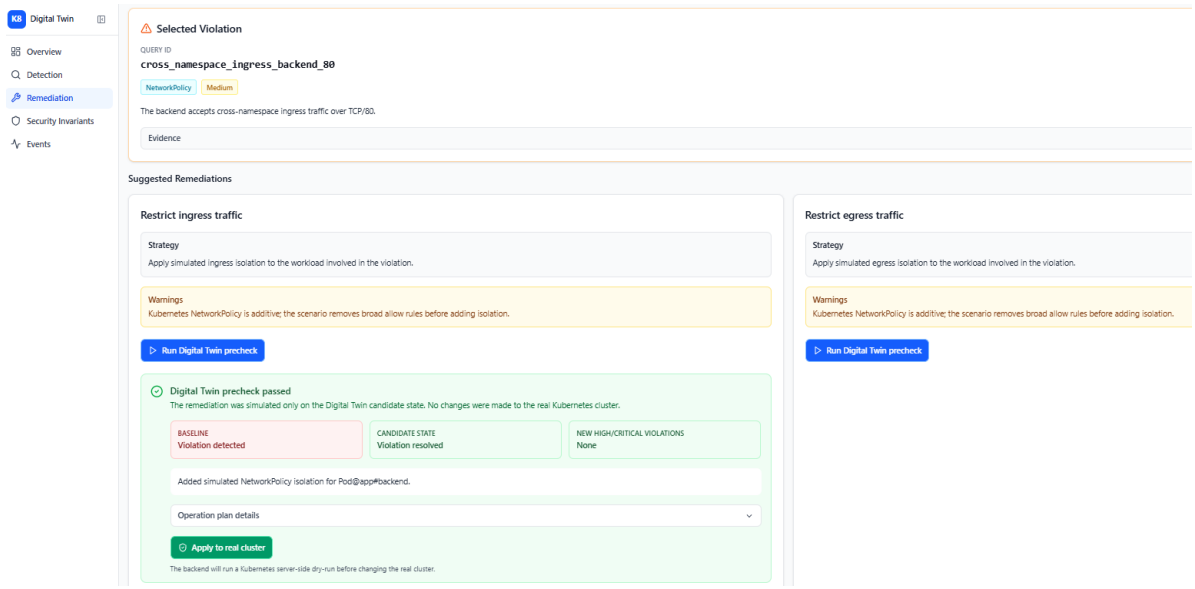


Figure 8.17: Digital Twin precheck for the selected NetworkPolicy remediation candidate.

When the checks described in the previous subsection are completed and the user's manual confirmation has been obtained, the handler generates the final manifest, whose name is based on a canonical form of the *dt-remediation-backend-ingress-<hash>* type, which follows Kubernetes' DNS-1123 constraints.

Once again, when the fix is applied to the live cluster, the *watcher* intercepts the creation of the new policy, and `network_policies.json` is regenerated to ensure that the post-verification acts on the synchronized version of the Digital Twin. The positive effect of the remediation can finally be evaluated by the absence of the query result in the violation report and by consulting the Overview, where the *backend* ingress badge now reflects the presence of the isolation, and the cross-namespace relationship used for the test no longer appears among the communications allowed by the model (Figure 8.18).

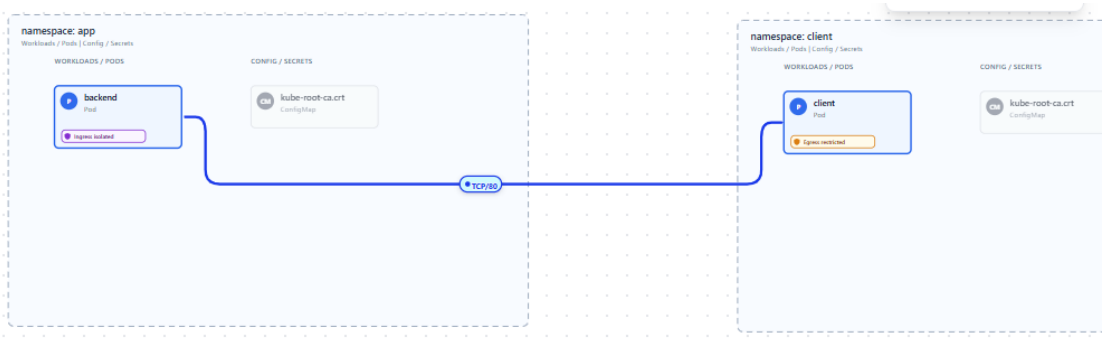


Figure 8.18: Overview after application and post-verification of the NetworkPolicy remediation.

