Deckhouse Airlock Documentation

Generated: October 26, 2026

Documentation

Product Overview 4
System Components 5
Architecture 8
Component Interaction 8
Authentication 13
Authorization 15
Proxy Service 19
Installation and Configuration 21
Installation 21
Binary Installation 21
Deploying the Server on Linux 23
Deploying in Kubernetes 26
Airlock Agents 29
Join-token enrollment 31
Linux SSH node 34
OpenSSH without an agent 37
Kubernetes agent 39
Database access 42
PostgreSQL 44
MySQL / MariaDB 46
OpenSearch / Elasticsearch 48
Web applications 50
RDP and Windows 54
Authentication 57
Local users 57
OIDC authentication 61
SAML authentication 65
Two-factor authentication (OTP/TOTP) 70
WebAuthn (second factor) 72
Access policies (RBAC) 75
Access to SSH hosts 78
Access to Kubernetes 81
Access to databases 84
User guide 88
airsh utility 88

Access Requests 94
Web interface 100
Reference 103
airsh reference 103
airctl reference 109

Product Overview

Airlock is a secure and convenient way to access and protect your infrastructure.

The Airlock Access platform provides least-privilege access based on cryptographic identity and the Zero Trust model, with built-in policy management and auditing.

Use Cases

Organizations use Airlock to:

- **Eliminate fragmented access systems:** a single platform for role-based access control, auditing, and connectivity across all infrastructure — from servers to Kubernetes clusters and databases.
- **Adopt Zero Trust with compromise-resistant credentials:** authenticate infrastructure access using short-lived certificates verified at every endpoint.
- **Meet security compliance requirements:** centralized auditing, session recording, and flexible access policy management.

Platform Capabilities

Airlock Access provides Zero Trust connectivity across all infrastructure. Users access servers, databases, Kubernetes clusters, and other resources through a secure proxy, even when resources are behind a firewall.

All infrastructure resources are part of a unified catalog with a shared role-based access control (RBAC) system. Connections use short-lived certificates that Airlock components verify cryptographically.

Access Requests are a built-in platform capability: users can request temporary privilege elevation to roles or specific resources, and designated reviewers approve or reject requests through the web interface or CLI. See the user guide for details.

Airlock Identity (optional module) extends access management capabilities:

- **Access Lists:** regular auditing and control of role membership.
- **Device Trust:** require authentication from a registered device.
- **Session and identity locking:** immediately stop activity from compromised users.

Architecture

A minimal Airlock cluster consists of **Auth Service** (authentication service) and **Proxy Service** (proxy service). Airlock agents running on servers or in Kubernetes proxy access to infrastructure resources.

Users authenticate through Proxy Service using short-lived certificates. Certificates contain Airlock user attributes, enabling agents to enforce RBAC policies.

Core Components

Component	Purpose
Auth Service	User, role, configuration, and certificate authority management
Proxy Service	User entry point: web interface, SSH, Kubernetes, databases
SSH Service	Server access via SSH with auditing and session recording
Kubernetes Service	Proxy access to the Kubernetes API
Database Service	Proxy access to PostgreSQL, MySQL, and other databases
Application Service	Proxy HTTP/TCP traffic to internal applications
Desktop Service	Access to Windows desktops via RDP

See the System Components and Architecture sections for details.

Client Utilities

Utility	Purpose
`airsh`	User client: cluster login, SSH, Kubernetes, databases
`airctl`	Admin tool: user, role, and resource management
`airlock`	Server and agent: Auth Service, Proxy Service, access services

System Components

This document describes the core concepts and components of an Airlock deployment. These terms appear throughout the platform configuration and operation documentation.

Airlock Cluster

The central concept in Airlock's architecture is the **cluster**. A cluster consists of **Auth Service** and **Proxy Service**, along with **Airlock services** that manage access to infrastructure resources: Kubernetes clusters, Windows desktops, databases, and applications.

A minimal cluster includes Auth Service and Proxy Service. In a demo environment, both services can run as a single `airlock` process on a Linux host.

For details on how components interact: Component Interaction.

Auth Service

Auth Service manages **local users** and **configuration resources** for the cluster. It maintains certificate authorities (CAs) that allow users and services to authenticate to the cluster. Auth Service issues certificates to clients and maintains an audit log.

Auth Service is the only cluster component that requires a backend connection for storing cluster state and CA private keys. **Airlock services** are stateless and communicate with Auth Service via the gRPC API.

For high availability, you can run multiple Auth Service instances.

See also: Authentication, Authorization.

Proxy Service

Proxy Service provides secure access to infrastructure resources from a public network without a VPN.

It establishes reverse tunnels to Auth Service and Airlock services that may run in private networks. In a minimal configuration, only port `443` needs to be exposed to the internet, while the rest of the infrastructure stays in a private network.

For production, you can run **multiple Proxy Service instances** behind a load balancer — this increases availability, distributes load, and reduces latency by placing proxies closer to users or agents.

Clients can also connect to resources directly using Airlock certificates, bypassing Proxy Service.

See also: Proxy Service.

Airlock Services

An **Airlock service** manages access to infrastructure resources. A single `airlock` process can run one or more services depending on the configuration.

Application Service

Proxies HTTP and TCP traffic to configurable endpoints — internal web applications and other services.

Database Service

Proxies TCP traffic using the native protocols of popular databases, including PostgreSQL and MySQL.

Desktop Service

Proxies Remote Desktop Protocol (RDP) traffic to Windows desktops.

Kubernetes Service

Proxies HTTP traffic to the Kubernetes API server.

SSH Service

An SSH server implementation that allows executing commands on remote machines with built-in access control, auditing, and session recording.

Agent

An instance of an Airlock service is called an **agent**. For all services except SSH Service, a single agent can provide access to multiple resources and run on a separate host from the target resources. All agents must be in the same network as their target resources.

Configuration Resources

A **configuration resource** is a document stored in the Auth Service backend that defines cluster parameters. Examples: **roles**, **local users**, **authentication connectors**.

Role

A **role** is a configuration resource that grants **Airlock users** privileges in the cluster. RBAC in Airlock denies access by default: users need explicit permissions to access a resource or perform administrative actions.

Airlock Users

Airlock supports two types of users:

- **Local users** — correspond to a `user` resource stored in the Auth Service backend.
- **SSO users** — stored in an external identity provider (GitHub, SAML, OIDC). When authenticating via SSO, Airlock issues a certificate and creates a temporary local user for the duration of the certificate.

Ultimately, an Airlock user is the subject of a certificate issued by Auth Service. Auth Service verifies the presence of a valid Airlock certificate and uses the certificate subject — username and roles — for authorization.

Authentication Connector

An authentication connector is a configuration resource that allows users to log into Airlock through an external identity provider (SSO).

Trusted Clusters

Airlock supports a **trusted cluster relationship** between a **root cluster** and one or more **leaf clusters** that trust the root cluster's CA. Users authenticated to the root cluster can access resources in a leaf cluster. Root and leaf clusters operate independently with their own users, roles, and resources; the trust relationship allows mapping root cluster roles to leaf cluster roles and permissions.

Architecture

Architecture guides for Airlock components.

Component Interaction

This document describes how the core Airlock components interact during installation, agent connection, and user access to resources.

Component Roles

Component	Where it runs	Purpose
Auth Service	Control node (private network)	Certificate authorities, users, roles, configuration, audit log
Proxy Service	Perimeter (port 443)	User entry point, web interface, traffic routing to agents
Agent	Near resources (private network)	SSH, Kubernetes, databases, applications, RDP — proxying to specific resources
User	Workstation	`airsh`, browser — authentication and connecting to resources

Auth Service is the only cluster component with persistent state storage. Proxy Service and agents do not store cluster configuration: they obtain credentials from Auth Service and maintain a connection to it through Proxy Service.

Overview

Users connect only to **Proxy Service** (typically `https://airlock.example.com`). Resources and agents in the private network **do not expose** inbound ports to the internet — agents themselves establish outbound reverse tunnels to the proxy.

Auth Service

Auth Service serves as the cluster’s trust center:

- Stores users, roles, SSO connectors, and resource definitions.
- Manages certificate authorities (CAs) for users, nodes, and services.

- Issues short-lived certificates after successful authentication.
- Processes agent join requests (join tokens) and issues long-lived certificates to agents.
- Records audit events.

Auth Service does not accept user traffic for SSH, Kubernetes, or databases directly. Clients and agents communicate with it via the gRPC API; for users, this path goes through Proxy Service.

Proxy Service

Proxy Service is the single entry point:

- **Web interface** — user login, resource listing, launching applications, SSH and RDP terminal in the browser.
- **Identity-Aware Proxy (IAP)** — issuing user certificates after SSO or local login; the `airsh` client uses these certificates for subsequent connections.
- **Routing** — forwarding SSH, Kubernetes, HTTPS application, and database traffic to the appropriate agent via reverse tunnel.
- **TLS routing** — multiple protocols over a single port 443.

Proxy Service maintains persistent **reverse tunnels** from agents. When a user connects to a resource, the proxy selects the agent registered for that resource and forwards traffic through the established tunnel.

Multiple Proxy Service Instances

In production, you can run **multiple Proxy Service instances** — on different nodes, in different availability zones, or regions. This provides:

- **High availability** — if one proxy fails, users and agents continue working through the remaining instances.
- **Load distribution** — incoming traffic is spread across proxies (load balancer or DNS).
- **Lower latency** — proxies can be placed closer to users or agents to reduce network path.

Typical setup:

```
flowchart TB
  users[Users]
lb[Load Balancer / DNS]
  p1[Proxy 1]
  p2[Proxy 2]
  p3[Proxy N]
auth[Auth Service]
  agent[Agent]

  users --> lb
    lb --> p1
    lb --> p2
    lb --> p3
  p1 --> auth
  p2 --> auth
  p3 --> auth
agent -->|tunnel| p1
agent -->|tunnel| p2
agent -->|tunnel| p3
```

Recommendations:

- Place a **load balancer** or DNS with multiple A records in front of the proxies; use **round-robin** algorithm, without sticky sessions.
- Agents establish a reverse tunnel **to each** Proxy Service in the cluster when connecting — see join-token connection.
- All Proxy Service instances use one Auth Service and shared TLS certificates for the same domain (`airlock.example.com`).
- Auth Service can also run as multiple instances for fault tolerance; cluster state is stored in a shared backend.

Placing proxies in regions close to user teams or agent pools reduces RTT during login and session proxying.

Agents

An **agent** is an `airlock` process with one or more services (SSH, Kubernetes, Database, Application, Desktop), deployed near the protected resources.

Joining an Agent to the Cluster

1. An administrator creates a join token: `airctl tokens add --type=node` (or `kube`, `db`, `app`, etc.).
2. The agent starts with the token and Proxy Service address.
3. The agent establishes a TLS connection to Proxy Service and registers with Auth Service.
4. Auth Service issues a certificate to the agent; the agent opens a reverse tunnel to the proxy.
5. The agent periodically renews its certificate and maintains the tunnel.

The agent does not need a public IP address or open port — outbound access to Proxy Service on port `443` is sufficient.

Agent Behavior During User Access

1. The user authenticates and receives a certificate with roles and attributes.
2. The user initiates a connection (e.g., `airsh ssh user@host` or `airsh db connect`).
3. Proxy Service validates the user certificate and RBAC with Auth Service.
4. The proxy routes traffic through the tunnel to the agent serving the resource.
5. The agent re-validates the certificate and policies, then establishes a connection to the target resource (SSH, Kubernetes API, PostgreSQL, etc.).
6. Session events are recorded in the Auth Service audit log.

Users

Login and Certificate Issuance

A user authenticates using one of the following methods:

- **Local account** — username and password (+ OTP or WebAuthn when MFA is enabled).
- **SSO (OIDC / SAML)** — external identity provider.

After successful login, Auth Service issues a **short-lived certificate** containing the username and roles. The `airsh` client stores the certificate locally; the browser uses a session for the web interface.

```
airsh login --proxy=airlock.example.com --user=alice
```

Accessing Resources

Method	Tool	Traffic path
SSH	`airsh ssh`	User → Proxy → SSH agent → server
Kubernetes	`airsh kube login`	User → Proxy → Kubernetes agent → API server
Database	`airsh db connect`	User → Proxy → Database agent → DBMS
Application	Browser / `airsh`	User → Proxy → Application agent → HTTP service
RDP	Web UI / `airsh`	User → Proxy → Desktop agent → Windows

At every step, the Airlock certificate is validated; static passwords to target resources are never exposed to the user.

Administrator

Administrators use `airctl` to manage the cluster: users, roles, tokens, resources. `airctl` can run:

- on the Auth Service host (with local configuration);
- from a workstation after `airsh login` (via Proxy Service).

Common Scenarios

Initial Deployment

1. Install Auth Service and Proxy Service on the control node (or in Kubernetes).
2. Configure DNS and TLS for `airlock.example.com`.
3. Create local users and roles using `airctl`.
4. Install agents on servers, in Kubernetes clusters, and near databases.
5. Users run `airsh login` and connect to resources.

Resource Behind a Firewall

Resources and agents remain in the private network without inbound internet access. The agent initiates an outbound connection to Proxy Service; internet users connect only to the proxy, which forwards traffic through the tunnel.

Direct Connection (Without Proxy)

In some configurations, a client with a valid Airlock certificate can connect directly to an agent, bypassing Proxy Service. This reduces latency but requires network reachability of the agent from the client. Routing through the proxy is recommended by default.

Authentication

Authentication and Authorization

Airlock handles both authentication and authorization:

- **Authentication** — verifying the identity of a user or service.
- **Authorization** — checking access rights to a resource.

This document describes authentication using short-lived certificates.

Short-Lived Certificates

Certificate authorities (CAs) and short-lived certificates are the foundation of Airlock authentication. At the start of every connection, a user or service presents a valid certificate issued by a trusted CA. Clients always initiate mutual TLS or SSH authentication.

The Airlock CA issues short-lived X.509 certificates for web services, databases, Kubernetes clusters, desktops, and SSH certificates for OpenSSH-compatible servers.

Advantages of Certificates

- Certificates are bound to a user or service identity — every action is traceable.
- Short-lived certificates expire automatically; revocation is not required.
- Certificates solve the Trust On First Use (TOFU) problem: all servers in the cluster have their own identities and certificates and do not accept client certificates signed by an untrusted CA.
- Certificates provide mutual authentication (mTLS), reducing the risk of impersonation, man-in-the-middle attacks, and credential brute-forcing.
- Certificates scale better: each service only needs to verify the CA signature, without copying credentials to every node.

> Airlock issues certificates with lifetimes ranging from a few hours to a few minutes. The shorter the lifetime, the better. Ideally, issue a certificate only for the duration of a session. In practice, a few hours or a working day is acceptable. The expiry date in a certificate cannot be forged without invalidating it.

X.509 Certificates

X.509 certificates are the same certificates used when accessing websites through a browser. They bind an identity to a public key with a CA signature.

Airlock uses X.509 for Kubernetes, databases, web services, and internal components (proxy, Auth Service) to establish mutual TLS authentication (mTLS).

OpenSSH Certificates

SSH certificates are similar to X.509 and also bind a user or server identity to a public key with a CA signature.

An SSH certificate contains metadata for authentication:

- A list of principals (identities) that the certificate belongs to.
- The signature of the CA that issued the certificate.
- An expiry date (TTL).
- Additional data (e.g., node role) in certificate extensions.

Time as a Security Factor

Expiration is a built-in property of certificates. SSH and X.509 certificates contain an expiry date that servers verify alongside the signature.

Airlock issues certificates with lifetimes from a few hours down to minutes; upon expiry they automatically become invalid. Instead of revocation lists, Airlock relies on time.

