Politecnico di Torino

this:

Master of Science in Cybersecurity Engineering

Master Degree Thesis

From Security Standard Definition to Centralized Posture Management: A Comprehensive Security Framework for Azure Kubernetes Service

Supervisors
prof. Fulvio Valenza

Candidate
Andrea Carcagnì

Academic Year 2024-2025

Summary

In the modern cloud landscape, Kubernetes has emerged as the de facto standard for container orchestration, allowing organizations to deploy, scale, and manage applications in an efficient and automated manner. Among the managed Kubernetes services, Azure Kubernetes Service (AKS) is one of the most widely adopted, thanks to its deep integration with the Microsoft Azure platform and its simplified operational model. As enterprises increasingly rely on AKS for critical workloads, securing these environments has become an increasingly important topic. Protecting Kubernetes clusters on Azure requires more than applying security best practices; it requires continuous compliance, observability, and governance to match enterprise security and regulatory requirements. The challenge becomes even greater for organizations that manage multiple, fully isolated customer environments. In these scenarios, strict isolation often conflicts with Azure's native management tools, such as Azure Lighthouse, which are designed for centralized multi-tenant administration. While these tools streamline operations, they inherently introduce shared control planes and permission boundaries that are unsuitable for environments requiring strong isolation guarantees.

The thesis addresses this challenge directly by performing a thorough analysis of the current state of security in AKS and by translating it into an enterprise security standard applicable to Azure-managed Kubernetes clusters. The research led to the definition of a structured set of technical and organizational requirements aimed at standardizing cluster hardening and governance. Based on that foundation, the work illustrates the design and implementation of a dedicated Azure-based infrastructure for automating the assessment of the defined security posture. The proposed solution allows centralized visibility and compliance monitoring while maintaining the complete isolation of customer environments. Through this architecture, the thesis provides a scalable and auditable model for securing and managing AKS clusters in complex enterprise multi-client scenarios filling the gap between isolated clusters and centralized governance.

Contents

Chapter 1

Introduction

1.1 Problem

Modern enterprises increasingly adopt Kubernetes as the foundational layer for running cloud-native workloads, and Azure Kubernetes Service (AKS) has quickly become one of the most widely deployed managed Kubernetes platforms. As the operational use of AKS grows across business units, subsidiaries, and customers, security becomes both more critical and more complex. Organizations must protect clusters that differ in maturity, configuration, and operational practices while ensuring compliance with internal standards, external regulations, and industry benchmarks.

The core difficulty lies in achieving this level of security and consistency at scale. When clusters are distributed across multiple isolated Azure tenants, `each belonging to a different customer or organizational domain`, the challenge intensifies. Azure-native tooling such as Azure Policy, Defender for Cloud, and centralized monitoring services provide strong capabilities, but they are designed with shared-control models in mind. For organizations that enforce strict isolation between tenants, these native mechanisms become difficult or impossible to use directly. Techniques such as Azure Lighthouse introduce shared administrative planes and cross-tenant privileges that may conflict with regulatory or contractual constraints. As a result, many enterprises continue to manage AKS security in a fragmented, manual, and error-prone manner, with limited visibility into the posture of the clusters they must secure.

In this context, the absence of a unified governance model leads to inconsistent configurations, uneven adoption of security features, and an inability to monitor compliance across customers. Without a structured baseline and a secure centralized mechanism to distribute and assess it, the security posture of AKS clusters diverges over time, increasing operational risk and audit complexity. The problem addressed by this thesis is therefore how to enforce a uniform, repeatable, and scalable security framework for AKS clusters across multiple isolated tenants, without compromising the strict separation required between customers.

1.2 Thesis Goal

The objective of this thesis is to define, implement, and validate a comprehensive security framework for Azure Kubernetes Service that enables centralized posture management across fully isolated Azure tenants. The work is structured around two major contributions.

First, the thesis develops a formal AKS security baseline, built exclusively on Microsoft-supported Azure Policy controls and aligned with recognized security standards such as the CIS Kubernetes Benchmark, Kubernetes Pod Security Standards, NIST guidance, and CNCF security principles. This baseline represents a unified set of mandatory requirements applicable to all AKS clusters, independently of the environment, application domain, or operational maturity of the teams managing them.

Second, the thesis introduces the design and implementation of an Azure-based architecture for securely distributing this baseline and monitoring compliance in a multi-tenant context. The proposed solution creates a dedicated "tenant server" acting as a centralized policy distribution and ingestion hub and a "tenant client" deployed inside each isolated customer tenant. This bidirectional model enables:

- secure and private distribution of the baseline through Azure Storage, Private Endpoints, and declarative manifests,
- independent evaluation and application of policies through local automation,
- continuous collection of compliance signals using Azure Policy, Gatekeeper, and the Azure Policy Add-on for AKS,
- centralized ingestion, analytics, and visualization via Azure Data Explorer and Azure Managed Grafana.

The final goal is to bridge the gap between strict tenant isolation and the need for centralized governance, providing an architecture that is scalable, fully auditable, and aligned with Azure best practices. The solution demonstrates that it is possible to maintain isolation at the identity, network, and operational levels while still achieving enterprise-wide standardization and visibility of AKS security posture.

1.3 Thesis Structure

The thesis is organized to guide the reader from the foundations of AKS and Kubernetes security to the design, implementation, and validation of the proposed framework.

- **Chapter 1** provides the technical background on AKS, including the managed control plane, agent pools, networking, Azure-native security features, workload hardening mechanisms, and the primitives that underpin secure Kubernetes operation. This chapter establishes the terminology and concepts that the rest of the thesis builds upon.

- **Chapter 2** introduces the high-level implementation design of the AKS Security Framework (AKSSF). It describes the overall architecture built around two main components: the tenant client, deployed in each tenant to synchronize and enforce security baselines and collect compliance snapshots, and the tenant server, which centralizes baseline publication, snapshot ingestion, and observability.
- **Chapter 3** introduce and formalize the AKS security baseline. They describe the selection criteria for Azure Policy controls, the mapping to industry benchmarks, and the structure of the baseline across governance, identity, workload hardening, networking, and data protection domains. These chapters represent the theoretical foundation supporting the architectural design that follows.
- **Chapter 4** presents the architecture of the tenant server, the centralized component responsible for storing the baseline, exposing manifests, securing access through Private Link, and ingesting compliance data from all tenant clients. The chapter explains the network architecture, authentication model, storage structure, and the rationale behind the chosen Azure services. An important part of the chapter is the presentation of the final dashboard deployed in the server demonstrating the solutions correctness, scalability, and ability to maintain strict tenant isolation while providing centralized posture visibility.
- **Chapter 5** focuses on the tenant client, the component deployed in each isolated Azure tenant. It details how policies are synchronized, validated, and applied to local clusters, how secrets are managed in Key Vault, and how audit results are exported through the ingestion pipeline.

1.4 Environment Context

The proposed framework has been designed for a multinational organization that manages multiple Azure tenants, each containing one or more AKS clusters. These tenants belong to different customers or independent business units and are subject to strict isolation requirements. No shared control planes, cross-tenant privileges, or administrative trust relationships are permitted, which excludes the use of common centralized management patterns such as Azure Lighthouse or cross-tenant Managed Identities.

In this environment, every tenant owns its virtual networks, identity boundaries, security policies, and operational autonomy. Clusters vary in size, purpose, and security maturity: some host internal applications for development teams, others run production workloads for external customers subject to regulatory constraints. Despite this heterogeneity, the organization must guarantee a uniform security posture, consistent application of mandatory controls, and centralized visibility of cluster compliance.

The architecture developed in this thesis responds precisely to this context. It enables a secure flow of configuration and compliance data across tenants using

private networking and isolated identities, while preserving the autonomy and privacy of each customer environment. The solution therefore represents not only a technical contribution but also a practical model for managing AKS security in large-scale, multi-tenant enterprise scenarios.

Chapter 2

Background

2.1 Introduction to Azure Kubernetes Service (AKS)

2.1.1 What AKS is and Why It Is a Managed Service

The landscape of cloud computing and modern architectures has evolved significantly with the adoption of containers. Containers enable applications and their dependencies to be packaged into portable and isolated units. Kubernetes has become the leading open-source platform for large-scale container orchestration, providing essential capabilities such as scheduling, auto-scaling, load balancing, self-healing, and storage management.

However, managing a Kubernetes cluster from scratch introduces substantial complexity. Administrators must maintain the control plane, update *etcd*, ensure high availability and security, and preserve a consistent configuration across the cluster.

Azure Kubernetes Service (AKS) [1] addresses these challenges by simplifying cluster operations through a fully managed service model. AKS runs Kubernetes on Microsoft Azure, with Microsoft responsible for managing the entire control plane: the API server, *etcd*, scheduler, and controllers. This management covers provisioning, patching, updating, uptime guarantees, and automatic scaling .

The managed service approach provides two major benefits. First, it reduces operational overhead: organizations do not need to maintain or update Kubernetes master components and can focus solely on the worker nodes where containers run. Second, it enables rapid deployment: creating an AKS cluster provides an immediately usable production-ready Kubernetes environment with networking, storage, and Azure integrations already configured.

AKS is therefore well-suited for organizations aiming to adopt cloud-native architectures, leverage containerization and microservices, and reduce infrastructure management efforts while relying on a secure, supported, and continuously updated platform.

2.1.2 Shared Responsibility Model: What Microsoft Protects and What the Customer Must Protect

Using a managed service such as AKS modifies the traditional division of responsibilities between the cloud provider and the customer, introducing a shared responsibility model. Microsoft ensures the security and availability of the control plane [2], managing patches, resilience, uptime, and the backend Kubernetes infrastructure.

However, the organization consuming AKS remains responsible for the worker nodes, containers, workloads, virtual network configuration, applications, secrets, and data. In other words, everything below the control plane, like virtual machines, storage, networking, Pod configuration, and security policies,remains under the customer's control. Consequently, the overall security posture of the environment depends largely on architectural decisions, operational practices, and governance strategies.

This hybrid model provides a balance. Microsoft offers reliability and ease of management for the control plane, while the customer maintains flexibility and responsibility for the security measures required by their environment. For these reasons, using AKS does not mean delegating security entirely. The platform alone does not guarantee a secure setup. It is necessary to implement appropriate best practices, configurations, and policies at both infrastructure and workload levels to ensure confidentiality, integrity, and availability.

2.2 Internal Architecture of AKS

The internal architecture of Azure Kubernetes Service (AKS) combines the core principles of Kubernetes with a high level of automation, abstraction, and native integration with Azure services. Its design aims to provide scalability, security, resilience, and operational simplicity, while remaining fully compatible with the Kubernetes open-source ecosystem. To understand how an AKS cluster works, it is necessary to consider both the components managed by Microsoft and the operational elements that remain under the user's control.

2.2.1 The Managed Control Plane

In AKS, the entire Kubernetes control plane is fully managed by Microsoft. Critical components such as the `kube-apiserver`, the distributed store `etcd`, the scheduler, and the controller-manager are hosted and maintained by Azure. Microsoft provides updates, security patches, monitoring, and service continuity, ensuring a reliable and highly available control plane [3].

The `kube-apiserver` is the entry point for all cluster operations. Through HTTP requests sent to the API endpoint, applications, operators, and DevOps tools define the desired state of the system. Azure ensures the availability of this endpoint through high-availability mechanisms and load balancing, as well as support for private clusters and authorized IP ranges that reduce exposure to the public internet.

The `etcd` database stores the entire state of the cluster and is one of the most sensitive components because it represents the core of Kubernetes declarative model. In AKS, `etcd` is fully isolated and not accessible to the customer. It is protected through encryption and managed using Azure operational mechanisms, including automated backups and failover.

The kube-scheduler and controller-manager ensure alignment between the current and the desired cluster state. The scheduler assigns new Pods to suitable nodes, while the controllers manage replicas, failed nodes, deployments, and other cluster objects. These functions are entirely handled by Azure, relieving the customer from complex operational tasks such as control plane tuning, updates, patching, or recovery procedures.

2.2.2 Cluster Nodes: Agent Pools, VMSS, Operating Systems, Container Runtime

While the control plane is fully managed by Microsoft, the cluster nodes, which are the machines that actually run user workloads, remain part of a shared responsibility model. Each AKS node is an Azure virtual machine that belongs to one or more agent pools. Each pool contains a homogeneous group of nodes with the same operating system, machine size, and networking configuration.

Agent pools are implemented through Azure Virtual Machine Scale Sets (VMSS). VMSS provides native capabilities such as autoscaling, orchestrated updates, rolling upgrades, and automatic VM health management. This model enables the cluster to scale quickly in response to application load changes [4].

Regarding operating systems, AKS supports Azure Linux, Ubuntu, and Windows Server for Windows-based containers. Linux nodes rely on CRI-compliant container runtimes which became the recommended and default runtime.

Each node includes essential data plane components such as the `kubelet`, responsible for maintaining Pod execution, and `kube-proxy`, which implements networking rules and packet forwarding within the cluster.

2.2.3 Pods, ReplicaSets, Deployments, and Basic Kubernetes Resources

Pods are the smallest execution unit in Kubernetes. A Pod can contain one or more containers sharing the same network namespace and, optionally, shared storage volumes. The ephemeral nature of Pods requires applications to be designed for resilience. They should be stateless or capable of handling the loss of an instance [5].

ReplicaSets provide a higher level of abstraction by automatically maintaining a specific number of Pod replicas. This control mechanism is essential to ensure availability and fault tolerance.

Deployments operate at an even higher level and define update strategies such as rolling updates and rollbacks. They simplify the lifecycle management of containerized applications. The combination of Deployments and ReplicaSets is central to

cloud-native architectures because it ensures operational continuity and predictable updates [6].

2.2.4 Namespaces and Logical Isolation

Kubernetes provides an initial segmentation layer through Namespaces, which divide resources into logical groups within the same cluster. Namespaces make it possible to implement multi-team, multi-project, or multitenant models by defining visibility boundaries and applying granular controls such as network policies, resource limits, RBAC roles, and labels [7].

In AKS, several namespaces are created automatically, including `kube-system`, `kube-public`, and `kube-node-lease`. The `kube-system` namespace is especially important because it hosts the internal cluster components and the system add-ons installed by AKS. As a result, namespaces act as operational boundaries that must be respected and protected, for example by applying security policies that prevent arbitrary workloads from running in reserved namespaces.

2.2.5 Node Resource Group and Integration with Azure Resources

A distinctive element of AKS compared to a self-hosted Kubernetes cluster is the presence of the Node Resource Group (NRG). When a cluster is created, Azure automatically generates a dedicated resource group that contains all the infrastructure objects required for the nodes to operate. These include virtual machines, disks, network interfaces, internal load balancers, managed identities, and related configurations.

This resource group should not be modified manually because it is managed by the system. Any unplanned change may affect the cluster's stability. At the same time, the NRG makes it possible to use Azure integration features such as advanced networking, disk encryption, monitoring, logging, and autoscaling. [8].

The Node Resource Group highlights the connection between Kubernetes and Azure. Kubernetes provides the declarative model and orchestration, while Azure supplies the cloud resources, security, networking, and monitoring that turn the cluster into a fully integrated environment.

2.3 Networking in AKS

Networking is one of the most complex and important aspects of a Kubernetes cluster. In Azure Kubernetes Service, networking is not only the mechanism through which Pods communicate. It is also a central component for security, service exposure, performance, and integration with the wider Azure ecosystem. AKS combines standard Kubernetes features such as Services, Ingress, and Network Policies with Azure-specific implementations including Azure CNI, Azure Load Balancer,

network security groups (NSG), and native integration with Azure Firewall and Private Link.

Understanding the networking models, ingress and egress behavior, and security mechanisms is essential when designing or securing an AKS cluster.

2.3.1 Core CNI Concepts: Overlay vs. Flat Networking

Kubernetes uses a Container Network Interface (CNI) to assign IP addresses to Pods, define routing rules, and enforce network policies. In AKS, users can select from different CNI plugins, each with its own characteristics and trade-offs.

Overlay and flat are the two main networking approaches used by modern CNIs.

Overlay model In an overlay model, Pods receive IP addresses from a range that is separate from the Azure subnet hosting the nodes. Traffic between Pods is encapsulated and routed through a tunneling or virtualization layer, which avoids consuming large numbers of IP addresses from Azure subnets. This reduces IP usage and improves scalability for very large clusters [9].

Flat model In a flat model, Pods and nodes share the same L3 subnet. Each Pod receives a native VNet IP address and becomes a fully addressable entity within Azure. This model simplifies integration with on-premises systems, security appliances, and monitoring solutions. The traditional Azure CNI implementation follows this flat approach [10].

In addition to Azure CNI, AKS also supports two other models:

- **Cilium (eBPF)** – Supported natively since 2023. Cilium relies on eBPF to deliver high-performance networking, advanced security, and deep observability [11, 12].
- **Kubenet** –A simpler and now legacy solution that relies on routing and NAT. It is no longer recommended for modern environments or large clusters [13].

The choice of CNI affects security, performance, scalability, and integration with Azure infrastructure. For this reason, it is one of the most strategic decisions when designing an AKS cluster.

2.3.2 Intra-Cluster Communication: Pod-to-Pod, Service-to-Pod, Node-to-Pod

The Kubernetes networking model ensures that any Pod can communicate with any other Pod in the cluster, regardless of the node they run on. This behavior, known as the flat networking model, is a core design principle of Kubernetes [14].

The main communication flows include:

Pod-to-Pod – This is the simplest communication pattern. With Azure CNI flat, routing is direct through the VNet. With overlay models, traffic passes through the encapsulation layer defined by the CNI.

Service-to-Pod – When a Pod accesses a Kubernetes Service, requests are routed using internal load balancing. Azure implements kube-proxy using iptables or eBPF (when Cilium is enabled) to send traffic to the correct Pods associated with the Service.

Node-to-Pod and Pod-to-Node – Nodes and their kubelet components communicate with Pods for health checks and management operations. These flows are essential to understand from a security perspective because they define the minimum network paths required for Kubernetes to function.

Microsoft documentation provides a detailed explanation of these internal flows [15].

2.3.3 Kubernetes Services: ClusterIP, NodePort, LoadBalancer, ExternalName

Kubernetes Services provide an abstraction for exposing applications. In AKS, Services integrate tightly with Azure networking components such as Load Balancer and Private Link.

ClusterIP is the default Service type and creates a virtual IP address accessible only within the cluster for Pod-to-Pod or Pod-to-Service communication.

NodePort exposes an application on a fixed port on every node. It is simple but not secure, and AKS does not recommend it for production environments.

LoadBalancer automatically creates an Azure Standard Load Balancer and associates a public or private IP with it. This is one of the most common Service types for applications that need external access.

ExternalName maps the Service to an external DNS name and is typically used for logical proxies or integrations with external services.

From a security perspective, NodePort and LoadBalancer represent attack surfaces that require careful control, especially when exposed to the public internet.

2.3.4 Ingress Controllers and L7 Traffic

Kubernetes Services do not directly manage HTTP or HTTPS traffic and do not provide capabilities such as TLS termination, hostname-based routing, URL rewriting, or rate limiting. For this reason, Kubernetes defines the Ingress resource and requires an Ingress controller.

In AKS, the most common Ingress controllers are:

- NGINX Ingress Controller
- Azure Application Gateway Ingress Controller (AGIC)

- Traefik
- Cilium Ingress (when using Cilium as the data plane)

The controller translates the routing rules defined in the Ingress object into operational configurations and exposes HTTP/HTTPS services efficiently at L7.

Ingress controllers introduce several important security aspects, including TLS handling, rate limiting, URL sanitization, Web Application Firewall integration (such as AGIC with WAF), and the use of certificates stored in Azure Key Vault.

2.3.5 Cluster Egress: SNAT, Azure Firewall, Private Endpoints

Egress traffic is often underestimated, but it represents one of the most critical parts of AKS security. By default, Pods reach the internet through the node, which performs Source Network Address Translation (SNAT). As a result, all outgoing traffic is masked behind the node's IP address.

The risks of SNAT include NAT table exhaustion, loss of fine-grained observability for individual workloads, and difficulty enforcing precise security controls.

Recommended best practices include:

- routing outbound traffic through Azure Firewall,
- enforcing traffic flows using User-Defined Routes (UDR),
- using private endpoints for Azure PaaS services,
- disabling direct outbound access in zero-trust environments [16].

2.3.6 Network Security Groups and Their Role Compared to Network Policies

Network Security Groups (NSG) are the main filtering mechanism in Azure. They can be applied to subnets or network interfaces and operate at layers 3 and 4, allowing administrators to control inbound and outbound traffic to nodes. However, NSGs are not aware of Pods. They can filter traffic to and from nodes but cannot distinguish individual containers. This is a major limitation in Kubernetes environments [17].

For AKS, NSGs play a complementary role. They protect the Azure infrastructure layer but do not replace Kubernetes Network Policies.

2.3.7 Network Policies: Models, Enforcement, Limitations, and Risks

Kubernetes Network Policies provide granular control over Pod-to-Pod traffic and make it possible to adopt zero-trust models inside the cluster. Unlike NSGs, Network Policies operate on Pods, namespaces, and labels, offering an application-level perspective on traffic [18].

Kubernetes provides a standard API, but the actual enforcement depends on the CNI implementation. AKS supports [19]:

- Calico Network Policy,
- Azure CNI Network Policy (available when using Azure CNI powered by Cilium),
- Cilium Network Policy (advanced eBPF enforcement).

The limitations of Network Policies include:

- lack of L7 visibility (unless combined with a service mesh or advanced CNI),
- increased operational complexity in large environments,
- dependency on the CNI provider,
- risk of misconfigurations that may block critical system traffic.

The most serious risk is the absence of Network Policies. Without them, any Pod can communicate with any other Pod, creating a large potential lateral movement path for attackers.

2.4 Kubernetes Security Model

The security model of Kubernetes differs significantly from that of traditional infrastructures. In a cluster, there is no single perimeter to protect. Instead, the environment consists of dozens or hundreds of cooperative processes distributed across multiple nodes, dynamically orchestrated, and often managed by several teams with different access levels. This makes the attack surface inherently broad and constantly evolving.

Kubernetes follows a declarative and highly flexible approach, but this flexibility introduces risks if not paired with strict controls. Security is not limited to protecting the control plane. It also involves nodes, containers, workloads, identity models, API objects, and intra-cluster communications. An effective strategy must begin by understanding the typical threats and risk categories and then link them to the native security mechanisms provided by Kubernetes.

2.4.1 Typical threats: container breakout, privilege escalation, lateral movement

Kubernetes was designed as a container orchestrator and therefore inherits many threat patterns from container runtimes. However, unlike standalone containers, a multi-tenant cluster amplifies risks because of workload density, shared nodes, and the dynamic nature of cluster resources.

The first major threat category is container breakout, which refers to an attacker escaping from a container and gaining access to the underlying node. This scenario becomes realistic when dangerous configurations are used, such as privileged containers, shared node namespaces (hostPID, hostNetwork), dangerous Linux capabilities such as `CAP_SYS_ADMIN`, or direct mounts of node filesystems. The National Institute of Standards and Technology (NIST) identifies these risks as among the most critical in modern containerized architectures [20].

A second major threat involves privilege escalation, either inside the container or through the `kube-apiserver`. Running containers as root, writable filesystems, and the absence of seccomp or AppArmor profiles increase the likelihood of privilege escalation. Similarly, an overly permissive RBAC role assigned to a service account can grant dangerous capabilities at the cluster level.

A third common threat is lateral movement. Without Network Policies, any Pod can communicate with any other Pod, resulting in a completely flat network. The MITRE ATT&CK framework for Containers highlights unrestricted intra-cluster communication as one of the most frequent paths used by attackers after compromising a single workload [21].

These three categories form the core of workload and node-level risks, and many defensive measures focus specifically on mitigating them.

2.4.2 The cluster as a distributed system: attack surfaces and vectors

Kubernetes is not a single component but an ecosystem of coordinated processes such as the API server, controllers, scheduler, kubelet, container runtime, internal DNS, CNI, and the `etcd` datastore. Cluster security emerges from the correct interaction of these elements rather than from the security of any individual component.

The attack surface can be divided into five main areas.

The control plane is the most critical part of the cluster. It hosts the `kube-apiserver`, scheduler, controller manager, and `etcd`. The `kube-apiserver` is the central entry point for all administrative operations and a natural target for attackers. `Etcd` contains cluster configuration and Secrets, which requires particularly strong protection.

Nodes are another fundamental attack surface. They are virtual machines running containers, the runtime, the kubelet, and `kube-proxy`. A compromised node effectively means a compromised cluster, which is why nodes must remain highly protected and not directly exposed.

Containers are isolated processes, but the isolation relies on kernel-level techniques and does not offer the same guarantees as virtual machines. Their security depends on mechanisms such as `securityContext`, filesystem configuration, Linux capabilities, and seccomp or AppArmor profiles.

Intra-cluster communications, if not restricted by Network Policies, create a privileged propagation path. East-west traffic between Pods was identified in the Red Hat and CNCF Cloud Native Security Report 2024 as the most commonly exploited vector after an initial compromise.

Finally, the Kubernetes API surface is extremely large. The declarative model enables many operations that, if misconfigured, can create systemic vulnerabilities. Examples include creating privileged Pods, modifying critical ConfigMaps, manipulating ServiceAccounts, obtaining tokens, port-forwarding to sensitive Pods, or changing RBAC permissions.

2.4.3 Kubernetes RBAC model: users, groups, service accounts, and the role of kube-apiserver

Kubernetes applies a Role-Based Access Control (RBAC) model to manage which operations an identity can perform against the cluster API. The official documentation describes RBAC as the primary mechanism to enforce the principle of least privilege across the cluster [22].

In the RBAC model, there are three main identity categories.

Users represent human identities that authenticate to the cluster. Kubernetes does not manage users directly; authentication is handled through external systems such as OpenID Connect, Microsoft Entra ID, or client certificates. After authentication, the user interacts with the `kube-apiserver`, which checks authorization using RBAC rules.

Groups aggregate users into logical units and are essential in organizational environments. Microsoft Entra ID, which integrates natively with AKS, supports direct mapping between enterprise groups and Kubernetes roles.

Service accounts represent non-human identities used by Pods to interact with the cluster API. Every Pod is associated with a service account. The `kube-apiserver` treats service accounts as first-class identities and RBAC defines what they can do.

The `kube-apiserver` acts as the central authority by authenticating, authorizing, and applying admission control. All requests flow through it, making it the primary enforcement point for RBAC.

RBAC configuration has a direct impact on cluster security. Broad roles such as `cluster-admin` or `edit` represent significant risks in multi-team environments. The CNCF reports that many Kubernetes security incidents stem from improper use of privileges rather than technical vulnerabilities [23].

2.4.4 Kubernetes Secrets: management, limitations, encryption, risks when mismanaged

Kubernetes Secrets provide the native mechanism for storing sensitive information such as tokens, passwords, API keys, and certificates. Their main benefit is that sensitive values do not need to be embedded directly in Pod manifests. However, Secrets come with important limitations.

By default, Kubernetes stores Secrets in `etcd` without encryption. This means that if `etcd` is not properly protected, an attacker could retrieve Secrets in plaintext. The documentation recommends encryption at rest as a minimum requirement for production clusters [24].

