Airbyte Self-Managed Enterprise

Complete Documentation

Contents

Airbyte Self-Managed Enterprise

Airbyte Self-Managed Enterprise is the best way to run Airbyte yourself. You get all 600+ pre-built connectors, data never leaves your environment, and Self-Managed Enterprise introduces new governance capabilities targeted towards large organizations designed to enhance your data platform's capabilities and security.

Feature	Description
User Management	Enable multiple users to concurrently move data from a single Airbyte deployment.
Single Sign-On	Manage user access to Airbyte from your Okta, Azure Entra ID or OIDC-compatible identity provider.
Multiple Workspaces	Manage multiple isolated projects or teams on a single Airbyte deployment.
Role-Based Access	Manage user permissions and access across workspaces from a single pane of glass.
Column Hashing	Protect sensitive information by hashing personally identifiable information (PII) as it moves through your pipelines.
Support with SLAs	Priority assistance with deploying, managing and upgrading Airbyte.

A valid license key is required to get started with Airbyte Self-Managed Enterprise. Talk to sales to receive your license key. The following pages outline how to:

1. Deploy Airbyte Enterprise using Kubernetes
2. Configure Okta for Single Sign-On (SSO) with Airbyte Self-Managed Self-Managed Enterprise

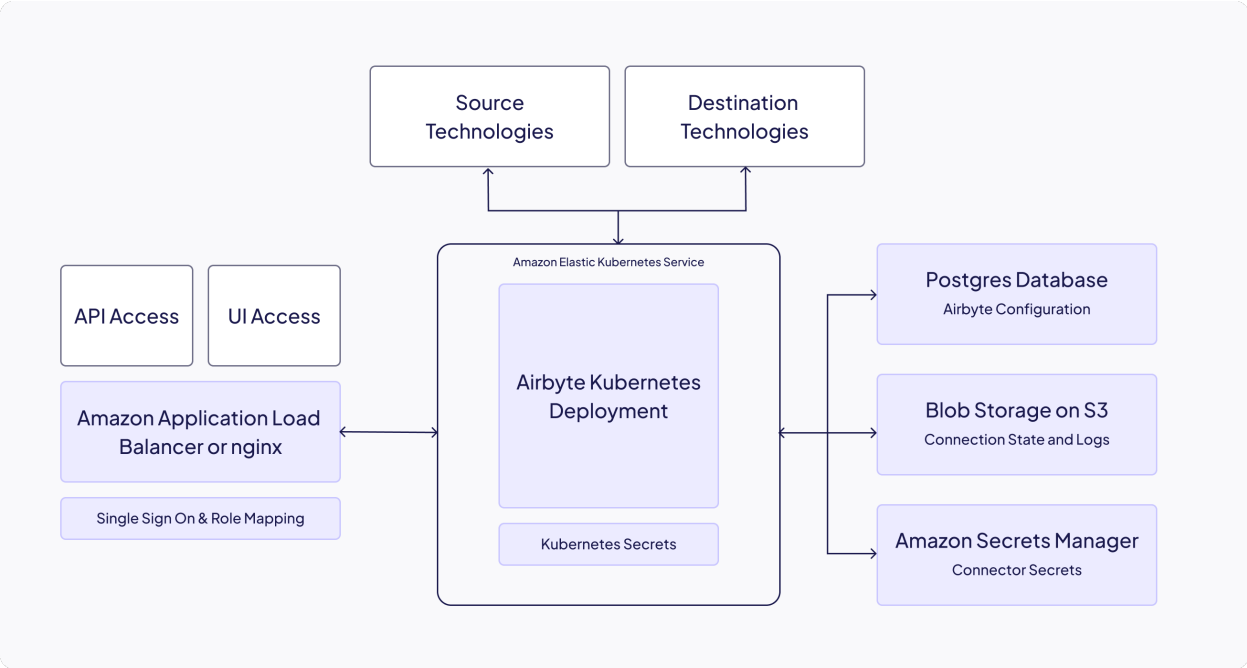

Figure 1: AWS Architecture Diagram

Implementation Guide

Once you have a license key, you can deploy Self-Managed Enterprise using the following instructions.