> If certificate expiration is insufficient (for example, during a security incident), Proxy Service can immediately terminate active connections using session and identity locking.

User Certificates

To issue a certificate to a user, Airlock presents a login screen, issues the certificate, and delivers it to the user's computer.

Using SSO (GitHub, SAML, OIDC, or another identity provider) to obtain a short-lived certificate is recommended.

Internal Certificates

Internal Airlock services — Auth, Proxy, and nodes — use certificates to identify themselves within the cluster. To connect proxies and nodes to the cluster, administrators use short-lived join tokens.

Unlike users, internal services receive long-lived certificates. To renew them, administrators perform CA rotation — revoking all previously issued certificates regardless of their expiry and issuing new ones with a new CA.

> To quickly lock a compromised node, proxy, or Auth Service without rotating certificates for the entire cluster, use session and identity locking.

Authorization

Authentication and authorization

Airlock handles both authentication and authorization:

- **Authentication** — verifying the identity of a user or service.
- **Authorization** — checking access rights to a resource.

This document describes authorization of users and services using RBAC.

Users and roles

Airlock supports several account types:

- Interactive and non-interactive.
- Local and external.

After successful authentication, each user is assigned one or more roles.

Interactive users

Interactive users can be local or external. Local accounts store a password hash and MFA data in the Airlock backend. External accounts authenticate through an identity provider using SSO protocols — OAuth 2.0, OIDC, or SAML.

External users from SSO

Each time an SSO user logs in, Airlock creates a temporary account record that expires automatically with the SSO session and writes events to the audit log. This prevents name conflicts with local users.

External users from other clusters

A user can be external to an Airlock cluster if another cluster or CA issues a certificate that this cluster trusts. In this case, Airlock applies trusted cluster role-mapping logic.

Local interactive users

Local interactive users have a record in the Airlock backend with credentials. An administrator creates accounts via `airctl users add` or the API.

Each local Airlock user must be associated with one or more roles — this is called “role mapping”.

Non-interactive users

Airlock supports non-interactive users for automation services (for example, CI/CD pipelines or microservices). Local non-interactive users have a record mapping a name to roles, but no credentials in the database.

Role-based access control (RBAC)

Each Airlock user is assigned one or more roles that determine access to resources and the Airlock API.

Allow and deny rules

Each role contains two rule lists: `allow` and `deny`:

- Everything is denied by default.
- `deny` rules are evaluated first and take precedence.
- A rule consists of resources and verbs.

Example `allow` rule for listing recorded SSH or Kubernetes sessions:

```
allow:
  - resources: [session]
    verbs: [list]
```

Principals

Roles define which principals (for example, Linux OS users or Kubernetes groups) users with that role may use:

```
spec:
  allow:
    logins: [ubuntu]
    kubernetes_groups: [viewer]
```

If a user has multiple roles, the principal lists are merged.

Labels

Role labels define which resources the rules apply to. Example granting access to SSH nodes and Kubernetes clusters:

```
spec:
  allow:
    node_labels:
      'environment': '^test|staging$'
    kubernetes_labels:
      'region': 'us-west-*'
      'cluster_name': '^us.*\.example\.com$'
```

Label matching rules:

- For an `allow` rule to match, all labels in the rule must match.
- For a `deny` rule to match, any single label match is sufficient.

Example: user Alice has roles `dev` and `prod`. The `dev` role allows SSH as `root` and full Kubernetes access as `system:masters` for resources with labels `test` or `stage`. The `prod` role allows SSH as `ubuntu` and Kubernetes access as `view` for resources with the label `prod`.

- Alice can SSH as `root` to a server labelled `test` or `stage`.
- Alice cannot SSH as `root` to a server labelled `prod` — the `prod` role only permits `ubuntu`.

Role templates

Roles support template variables:

```
spec:
  allow:
    logins: ['{{internal.logins}}']
    kubernetes_groups: ['{{external.groups}}']
```

Any role with variable interpolation is considered a role template.

Interpolation rules:

- If `external.groups` is the list `["dev", "prod"]`, the expression `["{{external.groups}}"]` interpolates to `["dev", "prod"]`.
- If a variable is absent, the expression `"{{external.groups}}"` produces an empty string.
- Invalid values (for example, `-foo` as a Unix login) are omitted.

Role templates are evaluated at the moment of resource access by the proxy or node. Variables from the identity provider are encoded in X.509 certificate extensions.

Role conditions

The `where` field restricts access by conditions. Example — access only to one's own sessions:

```
kind: role
metadata:
  name: only-own-sessions
spec:
  allow:
    rules:
    - resources: [session]
      verbs: [list, read]
      where: contains(session.participants, user.metadata.name)
```

Role options

In addition to `allow` and `deny` rules, roles define connection options:

```
kind: role
version: v5
metadata:
  name: relaxed
spec:
  options:
    max_session_ttl: 8h
    lock: strict
```

When options conflict across multiple roles, Airlock selects the most restrictive value (for example, the smaller `max_session_ttl` or `strict` mode over `best_effort`).

Access Requests

Roles can allow requesting elevated privileges — additional roles or individual resources. Roles define who may review requests and how many approvals or denials are required. In Airlock this feature is available through the web interface, `airsh`, and the built-in `requester` and `reviewer` roles.

```
spec:
  allow:
    review_requests:
      roles: ['dbadmin']
    request:
      roles: ['common', 'dev-*']
      thresholds:
        - approve: 2
          deny: 1
```

Proxy Service

Airlock Proxy Service is an identity-aware proxy with a built-in web interface. Key capabilities:

- Authenticating users through SSO or local credentials for SSH and Windows desktop access via the web interface.
- Intercepting traffic over SSH, Kubernetes, HTTPS, and database protocols with client authentication verification.
- Recording commands, API calls, and requests and sending them to the audit log.
- Reverse tunnels for connecting servers and proxies behind firewalls.
- TLS routing — multiplexing all ports and protocols onto a single TLS port.

> A minimal Airlock cluster consists of Auth Service and Proxy Service. In a home lab, both services can be run from a single `airlock` binary.

Web interface

In the web interface, Proxy Service uses WSS (secure WebSockets) to proxy the target resource — an SSH server or desktop.

> When using the web interface, Proxy Service terminates TLS traffic and reformats data for the client connection.

Identity-Aware Proxy (IAP) mode

In IAP mode, users initiate SSO or local login to sign public keys on the client machine.

> IAP mode is considered more secure than web interface access: private keys never leave the client, the connection to the resource is mutually authenticated, and the browser is used less — reducing the risk of CSRF attacks and cookie theft.

Tunnels

In this mode, resources behind a firewall establish reverse tunnels to the proxy. The proxy forwards client connections to target resources through these tunnels.

For example, a user connects to a Kubernetes cluster behind a firewall through two tunnels.

> All described modes are enabled in Proxy Service by default. No special configuration is required unless you need to disable individual modes.

Multiple Proxy Service instances

An Airlock cluster can include **multiple Proxy Service instances** running in parallel. This is the standard approach for scaling and improving fault tolerance.

Goal	How multiple proxies help
Availability	A single node failure does not block access: traffic switches to remaining instances
Load distribution	A load balancer distributes user sessions and connections across proxies
Latency	Proxies in different data centres or regions shorten the path to users and agents

Users connect to a single address (for example, `https://airlock.example.com`) through a load balancer. When agents register, they open reverse tunnels to **all** Proxy Service instances in the cluster, so any proxy can route traffic to the required agent.

> ⚠ The load balancer must use round-robin (or equivalent without session affinity). Sticky sessions are not suitable for Proxy Service.

For more on connecting agents to multiple proxies: join token.

Installation and Configuration

Instructions for installing and configuring Airlock for administrators.

- Installation
- Agents
- Authentication
- Access Policies

Installation

- Binary Installation — `airlock`, `airsh`, `airctl`
- Server on Linux — demo cluster with Auth + Proxy
- Server in Kubernetes — deployment via Helm

Binary Installation

This guide covers installing Airlock binaries on Linux:

- `airlock` — server and agent
- `airsh` — command-line client
- `airctl` — administration tool
- `airbot` — Machine ID agent

Client tools (`airsh`, `airctl`, `airbot`) must be the same major version as the cluster they connect to. Airlock servers are compatible with clients of the same major version or one major version older. Clients of a newer major version are not supported.

Supported Platforms

Airlock is supported on the following Linux distributions. All listed platforms support `airlock`, `airctl`, `airsh`, `airbot`, and the web interface.

Supported Linux Distributions

Distribution	Supported Versions
ALT Linux	8 SP (release 10), p10, 10.0, 10.1, 10.2, 11
Astra Linux Special Edition	1.7, 1.8
CentOS	7, 8, 9, 10*
Debian	10, 11, 12, 13
openSUSE	15.4, 15.5, 15.6
Rocky Linux	8, 9
Ubuntu	18.04, 20.04, 22.04, 24.04, 26.04
RED OS	7.3, 8.0
ROSA Server	7.9, 12.4, 12.5, 12.6
MOS OS	15.4, 15.5

Enhanced session recording requires Linux kernel v5.8+. Package manager installation (`apt`, `yum`, `zypper`) requires systemd. Repositories do not include packages for every distribution variant — you may need to replace `ID` with `ID_LIKE` for the nearest supported distribution.

Installation on Linux

The installation includes all components: `airlock`, `airsh`, `airctl`, and `airbot`.

Installation via Script

Run on the target host:

```
curl https://deckhouse.ru/downloads/airlock/install.sh | bash -s
```

The script updates the package manager repository and installs the latest version of Airlock.

Installing a Specific Version

To install a specific version, append it to the command:

```
curl https://deckhouse.ru/downloads/airlock/install.sh | bash -s v15.0.0
```

Verifying the Installation

```
airlock version
airsh version
airctl version
```

Installation from a tar Archive

Alternatively, download the archive from the official downloads page:

```
export version=v15.0.0
export os=linux
export arch=amd64
curl -O https://deckhouse.ru/downloads/airlock/airlock-${version}-${os}-${arch}-
bin.tar.gz
tar -xzf airlock-${version}-${os}-${arch}-bin.tar.gz
sudo ./install
```

Checksums

SHA256 checksums are available on the downloads page to verify binary integrity:

```
export version=v15.0.0
export os=linux
export arch=amd64
curl https://deckhouse.ru/downloads/airlock/airlock-${version}-${os}-${arch}-
bin.tar.gz.sha256
```

Next Steps

After installing the binaries:

- Deploy a cluster on a Linux server or in Kubernetes
- Connect your infrastructure using agents

Deploying the Server on Linux

Airlock provides connectivity, authentication, access control, and auditing for infrastructure. The platform includes an identity-aware proxy, a certificate authority with short-lived certificates, a unified access control system, and tunneling for accessing resources behind a firewall.

This guide shows how to deploy a single-node Airlock cluster on a Linux server. After deployment, you can configure RBAC, register resources, and secure test or home lab environments.

Demo Cluster Components

- **Auth Service** — the cluster certificate authority. Issues certificates and performs authentication checks. Typically not accessible from outside the private network.
- **Proxy Service** — the cluster frontend: handles user requests, forwards credentials to Auth Service, and communicates with agents.
- **SSH Service** — an SSH server with short-lived certificates, RBAC, session recording, and other Airlock capabilities.

Prerequisites

- A Linux host with port `443` open for inbound traffic. You must be able to install and run software. Initial setup requires SSH access (open the SSH port in addition to `443`).
- A multi-factor authentication app (Authy, Google Authenticator, 1Password, etc.).
- **One** of the following:
  - A registered domain name.
  - An authoritative organizational DNS server and an existing certificate authority. The browser must use the organizational DNS server.

Step 1 of 4. Configure DNS

Airlock uses TLS to secure Proxy Service and Auth Service, so a domain name is required for certificate verification. Create two `A` DNS records pointing to the IP address of your Linux host. For the domain `airlock.example.com`:

Domain	Purpose
`airlock.example.com`	User and service traffic to Proxy Service
`*.airlock.example.com`	Traffic to web applications registered in Airlock

Step 2 of 4. Install Airlock on the Linux Host

Installation

```
curl https://deckhouse.ru/downloads/airlock/install.sh | bash -s
```

Configuration

Generate the configuration file using `airlock configure`. The command requires a TLS certificate and private key.

Public network deployment with Let’s Encrypt

Let’s Encrypt verifies domain ownership via HTTPS on port 443 of Proxy Service. Replace `airlock.example.com` with your cluster domain and `admin@example.com` with the notification email:

```
sudo airlock configure -o file \
  --acme --acme-email=admin@example.com \
  --cluster-name=airlock.example.com
```

Port `443` on the Proxy Service host must accept traffic from any source.

Private network deployment

Place the private key and certificate chain at `/var/lib/airlock/privkey.pem` and `/var/lib/airlock/fullchain.pem`. The certificate must match the cluster domain.

```
sudo airlock configure -o file \
  --cluster-name=airlock.example.com \
  --public-addr=airlock.example.com:443 \
  --cert-file=/var/lib/airlock/fullchain.pem \
  --key-file=/var/lib/airlock/privkey.pem
```

Starting

```
sudo systemctl enable airlock
sudo systemctl start airlock
sudo systemctl status airlock
```

Open the web interface via HTTPS: `https://airlock.example.com`. The welcome screen should appear.

Step 3 of 4. Create a User and MFA

Create an `airlock-admin` administrator with login rights as `root`, `ubuntu`, or other local accounts:

```
sudo airctl users add airlock-admin --roles=editor,access --logins=root,ubuntu
```

The command outputs a one-time link to complete user registration (valid for 1 hour):

```
User "airlock-admin" has been created but requires a password. Share this URL with the
user to complete user setup, link is valid for 1h:
https://airlock.example.com:443/web/invite/...
```

Follow the link, set a password, and configure OTP using the QR code.

> Accounts specified in the `--logins` flag must exist on the Linux host. Create them with `adduser <login>` or specify the current user: `airctl users add airlock-admin $(whoami)`.

CLI Login

Install `airsh` on your workstation (if not already installed) and log in to the cluster:

```
airsh login --proxy=airlock.example.com --user=airlock-admin
```

Example output:

```
Profile URL:        https://airlock.example.com:443
Logged in as:       airlock-admin
Cluster:            airlock.example.com
Roles:              access, editor
Logins:             root, ubuntu
Kubernetes:         enabled
Valid until:        ...
```

Step 4 of 4. Register Infrastructure

In the web interface, open the section for the required resource type (**Servers**, **Applications**, **Kubernetes**, etc.) and follow the registration instructions.

The **Servers** tab will already show the Linux server running the cluster.

Next Steps

Airlock agents proxy traffic to servers, databases, Kubernetes clusters, and Windows desktops. See the Agents section and join-token enrollment for details.

Deploying in Kubernetes

Airlock provides unified secure access to Kubernetes clusters. This guide covers deploying Auth Service and Proxy Service in Kubernetes using Helm.

If Auth Service and Proxy Service are already running on another platform, use the existing Airlock cluster and connect Kubernetes using the Kubernetes agent guide.

