

Security requirement

Orchestrator

Deutsche Telekom Group

Version	5.0
Date	Jul 1, 2025
Status	Released

Publication Details

Published by
Deutsche Telekom AG
Vorstandsbereich Technology & Innovation
Chief Security Officer

Reuterstrasse 65, 53315 Bonn
Germany

File name	Document number 3.34	Document type Security requirement
Version 5.0	State Jul 1, 2025	Status Released
Contact Telekom Security psa.telekom.de	Validity Jul 1, 2025 - Jun 30, 2030	Released by Stefan Pütz, Leiter SEC-T-TST

Summary
This document describes the functional security requirements used to secure Orchestrators additional to the CIS Benchmark and hardening guides which also cover implementation-oriented requirements

Copyright © 2025 by Deutsche Telekom AG.
All rights reserved.

Table of Contents

1.	Introduction	4
2.	Requirements	5
2.1.	Hardening	5
2.1.1.	Control plane hardening	6
2.1.2.	Worker Nodes	8
2.2.	Secrets	9
2.3.	Network	11
2.4.	Authentication and authorization	14
2.5.	Audit Logging / Threat Detection	15
2.6.	Policies	16
2.7.	Deployment/Application Level Security	18
3.	Private Clouds	24
3.1.	Cloud Infrastructure Network and Cloud Management	24
3.1.1.	Around Cloud Management	24
3.2.	Cloud Usage	25
3.2.1.	Separation of Cloud Infrastructure Depending on the Purpose	25

1. Introduction

This security document has been prepared based on the general security policies of the Group.

The security requirement is used as a basis for an approval in the PSA process, among other things. It also serves as an implementation standard for units which do not participate in the PSA process. These requirements shall be taken into account from the very beginning, including during the planning and decision-making processes. When implementing these security requirements, the precedence of national, international and supranational law shall be observed.

If compliance with the described requirements can not be achieved or is only partially feasible in individual cases, risk assessments must be carried out together with a Security and/or Data Privacy Expert (in accordance with the relevant requirements) and possible alternative protective measures must be agreed.

2. Requirements

2.1. Hardening

There may be shared responsibility depending on the type of Cluster.

Self-Managed Kubernetes Cluster Admins have full control over all components.

Managed Kubernetes (EKS, GKE, AKS, 3rd Party, Telekom Managed.) Cluster Admins have limited or no control over control plane components (API Server, Controller Manager, etc.)

Only the cloud provider/provider manages the control plane. Cluster Admins can modify worker nodes and some configurations like CoreDNS, kube-proxy, and networking policies.

Worker Node Components (e.g., Kubelet, Container Runtime Interface)

In self-managed clusters, admins have full access. In cloud-managed/managed clusters, access is limited (e.g., AWS restricts kubelet config changes on EKS managed nodes).

Breakdown of Kubernetes Components and Access Control (may vary between providers - this as an example)

Component	Self-Managed Kubernetes (Cluster Admins)	Managed Kubernetes (AKS, GKE, EKS, 3rd Party, Telekom Managed)
API Server	Full control	No access
Controller Manager	Full control	No access
Cloud Controller Manager	Full control	No access
Scheduler	Full control	No access
etcd (Cluster Database)	Full control	No access (except limited access via snapshots in some clouds)
CoreDNS, Other DNS	Full control	Limited control (can not replace, but alter config)
kube-proxy	Full control	Limited control (can not replace, but alter config)
Kubelet (node agent)	Full control	Limited control (provider manages OS and kubelet settings, changing certain flags may be possible , e.g. Node taints/tolerations)
Container Runtime	Full control	Limited control (may be able to configure Runtime related setting such as resource limits)

Req 1 There must be a patch management process for the whole cloud environment.

Patch management process in a cloud environment must be implemented in a way that allows patching of all cloud components while keeping given SLA to tenants. If SLA requires so, the patching process must be designed to handle live migration of virtual machines for patching of compute nodes, or to provide cloud management APIs available during patching, so that tenants are able to reschedule workloads which were impacted by the update procedure. With different type of SLA, one might also opt to update whole cloud environment at once and restart all workloads.

All components of the cloud environment must be included in the patch management process, e.g. management, orchestration, compute, storage and networking functions. All supporting systems (monitoring, logging, jump hosts etc.) need to be covered in the procedure as well.

Even in such cases where patching is needed on short notice, the regular testing procedure must be followed.

Motivation: Since new security vulnerabilities emerge daily, it is necessary that patching process is designed so it can be carried out quickly.

Implementation example: Patching procedure should be create so that the patching of all compute nodes can be finished within 24-48 hours after critical security issue has been disclosed and patch became available.

For this requirement the following threats are relevant:

- Unauthorized access to the system
- Unauthorized access or tapping of data
- Unauthorized modification of data
- Unauthorized use of services or resources
- Disruption of availability
- Denial of executed activities
- Unnoticeable feasible attacks
- Attacks motivated and facilitated by information disclosure or visible security weaknesses

ID: 3.86-17/2.0

2.1.1. Control plane hardening

The following logical components make up the control plane and must be hardened:

- Controller manger
- Cloud controler manager
- API
- Etc
- Scheduler

Req 2 Control Plane node configuration Files must be set to most restrictive permissions.

These recommendations may not be directly applicable for cluster operators in environments where these components are managed by a 3rd party. This Requirement can only be fulfilled be the Cluster Operator/Admin.

Ensure that the API server pod specification file has permissions of `600` or more restrictive.
Ensure that the API server pod specification file ownership is set to `root:root`.
Ensure that the controller manager pod specification file has permissions of `600` or more restrictive.
Ensure that the controller manager pod specification file ownership is set to `root:root`.
Ensure that the scheduler pod specification file has permissions of `600` or more restrictive.
Ensure that the scheduler pod specification file ownership is set to `root:root`.
Ensure that the `/etc/kubernetes/manifests/etcd.yaml` file has permissions of `600`
Ensure that the `/etc/kubernetes/manifests/etcd.yaml` file ownership is set to `root:root`.
Ensure that the Container Network Interface files have permissions of `600` or more restrictive.
Ensure that the Container Network Interface files have ownership set to `root:root`.
Ensure that the etcd data directory has permissions of `700`.
Ensure that the etcd data directory ownership is set to `etcd:etcd`.
Ensure that the `admin.conf` file has permissions of `600`.
Ensure that the `admin.conf` file ownership is set to `root:root`.
Ensure that the `scheduler.conf` file has permissions of `600`.
Ensure that the `scheduler.conf` file ownership is set to `root:root`.
Ensure that the `controller-manager.conf` file has permissions of 600.
Ensure that the `controller-manager.conf` file ownership is set to `root:root`.
Ensure that the Kubernetes PKI directory and file ownership is set to `root:root`.
Ensure that Kubernetes PKI certificate files have permissions of `600` or more restrictive.
Ensure that Kubernetes PKI key files have permissions of `600`.