In AKS, encryption of Secrets is handled through Azure Key Management Service (KMS), which allows the use of keys stored in Azure Key Vault. This integration addresses one of Kubernetes' historical limitations and provides a stronger trust model [25].

A second limitation is the automatic mounting of service account tokens into Pods. This token is stored in the Pod's filesystem and may be accessible to attackers in case of remote code execution. Since 2020, the community strongly recommends disabling `automountServiceAccountToken` whenever it is not required.

A third risk involves duplication of Secrets. CI/CD pipelines and applications may create unprotected local copies of credentials, break privilege separation, or create drift between cluster-managed and externally managed credentials.

Finally, Secrets do not provide automatic rotation. Kubernetes does not rotate passwords, certificates, or tokens by default, which introduces risk if a credential is compromised.

Overall, Secrets are useful but must be complemented by strong mechanisms such as KMS, accurate RBAC, Workload Identity, and secure node configurations.

2.4.5 Pod Security Standards: baseline, restricted, and privileged

The Pod Security Standards (PSS) are the main official guidelines from the Kubernetes community for evaluating the security level of workloads. PSS define three profiles, each with specific requirements and increasing strictness: Privileged, Baseline, and Restricted [26].

The Privileged level allows almost all behaviors, including privileged containers, access to host namespaces, extended capabilities, hostPath volumes, and node filesystem mounts. It is intended for system components such as CNI or CSI plugins and should not be used for application workloads.

The Baseline level provides essential protection against the most dangerous configurations. It blocks privileged containers, excessive capabilities, unsafe hostPath usage, shared host namespaces, and other risky settings. It is compatible with most applications and is recommended for multi-team environments.

The Restricted level applies even stricter controls. It requires read-only filesystems, minimal Linux capabilities, the `seccomp RuntimeDefault` profile, and prevents almost all access to node primitives. This level represents a zero-trust approach for workloads.

PSS enforcement relies on Pod Security Admission (PSA), which is enabled as a native admission controller in recent Kubernetes versions.

The Pod Security Standards are a foundational element for understanding workload security. They map theoretical security expectations to concrete rules that should be allowed or denied at the manifest level.

2.5 AKS Security From the Azure Perspective

The security model of AKS extends beyond the Kubernetes surface. A significant part of cluster protection comes from its deep integration with Azure security services. Unlike a bare-metal Kubernetes cluster or an installation running on self-managed virtual machines, AKS can delegate critical aspects of security, compliance, monitoring, identity, key management, and data encryption to the Azure platform. Azure operates under certified security standards such as ISO 27001, SOC 1/2/3, and FedRAMP, which strengthens its overall security posture.

This section examines the main Azure-native mechanisms that complement Kubernetes security and form an essential layer of protection for AKS clusters.

2.5.1 Integration With Microsoft Entra ID (Identity Federation and OIDC)

One of the strengths of AKS is the ability to fully delegate user authentication to Microsoft Entra ID (formerly Azure Active Directory). Kubernetes does not include a built-in identity management system and always requires an external authentication provider. Integrating AKS with Entra ID allows organizations to use their existing enterprise identity platform and apply controls such as multi-factor authentication, Conditional Access, and Identity Protection.

The integration also enables the use of the cluster's OIDC provider, which is crucial for modern workloads. The Kubernetes API server exposes a compliant OIDC discovery endpoint that supports identity federation between service accounts and Microsoft Entra ID. Workloads can authenticate to Azure resources without relying on static secrets. This mechanism, known as Workload Identity, is now considered the standard for secure authentication between containerized applications and cloud services [27].

Integration with Entra ID improves overall security in at least three ways: it removes the need for the local `-admin` cluster account; it enables Kubernetes role assignments based on enterprise groups; it reduces the use of static credentials by replacing them with cryptographically signed tokens.

2.5.2 Azure Policy for Kubernetes: OPA Gatekeeper, Manifest Validation, Compliance Enforcement

Azure Policy is one of the key components for securing AKS because it allows policies to govern Kubernetes manifests just as they do for Azure resources defined through ARM or Bicep. To evaluate and enforce policies inside the cluster, administrators must enable the Azure Policy Add-on for Kubernetes.

This add-on installs an optimized version of Open Policy Agent (OPA) Gatekeeper, the CNCF-standard engine for policy enforcement and validation [28]. Microsoft provides native integration and management guidance for this component [29].

OPA Gatekeeper integrates with the Kubernetes admission control pipeline. When a resource is created or modified, Gatekeeper intercepts the request and checks whether it violates any constraint. AKS provides hundreds of built-in policies covering container privileges, security configurations such as seccomp, AppArmor, and Linux capabilities, image sources and registries, networking requirements, workload configuration, Pod Security Standards, and diagnostic and auditing requirements.

The combined integration of Azure Policy, OPA Gatekeeper, and Microsoft Defender for Cloud provides centralized compliance reporting across all clusters. This capability addresses Kubernetes' historical limitation of governance fragmentation by enabling consistent policy enforcement across both Azure resources and Kubernetes resources.

2.5.3 Microsoft Defender for Containers: Image Scanning, Runtime Protection, Anomaly Detection

Microsoft Defender for Containers is the cloud-native security service that enhances AKS protection at the image, workload, and node levels. It is part of Microsoft Defender for Cloud, which combines static analysis, image scanning, runtime monitoring, and threat detection [30].

Defender for Containers operates across multiple layers.

Image scanning The service scans container images stored in Azure Container Registry and other registries. It identifies vulnerabilities (CVEs), insecure configurations, outdated libraries, and other weaknesses. The results are integrated into Azure security dashboards and compliance reports.

Runtime protection Defender monitors processes running inside containers and detects suspicious behavior such as execution of unexpected commands, attempts to escalate privileges, creation of suspicious sockets, unauthorized filesystem modifications, fileless attack patterns, and actions correlated with MITRE ATT&CK techniques such as escape to host.

Cluster configuration analytics Which identify issues such as missing Network Policies, overly permissive RBAC roles, over-privileged service accounts, and insecure Pods.

Defender for Containers transforms Kubernetes security from a static configuration model into one that includes continuous runtime protection, aligning AKS with modern container runtime protection (CRP) practices.

2.5.4 Azure Key Vault, KMS, and Secure Key and Secret Management

Secret management is one of the most sensitive challenges in Kubernetes. As discussed in Section 1.4.4, Kubernetes Secrets are stored in `etcd` and may be exposed in plaintext without proper encryption. AKS addresses this challenge by integrating directly with Azure Key Vault and the Key Management Service (KMS).

Official documentation for 2024 describes how AKS supports KMS v2 to encrypt Secrets using keys stored in Key Vault [25]. KMS provides the ability to encrypt Secrets before they are written to `etcd`, use customer-managed keys stored in an isolated vault, and support hardware-backed models such as Managed HSM.

Beyond Kubernetes Secret encryption, Key Vault offers secure storage for TLS certificates, centralized management of disk encryption keys, automatic key rotation, and full auditing of key access.

For workloads, Key Vault can be integrated through Workload Identity, which enables Pods to obtain OIDC tokens, and through the CSI Secret Store Driver, which mounts secrets from Key Vault directly into Pods [31, 32].

2.5.5 Disk Management and Encryption (Host-Based, CMK, Managed Disks)

Protecting data at rest is a fundamental requirement in cloud security. Every AKS node uses Azure Managed Disks, which support several layers of encryption.

The first layer, always enabled, is server-side encryption with Microsoft-managed keys. This ensures that all VM disks are encrypted regardless of cluster configuration [33].

The second layer is customer-managed key (CMK) encryption, recommended for regulated environments. AKS supports specifying a Key Vault-managed CMK to encrypt both OS and data disks [34].

The third layer is host-based encryption, which encrypts temporary disks, disk caches, ephemeral volumes, and the container overlay filesystem. This is particularly important because temporary disks may contain sensitive non-persistent data, including swap files, cached credentials, network buffers, or application metadata.

AKS also supports advanced Azure Managed Disk security features, including Trusted Launch, Managed HSM integration, and double-layer encryption using both platform and customer keys.

2.6 State of the Art in Control Plane Security

The Kubernetes control plane is the core of the system and, in AKS, it is the only component fully managed by the cloud provider. It includes the API server, the etcd datastore, the scheduler, and the controller processes, all of which govern the declarative state of the cluster and the behavior of workloads.

Even though Microsoft operates the control plane in AKS, customers remain responsible for several important aspects, such as controlling access, defining isolation boundaries, enabling adequate audit logging, and ensuring that network and identity-based controls are aligned with zero trust principles. The following sections describe the key elements that define the current best practices for securing the control plane in AKS clusters.

2.6.1 Securing the kube-apiserver: public endpoint, private clusters, and authorized IP ranges

The `kube-apiserver` is the entry point for all administrative and application-level operations in Kubernetes. Every change to the cluster's state must pass through it, which makes it one of the most sensitive components of the platform. The exposure of the API server largely determines the level of risk associated with the control plane.

In AKS, the default configuration exposes the API server through a public HTTPS endpoint. This simplifies operational access but also increases the attack surface because the endpoint becomes reachable over the public internet. Microsoft explicitly classifies this as the option with the highest intrinsic risk and recommends limiting its exposure or adopting private endpoints.

The most secure option is the private cluster configuration, where the control plane endpoint does not receive a public IP address and can only be accessed through a virtual network, VPN, or ExpressRoute. This significantly reduces exposure to scanning, brute-force attempts, and token exfiltration attacks.

An intermediate option is the use of authorized IP ranges, which allow organizations to keep a public endpoint but restrict access to specific IP addresses or subnets. All non-authorized IPs are blocked by the Azure Load Balancer that fronts the API server [35].

From a modern security perspective, private clusters represent the gold standard. Authorized IP ranges are an acceptable compromise when a cluster cannot yet be integrated into a private network.

2.6.2 Trusted Access and internal communication through the Azure backbone

A major evolution in AKS control plane security is Trusted Access, a mechanism that allows Azure services to communicate with the `kube-apiserver` through the

internal Azure network without requiring public endpoints, authorized IP ranges, or customer-managed certificates.

Trusted Access enables services such as Azure Monitor, Microsoft Defender for Containers, Azure DevOps, and Azure Container Registry to interact with the cluster over authenticated and authorized channels based on Managed Identities.

This model provides two essential benefits: it removes the need to expose the control plane to external IP addresses, reducing the attack surface, and it eliminates the reliance on privileged static credentials because authentication is performed through managed identities with narrow scopes.

Communication occurs exclusively through the Microsoft Global Network, not over the public internet. This design aligns with zero trust principles, where privileged communication is based on identity, least privilege, and authenticated channels rather than network location.

2.6.3 Encryption of etcd and protection of sensitive data

`Etcd` is the key-value datastore that stores the entire cluster state, including configurations, ServiceAccount tokens, Kubernetes Secrets (unless protected by KMS), workload definitions, node mappings, and network configuration. Its highly sensitive nature is well documented in Kubernetes best practices and in NIST guidance such as NIST SP 800-204A. Compromising `etcd` would allow an attacker to retrieve Secrets or alter any cluster configuration.

In AKS, `etcd` security is enforced by Microsoft through encryption at rest, network isolation, automated backups, high-availability replication, and by blocking any form of direct customer access to `etcd`. The most important protection mechanism is the encryption of Secrets before they are written to `etcd`, provided through Azure Key Management Service (KMS), which allows the cluster to use keys stored in Azure Key Vault [25].

2.6.4 Logging, auditing, and diagnostic visibility of the control plane

A distributed system like Kubernetes requires full observability of the control plane for security, compliance, and operational governance. In AKS, observability is enabled through Diagnostic Settings, which allow control plane logs and metrics to be streamed to Azure Monitor Logs, Event Hub, or Azure Storage [36].

Control plane logs include `kube-apiserver` audit logs, controller-manager logs, scheduler logs, and authentication and authorization events. The audit log is especially important because it records every operation performed on the cluster, such as authentication attempts, manifest changes, RBAC modifications, and interactions between workloads and the API server. Kubernetes defines audit logging as a core requirement for cluster governance [37].

Azure also provides diagnostic insights that are not available in self-managed clusters, including control plane latencies, API throttling, internal failures, and Azure-specific data plane conditions.

Integration with Microsoft Defender for Cloud allows alert generation based on audit and diagnostic data, helping identify behaviors such as suspicious API enumeration, unauthorized access attempts, or modifications of sensitive cluster resources.

2.7 Node and Infrastructure Security

Node security is a fundamental part of the overall security posture of a Kubernetes cluster. Although the AKS control plane is fully managed by Microsoft, the data plane, which consists of the nodes running critical workloads, remains under the shared responsibility of the customer. Protecting the nodes requires a consistent set of measures that include OS patching, access control, kernel-level isolation, and the protection of disks and temporary storage.

2.7.1 Updates, patching, and OS image management

In AKS, nodes are implemented as Azure Virtual Machine Scale Set (VMSS) instances that run optimized images, typically based on Azure Linux or Ubuntu, or Windows for specific workloads. The security of the node depends largely on the timely application of OS, kernel, and container runtime updates.

Update immutability and upgrade models AKS does not apply in-place OS patches. Instead, it uses an immutable infrastructure model in which upgrades are performed by provisioning a new node with the updated OS image, draining the old node, reassigning Pods, and removing the outdated node. This approach eliminates configuration drift and ensures that every node adheres to the security baseline defined by Microsoft [38].

Node OS auto-upgrade From a security perspective, one of the most critical features is Node OS Auto-Upgrade, which automatically applies new OS image versions as soon as they are available. This significantly reduces the exposure window for zero-day vulnerabilities [39].

Container runtime hardening Node images include tested and hardened versions of the container runtime (containerd) and the kubelet. Continuous alignment of the runtime helps protect the cluster from vulnerabilities such as container escape exploits, cgroup driver vulnerabilities, and sandboxing issues. Microsoft publishes security advisories specific to node images to ensure timely updates.

2.7.2 Disabling SSH and Command Invoke

Administrative access to nodes is one of the main risk surfaces in any Kubernetes environment. In a mature cloud-native model, nodes should not be managed manually and should not be directly accessible, which aligns with the principle of node immutability.

SSH access to nodes In AKS, SSH access to nodes is disabled by default. If a user accidentally enables SSH, the policy "Azure Kubernetes Clusters should disable SSH" helps restore the secure configuration and ensure that no node exposes an SSH endpoint [40].

Removing SSH access mitigates risks such as brute-force attacks, accidental exposure of SSH keys, bypassing Kubernetes RBAC, and unauthorized modifications to the host OS.

Command Invoke Another potential attack vector is the Run Command / Command Invoke feature, which executes commands on nodes via the Azure management plane. Microsoft notes that, unless strictly necessary, this access may bypass cluster security controls and network restrictions [41]. The policy "AKS clusters should disable Command Invoke" allows organizations to block this entry point and enforce least-privilege practices.

2.7.3 Host isolation: seccomp, AppArmor, and kernel lockdown

Nodes form the boundary between containerized workloads and the underlying operating system. Securing them requires the use of Linux kernel isolation primitives, widely documented by the CNCF Cloud Native Security Whitepaper [23].

Seccomp Seccomp restricts the system calls available to containers, reducing the risk that an exploit inside a workload can execute unauthorized kernel-level operations. Since 2022, Kubernetes recommends the RuntimeDefault profile, which AKS fully supports [42]. AKS strengthens this approach through built-in audit and deny policies that block unsafe or unsupported profiles.

AppArmor AppArmor provides an additional layer of isolation by defining which files and system capabilities a container can access. It is natively supported on Linux nodes in AKS and is an important mechanism for mitigating lateral movement or data exfiltration by compromised processes [43].

Kernel lockdown and host hardening AKS node images include several additional host-level protections, such as kernel lockdown mode, restrictions on unsigned kernel modules, secure cgroups v2 configuration, and restrictions on exposure of node devices to containers. Together, these controls significantly reduce the risk of container escape and node compromise.

2.7.4 Temporary disks and host-based encryption

AKS nodes use local disks (OS disk, data disk, and temporary disk) for node operations, container images, logging, and caching. These disks can store sensitive short-lived information, making hardening essential.

Host-based encryption (HBE) AKS supports full-disk encryption of temporary disks and disk caches through Azure Host-Based Encryption. When enabled, all writes to local storage are encrypted before reaching the hardware with no operational impact.

Host-based encryption provides two main benefits: protection against unauthorized access in the event of improper disk snapshotting or recovery, and compliance with encryption requirements for sensitive data, even when the data is temporary in nature.

Managed disks and customer-managed keys In addition to host-based encryption, AKS nodes can use customer-managed keys (CMK) for OS and data disks. This allows organizations to control key management and rotation, an important capability for regulated environments [34].

Temporary disk and known risks The temporary disk of a node may contain container image layers, swap and journaling files, kubelet temporary data, and short-lived workload files. Host-based encryption is considered a best practice to mitigate risks associated with data recovery, improper snapshot creation, and memory scraping, aligning with guidance such as NIST SP 800-88.

2.8 Application Workload Security

Workload security is one of the most critical aspects of the overall security posture of a Kubernetes cluster. Even with a well-protected control plane and properly hardened nodes, a misconfigured workload can introduce severe vulnerabilities. Excessive privileges, untrusted images, missing probes, permissive network configurations, or improper use of credentials can turn a single compromised Pod into a serious breach affecting the entire cluster.

Application security in Kubernetes spans multiple dimensions, including supply chain integrity, container configuration, runtime behavior, networking, and identity management.

2.8.1 Container hardening: user, capabilities, mounts, filesystem, sysctl

The container is the closest security boundary to the application, making it one of the most important components to harden. Kubernetes defines minimum security requirements through the Pod Security Standards (PSS), which aim to reduce the container attack surface by removing all unnecessary privileges.

User and least-privilege execution (non-root) Running containers as root is one of the most common causes of compromise. Kubernetes and AKS recommend explicitly defining a non-privileged user through `securityContext.runAsUser` [44].

Root containers expand the attack surface for local privilege escalation, filesystem abuse, and exploitation of kernel primitives.

Linux capabilities Linux capabilities split root privileges into atomic components. Some capabilities, such as `CAP_SYS_ADMIN`, are effectively equivalent to full root privileges and should always be removed. CNCF guidelines recommend a drop-all add-minimal strategy [23].

Mounts and hostPath volumes Mounting the host filesystem through `hostPath` allows containers to access node-level files, which is one of the highest-risk configurations. The PSS baseline discourages hostPath, and the restricted profile almost completely prohibits it.

Read-only filesystem Setting `readOnlyRootFilesystem: true` helps protect against attacks that attempt to write malicious files inside the container's writable layer. Kubernetes recommends enforcing read-only filesystems and using temporary volumes only when necessary.

Sysctl settings Sysctl interfaces allow modification of kernel parameters. Kubernetes differentiates between safe and unsafe sysctls; unsafe sysctls are blocked because they can affect host-level behavior [45].

2.8.2 Supply chain security: image, registry, tags, signing, ImagePullSecrets

The container supply chain has become one of the most critical attack vectors in cloud-native systems. Compromising an image introduces vulnerabilities directly into the cluster.

Trusted images and authorized registries Workloads should pull images only from trusted registries such as Azure Container Registry (ACR), private registries, or verified official repositories.

Avoiding the latest tag The `latest` tag is mutable and breaks reproducibility. Kubernetes and Microsoft strongly discourage its use in production clusters.

Image signing Kubernetes supports Notation (via Notary v2) for image signing, and ACR integrates with content trust and Notation 1.0 to ensure image integrity.

ImagePullSecrets When a registry requires authentication, Kubernetes mounts credentials through ImagePullSecrets, preventing unauthorized registry access.

Vulnerability scanning Azure Container Registry and Microsoft Defender for Containers provide vulnerability scanning to detect emerging CVEs [30].

2.8.3 Runtime behavior: probes, anti-affinity, resilience as a security property

Resilience is not only a functional attribute but also a security property. Fragile workloads that cannot self-recover are more vulnerable to denial-of-service attacks and operational failures.

Liveness, readiness, and startup probes Probes allow the kubelet to detect and recover stuck or unresponsive applications. Workloads without probes cannot self-correct, weakening overall resilience [46].

Pod anti-affinity and topology spread constraints Distributing Pods across nodes improves resilience. References such as topology spread constraints and anti-affinity policies highlight their importance for workload survivability [47, 48].

Avoiding naked Pods Pods should be managed through controllers such as Deployments, StatefulSets, DaemonSets, and Jobs. Standalone Pods lack self-healing capabilities and introduce fragility.

2.8.4 Tokens and credentials: automountServiceAccount-Token and Workload Identity

Identity management is one of the most sensitive aspects of workload security.

Automatically mounted tokens Kubernetes automatically mounts a long-lived token for the default service account into every Pod. This token allows API authentication and may be accessible to attackers. Kubernetes recommends disabling `automountServiceAccountToken` unless necessary [49].

Workload Identity for AKS Workload Identity integrates Kubernetes service accounts with Microsoft Entra ID using OIDC. It provides short-lived, cryptographically signed tokens without static secrets, supporting Azure RBAC and secure access to Azure services [27].

2.8.5 Security of exposed services: port binding, host networking, ingress

The exposure surface of workloads is one of the main potential attack vectors.

Port binding and `hostPort` The `hostPort` directive exposes a port on the node directly, bypassing the container network namespace. This is considered risky and discouraged by the PSS restricted profile.

hostNetwork: true Enabling `hostNetwork` places the Pod inside the node's network stack and is only needed for specialized workloads. For application Pods, it is considered a major isolation risk [26].

Ingress and L7 security Ingress controllers manage HTTP and HTTPS traffic towards cluster services. Layer 7 security requires mandatory TLS, strict host rules, and careful path validation [50].

External IPs and CVE-2020-8554 CVE-2020-8554 demonstrated that attackers can abuse external IP assignments to intercept traffic. AKS implements policies restricting this behavior.

Internal Load Balancers Services using `LoadBalancer` should rely on internal load balancers unless public exposure is required [51].

2.9 Zero Trust Model Applied to AKS

The Zero Trust model is the dominant paradigm in modern cloud security. It is built on the principle "never trust, always verify". In Kubernetes, and especially in AKS, Zero Trust plays a central role because the cluster is a highly distributed, dynamic system composed of many layers of identities, networks, images, and artefacts. Every interface, workload, and service interaction becomes a potential attack vector, and the absence of a traditional network perimeter requires continuous verification of identity, privilege, network flows, and configuration.

This section explains how these principles translate into concrete controls within Azure Kubernetes Service.

2.9.1 Identity as the perimeter: Entra ID, Workload Identity, and OIDC tokens

In a Zero Trust model, identity replaces the network as the primary security boundary. In Kubernetes, this translates into three distinct layers of identity: human identities, cluster and infrastructure component identities, and application workload identities.

Human identities: Entra ID as the control plane authentication provider AKS supports native integration with Microsoft Entra ID, which provides federated authentication based on OAuth2 and OpenID Connect. Through Entra ID, organizations can enforce MFA, Conditional Access, and Identity Protection, delegate access through groups and roles, eliminate local cluster identities, and audit all control plane operations.

Cluster identities: Managed Identity Each AKS cluster uses a Managed Identity to interact with Azure resources, eliminating static secrets and reducing the risk of credential exposure [52].

Workload identities: Workload Identity and OIDC federation The most significant step towards Zero Trust in AKS is Workload Identity, which uses OIDC federation between Kubernetes and Microsoft Entra ID [27]. Each Pod can assume a dynamic, scoped identity using short-lived OIDC tokens that are renewed automatically and never stored on disk. This eliminates secrets inside Pods, long-lived service principals, and static JWT tokens. Workload Identity is the recommended Zero Trust mechanism for Pod-to-Azure authentication.

2.9.2 Least privilege and application segmentation

Zero Trust enforces a fundamental requirement: every entity must have only the privileges needed to perform its function, and nothing more.

Least privilege in Kubernetes RBAC The native RBAC model allows fine-grained segmentation of users, groups, and service accounts [22]. In AKS, this translates into using namespace-scoped roles instead of cluster-scoped roles, eliminating direct access to `cluster-admin`, creating dedicated service accounts for each workload, and applying role bindings that follow least privilege principles.

Application segmentation: service boundaries and east-west segmentation Segmentation in Kubernetes is implemented through application-level boundaries, internal Network Policies, and workload-level restrictions. Network Policies (Calico or Cilium) enable micro-segmentation by defining explicit communication rules between Pods [18].

2.9.3 Minimizing the attack surface of the cluster

Reducing the exposed surface is one of the pillars of the Zero Trust model. In AKS, this principle applies across several dimensions.

Reducing control plane exposure The control plane should not be accessible from the public internet unless necessary. AKS supports private clusters, authorized IP ranges, and Trusted Access for internal service communication [53].

Reducing node exposure Zero Trust requires eliminating direct host access. AKS provides mechanisms such as disabling SSH access, disabling Command Invoke, automatic node patching, and immutable OS image upgrades.

Reducing exposure of application services Application workloads should be exposed as little as possible. Best practices include using internal load balancers, avoiding NodePort, restricting `externalIPs` and configuring ingress controllers with mandatory TLS and explicit host definitions.

2.9.4 Zero Trust supply chain: controlled repositories and verified images

The supply chain is one of the primary threat vectors in Kubernetes environments. In a Zero Trust model, the provenance, integrity, and security of images must be continuously validated.

Controlled repositories Images should come only from trusted registries such as Azure Container Registry or verified enterprise repositories.

Signed images Container image signing with Notation and Notary v2 is supported, and Azure Container Registry integrates with content trust.

Automated scanners Microsoft Defender for Containers performs continuous scanning of images stored in registries and deployed workloads [30].

Image configuration validation A Zero Trust supply chain requires avoiding the `latest` tag, validating images through CI/CD pipelines, generating SBOMs, and enforcing dependency hygiene.

Chapter 3

Implementation High Level Design

3.1 Overall Architecture of the implementation

The infrastructure developed during this project represents the technical implementation of the AKS Security Framework (AKSSF). It is a system designed to centralize the publication of security policies for Kubernetes clusters and, at the same time, securely and at scale collect compliance snapshots from distributed tenants. The resulting architecture meets several converging goals: enabling centralized governance, maintaining precise control over the configurations applied to clusters, ensuring tenant isolation, and providing an observability layer capable of delivering an immediate, global view of the security posture across all managed environments. The architecture is organized around two main functional blocks: the **Tenant Client**, which is deployed inside each tenant hosting AKS clusters, and the **Tenant Server**, the central coordination environment that publishes security baselines, receives compliance evaluations, and exposes an analytical and multi-tenant view of the entire system.

3.1.1 Tenant Client: operational point for policy synchronization and enforcement

The left side of the architecture represents what is deployed inside tenant environments. Each tenant hosts two main subsystems: the **Audit and Sync** component, responsible for data exchange with the tenant server, and the **AKS and Azure Policy** subsystem, where the policy engine applies and evaluates configurations in real time on Kubernetes clusters.