Unlike the RBAC scenario, the NetworkPolicy scenario highlights a previously unseen feature, that is the state correction depends on the combined semantics of the selectors, direction, namespace, ports, and endpoint policies.

Chapter 9

Conclusions

The work carried out in this thesis successfully overcame the limitations of previous static verification tools and enabled the creation of a **runtime Kubernetes Digital Twin** capable of maintaining an **up-to-date representation** of the cluster and using it in the process of verifying security invariants. The initial goal was to extend an existing SMT approach (based on static snapshots) to create a Digital Twin that constantly reflects the Kubernetes cluster and automatically examines security properties after any configuration changes. The most significant contribution to achieving this goal was undoubtedly the introduction of a pipeline consisting of observation, update, verification, and remediation.

Unwrapping this chain, we begin with the *watcher* and the update logic, which allow the system to monitor the resources relevant to the model via the Kubernetes API Server and convert events into a uniform representation. This representation does not rely on independent snapshots as in the past, but on the current state that evolves with the cluster. The Proof of Concept served as a testing ground for this observation and **synchronization** phase, demonstrating how the addition, modification, or removal of resources in the cluster is immediately reflected in the materialized representation and dashboard.

This **materialized representation** is not a simple set of manifests, but rather distinct JSON files that include resources, grants, bindings, subjects, hierarchies, and network policies. These artifacts constitute the project's integration point because they are the data contract used both by the verification side for formal reasoning and by the dashboard backend for visualization, suppressing the need to maintain a copy of the model for the solver and a separate one for the UI.

A key achievement of this thesis is also the **integration** of this dynamic pipeline with the **existing SMT solver**, which is now activated whenever a significant change to the cluster is detected, bringing the framework to a significant level of autonomy. The prototype is modelled to distinguish semantically significant changes for the verification phase from operational variations present in each Kubernetes cluster, thus limiting the

number of checks triggered and saving time and resources.

The scenarios proposed in the Proof of Concept confirmed that the current prototype is capable of managing security mechanisms involving RBAC and NetworkPolicy. Tests on the second domain also demonstrated that the DT can evaluate properties that depend on the combination of namespace, selector, and policies applicable to the various endpoints.

Another result achieved is the generation of a **structured violation report** for invariants that fail during verification. These are no longer maintained in the basic form returned by the pre-existing solver but are converted into structured objects that include all the information useful for the *remediation engine* and the dashboard.

The secondary objective achieved with this thesis is to build an **automated remediation module** for calculating corrective actions if the cluster configuration violates at least one security property, restoring the system to a compliant state. The engine currently manages RBAC and NetworkPolicy cases starting from the violation report, which selects compatible strategies from a catalog of 16 templates. The precheck phase then ensures that no new violations are introduced with the proposed fix and that the current one disappears, while the dry-run ensures that the change can actually be applied to the real cluster from the Kubernetes API Server. The workflow concludes with the apply in the live environment, which in turn triggers a post-verification to ensure that the fix has been successfully applied and that the invariant has now the status Passing.

A lateral but equally important objective is the development of a **user interface** that complements the prototype, making information from the various architecture modules available through a dashboard:

- **Overview** provides a graph view of the cluster and allows for analyzing RBAC and NetworkPolicy relationships, overlaying them;

- **Events** displays the last 100 variations detected by the *watcher* and normalized to `ADDED`, `MODIFIED`, or `DELETED`;

- **Security Invariants** lists the properties being checked and their status;

- **Detection** collects unresolved violations, with category, severity, message, and evidence;

- **Remediation** guides the user through candidate fix selection, precheck, dry-run, manual-confirmation apply, and post-verification.