Airbyte Self-Managed Enterprise must be deployed using Kubernetes. This is to enable Airbyte's best performance and scale. The core Airbyte components (`server`, `workload-launcher`) run as deployments. The `workload-launcher` is responsible for managing connector-related pods (`check`, `discover`, `read`, `write`, `orchestrator`).

Prerequisites

Infrastructure Prerequisites

For a production-ready deployment of Self-Managed Enterprise, the following infrastructure components are required. Deploy to Amazon EKS or Google Kubernetes Engine. The following diagram illustrates a typical Airbyte deployment running on AWS:

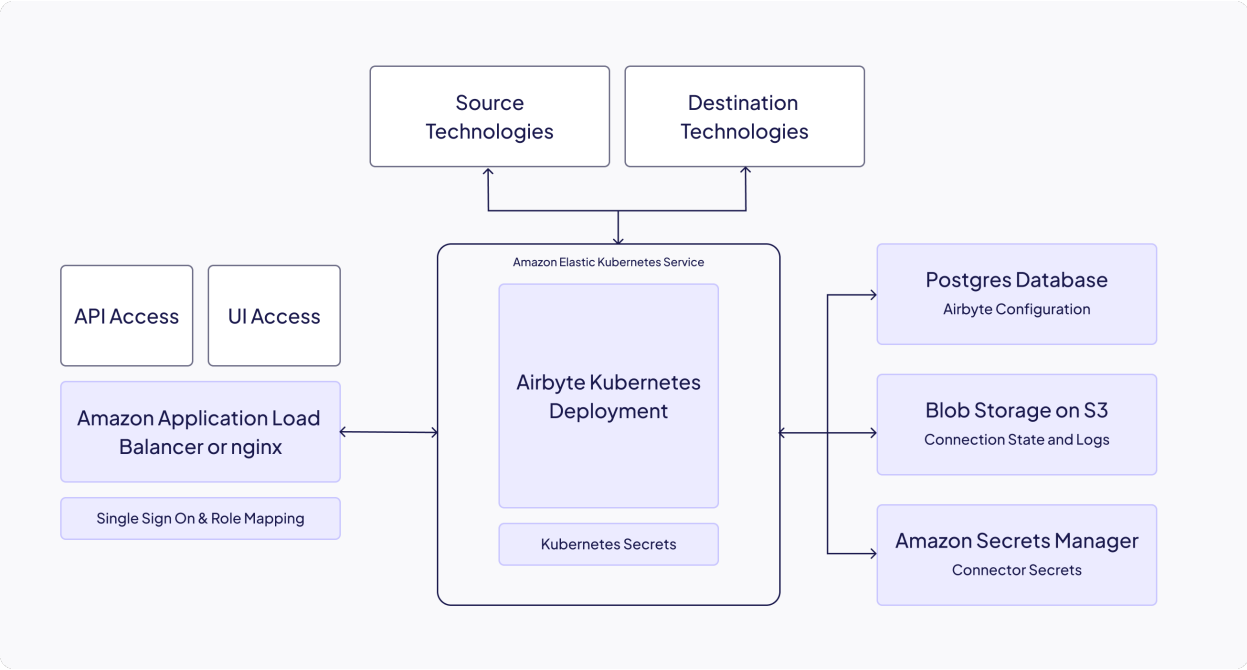

Figure 2: AWS Architecture Diagram

Prior to deploying Self-Managed Enterprise, Airbyte recommends having each of the following

infrastructure components ready to go. When possible, it's easiest to have all components running in the same VPC. The provided recommendations are for customers deploying to AWS:

Component	Recommendation
Kubernetes Cluster	Amazon EKS cluster running on EC2 instances in 2 or more availability zones on a minimum of 6 nodes.
Ingress	Amazon ALB and a URL for users to access the Airbyte UI or make API requests.
Object Storage	Amazon S3 bucket with two directories for log and state storage.
Dedicated Database	Amazon RDS Postgres with at least one read replica.
External Secrets Manager	Amazon Secrets Manager for storing connector secrets.