The central component is an **Azure Function** that performs two complementary operations. On one side, it fetches the configurations published by the tenant server, downloading global baselines, tenant-specific custom baselines, and cluster-level configurations using a Private Endpoint connected to the policy storage account. On the other side, it collects the compliance snapshots generated by Azure Policy and Gatekeeper inside the clusters, processes them locally, and sends them

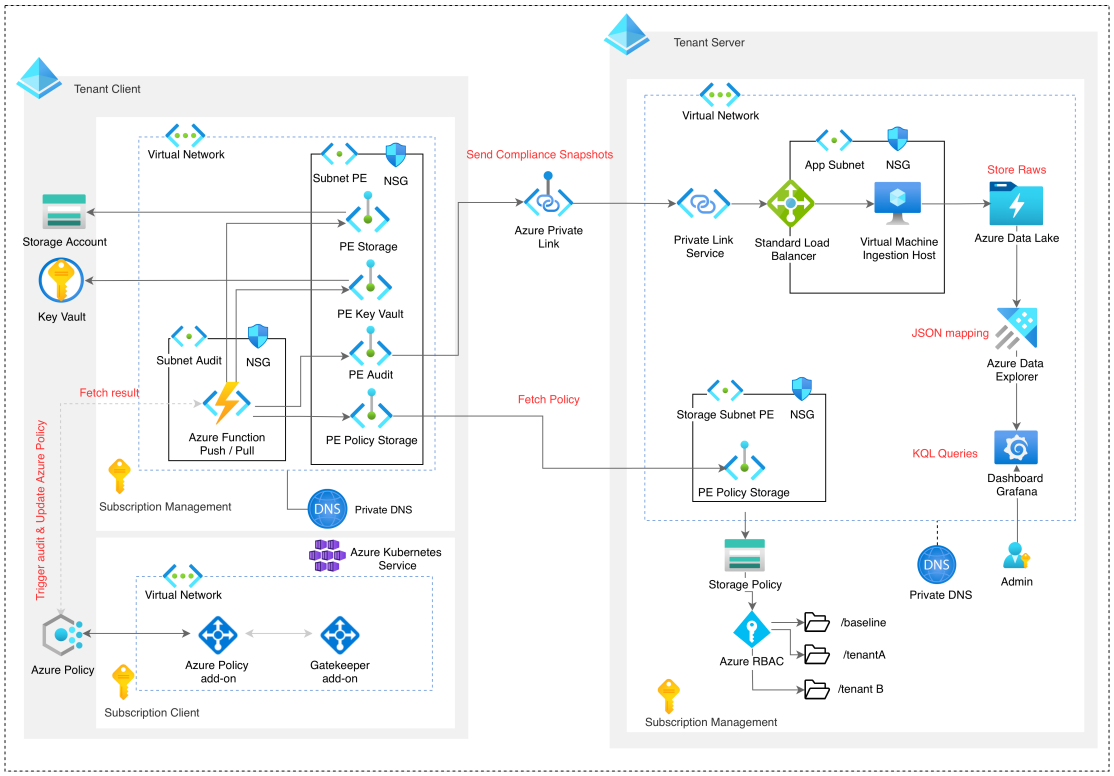

Figure 3.1. AKSSF High Level Design

to the tenant server through a private connection based on Private Link Service. The client environment is isolated through dedicated subnets (one for the audit function and one for the Private Endpoints), both protected by Network Security Groups, and uses a local Key Vault to store the credentials and identities required for cross-tenant communication.

Compliance evaluation is natively handled by the **Azure Policy add-on** and the **Gatekeeper add-on** in the AKS cluster, which produce the policy evaluation results. The client-side infrastructure ensures that each cluster receives the correct baseline, applies the prescribed configurations, and generates a uniform compliance report for the tenant server.

3.1.2 Tenant Server: central platform for publication, collection and observability

The right side of the architecture represents the central environment, which serves as both the *Policy Distribution Hub* and the *Collector and Analytics Engine*. This tenant operates as a governance domain: it does not interact directly with client clusters but instead publishes configurations, receives compliance snapshots, and exposes visualization and analysis capabilities.

The first core element is the **Policy Storage**, a storage account acting as a centralized repository for global baselines, tenant custom configurations, and cluster-specific definitions. The storage account is accessible through Private Endpoint and enforces granular access controls through Azure RBAC, ensuring that each tenant can access only its dedicated directory. This model supports the multi-tenant nature of the solution and enables efficient and secure distribution of desired configuration states.

The second core element is the **collection pipeline for compliance snapshots** sent by tenant clients. Each tenant sends its JSON snapshots through a Private Link Service exposed by the tenant server. Requests are routed to a **dedicated virtual machine** that functions as the Ingestion Host. This VM is isolated in an application subnet protected by an NSG and stores all incoming payloads in a second storage account that serves as a **Data Lake** for raw snapshots.

From this point onward, the analytical layer takes over. Using an ingestion connection between the Data Lake and **Azure Data Explorer (ADX)**, the snapshots are imported into the `snapshots_raw` table, expanded, and correlated through Update Policies to populate derived tables such as `snapshots`, `cluster`, `policy_results`, and `violations`. These tables form a highly indexed data model that supports real-time analysis and visualization. The system also builds Materialized Views to compute essential KPIs such as global compliance scores, time-based trends, per-tenant distribution, and the most frequently violated policies.

The final layer is the **visualization environment**, provided by a Grafana instance configured with a KQL data source pointing to ADX. Through thematic dashboards, administrators can view the overall multi-tenant security posture, analyze adherence to baselines, and track the evolution of compliance over time for each tenant or individual cluster. In this way, the tenant server becomes a complete platform for governance, data collection, and analytics, without ever interacting directly with tenant application resources.

3.1.3 Structure of the Following Chapters

Before addressing these components individually, the next chapter introduces the baseline defined specifically for this work. This baseline represents the reference model against which all AKS clusters are evaluated and constitutes the foundation on which the entire compliance validation process is built.

The following two chapters then analyze each block of the architecture in detail. The chapter dedicated to the Tenant Client explores the synchronization cycle, the application of policies through Azure Policy and Gatekeeper, and the management of Private Endpoints and identities. The subsequent chapter, dedicated to the Tenant Server, describes the organization of the centralized repository, the collection flow through the Private Link Service, the ingestion pipeline into Azure Data Explorer, and the construction of the multi-tenant dashboards. Together, these chapters provide a complete view of the AKSSF platform and the architectural decisions behind its design.

Chapter 4

Defining a security baseline for AKS using Azure Policy

4.1 Objectives and scope of the baseline

Defining a security baseline for AKS is a key element in building a reliable and manageable containerized infrastructure. In a distributed system like Kubernetes, which is highly elastic, dynamic and autonomous, the risk of misconfigurations or inconsistent setups is naturally high. The baseline acts as a minimal and shared set of controls that every cluster must follow in order to be considered secure according to criteria that are uniform, verifiable and easy to update.

The primary goal is not to enforce strict limitations, but to ensure that each cluster is automatically assessed against a consistent set of security expectations. This avoids situations where local team decisions or heterogeneous implementations could weaken the overall security posture.

4.1.1 Using the baseline in Audit mode as a tool to measure security posture

The entire baseline is initially applied in Audit mode, which is intended as an observation mechanism rather than an enforcement mechanism. The main goal is not to immediately block non-compliant configurations, but to obtain a complete and reliable picture of the actual security level reached by each cluster.

Audit mode allows the platform to:

1. **measure the current posture** before enforcing changes, avoiding operational disruptions;
2. **identify recurring patterns of risky configurations**, such as images without version tags, privileged containers or missing liveness probes;
3. **highlight differences between clusters**, helping to detect deviations from the baseline and to assess the maturity of each team;

4. **set intervention priorities** by identifying the most frequently violated policies or those associated with higher risks.

This approach fits well with the centralized governance model of the platform. Audit results are collected by the tenant server, stored in the data lake, analyzed through Azure Data Explorer and then shown in compliance dashboards. In this way, the baseline does not remain an abstract list of controls but becomes a continuous feedback mechanism that guides security decisions.

4.1.2 Scope

The baseline applies to all AKS clusters managed by the organization, regardless of the environment or deployment model. Although some controls may have a stronger impact on high-criticality environments, the baseline is designed to ensure a minimal level of uniformity that does not depend on the maturity of the team managing a cluster or on the specific characteristics of an application.

The document focuses exclusively on native AKS clusters and does not include external solutions such as Azure Arc-enabled Kubernetes, since they introduce their own policy set and a different architecture. The baseline also assumes the presence of some technical prerequisites. The first is the availability and enablement of the **Azure Policy Add-on for AKS**, which extends the Azure Policy engine to Kubernetes resources inside the cluster. It uses Open Policy Agent (OPA) and Gatekeeper to evaluate manifests and intercept requests to the kube-apiserver. Without this component, policies could be applied only to external Azure resources (such as the type `Microsoft.ContainerService/ManagedClusters`), but not to details of Pods, containers or other runtime objects.

4.2 Criteria for selecting built-in policies (principles, reference standards, Audit mode)

The baseline has been built by selecting only built-in policies provided directly by Microsoft, ensuring compatibility, updates and long-term support. The selection process followed several technical criteria:

Alignment with recognized security standards. Most of the policies included cover essential controls defined by the Kubernetes Pod Security Standards (PSS) and by the CIS Kubernetes Benchmark maintained by the Center for Internet Security. These standards provide a solid foundation for assessing workload security, especially through the "Baseline" and "Restricted" profiles.

Coverage of major risk vectors. Policy selection was guided by the main security domains relevant to a Kubernetes cluster: identity, networking, data protection, workload immutability, node isolation, container supply chain, service exposure and operational configurations that may affect resiliency.

Application neutrality and low operational risk. Since the baseline is initially applied only in Audit mode, each selected policy must be able to observe

and report potentially risky behaviors without modifying or blocking the operation of clusters. This mode supports a gradual assessment of risks and minimizes the possibility of application regressions.

Preference for policies that cover broad scenarios. At this stage, very specific policies (for example related to GitOps, backup or optional extensions) were excluded, because they are not universal requirements for all clusters and may not apply in heterogeneous environments.

Combining these criteria made it possible to create a baseline that is coherent, sufficiently comprehensive and supported by official guidelines and recommendations. The result is an effective tool to systematically measure and improve the security posture of the AKS infrastructure.

4.3 Logical structure of the baseline and security domains

To keep the baseline manageable, coherent and easy to consult, it is useful to represent it through a **map of security domains**, each associated with related policies. This structured approach helps to understand not only which policies to apply, but also why each one belongs to a specific domain, which risks it mitigates, and how it contributes to the overall security posture of the cluster.

4.3.1 Domain structure: governance, identity, network, data, workload

The baseline is organized into five main domains:

- **Governance and monitoring**: focuses on the ability to detect configurations, changes and anomalies, track compliance, and maintain centralized visibility over the entire cluster. It includes policies for enabling the policy add-on, collecting logs and diagnostics, and integrating with runtime security tools such as Microsoft Defender for Containers. This domain is essential because without visibility and auditing, it is not possible to ensure a consistent security posture.

- **Identity and access control**: covers authentication, authorization and identity management. This includes human and service identity access to the cluster, as well as workload identity. Policies in this domain involve integration with Microsoft Entra ID (formerly Azure AD), the use of RBAC, adoption of managed identities for the cluster, and the use of pod or workload identities to access external resources. Ensuring proper identity and minimal privileges is crucial to avoid escalation risks and unauthorized access.

- **Network and exposure surface**: focuses on network protection both inside the cluster and externally. This includes control plane access restrictions,

service exposure, the use of internal load balancers, limits on external IPs, disabling direct node access (SSH, Command Invoke), using HTTPS for external access, and reducing the overall network attack surface.

- **Data protection and storage**: concerns the protection of persistent and transient data for both the cluster and workloads. It includes disk encryption (OS, data, cache, temporary), the use of modern drivers such as CSI, protection of secrets and encryption of sensitive data, and secure management of encryption keys. This domain is especially important in enterprise or regulated environments.

- **Workload hardening and container supply chain**: relates to the security of containerized applications. It includes pod isolation, privilege restrictions, container image controls (trusted registries, avoiding the `latest` tag, using ImagePullSecrets), configuration of securityContext, seccomp, AppArmor, read-only root filesystems, restriction of host namespaces, CPU and memory limits, readiness and liveness probes, scheduling best practices (affinity and anti-affinity), and other measures to reduce the attack surface and improve resilience.

This structure makes it possible to **map each policy** to one or more domains, helping to clarify its purpose and simplifying the maintenance and evolution of the baseline.

4.3.2 Alignment with official guidelines, benchmarks and security standards

The baseline is not an ad-hoc construction. It is based on consolidated recommendations from authoritative sources and combines requirements from several references. In particular:

- The workload hardening domain closely follows the Pod Security Standards (PSS) maintained by the Kubernetes community. PSS define profiles (Baseline, Restricted, Privileged) that describe the minimum restrictions to apply to pods and containers so that the infrastructure can be considered secure. The Baseline profile prohibits known privilege escalations, while Restricted introduces stronger limitations [26].

- For the AKS ecosystem, the built-in policies provided by Azure Policy for Kubernetes allow many PSS recommendations to be translated into automated controls [54].

- The baseline is also inspired by the CIS Kubernetes Benchmark, a widely adopted standard in the cloud-native world[55]. Several of the policies included contribute to covering common requirements of this benchmark, such as RBAC management, privilege restrictions, pod isolation, network policies, auditing and logging.

- Finally, the official guidelines for Azure Kubernetes Service recommend practices aligned with the domains of this baseline [56]. These include control plane access protection, the use of a network CNI, secure storage, container hardening, monitoring and logging, and regular upgrades of nodes and clusters.

4.3.3 Application model: initiatives vs individual policies and systematic use of Audit mode

To make adoption of the baseline manageable and flexible, choosing how policies are grouped and applied (initiatives vs individual definitions) and selecting the appropriate enforcement mode are key decisions:

- The Azure Policy Add-on for AKS supports using built-in policies either as **single policy definitions** or as **initiatives** (collections of related policies). Initiatives allow activating consistent groups of policies through a single assignment. Documentation is available at: [57].

- In this baseline, and given the focus on an initial observation and assessment phase, all assigned policies are configured in **Audit** mode. This ensures that clusters continue to function without interruptions while non-compliant configurations are reported. This approach makes it possible to measure current posture, estimate remediation efforts, identify false positives or policies that may not apply, and plan a gradual transition to stricter enforcement where appropriate.

The idea is to make the baseline a transparent and safe-by-default tool. It does not force immediate changes but provides a compliance reporting framework that can later support policy enforcement through a gradual hardening process.

4.4 Governance, monitoring and posture management

A Kubernetes cluster cannot be considered secure if it is not observable. The ability to monitor, assess and govern cluster behavior centrally is a prerequisite for any credible security strategy. In AKS, the governance domain covers three main areas: the policy engine, control plane diagnostics and logging, and integration with threat detection tools such as Microsoft Defender for Containers. These elements form the foundation that allows an organization to identify risky configurations, track behavioral deviations and respond to incidents in a timely and documented way.

4.4.1 Azure Policy Add-on: prerequisite for controlling Kubernetes workloads

Azure Policy is the primary mechanism for defining requirements, verifying compliance and applying automated controls to Azure resources. However, in the case of AKS, many of the most critical configurations, such as container privileges, Pod Security Standards, dangerous capabilities, unsupported sysctl settings or volume configuration, exist inside the Kubernetes cluster and cannot be evaluated using policies that apply only to Azure Resource Manager objects.

To address this need, Microsoft provides the **Azure Policy Add-on for AKS**, an extension that brings policy evaluation directly into the Kubernetes control plane. The add-on is based on Open Policy Agent (OPA) Gatekeeper, which intercepts requests to the kube-apiserver, replicates policy definitions into ConstraintTemplates and generates a unified compliance event stream [58].

The baseline requires all clusters to have the add-on installed and properly configured. This requirement exists because without the add-on, many Kubernetes-focused policies cannot be evaluated. The baseline verifies this through the following policy:

- **Azure Policy Add-on for Kubernetes service (AKS) should be installed and enabled on your clusters** (Audit mode)

If an organization decides to move to enforcement, the add-on can be deployed automatically on clusters that do not have it by using this `DeployIfNotExists` policy:

- **Deploy Azure Policy Add-on to Azure Kubernetes Service clusters**

4.4.2 Cluster logs and diagnostics

A second pillar of governance is the availability of complete diagnostic logs. By default, AKS does not automatically send control plane logs to a centralized collection service. This means that without an explicit configuration, an organization has no access to kube-apiserver events or the information required to investigate incidents or anomalous behavior.

The baseline therefore includes a key control to ensure cluster logs are enabled:

- **Resource logs in Azure Kubernetes Service should be enabled**

If enforcement is required, AKS clusters can automatically receive a consistent diagnostic configuration using this `DeployIfNotExists` policy:

- **Configure diagnostic settings for Azure Kubernetes Service to Log Analytics workspace**

Log collection makes it possible to monitor:

- requests to the kube-apiserver (authentication, errors, unauthorized attempts)
- node and agent pool activity, together with internal cluster components
- network events, provisioning operations and system activity

In an enterprise environment, a lack of proper diagnostic settings represents a serious security gap: it prevents post-incident analysis, hides potentially malicious activity and makes it impossible to apply evidence-based security controls.

4.4.3 Integration with Microsoft Defender for Containers and its role in threat detection

The third element of the governance domain is integration with **Microsoft Defender for Containers**: [59].

This cloud-native security service continuously analyzes Kubernetes infrastructure to identify vulnerable images, risky configurations, suspicious runtime activity and anomalies in cluster behavior.

In AKS, Defender for Containers works through a Defender profile configured directly in the cluster. Enabling this profile allows deployment of the agent that collects security signals and supports:

- gathering security data from nodes, workloads and the network
- correlating known vulnerabilities (CVEs) in container images
- detecting anomalous behavior such as unexpected processes, filesystem changes or privilege escalation attempts
- providing security recommendations for non-compliant cluster configurations

The baseline includes a dedicated policy to ensure this profile is active:

- **Azure Kubernetes Service clusters should have Defender profile enabled**

If needed, consistent configuration of the profile can be enforced through automated assignments using the following policy:

- **Configure Azure Kubernetes Service clusters to enable Defender profile**

Integrating Defender for Containers into the baseline has strategic value. Kubernetes policies cover only static configuration analysis, while Defender adds a dynamic view of what actually happens at runtime. Combining these perspectives makes it possible to detect scenarios such as legitimate containers performing unusual actions, fileless attacks, lateral movement between pods, abuse of Linux capabilities or escalation attempts through processes or system calls.

4.5 Identity, authentication and access control

The security of a Kubernetes cluster is closely tied to the strength of the identity model that governs it. AKS integrates deeply with the Azure identity system, and an effective security baseline must ensure that every component, including human administrators, control services, internal agents and workloads running in the cluster, has the correct identity and only the minimum privileges needed to perform its tasks. Without a strong identity model, a cluster risks becoming an opaque system that is vulnerable to privilege abuse, accidental escalations or insufficient tracking of critical operations.

For this reason, the baseline dedicates an entire domain to identity and privilege management. It does so through a set of policies that govern user authentication, cluster identity and workload identity within the cluster.

4.5.1 Integration with Microsoft Entra ID and disabling local authentication methods

The most secure and scalable way to authenticate to an AKS cluster is through integration with **Microsoft Entra ID** [60]. This integration allows AKS to fully delegate user authentication to Entra ID and benefit from enterprise features such as MFA, Conditional Access, Identity Protection and centralized group management.

The baseline includes two policies that work together:

- **Azure Kubernetes Service Clusters should enable Microsoft Entra ID integration**
- **Azure Kubernetes Service Clusters should have local authentication methods disabled**

The first verifies that the cluster is configured with native Entra ID integration. The second ensures that unprotected local access methods, such as the automatically generated administrator account (`--admin`), are disabled. These local accounts are a significant risk because they bypass all enterprise identity controls.

Removing local identities ensures that all access occurs through verified and centrally managed identities. This enables full audit logging and significantly reduces the risk of anomalies or access that remains active after it should have been revoked.

4.5.2 Cluster identity: managed identities and their impact on operational security

Each AKS cluster needs an identity to interact with other Azure resources. This includes creating nodes, reading configurations, accessing Key Vault, managing load balancers or working with network interfaces. Historically, AKS relied on manually

managed service principals, which created operational complexity such as credential expiration, rotation requirements and accidental exposure of associated secrets.

To address these issues, the recommended identity model for AKS clusters is **Managed Identity**, a fully Azure-managed identity that does not require credentials, does not expire and does not need manual rotation. The baseline includes the following policy:

- **Azure Kubernetes Service Clusters should use managed identities**

This policy ensures that no cluster still uses legacy service principals. Moving to Managed Identity improves security by eliminating the accidental exposure of secrets, reducing operational overhead related to credential rotation and enabling the creation of granular identities for agent pools and specialized workloads.

Furthermore, adopting managed identities simplifies integration with Workload Identity, as it allows for a consistent identity approach across the entire cluster, from the control plane down to individual pods.

4.5.3 Workload identity: AKS Workload Identity and secure access to Azure resources

Workload identity management is a particularly sensitive area in AKS. Historically, pods accessed Azure resources through solutions such as Azure AD Pod Identity or, in worse cases, through static secrets mounted into containers. Both approaches are now deprecated or strongly discouraged.

The modern, secure and Microsoft-supported mechanism is **AKS Workload Identity**, which integrates Kubernetes Service Accounts with Microsoft Entra ID. It enables pods to obtain OIDC tokens validated by the cluster and use them to authenticate to Azure resources.

The baseline includes the following policy:

- **Azure Kubernetes Service Clusters should enable workload identity**

This control verifies that the cluster is configured with the required OIDC provider and that the feature is enabled.

The security benefits are significant. Pods no longer require secrets containing sensitive credentials, the identity is tightly bound to the Kubernetes Service Account, and access to Azure resources is fully governed through Entra ID. This makes it possible to apply the principle of least privilege consistently even to workloads.

4.5.4 RBAC hygiene: mandatory use of RBAC and limiting use of the `cluster-admin` role

Defining roles and permissions is the core of access control in Kubernetes. Role-Based Access Control (RBAC) allows fine-grained specification of which users or

service accounts can operate on which resources. Nevertheless, clusters that evolve without strong governance often accumulate overly permissive roles or inherited bindings, frequently including insecure patterns such as widespread use of the `cluster-admin` role.

The baseline includes two relevant policies:

- **Role-Based Access Control (RBAC) should be used on Kubernetes Services**
- **Kubernetes clusters should ensure that the cluster-admin role is only used where required**

The first ensures that RBAC is enabled at the cluster level, which is a necessary condition for any credible security model. The second highlights a common weakness: direct bindings to `cluster-admin`, often created for convenience during early development phases but becoming a major risk over time, especially in shared environments.

Kubernetes documentation repeatedly stresses the importance of avoiding global roles and recommends using namespace-scoped roles and strict least-privilege practices: [61].

An effective baseline must make these cases visible and give administrators the information needed to correct them before they develop into operational vulnerabilities or attack vectors.

4.6 Reducing the exposure surface and perimeter security

One of the core principles of cloud-native security is minimizing the attack surface. In a Kubernetes cluster, this principle applies both to the control plane and to the nodes and services exposed externally. Reducing the exposure surface does not directly affect workload functionality, but it plays a crucial role in preventing targeted attacks, lateral movement or unauthorized access to critical cluster components.

The baseline includes a set of controls that share a common purpose: ensuring that every AKS cluster is protected from unnecessary, unmanaged or excessive access at both the infrastructure and application level.

4.6.1 Control plane protection: private clusters and authorized IP ranges

The control plane is the heart of an AKS cluster. It hosts the kube-apiserver, which handles authentication, authorization and all administrative operations. By default, the control plane can be exposed through public endpoints protected by logical firewalls. However, direct exposure to the internet increases the attack surface and opens the door to scanning attempts or brute-force attacks.

To mitigate these risks, AKS provides the **private cluster** mode, which makes the control plane endpoint accessible only through the customer's VNet: [62].

The baseline includes the following policy:

- **Azure Kubernetes Service Private Clusters should be enabled**

Its purpose is to verify that all clusters are created in private mode, completely removing public exposure.

In cases where the endpoint must remain public and an exception is required (for example, sandbox clusters or certain automation scenarios), the baseline still requires the following policy:

- **Authorized IP ranges should be defined on Kubernetes Services**

This policy ensures that access to the public control plane is restricted to a strictly controlled list of authorized IP addresses. The use of authorized IP ranges is an essential hardening measure in all non-private scenarios.

4.6.2 Node hardening: disabling SSH and Command Invoke

A second critical area is direct access to agent nodes. Although AKS nodes are virtual machines owned by the customer, there is no operational need to access them through SSH or remote commands. AKS is designed to be managed at the orchestrator level, not through manual interventions on nodes, which can introduce undocumented changes, configuration drift or unexpected vulnerabilities.

For this reason, the baseline includes this policy:

- **Azure Kubernetes Clusters should disable SSH**

The second potential access path is **Command Invoke**, a feature that allows commands to be executed on nodes through ARM APIs. While useful in emergencies, Command Invoke represents a possible way to bypass network controls or RBAC permissions and to introduce changes on nodes that are not tracked at the Kubernetes level.

The baseline therefore includes:

- **Azure Kubernetes Service Clusters should disable Command Invoke**

This control verifies that the feature is disabled, reducing the possibility of external tools or automated processes interacting with nodes in unintended or untracked ways. The goal is to reinforce node isolation: the fewer direct access routes exist, the lower the chance they can be abused.

4.6.3 Secure exposure of services: HTTPS, internal load balancers and allowed external IPs

The third layer of surface-reduction focuses on services exposed by workloads. Even if the control plane is protected, a cluster can still be vulnerable if workloads expose unnecessary ports, unencrypted services or publicly reachable endpoints.

The first concerns the exclusive use of HTTPS for exposed services. The policy:

- **Kubernetes clusters should be accessible only over HTTPS**

ensures that services exposed through Ingress or LoadBalancer do not use insecure protocols.

The second measure concerns the adoption of **internal load balancers**, recommended whenever a service is intended to remain within the corporate network. The policy:

- **Kubernetes clusters should use internal load balancers**

verifies that services do not use public load balancers, reducing direct internet exposure.

Finally, the baseline includes a control for external IPs:

- **Kubernetes cluster services should only use allowed external IPs**

This ensures that any external IPs (the `externalIPs` field of a Service) come from an authorized list. This requirement was introduced following the disclosure of **CVE-2020-8554**: [63], which showed that an attacker could exploit arbitrary external IPs to intercept traffic intended for other services. Enforcing this policy is therefore a recommended preventive measure to mitigate potential application-traffic hijacking.

4.7 Data and storage protection

Data protection is one of the fundamental pillars of security in any cloud infrastructure. In Kubernetes, data exists in several forms: persistent workload data, application secrets, cluster state information stored in etcd and temporary data produced during application execution. The AKS security baseline includes a set of controls designed to ensure that all these components are properly protected, in line with the security requirements of enterprise environments.

4.7.1 Node and disk encryption with customer-managed keys (CMK)

AKS agent nodes are virtual machines and therefore include an operating system disk and optional attached data disks. By default, Azure managed disks are already encrypted with service-managed keys. However, in regulated or high-sensitivity environments, it is often necessary to use **customer-managed keys (CMK)**, which provide greater control over encryption, key rotation and key revocation.

The baseline includes the following policy:

- **Both operating systems and data disks in Azure Kubernetes Service clusters should be encrypted by customer-managed keys**

This control verifies that both OS and data disks are encrypted with CMK. This safeguard is especially important in environments subject to regulations such as GDPR, ISO 27001 or sector-specific standards that require the ability to revoke data access by revoking encryption keys.