The PoC allowed, among other things, to validate the functioning of all components of this interface, from event synchronization to data materialization, from violation generation to remediation evaluation.

Overall, the results achieved should not be understood as a large-scale validation or complete security coverage in the Kubernetes environment, but rather as the possibility

of transforming a **static verification system into a runtime approach** capable of reflecting the **evolution of the cluster throughout its lifecycle**.

9.1 Future Works

This prototype can be considered the starting point for future improvements. The scenarios used by the Proof of Concept are controlled and limited, and a possible development would be to use the system to validate **broader and less predictable scenarios**, such as multi-node clusters comprising more workloads, events, resources, and policies. With these premises, the Digital Twin would be tested under conditions more similar to a real operating environment, with the possibility of measuring synchronization latency, verification times, throughput and scalability.

Furthermore, this work focused exclusively on two security domains of a Kubernetes cluster (RBAC and NetworkPolicy), but the approach could be applied to numerous **other security-relevant mechanisms** in Kubernetes, such as securityContext, Pod Security Standards, and Secrets management.

A similar enrichment approach can also be applied to the remediation section, as the playbook catalog currently covers only the two domains mentioned above, maintaining a minimal and conservative modification policy. **Expanding the catalog** is certainly one of the most obvious and useful future enhancement of this prototype, which also aligns with the first point of this section: larger configurations may require more diverse and increasingly sophisticated interventions, and a more extensive catalog would offer better adaptation to the context and identified issues.

The current model does not provide **automatic rollback** yet in case of an unsatisfactory remediation. A future development could consist of maintaining the state prior to the actual change so that it can be restored in a controlled manner if necessary. In this context, the persistence of the various DT states might also be expanded, as well as the detected violations and applied remediations, for increasingly thorough audits and retrospective comparisons.

In conclusion, moving from a local environment to a full Kubernetes deployment would be a significant improvement for the prototype. The various architecture modules could run as distributed components, presenting additional development themes, including availability, service isolation, credential management, and remediation engine protection. Carrying out an experiment along these lines would allow for functional validation to be supported by an **extensive quantitative evaluation**.

Bibliography

[1] A. Korolov. "Security in Kubernetes: Your stack is lying to you," Cloud Native Now. [Online]. Available: https://cloudnativenow.com/topics/cloudnativedevelopment/kubernetes/security-in-kubernetes-your-stack-is-lying-to-you/.

[2] Q. Zeng, M. Kavousi, Y. Luo, L. Jin, and Y. Chen, "Full-stack vulnerability analysis of the cloud-native platform," *Computers & Security*, vol. 129, Jun. 2023, Art. no. 103173. DOI: 10.1016/j.cose.2023.103173. [Online]. Available: https://doi.org/10.1016/j.cose.2023.103173.

[3] M. S. Rahaman, A. Islam, T. Cerny, and S. Hutton, "Static-analysis-based solutions to security challenges in cloud-native systems: Systematic mapping study," *Sensors*, vol. 23, no. 4, Feb. 4, 2023, Art. no. 1755. DOI: 10.3390/s23041755. [Online]. Available: https://doi.org/10.3390/s23041755.

[4] O. Sgan Cohen, E. Malul, Y. Meidan, D. Mimran, Y. Elovici, and A. Shabtai, "KubeGuard: LLM-assisted Kubernetes hardening via configuration files and runtime logs analysis," *arXiv preprint arXiv:2509.04191*, 2025. DOI: 10.48550/arXiv.2509.04191. [Online]. Available: https://arxiv.org/abs/2509.04191.

[5] F. Zaritto, E. Bravi, S. Sisinni, and A. Lioy, "Extending Kubernetes for pods integrity verification," *Journal of Network and Systems Management*, vol. 34, no. 1, 2026, Art. no. 14. DOI: 10.1007/s10922-025-09988-z. [Online]. Available: https://doi.org/10.1007/s10922-025-09988-z.