Motivation: Ensuring the most restrictive permissions for Control Plane node configuration files minimizes the risk of unauthorized access or tampering, safeguarding the cluster's core functionality.

Req 3 API server must be hardened.

In a typical Orchestration Environment Cluster Admins can change the API Configuration.

They manage the Kubernetes control plane, including the API Server.

They have direct access to modify API Server configurations (e.g., flags, security settings, admission controllers).

In **managed Kubernetes (EKS, AKS, GKE, 3rd Party etc.)**, the provider controls API Server configuration, limiting what admins can change.

API server configuration flags for hardening that must be set :

Disable anonymous requests to the API server.

Do not use token based authentication.

Enable certificate based kubelet authentication.

Verify kubelet's certificate before establishing connection.

Restrict kubelet nodes to reading only objects associated with them.

Turn on Role Based Access Control.

Limit the rate at which the API server accepts requests.

Always pull images.

Enable auditing on the Kubernetes API Server and set the desired audit log path. Retain the logs for at least 30 days or as appropriate.

Set global request timeout for API server requests as appropriate.

Validate service account before validating token.

Explicitly set a service account public key file for service accounts on the apiserver.

Etcd should be configured to make use of TLS encryption for client connections.

Setup TLS connection on the API server.

Etcd should be configured to make use of TLS encryption for client connections.

Encrypt etcd key-value store.

Where `etcd` encryption is used, appropriate providers should be configured.

Motivation: Hardening the API server strengthens its defenses against potential attacks, ensuring the security and stability of the Kubernetes cluster.

Req 4 Controller Manager / Cloud Controller Manager must be hardened.

Cluster Provider/Cluster Admins are responsible for fulfilling this Requirement.

In **managed Kubernetes (EKS, GKE, AKS, 3rd Party)**, these components are managed by the provider, and admins may have limited or no access to change them.

Controller Manager configuration flags that must be set:

Activate garbage collector on pod termination, as appropriate.

Disable profiling, if not needed.

Use individual service account credentials for each controller.

Explicitly set a service account private key file for service accounts on the controller manager.

Allow pods to verify the API server's serving certificate before establishing connections.

Enable kubelet server certificate rotation on controller-manager.

Do not bind the Controller Manager service to non-loopback insecure addresses.

Motivation: Hardening the Controller Manager and Cloud Controller Manager enhances their resilience against attacks, ensuring secure orchestration of Kubernetes resources.

Req 5 Scheduler must be hardened

Cluster Provider/Cluster Admins are responsible for fulfilling this Requirement.

In **managed Kubernetes (EKS, GKE, AKS, 3rd Party)**, these components are managed by the provider, and admins

may have limited or no access to change them.

Scheduler configuration flags:

Disable profiling, if not needed.

Do not bind the scheduler service to non-loopback insecure addresses.

Motivation: Hardening the Scheduler strengthens its security posture, ensuring reliable and secure scheduling of workloads within the Kubernetes cluster.

ID: 3.34-5/5.0

Req 6 Etcd must be hardened.

Cluster Provider/Cluster Admins are responsible for fulfilling this Requirement.

In **managed Kubernetes (EKS, GKE, AKS, 3rd Party)**, these components are managed by the provider, and admins may have limited or no access to change them.

Etcd stores state information and cluster Secrets. Etcd must be encrypted a Rest.

Secure the etcd datastore with authentication and RBAC policies.

Configure TLS encryption for the etcd service.

Enable client authentication on etcd service. Do not use self-signed certificates for TLS. etcd should be configured to make use of TLS encryption for peer connections.

Etcd should be configured for peer authentication.

Do not use automatically generated self-signed certificates for TLS connections between peers.

Motivation: Hardening Etcd ensures the integrity and confidentiality of cluster data, protecting it from unauthorized access and potential breaches.

ID: 3.34-6/5.0

Req 7 Communication between Orchestrator components must be encrypted.

Encryption should be used for all traffic in the Kubernetes cluster— including between components, nodes, and the control plane

TLS must be enabled for every component that supports it to prevent traffic sniffing, verify the identity of the server, and (for mutual TLS) verify the identity of the client.

3_50_Cryptographic Algorithms and Security Protocols Document, TLS must be used in version 1.2 or 1.3 .

Motivation: Provide [privacy](#) and [data integrity](#) between two or more communicating computer applications.

Implementation example: Orchestrator components must use TLS to communicate.

For this requirement the following threats are relevant:

- Unauthorized access or tapping of data
- Unauthorized modification of data

ID: 3.34-7/5.0

2.1.2. Worker Nodes

Req 8 Worker Node file permissions must be set restrictive.

Cluster Provider/Cluster Admins are responsible for fulfilling this Requirement.

In **managed Kubernetes (EKS, GKE, AKS, 3rd Party)**, these components are managed by the provider, and admins may have limited or no access to change them.

Configuration files on the worker nodes:

Ensure that the kubelet service file permissions are set to 600.

Ensure that the kubelet service file ownership is set to root:root.
 If proxy kubeconfig file exists ensure permissions are set to 600.
 If proxy kubeconfig file exists ensure ownership is set to root:root.
 Ensure that the --kubeconfig kubelet.conf file permissions are set to 600.
 Ensure that the --kubeconfig kubelet.conf file ownership is set to root:root.
 Ensure that the certificate authorities file permissions are set to 600 .
 Ensure that the client certificate authorities file ownership is set to root:root.
 If the kubelet config.yaml configuration file is being used validate permissions set to 600 .
 If the kubelet config.yaml configuration file is being used validate file ownership is set to root:root.

Motivation: Setting restrictive file permissions on Worker Nodes prevents unauthorized access and ensures the security of workloads and sensitive data.

ID: 3.34-8/5.0

Req 9 Kubelet must be hardened.

Cluster Provider/Cluster Admins are responsible for fulfilling this Requirement.