4.7.2 Host-level encryption for temporary disks and cache

In addition to OS and data disks, AKS nodes use temporary disks and local cache, which are hosted directly on the physical hardware or on the VM. These volumes may contain temporary filesystem data, container overlays, network spool files, non-persistent credentials, swap files and other elements that might include sensitive information.

For this reason, Azure provides **host-based encryption**, a feature that automatically encrypts all temporary disks and cache on the node before they are written to physical storage.

The baseline includes the following policy:

- **Temp disks and cache for agent node pools in Azure Kubernetes Service clusters should be encrypted at host**

This control ensures that nodes are configured to avoid any unencrypted writes to physical disks. It is essential in scenarios involving potential hardware compromise, unauthorized access to storage devices or hypervisor-level attacks. It also mitigates risks related to data remanence by preventing sensitive information from persisting on temporary storage outside cluster control.

4.7.3 Protecting Secrets and etcd through Key Management Service (KMS)

Kubernetes Secrets contain credentials, access tokens, private keys and other highly sensitive information. Although Kubernetes supports optional Secret encryption at

the etcd level, this feature is not active by default in many upstream versions and requires explicit configuration by the cluster administrator.

In AKS, Secret protection is achieved through **etcd encryption** managed by the **Azure Key Management Service (KMS)**. This integration encrypts Secrets before they are stored in etcd, using a key stored in Azure Key Vault and protected through industry-standard security models.

The baseline includes the following policy:

- **Azure Kubernetes Clusters should enable Key Management Service (KMS)**

Enabling KMS ensures that no Secret is ever stored in plaintext in the etcd database, one of the most critical components of the cluster. Combined with CMK and managed identities, this creates an end-to-end security chain for sensitive information processed by the control plane.

4.7.4 Adopting CSI drivers as a prerequisite for advanced security scenarios

The Kubernetes storage provisioning model has evolved significantly in recent years. Traditional *in-tree* storage drivers were built directly into the Kubernetes codebase, which limited innovation and made it harder to introduce new security features. To address these issues, Kubernetes and Microsoft adopted the **Container Storage Interface (CSI)** model, which decouples storage drivers from the core Kubernetes release cycle and allows independent feature updates.

Microsoft documentation explicitly recommends using CSI for AKS, explaining that CSI drivers enable modern capabilities such as advanced encryption, integration with Key Vault and support for new protocols and storage backends: [64].

The baseline includes two controls:

- **Azure Kubernetes Clusters should enable Container Storage Interface (CSI)**
- **Kubernetes clusters should use Container Storage Interface (CSI) driver StorageClass**

These policies ensure that the cluster uses only CSI-based StorageClasses and does not rely on legacy in-tree drivers. Without this guarantee, advanced security features such as per-volume encryption, integration with external key management systems or fine-grained storage operations would not be possible.

4.8 Workload hardening: Pod Security and container configuration

The application layer is one of the most exposed areas in a Kubernetes cluster. Even when the control plane is protected and the nodes are properly configured, running insecure workloads can compromise the entire environment. Vulnerable images, permissive Pod configurations, excessive privileges or insufficient isolation from the node are all major attack surfaces in cloud-native systems.

For this reason, many of the most important policies in the baseline focus directly on Pod and container configuration. Microsoft consolidates these measures within the Pod Security Standards (PSS) initiatives and through extensions implemented using the Azure Policy Add-on. The goal is to ensure that every workload complies, at minimum, with the requirements of the PSS baseline level and avoids unsafe practices such as elevated privileges, node access or unsafe sysctl configurations.

4.8.1 "Pod security baseline standards for Linux-based workloads" initiative

The primary reference for workload security in Kubernetes is the **Pod Security Standards**, an official Kubernetes community specification that defines three security levels: Privileged, Baseline and Restricted. The Baseline level represents the minimum acceptable configuration for multi-tenant environments because it prevents the use of high-risk primitives while maintaining broad compatibility with most applications.

Microsoft incorporates these requirements into an Azure Policy initiative:

- **Kubernetes cluster pod security baseline standards for Linux-based workloads**

This initiative automatically verifies that Pods follow the fundamental constraints of the Baseline level. For example, it prevents the use of privileged containers, host namespaces or dangerous hostPath mounts.

Including this initiative in the baseline ensures broad and standardized coverage, directly aligned with Kubernetes community guidelines. Other sections of this chapter analyze the individual controls in detail, while here the initiative is introduced as a coherent collection of the most relevant aspects of the PSS model.

4.8.2 Privileges and capabilities: privileged mode, privilege escalation, CAP_SYS_ADMIN and disallowed capabilities

Privilege management is one of the most critical elements of workload hardening. Kubernetes allows containers to run with capabilities similar to root on the host

through settings such as `privileged: true`, enabling `allowPrivilegeEscalation` or assigning Linux capabilities. These configurations can jeopardize the security of the entire cluster.

The baseline includes several policies that address these risks. The first and most restrictive is:

- **Kubernetes cluster should not allow privileged containers**

which identifies all Pods configured as privileged. Privileged containers can access host devices, manipulate namespaces and load kernel modules, all of which are incompatible with a secure cluster environment.

A second relevant policy is:

- **Kubernetes clusters should not allow container privilege escalation**

which prevents containers from increasing their privileges, protecting against scenarios where a vulnerable process exploits setuid or setgid binaries to run actions as root.

The baseline also includes two more specific controls:

- **Kubernetes clusters should not grant CAP_SYS_ADMIN security capabilities**
- **Kubernetes clusters should not use specific security capabilities**

CAP_SYS_ADMIN is often described as the capability equivalent to root because it enables a wide range of kernel-level operations. Its use is discouraged by both Microsoft and the Kubernetes community. The second policy helps identify and prevent the use of other high-risk capabilities such as `CAP_NET_RAW` or `CAP_SYS_MODULE`.

4.8.3 Node isolation: host namespaces, seccomp, AppArmor and isolation primitives

Workload isolation is another cornerstone of Kubernetes security. Without proper isolation, a container may interact with the node or with other containers, gaining access to sensitive information or manipulating shared resources. The baseline includes several policies that address this challenge from different angles.

One of the most important policies is:

- **Kubernetes cluster containers should not share host namespaces**

which checks for the absence of configurations such as `hostPID`, `hostIPC` and `hostNetwork`. Sharing these namespaces allows containers to observe host processes, intercept traffic or manipulate low-level resources.

Another key element is syscall-level security using **seccomp**. The policy:

- **Kubernetes cluster containers should only use allowed seccomp profiles**

ensures that containers use secure profiles, typically `RuntimeDefault` or restrictive custom profiles. Seccomp provides strong barriers against unauthorized or unexpected syscalls, which are often used in kernel exploitation. Documentation: [65].

In addition to seccomp, the baseline includes:

- **Kubernetes cluster containers should only use allowed AppArmor profiles**

AppArmor defines fine-grained rules governing the behavior of a container, such as which files it can read or which binaries it can execute. Microsoft supports AppArmor in AKS through a Linux node extension. Documentation: [66].

The initiative also covers other isolation aspects, such as the use of hostPath volumes, direct access to host ports or operations involving FlexVolume drivers.

The overarching goal is to ensure that no container uses configurations that place it too close to the host, preserving a strong separation between workloads and infrastructure.

4.8.4 File systems and sysctl: read-only root filesystems and allowed sysctl interfaces

Beyond privileges and isolation, another important hardening area concerns filesystem behavior and kernel settings accessible from containers. A widely recognized best practice is enforcing a **read-only root filesystem**, which prevents attackers from modifying binaries or introducing persistent malware in the writable layers of a container.

The baseline includes the following policy:

- **Kubernetes cluster containers should run with a read only root file system**

which identifies containers configured with unnecessarily writable filesystems. This measure significantly reduces the attack surface, especially if an application is compromised through injection attacks or remote vulnerabilities.

Another critical area concerns **sysctl** settings, which allow modification of kernel parameters. Kubernetes allows certain safe sysctls but blocks unsafe ones by default. The following baseline policies address this:

- **Kubernetes cluster containers should not use forbidden sysctl interfaces**
- **Kubernetes cluster containers should use only allowed sysctl interfaces**

Controlling sysctl usage prevents containers from alterin

4.9 Workload hardening: supply chain, application networking and reliability

Workload security in Kubernetes cannot be considered complete without addressing the processes that occur before a container runs (the supply chain), the way application services are exposed and the application's ability to tolerate failures or anomalies. Even in a well-configured cluster, a compromised container, an unverified image or a fragile workload can become an attack vector or cause critical outages.

For this reason, the baseline includes policies that act on the image lifecycle, the management of internal credentials, the security of workload networking components and the reliability of distributed applications.

4.9.1 Container image controls: allowed registries, prohibition of `latest`, use of ImagePullSecrets

The container supply chain is one of the main weakness points in Kubernetes. Using images from unverified registries, relying on non-versioned tags or omitting authentication when pulling images can lead to accidental deployment of malicious, vulnerable or untraceable images.

The baseline includes three fundamental policies to mitigate these risks.

The first is:

- **Kubernetes cluster containers should only use allowed images**

which verifies that container images come only from approved registries. Defining an authorized set of registries prevents the accidental use of images that have not undergone quality or security checks.

A second policy targets the widespread but risky practice of using the `latest` tag. This tag does not guarantee stability, because its meaning can change at any time, making version pinning impossible and reducing reproducibility. The policy:

- **Kubernetes cluster container images should not include latest image tag**

identifies all containers that rely on `latest` and highlights one of the biggest supply-chain risks in container environments.

The third policy is:

- **Kubernetes cluster containers should only pull images when image pull secrets are present**

which ensures that ImagePullSecrets are configured when images reside in a private registry. This reduces the risk of unauthorized registry access, avoids authentication-related service disruptions and ensures that pull credentials are centrally managed.

4.9.2 Protection of internal credentials: managing `automountServiceAc`

Another frequently overlooked aspect of workload security is the handling of the tokens associated with Kubernetes ServiceAccounts. By default, Kubernetes automatically mounts a JWT token inside each Pod, allowing the workload to authenticate to the kube-apiserver. In most applications, this token is unnecessary. Leaving it accessible creates a potential vulnerability, because an RCE exploit or a compromised library could allow an attacker to use that token to interact with the API server.

The baseline includes a policy specifically designed to address this issue:

- **Kubernetes clusters should disable automounting API credentials**

which verifies that Pod specifications do not explicitly enable mounting the ServiceAccount token when it is not required. The recommended behavior is setting `automountServiceAccountToken: false` for all Pods that do not need direct API access.

This measure is especially important in environments using Workload Identity, as it ensures that the OIDC token becomes the only identity mechanism for workloads that require access to Azure resources, removing a significant attack surface.

4.9.3 Services and ingress: allowed ports and required configuration parameters

Workload security also depends on how applications are exposed through Kubernetes Services or Ingress resources. Unnecessary open ports, Services with unauthorized external IPs or incomplete ingress rules can allow unintended interactions with the cluster or its workloads.

The baseline includes three policies for this area.

The first is:

- **Kubernetes cluster services should listen only on allowed ports**

which restricts the TCP/UDP ports that Services can expose. This prevents the creation of Services listening on unusual ports that may be used by malware, unsupported protocols or insecure legacy components.

The second policy is:

- **Kubernetes cluster pods should only use approved host network and port list**

which ensures that Pods do not access host ports or use `hostNetwork: true`, a highly risky practice not suitable for multi-tenant environments.

Finally, the baseline includes:

- **Kubernetes cluster services should use unique selectors**

which identifies configurations where multiple Services share the same selector, causing ambiguous traffic routing and unintended exposure.

4.9.4 Reliability as a security requirement: liveness/readiness probes, anti-affinity, topology spread and prohibition of naked pods

Operational resilience is not only a functional requirement; it is also a security requirement. Fragile workloads that are easy to saturate or lack self-healing mechanisms become vulnerable points that attackers can exploit to cause denial-of-service or systemic instability.

The baseline includes several controls designed to improve workload availability.

The first is:

- **Ensure cluster containers have readiness or liveness probes configured**

which verifies that containers expose explicit health signals. These probes allow the kubelet to detect blocked, unresponsive or malfunctioning applications and ensure automatic Pod restarts or removal from load balancing.

A second control focuses on how workloads are distributed across the cluster:

- **Must Have Anti Affinity Rules or Topology Spread Constraints Set**

which identifies Pods that do not define anti-affinity rules or spread constraints. These rules prevent critical Pods from being scheduled on the same node, reducing the risk of service loss in case of node failure.

Another essential policy is:

- **Kubernetes cluster should not use naked pods**

which identifies Pods not managed by a controller such as Deployment, StatefulSet or DaemonSet. Naked Pods are not automatically recreated after failure and are therefore unreliable and incompatible with AKS orchestration.

Finally, the baseline ensures that services do not conflict with each other and do not expose unexpected behaviors by requiring that:

- ingress resources define an explicit host
- Services do not use unauthorized external IPs
- traffic routing remains deterministic

4.10 "Out-of-baseline" policies

An effective security baseline is not simply an exhaustive list of every available policy. It is the result of a careful and reasoned selection process. Including every possible control would make cluster configuration overly rigid, poorly aligned with real operational scenarios and, in some cases, even counterproductive. For this reason, some policies were intentionally excluded from the minimum baseline, even though they can offer higher levels of protection.

This section explains the reasons for these exclusions and discusses the trade-offs involved.

4.10.1 Policies deliberately excluded from the minimum baseline

Several controls were excluded not due to lack of value but because they are not appropriate for a baseline intended to be applicable across a wide range of organizations and scenarios.

1. Pod Security Standards – Restricted level

The initiative *Kubernetes cluster pod security restricted standards for Linux-based workloads* represents the highest and most strict level of the PSS model. This level requires extremely restrictive configurations (nearly complete prohibition of capabilities, mounts, privileges and node access) and is often incompatible with existing applications. Applying the Restricted level indiscriminately may break workloads, especially legacy applications or those not designed for native zero-trust models.

Reason for exclusion: the baseline must guarantee high compatibility and offer a progressive adoption path. The Restricted level remains recommended for sensitive workloads, but cannot be imposed as a universal requirement.

2. Image Integrity and Notation signature verification

Some policies — *Use Image Integrity to ensure only trusted images are deployed* and *[Image Integrity] Kubernetes clusters should only use images signed by notation* — require the Image Integrity add-on and a fully operational image-signing infrastructure.

Reason for exclusion: the Image Integrity add-on implies a mature supply chain and a robust enterprise-level signing pipeline. Many organizations are not yet prepared to manage such complexity.

These policies are ideal in regulated environments or in advanced DevSecOps contexts, but they cannot be required as part of a general-purpose baseline.

3. Azure Backup Extension for AKS

Policies related to *Azure Backup for AKS* verify the presence of the backup extension and the workload protection infrastructure.

Reason for exclusion: AKS backup strategy is an architectural and organizational decision that depends on the deployment model, workload characteristics

and operational continuity requirements. A security baseline cannot impose a specific backup approach, especially when many workloads are CI/CD managed and designed to be recreated.

4. Forced GitOps policies (Flux v2 and Source Control Configuration)

Azure Policy includes several controls designed to enforce GitOps (Flux v2), such as automatic deployment of the Flux extension, Git source definitions and authentication methods.

Reason for exclusion: GitOps is an architectural choice, not a minimum security requirement. Enforcing GitOps through policy may reduce application team flexibility and introduce unwanted operational dependencies. These controls belong to automation or governance strategy, not to the technical security baseline.

4.10.2 Controls dependent on the application model: Network Policy, advanced RBAC and tenant-specific configurations

There is a set of controls that are essential for many Kubernetes clusters but cannot be included in the minimum baseline because they require customization that varies significantly between organizations. The infrastructure described in this thesis, based on the tenant server and centralized governance through Azure Policy, enables the use of these controls but does not enforce them.

The most notable categories are **Network Policies** and **advanced RBAC configurations**.

Network Policy

Kubernetes Network Policies explicitly define which Pods can communicate with each other or with external services. They are critical for workload isolation, intra-cluster zero-trust models and prevention of lateral movement.

However, their effectiveness depends entirely on workload behavior. Each application has different communication patterns, internal services with specific needs and external dependencies restricted to certain Pods.

For this reason, the baseline cannot include Azure policies that require NetworkPolicies or that validate their content. Doing so would lead to two issues:

- it would impose restrictions that are not compatible with legacy workloads or complex microservice architectures
- it would require deep knowledge of application dependencies, which cannot be generalized at baseline level

The governance infrastructure, however, allows customer security teams to **add custom controls** using Gatekeeper or CustomPolicy, leveraging the same Azure Policy Add-on mechanisms.

Advanced RBAC and custom permissions

Similarly, many organizations require RBAC models that are far more granular than those covered by the baseline's generic policies. These include custom roles for teams, multi-namespace access hierarchies or tightly scoped permissions.

The baseline includes only two general controls:

- mandatory use of RBAC
- limiting the use of the `cluster-admin` role

However, it cannot include controls such as:

- definitions of custom roles
- restrictions on specific Kubernetes APIs
- limits on permissions per namespace or per group
- enforcement of particular multi-tenancy access models

The reason is straightforward: each customer has a different organizational structure, release process and maturity level in privilege management. Adding generalized RBAC rules would make the baseline too rigid and potentially incompatible with internal workflows.

The platform, however, enables teams within each tenant to **apply tenant-specific RBAC policies** through the Azure Policy Add-on, CustomPolicy and Gatekeeper ConstraintTemplates, while retaining centralized visibility through the tenant server.

Tenant-specific advanced configurations

Beyond Network Policies and RBAC, other aspects are entirely dependent on workload context:

- use of dedicated ServiceAccounts for specific microservices
- enforcement of CPU and memory limits per namespace
- compliance-driven labeling and annotation requirements
- restrictions on volume types or StorageClasses for sensitive workloads
- application-specific Ingress rules

These configurations cannot be imposed by the baseline because they require contextual knowledge the centralized platform does not possess. However, the implemented model **enables** enforcement of such rules through custom policies, preserving a balance between central governance and team autonomy.

Role of the infrastructure in supporting these customizations

The strength of the proposed solution lies in its ability to provide a **uniform security core**, through the baseline, while allowing tenants to extend policies as needed.

The tenant server, centralized data lake and configuration collector described in the following chapters provide full visibility over custom policies. This allows cluster owners to:

- apply advanced policies while remaining compatible with the baseline
- maintain control and observability even when clusters belong to different teams
- build a security model that is both scalable and aligned with workload requirements

This balance is the key reason these controls were excluded from the baseline: not because they are less important, but because they are **inherently specific**, and their effectiveness depends on the platform's ability to support and monitor them rather than on imposing them globally.

4.11 Summary table of the security baseline

The definition of the baseline concludes with a unified view of all selected policies. After discussing the security domains, the rationale behind the choices and the intentional exclusions, this section provides a compact representation of the complete set of controls considered essential to ensure a minimum security level for every AKS cluster.

The following table allows the baseline to be used as an operational reference for:

- assigning the baseline through Azure Policy initiatives
- verifying domain coverage and alignment with Kubernetes security standards
- simplifying maintenance and updates over time
- supporting auditing, onboarding of new clusters and posture validation

All listed policies are intended to be applied in **Audit** mode to maximize visibility and minimize operational impact during initial adoption.

Note: the table is organized by domain, with a short technical description and essential information for operational adoption.

4.11.1 Domain 1 — Governance, monitoring and posture management

Policy name	Version	Effect	Short description
Azure Policy Add-on for Kubernetes should be installed	1.0.2	Audit	Verifies that the add-on is enabled to evaluate cluster workloads.
Resource logs in Azure Kubernetes Service should be enabled	1.0.0	Audit	Ensures cluster logs are collected.
AKS clusters should have Defender profile enabled	2.0.1	Audit	Checks that Microsoft Defender for Containers is active.

4.11.2 Domain 2 — Identity, authentication and access control

Policy name	Version	Effect	Short description
AKS should enable Microsoft Entra ID integration	1.0.2	Audit	Verifies Entra ID integration for authentication.
AKS should disable local authentication methods	1.0.1	Audit	Ensures local admin accounts are disabled.
AKS should use managed identities	1.0.1	Audit	Requires use of managed identities for the cluster.
AKS should enable workload identity	1.0.0	Audit	Enables secure Azure resource access for Pods.
RBAC should be used on Kubernetes	1.1.0	Audit	Requires RBAC to be enabled.
Cluster-admin role should be used only when required	1.1.0	Audit	Detects misuse of the administrator role.

4.11.3 Domain 3 — Reduction of the attack surface

Policy name	Version	Effect	Short description
AKS Private Clusters should be enabled	1.0.1	Audit	Ensures the control plane is private.
Authorized IP ranges should be defined	2.0.1	Audit	Restricts API server access to allowed IPs.
AKS should disable SSH	1.0.0	Audit	Prevents direct node access via SSH.
AKS should disable Command Invoke	1.0.1	Audit	Disables remote command execution on nodes.
Kubernetes should be accessible only over HTTPS	8.2.0	Audit	Enforces secure connections.

AKS clusters should use internal load balancers	8.2.0	Audit	Requires internal load balancers for private services.
Kubernetes services should use only allowed external IPs	5.2.0	Audit	Restricts external IP usage.

4.11.4 Domain 4 — Data and storage protection

Policy name	Version	Effect	Description
OS and data disks should be encrypted by CMK	1.0.1	Audit	Ensures disks are encrypted with customer-managed keys.
Temp disks should be encrypted at host	1.0.1	Audit	Verifies host-based encryption for temporary disks.
AKS should enable KMS encryption	1.1.0	Audit	Enables Secret encryption through KMS.
AKS should enable CSI	1.0.0	Audit	Requires use of CSI drivers.
Kubernetes should use CSI StorageClass	2.3.0	Audit	Ensures StorageClass does not use deprecated drivers.

4.11.5 Domain 5 — Workload hardening (Pod Security, isolation, permissions)

Policy	Version	Effect	Description
Pod security baseline standards	1.4.0	Audit	Applies PSS Baseline level.
Not allow privileged containers	9.2.0	Audit	Detects privileged containers.
Not allow privilege escalation	8.0.0	Audit	Blocks use of allowPrivilegeEscalation.
Not grant CAP_SYS_ADMIN	5.1.0	Audit	Blocks the critical CAP_SYS_ADMIN capability.
Not use specific capabilities	5.2.0	Audit	Restricts unnecessary Linux capabilities.
Not share host namespaces	6.0.0	Audit	Prevents hostPID/hostIPC/hostNetwork.
Allowed seccomp profiles	7.2.0	Audit	Requires RuntimeDefault or approved profiles.
Allowed AppArmor profiles	6.2.1	Audit	Enforces allowed AppArmor profiles.
Run with read-only filesystem	6.3.0	Audit	Requires read-only root filesystems.
Forbidden sysctls	7.2.0	Audit	Blocks unsafe sysctl settings.

4.11.6 Domain 6 — Workload hardening (supply chain, application networking, reliability)

Policy	Version	Effect	Description
Allowed container images	9.3.0	Audit	Ensures use of approved registries.
No latest tag	2.0.1	Audit	Requires versioned images.
Pull images only with ImagePullSecrets	1.3.1	Audit	Enforces authentication for private registries.
Disable automountServiceAccountToken	4.2.0	Audit	Prevents automatic token mounting.
Services listen only on allowed ports	8.2.0	Audit	Restricts allowed service ports.
Pods use approved host network and ports	7.0.0	Audit	Blocks unsafe hostPort or hostNetwork usage.
Readiness/liveness probes required	3.3.0	Audit	Ensures availability and self-healing.
Must have anti-affinity or topology spread	1.2.2	Audit	Prevents workload concentration on one node.
Should not use naked pods	2.3.1	Audit	Flags Pods not backed by controllers.

Chapter 5

Tenant Server Architecture

5.1 Introduction to the Tenant Server

The tenant server is the architectural core of the proposed security platform. Within the multi-tenant model described in this thesis, it provides two essential and complementary functions. On one side, it ensures the consistent and controlled distribution of governance policies to all tenant clients. On the other, it collects, analyzes, and visualizes compliance evidence coming from the various AKS clusters deployed inside client tenants. These two dimensions, policy distribution and security posture visibility, form the operational pillars of the platform.

Policy Distribution Hub

The first function concerns the definition and centralized management of policies. In a distributed environment, where each tenant may have multiple AKS clusters with independent lifecycles, it is crucial to rely on a secure and overarching domain where the security baseline is maintained. The tenant server takes on this role by acting as the single source of truth for all the policies that define the governance framework applied to Kubernetes workloads. Within this controlled environment, policies are validated and stored before they are made available to tenant clients.

The goal is not to impose direct operational control over tenant resources, but to provide a model that ensures consistency, verifiability, and protection from misconfigurations. This approach reduces risks related to manual configuration, limits operational drift, and provides a clear boundary of responsibility while preserving the technical autonomy of each tenant. The separation between the governance domain (tenant server) and the execution domain (tenant client) is one of the key contributions of this architecture, as it creates a balance between centralized policy definition and the operational independence of application teams.

Central Posture Visibility

The second function of the tenant server focuses on the collection of compliance information. While tenant clients handle policy enforcement and the generation

of audit data, the tenant server is responsible for aggregating and analyzing that information. To support a complete governance model, the platform must include an infrastructure capable of receiving Azure Policy assessment results from multiple tenants, normalizing and contextualizing them, and turning them into clear and meaningful indicators. To meet this requirement, the platform integrates an ingestion system based on distributed agents, centralized storage for snapshot retention, a scalable analytical engine (Azure Data Explorer), and a consolidated visualization layer built on Grafana.

Together, these components allow the tenant server to offer centralized dashboards from which it is possible to observe the overall posture of the multi-tenant environment, identify relevant deviations, evaluate compliance levels across security domains, and analyze violations detected within individual clusters. The ability to correlate data coming from heterogeneous environments, with different volumes and frequencies, is essential for periodic audits, maturity assessments, early detection of risky configurations, and continuous security monitoring.

This chapter therefore provides an in-depth analysis of both tenant server functions: the centralized definition and distribution of governance policies, and the collection and visualization of security KPIs from tenant clients. Although distinct, these two components work together to create a coherent and scalable ecosystem capable of ensuring strong governance across complex multi-tenant AKS environments.

5.2 Policy Distribution Hub

Each client manages its own AKS clusters and their configurations, but the goal is to ensure that these configurations follow a set of common security standards defined at a central level. The tenant server fulfils this requirement by acting as the authoritative source for configuration management. It first defines and publishes a shared baseline that includes all the policies and initiatives representing the minimum security requirements that must be applied to every environment, regardless of the tenant or cluster. It also handles tenant-specific customisations by allowing additional baselines and tailored policies when needed, while still preserving alignment with the global baseline.

Rather than applying policies directly on the clients' clusters, the tenant server provides a controlled and structured repository from which each client can retrieve the relevant configurations. These configurations are then applied locally through a function running in the client tenant.

This approach was designed to clearly separate governance from operational activities, improving scalability and decoupling between tenants. The tenant server therefore acts as an internal service that does not directly enforce configurations, but simply publishes them so they can be consumed by the clients. Using this model the tenant server remains free of any logic or dependencies related to specific clusters. Instead, it provides the configurations that a client-side agent uses to stay aligned, for example after an update to the global baseline.