[6] F. Pizzato, D. Bringhenti, R. Sisto, and F. Valenza, "An intent-based solution for network isolation in Kubernetes," in *2024 IEEE 10th Conference on Network Softwarization (NetSoft)*, Saint Louis, MO, USA, 2024, pp. 381–386. DOI: 10.1109/NetSoft60951.2024.10588939. [Online]. Available: https://doi.org/10.1109/NetSoft60951.2024.10588939.

[7] D. Bringhenti, G. Marchetto, R. Sisto, F. Valenza, and J. Yusupov, "Introducing programmability and automation in the synthesis of virtual firewall rules," in *2020 IEEE 6th Conference on Network Softwarization (NetSoft)*, Ghent, Belgium, 2020,

pp. 473–478. DOI: 10.1109/NetSoft48620.2020.9165434. [Online]. Available: https://doi.org/10.1109/NetSoft48620.2020.9165434.

[8] T. Cerny and D. Taibi, "Static analysis tools in the era of cloud-native systems," *arXiv preprint arXiv:2205.08527*, 2022. DOI: 10.48550/arXiv.2205.08527. [Online]. Available: https://arxiv.org/abs/2205.08527.

[9] M. Singh, E. Fuenmayor, E. P. Hinchy, Y. Qiao, N. Murray, and D. Devine, "Digital twin: Origin to future," *Applied System Innovation*, vol. 4, no. 2, 2021, Art. no. 36. DOI: 10.3390/asi4020036. [Online]. Available: https://doi.org/10.3390/asi4020036.

[10] M. El-Hajj, T. Itäpelto, and T. Gebremariam, "Systematic literature review: Digital twins' role in enhancing security for industry 4.0 applications," *Security and Privacy*, vol. 7, no. 5, 2024, Art. no. e396. DOI: 10.1002/spy2.396. [Online]. Available: https://doi.org/10.1002/spy2.396.

[11] M. Homaei, Ó. Mogollón-Gutiérrez, J. C. Sancho, M. Ávila, and A. Caro, "A review of digital twins and their application in cybersecurity based on artificial intelligence," *Artificial Intelligence Review*, vol. 57, 2024, Art. no. 201. DOI: 10.1007/s10462-024-10805-3. [Online]. Available: https://doi.org/10.1007/s10462-024-10805-3.

[12] M. Repetto, "Cybersecurity digital twins: Concept, blueprint, and challenges for multi-ownership digital service chains," *Journal of Information Security and Applications*, vol. 96, Jan. 2026, Art. no. 104299. DOI: 10.1016/j.jisa.2025.104299. [Online]. Available: https://doi.org/10.1016/j.jisa.2025.104299.

[13] T. Zhao, E. Foo, and H. Tian, "A digital twin framework for cyber security in cyber-physical systems," *arXiv preprint arXiv:2204.13859*, 2022. DOI: 10.48550/arXiv.2204.13859. [Online]. Available: https://arxiv.org/abs/2204.13859.

[14] L. Huang, L. R. Varshney, and K. E. Willcox, "Formal verification of digital twins with TLA and information leakage control," *arXiv preprint arXiv:2411.18798*, 2024. DOI: 10.48550/arXiv.2411.18798. [Online]. Available: https://arxiv.org/abs/2411.18798.

[15] V. S. Barletta, D. Caivano, C. Catalano, A. Gentile, D. P. Posa, and A. Piccinno, "Cyber digital twin to improve security education: The RedTeam Knife approach," in *Proceedings of the 16th Biannual Conference of the Italian SIGCHI Chapter (CHItaly 2025)*, Salerno, Italy: ACM, 2025, 68:1–68:7. DOI: 10.1145/3750069.3750102. [Online]. Available: https://doi.org/10.1145/3750069.3750102.

[16] Y. Dai, J. Wu, S. Mao, X. Rao, B. Gu, Y. Qu, and Y. Lu, "Blockchain empowered access control for digital twin system with attribute-based encryption," *Future Generation Computer Systems*, vol. 160, pp. 564–576, 2024. DOI: 10.1016/j.future.

2024.06.037. [Online]. Available: https://doi.org/10.1016/j.future.2024.06.037.