In **managed Kubernetes (EKS, GKE, AKS, 3rd Party)**, these components are managed by the provider, and admins may have limited or no access to change them.

Kubelet configuration hardening:

Ensure that the --anonymous-auth argument is set to false.
 Ensure that the --authorization-mode argument is not set to AlwaysAllow.
 Ensure that the --client-ca-file argument is set as appropriate.
 Verify that the --read-only-port argument is set to 0.
 Ensure that the --streaming-connection-idle-timeout argument is not set to 0.
 Ensure that the --make-iptables-util-chains argument is set to true.
 Ensure that the --hostname-override argument is not set.
 Ensure that the eventRecordQPS argument is set to a level which ensures appropriate event capture.
 Ensure that the --tls-cert-file and --tls-private-key-file arguments are set as appropriate.
 Ensure that the --rotate-certificates argument is not set to false.
 Verify that the RotateKubeletServerCertificate argument is set to true.
 Ensure that a limit is set on pod PIDs.

Motivation: Hardening the Kubelet ensures secure communication and control over node operations, protecting the cluster from potential vulnerabilities.

ID: 3.34-9/5.0

Req 10 Management and worker nodes must be separated on different hosts.

An Orchestrator must be setup containing at least one master node and two worker nodes each on dedicated hosts (either physical or virtual machine).

Motivation: Master nodes are critical to the operation of the cluster. If no masters are running, or the master nodes are unable to reach a quorum, then the cluster is unable to schedule and execute applications, although the workers can run without a master node.

For this requirement the following threats are relevant:

- Unauthorized access to the system
- Unauthorized access or tapping of data
- Unauthorized modification of data
- Unauthorized use of services or resources

ID: 3.34-10/5.0

2.2. Secrets

Each tenant is obliged to properly use network protection mechanisms and secret management functions according to available Kubernetes platform technologies.

These are two most common deployment scenarios for network protection services in Kubernetes environments:

- Network protection services are provided by the Kubernetes platform (such as Network Policies, Service Meshes, or CNI plugins with advanced security features) and the tenant configures network protection in an appropriate way for their applications. This is the preferred method of protection.
- Network protection services are not provided by the Kubernetes platform or they are provided but the tenant cannot use them. In this case, the tenant should deploy dedicated nodes for running network protection mechanisms (such as dedicated security pods or DaemonSets running network security tools), following the assessment of the responsible Platform Security Manager.

In a similar way, tenants in Kubernetes typically require secret management (e.g., keys, passwords, certificates) and other management and support functions (e.g., configuration management) available on demand to support their workloads. Therefore, there are two most common deployment scenarios:

- Secret management is provided by the Kubernetes platform (using native Kubernetes Secrets, external secret stores like HashiCorp Vault integrated with Kubernetes, or Kubernetes Secrets Store CSI Driver) and the tenant uses them appropriately for their applications. This is the preferred method (this also applies where management services are available separately - not directly part of the Kubernetes platform but can be used through operators or custom resources).
- Secret management and additional management functions are not provided by the Kubernetes platform or are provided but the tenant cannot use them. In this case, the tenant needs to deploy dedicated nodes for running secret management and other necessary services, following the assessment of the responsible Platform Security Manager.

Motivation: The security of applications in Kubernetes is not assured if security functions are not utilized properly.

Implementation example: Use of security services offered by the Kubernetes platform, such as Network Policies, RBAC, Secrets, Pod Security Standards, and Service Mesh implementations to achieve application security. For more advanced requirements, leverage Kubernetes Operators that integrate with external security tools and services.

Motivation: The security of applications in the cloud is not given if security functions are not utilized in proper way.

Implementation example: Use of security services offered by the platform to achieve the security of the application.

ID: 3.34-11/5.0

Secret management plays a crucial role in security of Kubernetes systems. Secrets in this sense include:

- keys
- tokens
- passwords
- certificates

Here is the list of use case examples for KMS:

- storage encryption
- key management for SSH access
- API communication encryption
- tenant data encryption
- certificates for tenants
- secret management for tenants

The KMS may be delivered as the part of the cloud platform or separately, but it is practically impossible to run secure cloud platform or secure tenant workloads in the cloud without KMS. Therefore, even if the KMS is not delivered as the part of the cloud platform, it is necessary that the KMS is reachable and usable in automated way (like other cloud APIs).

If the KMS is not available or it is not possible to use KMS, the secrets must be handled in secure way by using tools like SOPS (<https://github.com/getsops/sops>).

In such case where KMS cannot be used and tools like SOPS is used, the process becomes critical for maintenance of security posture, as automation of key management, access controls and audit capabilities provided by KMS is replaced by manual or semi-automated processes. It is necessary to consider:

Regular Rotation Schedule: Establish and follow a strict key rotation schedule (at least once a year)

Onboarding/Offboarding: Have strict processes for handling keys (rotation/deletion) during personnel changes

Re-encryption Process: Define clear procedures for re-encrypting all secrets when keys change.

Emergency Rotation: Have documented procedures for emergency key rotation in case of compromise, including suspected compromise. Compromise or Suspicion of Compromise: Rotate all keys/secrets immediately if you suspect a compromise.

Need-to-Know Basis: Grant access only to team members who absolutely require it.

Activity Logging: Implement comprehensive logging of all encryption/decryption activities to create immutable audit trail.

Git Integration: When storing encrypted files in Git, configure .gitattributes to prevent accidental unencrypted commits.

Tool Consideration: Consider using age encryption as a simpler alternative to PGP if possible . PGP Keys: When using PGP keys instead of KMS, ensure proper key management with tools like GnuPG.

AGE (Actually Good Encryption) is a modern encryption tool designed to be simple, secure, and user-friendly. It uses state-of-the-art cryptographic primitives like X25519, ChaCha20-Poly1305, and HMAC-SHA256 to ensure security while avoiding the complexities and legacy issues of older tools like PGP

Motivation: KMS allows generation, storage and distribution of secrets in a proper way in cloud environment. If secrets would be stored on systems in the cloud in plaintext, they could be possibly accessed by unauthorized users.

Implementation example: A solution like HashiCorp Vault can be installed as part of the cloud infrastructure.

ID: 3.34-12/5.0

Req 13	Encryption keys must be rotated.
--------	----------------------------------

The symmetric encryption keys that the API server uses to encrypt etcd values are not automatically rotated - they must be rotated manually at least once a year. Master access is required to do this.