5.2.1 Policy repository: Blob Storage Account

The tenant server must expose all policies in a clear and structured way so that client tenants can identify the global baseline, retrieve their specific customisations, and update their cluster configurations independently. To achieve this, the solution uses a logical hierarchy that separates the baseline shared by all tenants from tenant-specific settings and cluster-level configurations. This organisation ensures strong isolation and consistency across the entire system. Since all client-side infrastructure relies on the tenant server to determine which security policies must be applied, the central platform needs a highly available and cost-effective repository capable of storing JSON configuration files, keeping their versions up to date, and efficiently supporting both occasional and frequent access.

For these reasons, Azure Blob Storage is the natural choice for the repository. It is designed to handle large volumes of static objects, offers high scalability, and provides an object-store model that is ideal for immutable or versioned content. Unlike Cosmos DB or SQL Database, which introduce the complexity typical of transactional or NoSQL systems and would exceed the needs of a simple configuration store, Blob Storage offers a far more straightforward model that aligns well with the declarative nature of the solution.

Another key factor is that Azure Blob Storage is specifically built for scenarios requiring efficient distribution of static content and configuration artifacts. This model fits perfectly with the need to publish a set of configurations that client tenants can consume without transactional operations or queries. Blob Storage also provides native integration with Private Endpoints, ensuring that access occurs exclusively from the client tenant's private network. This allows public access to be completely disabled, which is essential for the multilayered security requirements of the solution.

Finally, choosing Blob Storage makes it easy to introduce manifest files, which act as version indexes and describe the structure of the entire repository. These manifests simplify the detection of drift between server and client configurations, since they can be read as static objects directly from the container without relying on complex query mechanisms or transformations. The lightweight object-based model and the service's ability to return content efficiently make Blob Storage the most suitable option for delivering a reliable, secure, and scalable distribution point for all policies used by client tenants.

5.2.2 Centralized Storage Structure

The centralized storage layer is the core of the entire AKS policy governance system. It acts as the authoritative source from which client tenants retrieve the configurations they must apply within their own environments. Because it plays such a critical role, its structure must ensure clarity, isolation, traceability, and reliable verification of all stored content.

Design principles of the structure

The organization of the centralized storage is based on four key principles.

The first is the clear separation of information domains: the global baseline must be stored independently from tenant-specific configurations. This separation ensures that global policies remain consistent, tenant workloads do not interfere with each other, and each layer of configuration maintains its own autonomy.

The second principle is internal multi-tenant isolation. Each tenant is assigned a dedicated container or directory that is not shared with others.

The third principle is the use of a recursive and declarative structure. The distinction between the global baseline, custom baselines, and cluster-level configurations follows the logic of declarative architectures, where each level defines its desired state through initiatives, assignments, and policy definitions. This design makes configurations easier to validate, manage, and version over time.

The final principle is referential clarity through manifest files. Every section of the repository includes a manifest that serves as a structured, versioned index. These manifests make it possible to identify the current version of the configuration, confirm the consistency of the stored material, and perform automated comparisons on the client side.

5.2.3 The global baseline

The **global baseline** is the regulatory core of the entire AKS Security Framework. It defines the essential, non-negotiable set of security requirements that every tenant must follow in order to maintain a consistent level of protection, reduce the attack surface, and ensure coherent governance across different domains.

The baseline consists of a collection of Azure Policies, either individual policies or policies grouped into a single initiative, that describe the minimum acceptable behaviour of AKS clusters and enforce fundamental security configurations regardless of the tenant. These policies cannot be removed or weakened, although tenants are free to introduce additional constraints through their own custom baselines. In this way, the global baseline ensures uniform behaviour across clusters, prevents weak or misaligned configurations, and simplifies auditing thanks to a single, documented model.

The global baseline is stored in a dedicated container within the tenant server:

```
akssf-baseline/
    manifest-baseline.json
    baseline-initiative.json
    policy-definition/
        <policy-id-1>.json
        <policy-id-2>.json
```

`manifest-baseline.json`

This file is a declarative document that describes the baseline version, the list of files that make it up, any integrity metadata such as hashes, and references to the global initiative. Thanks to this structure, client tenants can quickly determine whether their local baseline matches the version published by the server.

`baseline-initiative.json`

This file gathers all mandatory policies into a single Azure Policy Initiative. By grouping them in this way, updates become atomic, versioning becomes easier to manage, and client tenants can create assignments more easily and reliably.

The `policy-definition/` folder

This folder contains the definitions of any custom policies that form part of the global baseline. Built-in Azure Policies are not duplicated in the tenant server, as Microsoft already guarantees their presence and immutability within every tenant environment.

5.2.4 Structure dedicated to each tenant

The structure dedicated to each tenant is designed to create a clear logical boundary between the different domains that participate in the framework. Its purpose is to guarantee isolation, architectural clarity, and long-term scalability. Each tenant therefore has its own root directory, which contains all the artifacts required to describe its security posture, its customisations, and the configurations of the clusters it manages. This design follows the principle that every tenant must be governed and analysed as an independent unit, even though all of them share the same global baseline. Unlike a monolithic repository where the content of different tenants is stored in the same space, it is better to use separate directories that eliminate the risk of collisions, make it immediately clear which artifacts belong to which customer, and reduce the possibility of unauthorised access to data. From an operational point of view, this structure also simplifies onboarding new tenants and evolving their configurations over time, since each domain can be handled independently.

To better understand the architectural model, it is useful to observe how each tenant is represented within the tenant server:

```
<tenant-id-x>/
    manifest-<tenant-id-x>.json
    baseline-assignment-<tenant-id-x>.json

    custom-baseline/
        manifest-custom-baseline-<tenant-id-x>.json
        custom-baseline-initiative-<tenant-id-x>.json
        custom-baseline-assignment-<tenant-id-x>.json
```

```
        policy-definition/
            custom-baseline-<tenant-id-x>-<policy-id-1>.json
            custom-baseline-<tenant-id-x>-<policy-id-2>.json

    clusters/
        <tenant-id-x>-<cluster-id-1>/
            manifest-<tenant-id-x>-<cluster-id-1>.json
            initiative-<tenant-id-x>-<cluster-id-1>.json
            assignment-<tenant-id-x>-<cluster-id-1>.json
            policy-definition/
                <tenant-id-x>-<cluster-id-1>-<policy-id-1>.json
                <tenant-id-x>-<cluster-id-1>-<policy-id-2>.json

        <tenant-id-x>-<cluster-id-2>/
            manifest-<tenant-id-x>-<cluster-id-2>.json
            initiative-<tenant-id-x>-<cluster-id-2>.json
            assignment-<tenant-id-x>-<cluster-id-2>.json
            policy-definition/
                <tenant-id-x>-<cluster-id-2>-<policy-id-1>.json
                <tenant-id-x>-<cluster-id-2>-<policy-id-2>.json
```

This structure shows that each tenant is treated as an independent information domain with its own organised space containing the relevant manifests, assignments, and policy definitions.

Tenant level

The tenant level acts as the entry point to the entire customer domain. Here, the main manifest is stored: a document that describes the overall structure and version of the tenant's configuration and serves as an index for the client-side function. Placing this file at the root ensures that the tenant client immediately has the information it needs to understand which baselines must be applied, which versions are active, and which clusters are managed. Alongside the manifest, this level also contains the assignment of the global baseline, which declares the link between the client tenant and the global baseline published by the server. Keeping this assignment at the tenant level highlights the separation between the shared global baseline and the tenant's own configuration: the baseline is common and centralised, while the assignment is the tenant-specific element that enables its application.

Tenant custom baseline

The tenant-level custom baseline exists to provide each customer with a controlled space in which to define additional policies beyond those included in the global baseline. In enterprise environments, different tenants often adopt distinct security requirements due to internal regulations, application needs, or operational differences. The custom baseline accommodates these variations by allowing tenants to extend the global security model in a structured and traceable way, without

introducing changes that could affect other tenants or the shared platform. By organising this area into a manifest, an initiative, an assignment, and the related policy definitions, the model supports a modular and declarative workflow in which every change is explicit and immediately detectable by the tenant client. This structure also offers a clear governance benefit: tenants can manage their own extensions independently while maintaining strict alignment with the global baseline, which remains centralised and immutable.

Per-cluster configurations

The cluster level is the most granular part of the structure, reflecting the idea that each Kubernetes cluster should be treated as an independent domain with its own operational needs. In practice, clusters within the same tenant often differ significantly: they may use distinct network topologies, expose applications in different ways, run workloads with varying risk profiles, or include additional components such as ingress controllers, service meshes, or customised CSI drivers. Some clusters also require stricter isolation than others. These differences make it impossible to define certain policies at the global or tenant level. Instead, those policies must be associated directly with the specific cluster for which they are relevant.

This becomes evident in scenarios where, for example, a production cluster hosting public-facing applications requires network policies that differ from those used in a cluster that is completely isolated. Similarly, some Gatekeeper constraints may apply only to selected environments, certain clusters dedicated to regulated workloads may need mandatory labels, and high-security clusters might require container runtime policies that would be unnecessary elsewhere. By keeping configurations separate at the cluster level, the model allows tenants to describe the precise behaviour of each environment without affecting the rest of the system.

The recursive organisation of this layer based on manifests, initiatives, assignments, and cluster-specific policy definitions ensures consistency while allowing the repository to scale naturally. Adding a new cluster does not require any architectural changes; it simply involves creating a new dedicated directory that is isolated, easy to manage, and fully aligned with the existing structure.

5.2.5 Artifacts: Policy Definition, Initiative and Assignment

The governance platform implemented in the tenant server is built around three key Azure Policy artifacts: **policy definitions**, **initiatives**, and **policy assignments**. These elements are often mentioned together, but they serve very different and complementary roles. Understanding how they work is essential for appreciating the architectural value of the solution, since the entire control, verification, and synchronisation process between the tenant server and the tenant client revolves around them. They form the formal language that Azure uses to describe and enforce security policies in the cloud.

Policy Definition: the atomic rule of governance

A **policy definition** is the smallest unit of governance in Azure Policy. It describes a condition that must be evaluated on cloud resources and the action to take when that condition is met or violated. Formally, a policy definition is a JSON document that specifies the type of control being enforced, the scope of applicable resources, the configurable parameters, the conditional logic expressed in the `if` and `then` structure, and the effect that the policy should trigger.

Within the tenant server, policy definitions are used in two main contexts: the global baseline policies stored in the `akssf-baseline` container, which represent the mandatory rules that every AKS cluster must follow, and the custom policies stored within each tenant or cluster directory, which address requirements that fall outside the global baseline. These definitions are therefore the fundamental building blocks of the entire governance system. They are fully declarative, versionable, referenced in the manifest files, and interpreted consistently by both the tenant server and the tenant clients.

Initiative: the structured aggregation of rules

Managing large numbers of individual policies becomes complex and error-prone in enterprise environments. For this reason, Azure introduces the concept of an **initiative**, a logical container that groups multiple policy definitions into a single, coherent, and versionable unit. Using an initiative makes it possible to maintain consistent versions of entire rule sets, apply a single assignment to many policies at once, ensure atomic and predictable policy enforcement, and maintain transparency in audit and compliance processes.

In the proposed architecture, initiatives are used at three levels. The global baseline initiative, stored in `baseline-initiative.json`, aggregates all the mandatory AKS security policies that every cluster must comply with. Tenant-specific initiatives, stored in the `custom-baseline` directory, allow tenants to introduce additional rules that extend the global baseline. Cluster-specific initiatives, found in the `clusters/<tenant>-<cluster-id>` directories, capture rules associated with the characteristics or sensitivity of a particular cluster. Keeping these initiatives separate ensures a scalable governance model in which updating the global baseline does not affect tenant or cluster configurations, and cluster-level changes do not impact other environments.

Policy Assignment: the effective application of rules

If a policy definition expresses the rule and an initiative organises groups of rules, a **policy assignment** represents the actual act of applying those rules to an Azure resource. The assignment specifies the scope to which the policy applies, which may be a management group, a subscription, a resource group, or an individual resource such as an AKS cluster.

In this model, the tenant server publishes all required assignments in a declarative form. It never applies them directly; instead, the tenant clients read the

assignments from the server's storage, replicate them in their own local repository, and apply them to their AKS clusters through the Function App. Assignments are therefore crucial because they are the only artifact that binds a policy to a cluster and activates it. They also enable differentiated policy application across tenants and clusters, since the same initiative can be assigned to different scopes with different parameters. Finally, assignments include the context-specific settings that control policy behaviour, such as enforcement mode, exemptions, or configurable values.

Through this separation of roles, the architecture ensures that the tenant server remains a pure governance and publication layer, while the tenant clients are responsible for applying the rules within their own operational domains.

5.2.6 Artifacts: Manifests as the Declarative Layer

Manifests form the declarative orchestration layer that allows the tenant server to expose a structured, verifiable, and versioned configuration of security policies to tenant clients. Conceptually, they play a role similar to the *desired state files* used in GitOps systems: they provide an abstract representation of the intended system state and make it possible to compare the *declared* state with the *actual* one.

This solution adopts the same principles. Every area of the tenant server (the global baseline, the custom baseline, and the cluster level) contains a manifest that describes which policies, initiatives, and assignments define the "correct state" for that scope. In this way, the tenant client can independently detect drift, meaning a divergence between its local configuration and the version published by the server, without requiring complex imperative logic.

General Purpose of the Manifests

Manifests exist for three main reasons:

1. **Provide a declarative logical schema**
   They describe what must be included in the configuration (policy, initiative, assignment), not how it should be applied. This supports a clean, modular, and easily extensible architecture.

2. **Enable semantic versioning through `configVersion`**
   Every change to the configuration updates the `configVersion` value. The tenant client can compare its local version with the remote one and immediately understand whether an update is required.

3. **Offer a consistent and stable index of paths**
   The client knows exactly which file to download and where it is located, avoiding storage exploration and reducing the chance of logical inconsistencies.

This architecture aligns with the state reconciliation paradigm typically used in declarative operations.

Main Tenant Manifest

This is the entry point for the tenant client:

`manifest-<tenant-id>.json`

This file connects all other manifests into a single coherent view and acts as the truth map for the tenant domain.

Example of a main manifest

```
{
  "schemaVersion": "1.0",
  "configVersion": "2025.11.18-tenant-001",
  "tenantId": "asl06",

  "baseline": {
    "configVersion": "2025.11.18-baseline-001",
    "globalBaselineManifestPath":
       "../akssf-baseline/manifest-baseline.json",
    "baselineAssignmentPath":
       "baseline/baseline-assignment-asl06.json"
  },

  "customBaseline": {
    "configVersion": "2025.11.18-custom-baseline-003",
    "path": "custom-baseline/manifest-custom-baseline-asl06.json"
  },

  "clusters": [
    {
      "clusterId": "aks-asl06-cl1",
      "configVersion": "2025.11.18-cluster1-002",
      "path":
         "clusters/asl06-aks-asl06-cl1/manifest-asl06-cl1.json"
    }
  ],

  "metadata": {
    "lastUpdated": "2025-11-18T12:00:00Z",
    "source": "server"
  }
}
```

The tenant client uses this manifest to determine:

- which sections of the configuration need validation,

- which secondary manifests must be downloaded,
- which versions must be compared.

5.2.7 Global Baseline Manifest

The main global baseline manifest is stored in the container:

`akssf-baseline/manifest-baseline.json`

It defines the baseline shared by all tenants and represents the root of the governance model.

Example of manifest-baseline.json

```
{
  "schemaVersion": "1.0",
  "configVersion": "2025.11.18-baseline-001",
  "description": "Global baseline for AKS Security Framework",
  "initiativePath": "baseline-initiative.json",
  "policyDefinitions": [
    "policy-definition/policy-network-restriction.json",
    "policy-definition/policy-approved-images.json"
  ],
  "metadata": {
    "lastUpdated": "2025-11-18T09:00:00Z",
    "scope": "global"
  }
}
```

From this manifest, the tenant client understands:

- which baseline version it must have,
- which policies define the global baseline,
- which initiative should be applied locally.

Tenant Custom Baseline Manifest

Each tenant can extend the baseline with additional policies. This level is recorded in:

`custom-baseline/manifest-custom-baseline-<tenant-id>.json`

Example

```
{
  "schemaVersion": "1.0",
  "configVersion": "2025.11.18-custom-baseline-003",
  "tenantId": "asl06",
  "customInitiativePath": "custom-baseline-initiative-asl06.json",
  "customAssignmentPath": "custom-baseline-assignment-asl06.json",
  "policyDefinitions": [
    "policy-definition/custom-baseline-asl06-allowed-ingress.json",
    "policy-definition/custom-baseline-asl06-image-scanning.json"
  ],
  "metadata": {
    "lastUpdated": "2025-11-18T10:00:00Z",
    "scope": "tenant"
  }
}
```

This manifest allows the tenant server to declare:

- which additional policies apply to that tenant,
- where these policies are stored,
- which version represents the current state.

Cluster Manifests

Each cluster is an isolated domain where different policies may be applied.

`clusters/<tenant-id>-<cluster-id>/manifest-<tenant>-<cluster>.json`

Example of a cluster manifest

```
{
  "schemaVersion": "1.0",
  "configVersion": "2025.11.18-cluster1-002",
  "clusterId": "aks-asl06-cl1",
  "initiativePath": "initiative-asl06-aks-asl06-cl1.json",
  "assignmentPath": "assignment-asl06-aks-asl06-cl1.json",
  "policyDefinitions": [
    "policy-definition/asl06-aks-asl06-cl1-network-boundary.json",
    "policy-definition/asl06-aks-asl06-cl1-kubelet-hardening.json"
  ],
  "metadata": {
    "environment": "production",
    "lastUpdated": "2025-11-18T11:30:00Z",
    "scope": "cluster"
```

```
15     }
16 }
```

By reading this manifest, the client can identify:

- which cluster artifacts need updating,
- whether new policies are required for the cluster,
- whether a mismatch exists between client and server.

The Role of `configVersion` as the Pivot of the Algorithm

The `configVersion` value is a central element of the entire orchestration model. The tenant client's Function App uses these versions to determine:

- whether the baseline has changed,
- whether the tenant has new policies,
- whether a cluster needs updates.

Compared to a file-by-file comparison, which is expensive, inefficient, and fragile, version comparison enables a model that is idempotent, scalable, and adaptable.

5.2.8 Manifest Evolution to Increase Security

In the basic version of the architecture, manifests describe the desired configuration state in a declarative way (global baseline, tenant baseline, and per-cluster configurations). This allows the tenant client to detect a shift by comparing the `configVersion` and the file paths. This model already ensures consistency and detects configuration drift, but it implicitly assumes that the tenant server is always trustworthy and that the distribution channel (Blob Storage) cannot be tampered with. In a more advanced scenario, where partial compromise of the server domain or attempts to impersonate the client must also be considered, the manifest can evolve into a **signed object** from an integrity and authenticity point of view.

The main idea is to enrich the logical description (list of files, versions, paths) with a set of **cryptographic metadata**. These metadata allow the client to answer two key questions: "Are the manifest and policies I am reading exactly those published by the legitimate source?" and "Can the server uniquely recognize me as a client so that a third party cannot impersonate my identity?"

Symmetric Secrets in the Server and Client Key Vaults

To address the first problem (malicious server behavior or manipulated storage), a **shared symmetric secret** can be introduced between the platform and each tenant. This secret is stored:

- in the tenant server's Key Vault, under a name such as `AKSSF-MANIFEST-INTEGRITY-KEY`, accessible only to the process responsible for publishing manifests;
- in the tenant client's Key Vault, containing the same value, accessible only to the Function that performs synchronization.

This secret is used to compute an **HMAC** (Hash-based Message Authentication Code) over the manifest contents and, optionally, over the associated policy files. HMAC is a standard construction that combines a cryptographic hash function with a shared secret to provide integrity and message authentication.[67]

In practice, each manifest is extended with an additional field such as:

```
"integrity": {
  "algorithm": "HMAC-SHA256",
  "hmac": "b5f86a9c...abcd",
  "scope": "manifest+policy-files"
}
```

The publishing process on the tenant server computes the HMAC on the manifest JSON (and, in a stronger variant, on an ordered concatenation of the hash values of the individual policy files) using the key stored in the server Key Vault. After downloading the manifest from Blob Storage, the tenant client reconstructs the same input locally, computes the HMAC using the key stored in its Key Vault, and compares it with the received value. If they differ, the Function treats the manifest as untrusted and stops the update.

This evolution has an important implication: if an attacker compromises **only the storage**, they cannot transparently modify manifests or policies because they do not have the secret required to produce a valid HMAC. Even a partially malicious operator who modifies blobs directly cannot generate a manifest accepted by the client.

File Hashes and Rollback Protection

In addition to the manifest-level HMAC, the evolution includes adding **file-level hashes** for each policy file inside the manifest. For example:

```
"policyFiles": [
  {
    "path":
       "policy-definition/policy-aks-disable-local-accounts.json",
    "sha256": "9a7c1e4b..."
```

```
  },
  {
    "path":
       "policy-definition/policy-aks-restrict-public-ip.json",
    "sha256": "3bd09f2a..."
  }
]
```

This allows the client to compute the hash of each downloaded policy file and compare it with the declared value. Any alteration is detected immediately.

To mitigate the risk of **malicious rollback** (redistributing an old but correctly signed manifest), the `configVersion` value is treated as a **monotonic logical version**. The tenant client stores the last accepted version and rejects lower ones unless explicitly authorized. This approach follows the versioning semantics commonly used in GitOps and Kubernetes.

Manifests at All Levels: Baseline, Tenant, Cluster

The proposed evolution applies uniformly across all four manifest levels:

1. `manifest-baseline.json` in the global `akssf-baseline` container.
2. `manifest-<tenant-id>.json` orchestrating tenant configuration.
3. `manifest-custom-baseline-<tenant-id>.json`.
4. `manifest-<tenant>-<cluster>.json` for each cluster.

For each artifact, the `integrity` field is computed using the same symmetric secret, while the file-hash list covers only the resources belonging to that manifest. This layered design allows precise localization of any suspicious alteration.

Detecting an Impersonated Client

The second axis of the evolution addresses the possibility that an external actor might attempt to impersonate the tenant client.

Initially, protection relies on:

- a Service Principal with `Storage Blob Data Reader`,
- its credentials stored in the tenant's Key Vault,
- OAuth2 authentication flows.

To strengthen identity guarantees, a second symmetric secret can be introduced: **client attestation key**, such as `AKSSF-CLIENT-ATTESTATION-KEY`, stored in the client and server Key Vaults. In a possible evolution of the architecture, instead of accessing blobs directly, the client could call a minimal endpoint exposed by the tenant server (for example a Function or internal API behind Private Link), and include in each request an HMAC signature over a nonce or request payload using this attestation key. The server verifies the HMAC using the same key, ensuring that the request comes from a client that possesses the secret. If an attacker reuses the Service Principal but does not have access to the client Key Vault (and therefore to the attestation key), they cannot produce a valid signature.

This approach follows classic models of "message-based authentication", recommended when it is necessary to prove both application identity and the origin of specific requests. It is conceptually similar to "message integrity" and "sender authentication" techniques described in NIST cryptographic security documents and IETF MAC guidelines: a symmetric secret shared between two trusted parties makes spoofing impossible for anyone who does not possess the key.

Within the proposed solution, this evolution can be considered a next step, especially if the architecture is expanded to scenarios where the tenant server offers additional functionality (for example active validation or orchestration of change requests) rather than only static file distribution. Even without immediately introducing a server-side endpoint, combining HMAC on manifests, per-file hashes, and rigorous secret management in Key Vault already provides a major improvement to the trust model between tenant server and tenant client, and forms a solid foundation for future extensions toward a full attestation model.

5.2.9 Security of Access to the Tenant Server Storage

The storage used by the tenant server is one of the most sensitive components of the entire architecture. It contains the global security baseline, the tenant-specific configurations, and the policy definitions that tenant clients rely on to govern their AKS clusters. If this central repository were to be compromised or exposed incorrectly, the entire governance framework would lose reliability. For this reason, the access model has been designed according to several strict principles: no public endpoints, strong network isolation, secure cross-tenant authentication, and minimal permissions.

The core idea is to treat the tenant server as an internal *policy as a service* component. It is not an operational system acting directly on clusters, but an authoritative and tightly controlled source of configuration, reachable only by trusted agents (the Function Apps in each tenant client) through Azure's private backbone.

Network Architecture: Private Link, Dedicated VNets, and Private DNS

The first architectural choice is the complete removal of public endpoints from the tenant server storage account. This aligns with Microsoft Cloud Security Benchmark recommendations for Azure Storage, which advise disabling anonymous and

public access and using Private Endpoints for sensitive data flows. This significantly reduces the attack surface by eliminating scenarios such as internet-wide scanning, random IP access attempts, or firewall misconfiguration.

Communication between tenant clients and the server's storage occurs exclusively through **Azure Private Link**. The storage account is exposed as a private endpoint inside a subnet of the client's VNet. From the client's perspective, the storage behaves like an internal service reachable through a private IP address. From the server's perspective, every request flows through an approved Private Endpoint.

The network architecture includes a **dedicated Virtual Network** in the tenant server containing subnets for:

- Private Endpoints to storage, optionally protected with Network Security Groups,
- administrative components (e.g., jumpboxes or automation services),
- other internal services with no public exposure.

For Private Endpoints to function correctly, DNS resolution must follow a private-first approach. Azure requires that the public FQDN of the storage account (e.g., `mystorage.blob.core.windows.net`) resolve to the Private Endpoint's private IP. This is achieved through a **Private DNS Zone**, typically `privatelink.blob.core.windows.net`, containing A records such as:

```
<storage-account-name>.privatelink.blob.core.windows.net → <private-PE-IP>
```

and linked to the client VNet.

This combination guarantees that the traffic never leaves the private network, no public endpoint can be used accidentally and that the access is jointly controlled by the client (owner of the VNet) and the server (approver of Private Endpoints).

Security of the Management Plane

Even though the tenant server exposes only static content, its management plane is highly sensitive because it controls updates to the baseline and tenant configurations. Administrative operations on storage are protected by:

- strict Azure RBAC separation of duties,
- Conditional Access and MFA for privileged users,
- Privileged Identity Management (PIM),
- device compliance and session protections.

These controls align with Microsoft recommendations for protecting Entra ID privileged accounts.

Isolation and Multi-Tenancy in Storage

The storage is organized to ensure **strong multi-tenancy**. Its structure includes:

- one container for the global baseline,
- one container per tenant, containing only that tenant's content.

This layout enables a highly granular RBAC model: the Service Principal assigned to a tenant receives the Storage Blob Data Reader role only on the container corresponding to that tenant, not on the global baseline container or the entire storage account. Microsoft explicitly recommends using Azure AD and RBAC instead of account keys or long-lived SAS tokens, as this enables fine-grained authorization and full auditability. Another security benefit is that the tenant server does not execute any client-provided code. No functions or containers run on behalf of tenants. This eliminates entire classes of attacks related to code injection, runtime exploitation, or privilege escalation from customer workloads. The communication flow is deliberately one-way: the tenant client reads configuration from the server's storage but cannot write or modify anything.