Prerequisites

- A registered domain name (for TLS via Let's Encrypt and Proxy Service verification by clients).
- A Kubernetes cluster with load balancer and persistent volume support.
- A Persistent Volume for storing Auth Service state:

```
kubectl get pv
```

If no volumes exist, enable dynamic volume provisioning or create a PV manually. Check for a default StorageClass:

```
kubectl get storageclasses
```

- The `airsh` client installed on your workstation.
- Kubernetes >= 1.28, Helm >= 3.14:

```
helm version
kubectl version
```

Step 1 of 2. Install Airlock

Add the Helm Repository

```
helm repo add airlock https://charts.airlock.deckhouse.ru
helm repo update
```

Create the Namespace

```
kubectl create namespace airlock-cluster
kubectl label namespace airlock-cluster 'pod-security.kubernetes.io/enforce=baseline'
kubectl config set-context --current --namespace=airlock-cluster
```

Helm Values File

Set `clusterName` (for example, `airlock.example.com`) and `acmeEmail` for Let's Encrypt:

```
cat << EOF > airlock-cluster-values.yaml
clusterName: airlock.example.com
proxyListenerMode: multiplex
acme: true
acmeEmail: admin@example.com
EOF
```

Install the `airlock-cluster` Chart

```
helm install airlock-cluster airlock/airlock-cluster \
  --create-namespace \
  --values airlock-cluster-values.yaml
```

Wait for the Auth Service and Proxy Service pods to start:

```
kubectl get pods
```

Expected output:

```
NAME                                        READY   STATUS    RESTARTS   AGE
airlock-cluster-auth-000000000-00000     1/1     Running   0          114s
airlock-cluster-proxy-0000000000-00000   1/1     Running   0          114s
```

Configure DNS

Get the external load balancer address:

```
kubectl get services/airlock-cluster
```

Create DNS records for `airlock.example.com` and `*.airlock.example.com`:

Record Type	Domain	Value
A or CNAME	`airlock.example.com`	Load balancer IP or domain
A or CNAME	`*.airlock.example.com`	Load balancer IP or domain

Verify cluster availability:

```
curl https://airlock.example.com/webapi/ping
```

Step 2 of 2. Create a Local User

Local users provide a reliable fallback when SSO is unavailable. Create a `member` role with access to the Kubernetes `system:masters` group:

```
kind: role
version: v7
metadata:
  name: member
spec:
  allow:
    kubernetes_labels:
      '*': '*'
    kubernetes_groups:
    - system:masters
```

```
kubectl exec -i deployment/airlock-cluster-auth -- airctl create -f - < member.yaml
```

Create a user and get the invitation link:

```
kubectl exec -ti deployment/airlock-cluster-auth -- airctl users add myuser --roles=me
mber,access,editor
```

Follow the link from the command output and activate the account.

Log in via `airsh`:

```
airsh login --proxy=airlock.example.com:443 --user=myuser
airsh kube ls
```

For safe kubeconfig management, use a separate file:

```
KUBECONFIG=$HOME/airlock-kubeconfig.yaml airsh kube login airlock.example.com
KUBECONFIG=$HOME/airlock-kubeconfig.yaml kubectl get -n airlock-cluster pods
```

Troubleshooting

If connectivity is not working, check pod status:

```
kubectl get pods -n airlock-cluster
```

The Auth Service pod may remain in `Pending` state if a PersistentVolumeClaim cannot be provisioned. Use `kubectl describe pod` and `kubectl logs` for diagnostics:

```
kubectl get events --sort-by='.metadata.creationTimestamp' -A
```

Next Steps

- Configure single sign-on (SSO) for production deployments.
- Connect additional Kubernetes clusters via the Kubernetes agent.
- Restrict Kubernetes access rights using Airlock RBAC instead of `system:masters`.

Airlock Agents

Airlock agents are Airlock instances configured to proxy traffic to infrastructure resources: servers, databases, Kubernetes clusters, Windows desktops, and web applications.

Architecture overview

Services

Each Airlock process can run one or more **services**. A service is enabled in the instance configuration file.

Agent pools

Agents typically run in the same private networks as the resources they proxy. They should be the only clients with direct access to a resource without Airlock.

Agents establish reverse SSH tunnels to the Proxy Service. The Proxy Service is reachable from the internet over HTTPS, and agents do not need open ports or a public address.

The Proxy Service uses reverse tunnels to forward traffic over supported protocols to an available agent. The agent enforces RBAC rules and forwards traffic to the resources.

Connecting agents

Initial enrollment into the cluster

Agents need to establish trust with the Auth Service. Several enrollment methods are available to automate this in your environment — see join-token enrollment.

On first startup, the Airlock process reads its configuration and determines which services are enabled. Each service independently connects to the Auth Service, which checks for a **join token** for that service. On success, the Auth Service issues credentials signed for that specific service.

Adding a new service to an existing agent

Agent credentials are signed for specific services. To run new services, repeat the enrollment procedure:

1. Generate a new join token for all agent services, including the new ones.
2. Update the token in the configuration (`airlock.join_params` or `airlock.auth_token`).
3. Remove the agent state directory (default `/var/lib/airlock` , field `airlock.data_dir`).
4. Restart the agent.

It is recommended to deploy agents via infrastructure as code and update the service configuration in the IaC project followed by a redeploy.

Registering infrastructure

Two ways to register resources through agents:

- **Static** — specify the resource in the agent configuration file.
- **Dynamic** — apply a configuration resource (YAML via `airctl`) so that Airlock manages proxying.

Detailed instructions by resource type:

- SSH node — Linux servers
- OpenSSH — legacy servers with OpenSSH
- Kubernetes agent — Kubernetes clusters
- Windows RDP — Windows desktops
- Databases — PostgreSQL, MySQL, OpenSearch
- Web applications — HTTP/HTTPS applications

Guides in this section

Document	Description
join-token	Enrolling an agent with a one-time token
ssh-node	Registering a Linux server with SSH Service
openssh	Accessing OpenSSH servers without an agent
kubernetes-agent	Connecting a Kubernetes cluster
rdp-windows	Accessing Windows desktops via RDP
databases	Database Service overview
database-postgresql	PostgreSQL
database-mysql	MySQL / MariaDB
database-opensearch	OpenSearch / Elasticsearch
applications	Internal web applications

Join-token enrollment

This guide shows how to register an Airlock process with one or more services in a cluster by presenting a **join token**.

Prerequisites

- A Linux server for the Airlock process (a Linux-based virtual machine or container).
- This guide enrolls an SSH Service (`node`); the same approach applies to Kubernetes Service, Database Service, Proxy Service, and other services.
- `airctl` installed on the administrative machine with cluster access.

> A join token works with a single Proxy Service and with multiple Proxy Services behind a load balancer or DNS with multiple A records. During enrollment, the process establishes a tunnel to each Proxy Service. The load balancer must use round-robin, not sticky sessions.

Step 1 of 3. Install Airlock

On the Linux host:

```
curl https://deckhouse.ru/downloads/airlock/install.sh | bash -s
```

Step 2 of 3. Enroll in the cluster

Generate a token

On the local machine with `airctl`:

```
airctl tokens add --ttl=5m --type=node
```

Example output:

```
The invite token: abc123...
This token will expire in 5 minutes.

Run this on the new node to join the cluster:

> airlock start \
   --roles=node \
   --token=abc123... \
   --ca-pin=sha256:... \
   --auth-server=192.0.2.0:3025
```

Supported token types

Type	Airlock service
`app`	Application Service
`auth`	Auth Service
`bot`	Machine ID
`db`	Database Service
`discovery`	Discovery Service
`kube`	Kubernetes Service
`node`	SSH Service
`proxy`	Proxy Service
`windowsdesktop`	Windows Desktop Service

List active tokens:

```
airctl tokens ls
```

> ⚠ Prefer short-lived tokens over static ones. Static tokens in the Auth Service configuration are easier to steal or may be inadvertently leaked.

Start the process with the token

On the new agent host (replace `JOIN_TOKEN` and `airlock.example.com`):

```
sudo airlock configure \
  --roles=node \
  --token=JOIN_TOKEN \
  --proxy=airlock.example.com:443 \
  -o file
```

For SSH Service you can use `airlock node configure` without the `--roles=node` flag:

```
sudo airlock node configure \
  --output=file:///etc/airlock.yaml \
  --token=JOIN_TOKEN \
  --proxy=airlock.example.com:443
```

Start the service:

```
sudo systemctl enable airlock
sudo systemctl start airlock
```

Verify node registration:

```
airctl nodes ls
```

Direct connection to the Auth Service

If the Proxy Service is unreachable from the agent host, connect directly to the Auth Service only when no other option is available. Minimize inbound traffic to the Auth Service.

Get the CA pin:

```
airctl status
```

Configure the agent:

```
sudo airlock configure \
  --roles=node \
  --token=JOIN_TOKEN \
  --auth-server=airlock.example.com:3025 \
  -o file
```

Set the CA pin in `/etc/airlock.yaml` (field `airlock.ca_pin`):

```
sudo sed -i 's|  ca_pin: ""|  ca_pin: "sha256:..."|' /etc/airlock.yaml
sudo systemctl restart airlock
```

> The CA pin becomes invalid after CA rotation (`airctl auth rotate`).

Step 3 of 3. Revoke the token

To prevent further use of the token:

```
airctl tokens rm TOKEN_TO_DELETE
```

Next steps

- SSH node — complete server access scenario
- Kubernetes agent — connecting a Kubernetes cluster

Linux SSH node

A server is protected by running the SSH Service on it and registering it in the Airlock cluster. RBAC controls access, and the audit log records activity.

The SSH Service opens a reverse SSH tunnel to the Proxy Service; client traffic flows through this tunnel — similar to a bastion host pattern.

> Do not run the SSH Service as a Kubernetes pod to access cluster nodes — the pod may end up on a different server than the one you need access to.

Prerequisites

- A Linux host (Ubuntu 20.04, CentOS 8, Debian 10, etc.) — the future Airlock node.
- A running Airlock cluster and `airctl` on the administrative machine.
- Port 22 temporarily open for initial configuration.

Step 1 of 4. Install Airlock on the server

On the target host:

```
curl https://deckhouse.ru/downloads/airlock/install.sh | bash -s
```

Step 2 of 4. Add the server to the cluster

Create a join token

On your workstation, save the token to a file:

```
airctl tokens add --type=node --format=text > token.file
```

Enroll the server

Copy `token.file` to the server in a secure directory and generate the configuration (replace `airlock.example.com`):

```
sudo airlock node configure \
  --output=file:///etc/airlock.yaml \
  --token=/path/to/token.file \
  --proxy=airlock.example.com:443
```

Start the SSH Service:

```
sudo systemctl enable airlock
sudo systemctl start airlock
sudo systemctl status airlock
```

Create a user for the web interface

```
airctl users add myuser --roles=editor,access,reviewer --logins=root,ubuntu
```

Follow the link in the command output, set a password and MFA. The server will appear in the web interface under the **Servers** tab.

Step 3 of 4. SSH to the server

Log in to the cluster

```
airsh login --proxy=airlock.example.com --user=myuser
```

Example profile:

```
Profile URL:      https://airlock.example.com:443
Logged in as:     myuser
Cluster:          airlock.example.com
Roles:            access, editor, reviewer
Logins:           root, ubuntu
```

List nodes

```
airsh ls
```

```
Node Name        Address        Labels
---------------- -------------- -------------------------------------
bastion-host     ← Tunnel
web-server       127.0.0.1:3022 env=prod, hostname=web-server
```

Connect

```
airsh ssh root@web-server
```

Equivalent using standard `ssh`:

```
airsh config > ssh_config_airlock
ssh -F ssh_config_airlock root@web-server.airlock.example.com
```

All commands are recorded in the audit log.

Step 4 of 4. Review cluster resources

```
airctl nodes ls
```

Filter by label:

```
airsh ls env=prod
```

Run a command on all nodes with a label:

```
airsh ssh root@env=prod ls
```

Hardening (optional)

After configuration, close port 22 on the server — access remains available via `airsh ssh`. Port 22 on the Proxy Service host can also be closed.

OpenSSH without an agent

Airlock can proxy SSH to existing **OpenSSH** (`sshd`) servers without installing the `airlock` binary on the node. Servers accept dynamically issued certificates from the Airlock CA.

This is a quick way to connect legacy systems. Going forward, replacing `sshd` with `airlock` is recommended for full RBAC support, session recording without connection interruption, session sharing, and enhanced recording capabilities.

The Proxy Service verifies the user's roles, authenticates to the OpenSSH node using an Airlock CA certificate, and maintains the audit trail.

Prerequisites

- OpenSSH 6.9+ on the client machine: `ssh -V`
- A Linux host running `sshd` 7.4+, **without** Airlock; the SSH port open to the Proxy Service
- `airctl` and `airsh` on the administrative machine

Step 1. Node resource in the cluster

Create `openssh-node-resource.yaml` :

```
kind: node
version: v2
sub_kind: openssh
metadata:
  name: a100fdd0-52db-4eca-a7ab-c3afa7a1564a
  labels:
    env: prod
spec:
  addr: 1.2.3.4:22
  hostname: openssh-node
```

- `spec.addr` — SSH IP and port
- `spec.hostname` — name shown to users in Airlock
- `metadata.name` — UUID (`uuidgen`)

```
airctl create -f openssh-node-resource.yaml
```

Step 2. Trust the Airlock CA in sshd

On the host running `sshd` (replace `airlock.example.com`):

```
export KEY=$(curl 'https://airlock.example.com/webapi/auth/export?type=openssh' | sed
"s/cert-authority\ //")
sudo bash -c "echo \"$KEY\" > /etc/ssh/airlock_openssh_ca.pub"
sudo bash -c "echo 'TrustedUserCAKeys /etc/ssh/airlock_openssh_ca.pub' >> /etc/ssh/
sshd_config"
sudo systemctl restart sshd
```

Step 3. Host certificate

Create a role for issuing host certificates (`host-certifier.yaml`):

```
kind: role
version: v5
metadata:
  name: host-certifier
spec:
  allow:
    rules:
      - resources: [host_cert]
        verbs: [list, create, read, update, delete]
```

```
airctl create -f host-certifier.yaml
```

Assign the role to your user via `airctl users update` .

Get the node UUID:

```
airctl get node/openssh-node --format=json | jq -r ".[0].metadata.name"
```

Issue a host certificate:

```
ADDR=1.2.3.4,openssh-node,a100fdd0-52db-4eca-a7ab-c3afa7a1564a
airctl auth sign --host=${ADDR} --format=openssh --out=myhost
```

Copy `myhost` and `myhost-cert.pub` to the server under `/etc/ssh` with permissions `0600` , and add to `sshd_config` :

```
HostKey /etc/ssh/myhost
HostCertificate /etc/ssh/myhost-cert.pub
```

```
sudo systemctl restart sshd
```

Step 4. SSH client configuration

Log in to the cluster:

```
airsh login --proxy=airlock.example.com --user=myuser
airsh config > ssh_config_airlock
```

Step 5. Connect

```
USER=ubuntu
CLUSTER=airlock.example.com
PORT=22
ADDR_NODE=openssh-node
ssh -p ${PORT} -F ssh_config_airlock "${USER}@${ADDR_NODE}.${CLUSTER}"
```

Port 22 is specified explicitly: an OpenSSH node has no reverse tunnel on port 3022; the Proxy Service connects directly to `spec.addr`.

> Airlock uses OpenSSH certificates. The `Principals` section of the certificate must include all names and addresses you connect through.

Next steps

For new servers, an SSH node with an agent is preferred.

Kubernetes agent

This guide shows how to register a Kubernetes cluster by deploying the Kubernetes Service (chart `airlock-kube-agent`) on the cluster to be protected.

The Kubernetes Service pod detects that it is running inside Kubernetes and automatically registers the cluster in Airlock.

Prerequisites

- A running Airlock cluster, `airctl` and `airsh` at a version compatible with the server:

```
airctl version
airsh version
```

- Kubernetes >= 1.28, Helm >= 3.14, `kubectl` with access to the target cluster:

```
helm version
kubectl version
```

- Administrator access to the Airlock web interface (for the enrollment wizard) or the ability to perform the steps manually below.