Motivation: Orchestrators (e.g. Kubernetes) will rotate some certificates automatically (notably, the kubelet client and server certs) by creating new CSRs as its existing credentials expire. Orchestrators do not do this for all Keys , so this must be checked.

ID: 3.34-13/5.0

2.3. Network

Req 14	Network and namespace policies must be used to limit communication.
--------	---

A restricted default policy must be enabled (default-deny) . Access must be whitelisted explicitly.

Ensure that the CNI in use supports Network Policies. Ensure that all Namespaces have Network Policies defined.

Motivation: By default, pods are non-isolated; they accept traffic from any source. Pods become isolated by having a NetworkPolicy that selects them. Once there is any NetworkPolicy in a namespace selecting a particular pod, that pod will reject any connections that are not allowed by any NetworkPolicy. (Other pods in the namespace that are not selected by any NetworkPolicy will continue to accept all traffic.)

Use a default policy to deny all ingress and egress traffic. Ensures unselected Pods are isolated to all namespaces ex-

cept kube-system.

For this requirement the following threats are relevant:

- Unauthorized access to the system
- Unauthorized access or tapping of data
- Unauthorized modification of data
- Unauthorized use of services or resources

ID: 3.34-14/5.0

Req 15	Access to administrative interfaces, such as Kubernetes APIs, dashboards and systems must be controlled and protected by higher layer firewalls or proxies (preferably hidden behind jump host), while communication must be encrypted.
--------	---

Per requirement which mandates absence of direct IP connectivity from external networks and tenant networks to Kubernetes API endpoints, access would be necessary for management functions of applications running in the cloud and for external people and/or machines. The access to Kubernetes API endpoints must traverse through higher layer firewalls (above L4 - TCP), WAFs or proxies.

Communication from end-clients (coming from both external and tenant networks) to firewall/WAF/proxy and from firewall/WAF/proxy to the exposed Kubernetes API endpoints must be encrypted, including both-way certificate verification to avoid man-in-the-middle attacks. Internal cloud/Kubernetes APIs must be naturally not reachable by tenants.

Specific web application checks need to be used for web-based UI (e.g. dashboards).

Strong authentication must be used for getting access to all API endpoints, according to existing security requirement documents about web applications and web services (e.g. document 3.02 - "Web Services").

Motivation: Cloud API endpoints have big attack surface with relatively low built-in protection against unexpected input and potentially high impact if breached.

Implementation example: It is preferred that all administrative communication is channeled through jump host. If an approach with jump host is not possible, strict VPN or ACLs can be used to limit the access to administrative cloud interfaces, APIs, dashboards etc. for only cloud administrators.

ID: 3.34-15/5.0

Req 16	Access to cloud API endpoints must be controlled and protected by higher layer firewalls or proxies, both for access from external networks and tenants. Communication must be encrypted. Authentication is mandatory.
--------	--

Per requirement which mandates absence of direct IP connectivity from external networks and tenant networks to cloud API endpoints, access would be necessary for management functions of applications running in the cloud and for external people and/or machines. The access to cloud API endpoints must traverse through higher layer firewalls (above L4 - TCP), WAFs or proxies.

Communication from end-clients (coming from both external and tenant networks) to firewall/WAF/proxy and from firewall/WAF/proxy to the exposed cloud API endpoints must be encrypted, including both-way certificate verification to avoid man-in-the-middle attacks. Internal cloud APIs must be naturally not reachable by tenants.

Alongside all regular APIs, this applies to Nova metadata server as well. In typical modern OpenStack deployments, there is Nova metadata proxy which fulfills this requirement for Nova metadata server. As of today, there is no encryption available for Nova metadata, so this endpoint does not have to be encrypted.

Specific web application checks need to be used for web-based UI (e.g. dashboards).

Strong authentication must be used for getting access to all API endpoints, according to existing security requirement documents about web applications and web services (e.g. document 3.02 - "Web Services").

Motivation: Cloud API endpoints have big attack surface with relatively low built-in protection against unexpected input and potentially high impact if breached.

Implementation example: Securing cloud APIs them with proxies, limiting API requests (including restricting possible API calls to only necessary) and filtering input is crucial to hinder possible attacks through cloud APIs. Firewalls, WAFs and proxies can be integrated with logging and monitoring so that attacks are detected early and properly mitigated.

In some very specific use cases where APIs cannot be properly secured while there is very small number of only internal tenants, there is another additional measure which can be used - hiding API endpoints behind jump host, assuming if tenant use cases allow for this. Then the access to cloud API endpoints for the tenant would be possible only from jump hosts, where each tenant could get a jump host created during the tenant creation.

For this requirement the following threats are relevant:

- Unauthorized access to the system
- Unauthorized access or tapping of data
- Unauthorized modification of data
- Unauthorized use of services or resources
- Disruption of availability
- Denial of executed activities
- Attacks motivated and facilitated by information disclosure or visible security weaknesses

ID: 3.86-12/2.0

Req 17	Tenants must be able to separate their Kubernetes clusters from others.
--------	---

The Kubernetes network controller needs to allow each tenant to create private namespaces which are visible and reachable only by the tenant itself. The network controller also must offer the possibility for each tenant to isolate certain parts of their namespaces from other namespaces within the same tenant's cluster.

Motivation: This allows tenants to build n-tier applications.

Implementation example: Kubernetes with regular CNI networking fulfills this requirement.

ID: 3.34-17/5.0

Req 18	All network connections to storage must be encrypted.
--------	---

All network connections (access) to data, either from hypervisors or object storage proxies for customers, must be encrypted.

Motivation: This prevents attackers on network to obtain data in clear text.

Implementation example: Use of TLS for all connections to storage system.

ID: 3.34-18/5.0

Req 19	Tenant data must be encrypted while in transit or at rest.
--------	--

Depending on the data classification and security level of the cloud infrastructure (and network connecting different cloud environments), it is necessary to:

- encrypt all the network traffic between virtual machines (or containers residing in different virtual machines)
- encrypt data which is stored on block or object storage

Application deployed in the cloud might end up running on a single hypervisor, inside same cloud environment (data

center) or distributed between different cloud environments (data centers). Although connectivity for the tenant may be transparent, tenants must be aware that communication between their workloads will go through network cables on public land. Therefore, all tenant traffic which leaves a single unit of workload (virtual machine or a container which talks to other containers which may reside elsewhere), it is necessary to encrypt that traffic. Best practice is to encrypt all the traffic to make sure that application doesn't need to be re-architected in case of changes in the cloud environments or further deployment of application to multiple sites. Encryption inside the data center also offers additional protection against hackers who managed to get into the cloud infrastructure or from rogue cloud administrators, so it is still beneficial to use it even if cloud environments are connected over encrypted links.