The Challenge of Cross-Tenant Authentication

The tenant server and tenant clients live in separate Entra ID tenants. A Function's Managed Identity cannot authenticate across tenants, as formally documented by Microsoft. For this reason, MI cannot be used for this architecture.

Workload Identity Federation was also evaluated but rejected due to:

- complexity of cross-tenant trust configuration,
- added operational overhead,
- its design focus on DevOps rather than multi-tenant serverless clients.

5.2.10 Service Principal + Key Vault: the Selected Model

The chosen strategy for cross-tenant authentication uses a **Service Principal** (App Registration) created in the tenant server, with its credentials securely stored in an **Azure Key Vault** inside the tenant client. This model fully aligns with Microsoft's official documentation on how to expose resources to applications in another tenant through Service Principals and Azure AD.

The logical flow works as follows:

1. In the tenant server, an App Registration dedicated to a single tenant client is created (for example `akssf-storage-client-asl06`).
2. This application is granted, on the server's storage, only the **Storage Blob Data Reader** role *on the tenant's container* (not on the entire account, and not on the global baseline container).

3. In the tenant client, the following values are stored as secrets in an Azure Key Vault:
   - server Tenant ID,
   - Service Principal Client ID,
   - Client Secret,
   - storage account name,
   - tenant container name.
4. The client Function uses its own Managed Identity to read these secrets from the Key Vault, then uses these values to create a `ClientSecretCredential` (or equivalent) to authenticate against the tenant server and obtain an OAuth2 token valid for accessing the server's storage.
5. This token is then used by the Azure Storage SDK to authorize read operations on the dedicated container.

In this model, no credentials are hard-coded in the Function code; the secrets are stored in Key Vault and accessible only to the client's Managed Identity, while storage permissions are controlled through RBAC. Microsoft recommends this exact combination—application identities, RBAC, and Key Vault—as a secure alternative to account keys or broad, long-lived SAS tokens.

5.2.11 Minimization of Permissions and Domain Separation

A key advantage of the Service Principal + Key Vault approach is the ability to apply the *least privilege* principle strictly. Each tenant client receives:

- a dedicated Service Principal in the tenant server,
- a **Storage Blob Data Reader** role limited to its own container.

No permissions are granted on the entire storage account or on the global baseline container. This reduces the risk that misuse of the Service Principal could:

- access configurations belonging to other tenants,
- expose global administrative policy definitions,
- use the server storage as a source of information beyond the intended scope.

In addition, the separation works in both directions: the server manages who is authorized to access which container, while the client controls who within its environment can actually use the credentials (for example ensuring that only the Function and an administrator can read from the Key Vault).

5.3 Central Posture Visibility

5.3.1 Overview of the Main Components

The central posture visibility in the server architecture is composed of four main building blocks:

1. **Ingestion API / VM Collector**

   A Linux VM hosts a private HTTP API (endpoint `/ingest`) that receives JSON snapshots from tenant clients through a private connection (*Private Endpoint → Private Link Service*). This API decouples the collection logic from the analytics systems, allowing the writing and validation layer to be centralized.

   A more desirable alternative would be a serverless Function App, which offers built-in scalability and lower costs. However, this is not feasible because integrating a Function App with a Private Link Service currently requires a Load Balancer, which is not supported.

2. **Data Lake on Azure Storage**

   The raw snapshots are stored in a container inside a Storage Account with a Hierarchical Namespace (Data Lake Gen2). This option supports storing raw data and enables automated ingestion into analytical systems.

   The alternative options considered were:

   - **Log Analytics**, which is excellent for system logs but less efficient for large-scale storage and processing of compressed JSON files intended for custom analytics.
   - A **flat Blob storage**, which lacks structure and reduces optimization opportunities for partitioning and ingestion.

   The chosen solution is a Gen2 Storage Account with hierarchical namespace because it supports standard partitioning patterns, efficient ingestion, and compatibility with Event Grid for ADX triggers.

3. **Azure Data Explorer (ADX)**

   For near real-time analysis of compliance results, ADX is used as the query engine. It supports ingestion from Blob/Event Grid, JSON-based queries, update policies, materialized views, and operations on high-volume datasets.

   Alternatives such as Azure SQL Data Warehouse are oriented toward batch workloads and are not optimized for near real-time ingestion of JSON or ad hoc log queries. They also require schema rigidity and incur higher cost and complexity for telemetry-heavy scenarios.

4. **Azure Managed Grafana**

   Azure Managed Grafana is used to present and visualize compliance KPIs and multi-tenant dashboards. It integrates directly with ADX, supports Azure AD authentication, and provides a rich set of visualization plugins.

The alternatives were:

- **Power BI**, which is excellent for reporting but less suitable for interactive near real-time dashboards, often requiring refresh intervals and predefined data models.
- **Workbooks**, which are lightweight but less flexible and limited in design.
- A fully custom portal, which would require significant development and ongoing maintenance.

For these reasons, Azure Managed Grafana was selected. It provides dynamic visualization, filtering, drill-down capabilities, support for variables (tenant, cluster, policy), native integration with ADX and Azure AD, and no additional licensing cost.

5.3.2 Network Architecture of the Tenant Server

The network architecture of the tenant server is where the balance between **centralization** (all data converging into a single point) and **isolation** (no customer can see beyond their own boundary) becomes most visible. This section describes how the network is structured inside the tenant server, how access is provided through Private Link, and how the full end-to-end flow works from the customer's Function App to the collector VM.

Tenant Server Virtual Network and Internal Segmentation

The tenant server includes a **dedicated Virtual Network** hosting all components involved in ingestion, analytics, and visualization:

- the **console subnet**, containing the collector VM and the internal Load Balancer,
- a subnet dedicated to **Azure Bastion**, used for administrative access without public IPs,
- optional service subnets for Private Endpoints or future integrations.

In practice:

- the **collector VM** runs inside a subnet without any public IP address and is protected by a Network Security Group (NSG) that:
  - allows only inbound traffic from the **Load Balancer** (toward the API listening port, for example 8080),
  - allows only SSH access from **Azure Bastion** (port 22),
  - blocks all other inbound traffic (default deny).

- the Bastion subnet follows Azure's required setup (`AzureBastionSubnet`), allowing Bastion to reach VMs using private IPs.

Applying NSGs at the subnet level rather than per network interface simplifies management and reduces the risk of inconsistent configurations.

Private Link Service as the Multi-Tenant Entry Point

The entry point for tenant clients is an **internal Azure Load Balancer**, which fronts the collector VM and is exposed externally only through an **Azure Private Link Service (PLS)**.

The pattern works as follows:

1. The internal Load Balancer has a **private IP** in the console subnet, with a rule forwarding port 80 (frontend) to the collector API port (for example 8080) on the backend.

2. The Private Link Service is configured in front of the Load Balancer, effectively publishing the service as a private endpoint that tenant clients can connect to.

3. Each tenant client creates a **Private Endpoint** inside its own VNet, pointing to the PLS Alias. This Private Endpoint receives a private IP inside the client's VNet that the client Function uses to call the `/ingest` API.

Azure's Private Link documentation highlights exactly this scenario: one **Private Link Service** can be reached by multiple Private Endpoints belonging to different VNets, subscriptions, and tenants, with explicit approval workflows controlled by the service owner.

This is precisely what enables **secure multi-tenancy**:

- the tenant server publishes its service once via PLS,

- each tenant client creates its own Private Endpoint,

- the PLS owner approves or rejects each connection request, maintaining tight control over cross-tenant access.

From a security perspective, Microsoft recommends Private Link for scenarios where a service must be exposed to customers or other tenants without using public endpoints, relying instead on private IPs inside the consumer VNet and the Azure backbone for transport.

End-to-End Network Flow

Once the server-side network is defined (VNet, subnets, NSG, Bastion, internal Load Balancer, PLS), the **complete flow** from a tenant client to the collector is:

1. In the **tenant client**, an Azure Function (or equivalent agent), running inside an integrated VNet and without a public IP, periodically sends a JSON snapshot to an address such as `http://<IP_PE>/ingest`, where `<IP_PE>` is the Private Endpoint's private IP in the client's VNet.
2. The HTTP request to `<IP_PE>` is routed to the **Private Endpoint**, which acts as a local private representation of the remote service in the tenant server.
3. The Private Endpoint forwards the traffic to the **Private Link Service** in the tenant server, using the Azure backbone without ever traversing the public internet.
4. The PLS forwards the traffic to the **frontend of the internal Load Balancer**, which then distributes the request to the collector VM (backend pool) on the API listener port.
5. The collector VM receives the request through its private IP, passes NSG rules that allow only legitimate traffic (from the relevant service tags and the Load Balancer subnet), and processes the snapshot (validation, writing to Blob, etc.).

Overall, the network flow remains fully private, controlled, and free of any public endpoints or internet-exposed components.

Comparison with Alternative Network Architectures

This network architecture is not the only possible approach, but it is selected based on clear trade-offs compared with alternative models.

Public Exposure of the Ingestion API A simpler option would be to expose the collector API on the public internet, for example through:

- an Application Gateway with WAF,
- an API Management gateway and a public Web App or Function.

Tenant clients would send snapshots to a public FQDN secured with TLS and Azure AD authentication. While this model is technically valid, it has several disadvantages:

- it increases the **attack surface**, requiring more complex hardening (WAF, DDoS protection, IP filtering),

- it requires tenant clients to allow outbound internet traffic specifically to that endpoint, which can conflict with internal security policies,
- it reduces the ability to guarantee that all traffic stays on the Azure backbone.

For these reasons, **Private Link** is the recommended pattern, as it enables private connections between VNets and PaaS or custom services in another tenant, without relying on any public exposure.

Peering Between Client and Server VNets Another alternative would be **VNet peering** between client VNets and the tenant server VNet, exposing the collector API directly as a private IP inside the server's network. However:

- it does not scale well as the number of tenants grows (N-to-1 peering relationships),
- it introduces risks of overlapping IP ranges across independent customers, which Private Link avoids,
- it complicates routing and firewall governance between separate organizations.

Access via VPN or ExpressRoute A more traditional approach would connect tenant clients to the server through VPN or ExpressRoute. This also comes with drawbacks:

- high cost and operational complexity at scale,
- the need to manage routing, BGP, and AS numbers for each customer,
- no built-in per-service approval mechanism like the one provided by Private Link.

5.3.3 Collection Service: VM Collector and Ingestion API

The collection service is the entry point for all snapshots coming from tenant clients and forms the operational boundary between the multi-tenant domain and the centralized domain of the tenant server. It is a critical component, both for security and for orchestrating the data flow. Its design follows principles of robustness, isolation, and simplicity. The decision to base the collector on one or more Linux VMs comes from the need for full control over the runtime, networking configuration, and software dependencies, while keeping the component lightweight and easy to scale.

The collector's primary responsibility is not analysis, normalization, or KPI computation. Instead, it functions as a focused ingestion service: receiving snapshots, performing minimal validation, and storing them. All interpretive logic is delegated to the tenant server pipeline, and especially to Azure Data Explorer (ADX), which is the sole component responsible for parsing, data extraction, and

KPI computation. This approach follows the typical "collect → store → process" pattern of log ingestion and data lake architectures, helping keep the exposed component simple, reducing attack surface, and decoupling ingestion throughput from analytical throughput.

Role of the Collector in Decoupling Tenants from Analytics Systems

The collector acts as the single ingestion layer for all tenants. It receives compliance snapshots produced by client agents and moves them into the centralized domain, where they can be stored and later processed. This design allows client agents to remain extremely simple. Their job is only to periodically gather AKS policy results through the Azure Policy API and send a single payload to the server, without handling aggregation or transformation logic.

The collector is intentionally stateless. Each request is treated as an independent event. When a snapshot arrives, the collector checks its minimal structure, computes technical metadata (such as content hash or file size), assigns a server-side timestamp, and stores the file in the tenant server's data lake. The operation ends with a response to the client tenant. Semantic analysis, field extraction, correlation, and KPI production are all handled exclusively by ADX, which is designed to process large, heterogeneous datasets with evolving schemas.

This model allows the collector to serve as a resilient buffer between multiple tenants and the central analytics layer. Even if ADX or downstream systems are temporarily unavailable, snapshots remain safely stored in the data lake and can be processed later without requiring the tenant to resend them.

Design of the Ingestion API

The collector exposes a minimal set of API endpoints to keep the interaction surface small and easy to govern:

- a main endpoint (`POST /ingest`) that receives compressed JSON snapshots,
- a health check endpoint (`GET /health`) also used by the internal Load Balancer,
- a version endpoint (`GET /version`) for compatibility checks and debugging.

The snapshot structure is designed to be flexible and extensible. Each payload contains a unique identifier, the client tenant ID, a generation timestamp, and the full set of policy results and violations at the time of collection. The collector does not impose a rigid schema. It verifies only a minimal set of fields and stores the content as-is, following the *schema-on-read* principle used in most data lake architectures, where interpretation is deferred to the analytics layer.

In addition to the JSON body, client agents send technical headers such as schema version or agent version. These allow the tenant server to track the evolution of the format and maintain long-term backward compatibility. The separation between "content" and "transport metadata" simplifies API maintenance without adding constraints to the payload itself.

Storage Model: Write-Once, Append-Only, and Immutability

Snapshots received by the collector are stored in a dedicated area of the data lake using a *write-once*, *append-only* model, with no overwriting. Each snapshot is a complete representation of a tenant's state at a specific moment and is treated as an immutable document. This approach reflects best practices for managing security logs and audit data, where preserving history and chain of custody is essential.

Snapshot storage follows a hierarchical partitioning pattern based on tenant and date:

```
snapshots/{tenant_id}/{yyyy}/{MM}/{dd}/{snapshot_id}.json.gz
```

Compression improves transfer speed and reduces storage cost. ADX can ingest compressed JSON files directly, simplifying the architecture and removing the need for intermediate preprocessing steps.

The collector simply stores the file and records minimal metadata. Parsing is handled by ADX through update policies that read the raw file, extract relevant fields, and populate normalized tables.

Robustness and Resilience Mechanisms

To ensure scalability and reliability in a multi-tenant scenario, the collector implements several defensive mechanisms to prevent abuse, operational mistakes, and abnormal load patterns. Payload validation ensures that each snapshot contains at least the essential fields and does not exceed size thresholds incompatible with the pipeline's capacity. Idempotent handling, based on content hash and the uniqueness of the `snapshot_id`, avoids duplicates in the data lake while supporting retry mechanisms from client agents. The VM also applies rate limiting to prevent malfunctioning or compromised tenants from overwhelming the collector with excessive or continuous requests.

A core design principle is that the collector performs no semantic processing of snapshots. It does not interpret policies, aggregate violations, or compute compliance metrics. This dramatically reduces the complexity of the exposed component and ensures that a compromise or malfunction of the collector cannot affect the validity of the KPIs. The collector is a controlled transit node, while ADX remains the only point where computation and interpretation take place.

5.3.4 Centralized Data Lake on Azure Storage

This section describes the "raw" storage layer of the solution: the **Data Lake** in the tenant server where all snapshots from tenant clients are stored. It explains why Azure Storage Gen2 was selected, how the folder structure is organized, and which security controls are applied.

Role of the Data Lake in the Two-Layer Model (raw + analytical)

In log analytics and compliance architectures, it is common to distinguish two main layers:

- **Raw (landing) layer:** where original payloads from clients are stored with minimal transformation.
- **Analytical (processed) layer:** where data is indexed, normalized, correlated, and queried.

This separation allows you to:

- keep **original payloads immutable**, useful for audits, forensics, chronological analysis, or historical recovery,
- support schema evolution without losing raw data,
- preserve a fallback location in case the analytics pipeline has issues, since raw files are always available.

For these reasons, using a centralized Data Lake on Azure Storage aligns well with Microsoft's best practices for log analytics, telemetry, and audit scenarios.

Data Organization Inside the Container

Data is stored in a **Storage Account with Hierarchical Namespace** (Data Lake Gen2) inside a dedicated container (for example `policy-snapshots`). The directory structure is:

```
/policy-snapshots
   /snapshots
      /{customer_tenant_id}
         /{yyyy}
            /{MM}
               /{dd}
                 /{snapshot_id}.json.gz
```

Where:

- {`customer_tenant_id`} identifies the client tenant,
- {`yyyy`}/{`MM`}/{`dd`} provides date-based partitioning,
- {`snapshot_id`} is a unique UUID for the snapshot.

Main advantages:

- **Logical filtering and partitioning:** selecting all files for a given tenant or day is straightforward.
- **Clear organization:** the structure immediately shows which tenant produced the snapshot and on which date.
- **Scalability:** time-based partitioning works well with retention policies and long-term archival.

This naming and path structure is commonly recommended in Azure Big Data patterns, where the hierarchy tenant → year → month → day ensures balanced partitions and efficient querying.

Storage Account Hardening

From a security and compliance perspective, the Data Lake is protected using the following controls:

- **Public access disabled:** the `allowBlobPublicAccess` attribute is set to `false`.
- **Network rules (firewall and VNets):** the storage account is accessible only from the collector VM subnet and the internal Private Link service; everything else is blocked.
- **Encryption at rest:**
  - **Customer-Managed Keys (CMK)** stored in Key Vault can be used for encryption at rest when required.
  - Double encryption can be enabled in highly regulated scenarios.
- **Soft delete and immutability (optional):**
  - The `policy-snapshots` container can use soft delete retention and, if required, WORM immutability policies to preserve logs unchanged for compliance periods.
  - This is particularly useful when long-term audit evidence must be stored.
- **Managed Identity + RBAC for write access:**
  - The collector VM uses a **system-assigned Managed Identity** with the `Storage Blob Data Contributor` role on the storage account.
  - This removes the need for static keys or stored credentials, reducing secret exposure risk.
- **Storage access logging and monitoring:**
  - Storage analytics for read/write operations is enabled,
  - Unexpected access attempts (for example from unexpected IP addresses) trigger alerts.

5.3.5 Logical Data Model in Azure Data Explorer

The data model implemented in Azure Data Explorer (ADX) is the analytical core of the platform.

Unlike the Data Lake, which stores raw files (raw snapshots) in a write-once approach, ADX provides a highly indexed, query-oriented model optimized for near real-time telemetry and log analysis.

This section describes:

- why ADX was chosen as the analytical engine,
- the logical data model and relationships between tables,
- the use of Update Policies and Materialized Views,
- how multi-tenancy and schema evolution are handled.

Why Azure Data Explorer Fits the Scenario: Real-Time Ingestion and Telemetry

Azure Data Explorer (ADX) is designed specifically for:

- high-throughput ingestion of telemetry and log data,
- interactive analysis over large datasets,
- semi-structured and nested JSON data,
- real-time and near real-time analytics.

Microsoft documentation highlights that ADX is optimized for append-only data, with fast ingestion and low-latency queries, making it ideal for large volumes of events, audit records, and diagnostic logs. It is used internally by Azure Monitor and Log Analytics precisely because it handles complex and multi-structure log formats efficiently.

In this project, compliance snapshots:

- are semi-structured (JSON with nested objects),
- arrive at regular intervals,
- must be correlated, expanded, and aggregated,
- must feed near real-time dashboards.

Traditional relational databases are not suitable without heavy investments in scaling and index maintenance.

ADX instead provides:

- native JSON mapping into columns,
- `mv-expand` for array expansion,
- Update Policies for populating derived tables,
- Materialized Views for KPI computation,
- KQL queries optimized for logs and time series.

These features match precisely the analytical needs of the system.

Core Tables and Their Relationship to the Conceptual Model

The data model is built to reflect logical concepts of configuration, state, and violations, starting from the snapshots sent by tenant clients.

There are six main entities.

1. `snapshots_raw`

Contains:

- the original JSON payload,
- raw file metadata,
- `raw_uri`,
- identifiers for the customer and the snapshot.

It serves as an immutable archive within ADX and as a bridge between the Data Lake and the analytical database.

2. `snapshots`

Holds extracted metadata from the payload:

Field	Description
`snapshot_id`	Unique UUID
`customer_tenant_id`	Source tenant
`snapshot_time`	Timestamp from client
`full_snapshot`	Complete or partial snapshot
`clusters_count`, `policies_count`	Inventory size
`content_hash`	Hash of raw JSON for idempotency
`payload_size_bytes`	File size
`raw_uri`	Path in the Data Lake

This table enables audit, volume reporting, and temporal correlation.

3. `cluster`

Represents all known AKS clusters for a tenant.

Field	Description
`cluster_id`	Logical cluster identifier
`customer_tenant_id`	Tenant owner
`cluster_name`, `subscription_id`, `resource_group`	ARM metadata
`first_seen`, `last_seen`	SCD2 tracking fields

Useful for maintaining the catalog of clusters over time.

4. `policy_definitions`

Contains policy definitions:

- `policy_definition_id`
- `display_name`
- `category`
- `effect`
- `metadata` (dynamic JSON)
- `version`

This table changes infrequently but is essential for enriching KPI outputs with descriptions.

5. `policy_assignments`

Describes which policies are assigned to which tenant.

Fields include:

- `policy_assignment_id` (not the ARM ID, which is not unique across tenants),
- `policy_definition_id`,
- `customer_tenant_id`,
- `scope` (subscription, resource group, or cluster),
- `first_seen`, `last_seen`, `is_active`.

This implements an SCD2 (slowly changing dimensions) model to track assignment changes over time.

6. `policy_results`

The primary table: contains the evaluation results for each snapshot.

Main fields:

- `snapshot_id`
- `customer_tenant_id`
- `cluster_id`
- `policy_assignment_id`
- `policy_definition_id`
- `status` (Compliant, NonCompliant, NotApplicable)
- `severity` (Low, Medium, High)
- `evaluation_time`
- `details_json`

Each row represents a single Azure Policy evaluation on an AKS cluster.

7. `violations`

Detailed violations at the Kubernetes resource level:

- `resource_kind` (Pod, Deployment, Namespace, etc.),
- `resource_name`,
- `namespace`,
- `reason`,
- `remediation`,
- `evidence_json`.

This table enables deep drill-down, supporting Grafana dashboards and audit queries.

Multi-Tenancy in the ADX Data Model

The system must support multiple tenants in the same ADX cluster.

Multi-tenancy is handled through the following mechanisms.

1. Logical separation via `customer_tenant_id`

All tables include the field:

```
customer_tenant_id: string
```

Queries in Grafana (or future APIs) are always filtered by tenant.

2. Controlled de-normalization

In the `policy_results` table, both `policy_assignment_id` and `policy_definition_id` are stored, even though this introduces duplication.

This is intentional, because:

- ADX is not designed for repeated complex joins,
- controlled de-normalization significantly improves time-series query performance,
- snapshots must remain independent of changes to policy assignments over time.

This follows ADX guidance: *prefer flat schemas and minimize joins.*

5.4 Final Visualization of the Security Posture in the Tenant Server

The final stage of the tenant server workflow is the design and implementation of the summary dashboards, which form the "reading surface". These dashboards bring together all parts of the pipeline described in previous sections: the *JSON snapshots* sent by the agents running in customer AKS clusters, their storage in the centralized data lake, the ingestion into Azure Data Explorer, and finally the queries executed by Grafana using KQL.

This section presents two main types of dashboards:

- a **Global Overview dashboard**, which aggregates data from all tenants and all clusters (Figure 5.1),
- a **Customer Dashboard**, which filters the same metrics to a selected tenant using a Grafana variable (Figure 5.2 and Figure 5.3).

In all screenshots, the tenant and cluster names are **entirely fictitious** (for example *finbank-tenant*, *medcare-tenant*, *retailx-tenant*, *startupy-tenant* and their respective AKS clusters). Numerical values are synthetic but designed to reflect realistic ranges, temporal variability, and the patterns expected from Azure Policy evaluations across multiple Kubernetes clusters.

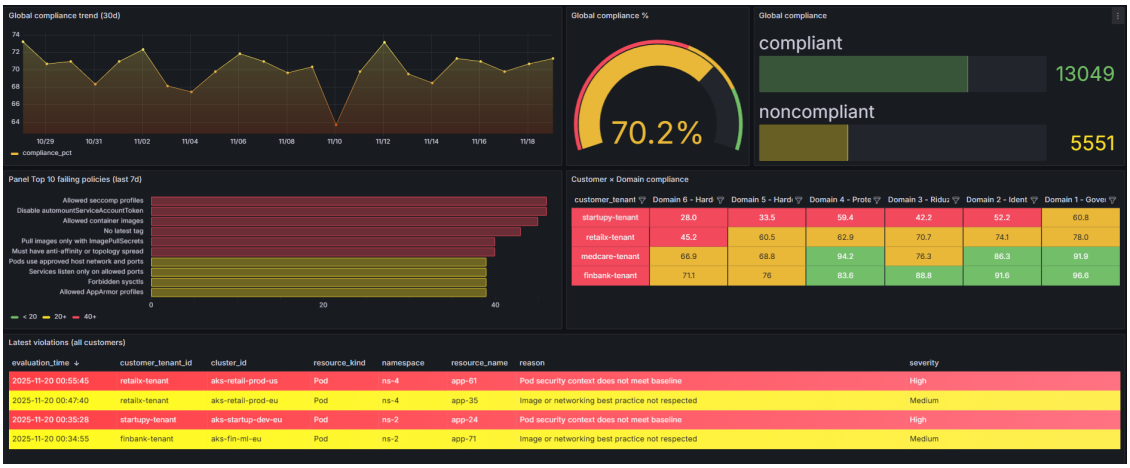

Figure 5.1. Global Overview Dashboard of the AKSSF platform, aggregating compliance data from all tenants and all AKS clusters.

5.4.1 Global Overview Dashboard

The *Overview Dashboard* (Figure 5.1) is the main observation point for governance roles such as CISO, Security Leads, and cloud platform owners. Its goal is to provide an immediate view of the security posture across the entire multi-tenant environment, without focusing on individual violations.

At the top left is the **Global compliance trend (30d)** panel, a time-series chart showing the compliance percentage over the last thirty days. Each point on the chart is generated from a materialized view in Azure Data Explorer that computes, for each day, the ratio of *Compliant* evaluations to the sum of *Compliant* and *NonCompliant* within the `policy_results` table. The simulated oscillations represent the effects of new releases, stricter policies, or remediation campaigns: a sudden drop indicates regression, while an upward trend shows alignment to the security baseline.

At the center of the top row is the **Global compliance %** gauge, which condenses the current posture into a single value. The scale is color-coded (red–yellow–green) so that decision-makers can immediately assess whether the platform is in a risk zone, a caution zone, or a stable zone. The displayed value (around 70% in the example) represents compliance across all clusters and all tenants. This high-level metric fits well with typical management expectations for a small set of clear KPIs.

To the right, the **Global compliance** panel breaks down the same value into absolute counts of *compliant* and *noncompliant* evaluations. This contextualizes

the percentage: 70% compliance on a few hundred evaluations has a very different meaning compared to the same percentage over more than ten thousand checks.

At the center of the dashboard is the **Customer × Domain compliance** heatmap. Each row corresponds to a fictitious tenant (startupy-tenant, retailx-tenant, medcare-tenant, finbank-tenant), while the columns represent the six security domains defined in the logical model: governance and monitoring, identity and access control, exposure reduction, data protection, workload hardening, and supply chain/network/reliability. Each cell displays the tenant's compliance percentage for that domain, color-coded from red to green.