Step 1 of 3. RBAC in Airlock and Kubernetes

Airlock role

Create `kube-access.yaml`:

```
kind: role
metadata:
  name: kube-access
version: v6
spec:
  allow:
    kubernetes_labels:
      '*': '*'
    kubernetes_resources:
      - kind: 'pod'
        namespace: '*'
        name: '*'
    kubernetes_groups:
    - viewers
  deny: {}
```

```
airctl create -f kube-access.yaml
```

Assign the role to the user via `airctl users update`.

ClusterRoleBinding in Kubernetes

The `viewers` group must have permissions in the cluster. Create `viewers-bind.yaml`:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: viewers-crb
subjects:
- kind: Group
  name: viewers
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: ClusterRole
  name: view
  apiGroup: rbac.authorization.k8s.io
```

```
kubectl apply -f viewers-bind.yaml
```

Step 2 of 3. Deploy the Kubernetes Service

Helm repository

```
helm repo add airlock https://charts.airlock.deckhouse.ru
helm repo update
```

Manual installation

Generate a join token:

```
airctl tokens add --type=kube --ttl=1h
```

Create a values file `airlock-kube-agent-values.yaml` (substitute the token and proxy address):

```
roles: kube
proxyAddr: airlock.example.com:443
authToken: "YOUR_KUBE_JOIN_TOKEN"
kubeClusterName: my-k8s-cluster
```

Install the chart into the `airlock-agent` namespace:

```
kubectl create namespace airlock-agent
helm install airlock-agent airlock/airlock-kube-agent \
  --namespace airlock-agent \
  --values airlock-kube-agent-values.yaml
```

Verify the pod:

```
kubectl get pods -n airlock-agent
```

Via the web interface: **Enroll New Resource → Kubernetes** — copy the generated command and run it on your workstation.

Step 3 of 3. Verify access

Log in to Airlock:

```
airsh login --proxy=airlock.example.com:443 --user=myuser
```

List clusters:

```
airsh kube ls
```

Obtain credentials for `kubectl`:

```
airsh kube login my-k8s-cluster
kubectl get pods -n airlock-agent
```

Expected output:

```
NAME               READY   STATUS    RESTARTS   AGE
airlock-agent-0   1/1     Running   0          8m
```

Next steps

- Restrict access using Kubernetes labels and Airlock roles instead of broad `view` rights across all namespaces.
- If Auth Service and Proxy Service are already deployed separately, an agent without the `airlock-cluster` chart is sufficient.

Database access

Airlock Database Service proxies TCP traffic to databases using their native protocols. Users connect via `airsh db connect`, and Airlock validates certificates and enforces RBAC.

Supported self-hosted databases

Database	Airlock protocol	Guide
PostgreSQL	`postgres`	PostgreSQL
MySQL / MariaDB	`mysql`	MySQL / MariaDB
OpenSearch / Elasticsearch	`elastic`	OpenSearch

General flow

1. Database Service connects to the Airlock cluster (join token of type `db`).
2. Mutual TLS authentication (mTLS) is configured between Database Service and the database.
3. The administrator creates RBAC roles with `db_labels`, `db_names`, `db_users` permissions.
4. The user runs `airsh db ls` and `airsh db connect`.

Join token for Database Service

```
airctl tokens add --type=db --format=text > /tmp/db-token
```

Start Database Service (Linux)

```
sudo airlock db configure create \
  --token=/tmp/db-token \
  --name=example-db \
  --proxy=airlock.example.com:443 \
  --protocol=postgres \
  --uri=postgres.example.com:5432 \
  --output file:///etc/airlock.yaml

sudo systemctl enable --now airlock
```

Role for database access

```
airctl create <<EOF
kind: role
version: v3
metadata:
  name: db
spec:
  allow:
    db_labels:
      '*': '*'
    db_names:
    - '*'
    db_users:
    - '*'
EOF

airctl users add --roles=access,db alice
```

For more on policies, see Database access policies.

User connection

```
airsh login --proxy=airlock.example.com --user=alice
airsh db ls
airsh db connect --db-user=postgres --db-name=postgres example-db
```

PostgreSQL

This guide covers connecting self-hosted PostgreSQL to Airlock via Database Service with mutual TLS authentication.

Prerequisites

- A PostgreSQL instance in your infrastructure.
- The `psql` client in `PATH` on the user's workstation.
- A host for Database Service (Linux or Kubernetes).
- Access to `airctl` on the administrator's machine.

Step 1. Join token and Airlock user

```
airctl tokens add --type=db --format=text > /tmp/token

airctl users add --roles=access,db alice
```

Step 2. Certificates for mTLS

Database Service connects to PostgreSQL over TLS. Generate a certificate for the host from which the agent reaches PostgreSQL:

```
airctl auth sign --format=db --host=db.example.com --out=server --ttl=2190h
```

This creates the files `server.cas`, `server.crt`, `server.key` — place them on the PostgreSQL server.

Step 3. Configure PostgreSQL

In `postgresql.conf`:

```
ssl = on
ssl_cert_file = '/path/to/server.crt'
ssl_key_file = '/path/to/server.key'
ssl_ca_file = '/path/to/server.cas'
```

In `pg_hba.conf`, add client certificate authentication:

```
hostssl all             all             ::/0                    cert
hostssl all             all             0.0.0.0/0               cert
```

Make sure there are no `md5` rules with higher priority for the same connections — otherwise Airlock cannot authenticate by certificate.

Restart PostgreSQL.

Step 4. Database Service

Linux

```
curl https://deckhouse.ru/downloads/airlock/install.sh | bash -s

sudo airlock db configure create \
  --token=/tmp/token \
  --name=example-postgres \
  --proxy=airlock.example.com:443 \
  --protocol=postgres \
  --uri=postgres.example.com:5432 \
  --output file:///etc/airlock.yaml

sudo systemctl enable --now airlock
```

Kubernetes

Install the `airlock-kube-agent` chart with the `db` role and database parameters (see Kubernetes agent for adding the Helm repository).

Step 5. Connect

```
airsh login --proxy=airlock.example.com --user=alice
airsh db ls
airsh db connect --db-user=postgres --db-name=postgres example-postgres
```

End the session:

```
airsh db logout example-postgres
```

The list of accessible databases depends on the RBAC role — see Database access policies.

MySQL / MariaDB

This guide covers connecting self-hosted MySQL or MariaDB to Airlock via Database Service.

Prerequisites

- A MySQL or MariaDB instance.
- A host for Database Service.
- Access to `airctl`.

Step 1. Join token

```
airctl tokens add --type=db --format=text > /tmp/token
```

Step 2. Certificates for mTLS

```
airctl auth sign --format=db --host=db.example.com --out=server --ttl=2190h
```

Copy `server.cas`, `server.crt`, `server.key` to the MySQL/MariaDB host.

Step 3. Configure MySQL

In `mysql.cnf`:

```
[mysqld]
require_secure_transport=ON
ssl-ca=/path/to/server.cas
ssl-cert=/path/to/server.crt
ssl-key=/path/to/server.key
```

Step 4. Configure MariaDB

In `mysql.cnf`:

```
[mariadb]
require_secure_transport=ON
ssl-ca=/path/to/server.cas
ssl-cert=/path/to/server.crt
ssl-key=/path/to/server.key
```

Database user

Create a user with certificate authentication:

```
CREATE USER 'alice'@'%' REQUIRE SUBJECT '/CN=alice';
GRANT ALL ON `mydb`.* TO 'alice'@'%';
```

For an existing user:

```
ALTER USER 'alice'@'%' REQUIRE SUBJECT '/CN=alice';
```

Airlock authenticates users by certificate; a password is not used for such accounts.

Step 5. Database Service and Airlock user

```
airctl users add --roles=access,db alice

sudo airlock db configure create \
  --token=/tmp/token \
  --name=example-mysql \
  --proxy=airlock.example.com:443 \
  --protocol=mysql \
  --uri=mysql.example.com:3306 \
  --output file:///etc/airlock.yaml

sudo systemctl enable --now airlock
```

Step 6. Connect

```
airsh login --proxy=airlock.example.com --user=alice
airsh db ls
airsh db connect --db-user=root --db-name=mysql example-mysql
```

The `mysql` or `mariadb` client must be in `PATH`.

```
airsh db logout example-mysql
```

OpenSearch / Elasticsearch

Airlock supports self-hosted OpenSearch and Elasticsearch via the `elastic` protocol. Mutual TLS with client certificates is required.

Prerequisites

- Self-hosted OpenSearch or Elasticsearch (managed cloud services without client certificate support are not compatible).
- A host for Database Service.
- Access to `airctl`.

Step 1. Join token and Database Service

```
airctl tokens add --type=db --format=text > /tmp/token

sudo airlock db configure create \
  --token=/tmp/token \
  --name=my-opensearch \
  --proxy=airlock.example.com:443 \
  --protocol=elastic \
  --uri=opensearch.example.com:9200 \
  --output file:///etc/airlock.yaml

sudo systemctl enable --now airlock
```

Step 2. Airlock user

```
airctl users add --roles=access,db alice
```

Step 3. Role mapping in OpenSearch

Map the Airlock user to an OpenSearch role:

```
curl -u admin:password -X POST "https://opensearch.example.com:9200/_plugins/
_security/api/rolesmapping/all_access" \
  -H 'Content-Type: application/json' -d '{
  "users": ["alice"]
}'
```

For Elasticsearch, use the `_security/role_mapping/` API with a rule on the `username` field.

Example for Elasticsearch:

```
curl -u elastic:password -X POST "https://opensearch.example.com:9200/_security/
role_mapping/mapping1?pretty" \
  -H 'Content-Type: application/json' -d '{
  "roles": ["user"],
  "enabled": true,
  "rules": {
    "field": { "username": "alice" }
  }
}'
```

Step 4. Mutual TLS

```
airctl auth sign --format=elasticsearch --host=opensearch.example.com --out=elastic --
ttl=2160h
```

Configure the files `elastic.cas`, `elastic.crt`, `elastic.key` in the cluster configuration.

For OpenSearch in `opensearch.yml`:

```
plugins.security.ssl.http.enabled: true
plugins.security.ssl.http.pemcert_filepath: /path/to/elastic.crt
plugins.security.ssl.http.pemkey_filepath: /path/to/elastic.key
plugins.security.ssl.http.pemtrustedcas_filepath: /path/to/elastic.cas
plugins.security.ssl.http.clientauth_mode: REQUIRE
```

For Elasticsearch 7.x+ with X-Pack in `elasticsearch.yml`:

```
xpack.security.http.ssl:
  certificate_authorities: /path/to/elastic.cas
  certificate: /path/to/elastic.crt
  key: /path/to/elastic.key
  enabled: true
  client_authentication: required
  verification_mode: certificate

xpack.security.authc.realms.pki.pki1:
  order: 1
  enabled: true
  certificate_authorities: /path/to/elastic.cas
```

Restart the cluster.

Step 5. Connect

```
airsh login --proxy=airlock.example.com --user=alice
airsh db ls
airsh db connect my-opensearch --db-user=alice
```

Tunnel for GUI clients:

```
airsh proxy db my-opensearch --db-user=alice --tunnel
```

Connect your client (OpenSearch Dashboards, Elasticvue, etc.) to `http://localhost:<port>`.

Web applications

Airlock Application Service proxies HTTP/HTTPS traffic to internal web applications: monitoring dashboards, CI/CD systems, wikis, Kubernetes Dashboard, and other services accessible only from a private network.

Users open applications through the Airlock web interface or at `https://<app-name>.<proxy-domain>`.

Prerequisites

- A running Airlock cluster (Auth + Proxy).
- A `*.airlock.example.com` DNS record pointing to the Proxy Service (for application subdomains).
- A TLS certificate for the Proxy Service and a wildcard certificate for application subdomains.
- A host, Docker container, or Kubernetes for the application and Application Service.
- Access to `airctl`.

Step 1. Join token

```
airsh login --user=admin --proxy=airlock.example.com
airctl tokens add \
  --type=app \
  --app-name=grafana \
  --app-uri=http://localhost:3000 \
  --format=text > /tmp/token
```

Step 2. Start the application

Example: Grafana in Docker:

```
docker run --detach --name grafana --publish 3000:3000 grafana/grafana
```

In `grafana.ini`, set the Proxy Service domain:

```
[server]
domain = airlock.example.com
```

Example: Grafana in Kubernetes:

```
helm repo add grafana https://grafana.github.io/helm-charts
helm install example-grafana grafana/grafana \
  --create-namespace \
  --namespace example-grafana
```

Internal cluster URI: `http://example-grafana.example-grafana.svc.cluster.local`.

Step 3. Application Service

Linux / Docker

```
sudo airlock configure \
  --output=file \
  --proxy=airlock.example.com:443 \
  --token=/tmp/token \
  --roles=app \
  --app-name=grafana \
  --app-uri=http://localhost:3000

sudo systemctl enable --now airlock
```

Configuration file

```
version: v3
airlock:
  data_dir: /var/lib/airlock-app
  auth_token: /tmp/token
  proxy_server: airlock.example.com:443
app_service:
  enabled: yes
  apps:
  - name: grafana
    uri: http://localhost:3000
    public_addr: grafana.airlock.example.com
    labels:
      env: prod
auth_service:
  enabled: "no"
ssh_service:
  enabled: "no"
proxy_service:
  enabled: "no"
```

```
sudo airlock start --config=/etc/airlock.yaml
```

Kubernetes

```
JOIN_TOKEN=$(cat /tmp/token)
helm install airlock-kube-agent airlock/airlock-kube-agent \
  --create-namespace \
  --namespace airlock-agent \
  --set roles=app \
  --set proxyAddr=airlock.example.com:443 \
  --set authToken=${JOIN_TOKEN} \
  --set "apps[0].name=grafana" \
  --set "apps[0].uri=http://example-grafana.example-grafana.svc.cluster.local" \
  --set "apps[0].labels.env=dev"
```

Step 4. User and access

```
airctl users add --roles=access appuser
```

Access methods:

1. Airlock web UI → **Applications** tab → **Launch**.
2. Direct URL: `https://grafana.airlock.example.com`.

Application name

The application name (`--app-name`) must be a valid subdomain: up to 63 characters, only `a-z` , `0-9` , `-` .

After registration, the application is available at `<app-name>.<proxy-host>` , for example `grafana.airlock.example.com` . This can be overridden via `public_addr` in the configuration.

Additional options

Application under a subpath

```
- name: k8s-dashboard
  uri: "http://10.0.1.60:8001/api/v1/namespaces/kube-system/services/https:kubernetes-
dashboard:/proxy/"
  public_addr: k8s.airlock.example.com
```

Redirect rewriting

For applications with internal redirects:

```
- name: jenkins
  uri: https://localhost:8001
  public_addr: jenkins.airlock.example.com
  rewrite:
    redirect:
    - localhost
    - jenkins.internal.dev
```

Self-signed application certificate

```
- name: app
  uri: https://localhost:8443
  public_addr: app.airlock.example.com
  insecure_skip_verify: true
```

⚠ Not recommended for production.

Headers and JWT

Airlock can add headers to requests, including a JWT with user information:

```
- name: dashboard
  uri: https://localhost:4321
  public_addr: dashboard.airlock.example.com
  rewrite:
    headers:
    - "Authorization: Bearer {{internal.jwt}}"
```

Application logout

Force logout before the session expires:

```
https://grafana.airlock.example.com/airlock-logout
```

TCP applications

For non-HTTP applications (plain TCP), use TCP App Access — specify the TCP protocol in the Application Service configuration.

Access policies

Application access is controlled via RBAC (`app_labels` in roles). See Access policies.

RDP and Windows

Airlock provides secure passwordless access to Microsoft Windows desktops for local Windows users (not an Active Directory domain).

Prerequisites

- A Linux server for the Windows Desktop Service (the service can be added to an existing Airlock instance using a new join token).
- A physical or virtual Windows machine with Remote Desktop enabled; the RDP port (usually 3389) open to the Linux server running the agent.
- `airctl` on the administrative machine with cluster access.

Step 1 of 4. Prepare Windows

On the Windows computer in the command prompt (`cmd.exe`):

1. Export the Airlock user certificate CA:

```
curl.exe -fo airlock.cer https://airlock.example.com/webapi/auth/export?type=windows
```

1. Download the Airlock Windows Auth installer (substitute the current version):

```
curl.exe -fo airlock-windows-auth-setup-amd64.exe https://deckhouse.ru/downloads/
airlock/airlock-windows-auth-setup-amd64.exe
```

1. Run the installer and select the exported certificate `airlock.cer`. The installer:
    - trusts the Airlock CA;
    - installs the integration DLL;
    - disables NLA for RDP;
    - enables RemoteFX if needed.
2. Reboot the computer.

Automated installation from an elevated cmd/PowerShell:

```
airlock-windows-auth-setup-amd64.exe install --cert=airlock.cer -r
```

Step 2 of 4. Windows Desktop Service

1. Generate a join token:

```
airctl tokens add --type=windowsdesktop
```

1. Save the token to a file on the Linux server, for example `/tmp/token`.
2. Verify that Airlock is installed:

```
airlock version
```

If needed:

```
curl https://deckhouse.ru/downloads/airlock/install.sh | bash -s
```

1. Configure `/etc/airlock.yaml`:

```
version: v3
airlock:
  nodename: windows-agent.example.com
  proxy_server: airlock.example.com:443
  auth_token: /tmp/token
windows_desktop_service:
  enabled: yes
  static_hosts:
  - name: win-desktop-1
    ad: false
    addr: 192.0.2.155
    labels:
      datacenter: dc1
auth_service:
  enabled: no
proxy_service:
  enabled: no
ssh_service:
  enabled: no
```

- `proxy_server` — address of the cluster Proxy Service.
- `static_hosts` / `non_ad_hosts` — IPs or hostnames of Windows machines (autodiscovery is unavailable without AD).

1. Start the service:

```
sudo systemctl enable airlock
sudo systemctl start airlock
sudo systemctl status airlock
```

Step 3 of 4. Role for desktop access

Create `windows-desktop-admins.yaml`:

```
kind: role
version: v6
metadata:
  name: windows-desktop-admins
spec:
  allow:
    windows_desktop_labels:
      "*": "*"
    windows_desktop_logins: ["Administrator", "alice"]
```

```
airctl create -f windows-desktop-admins.yaml
```

Assign the role to the user via `airctl users update`.

Step 4 of 4. Connect

1. Log in to the Airlock web interface with an account that has the `windows-desktop-admins` role.
2. Go to **Resources** → filter by **Desktops** → click **Connect** next to the desired desktop → select the Windows login.
3. The RDP session opens in the browser and is recorded. Use **Disconnect** to end the session.
4. Recordings are available under **Management** → **Session Recordings**.

Next steps

- Narrow access using `windows_desktop_labels` and separate roles for different user groups.
- For multiple desktops, use labels in `static_hosts` and label-based RBAC.

Authentication

This section describes the methods for authenticating users in Airlock: local accounts, single sign-on protocols, and multi-factor authentication.

- Local users — creating accounts, assigning roles, custom policies
- OIDC — configuring OpenID Connect (Keycloak and other IdPs)
- SAML — single sign-on via SAML (ADFS and other IdPs)
- OTP/TOTP — one-time codes as a second factor
- WebAuthn — hardware keys and biometrics as a second factor

Local users

In Airlock, any local, SSO, or bot user can be assigned one or more roles. Roles determine access to databases, SSH servers, Kubernetes clusters, Windows desktops, and web applications.

Local users are accounts managed directly through Airlock rather than through an external identity provider. All local users are stored in the cluster state backend: name, roles, traits, and a bcrypt-hashed password.

Preset roles

Airlock provides several preset roles:

Role	Description
`access`	Access to cluster resources
`editor`	Edit cluster configuration settings
`auditor`	Read cluster events, audit logs, and replay session recordings
`requester`	Create Access Requests
`reviewer`	Review and approve Access Requests

Adding local users

Use the `airctl` utility to manage local users. Add user Alice with the `editor` role:

```
airctl users add alice --roles=editor
```

You can specify multiple roles and permitted OS logins if needed:

```
airctl users add joe --logins=joe,root --roles=access,editor
```

Airlock generates a one-time invitation link (valid for one hour). The user follows the link, sets a password, and configures MFA if required. After registration, the account appears in the list:

```
airctl users ls
```

Example output:

```
User                 Roles
-------------------- --------------
alice                editor
joe                  access, editor
```

Updating roles

Update a user's roles with `airctl users update`:

```
airctl users update alice --set-roles=editor,auditor
```

If a user has multiple roles, permissions are combined (logical OR). A user can simultaneously act as an administrator and an auditor.

Editing and deleting

Retrieve a user definition for editing:

```
airctl get user/joe > joe.yaml
# edit joe.yaml
airctl create -f joe.yaml
```

Delete a local user:

```
airctl users rm joe
```

Creating a custom role

Create a role for interns with access to test and staging SSH servers, monitoring applications, and the dev Kubernetes cluster.

Save the role to `interns.yaml`:

```
kind: role
version: v6
metadata:
  name: interns
spec:
  allow:
    logins: ['readonly']
    kubernetes_groups: ["view"]
    node_labels:
      'env': ['staging', 'test']
    kubernetes_labels:
      'env': 'dev'
    kubernetes_resources:
      - kind: 'pod'
        namespace: "*"
        name: "*"
    app_labels:
      'type': ['monitoring']
  deny:
    node_labels:
      'env': 'prod'
    kubernetes_labels:
      'env': 'prod'
    kubernetes_resources:
      - kind: 'pod'
        namespace: 'prod'
        name: '*'
    db_labels:
      'env': 'prod'
    app_labels:
      'env': 'prod'
```

Create the role:

```
airctl create -f interns.yaml
airctl get roles --format text
```

RBAC principles

An Airlock role contains two rule lists: `allow` and `deny`.

- Access is denied by default.
- `deny` rules are checked first and take priority.

Access to resources is configured through labels on resources and corresponding rules in the role. Multiple labels in a single rule are interpreted as a logical AND.

Lockout after failed login attempts

A local user is locked for 20 minutes if there are several failed login or password reset attempts within a 30-minute window.

An administrator with `user` resource permissions (the `editor` role) can unlock the user:

```
airctl get users/alice > user.yaml
```

Edit the `status.is_locked` field to `false` and apply the changes:

```
airctl create -f user.yaml
```

Next steps

- Configure OIDC or SAML for single sign-on
- Enable OTP or WebAuthn for two-factor authentication
- Review access policies for SSH, Kubernetes, and databases

OIDC authentication

This guide describes how to configure an OpenID Connect (OIDC)-based SSO provider to issue Airlock credentials to user groups. Combined with RBAC, administrators can define policies such as:

- Only members of the “DBA” group can connect to PostgreSQL.
- Developers are prohibited from SSH access to production servers.

Prerequisites

- Administrative access to the SSO/IdP with users organized into groups.
- An Airlock role with permission to manage `oidc` resources (available in the `editor` role by default).
- The `airctl` utility installed and cluster access configured.

Registering the application in the IdP

Register Airlock in your external identity provider and obtain a `client_id` and `client_secret`.

OIDC redirect URL

OIDC uses HTTP redirects to return control to Airlock after authentication. The redirect URL must be agreed upon with the IdP in advance.

URL format:

```
https://<proxy-address>:443/v1/webapi/oidc/callback
```

Replace `<proxy-address>` with the public address of the Airlock Proxy Service (for example, `airlock.example.com`).

Configuring the OIDC connector

Connectors are created, tested, and deleted via `airctl` or the Airlock web interface.

Create a file `client-secret.txt` containing only the client secret. Generate the connector YAML configuration:

```
airctl sso configure oidc --name keycloak \
  --issuer-url https://keycloak.example.com/realms/master \
  --id airlock \
  --secret $(cat client-secret.txt) \
  --claims-to-roles groups,/users,access \
  --claims-to-roles groups,/admins,editor > oidc-connector.yaml
```

Parameter	Description
`--name`	Connector name in Airlock (typically the IdP name)
`--issuer-url`	Base URL of the OIDC provider without `.well-known/openid-configuration`
`--id`	Client ID configured in the IdP
`--secret`	Client secret from the IdP
`--claims-to-roles`	Mapping of a claim/value pair to Airlock roles

Example of a complete connector:

```
kind: oidc
metadata:
  name: keycloak
spec:
  claims_to_roles:
  - claim: groups
    roles:
    - access
    value: /users
  - claim: groups
    roles:
    - editor
    value: /admins
  client_id: airlock
  client_secret: abc123...
  issuer_url: https://keycloak.example.com/realms/master
  redirect_url: https://airlock.example.com:443/v1/webapi/oidc/callback
version: v3
```

Example: Keycloak

In Keycloak on `keycloak.example.com`, create a client named `airlock`. In the `airlock-dedicated` client scope, add a “Group Membership” mapper so that the `groups` claim is included in the token.

Testing and applying

Test the connector before applying it:

```
cat oidc-connector.yaml | airctl sso test
```

The command opens a browser and attempts to log in through the IdP. If an error occurs, examine the output — the “[OIDC] Claims” section shows all user claims from the IdP.

After a successful test, create the connector:

```
airctl create -f oidc-connector.yaml
```

Log in via OIDC:

```
airsh --proxy=airlock.example.com login --auth=keycloak
```

Additional parameters

ACR Values

Airlock can send an Authentication Context Class Reference (ACR) when requesting an authorization code. If the `acr_values` field is set, Airlock expects the same value in the `acr` claim.

```
kind: oidc
version: v2
metadata:
  name: oidc-connector
spec:
  issuer_url: "https://oidc.example.com"
  client_id: "xxxxxxxxxxxxxxxxxxxxxxx.example.com"
  client_secret: "zzzzzzzzzzzzzzzzzzzzzzz"
  redirect_url: "https://airlock.example.com/v1/webapi/oidc/callback"
  display: "Login with Example"
  acr_values: "foo/bar"
  scope: [ "group" ]
  claims_to_roles:
     - claim: "group"
       value: "editor"
       roles: [ "editor" ]
     - claim: "group"
       value: "user"
       roles: [ "access" ]
```

Max age

The `max_age` field sets the maximum age of an OIDC session before forced re-authentication. Not set by default. A value of `0s` forces re-authentication on every login.

```
spec:
  max_age: 24h
```

Prompt

Controls the behavior of the Authorization Server during re-authentication and consent. Airlock uses `select_account` by default.

```
spec:
  # Allowed values: none, select_account, login, consent
  prompt: 'login'
```

Disabling email verification

By default Airlock verifies the `email_verified` claim. For testing you can enable `allow_unverified_email` (reduces security):

```
kind: oidc
version: v2
metadata:
  name: connector
spec:
  allow_unverified_email: true
```

Username claim

By default Airlock uses the email as the username. To use a different claim:

```
spec:
  username_claim: preferred_username
```

Default authentication

To make OIDC the default login method, update `cluster_auth_preference`:

```
airctl get cap > cap.yaml
```

```
kind: cluster_auth_preference
metadata:
  name: cluster-auth-preference
spec:
  type: oidc
version: v2
```

```
airctl create -f cap.yaml
```

Troubleshooting

If you see `Failed to calculate user attributes`, check the `claims_to_roles` mapping — claim values must exactly match what the IdP returns. Use `airctl sso test` to view all claims.

SAML authentication

This guide describes how to configure SAML for single sign-on (SSO) in Airlock. Combined with RBAC, administrators can define policies such as:

- Only members of the “DBA” group can connect to PostgreSQL.

- Developers are prohibited from SSH access to production servers.

Airlock acts as a SAML Service Provider (SP) and accepts authentication from an external Identity Provider (IdP).

Prerequisites

- A deployed SAML IdP (for example, Active Directory Federation Services) with users organized into groups.
- An Airlock role with permission to manage `saml` resources (available in the `editor` role).
- The `airctl` utility installed.

SAML concepts

SAML authentication involves:

1. **IdP (Identity Provider)** — the provider that authenticates the user and issues a SAML assertion.
2. **SP (Service Provider)** — Airlock, which accepts the assertion and creates a session.
3. **ACS URL** — the Airlock endpoint where the IdP sends the SAML response: `https://<proxy>/v1/webapi/saml/acs`.
4. **Entity Descriptor** — IdP XML metadata containing certificates and endpoints.
5. **Attributes (claims)** — groups, email, username — mapped to Airlock roles via `attributes_to_roles`.

Step 1. Configure the IdP (ADFS example)

In Active Directory Federation Services, configure:

Claims Provider Trust — incoming claims from Active Directory:

- `Name ID`: LDAP attribute `E-Mail-Addresses` → `Name ID`
- `Group`: group membership claim for mapping to roles
- If needed: `SAM-Account-Name` → `Windows account name`, `E-Mail-Addresses` → `UPN`

Relying Party Trust — trust for Airlock:

- Display name: `Airlock`
- SAML 2.0 Web SSO URL: `https://airlock.example.com/v1/webapi/saml/acs`
- Relying party trust identifier: the same ACS URL
- Claim Issuance Policy: send at least `Name ID` and `Group`

Ensure that Active Directory users have the email field populated.

Step 2. Create Airlock roles

Create roles for administrators and regular users.