Motivation: Not all cloud environments may be deployed in fully secure locations (e.g. "edge clouds") and despite hard disk encryption offered by the platform, attackers who managed to get access to the cloud infrastructure wouldn't be able to easily extract the data if the data stored on volumes and in object storage buckets is encrypted. Encrypted data in storage buckets with proper use of different keys for different buckets also helps to prevent accidental data leaks, either to the public or to other workloads (e.g. accidental transfer from production to the test system).

Implementation example: Use of secure communications protocols (e.g. HTTPS, TLS and SSH) for communication and using encryption for data at rest - encrypted disks or objects.

ID: 3.34-19/5.0

2.4. Authentication and authorization

Req 20	Role-based access control(RBAC) must be used to control access.
--------	---

RBAC, enabled by default, is one method to control access to cluster resources based on the roles of individuals within an organization. RBAC can be used to restrict access for user accounts and service accounts. To check if RBAC is enabled in a cluster using kubectl, execute `kubectl api-versions`. The API version for `.rbac.authorization.k8s.io/v1` should be listed if RBAC is enabled.

Two types of permissions can be set:

- Roles – Set permissions for particular namespaces
- ClusterRoles – Set permissions across all cluster resources regardless of namespace Both Roles and ClusterRoles can only be used to add permissions. RBAC should always follow least privilege concept preventing ClusterRoles wherever possible.

There are no deny rules. If a cluster is configured to use RBAC and anonymous access is disabled, the Kubernetes API server will deny permissions not explicitly allowed.

The RBAC role `cluster-admin`` provides wide-ranging powers over the environment and should be used only where and when needed.

The Kubernetes API stores secrets, which may be service account tokens for the Kubernetes API or credentials used by workloads in the cluster.

Access to these secrets should be restricted to the smallest possible group of users to reduce the risk of privilege escalation.

Kubernetes Roles and ClusterRoles provide access to resources based on sets of objects and actions that can be taken on those objects.

It is possible to set either of these to be the wildcard `"*"` which matches all items. Use of wildcards must be avoided where ever possible

This Requirement can be subject to shared Responsibility. Either a Cluster Provider hands out pre defined RBAC Roles , or it could give you Cluster Admin and e.g. the User of the Cluster will be responsible for setting appropriate RBAC Controls based on least privilege.

Motivation: Restrict access to resources.

ID: 3.34-20/5.0

Req 21 A centralized authentication system for the API Server must be used.

Integrating Orchestrators with third party auth providers e.g. OpenID Connect Identity (OIDC) and OAuth 2.0 providers with pluggable connectors prevents administrators having to reconfigure the API server to add or remove users. Therefore you must implement them.

Motivation: Centralising authentication and authorisation (aka Single Sign On) helps onboarding, offboarding, and consistent permissions for users.

For this requirement the following threats are relevant:

- Unauthorized access to the system

ID: 3.34-21/5.0

2.5. Audit Logging / Threat Detection

Req 22 Security Relevant Events must be logged and Audit logging must be enabled.

Monitoring/logging should include the following:

- API request history
- Performance metrics
- Deployments
- Operating system calls
- Protocols,
- permission changes
- Network traffic
- Pod scaling
- Volume mount actions
- Image and container modification
- Privilege changes
- Scheduled job (cronjob) creations and modifications

Ensure that a minimal audit policy is created.

Ensure that the audit policy covers key security concerns.

Motivation: Logging of orchestration events for tenants is a prerequisite for the security of tenants' workloads (e.g. who, when created a specific cloud resource, modified it or deleted it).

Implementation example: Some examples include logging of:

- creation, deletion and changes of tenants (users, projects, roles)
- creation, deletion and changes to tenant networks
- creation, deletion and other virtual machine actions
- creation, deletion, modification and reading of object storage buckets and objects

ID: 3.34-22/5.0

Req 23 Logs from all Kubernetes infrastructure components must be forwarded to central logging solution in near-real-time. In addition, the logs need to be provided to an appropriate SIEM solution for collection, storage and analysis of security relevant events. SIEM must correlate security events and generate alarms for attacks.

Storing logs is not sufficient, they must be analyzed and properly correlated. A single system may detect an event and log it, but the information might not be sufficient to detect an ongoing attack. This is especially true in cloud environment where a single tenant is distributed among many servers. Anomaly detection should be used in addition to the definition of the rules for log filtering, monitoring and alarming.

Therefore, SIEM (Security Information and Event-Management) solution must be used on top of the central logging solution to analyze and correlate security events, detect anomalies and raise alarms in case of issues. This enables successful implementation of an incident management process, which has to include 24/7 follow up of alarms.

Due to attack surface of cloud environment, it is necessary that log collection, analysis and correlation, as well as alarming work in near real-time (with minimal reasonable delay possible, such as 5 minutes) to detect and mitigate ongoing attacks.

Assuming that the SIEM solution is available from responsible SOC (Security Operations Center), the cloud platform is responsible to provide and forward the logs and sufficient information for analysis and alerting, and also to provide sufficient information and help in setting up alerts.

Motivation: Storing logs on servers where they originated is not sufficient since the attacker can then easily manipulate logs and erase the evidence of a successful attack. Therefore, all logs must be forwarded to the central system.

Implementation example: SyslogNG, CheckMK, Nagios, Splunk etc. can be used as a central system.

ID: 3.34-23/5.0

Req 24	Every Kubernetes Node must have a Endpoint Detection and Response or Container Runtime solution.
--------	--

An up-to-date Endpoint Detection and Response (EDR) solution must be used on the operating system. An EDR solution collects security-relevant activity data from processes and evaluates them centrally. Alarms can be generated from malicious behavior of processes or by a specific signature (like classic virus scanners). In addition, it is possible to react directly to suspicious program behavior via a central console. For example, the operating system or rather the client can be isolated or the malicious process terminated. Furthermore, an overview of the vulnerabilities of all operating systems is forwarded to a central location.