[17] C. Grasselli, A. Melis, L. Rinieri, D. Berardi, G. Gori, and A. Al Sadi, "An industrial network digital twin for enhanced security of cyber-physical systems," in *2022 International Symposium on Networks, Computers and Communications (ISNCC)*, 2022, pp. 1–7. DOI: 10.1109/ISNCC55209.2022.9851731. [Online]. Available: https://doi.org/10.1109/ISNCC55209.2022.9851731.

[18] European Cyber Security Organisation (ECSO), "Ecso technical paper on digital twin security," European Cyber Security Organisation, Brussels, Belgium, Technical Paper, 2023. [Online]. Available: https://ecs-org.eu/?publications=ecso-technical-paper-on-cybersecurity-scenarios-and-digital-twins.

[19] T. Itäpelto, M. Elhajj, and M. van Sinderen, "Digital twin application in lifecycle security of critical infrastructures: A systematic literature review," *International Journal of Critical Infrastructure Protection*, vol. 50, Sep. 2025, Art. no. 100783. DOI: 10.1016/j.ijcip.2025.100783. [Online]. Available: https://doi.org/10.1016/j.ijcip.2025.100783.

[20] M. Kuštelega, R. Mekovec, and M. A. Shareef, "Privacy and security challenges of the digital twin: Systematic literature review," *Journal of Universal Computer Science*, vol. 30, no. 13, pp. 1782–1806, 2024. DOI: 10.3897/jucs.114607. [Online]. Available: https://doi.org/10.3897/jucs.114607.

[21] A. Mohsin, H. Janicke, S. Nepal, and D. Holmes, "Digital twins and the future of their use enabling shift left and shift right cybersecurity operations," in *2023 5th IEEE International Conference on Trust, Privacy and Security in Intelligent Systems and Applications (TPS-ISA)*, 2023, pp. 277–286. DOI: 10.1109/TPS-ISA58951.2023.00042. [Online]. Available: https://doi.org/10.1109/TPS-ISA58951.2023.00042.

[22] M. Eckhart, A. Ekelhart, D. Allison, M. Almgren, K. Ceesay-Seitz, H. Janicke, S. Nadjm-Tehrani, A. Rashid, and M. Yampolskiy, "Security-enhancing digital twins: Characteristics, indicators, and future perspectives," *IEEE Security & Privacy*, vol. 21, no. 6, pp. 64–75, 2023. DOI: 10.1109/MSEC.2023.3271225. [Online]. Available: https://doi.org/10.1109/MSEC.2023.3271225.

[23] Y. Jiang, W. Wang, J. Ding, X. Lu, and Y. Jing, "Leveraging digital twin technology for enhanced cybersecurity in cyber-physical production systems," *Future Internet*, vol. 16, no. 4, 2024, Art. no. 134. DOI: 10.3390/fi16040134. [Online]. Available: https://doi.org/10.3390/fi16040134.

[24] B. Canaan, Y. Makhlouf, and D. Ould Abdeslam, "Cyber-physical security for microgrids through digital twin technologies: A review and research outlook," *Energy Conversion and Management: X*, vol. 28, Oct. 2025, Art. no. 101406. DOI: 10.1016/j.ecmx.2025.101406. [Online]. Available: https://doi.org/10.1016/j.ecmx.2025.101406.

[25] A. D. Zemskov, Y. Fu, R. Li, X. Wang, V. Karkaria, Y.-K. Tsai, W. Chen, J. Zhang, R. Gao, J. Cao, K. A. Loparo, and P. Li, "Security and privacy of digital twins for advanced manufacturing: A survey," *arXiv preprint arXiv:2412.13939*, 2024. DOI: 10.48550/arXiv.2412.13939. [Online]. Available: https://arxiv.org/abs/2412.13939.

[26] A. Stefanidou, J. Cani, T. Papadopoulos, P. Radoglou-Grammatikis, P. G. Sarigiannidis, I. Varlamis, and G. T. Papadopoulos, "Leveraging digital twin technologies for public space protection and vulnerability assessment," in *2024 IEEE International Conference on Big Data (BigData)*, Washington, DC, USA, 2024, pp. 2836–2845. DOI: 10.1109/BigData62323.2024.10825359. [Online]. Available: https://doi.org/10.1109/BigData62323.2024.10825359.