```
# admin-role.yaml
kind: role
version: v3
metadata:
  name: admin
spec:
  options:
    max_session_ttl: "8h0m0s"
  allow:
    logins: [ root ]
    node_labels:
      "*": "*"
    rules:
      - resources: ["*"]
        verbs: ["*"]
```

```
# user-role.yaml
kind: role
version: v3
metadata:
  name: dev
spec:
  options:
    max_session_ttl: "1h"
  allow:
    logins:
      - '{{external["http://schemas.microsoft.com/ws/2008/06/identity/claims/
windowsaccountname"]}}'
      - ubuntu
    node_labels:
      "access": "relaxed"
```

The `dev` role allows login only to nodes labeled `access: relaxed`. The `{{external[...]}}` template substitutes the SAML attribute value as a permitted OS login.

Step 3. Create the SAML connector

Generate the connector configuration:

```
airctl sso configure saml --acs https://airlock.example.com/v1/webapi/saml/acs \
  --preset adfs \
  --entity-descriptor https://adfs.example.com/FederationMetadata/2007-06/
FederationMetadata.xml \
  --attributes-to-roles http://schemas.xmlsoap.org/claims/Group,teleadmins,editor \
  --attributes-to-roles http://schemas.xmlsoap.org/claims/Group,Users,access \
  > adfs.yaml
```

Parameter	Description
`--acs`	Assertion Consumer Service URL (must match the IdP configuration)
`--entity-descriptor`	URL or path to the IdP XML metadata
`--attributes-to-roles`	Mapping of attribute, value, and Airlock role

Test the connector before applying it:

```
cat adfs.yaml | airctl sso test
```

Apply the connector:

```
airctl create -f adfs.yaml
```

Exporting the signing key

Export the Airlock signing certificate and add it to the IdP as a certificate for signature verification:

```
airctl saml export adfs > saml.cer
```

Logging in via SAML

```
airsh --proxy=airlock.example.com login
```

If you have multiple SAML connectors, specify the name:

```
airsh login --auth=adfs
```

Attribute mapping

The `attributes_to_roles` field maps SAML attributes to Airlock roles:

```
kind: saml
version: v2
metadata:
  name: saml-connector
spec:
  acs: https://airlock.example.com/v1/webapi/saml/acs
  entity_descriptor: |
    <?xml version="1.0"?>
    <!-- IdP XML metadata -->
  attributes_to_roles:
    - name: "groups"
      value: "admins"
      roles: ["editor"]
    - name: "groups"
      value: "developers"
      roles: ["access"]
```

The full attribute name depends on the IdP. For ADFS, groups often use the URI `http://schemas.xmlsoap.org/claims/Group`.

Dynamic traits

SAML attribute values are available in roles via `{{external["attribute-name"]}}`. This allows assigning OS logins, environment labels, and other parameters based on IdP data.

Default authentication

To make SAML the default login method:

```
kind: cluster_auth_preference
metadata:
  name: cluster-auth-preference
spec:
  type: saml
version: v2
```

```
airctl get cap > cap.yaml
# edit cap.yaml
airctl create -f cap.yaml
```

Troubleshooting

- Verify that the ACS URL in the IdP and the connector match.
- Confirm that the IdP sends the required attributes (groups, email).
- Use `airctl sso test` to diagnose the SAML flow.
- If you see signature errors, check that the Airlock certificate has been added to the IdP.

Two-factor authentication (OTP/TOTP)

Airlock supports multi-factor authentication (MFA) for local users. The `otp` type implements the TOTP standard — time-based one-time codes.

You can use any TOTP client to generate codes: Authy, FreeOTP, KeePassXC, or similar apps.

Available second_factor modes

Value	Description
`otp`	TOTP only (default)
`webauthn`	WebAuthn only (hardware keys, biometrics)
`on`	TOTP and WebAuthn; each user must have at least one MFA device
`optional`	TOTP and WebAuthn are available but not required
`off`	MFA disabled

> For SSO users, MFA is configured on the IdP side. Airlock does not prompt for MFA during SSO login, although users can still register devices.

Enabling TOTP

Dynamic configuration

Retrieve the current `cluster_auth_preference` resource:

```
airctl get cap > cap.yaml
```

Set `second_factor`:

```
kind: cluster_auth_preference
metadata:
  name: cluster-auth-preference
spec:
  type: local
  second_factor: "otp"
version: v2
```

Apply:

```
airctl create -f cap.yaml
```

For mandatory MFA use `second_factor: "on"`. For gradual rollout use `"optional"`.

Static configuration

Add to `/etc/airlock.yaml`:

```
auth_service:
  authentication:
    type: local
    second_factor: otp
```

Restart the Auth Service after making changes.

Registering a TOTP device

A user registers TOTP via `airsh`:

```
airsh mfa add
# Choose device type [TOTP, WEBAUTHN]: totp
# Enter device name: phone
# Scan QR code with your TOTP app
# Enter an OTP token from the device: 123456
# MFA device "phone" added.
```

List registered devices:

```
airsh mfa ls
```

Remove a device:

```
airsh mfa rm phone
```

Logging in with TOTP

After registering a device, a one-time code is requested on login:

```
airsh login --proxy=airlock.example.com
# Enter password for Airlock user alice:
# Enter an OTP token from a registered device: 654321
```

Lockout after failed attempts

A local user is locked for 20 minutes if there are several failed login or password reset attempts within a 30-minute window.

To unlock, an administrator runs:

```
airctl get users/alice > user.yaml
```

Set `status.is_locked` to `false` and apply:

```
airctl create -f user.yaml
```

Next steps

- Configure WebAuthn for hardware keys
- Review local users and role management

WebAuthn (second factor)

Airlock supports WebAuthn as a second authentication factor. WebAuthn is used when logging in to Airlock (`airsh login` or the web interface), as well as when connecting to SSH nodes and Kubernetes (`airsh ssh`, `kubectl`).

Hardware keys (YubiKey, SoloKey) and biometric authenticators (Touch ID, Windows Hello) are supported.

Prerequisites

- A WebAuthn hardware key or a built-in biometric authenticator.
- A browser with WebAuthn support.
- The `airctl` utility for cluster configuration.

Step 1. Enable WebAuthn

Dynamic configuration

Edit the `cluster_auth_preference` resource:

```
airctl edit cap
```

Example configuration:

```
kind: cluster_auth_preference
version: v2
metadata:
  name: cluster-auth-preference
spec:
  type: local
  second_factor: "on"
  webauthn:
    rp_id: airlock.example.com
    attestation_allowed_cas:
    - "/path/to/allowed_ca.pem"
    attestation_denied_cas:
    - "/path/to/denied_ca.pem"
```

Static configuration

Fragment of `/etc/airlock.yaml`:

```
auth_service:
  authentication:
    type: local
    second_factor: on
    webauthn:
      rp_id: airlock.example.com
      attestation_allowed_cas:
      - "/path/to/allowed_ca.pem"
      attestation_denied_cas:
      - "/path/to/denied_ca.pem"
```

Restart the Auth Service after making changes.

webauthn fields

Field	Description
`rp_id`	Public domain of the Proxy Service (without `https://` and port)
`attestation_allowed_cas`	Allow-list CAs for verifying devices during registration
`attestation_denied_cas`	Deny-list CAs — devices with these CAs are rejected

All devices are allowed by default. Use `attestation_denied_cas` to restrict specific device models.

> Starting from Airlock v10, WebAuthn replaces U2F. Existing U2F devices are supported automatically.

Step 2. Register a device

A user registers a WebAuthn device:

```
airsh mfa add
# Choose device type [TOTP, WEBAUTHN]: webauthn
# Enter device name: desktop yubikey
# Tap any *registered* security key or enter a code from a *registered* OTP device:
# Tap your *new* security key
# MFA device "desktop yubikey" added.
```

If only OTP confirmation is needed during registration (without WebAuthn):

```
airsh mfa add --mfa-mode=otp
```

Step 3. Log in via WebAuthn

After registering a device, confirmation is requested on login:

```
airsh login --proxy=airlock.example.com
# Enter password for Airlock user alice:
# Tap any security key or enter a code from a OTP device:
# > Profile URL:        https://airlock.example.com
#   Logged in as:       alice
#   Cluster:            airlock.example.com
#   Roles:              access, editor
#   Logins:             alice
#   Kubernetes:         enabled
#   Valid until:        2026-07-05 14:00:00 +0300 MSK [valid for 12h0m0s]
```

> WebAuthn on Airlock login is required only for local users. For SSO, configure MFA on the IdP side.

Migrating from U2F

If U2F was previously used, Airlock automatically derives the WebAuthn configuration from U2F. Preserve the `u2f.app_id` field to avoid re-registering devices:

```
auth_service:
  authentication:
    type: local
    second_factor: on
    u2f:
      app_id: https://airlock.example.com
    webauthn:
      rp_id: airlock.example.com
      attestation_allowed_cas:
      - "/path/to/u2f_attestation_ca.pem"
```

Next steps

- Configure OTP/TOTP as an alternative second factor
- Review access policies

Access policies (RBAC)

Airlock uses a role-based access control (RBAC) model. A role contains two rule lists: `allow` and `deny`.

- Access is denied by default.
- `deny` rules are checked first and take priority.

Roles and other dynamic resources are managed through the Airlock web interface, the `airctl` utility, or the API.

Viewing roles

```
airsh login --user=admin --proxy=airlock.example.com
airctl get roles
```

Preset roles

Role	Description
`access`	Access to cluster resources
`editor`	Edit cluster configuration settings
`auditor`	Read events, audit logs, and replay sessions
`requester`	Create Access Requests
`reviewer`	Review and approve Access Requests

Role versions

Versions `v3` – `v7` are supported. Versions `v4` and above are backward compatible with `v3`; the difference lies in the default values for unset fields.

Label	`v3` default	`v4+` default
`node_labels`	`[{"*": "*"}]` when logins are present, otherwise `[]`	`[]`
`app_labels`	`[{"*": "*"}]`	`[]`
`kubernetes_labels`	`[{"*": "*"}]`	`[]`
`db_labels`	`[{"*": "*"}]`	`[]`

Version `v6` added the `kubernetes_resources` field for fine-grained control over Kubernetes resources.

RBAC for infrastructure resources

A role defines which resources (applications, servers, databases) a user can access. The mechanism: labels on resources combined with `allow` / `deny` rules in a role.

Example: infrastructure is divided by labels `environment=production` and `environment=staging`. An intern's role may allow access only to `environment=staging`.

Example role with labels

```
kind: role
version: v5
metadata:
  name: example-role
spec:
  allow:
    node_labels:
      'env': 'stage'
  deny:
    node_labels:
      'workload': ['database', 'backup']
```

Multiple labels in a single rule are interpreted as a logical AND:

```
    db_labels:
      'env': 'prod'
      'region': ['eu-central-1', 'eu-west-1']
```

Extended label syntax

```
kind: role
version: v5
metadata:
  name: example-role
spec:
  allow:
    node_labels:
      'environment': 'test'
      '*': '*'
      'environment': ['test', 'staging']
      'environment': '^test|staging$'
```

Role options

Options constrain session parameters. When a user has multiple roles, combined rules apply:

Option	Description	Behavior with multiple roles
`max_session_ttl`	Maximum SSH certificate TTL	Minimum TTL
`forward_agent`	SSH agent forwarding	Logical OR
`port_forwarding`	TCP port forwarding	Logical OR
`ssh_file_copy`	SCP/SFTP	Logical AND
`client_idle_timeout`	Terminate idle sessions	Minimum value
`require_session_mfa`	MFA per session	Logical OR
`pin_source_ip`	Bind certificate to source IP	Logical OR

Creating a role

```
airctl create -f my-role.yaml
airctl get roles --format text
```

Assign a role to a user:

```
airctl users update alice --set-roles=access,my-role
```

Documentation sections

- SSH hosts — logins, node labels, and SSH session options

- Kubernetes — clusters, groups, users, and Kubernetes resources
- Databases — database labels, accounts, and database names

Access to SSH hosts

Airlock RBAC lets you define granular permissions for authenticating to Linux servers connected to Airlock.

Example policy: server administrators have access to everything, QA and engineers have full access to staging, and engineers can get temporary access to production in emergencies.

Role configuration

The `role` resource provides the following controls for restricting server access:

```
kind: role
version: v5
metadata:
  name: developer
spec:
  allow:
    logins: [ubuntu, debian, '{{internal.logins}}']
    node_labels:
      'env': 'test'
      '*': '*'
      'region': ['eu-central-1', 'eu-west-1']
      'reg': '^eu-central-1|eu-west-1$'
    host_groups: [ubuntu, nginx, other]
    host_sudoers:
    - 'ALL=(ALL) NOPASSWD: ALL'
  deny:
    logins:
    - root
```

⚠ Deny rules take priority. In the example above, any attempt to log in as `root` will be rejected.

Template variables

Role fields support template variables.

`{{external.xyz}}` variables are replaced with values from the SSO provider. For OIDC, `xyz` is a claim; for SAML, it is an assertion.

Example — assigning environments and logins from SAML attributes:

```
spec:
  allow:
    node_labels:
    - env: '{{external.environments}}'
    logins:
    - '{{external.allowedlogins}}'
```

The `{{internal.logins}}` variable applies to local users and trusted clusters. A local user with the trait `logins: jeff` will receive logins `jeff` and `ubuntu`:

```
spec:
  allow:
    logins: ['{{internal.logins}}', ubuntu]
```

SSH role options

The `options` block configures Airlock capabilities for users with the role:

```
spec:
  options:
    create_host_user_mode: drop
    forward_agent: true
    port_forwarding: true
    ssh_file_copy: false
    client_idle_timeout: never
    disconnect_expired_cert: no
    max_sessions: 10
    enhanced_recording:
    - command
    - disk
    - network
    permit_x11_forwarding: true
    max_connections: 2
    record_session:
      default: best_effort
      ssh: best_effort
    require_session_mfa: true
    pin_source_ip: true
    cert_extensions:
     - type: ssh
       mode: extension
       name: login@example.com
       value: "{{ external.login }}"
```

Option	Description
`create_host_user_mode`	Auto-create a user on the host (`drop` , `keep` , `off`)
`forward_agent`	SSH agent forwarding
`port_forwarding`	TCP port forwarding
`ssh_file_copy`	SCP/SFTP (default `true`)
`require_session_mfa`	Additional MFA when starting a session
`pin_source_ip`	Bind the certificate to the login source IP

Example: environment separation

```
kind: role
version: v5
metadata:
  name: staging-access
spec:
  allow:
    logins: [ubuntu, deploy]
    node_labels:
      'env': 'staging'
  deny:
    node_labels:
      'env': 'prod'
```

Create and assign:

```
airctl create -f staging-access.yaml
airctl users update developer --set-roles=access,staging-access
```

Verifying access

A user can see available servers:

```
airsh ls
```

Connect to a server:

```
airsh ssh user@hostname
```

Access to Kubernetes

The Airlock Kubernetes Service acts as a proxy between Kubernetes users and one or more clusters.

Upon authentication, a user receives a kubeconfig for sending requests to authorized clusters through the Kubernetes Service. The service checks, modifies, or rejects requests based on the user's Airlock roles.

How it works

1. The user authenticates with Airlock.
2. The Kubernetes Service determines permissions from the user's roles.
3. For allowed requests, the service rewrites headers to impersonate the appropriate Kubernetes user and groups.
4. The request is forwarded to the Kubernetes API server.

Role fields for Kubernetes

```
kind: role
version: v6
metadata:
  name: kube-access
spec:
  allow:
    kubernetes_labels:
      'region': '*'
      'platform': 'production'
    kubernetes_resources:
      - kind: pod
        namespace: "production"
        name: "^webapp-[a-z0-9-]+$"
      - kind: pod
        namespace: "development"
        name: "*"
    kubernetes_groups:
    - developers
    kubernetes_users:
    - admin
  deny: {}
```

kubernetes_labels

Defines which registered Kubernetes clusters the user can connect to. Labels are set when registering the agent:

```
--set labels.region=local --set labels.platform=production
```

kubernetes_resources

Available in `v6` and above. Restricts access to specific resources by kind, namespace, and name. Names support regular expressions (using `^` and `$`).

Role version	Default `kubernetes_resources`
`v3`, `v4`, `v5`	All pods in all namespaces
`v6`	Empty list (no access)
`v7`	All resources in all namespaces

kubernetes_groups and kubernetes_users

Define which Kubernetes identity the user acts as. Groups must be bound via Kubernetes RBAC (ClusterRoleBinding) in the target cluster.

Example Kubernetes RBAC in a cluster:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: pod-viewer
rules:
- apiGroups: [""]
  resources: ["pods"]
  verbs: ["get", "watch", "list"]
---
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: pod-viewer
subjects:
- kind: Group
  name: developers
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: ClusterRole
  name: pod-viewer
  apiGroup: rbac.authorization.k8s.io
```

Example: namespace restriction

```
kind: role
version: v6
metadata:
  name: dev-kube
spec:
  allow:
    kubernetes_labels:
      'env': 'dev'
    kubernetes_resources:
      - kind: pod
        namespace: "development"
        name: "*"
      - kind: deployment
        namespace: "development"
        name: "*"
    kubernetes_groups:
    - developers
    kubernetes_users:
    - dev-user
  deny:
    kubernetes_resources:
      - kind: '*'
        namespace: 'production'
        name: '*'
```

Create the role:

```
airctl create -f dev-kube.yaml
airctl users update developer --set-roles=access,dev-kube
```

Connecting to a cluster

List available clusters:

```
airsh kube ls
```

Authenticate and update kubeconfig:

```
airsh kube login my-cluster
d8 k get pods --all-namespaces
```

The Kubernetes Service filters the output — a user sees only resources permitted by their role.