Alternatively, Container Runtime Security can be implemented, particularly in environments relying heavily on containerized workloads. Container Runtime Security focuses on monitoring and protecting containers during their execution, enabling detection and response to malicious activities in real time. This includes identifying unauthorized processes, anomalous filesystem changes, and suspicious network behaviors. By enforcing security policies and monitoring runtime behavior, Container Runtime Security provides a robust alternative or complement to EDR, especially in modern cloud-native and Kubernetes-based infrastructures.

Motivation: Normal virus scanners rely purely on signatures, whereas an EDR solution also searches for anomalies in the process behavior. Signatures have the disadvantage that new signatures have to be written for new virus variants before they can be detected

Implementation example: Have a look at the PSA Wiki - Security Requirements: Endpoint Detection and Response (EDR) for guidance.

<https://yam-united.telekom.com/pages/psa/apps/wiki/wiki-sicherheitsanforderungen/list/view/7214872e-276f-4cb0-acd7-4c85860d0ee8>

For Container Runtime Security, you could consider options like:

<https://github.com/falcosecurity/falco>

<https://github.com/cilium/tetragon/>

ID: 3.34-24/5.0

2.6. Policies

Policies which are important to the security of the environment.

Req 25	Pod Security Admissions or a 3rd party admission plugin must be used limit container privileges and provide platform rules.
--------	---

A restricted default policy must be enabled.

The following **MUST** be limited by a Policy Engine:

Prohibit the admission of privileged containers
Prohibit the admission of containers sharing the host process ID namespace
Prohibit the admission of containers sharing the host IPC namespace
Prohibit the admission of containers wishing to share the host network namespace
Prohibit the admission of containers with allowPrivilegeEscalation
Prohibit the admission of root containers
Prohibit the admission of containers with the NET_RAW capability
Prohibit the admission of containers with added capabilities
Prohibit the admission of containers with capabilities assigned
Prohibit the admission of Windows HostProcess Containers
Prohibit the admission of HostPath volumes
Prohibit the admission of containers which use HostPorts
ReadOnly Filesystem for Containers must be enforced

Policies can be enforced in multiple ways:

- Pod Security Admissions <https://kubernetes.io/docs/concepts/security/pod-security-admission/>
- Open Policy Agent / Gatekeeper <https://www.openpolicyagent.org/>
- Kyverno <https://kyverno.io/>

On shared/multi-tenant Platforms no Policy exceptions are possible. Policy exceptions are only possible with PSM approval on dedicated Clusters

Motivation: Most Pods don't need privileged access or even host access, so it should be ensured that a Pod requesting such access needs to be whitelisted explicitly. By default no one should be able to request privileges above the default to avoid being vulnerable through misconfiguration or malicious content of a Container image.

Implementation example: Shared Responsibility , the Platform (Orchestrator) has to provide the utility to provide this. An Application has to utilize this.
Most Platforms have Policies defined which Applications need to follow.

Containers / Pods must specify a Security Context <https://kubernetes.io/docs/tasks/configure-pod-container/security-context/>

<https://kubernetes.io/docs/concepts/security/pod-security-standards/>

For this requirement the following threats are relevant:

- Unauthorized access to the system
- Unauthorized access or tapping of data
- Unauthorized modification of data
- Unauthorized use of services or resources

ID: 3.34-25/5.0

Req 26	Resource policies must be used.
--------	---------------------------------

Resource policies LimitRanges, ResourceQuotas, and Process ID Limits restrict resource usage for namespaces, nodes, or Pods. These policies are important to reserve compute and storage space for a resource and avoid resource exhaustion.

Motivation: Avoid resource exhaustion.

Implementation example: Enforce minimum and maximum compute resources usage per Pod or Container in a namespace.
Enforce minimum and maximum storage request per PersistentVolumeClaim in a namespace.
Enforce a ratio between request and limit for a resource in a namespace.
Set default request/limit for compute resources in a namespace and automatically inject them to Containers at runtime.

ID: 3.34-26/5.0

Req 27	Untrusted Workloads must be isolated if they pose a risk for the Kubernetes platform and its management components.
--------	---

Workload isolation is the practice of establishing security boundaries between containers and their host environment to mitigate the risk of container escape vulnerabilities and limit the potential blast radius of compromised containers.

Different workloads may need different levels of isolation based on their risk profile:

Basic isolation (Standard containers with security best practices)

For trusted workloads with minimal risk.
Relies on Linux namespaces and cgroups.
Supplemented with security context restrictions, seccomp, AppArmor, etc.

Enhanced isolation (User namespaces)

<https://kubernetes.io/docs/concepts/workloads/pods/user-namespaces/>
For workloads where you want to run as root inside containers but limit privilege on the host.
Mitigates certain categories of container escape vulnerabilities.
Protects against UID/GID conflicts between containers.

Strong isolation, Sandboxing (gVisor, Kata Containers)

For untrusted or high-risk workloads.
Application does not fit into cloud native environment at all.
Security Policies deviating at to high margin from best practices and pose a risk for the platform.

The appropriate isolation strategy depends on your specific threat model, performance requirements, and the level of trust in the workloads running on your cluster. As the risk of the workload increases, you would escalate from basic isolation to enhanced isolation (user namespaces) and finally to strong isolation (sandboxing runtimes) as necessary.

ID: 3.34-27/5.0

2.7. Deployment/Application Level Security

This section describes the security measures Applications have to follow when running on top of an Orchestrator.

Req 28	Usage of Kubernetes default Namespace is prohibited.
--------	--

A Kubernetes namespace allows you to partition created resources into logically named groups. Resources created in one namespace can be hidden from other namespaces. By default, each resource created by a user in Kubernetes cluster runs in a default namespace, called default. You can create additional namespaces and attach resources and users to them.
Use Namespaces to isolate your Kubernetes objects.

ID: 3.34-28/5.0

Req 29	Sensitive information must not be stored in pod-type YAML resources.
--------	--

Using a KMS is the preferred way.
If the KMS is not available or it is not possible to use KMS, the secrets must be handled in secure way by using tools like SOPS (<https://github.com/getsops/sops>).
Sensitive information must not be stored in pod-type YAML resource (deployments, pods, sets, etc.), and sensitive con-

figmaps and secrets must be encrypted.

The following must be considered:

Regular Rotation Schedule: Establish and follow a strict key rotation schedule (at least once a year)

Onboarding/Offboarding: Have strict processes for handling keys (rotation/deletion) during personnel changes

Re-encryption Process: Define clear procedures for re-encrypting all secrets when keys change.