[27] M. Masi, G. P. Sellitto, H. Aranha, and T. Pavleska, "Securing critical infrastructures with a cybersecurity digital twin," *Software and Systems Modeling*, vol. 22, no. 2, pp. 689–707, 2023. DOI: 10.1007/s10270-022-01075-0. [Online]. Available: https://doi.org/10.1007/s10270-022-01075-0.

[28] M. Dietz, L. Hageman, C. von Hornung, and G. Pernul, "Employing digital twins for security-by-design system testing," in *Proceedings of the 2022 ACM Workshop on Secure and Trustworthy Cyber-Physical Systems (SaT-CPS '22)*, Baltimore, MD, USA: ACM, 2022, pp. 97–106. DOI: 10.1145/3510547.3517929. [Online]. Available: https://doi.org/10.1145/3510547.3517929.

[29] M. Eckhart and A. Ekelhart, "Towards security-aware virtual environments for digital twins," in *Proceedings of the 4th ACM Workshop on Cyber-Physical System Security (CPSS '18)*, ACM, 2018, pp. 61–72. DOI: 10.1145/3198458.3198464. [Online]. Available: https://doi.org/10.1145/3198458.3198464.

[30] C. Zhou, H. Yang, X. Duan, D. Lopez, A. Pastor, Q. Wu, M. Boucadair, and C. Jacquenet, "Digital twin network: Concepts and reference architecture," Internet Research Task Force, Internet-Draft draft-zhou-nmrg-digitaltwin-network-concepts-07, Mar. 2022. [Online]. Available: https://datatracker.ietf.org/doc/html/draft-zhou-nmrg-digitaltwin-network-concepts-07.

[31] A. J. H. Redelinghuys, A. H. Basson, and K. Kruger, "A six-layer digital twin architecture for a manufacturing cell," in *Service Orientation in Holonic and Multi-Agent Manufacturing*, ser. Studies in Computational Intelligence, vol. 803, Cham,

Switzerland: Springer, 2019, pp. 412–423. DOI: 10.1007/978-3-030-03003-2_32. [Online]. Available: https://doi.org/10.1007/978-3-030-03003-2_32.

[32] D. Borsatti, W. Cerroni, L. Foschini, G. Y. Grabarnik, F. Poltronieri, D. Scotece, L. Shwartz, C. Stefanelli, M. Tortonesi, and M. Zaccarini, "Modeling digital twins of Kubernetes-based applications," in *2023 IEEE Symposium on Computers and Communications (ISCC)*, Gammarth, Tunisia, 2023, pp. 219–224. DOI: 10.1109/ISCC58397.2023.10217853. [Online]. Available: https://doi.org/10.1109/ISCC58397.2023.10217853.

[33] Kubernetes Authors. "Overview," Kubernetes Documentation. [Online]. Available: https://kubernetes.io/docs/concepts/overview/.

[34] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes, "Borg, omega, and Kubernetes," *ACM Queue*, vol. 14, no. 1, pp. 70–93, 2016. DOI: 10.1145/2898442.2898444. [Online]. Available: https://doi.org/10.1145/2898442.2898444.

[35] Kubernetes Authors. "Kubernetes components," Kubernetes Documentation. [Online]. Available: https://kubernetes.io/docs/concepts/overview/components/.

[36] Kubernetes Authors. "Nodes," Kubernetes Documentation. [Online]. Available: https://kubernetes.io/docs/concepts/architecture/nodes/.

[37] Kubernetes Authors. "Controllers," Kubernetes Documentation. [Online]. Available: https://kubernetes.io/docs/concepts/architecture/controller/.

[38] Kubernetes Authors. "Operator pattern," Kubernetes Documentation. [Online]. Available: https://kubernetes.io/docs/concepts/extend-kubernetes/operator/.