Check permissions:

```
d8 k auth can-i create pods
```

Combining Airlock RBAC and Kubernetes RBAC

Airlock RBAC determines which clusters and resources a user can access. Kubernetes RBAC in the target cluster determines what the user can do with those resources (verbs: get, list, create, etc.).

Recommended approach:

1. Create a Kubernetes ClusterRole with the required permissions.
2. Bind it to the group specified in `kubernetes_groups` of the Airlock role.
3. Restrict Airlock access via `kubernetes_labels` and `kubernetes_resources`.

Access to databases

RBAC lets administrators define granular access policies for databases connected to Airlock.

Example policy: database administrators have access to everything, QA and engineers have full access to staging, and engineers can get temporary access to production in emergencies.

Role configuration

The `role` resource provides the following controls for restricting database access:

```
kind: role
version: v5
metadata:
  name: developer
spec:
  allow:
    db_labels:
      environment: ["dev", "stage"]
    db_users: ["viewer", "editor"]
    db_names: ["main", "metrics", "postgres"]
```

Wildcards

You can use `*` to match any usernames or database names:

```
kind: role
version: v5
metadata:
  name: prod-reader
spec:
  allow:
    db_labels:
      environment: ["prod"]
    db_users: ["*"]
    db_names: ["*"]
  deny:
    db_users: ["postgres"]
    db_names: ["postgres"]
```

⚠ Deny rules take priority. Any attempt to connect with the `postgres` account or to the `postgres` database will be rejected.

Database names

The behavior of the `db_names` field depends on the database engine.

PostgreSQL supports multiple logical databases. On connection, `db_names` is checked against the target database name.

MySQL — the terms “database” and “schema” are synonymous; permissions are determined by grants on the account. The `db_names` field is not checked when connecting to MySQL.

Template variables

`db_*` fields support template variables.

`{{external.xyz}}` is replaced with values from SSO: for OIDC, `xyz` is a claim; for SAML, it is an assertion.

Example:

```
spec:
  allow:
    db_names: ["{{external.databases}}"]
```

The `{{internal.db_users}}` and `{{internal.db_names}}` variables are used in trusted clusters to propagate permitted accounts and database names from the root cluster to the leaf cluster.

Root cluster:

```
spec:
  allow:
    db_users: ["postgres"]
    db_names: ["postgres"]
```

Leaf cluster:

```
spec:
  allow:
    db_users: ["{{internal.db_users}}"]
    db_names: ["{{internal.db_names}}"]
```

create_db_user_mode option

Automatically create a database user at session start:

Value	Description
`off`	Auto-creation disabled
`keep`	User is disabled at the end of the session
`best_effort_drop`	Attempt to drop the user; if it fails, disable instead

With multiple roles, logical OR applies — if at least one role permits it, auto-creation is enabled.

```
spec:
  options:
    create_db_user_mode: keep
```

Example: environment separation

```
kind: role
version: v5
metadata:
  name: db-staging
spec:
  allow:
    db_labels:
      environment: ["staging"]
    db_users: ["app_user", "readonly"]
    db_names: ["app", "analytics"]
  deny:
    db_labels:
      environment: ["prod"]
```

Create and assign:

```
airctl create -f db-staging.yaml
airctl users update developer --set-roles=access,db-staging
```

Connecting to a database

List available databases:

```
airsh db ls
```

Connect:

```
airsh db connect --db-user=readonly --db-name=app postgres-instance
```

User guide

Guide for end users of the Airlock platform.

- airsh utility — log in to the cluster, SSH, Kubernetes, databases
- Access Requests — creating, approving, and using temporary access
- Web interface — active sessions, idle timeout

airsh utility

This guide describes how to use the Airlock client utility — `airsh`.

You will learn how to:

- Log in to the cluster and obtain a user certificate.
- Connect to servers via SSH.
- Work with Kubernetes clusters using `kubectl`.
- Connect to databases.
- Copy files, list nodes, and browse session recordings.

The full list of commands is also available via `airsh help` in the terminal.

Quick start

```
# Log in to the Airlock cluster. The certificate is saved to ~/.airsh/
airlock.example.com
airsh login --proxy=airlock.example.com

# SSH connection to a node
airsh ssh user@node

# airsh ssh accepts the same arguments as OpenSSH:
airsh ssh -o ForwardAgent=yes user@node

# You can create a symlink:
ln -s /path/to/airsh /path/to/ssh
ssh user@host

# Remove certificates from the local machine:
airsh logout
```

Airlock is fully compatible with existing SSH workflows: simply run `airsh login` before starting work.

Installing airsh

Install the `airsh` binary of the same major version as the Airlock server in your cluster.

To find out the server version:

- In the web interface, select your username in the top-right corner → **Help & Support**. The cluster version is shown under **CLUSTER INFORMATION**.
- Via the API:

```
curl https://airlock.example.com/webapi/find | jq '.server_version'
```

User identification

A user's identity in Airlock exists within the cluster. Nodes in the cluster may have different OS users. The Airlock administrator assigns permitted logins to each account.

When connecting to a remote node, specify the Airlock login (flag `--user`) and the OS login in the format `login@host`:

```
# Authenticate to cluster "work" as joe, log in to the node as root:
airsh ssh --proxy=work.example.com --user=joe root@node
```

Logging in to the cluster

To obtain a certificate, run:

```
# Full form:
airsh login --proxy=proxy_host:<https_proxy_port>

# Default ports:
airsh login --proxy=work.example.com

# Non-standard HTTPS port:
airsh login --proxy=work.example.com:5000
```

Port	Description
https_proxy_port	HTTPS port of the proxy (default `443` or `3080`)

The `login` command saves the certificate to `~/.airsh` and to ssh-agent if it is running. By default, certificates are valid for 12 hours.

> Run `airsh login` before other commands — then the `--proxy` flag can be omitted: `airsh ssh user@host` will work.

A cluster may support multiple identity sources. Use the `--auth` flag to select a connector:

```
# Local user admin:
airsh --proxy=proxy.example.com --auth=local --user=admin login

# SSO via GitHub (connector "github"):
airsh --proxy=proxy.example.com --auth=github login
```

With external authentication, `airsh` opens a browser. To suppress this:

```
airsh login --proxy=work.example.com --browser=none
```

The login URL will be printed to the terminal — copy it into a browser.

Viewing the certificate

```
airsh status

# > Profile URL:  https://proxy.example.com:3080
#   Logged in as: johndoe
#   Cluster:      proxy.example.com
#   Roles:        access, auditor, editor
#   Logins:       root, admin, guest
#   Kubernetes:   enabled
#   Valid until:  2017-04-25 15:02:30 -0700 PDT [valid for 1h0m0s]
#   Extensions:   permit-agent-forwarding, permit-port-forwarding, permit-pty
```

SSH-agent integration

If ssh-agent is running, `airsh login` saves the certificate to the agent. Verify:

```
ssh-add -L
```

To disable agent integration, pass `--no-use-local-ssh-agent` or set `AIRLOCK_USE_LOCAL_SSH_AGENT=false`.

Identity files

```
# Save certificate to file joe.pem:
airsh login --proxy=proxy.example.com --out=joe

# Use the file for login:
airsh ssh --proxy=proxy.example.com -i joe joe@db
```

For OpenSSH compatibility, add `--format=openssh` — files `joe` and `joe-cert.pub` will be created.

SSH access

Listing nodes

```
airsh ls

# Node Name     Address            Labels
# ---------     -------            ------
# turing        ← Tunnel          os=linux
# graviton      10.1.0.7:3022      os=osx

# Filter by label:
airsh ls os=osx
```

Interactive shell

```
airsh ssh user@node
airsh ssh -p 6122 user@node ls
airsh ssh -o ForwardAgent=yes user@node
```

`airsh ssh` supports OpenSSH flags: `-p`, `-l`, `-L`, `-A`, and others.

Port forwarding

```
airsh ssh -L 5000:web.remote:80 node
```

The `--local` flag runs a local command through the tunnel:

```
airsh ssh -L 5000:example.com:80 --local node curl http://localhost:5000
```

Jump host

```
airsh ssh -J proxy.example.com telenode
```

Only one jump host is supported; with `-J user@proxy`, port forwarding is used instead of the Airlock proxy.

Node name resolution

- By IP address or DNS.
- By node name (`nodename` in the agent configuration).
- By labels: `airsh ssh os=osx` (if there is only one matching node).

Copying files

```
airsh scp example.txt root@node:/path/to/dest
scp -P 61122 -r files root@node:/path/to/dest
```

Airlock supports SCP and SFTP protocols.

Session sharing

On the remote server, run:

```
airlock status
# Session ID : 7645d523-60cb-436d-b732-99c5df14b7c4
# Session URL: https://work:3080/web/sessions/7645d523-...
```

Another user can join:

```
airsh join <session_ID>
```

> Joining sessions requires special permissions configured by the administrator.

Kubernetes access

List available clusters:

```
airsh kube ls

# Kube Cluster Name Labels                      Selected
# ----------------- --------------------------- --------
# mycluster         env=dev
```

Log in to a cluster:

```
airsh kube login mycluster
# Logged into kubernetes cluster "mycluster". Try 'kubectl version' to test the
connection.
```

After `airsh kube login`, the kubeconfig is updated. Then use `kubectl`:

```
kubectl get pods
```

`airsh kube login` flags:

Flag	Description
`--all`	Generate kubeconfig for all available clusters
`--as`	Kubernetes user for impersonation
`--as-groups`	Kubernetes group for impersonation
`--cluster`	Airlock cluster name (for leaf clusters)
`-n, --kube-namespace`	Default namespace

Database access

List available databases:

```
airsh db ls

# Name                    Description  Allowed Users   Labels  Connect
# mysql-server1           ...          [alice]         env=dev airsh db connect mysql-
server1
```

Direct connection via airsh

For MySQL/MariaDB, the `mysql` client is required:

```
airsh db connect --db-user=alice --db-name=airlock_example mysql-server1
```

Local tunnel

For GUI clients or other database types, create a local tunnel:

```
airsh proxy db mysql-db --db-user=alice --tunnel
# Started authenticated tunnel for the MySQL database "mysql-db" on 127.0.0.1:49415.

# Connect:
mysql --port 49415 --host localhost --protocol TCP
```

With TLS routing:

```
airsh proxy db --port 10700 mysql-db
# Started DB proxy on 127.0.0.1:10700
```

Session recordings

```
airsh recordings ls

airsh play <session-id>
```

airsh configuration

Configuration files:

- `/etc/airsh.yaml` — global configuration (overridden by `AIRLOCK_GLOBAL_AIRSH_CONFIG`).
- `~/.airsh/config/config.yaml` — user configuration.

User configuration takes precedence over global.

Aliases

```
aliases:
    "l": "airsh login --auth=local"
    "status": "airsh status --format=json"
```

Additional HTTP headers

```
add_headers:
  - proxy: "*.example.com"
    headers:
      foo: bar
```

Trusted clusters

```
airsh --proxy=work clusters

# Cluster Name     Status
# staging          online
# production       offline

airsh --proxy=work ls --cluster=production
airsh --proxy=work ssh --cluster=production db-1
```

Access Requests

Access Requests allow users to obtain **temporary** privilege escalation — to additional roles or specific resources. A request goes through an approval workflow: one or more reviewers approve or deny it.

In Airlock, Access Requests are available through the web interface, the CLI (`airsh`), and the `requester` / `reviewer` roles. This includes role requests, resource requests (SSH, databases, Kubernetes, applications), and approval without requiring an administrator on the Auth Service host.

This approach enforces the principle of least privilege: the number of permanent wide-access administrators decreases, and elevated access is granted only for a limited time and a specific task.

Airlock supports two scenarios:

Type	When to use
Role Access Request	Additional roles are needed for system tasks — for example, the `dba` role for a database migration
Resource Access Request	Access to a specific server, database, application, or Kubernetes object is needed without knowledge of the internal RBAC model

A single request can contain **either** roles **or** resources — mixing both types is not allowed.

requester and reviewer roles

To work with Access Requests, the administrator assigns users the built-in roles:

Role	Purpose
`requester`	Create Access Requests
`reviewer`	Review and approve or deny Access Requests

If your account lacks one of these roles, contact the cluster administrator.

> ⚠ Granting a role that can edit other roles may allow a user to **permanently** escalate their privileges. Always carefully review the requested roles and resulting permissions before approving.

Creating a request via the web interface

Role request

1. In the left panel, select **Resources** → **Access Requests** → **New Request**.
2. In the dropdown, select **Roles**.
3. Check the required roles and click **ADD TO REQUEST**.
4. Click **PROCEED TO REQUEST**, enter a reason, and submit.

Resource request

1. Open **Resources** → **Access Requests** → **New Request**.
2. Select **Resources** instead of roles.
3. Find the required server, database, application, or Kubernetes object.
4. Add resources to the request, enter a reason, and submit.

Track the request status on the **Access Requests** page.

Creating a request via airsh

Role request

```
# Log in to the cluster
airsh login --proxy=airlock.example.com --user=alice

# Create a request for the dba role
airsh request create \
    --roles=dba \
    --reason="database migration tonight"
```

By default, the command waits for approval. To submit without waiting, add the `--nowait` flag.

You can request a role directly at login:

```
airsh login --user=alice --request-roles=dba
# Seeking request approval... (id: bc8ca931-fec9-4b15-9a6f-20c13c5641a9)
```

After approval, a certificate is issued with the requested role. To log in without waiting, use `--request-nowait` — regular roles are issued immediately, and elevated access is activated after approval.

Resource request

First, find the resource:

```
airsh login --proxy=airlock.example.com --user=alice

# Search for SSH nodes (an empty airsh ls list is normal without direct access)
airsh request search --kind node

# Search with a filter
airsh request search --kind node --search iot
```

Supported resource types: `node`, `kube_cluster`, `db`, `app`, `windows_desktop`, and Kubernetes objects (`pod`, `namespace`, `deployment`, and others).

Create a request using the resource ID from `airsh request search` output:

```
airsh request create \
    --resource /airlock.example.com/node/b1168402-9340-421a-a344-af66a6675738 \
    --reason="incident response 123"
```

The command prints the request ID and waits for approval:

```
Creating request...
Request ID: f406f5d8-3c2a-428f-8547-a1d091a4ddab
Status:     PENDING

Waiting for request approval...
```

Automatic request on access denial

If resource requests are configured, `airsh ssh` can create a request automatically when access is denied:

```
airsh ssh alice@iot
# ERROR: access denied to alice connecting to iot on cluster airlock.example.com
#
# You do not currently have access to alice@iot, attempting to request access.
#
# Enter request reason: need access for diagnostics
# Creating request...
# Waiting for request approval...
```

After approval, the connection continues without any additional steps.

Reviewing requests

Via the web interface

Users with the `reviewer` role open **Management** → **Access Requests** → **Review Requests** and see the list of pending requests. Each request can be approved or denied.

Via airsh

```
airsh login --proxy=airlock.example.com --user=bob

# List requests available for review
airsh request ls --reviewable

# Request details
airsh request show f406f5d8-3c2a-428f-8547-a1d091a4ddab

# Approve
airsh request review --approve f406f5d8-3c2a-428f-8547-a1d091a4ddab

# Deny
airsh request review --deny f406f5d8-3c2a-428f-8547-a1d091a4ddab
```

A cluster administrator can also review requests via `airctl` on the Auth Service host if needed:

```
airctl requests ls
airctl request approve f406f5d8-3c2a-428f-8547-a1d091a4ddab
airctl request deny --reason="task cancelled" f406f5d8-3c2a-428f-8547-a1d091a4ddab
```

Using an approved request

Via airsh

After approval, log in with the request ID:

```
airsh login --request-id=f406f5d8-3c2a-428f-8547-a1d091a4ddab
```

Check the active profile:

```
airsh status
#   Active requests:    f406f5d8-3c2a-428f-8547-a1d091a4ddab
#   Roles:              access, requester
#   Allowed Resources:  ["/airlock.example.com/node/b1168402-..."]
```

With a **role** request, permissions are **added** to existing ones — the user retains their regular roles plus the approved ones.