Emergency Rotation: Have documented procedures for emergency key rotation in case of compromise, including suspected compromise. Compromise or Suspicion of Compromise: Rotate all keys/secrets immediately if you suspect a compromise.

Motivation: Prevent secrets from being visible in yaml files.

For this requirement the following threats are relevant:

- Unauthorized access to the system
- Unauthorized access or tapping of data
- Unauthorized modification of data
- Unauthorized use of services or resources

ID: 3.34-29/5.0

Req 30	Network Policies must be used to restrict traffic between pods and ensure proper network segmentation.
--------	--

Define ingress and egress rules to control the flow of traffic between pods, namespaces, and external systems.

Motivation: Implementing Network Policies ensures proper network segmentation and restricts traffic between pods, reducing the risk of unauthorized access and lateral movement within the cluster.

Implementation example: apiVersion: networking.k8s.io/v1

kind: NetworkPolicy

metadata:

name: allow-app-to-db

spec:

podSelector:

matchLabels:

app: my-app

ingress:

- from:

- podSelector:

matchLabels:

app: my-database

ports:

- protocol: TCP

port: 3306

ID: 3.34-30/5.0

Req 31	RBAC must be used to control access to Kubernetes resources based on the principle of least privilege.
--------	--

Define Roles and RoleBindings:

Grant only the necessary permissions to users, groups, and service accounts.

Ensure that service accounts used by applications have restricted permissions, especially on sensitive resources like secrets and config maps.

Motivation: Using RBAC to control access to Kubernetes resources enforces the principle of least privilege, reducing the risk of unauthorized actions and potential security breaches.

Req 32 Deployments/Applications must specify their Security Context.

The following must be defined within the Security Context:

<https://kubernetes.io/docs/tasks/configure-pod-container/security-context/>

The following restrictions must be defined in the Security Context to ensure the security of Kubernetes workloads:

Prohibition of Privileged Containers

privileged: false must be set in the Security Context.

This prevents containers from running with root privileges on the host.

Prohibition of Host PID Namespace Usage

hostPID: false must be set in the Pod spec.

Prevents containers from viewing or manipulating processes on the host system.

Prohibition of Host IPC Namespace Usage

hostIPC: false must be set in the Pod spec.

Prevents the use of host inter-process communication mechanisms.

Prohibition of Host Network Namespace Usage

hostNetwork: false must be set in the Pod spec.

Prevents direct access to the host's network interface.

Prohibition of Privilege Escalation

allowPrivilegeEscalation: false must be set in the Security Context.

Prevents container processes from expanding their privileges.

Prohibition of Root Containers

runAsNonRoot: true must be set in the Security Context.

Enforces containers to run as non-root users.

Prohibition of NET_RAW Capability

The NET_RAW capability must explicitly not be included in the list of capabilities.add in the Security Context.

This capability enables the creation of raw sockets and should be removed for security reasons.

Prohibition of Additional Capabilities

No additional capabilities should be added through capabilities.add in the Security Context.

Limits the possibilities for container escape attacks.

Removal of All Capabilities

capabilities.drop: ["ALL"] must be set in the Security Context.

Removes all Linux capabilities by default and increases security.

Prohibition of Windows HostProcess Containers

For Windows containers, windowsOptions.hostProcess: false must be set.

Prevents containers from interacting directly with the Windows host.

Prohibition of HostPath Volumes

No hostPath volumes should be defined in the Pod spec.

Prevents direct access to the host filesystem.

Prohibition of Host Ports Usage

No hostPort definitions should be present in the container spec.

Prevents containers from binding directly to host ports.

Enforcement of ReadOnly Filesystems

readOnlyRootFilesystem: true must be set in the Security Context.

Prevents write access to the container filesystem.

Motivation: These requirements implement multiple layers of protection against container escape vulnerabilities, which could otherwise allow malicious containers to break out and access the host system.

Principle of Least Privilege: By restricting capabilities, access rights, and namespaces, containers operate with only the minimum privileges necessary to function, reducing the attack surface significantly.

Prevention of Privilege Escalation: These controls prevent containers from gaining additional privileges during runtime, which is a common technique used in container escape attacks.

Host System Protection: By isolating containers from host resources (network, IPC, PID namespaces), the host system is protected from direct manipulation by containerized applications.

Implementation example: <h1>apiVersion: apps/v1 </h1>

kind: Deployment

metadata:

name: compliant-nginx

labels:

app: compliant-nginx

spec:

replicas: 3 # Multiple replicas for high availability

selector:

matchLabels:

app: compliant-nginx

template:

metadata:

labels:

app: compliant-nginx

annotations:

SECURITY: Applies seccomp profile to restrict syscalls, reducing attack surface

seccomp.security.alpha.kubernetes.io/pod: "runtime/default"

SECURITY: Applies AppArmor profile to restrict file access and capabilities

container.apparmor.security.beta.kubernetes.io/compliant-nginx: "runtime/default"

spec:

securityContext:

SECURITY: Prevents container from running as root user

runAsNonRoot: true

SECURITY: Specifies non-root user ID to run container processes

runAsUser: 101

SECURITY: Sets group ID for container processes

runAsGroup: 101

SECURITY: Sets group ID for mounted volumes

fsGroup: 101

SECURITY: Applies seccomp profile at pod level (newer API)

seccompProfile:

type: RuntimeDefault

containers:

- name: compliant-nginx

SECURITY: Uses specific version instead of "latest" to ensure consistency and proper security testing

image: nginx:1.25.3-alpine

SECURITY: Avoids automatically pulling potentially compromised new images

imagePullPolicy: IfNotPresent

securityContext:

SECURITY: Prevents processes from gaining additional privileges

allowPrivilegeEscalation: false

SECURITY: Prevents modification of container filesystem during runtime

readOnlyRootFilesystem: true

capabilities:

SECURITY: Drops all Linux capabilities by default

drop: ["ALL"]