Below the time-series graph, the **Top 10 failing policies (last 7d)** panel lists, through a horizontal bar chart, the ten policies that produced the most violations in the last week. The labels reference policies such as "Allowed seccomp profiles", "Disable automountServiceAccountToken", "Pull images only with ImagePullSecrets", and "Should not use naked pods", mainly across domains 5 and 6.

At the bottom, the **Latest violations (all customers)** table shows the most recent violations across all tenants. Rows are colored based on severity (red for High, yellow for Medium), making critical issues easy to spot.

Overall, the Overview Dashboard visually expresses the idea of **multi-tenant security posture management**. A single screen reveals the aggregated time trend, global KPIs, domain-by-tenant distribution, the worst-performing policies, and the latest violations.

5.4.2 Customer Dashboard: Per-Tenant Analysis

Figure 5.2. Customer Dashboard for a selected tenant (example: finbank-tenant), filtering compliance KPIs and violations to a single customer.

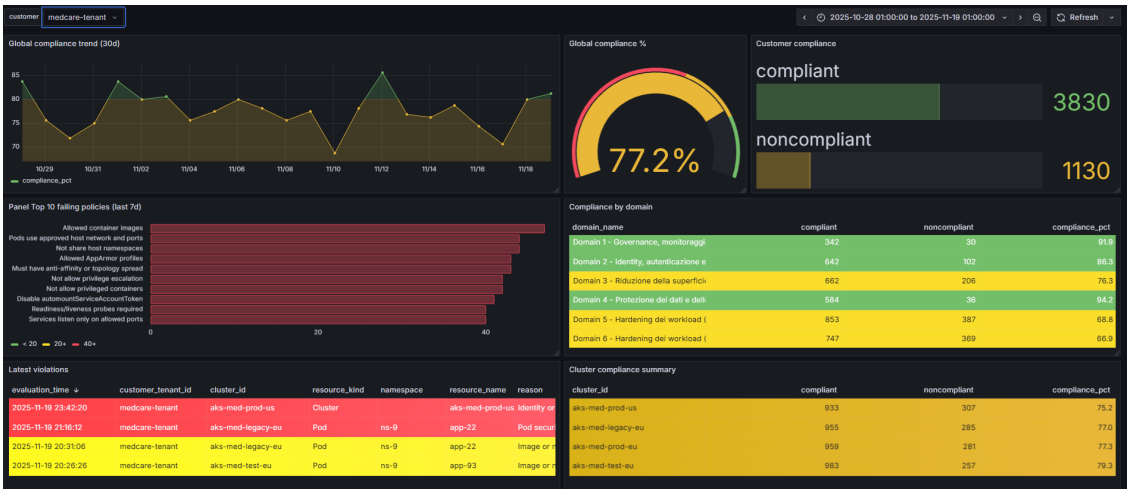

Figure 5.3. Customer Dashboard for a selected tenant (example: medcare-tenant), showing the tenant-specific trend, failing policies, compliance by domain, and latest violations.

The second dashboard type, the *Customer Dashboard*, focuses on a single tenant. A Grafana variable (`customer`) selects the tenant of interest, and all panels are recalculated using this filter.

As in the Overview Dashboard, the upper-left panel is the **Compliance trend (30d)**. For finbank-tenant (Figure 5.2), the curve oscillates around 80–85%, while for medcare-tenant (Figure 5.3) the trend reflects a slightly more heterogeneous environment.

The central gauge shows the tenant's **Global compliance %**. The **Customer compliance** panel expands this view by listing total compliant and noncompliant evaluations.

The **Top 10 failing policies (last 7d)** panel identifies the policies that generate the most violations for the selected tenant, often exposing differences between organizational processes or Kubernetes workload maturity across tenants.

The **Compliance by domain** table displays the six domains with compliant/noncompliant counts and a color-coded compliance percentage.

At the bottom, the **Latest violations** table lists the most recent issues for the selected tenant, while the **Cluster compliance summary** panel breaks down posture by AKS cluster, showing heterogeneous maturity levels or consistent deployment practices depending on the tenant.

Chapter 6

Tenant Client Architecture

6.1 Role of the Tenant Client

In the AKS security framework, the tenant client is the place where activity actually happens. This is where AKS clusters live, where policies are concretely applied, and where security and compliance evidence is collected. The tenant server defines and publishes the rules, but the tenant client is the component that enforces them on its clusters and measures how the real environment deviates from the desired state.

From a conceptual perspective, this creates a clear separation between the **governance plane** (tenant server) and the **operational plane** (tenant client). The tenant server does not have administrative permissions on client AKS clusters and does not perform any direct action on resources: it only exposes configuration-as-data such as baselines, initiatives, assignments, and manifests. The tenant client, on the other hand, is responsible for:

- securely reading the configuration published by the server;
- projecting it into its own domain (local subscriptions and AKS resources);
- collecting telemetry, logs, and security indicators;
- calculating and sending drift and compliance indicators back to the server.

This separation reflects a core principle of modern cloud architectures: keeping the governance plane and the operational plane clearly distinct to achieve scalability, control, and independence across administrative domains.

6.1.1 Operational Role of the Client in Relation to the Tenant Server

The tenant client can be viewed as an intelligent agent operating within its own perimeter while following centrally defined rules. In particular, it:

- receives the **global AKS security baseline** from the tenant server. This baseline is implemented through Azure Policy and initiatives and defines the minimum mandatory requirements for all clusters;
- applies local **tenant customizations** (custom baseline) and **cluster-specific policies**, for example to enforce or strengthen Network Policy, define specific ingress restrictions, or introduce additional controls on container images and registries;
- observes the actual behavior of clusters and resources using logs sent to Log Analytics, and the evaluations from Azure Policy, Azure Monitor, and, when enabled, Microsoft Defender for Cloud.

Operationally, the client is the execution arm of the framework. It is the only component that has the permissions required to create or update policy assignments on AKS clusters, interact with the cluster data plane, and access security logs. The tenant server remains intentionally unaware of these operational details.

This architectural choice reduces coupling between domains and makes it explicit that responsibility for policy enforcement lies with the tenant client, in accordance with the cloud shared responsibility model.

6.1.2 Dual Logical Flow: Pull of Policies and Push of Audit Data

The interaction between tenant client and tenant server can be described as the combination of two complementary logical flows.

Pull flow. The client periodically downloads the security configuration from the tenant server. An Azure Function authenticates securely to the server's storage using a dedicated Service Principal and OAuth2, through a Private Endpoint on the Azure backbone.

The Function reads:

- the main tenant manifest, which indicates the current version of the baseline, the custom baseline, and individual clusters;
- the detailed manifests (baseline, custom baseline, cluster), which contain references to initiatives, assignments, and policy definitions;
- the JSON files containing the actual policies.

Using the `configVersion` field in the manifests, the client detects differences between the local configuration and the one exposed by the server, downloading only modified files and applying updates through Azure Policy.

Push flow. The client sends audit information and compliance indicators to the tenant server. This task is executed by another Azure Function (or extended logic), which:

- reads from Log Analytics and from Azure Policy endpoints to determine compliance;
- sends this information to the tenant server through a private network channel implemented as an Azure Private Link Service.

This mechanism provides the tenant server with the metrics required for centralized dashboards, audit reports, and security alerts.

6.1.3 Security and Isolation Requirements in the Client Domain

To support this dual role, the tenant client must satisfy several security requirements.

Controlled connectivity. Connectivity to the tenant server must be tightly controlled. The use of Azure Private Endpoint and Private Link ensures that all traffic remains on the Microsoft private network.

Strong identity and access management.

- The Function that reads secrets from the client Key Vault uses a Managed Identity.
- The tenant server's Service Principal has only minimal permissions (e.g., *Storage Blob Data Reader* on the dedicated container).
- RBAC ensures that no human identity or other application can use the cross-tenant channel.

Internal isolation.

- A dedicated management VNet hosts the *Audit* subnet containing the Function and the Private Endpoints.
- NSG rules restrict outbound traffic to only required Azure services.
- AKS clusters reside in a separate VNet and cannot directly access the tenant server's storage.

All communication flows through the Function, which acts as an application proxy and single point of control.

6.2 Architectural Overview of the Tenant Client

The tenant client architecture represents the operational side of the framework and provides the context in which the policies defined by the tenant server are concretely applied to AKS clusters and where audit and compliance evidence is collected. Its design is based on a model that clearly separates the governance domain (tenant server) from the execution domain (tenant client), following the principle of separation of concerns.

6.2.1 Overview of the Resources

The tenant client hosts a set of PaaS and IaaS resources that work together to perform the full cycle of policy alignment, enforcement on AKS environments, and transmission of indicators back to the tenant server. The infrastructure is typically organized into two main areas: a **management domain**, built on PaaS resources, and an **application domain**, where one or more AKS clusters and their connected resources reside.

The key components are:

Azure Function. This Function periodically downloads policies from the tenant server and sends audit and compliance indicators back to the central domain. It is not publicly exposed and communicates only through Private Endpoints, following Microsoft's guidelines on private networking for serverless applications.

Tenant Client Key Vault. The Key Vault stores the secret of the Service Principal issued by the tenant server. It is also reachable only through a Private Endpoint. The Function retrieves the secret using its Managed Identity, without exposing credentials in clear text or configuration.

Local Storage Account. The tenant client contains a storage account that keeps the synchronized copy of the policies and manifests received from the server. As this store is part of the security chain, it has *Network Access disabled* and is reachable only through a Private Endpoint.

Private Endpoint to the Server's PLS. A Private Endpoint connects the tenant client to the Private Link Service (PLS) exposed by the tenant server. This is the channel through which the Function sends audit and compliance indicators. The PLS allows clients to reach an internal endpoint without using the public network.

Together, the Function, the Key Vault, the local storage account and the Private Endpoints form the core operational layer of the tenant client.

Parallel to the management domain, the tenant client hosts one or more **AKS clusters**, placed in separate operational VNets and subnets. The clusters consume the policies provided by the framework and return security, drift, and compliance

signals. This model enforces a clear separation between the policy orchestration layer (management network) and the application plane (workloads).

6.2.2 Relationship Between the Management Subscription and the Operational Subscription

Many enterprise environments adopt a separation between the subscription used for lifecycle and control tools and the subscription that hosts AKS clusters and application resources. This separation is useful for security, audit, and accountability.

In the implemented model, the **management subscription** hosts:

- the AKSSF Function,
- the management VNet,
- the Private Endpoints,
- the Key Vault and local storage account,
- identities and roles required to apply policies to clusters.

The **operational subscription** contains:

- AKS clusters,
- workload application resources,
- monitoring and security agents.

The Function, operating in the management domain, only has the minimal scopes required to apply policies through Azure Policy. It does not have visibility into application resources. This ensures that any defect or compromise of the management layer does not propagate to business workloads.

The relationship between the two subscriptions is governed through RBAC, minimal roles, and explicit assignments. The Function receives write access only to the scopes required for enforcement, while access to workload resources remains under the full responsibility of the tenant client.

6.2.3 End-to-End Flow: From Policy Distribution to Audit Collection

The tenant client's operational model consists of two complementary movements: from server to client, and from client to server.

Flow from the Tenant Server to the Client. The Function authenticates with OAuth2 using a Service Principal defined in the tenant server and protected in the local Key Vault. It accesses the server's storage account through a Private Endpoint, downloads manifests, identifies the desired state via the `configVersion` field, and updates the local storage account. It then applies or updates Azure Policy assignments on the tenant's AKS clusters.

Flow from the Tenant Client to the Server. The tenant client aggregates and synthesizes compliance results from AKS clusters and sends them to the tenant server through the Private Link Service exposed in the server domain.

The result is a fully private, secure, and declarative feedback loop: the tenant server publishes the desired state, the tenant client applies and measures it, and the results are returned to support centralized observability and higher-level security controls.

6.3 Network Domain of the Tenant Client

The network architecture of the tenant client forms the security foundation of the entire framework. All flows required for exchanging configurations and transmitting audit data to the tenant server converge here. The tenant client network is designed according to Zero Trust Networking and workload isolation principles: every PaaS service must be reachable only through private connections (Azure Private Link), and every application component must operate inside a controlled network domain.

The client network domain is therefore composed of a **dedicated Virtual Network**, an **audit/policy subnet**, **Network Security Groups**, **Private Endpoints to the tenant server**, **Private Endpoints to local services**, and supporting **private DNS zones** for all involved service names.

6.3.1 Management Virtual Network and Audit/Policy Subnet

At the center of the tenant client network domain is the Management Virtual Network, created specifically to isolate framework components from AKS clusters and user applications. This VNet is not shared with workload environments and does not host publicly exposed resources.

A dedicated subnet (e.g. *audit*, *policy*, or *management-core*) hosts three key resource types:

1. **The Azure Function** responsible for pulling and pushing policies and audit data.
2. **The Private Endpoints** used to securely reach remote and local services, such as the tenant server storage account, the client Key Vault, and the ingestion endpoint exposed through the tenant server's Private Link Service.

3. **The local storage account** (`aks-security-framework-client-<tenant-id>`), accessible only via Private Endpoint and used as the repository for updated policies and manifests.

This management subnet forms a tightly constrained domain, disconnected from the public network and isolated from the AKS cluster. As a result, any compromise in the workload plane cannot interfere with the governance plane. Interactions between the management VNet and operational VNets occur through Azure Resource Manager RBAC rather than direct network traffic.

6.3.2 Network Security Group of the Management Subnet

The audit subnet is protected by a **Network Security Group (NSG)** enforcing restrictive inbound and outbound rules. The Function requires no internet access, as all communications occur through Private Endpoints.

The NSG enforces three principles:

1. **Inbound traffic is fully blocked** except for flows originating from Private Endpoints. The Function is not reachable externally; management is done through the Azure Portal.
2. **Outbound traffic is restricted** to the IPs of local and remote Private Endpoints only, preventing unintended communication with the internet or unrelated services.
3. **No direct network flows toward the AKS cluster**. All cluster operations occur through Azure Policy, preventing role escalation or lateral movement.

This creates a domain where the network is not an attack vector: all critical services are reachable only through private backend IPs, without public exposure or complex routing.

6.3.3 Private Endpoint to the Tenant Server Storage

A key architectural element is the **Private Endpoint connecting the tenant client to the tenant server's storage account**. This allows the Function to read manifests and policy files via a private channel, without crossing the public internet. The connection request is initiated by the tenant client but must be **explicitly approved** by the tenant server.

Once approved, DNS resolutions for:

```
<storage-server>.blob.core.windows.net
```

resolve to a private IP inside the client VNet via the Azure Private Link DNS override mechanism. This ensures that:

- the Function accesses the server storage as if it were an internal service,
- no packet leaves the Microsoft backbone,
- the server may revoke access at any time.

This provides secure multi-tenancy without requiring VPNs, ExpressRoute, or complex firewalls.

6.3.4 Private Endpoints and DNS for Remaining Services

The tenant client uses multiple Private Endpoints to access internal and remote services needed by the framework.

Private Endpoint to the local Key Vault. Essential because the Key Vault has no public endpoint. The Function retrieves the Service Principal secret only through this private connection.

Private Endpoint to the server's Private Link Service. Used for sending audit and compliance indicators. The ingestion endpoint is exposed as a *custom* Private Link Service rather than a native PaaS resource. The tenant client creates a Private Endpoint receiving a private IP in the audit subnet; the Function sends traffic as if contacting an internal service.

Private DNS Zones. A Private DNS Zone ensures resolution for:

1. the server storage account (`privatelink.blob.core.windows.net`),
2. the client Key Vault (`privatelink.vaultcore.azure.net`),
3. the private HTTP service exposed via the Private Link Service.

The zone is linked to the audit subnet VNet, ensuring all DNS resolutions occur internally and preventing leakage to public DNS systems.

6.4 Local Policy Storage in the Tenant Client

The local storage account of the tenant client is the main connection point between the governance logic provided by the tenant server and the operational plane that applies configurations to AKS clusters. Unlike the tenant server storage, which serves only as a publishing layer, the client storage acts as a local working area where policies are replicated and consumed to manage the tenant's clusters.

From a technological perspective, the client uses **Azure Blob Storage** to host JSON files describing initiatives, assignments, policy definitions, and orchestration

manifests. Blob Storage aligns well with the static, declarative, file-based nature of these artefacts and scales without requiring rigid schemas. It also integrates natively with RBAC, data encryption, Private Endpoints, and high availability.

A dedicated logical container is used, for example:

`aks-security-framework-client-<tenant-id>/`

This container acts as the root of the tenant's entire security configuration: replicated baseline, custom baseline, per-cluster configurations, and manifests.

6.4.1 Structure of the Local Storage

Inside the `aks-security-framework-client-<tenant-id>` container, the structure follows a stable and predictable schema, independent of the number of clusters. It contains three main sections:

- a **replicated global baseline**,
- a **tenant-specific custom baseline**,
- a **per-cluster configuration area**.

This mirrors the conceptual separation between global requirements, tenant-level extensions, and cluster-specific specializations.

6.4.2 General Manifest

The general manifest provides the index for the entire configuration:

`manifest-<tenant-id>.json`

If its `configVersion` matches the version from the tenant server, no update is required. If it differs, the manifest indicates which subordinate manifests must be updated.

6.4.3 Replicated Global Baseline

The global baseline defined by the tenant server is replicated locally under:

```
baseline/
    manifest-baseline-<tenant-id>.json
    baseline-assignment-<tenant-id>.json
    baseline-initiative.json
    policy-definition/
        <policy-id-1>.json
        <policy-id-2>.json
        ...
```

The file `baseline-initiative.json` aggregates baseline policies. Custom policy definitions are stored locally; built-in policies are referenced by ID.

The file `baseline-assignment-<tenant-id>.json` describes how the baseline is assigned to AKS clusters. It is stored locally so that the Function can:

- verify consistency with the tenant server,
- reapply or realign assignments when needed.

The manifest `manifest-baseline-<tenant-id>.json` serves as the drift identifier and index for all baseline artefacts.

6.4.4 Tenant Custom Baseline

Above the global baseline, the tenant server defines a **custom baseline**, stored as:

```
custom-baseline/
    manifest-custom-baseline-<tenant-id>.json
    custom-baseline-initiative-<tenant-id>.json
    custom-baseline-assignment-<tenant-id>.json
    policy-definition/
        custom-baseline-<tenant-id>-<policy-id-1>.json
        ...
```

The initiative aggregates tenant-specific policies; the assignment describes how they are applied to clusters. The manifest provides the drift identifier and paths for updates.

6.4.5 Per-Cluster AKS Configurations

Each cluster has its own configuration directory:

```
clusters/
    <tenant-id>-<cluster-id>/
        manifest-<tenant-id>-<cluster-id>.json
        initiative-<tenant-id>-<cluster-id>.json
        assignment-<tenant-id>-<cluster-id>.json
        policy-definition/
            <tenant-id>-<cluster-id>-<policy-id-1>.json
            ...
```

These files describe cluster-specific requirements that should not be elevated to the tenant-wide baseline. Typical examples include stricter controls for internet-facing clusters or special requirements for regulated workloads.

The manifest always serves as the drift identifier for the cluster.

6.4.6 Relationship Between Local and Server Structures

The tenant client's structure is a consistent reflection of that published by the tenant server:

- the server exposes a global baseline container,
- each tenant has a dedicated area with its main manifest, baseline assignment, custom baseline, and per-cluster configurations.

Locally, the baseline is projected into the tenant namespace through:

```
manifest-baseline-<tenant-id>.json
```

The **main tenant manifest** downloaded from the server establishes the link. It tells the Function which secondary manifests must be consulted to determine the complete desired state. This design allows the tenant server structure to evolve (new clusters, updated baselines) without requiring logic changes in the Function.

6.4.7 Local Manifests and Desired State

Local manifests implement the **desired state** model for AKS security.

During synchronization, the Function:

1. compares local and remote `configVersion` values,
2. downloads updated manifests and JSON files if required,
3. updates the corresponding content locally,
4. applies or realigns policy assignments to AKS clusters.

The Function therefore operates in a declarative manner, aligning actual configuration with the desired state. The local Blob Storage becomes the materialized form of the tenant's desired security model: aligned with the tenant server but enriched with tenant-specific and cluster-specific elements.

6.5 Azure Key Vault of the Tenant Client and Secret Management

The Azure Key Vault of the tenant client is the central component responsible for protecting the credentials required to establish secure communication with the tenant server. In a multi-tenant and cross-directory context such as the AKS Security Framework, the Key Vault guarantees isolation of secrets, controlled access

through managed identities, and the complete absence of plaintext credentials in application components.

Integrated in the tenant client VNet through a Private Endpoint, the Key Vault is accessed only at execution time by the Azure Function, which retrieves the tokens and identifiers required for cross-tenant communication. No secret is ever exposed in application code, configuration files, or public networks.

6.5.1 Key Vault as the Central Point for Secret Management

Azure Key Vault is the recommended service for storing application secrets, certificates, and keys, providing hardware-assisted encryption, native auditing, and fine-grained access control through RBAC or access policies.

In the tenant client, the Key Vault stores only the secrets required for communication with the tenant server, specifically for OAuth2 authentication through a Service Principal.

The Function does not contain credential files nor permanent environment variables. At every execution, it dynamically retrieves the needed values via its Managed Identity. This avoids accidental exposure of secrets in repositories, logs, or configuration artefacts.

6.5.2 Secrets Related to Cross-Tenant Communication

To access the tenant server's storage account using OAuth2, the Key Vault stores five categories of values:

1. Tenant ID of the server Used to build the token request:

```
https://login.microsoftonline.com/<tenant-id>/oauth2/v2.0/token
```

2. Client ID of the Service Principal The public identifier of the application registered in the tenant server directory.

3. Client Secret The secret associated with the Service Principal, required for the `client_credentials` OAuth2 flow. It is stored securely and accessible only via the Function's Managed Identity.

4. Name of the tenant server storage account Used to compose the Blob Storage URL:

```
https://<storage-name>.blob.core.windows.net
```

5. Name of the tenant's dedicated container Each tenant has an isolated container governed by RBAC.

These values form the minimal set required to securely pull manifests, initiatives, and policies from the tenant server.

6.5.3 Key Vault Private Endpoint and VNet Integration

To ensure the Key Vault is never reachable on public endpoints, the tenant client configures a Private Endpoint in the management subnet.

The integration steps are:

1. Public network access is disabled on the Key Vault.
2. A Private Endpoint is created inside the management subnet.
3. A Private DNS record is added automatically:

   `<kv-name>.privatelink.vaultcore.azure.net → <private-ip>`

4. The Function, integrated with the same VNet, resolves and reaches the Key Vault internally.

This guarantees full isolation from public networks and prevents unauthorized access.

6.5.4 Accessing the Key Vault via Managed Identity

The Azure Function uses its system-assigned Managed Identity to authenticate to the Key Vault. The identity is granted the role:

```
Key Vault Secrets User
```

(or an equivalent RBAC role).

After configuration, the Function retrieves secrets by calling Key Vault endpoints through the Private Endpoint using the Azure Identity library, such as `DefaultAzureCredential` or `ManagedIdentityCredential`.

All communication occurs:

- without public internet traffic,
- without plaintext secrets,
- without SAS tokens,
- without unauthenticated flows.

6.5.5 Secret Rotation Model

Secret lifecycle management is crucial. If a Service Principal secret were compromised, an attacker could read tenant policies from the central server. For this reason, periodic rotation is recommended.

When the secret is rotated:

1. A new client secret is generated in the tenant server directory.
2. The updated secret is securely transmitted to the tenant client.
3. The value is updated in the client Key Vault.
4. The Function automatically uses the new secret at the next execution.

No code changes, redeployments, or configuration updates are required. This is a direct benefit of the *secret-less application* model enabled by Managed Identity.

6.6 Azure Functions in the Tenant Client

In the tenant client, the orchestration of all policy alignment and compliance collection tasks is delegated to a set of specialized Azure Functions. These functions are not generic automation components; they explicitly implement the two core flows of the framework:

1. **Pull of configurations** from the tenant server (policy synchronization)
2. **Push of audit and compliance indicators** to the tenant server (results export)

A separate auxiliary function provides *connectivity testing and diagnostics*, validating network configuration and identity setup.

All functions share key architectural foundations:

- execution inside a **Function App** integrated with the **management VNet** via regional VNet integration
- secure access to secrets through **Azure Key Vault** using the Function's **Managed Identity**
- communication with the **tenant server storage account** and with the **audit ingestion endpoint** exclusively through **Private Endpoints** and **Private Link Service**
- use of a **dedicated Service Principal** authorized via RBAC at the container level for remote storage access

In this architecture, Azure Functions act as the execution agent of the framework within the tenant client domain: they apply the configuration published by the tenant server and send back compliance results.

6.6.1 Policy Synchronization Function

The primary function, **AKSSF Policy Sync**, keeps the *local policy storage* of the tenant client aligned with the *central storage* of the tenant server. It is implemented as a **Timer Trigger**, executed periodically (e.g. every 15–30 minutes), consistent with Azure's guidance for scheduled serverless operations.

Although implemented in Python, its logical workflow is language agnostic and follows well-defined phases.

a) Reading Secrets from Key Vault

At startup, the Function uses its **Managed Identity** to authenticate to the tenant client Key Vault and retrieve:

- `AKSSF-SERVER-TENANT-ID`
- `AKSSF-SERVER-SP-CLIENT-ID` and `AKSSF-SERVER-SP-CLIENT-SECRET`
- `AKSSF-SERVER-STORAGE-ACCOUNT`
- `AKSSF-SERVER-TENANT-CONTAINER`
- additional operational parameters (e.g. local container names or feature flags)

This step uses the Azure Key Vault Secrets SDK, which authenticates through Azure AD without static credentials. Because the secrets are retrieved dynamically, rotating the Service Principal credential on the tenant server simply requires updating the Key Vault value.

b) Authenticating to the Tenant Server Storage

With the retrieved parameters, the Function initializes a `ClientSecretCredential` bound to the tenant server directory and creates a `BlobServiceClient` targeting the server storage account.

Two security layers apply:

1. **Authentication**: the Service Principal obtains an OAuth2 token for storage access.
2. **Authorization**: the Service Principal holds only the *Storage Blob Data Reader* role on its dedicated container, ensuring tenant isolation.

All traffic routes through a **Private Endpoint**, not the public internet, using private DNS resolution via `<storage>.privatelink.blob.core.windows.net`.