[39] Kubernetes Authors. "Declarative management of Kubernetes objects using configuration files," Kubernetes Documentation. [Online]. Available: https://kubernetes.io/docs/tasks/manage-kubernetes-objects/declarative-config/.

[40] Kubernetes Authors. "Namespaces," Kubernetes Documentation. [Online]. Available: https://kubernetes.io/docs/concepts/overview/working-with-objects/namespaces/.

[41] Kubernetes Authors. "Pods," Kubernetes Documentation. [Online]. Available: https://kubernetes.io/docs/concepts/workloads/pods/.

[42] Kubernetes Authors. "Pod lifecycle," Kubernetes Documentation. [Online]. Available: https://kubernetes.io/docs/concepts/workloads/pods/pod-lifecycle/.

[43] Kubernetes Authors. "Workload resources," Kubernetes Documentation. [Online]. Available: https://kubernetes.io/docs/concepts/workloads/controllers/.

[44] Kubernetes Authors. "Service," Kubernetes Documentation. [Online]. Available: https://kubernetes.io/docs/concepts/services-networking/service/.

[45] Kubernetes Authors. "Labels and selectors," Kubernetes Documentation. [Online]. Available: https://kubernetes.io/docs/concepts/overview/working-with-objects/labels/.

[46] Kubernetes Authors. "Configmaps," Kubernetes Documentation. [Online]. Available: https://kubernetes.io/docs/concepts/configuration/configmap/.

[47] Kubernetes Authors. "Secrets," Kubernetes Documentation. [Online]. Available: https://kubernetes.io/docs/concepts/configuration/secret/.

[48] Kubernetes Authors. "Service accounts," Kubernetes Documentation. [Online]. Available: https://kubernetes.io/docs/concepts/security/service-accounts/.

[49] Kubernetes Authors. "Overview of cloud native security," Kubernetes Documentation. [Online]. Available: https://kubernetes.io/docs/concepts/security/overview/.

[50] Kubernetes Authors. "Authorization," Kubernetes Documentation. [Online]. Available: https://kubernetes.io/docs/reference/access-authn-authz/authorization/.

[51] Kubernetes Authors. "Using RBAC authorization," Kubernetes Documentation. [Online]. Available: https://kubernetes.io/docs/reference/access-authn-authz/rbac/.

[52] B. Geesaman. "Hacking and hardening Kubernetes clusters by example," KubeCon + CloudNativeCon North America 2017. [Online]. Available: https://kccncna17.sched.com/event/CU6z/hacking-and-hardening-kubernetes-clusters-by-example-i-brad-geesaman-symantec.

[53] N. Yang, W. Shen, J. Li, X. Liu, X. Guo, and J. Ma, "Take over the whole cluster: Attacking Kubernetes via excessive permissions of third-party applications," in *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security (CCS '23)*, Copenhagen, Denmark: ACM, 2023, pp. 3048–3062. DOI: 10.1145/3576915.3623121. [Online]. Available: https://doi.org/10.1145/3576915.3623121.

[54] A. Sissodiya, E. Chiquito, U. Bodin, and J. Kristiansson, "Formal verification for preventing misconfigured access policies in Kubernetes clusters," *IEEE Access*, vol. 13, pp. 141 798–141 813, 2025. DOI: 10.1109/ACCESS.2025.3597504. [Online]. Available: https://doi.org/10.1109/ACCESS.2025.3597504.

[55] Kubernetes Authors. "Cluster networking," Kubernetes Documentation. [Online]. Available: https://kubernetes.io/docs/concepts/services-networking/.

[56] Kubernetes Authors. "Network policies," Kubernetes Documentation. [Online]. Available: https://kubernetes.io/docs/concepts/services-networking/network-policies/.

[57] J. Bufalino, J. L. Martin-Navarro, M. Di Francesco, and T. Aura, "Inside job: Defending kubernetes clusters against network misconfigurations," *Proceedings of the ACM on Networking*, vol. 3, no. CoNEXT3, pp. 1–25, 2025, Art. no. 20. DOI: 10.1145/3749220. [Online]. Available: https://doi.org/10.1145/3749220.