With a **resource** request, the certificate is restricted to the approved resources only — access to other resources is temporarily blocked.

Example of accessing an approved SSH node:

```
airsh ls
airsh ssh alice@iot
```

Via the web interface

Open the approved request on the **Review Requests** page and click **ASSUME ROLES** (for role requests), or connect to the resource from the **Resources** section.

While elevated access is active, a banner is displayed at the top of the page. To return to regular permissions, click **Switch Back**.

Dropping elevated access

When temporary access is no longer needed, drop the active request:

```
airsh request drop
```

You can specify a particular ID:

```
airsh request drop f406f5d8-3c2a-428f-8547-a1d091a4ddab
```

In the web interface, use the **Switch Back** button in the active request banner.

Viewing your requests

```
# All requests for the current user
airsh request ls --my-requests

# Details of a specific request
airsh request show <request-id>
```

Kubernetes: searching and requesting objects

Search for pods in a cluster:

```
airsh request search --kind=pod --kube-cluster=mycluster --kube-namespace=default
```

Request access to a pod:

```
airsh request create \
    --resource /airlock.example.com/pod/mycluster/development/nginx-1 \
    --reason="application debugging"
```

To access all pods in a namespace you can use a wildcard or a regular expression:

```
airsh request create \
    --resource /airlock.example.com/pod/mycluster/*/^nginx-[a-z0-9-]+$
```

Web interface

The Airlock web interface is a browser-based interface for accessing resources, viewing active sessions and recordings, managing access requests, users, and roles.

Joining an active session

The web interface lets you view and join active SSH sessions through a terminal in the browser.

Active SSH sessions you have access to are shown on the **Active Sessions** page in the left navigation panel. If the **Active Sessions** tab is missing, your role lacks the `list` permission for the `ssh_session` resource. Contact the administrator to update the role.

When you click **Join**, you must select a participant mode:

- **observer** — view-only, no input.
- **moderator** — view and forcibly terminate or pause the session, no input.
- **peer** — collaborative: view and input.

If the join button is absent, you have no permissions in any mode.

Idle timeout

After login, the web interface checks session activity every 30 seconds. If there has been no interaction with any browser tab (keyboard, mouse) for more than 10 minutes, the user is logged out.

To change the default timeout (10 minutes), the administrator sets the `web_idle_timeout` parameter in the Auth Service configuration.

Dynamic configuration

```
airctl edit cluster_networking_config
```

Change the `spec.web_idle_timeout` value:

```
kind: cluster_networking_config
metadata:
  ...
spec:
  ...
  web_idle_timeout: 10m0s
  ...
version: v2
```

After saving, `airctl` updates the resource:

```
cluster networking configuration has been updated
```

Static configuration

Update `/etc/airlock.yaml` in the `auth_service` section and restart the `airlock` daemon:

```
auth_service:
  web_idle_timeout: 10m0s
```

Web interface sections

Section	Purpose
Resources	List of available servers, databases, and Kubernetes clusters
Active Sessions	Active SSH sessions and the ability to join them
Session Recordings	Recorded SSH and Kubernetes sessions
Access Requests	Create and review access requests (see Access Requests)
Management	Manage users, roles, and settings (for administrators)

Kubernetes via the web interface

The **Kubernetes** tab shows clusters you have access to. Click **Connect** — a window appears with commands for connecting via `airsh kube login` in the terminal.

Databases via the web interface

The web interface does not establish a direct connection to databases, but shows available databases on the **Databases** tab and provides `airsh` commands for connecting from the local terminal.

Reference

Reference information for command-line utilities.

- airsh reference — user client commands
- airctl reference — administrator commands

airsh reference

`airsh` is the CLI client for Airlock users: log in to a cluster, SSH, Kubernetes, databases, and file copying.

Full reference: `airsh help <command>`.

Environment variables

Variable	Description	Example
`AIRLOCK_AUTH`	Authentication connector name (SAML, OIDC, GitHub, local)	github
`AIRLOCK_CLUSTER`	Root or leaf cluster name	cluster.example.com
`AIRLOCK_LOGIN`	Default OS login on the remote host	root
`AIRLOCK_LOGIN_BROWSER`	`none` — do not open the browser during SSO	none
`AIRLOCK_PROXY`	Proxy Service address	cluster.example.com:3080
`AIRLOCK_HOME`	airsh configuration directory	~/.airsh
`AIRLOCK_USER`	Airlock username	alice
`AIRLOCK_USE_LOCAL_SSH_AGENT`	ssh-agent integration	true, false
`AIRLOCK_GLOBAL_AIRSH_CONFIG`	Path to global config	/etc/airsh.yaml
`AIRLOCK_IDENTITY_FILE`	Path to identity file	/opt/identity

Global flags

Flag	Description
`-l, --login`	OS login on the remote host
`--proxy`	Proxy Service address: `host:https_port[,ssh_proxy_port]`
`--user`	Airlock username
`--ttl`	Certificate validity in minutes (default 720)
`-i, --identity`	Identity file
`--auth`	Authentication connector (default `local`)
`--insecure`	Do not verify the server certificate (tests only)
`-d, --debug`	Verbose output
`-J, --jumphost`	SSH jump host

airsh login

Log in to the cluster. The certificate is saved to `~/.airsh`.

```
airsh login [<flags>] [<cluster>]
```

Flag	Description
`-o, --out`	Save identity to a file
`--format`	`file`, `openssh`, `kubernetes`
`--browser`	`none` — do not open the browser
`--no-use-local-ssh-agent`	Do not load the certificate into ssh-agent

```
airsh --proxy=proxy.example.com login
airsh --proxy=proxy.example.com --auth=local --user=admin login
airsh --proxy=proxy.example.com --auth=github login
airsh --ttl=1 login
airsh login --format=kubernetes -o kubeconfig
```

airsh logout

Removes the client certificate.

```
airsh logout
```

airsh status

Shows the current profile, roles, logins, and certificate expiry.

```
airsh status
```

airsh ls

List cluster nodes.

```
airsh ls [<flags>] [<label>]
```

Flag	Description
`-v, --verbose`	Show Node ID
`--cluster`	Leaf cluster

```
airsh ls
airsh ls os=linux
airsh ls -v
```

airsh ssh

SSH connection or command execution.

```
airsh ssh [<flags>] <[user@]host> [<command>...]
```

Flag	Description
`-p, --port`	SSH port
`-A, --forward-agent`	Agent forwarding
`-L, --forward`	Local port forwarding
`-D, --dynamic-forward`	SOCKS5 proxy
`--cluster`	Leaf cluster
`-X, --x11-untrusted`	X11 forwarding
`-Y, --x11-trusted`	Trusted X11 forwarding

```
airsh ssh root@grav-00
airsh ssh -o ForwardAgent=yes root@grav-00
airsh ssh root@env=prod hostname
```

airsh scp

Copy files.

```
airsh scp [<flags>] <source>... <dest>
```

Flag	Description
`-r, --recursive`	Recursive copy
`-P, --port`	Port
`--cluster`	Leaf cluster

airsh join

Join an active session.

```
airsh join [<flags>] <session-id>
```

airsh clusters

List trusted clusters.

```
airsh clusters [--quiet]
```

airsh kube ls

List available Kubernetes clusters.

```
airsh kube ls
```

airsh kube login

Log in to a Kubernetes cluster, update kubeconfig.

```
airsh kube login <kube-cluster>
```

Flag	Description
`--all`	Kubeconfig for all available clusters
`--as`	Kubernetes user impersonation
`--as-groups`	Kubernetes group impersonation
`--cluster`	Airlock cluster
`-n, --kube-namespace`	Default namespace

airsh db ls

List available databases.

```
airsh db ls
```

airsh db connect

Connect to a database using the built-in client.

```
airsh db connect --db-user=<user> --db-name=<db> <database>
```

airsh proxy db

Local proxy for connecting to a database.

```
airsh proxy db [<flags>] <db>
```

Flag	Description
`--db-user`	Database user
`--db-name`	Database name
`--port`	Local port
`--tunnel`	Authenticated tunnel without additional credentials
`--cluster`	Leaf cluster

```
airsh proxy db mysql-db
airsh proxy db --port 10700 mysql-db
airsh proxy db --tunnel mysql-db
```

airsh recordings ls

List recorded sessions.

```
airsh recordings ls [--last=6h0m0s] [--from-utc=YYYY-MM-DD] [--to-utc=YYYY-MM-DD]
```

airsh play

Play back a recorded session.

```
airsh play <session-id>
```

airsh config

Print the OpenSSH configuration for use with Airlock.

```
airsh config >> ~/.ssh/config
```

airsh proxy ssh

Local TLS proxy for SSH (TLS routing mode).

```
airsh proxy ssh [<flags>] <[user@]host>
```

airsh version

Client and server version.

```
airsh version [--format=json] [--client]
```

airsh mfa add / ls / rm

Manage MFA devices.

```
airsh mfa add
airsh mfa ls
airsh mfa rm <type>
```

airsh request create / ls / review / show / drop / search

Manage Access Requests.

```
airsh request create --roles=prod --reason="..."
airsh request ls [--my-requests] [--reviewable]
airsh request review --approve <request-id>
airsh request show <request-id>
airsh request drop [<request-id>...]
airsh request search --kind=node --labels=env=dev
```

airctl reference

`airctl` is the administrator CLI tool for managing cluster resources: users, nodes, tokens, certificates, and configuration.

airctl authentication

Before running commands, the administrator must authenticate to the cluster.

With an identity file on a remote host

```
airctl --identity=identity.pem --auth-server=proxy.example.com:443 status
```

The identity file is created via `airctl auth sign` or `airsh login --out=<path>`.

On the same host as Auth Service

If `/etc/airlock.yaml` exists on the host, `airctl` authenticates to the Auth Service from the configuration.

> When run locally on the Auth Service host, the audit log records the action as performed by the Auth Service itself. It is recommended to restrict direct SSH access to the Auth Service.

After airsh login on a remote host

If there is no local config file or identity file, `airctl` uses the `airsh` profile created by `airsh login`. Ensure `AIRLOCK_CONFIG_FILE` is empty or `/etc/airlock.yaml` is absent.

Global flags

Flag	Description
`-d, --debug`	Verbose output
`-c, --config`	Config path (default `/etc/airlock.yaml`)
`--auth-server`	Auth/Proxy Service address
`-i, --identity`	Identity file
`--insecure`	Do not verify TLS certificate (dangerous)

airctl status

Cluster status.

```
airctl status
airctl status --auth-server=192.168.99.102:3025 --identity=identity.pem
```

airctl users add

Create a user and an invitation token.

```
airctl users add [<flags>] <account>
```

Flag	Description
`--roles`	Roles (comma-separated)
`--logins`	Permitted SSH logins
`--kubernetes-groups`	Kubernetes groups
`--kubernetes-users`	Kubernetes users
`--db-users`	Database users
`--db-names`	Database names
`--ttl`	Token validity (max 48h)

```
airctl users add joe --roles=access,editor --logins=joe,ubuntu
```

airctl users ls

List users.

```
airctl users ls
```

airctl users rm

Delete users.

```
airctl users rm <logins>
airctl users rm sally,tim
```

airctl users reset

Reset a local user’s password.

```
airctl users reset <account> [--ttl=8h]
```

airctl users update

Update an account.

```
airctl users update [<flags>] <account>
```

airctl nodes add

Generate a token for joining a node.

```
airctl nodes add [<flags>]
```

Flag	Description
`--roles`	`proxy`, `auth`, `node`, `db`, `app`, `windowsdesktop`
`--ttl`	Token validity (default 30m)
`--token`	Custom token value

```
airctl nodes add
airctl nodes add --roles=node,proxy --ttl=1h
```

airctl nodes ls

List active SSH nodes.

```
airctl nodes ls [--namespace=<ns>]
```

airctl tokens add

Create an invitation token.

```
airctl tokens add --type=TYPE [<flags>]
```

Flag	Description
`--type`	`proxy`, `auth`, `trusted_cluster`, `node`, `db`, `kube`, `app`, `windowsdesktop`
`--ttl`	Validity (default 1h)
`--labels`	Token labels
`--value`	Token value

```
airctl tokens add --type=node
airctl tokens add --type=trusted_cluster --ttl=5m
airctl tokens add --type=kube
airctl tokens add --type=node,db
```

airctl tokens ls

List tokens.

```
airctl tokens ls [--format=json]
```

airctl tokens rm

Remove (revoke) a token.

```
airctl tokens rm <token>
```

airctl get

Get resources in YAML/JSON.

```
airctl get [<flags>] <resource-type/resource-name>,...
```

Flag	Description
`--format`	`yaml`, `json`, `text`
`--with-secrets`	Include secrets

```
airctl get users
airctl get user/joe > joe.yaml
airctl get cluster/east
airctl get clusters,users
airctl get all > state.yaml
```

airctl create

Create a resource from a file.

```
airctl create -f <file.yaml>
```

airctl edit

Edit a resource in a text editor.

```
airctl edit <resource-type/resource-name>
```

Editor priority: `AIRLOCK_EDITOR` → `VISUAL` → `EDITOR` → `vi`.

```
airctl edit role/sre
AIRLOCK_EDITOR=nano airctl edit user/alice
```

airctl rm

Delete a resource.

```
airctl rm <resource-type/resource-name>
```

```
airctl rm users/admin
airctl rm role/old-role
```

airctl auth export

Export the cluster’s public CA certificates.

```
airctl auth export [--type=tls-host] [--fingerprint=SHA256:...]
```

Flag	Description
`--type`	`user`, `host`, `tls-host`, `tls-user`, `db`, `openssh`, `saml-idp`
`--fingerprint`	Filter by fingerprint
`--keys`	Export private keys

airctl auth sign

Create an identity file for a user or host.

```
airctl auth sign -o <filepath> [--user <user> | --host <host>] [<flags>]
```

Flag	Description
`--user`	Airlock username
`--host`	Airlock host name
`-o, --out`	File path
`--format`	`file`, `openssh`, `tls`, `kubernetes`
`--ttl`	Validity (default 12h)

```
airctl auth sign --ttl=24h --user=jenkins --out=jenkins.pem
airctl auth sign --format=openssh --host grav-00
airctl auth sign --user=admin --out=identity.pem
```

airctl auth rotate

Rotate certificate authorities.

```
airctl auth rotate [--type=user|host] [--grace-period=200h]
```

airctl request create / approve / deny / ls / rm

Manage Access Requests.

```
airctl request create myuser --roles=prod --reason="Fix an outage"
airctl request approve request-id-1,request-id-2
airctl request deny request-id-1
airctl request ls
airctl request rm request-id-1
```

airctl alerts create / list / ack

Manage cluster alerts.

```
airctl alerts create --severity=LOW "System under maintenance"
airctl alerts list
airctl alerts ack --reason="scheduled" upgrade-suggestion
airctl alerts ack --clear upgrade-suggestion
```

airctl top

Diagnostic information (requires `airlock start --diag-addr=<bind-addr>`).

```
airctl top http://127.0.0.1:3000 5s
```

airctl help

```
airctl help
airctl help users add
```