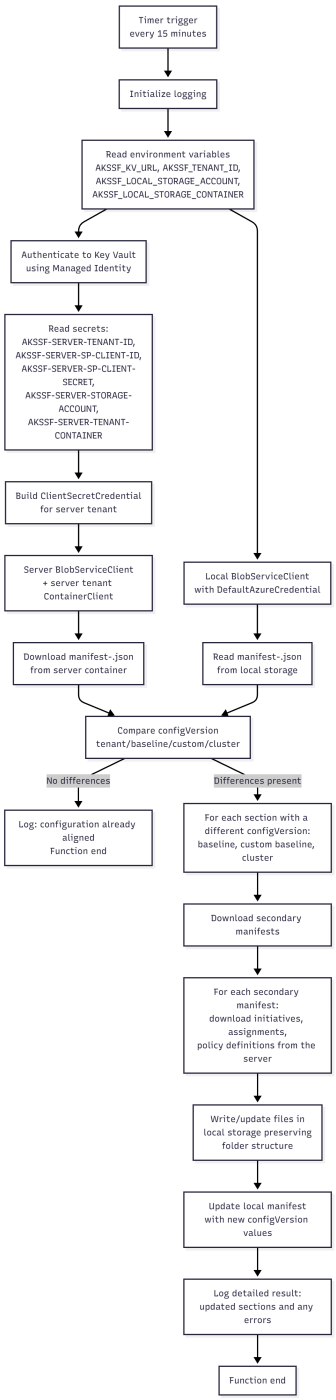

Figure 6.1. Synchronization Function Flow Chart

c) Reading the Main Manifest and Secondary Manifests

The Function downloads the tenant's *main manifest*:

`manifest-<tenant-id>.json`

This file provides:

- the tenant-level `configVersion`
- pointers to the global baseline manifest
- pointers to the tenant custom baseline manifest
- a list of cluster manifests with their versions

The Function then:

1. retrieves the global baseline manifest
2. retrieves the tenant custom baseline manifest
3. retrieves all cluster manifests

For each manifest, the Function compares the server's `configVersion` with the local one. On mismatch—or if the local version is missing—it proceeds to update the relevant content.

This implements a **desired state model**: the tenant server defines the desired configuration, and the client reconciles its local state accordingly.

d) Selective Download of Modified Artifacts

For each component whose `configVersion` has changed, the Function:

- downloads the referenced artefacts (initiatives, assignments, policy definitions)
- stores them in the tenant client's local storage, preserving the hierarchical structure
- updates the corresponding local manifest

The download is **selective**: unchanged artefacts are not fetched, minimizing execution time and network usage. The local storage therefore becomes a **coherent, versioned mirror** of the policy state published by the tenant server.

6.6.2 Audit Collection and Export Function

The second function, **AKSSF Audit Exporter**, completes the logical cycle of the framework by enabling the flow in the opposite direction: from the tenant client to the tenant server. While the Policy Sync function ensures that the client is aligned with the baseline, the Audit Exporter allows the tenant server to *observe and measure* this alignment.

This function is also typically implemented as a **Timer Trigger**, executed at regular intervals (e.g. every 30 or 60 minutes) to collect and consolidate audit and compliance information.

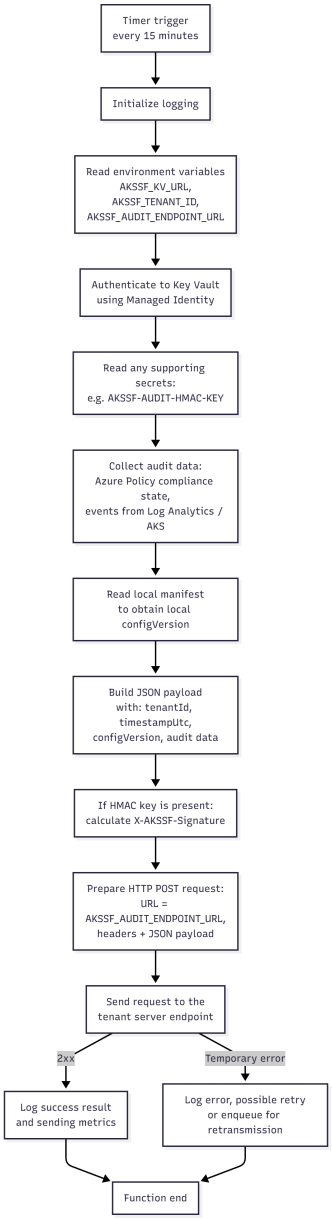

Figure 6.2. Audit Collection and Export Flow Chart

a) Role as a Compliance Agent in the Tenant Client

The function acts as a **local compliance agent**. It queries the available APIs and data sources inside the tenant client domain and builds a *structured JSON payload* to send to the tenant server.

The main data sources include:

- compliance state from Azure Policy (policies and initiatives applied to AKS clusters), obtained through **Azure Policy Insights / Policy States**
- logs and metrics from **Azure Monitor / Log Analytics** for AKS clusters and security components (less relevant in the reference implementation)

The function has complete visibility into the tenant client security perimeter while remaining isolated within its own domain.

b) Aggregation and Normalization of Audit Data

After collecting data from the various sources, the function performs a **normalization phase** in which it:

- converts compliance results into a concise form (e.g. number of non-compliant assignments, list of critical failed policies, summary indicators)
- enriches each record with metadata related to the tenant, the cluster, the policy version (`configVersion`), and the collection timestamp
- structures the final result into a JSON payload suitable for ingestion by the *collection service* running in the tenant server (e.g. an HTTP Function or a custom endpoint connected to Event Hub or Data Explorer)

This normalization allows the tenant server to **treat all tenants uniformly**, regardless of their AKS topology or policy set.

c) Secure Transmission Through Private Link Service

The payload is not sent to a public endpoint. Instead, it is delivered to a **private HTTP endpoint** exposed in the tenant server through **Azure Private Link Service**. From the tenant client perspective, the function calls an URL resolved by the **local Private DNS zone** into a private IP associated with the Private Endpoint linked to the Private Link Service.

This mechanism ensures that:

- audit traffic never crosses the public internet
- only tenants with an approved Private Endpoint can reach the ingestion endpoint
- any attempt to impersonate the client from unauthorized hosts is blocked by routing and authorization controls

The HTTP ingestion endpoint in the tenant server may additionally validate:

- the identity of the caller (e.g. via a token linked to the client Managed Identity or a dedicated telemetry secret)
- the consistency of the tenant ID and cluster identifiers contained in the payload
- the integrity of the message (e.g. through HMAC signatures or content hash verification)

Chapter 7

Conclusions

The work presented in this thesis started from a concrete operational problem faced by modern organizations that rely on AKS as a strategic platform for containerized workloads. In particular, the focus was on environments where a single organization manages multiple Azure tenants, each hosting one or more AKS clusters that often belong to distinct customers or independent business units. In these scenarios, security and governance requirements are typically strict, the need for isolation between tenants is non negotiable, and at the same time there is a strong demand for centralized visibility and standardization. Native Azure tools partially address these needs, but they tend to assume a shared management plane, and in some cases introduce trust relationships, like those required by Azure Lighthouse, that are difficult to accept in highly regulated or strongly segregated contexts. The central question of this thesis was therefore how to reconcile a rigorous separation of customer environments with the need to define, distribute, and continuously assess a common security standard for AKS clusters at scale.

To answer this question, the thesis first provided a structured analysis of the AKS ecosystem and of its main security challenges. The background chapter examined the internal architecture of AKS, the shared responsibility model between Microsoft and the customer, the main attack surfaces of Kubernetes, and the role of identity, network security, and workload hardening in a Zero Trust perspective. This analytical foundation was essential to understand that securing AKS cannot be reduced to a set of isolated best practices, but requires a systemic approach that covers governance, identity, network, data protection, and workloads in a coherent way. From this analysis emerged the need for a formalized security baseline capable of translating high level requirements and official recommendations into concrete, verifiable, and actionable controls that can be applied consistently across heterogeneous clusters.

Starting from this foundation, the thesis defined an AKS specific security baseline built entirely on Azure Policy for Kubernetes, used as the primary enforcement and measurement tool. The baseline was structured into logical domains that reflect the main dimensions of cluster security, such as governance and observability, identity and access management, network exposure and segmentation, protection of data and secrets, and workload hardening. Each domain groups built in policies that implement practices recommended by official documentation, community

benchmarks, and recognized security standards. Particular emphasis was given to the systematic use of Audit mode for all policies, so that the baseline could initially be adopted as a measurement and assessment instrument rather than as an enforcement mechanism. This choice allowed the baseline to become a tool for understanding the real security posture of each cluster, for identifying recurring misconfigurations, and for planning remediation and progressive hardening without introducing operational risk.

On top of this baseline, the thesis designed and implemented the AKS Security Framework (AKSSF), a complete Azure based architecture for the centralized management of the defined standard and for the continuous collection of compliance data from distributed tenants. The design is explicitly oriented toward multi tenant enterprise scenarios where each customer or business unit is hosted in a dedicated Azure tenant. The framework introduces a clear separation between two functional blocks. The tenant server acts as the governance domain: it publishes the global baseline, allows the definition of tenant specific extensions and per cluster configurations, receives compliance snapshots, and exposes visualization and analysis capabilities for the entire ecosystem. The tenant client, deployed in each tenant that hosts AKS clusters, acts as the operational execution point: it synchronizes policies from the central repository, applies them locally through Azure Policy and Gatekeeper, and periodically exports audit data to the tenant server.

A key contribution of the work is the declarative model used to represent the configuration state of the system. Baselines, tenant customizations, and cluster specific configurations are described through a hierarchy of JSON manifests stored in Azure Blob Storage. At the global level, a manifest describes the standard baseline shared by all clusters. Additional manifests define, for each tenant, the policies that extend the global standard, and further manifests describe the configuration of individual clusters. This approach allows a clear separation of concerns: the global baseline expresses the minimum common security level, tenant manifests represent organizational specific requirements, and cluster manifests capture local variations that are justified by the characteristics of individual environments. The manifest model is versioned, supports evolution of the configuration over time, and provides the basis for future extensions involving integrity checks and more advanced attestation mechanisms.

From an infrastructure perspective, the thesis shows how it is possible to combine Azure native services to build a secure and scalable framework without compromising tenant isolation. On the tenant server side, the policy repository is hosted in a storage account accessible only through Private Endpoints, protected by Azure RBAC with per tenant segregation at the container and path level. A second storage account acts as a centralized data lake that collects raw compliance snapshots from all tenants. An ingestion pipeline imports these snapshots into Azure Data Explorer, where they are expanded, normalized, and correlated into a logical data model that includes tables for snapshots, clusters, policy results, and violations. Materialized views compute key indicators such as global compliance scores, trends over time, per tenant posture, and the most frequently violated policies. Grafana, configured with a KQL data source, provides the visualization layer through dashboards that offer both a global overview and detailed per tenant or per cluster analysis.

On the tenant client side, the framework is deliberately lightweight but strongly segregated. Each client environment hosts a management virtual network with dedicated subnets for the audit and synchronization functions and for Private Endpoints to the tenant server. Network Security Groups restrict both inbound and outbound connectivity to the minimum necessary, and Azure Private Link ensures that traffic towards the tenant server never traverses the public internet. Credentials used for cross tenant communication are stored in an Azure Key Vault integrated into the same virtual network via Private Endpoint, and access to the vault is mediated through managed identities to avoid static secrets. The core of the client logic is implemented through Azure Functions: one function periodically synchronizes local storage with the manifests published by the tenant server, while another collects compliance data from Azure Policy Insights and Gatekeeper and pushes it to the central ingestion endpoint. This design ensures that the tenant server never requires direct access to client clusters or to their subscriptions; all interactions are initiated from within the tenant client domain, preserving the required security boundaries.

From the perspective of the initial problem, the implemented framework demonstrates that it is possible to obtain centralized visibility and governance over multiple, isolated AKS environments without introducing a central management plane that can modify or directly control customer resources. The AKSSF solution offers a single, consistent security baseline, adjustable at tenant and cluster level through declarative manifests, and a continuous feedback loop in which audit data is collected, analyzed, and visualized across the entire landscape. The dashboards built on Azure Data Explorer provide immediate insight into the adherence of each cluster to the baseline, highlight the most critical deviations, and make it easier to prioritize remediation efforts. At the same time, the architecture respects the separation between tenants by using only pull and push flows initiated from the client side and by relying on private connectivity and scoped access rights.

Despite these achievements, the work also has intrinsic limitations that define its scope and suggest future research directions. First, the baseline and the framework focus exclusively on native AKS clusters and do not cover Azure Arc enabled Kubernetes or other hybrid and multi cloud scenarios. Extending the model to those environments would require revisiting some assumptions about the policy engine, the available telemetry, and the management plane. Second, the baseline is intentionally limited to built in Azure policies and does not explore advanced scenarios involving custom OPA rules or complex policy compositions tailored to specific regulatory frameworks. Third, the framework operates entirely in Audit mode and does not implement automatic remediation or enforcement mechanisms; remediation remains a manual or external process, guided by the insights provided by the dashboards. Finally, the performance and scalability of the data model in Azure Data Explorer, while adequate for the reference environment used in this work, would need empirical evaluation and possible optimization in the presence of thousands of clusters and very high frequency of compliance snapshots.

Chapter 8

Future Work

This chapter outlines the most relevant areas of future work, focusing both on technical extensions of the current design and on improvements aimed at making the framework more usable and more deeply integrated into enterprise processes.

8.1 From Audit-Only to Progressive Enforcement and Remediation

The current implementation intentionally applies all Azure Policy definitions in *Audit* mode. This choice is aligned with the goal of understanding the existing security posture without risking disruptions to workloads that may not yet conform to the baseline. A natural next step is to move towards a progressive enforcement model in which selected controls, once validated and stabilized, are gradually switched to *Deny* or *DeployIfNotExists*.

Future work could define a structured roadmap for this transition, including:

- criteria for identifying mature clusters or tenants that are ready for enforcement (e.g., low violation rates, stable configuration, critical business impact),
- classification of policies according to their enforcement risk, starting with low-impact controls (such as diagnostic configuration) before moving to more intrusive ones (such as Pod Security restrictions),
- integration with remediation workflows so that violations can trigger automated or semi-automated corrections, rather than remaining purely observational signals.

By closing the loop between detection and enforcement, the framework would evolve from a posture monitoring system into a more comprehensive security control plane, while still preserving the flexibility needed in heterogeneous environments.

8.2 Extended Telemetry and Runtime Signal Integration

Another important axis of evolution is the integration of additional security signals beyond static configuration posture. While this thesis focuses on Azure Policy results and related compliance data, modern Kubernetes security also relies heavily on runtime protections and supply-chain integrity.

Future work could extend the ingestion pipeline and the underlying data model to include:

- alerts and recommendations from Microsoft Defender for Containers, such as anomalous process behavior, suspicious network activity, or vulnerability findings on running workloads;
- image-scanning results from Azure Container Registry or external scanners, including CVE status, base image provenance, and adherence to organizational image registries;
- evidence related to software supply chain, such as attestations, signature verification outcomes, or SBOM-based checks.

These additional data sources could be correlated with policy violations to provide a richer view of risk. For instance, repeated violations of image-related policies combined with high-severity vulnerabilities on the same images would highlight clusters or tenants requiring immediate attention. The dashboards could be extended to surface these combined indicators and to support prioritization of remediation efforts.

8.3 Policy Maker: User-Friendly, Granular Policy Authoring

One of the most promising directions for future work is the introduction of a *Policy Maker* component: a user-friendly layer that sits on top of the current manifest and Azure Policy model and allows security teams to design, refine, and apply policies in a more intuitive way. While the present framework assumes that policies and initiatives are authored directly as JSON artifacts, this approach can be complex and error-prone, especially when dealing with large numbers of rules or with highly granular configurations.

The Policy Maker would aim to:

- provide a high-level, intent-based interface for expressing security requirements (e.g., “this tenant’s internal services must not be reachable from other tenants” or “developers in team X can only administer namespaces A and B”);
- automatically translate these intents into concrete artifacts:

  - Azure Policy definitions and assignments for AKS clusters,
  - Kubernetes `NetworkPolicy` objects for fine-grained traffic segmentation,
  - RBAC roles and bindings that enforce least privilege at user, group, and service account level.

- manage policy variants at different scopes (global baseline, per-tenant customizations, per-cluster overrides) without requiring manual duplication of JSON files;
- offer policy simulation and impact analysis, showing which clusters, namespaces, or users would be affected by a proposed change before it is actually applied.

In particular, network segmentation and RBAC are areas where the need for granularity is especially strong. Today, designing a coherent set of `NetworkPolicy` resources across multiple namespaces and tenants requires deep Kubernetes expertise and careful manual coordination. Similarly, crafting RBAC roles that are both sufficiently permissive for operational needs and sufficiently restrictive for security is a complex task. A Policy Maker tool could guide administrators through templates, wizards, and visualizations, reducing the risk of misconfiguration and making advanced hardening strategies accessible to a wider audience.

From an architectural perspective, the Policy Maker would likely interact with the Tenant Server's manifest repository, generating or updating manifests and policy artifacts while respecting the versioning model already in place. This would preserve the declarative nature of the framework while significantly improving its usability.

8.4 Automated Onboarding of New Tenants and Clusters

Another area where the framework can be extended is the automation of onboarding for new tenants and new AKS clusters. In the current implementation, the steps required to integrate a new tenant or cluster into the AKSSF architecture are well-defined but still involve several manual or semi-manual actions, such as:

- provisioning the storage container and manifest structure for the new tenant on the Tenant Server;
- creating the Service Principal and associated RBAC assignments needed for cross-tenant access to storage;
- configuring the Tenant Client's network (Virtual Network, subnets, Private Endpoints, Private DNS zones) and the corresponding Private Link Service connections;
- deploying the local components (Function Apps, storage accounts, Key Vault) and wiring them into the existing manifest and synchronization process.

Future work could transform this sequence into a standardized, fully automated onboarding pipeline. Possible directions include:

- defining an Infrastructure-as-Code (IaC) template (for example using Bicep or Terraform) that encapsulates all the required resources and configurations for both the Tenant Server and the Tenant Client side;
- integrating the onboarding flow with a self-service portal or ticketing system, where a request for a new tenant or new cluster triggers the execution of the IaC pipeline and automatically registers the new environment in the manifest hierarchy;
- generating initial policy assignments and manifests for the new tenant based on predefined profiles (e.g., "development", "standard production", "highly regulated"), which can then be refined through the Policy Maker component;
- automatically hooking the new tenant's data sources into the ingestion pipeline and dashboards, so that posture visibility becomes available as soon as the first compliance snapshots are produced.

Such automation would not only reduce the operational overhead and the risk of misconfiguration during onboarding, but also enforce consistency from the very first day a tenant or cluster is brought under governance. In large organizations with frequent onboarding events, this could be a decisive factor for the practical adoption of the framework.

Bibliography

[1] M. Learn, "What is azure kubernetes service (aks)," https://learn.microsoft.com/en-us/azure/aks/what-is-aks, 2024.

[2] ——, "Aks core concepts: Azure managed control plane," https://learn.microsoft.com/en-us/azure/aks/core-aks-concepts#control-plane, 2024.

[3] ——, "Aks architecture and workloads," https://learn.microsoft.com/azure/aks/concepts-clusters-workloads, 2024.

[4] ——, "Use multiple node pools in aks," https://learn.microsoft.com/it-it/azure/aks/use-system-pools?source=recommendations&tabs=azure-cli, 2024.

[5] K. Documentation, "Pods: Kubernetes concepts," https://kubernetes.io/docs/concepts/workloads/pods, 2024.

[6] ——, "Kubernetes deployments," https://kubernetes.io/docs/concepts/workloads/controllers/deployment, 2024.

[7] ——, "Kubernetes namespaces," https://kubernetes.io/docs/concepts/overview/working-with-objects/namespaces, 2024.

[8] M. Learn, "Aks node resource group architecture," https://learn.microsoft.com/azure/aks/faq?tabs=azure-cli#what-is-the-aks-node-resource-group, 2024.

[9] ——, "Azure cni overlay," https://learn.microsoft.com/azure/aks/azure-cni-overlay, 2024.

[10] ——, "Aks networking concepts," https://learn.microsoft.com/azure/aks/concepts-network, 2024.

[11] ——, "Azure cni powered by cilium," https://learn.microsoft.com/azure/aks/azure-cni-powered-by-cilium, 2023.

[12] C. Project, "Cilium documentation," https://docs.cilium.io/en/stable, 2024.

[13] K. Documentation, "Kubernetes network plugins," https://kubernetes.io/docs/concepts/extend-kubernetes/compute-storage-net/network-plugins, 2024.

[14] ——, "Kubernetes networking concepts," https://kubernetes.io/docs/concepts/cluster-administration/networking, 2024.

[15] M. Learn, "How kubernetes networking works in aks," https://learn.microsoft.com/azure/aks/concepts-network#how-kubernetes-networking-works, 2024.

[16] ——, "Limit and control aks egress traffic," https://learn.microsoft.com/azure/aks/limit-egress-traffic, 2024.

[17] M. Azure, "Network security groups overview," https://learn.microsoft.com/azure/virtual-network/network-security-groups-overview, 2025.

[18] K. Documentation, "Kubernetes network policies," https://kubernetes.io/docs/concepts/services-networking/network-policies, 2024.

[19] M. Learn, "Use network policies in aks," https://learn.microsoft.com/azure/aks/use-network-policies, 2024.
[20] NIST, "Nist sp 800-190: Application container security guide," https://csrc.nist.gov/publications/detail/sp/800-190/final, 2017.
[21] MITRE, "Mitre att&ck for containers," https://attack.mitre.org/matrices/enterprise/containers, 2025.
[22] K. Documentation, "Rbac authorization," https://kubernetes.io/docs/reference/access-authn-authz/rbac, 2024.
[23] CNCF, "Cloud native security whitepaper 2023–2024," https://github.com/cncf/tag-security/blob/main/community/resources/security-whitepaper/v2/CNCF_cloud-native-security-whitepaper-May2022-v2.pdf, 2024.
[24] K. Documentation, "Encrypting secret data at rest," https://kubernetes.io/docs/tasks/administer-cluster/encrypt-data, 2024.
[25] M. Learn, "Use azure key management service (kms) with aks," https://learn.microsoft.com/en-us/azure/aks/use-kms-etcd-encryption?tabs=rbac-kv&pivots=public-kv, 2024.
[26] K. Documentation, "Pod security standards," https://kubernetes.io/docs/concepts/security/pod-security-standards, 2025.
[27] M. Learn, "Workload identity overview," https://learn.microsoft.com/azure/aks/workload-identity-overview, 2024.
[28] C. . O. P. Agent, "Opa gatekeeper for kubernetes," https://www.openpolicyagent.org/docs/latest/kubernetes-introduction, 2024.
[29] M. Learn, "Azure policy for kubernetes," https://learn.microsoft.com/azure/governance/policy/concepts/policy-for-kubernetes, 2024.
[30] ——, "Microsoft defender for containers," https://learn.microsoft.com/azure/defender-for-cloud/defender-for-containers-introduction, 2024.
[31] ——, "Csi secret store driver for aks," https://learn.microsoft.com/azure/aks/csi-secrets-store-driver, 2024.
[32] S. Storage, "Secrets store csi driver documentation," https://secrets-store-csi-driver.sigs.k8s.io, 2024.
[33] M. Learn, "Azure disk encryption overview," https://learn.microsoft.com/azure/virtual-machines/disk-encryption, 2024.
[34] ——, "Use customer-managed keys with aks disks," https://learn.microsoft.com/azure/aks/azure-disk-customer-managed-keys, 2024.
[35] ——, "Api server authorized ip ranges," https://learn.microsoft.com/azure/aks/api-server-authorized-ip-ranges, 2025.
[36] ——, "Monitor aks clusters," https://learn.microsoft.com/azure/aks/monitor-aks, 2024.
[37] K. Documentation, "Kubernetes audit logging guide," https://kubernetes.io/docs/tasks/debug/debug-cluster/audit, 2024.
[38] M. Learn, "Node image upgrade in aks," https://learn.microsoft.com/azure/aks/node-image-upgrade, 2024.
[39] ——, "Aks node auto-upgrade for os images," https://learn.microsoft.com/azure/aks/auto-upgrade-node-image, 2024.
[40] ——, "Disable ssh access in aks," https://learn.microsoft.com/azure/aks/ssh, 2024.
[41] ——, "Use command invoke in aks," https://learn.microsoft.com/azure/aks/use-command-invoke, 2024.

[42] K. Documentation, "Seccomp profiles for kubernetes," https://kubernetes.io/docs/tutorials/security/seccomp, 2024.
[43] ——, "Apparmor in kubernetes," https://kubernetes.io/docs/tutorials/security/apparmor, 2024.
[44] ——, "Pod security standards - baseline," https://kubernetes.io/docs/concepts/security/pod-security-standards/#baseline, 2024.
[45] ——, "Sysctl cluster management," https://kubernetes.io/docs/tasks/administer-cluster/sysctl-cluster/, 2024.
[46] ——, "Configure liveness, readiness, and startup probes," https://kubernetes.io/docs/tasks/configure-pod-container/configure-liveness-readiness-startup-probes/, 2024.
[47] ——, "Topology spread constraints," https://kubernetes.io/docs/concepts/scheduling-eviction/topology-spread-constraints/, 2024.
[48] ——, "Pod antiaffinity," https://kubernetes.io/docs/concepts/scheduling-eviction/assign-pod-node/, 2024.
[49] ——, "Service account token projection," https://kubernetes.io/docs/tasks/configure-pod-container/configure-service-account/#service-account-token-volume-projection, 2024.
[50] ——, "Ingress," https://kubernetes.io/docs/concepts/services-networking/ingress/, 2024.
[51] M. Learn, "Use an internal load balancer with aks," https://learn.microsoft.com/azure/aks/internal-lb, 2025.
[52] ——, "Use managed identity with aks," https://learn.microsoft.com/azure/aks/use-managed-identity, 2024.
[53] ——, "Private clusters in aks," https://learn.microsoft.com/azure/aks/private-clusters, 2025.
[54] "Azure policy for aks - built-in policy reference," https://learn.microsoft.com/azure/aks/policy-reference, Microsoft Learn, 2024.
[55] "Cis kubernetes benchmark guidance for aks," https://learn.microsoft.com/azure/aks/cis-kubernetes, Microsoft Learn, 2024.
[56] "Best practices for azure kubernetes service (aks)," https://learn.microsoft.com/azure/aks/best-practices, Microsoft Learn, 2024.
[57] "Use azure policy with aks," https://docs.azure.cn/en-us/aks/use-azure-policy, Microsoft, 2024.
[58] "Azure policy for kubernetes," https://learn.microsoft.com/azure/governance/policy/concepts/policy-for-kubernetes, Microsoft Learn, 2024.
[59] "Microsoft defender for containers," https://learn.microsoft.com/azure/defender-for-cloud/defender-for-containers-introduction, Microsoft Learn, 2024.
[60] "Aks operator best practices - identity," https://learn.microsoft.com/azure/aks/operator-best-practices-identity, Microsoft Learn, 2024.
[61] "Role-based access control (rbac)," https://kubernetes.io/docs/reference/access-authn-authz/rbac/, Kubernetes Documentation, 2024.
[62] "Private clusters in azure kubernetes service," https://learn.microsoft.com/azure/aks/private-clusters, Microsoft Learn, 2024.
[63] "Cve-2020-8554 kubernetes vulnerability," https://nvd.nist.gov/vuln/detail/CVE-2020-8554, NIST NVD, 2020.

[64] "Use csi storage drivers in aks," https://learn.microsoft.com/azure/aks/csi-storage-drivers, Microsoft Learn, 2024.

[65] "Seccomp profiles for kubernetes," https://kubernetes.io/docs/tutorials/security/seccomp/, Kubernetes Documentation, 2024.

[66] "Apparmor support in kubernetes," https://kubernetes.io/docs/tutorials/security/apparmor/, Kubernetes Documentation, 2024.

[67] Krawczyk, Hugo, Bellare, Mihir, and Canetti, Ran. (1997) Rfc 2104: Hmac — keyed-hashing for message authentication. IETF. [Online]. Available: https://datatracker.ietf.org/doc/html/rfc2104