[58] Tigera. "Calico network policy," Calico Documentation. [Online]. Available: https://docs.tigera.io/calico/latest/network-policy/.

[59] Cilium Project. "Network policy," Cilium Documentation. [Online]. Available: https://docs.cilium.io/en/stable/security/policy/.

[60] H. Kang and S. Shin, "Verikube: Automatic and efficient verification for container network policies," *IEICE Transactions on Information and Systems*, vol. E105-D, no. 12, pp. 2131–2134, 2022. DOI: 10.1587/transinf.2022EDL8046. [Online]. Available: https://doi.org/10.1587/transinf.2022EDL8046.

[61] D. Bringhenti, G. Marchetto, R. Sisto, S. Spinoso, F. Valenza, and J. Yusupov, "Improving the formal verification of reachability policies in virtualized networks," *IEEE Transactions on Network and Service Management*, vol. 18, no. 1, pp. 713–728, 2021. DOI: 10.1109/TNSM.2020.3045781. [Online]. Available: https://doi.org/10.1109/TNSM.2020.3045781.

[62] D. Bringhenti, S. Bussa, R. Sisto, and F. Valenza, "Atomizing firewall policies for anomaly analysis and resolution," *IEEE Transactions on Dependable and Secure Computing*, vol. 22, no. 3, pp. 2308–2325, 2025. DOI: 10.1109/TDSC.2024.3495230. [Online]. Available: https://doi.org/10.1109/TDSC.2024.3495230.

[63] D. Bringhenti, L. Seno, and F. Valenza, "An optimized approach for assisted firewall anomaly resolution," *IEEE Access*, vol. 11, pp. 119 693–119 710, 2023. DOI: 10.1109/ACCESS.2023.3328194. [Online]. Available: https://doi.org/10.1109/ACCESS.2023.3328194.

[65] D. Bringhenti, G. Marchetto, R. Sisto, and F. Valenza, "A novel approach for security function graph configuration and deployment," in *2021 IEEE 7th International Conference on Network Softwarization (NetSoft)*, Tokyo, Japan, 2021, pp. 457–463. DOI: 10.1109/NetSoft51509.2021.9492654. [Online]. Available: https://doi.org/10.1109/NetSoft51509.2021.9492654.

[66] D. Bringhenti and F. Valenza, "Optimizing distributed firewall reconfiguration transients," *Computer Networks*, vol. 215, 2022, Art. no. 109183. DOI: 10.1016/j.comnet.2022.109183. [Online]. Available: https://doi.org/10.1016/j.comnet.2022.109183.

[67] F. Pizzato, D. Bringhenti, R. Sisto, and F. Valenza, "Intent-driven network isolation for the cloud computing continuum," *Journal of Network and Systems Management*, vol. 34, no. 1, 2026, Art. no. 6. DOI: 10.1007/s10922-025-09986-1. [Online]. Available: https://doi.org/10.1007/s10922-025-09986-1.

[68] D. Bringhenti, R. Sisto, and F. Valenza, "Security automation for multi-cluster orchestration in Kubernetes," in *2023 IEEE 9th International Conference on Network Softwarization (NetSoft)*, Madrid, Spain, 2023, pp. 480–485. DOI: 10.1109/NETSOFT57336.2023.10175419. [Online]. Available: https://doi.org/10.1109/NETSOFT57336.2023.10175419.

[69] F. Pizzato, D. Bringhenti, and F. Valenza, "A comparative study of authorisation mechanisms in Kubernetes-based service platforms," *Applied Cybersecurity & Internet Governance*, vol. 5, no. 2, pp. 1–29, 2026. DOI: 10.60097/ACIG/220492. [Online]. Available: https://doi.org/10.60097/ACIG/220492.

[70] K. Kanna and T. H. Kolbe, "Advanced user interaction with urban digital twins using large language models," *ISPRS Annals of the Photogrammetry, Remote Sensing and Spatial Information Sciences*, vol. X-G-2025, pp. 469–476, 2025. DOI: 10.5194/isprs-annals-X-G-2025-469-2025. [Online]. Available: https://doi.org/10.5194/isprs-annals-X-G-2025-469-2025.