SECURITY: Explicitly disables privileged mode which would give host-level access

privileged: false

```

resources:
  # SECURITY: Sets resource limits to prevent resource exhaustion attacks
  limits:
    cpu: "500m"
    memory: "256Mi"
  requests:
    cpu: "100m"
    memory: "128Mi"
ports:
  # SECURITY: Uses non-privileged port (>1024) to minimize required capabilities
  - containerPort: 8080
    name: http
volumeMounts:
  # SECURITY: Mounts required writable directories as isolated volumes since root filesystem is read-only
  - name: tmp-volume
    mountPath: /tmp
  - name: nginx-cache
    mountPath: /var/cache/nginx
  - name: nginx-run
    mountPath: /var/run
  # SECURITY: Mounts configuration as read-only to prevent runtime modification
  - name: nginx-conf
    mountPath: /etc/nginx/conf.d
    readOnly: true
# SECURITY: Implements health probes to ensure container is functioning properly
livenessProbe:
  httpGet:
    path: /
    port: http
  initialDelaySeconds: 10
  periodSeconds: 20
readinessProbe:
  httpGet:
    path: /
    port: http
  initialDelaySeconds: 5
  periodSeconds: 10
volumes:
  # SECURITY: Uses ephemeral volumes instead of persistent storage for writable directories
  - name: tmp-volume
    emptyDir: {}
  - name: nginx-cache
    emptyDir: {}
  - name: nginx-run
    emptyDir: {}
  # SECURITY: Uses ConfigMap for configuration instead of hardcoded values
  - name: nginx-conf
    configMap:
      name: compliant-nginx-config
# SECURITY: Prevents pod from accessing host network interfaces
hostNetwork: false
# SECURITY: Prevents pod from viewing or manipulating host processes
hostPID: false
# SECURITY: Prevents pod from accessing host IPC mechanisms
hostIPC: false
# SECURITY: Prevents automatic mounting of service account token to limit access to Kubernetes API
automountServiceAccountToken: false

```

ID: 3.34-32/5.0

Req 33	Patching of the cloud infrastructure must not be delayed or blocked by tenants (neither technically nor administratively).
--------	--

It is mandatory that SLA reflects the fact that no tenant can block or prevent patching of the underlying infrastructure.

The project which wants to get onboarded on cloud even with an idea that they can delay or block patching and other necessary works of the cloud infrastructure must never be onboarded.

Motivation: Patching protects all other workloads and the cloud infrastructure, and it would be extremely unreasonable and dangerous to jeopardize security of whole cloud and all other systems just because of one tenant.

Implementation example: For certain technologies and workloads this can be achieved by live migration, and for those where there is no technical capability, it can be covered by a clause in SLA terms.

ID: 3.34-33/5.0

3. Private Clouds

3.1. Cloud Infrastructure Network and Cloud Management

Cloud infrastructure network (also known as underlay network) of the cloud is cloud internal network necessary for IP connectivity between all hardware components and cloud software, including:

- compute nodes
- storage nodes
- network nodes
- server management (BMC - Basedboard Management Controller, e.g. IPMI or Redfish)
- network devices (switches, routers, firewalls, ...)
- cloud management functions

This does not include tenant networks.

3.1.1. Around Cloud Management

Cloud environment requires variety of other systems to be in place for secure operation of the cloud environment. Most of the systems (e.g. logging, monitoring, authentication, ...) are well known and covered in other security documents in details, however, here is a shorter reminder in form of a list of the most important systems.

It is worth repeating baseline principles when it comes to preparation, configuration, installation and operation of cloud infrastructure:

- attack surface of systems and services must be limited
- systems must follow principle of minimal installation
- any sensitive data must be encrypted during transmission and storage (and if possible, in use)
- permissions of users (human and M2M) must follow least privilege principle
- communication partners must be properly authenticated and authorized
- systems must be segregated accordingly
- incoming, outgoing and internal communication must be controlled and filtered
- infrastructure, systems and services must be monitored and regularly tested
- systems must be updated and patched regularly
- incident management process must be in place

Req 34	Security compliance check must be implemented in an automated way.
--------	--

Due to variety of systems which will be hosted in the cloud and complexity of the cloud platform, it is necessary to implement compliance checks for relevant security requirements on all servers and devices.

The implementation must be done as part of CI/CD chain in pre-deployment (if CI/CD and automation are used) and during lifetime on a regular basis. Again, as for hardening, automation must be used where possible to avoid risk of missing compliance checks and/or systems.

Motivation: Regular compliance checks validate current security status of systems and can identify weaknesses in security configurations or detect unpatched systems with known vulnerabilities.

Implementation example: Some examples of compliance verification tools include:

- CIS Benchmarks
- Ansible
- InSpec
- Orpheus (Telekom internal)
- TASTE-OS (Telekom internal, list of all TOS instances can be found at <https://yam.telekom.de/docs/DOC->

For this requirement the following threats are relevant:

- Unauthorized access to the system
- Unauthorized access or tapping of data
- Unauthorized modification of data
- Unauthorized use of services or resources
- Disruption of availability
- Denial of executed activities
- Unnoticeable feasible attacks
- Attacks motivated and facilitated by information disclosure or visible security weaknesses

ID: 3.86-28/2.0

3.2. Cloud Usage

Clouds are typically deployed to enable automation and minimize cost. It is very common step also to use the same cloud environment for deployment of development, test, reference and production systems/applications on top. Mixing development, test, reference and production systems brings an additional risk. With proper application of all security measures in this document and best security practices in general, it is possible to lower the risk to an acceptable level for operation of development, test, reference and production systems in the same cloud.

This portion of the document describes in addition mandatory security requirements which allow mixing of development, test, reference and production cloud infrastructures, as well as mixing of development, test, reference and production systems on the same cloud infrastructure.

3.2.1. Separation of Cloud Infrastructure Depending on the Purpose

Req 35	Development, test and reference cloud infrastructures themselves must be operated on a fully separate hardware from production cloud infrastructure, including cloud management and orchestration system.
--------	---

Compute nodes, storage nodes, network devices and other supporting systems which belong to development, test or reference cloud infrastructure must be operated on separate hardware from production cloud environment. This extends also to having a separate DCR (data center router).

Even if security requirements for test and reference systems align with security requirements for production systems, one need to consider operational aspects - testing a new change or feature on shared hardware between test and production cloud environments might bring the whole production cloud environment down.

Motivation: Development, test and reference systems do not always have the same security requirements as production systems and mixing them would introduce unnecessary risk to the production cloud environment.

Implementation example: Separate (disjunct) set of hardware (or software or externally used services) for each of the development, test and production cloud infrastructure platforms.

For this requirement the following threats are relevant:

- Unauthorized access to the system
- Unauthorized access or tapping of data
- Unauthorized modification of data
- Unauthorized use of services or resources
- Disruption of availability
- Denial of executed activities
- Attacks motivated and facilitated by information disclosure or visible security weaknesses

ID: 3.86-63/2.0