A few notes on Kubernetes cluster provisioning for Airbyte Self-Managed Enterprise:

- Airbyte supports Amazon Elastic Kubernetes Service (EKS) on EC2, Google Kubernetes Engine (GKE) on Google Compute Engine (GCE), and Azure Kubernetes Service (AKS).
- Airbyte recommends running Airbyte on memory-optimized instances, such as M7i / M7g instance types.
- While Airbyte supports GKE Autopilot, it doesn't support Amazon EKS on Fargate.
- You should run Airbyte on instances with at least 2 cores and 8 gigabytes of RAM.

We require you to install and configure the following Kubernetes tooling:

1. Install `helm` by following these instructions
2. Install `kubectl` by following these instructions.
3. Configure `kubectl` to connect to your cluster by using `kubectl use-context my-cluster-name`:

Configure kubectl to connect to your cluster

Amazon EKS

1. Configure your AWS CLI to connect to your project.
2. Install eksctl.
3. Run `eksctl utils write-kubeconfig --cluster=$CLUSTER_NAME` to make the context available to kubectl.
4. Use `kubectl config get-contexts` to show the available contexts.
5. Run `kubectl config use-context $EKS_CONTEXT` to access the cluster with kubectl.

GKE

1. Configure `gcloud` with `gcloud auth login`.
2. On the Google Cloud Console, the cluster page will have a "Connect" button, with a command to run locally: `gcloud container clusters get-credentials $CLUSTER_NAME --zone $ZONE_NAME --project $PROJECT_NAME`.
3. Use `kubectl config get-contexts` to show the available contexts.
4. Run `kubectl config use-context $EKS_CONTEXT` to access the cluster with kubectl.

We also require you to create a Kubernetes namespace for your Airbyte deployment:

```
kubectl create namespace airbyte
```

Configure Kubernetes Secrets

Sensitive credentials such as AWS access keys are required to be made available in Kubernetes Secrets during deployment. The Kubernetes secret store and secret keys are referenced in your `values.yaml` file. Ensure all required secrets are configured before deploying Airbyte Self-Managed Enterprise.

You may apply your Kubernetes secrets by applying the example manifests below to your cluster, or using `kubectl` directly. If your Kubernetes cluster already has permissions to make requests to an external entity via an instance profile, credentials are not required. For example, if your Amazon EKS cluster has been assigned a sufficient AWS IAM role to make requests to AWS S3, you do not need to specify access keys.

Creating a Kubernetes Secret

While you can set the name of the secret to whatever you prefer, you will need to set that name in various places in your values.yaml file. For this reason we suggest that you keep the name of `airbyte-config-secrets` unless you have a reason to change it.

airbyte-config-secrets

S3

```
apiVersion: v1
kind: Secret
metadata:
  name: airbyte-config-secrets
type: Opaque
stringData:
  # Enterprise License Key
  license-key: ## e.g. xxxxx.yyyyy.zzzzz

  # Database Secrets
  database-host: ## e.g. database.internal
  database-port: ## e.g. 5432
  database-name: ## e.g. airbyte
  database-user: ## e.g. airbyte
  database-password: ## e.g. password

  # Instance Admin
  instance-admin-email: ## e.g. admin@company.example
  instance-admin-password: ## e.g. password

  # SSO OIDC Credentials
  client-id: ## e.g. e83bbc57-1991-417f-8203-3affb47636cf
  client-secret: ## e.g. wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY

  # AWS S3 Secrets
```

```
  s3-access-key-id: ## e.g. AKIAIOSFODNN7EXAMPLE
  s3-secret-access-key: ## e.g. wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY

  # Azure Blob Storage Secrets
  azure-blob-store-connection-string: ##
  ↪ DefaultEndpointsProtocol=https;AccountName=azureintegration;AccountKey=wJa
  ↪ lrXUtnFEMI/wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY/wJalrXUtnFEMI/K7MDENG/
  ↪ bPxRfiCYEXAMPLEKEY==;EndpointSuffix=core.windows.net

  # AWS Secret Manager
  aws-secret-manager-access-key-id: ## e.g. AKIAIOSFODNN7EXAMPLE
  aws-secret-manager-secret-access-key: ## e.g.
  ↪ wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY

  # Azure Secret Manager
  azure-key-vault-client-id: ## 3fc863e9-4740-4871-bdd4-456903a04d4e
  azure-key-vault-client-secret: ## KWP6egqixiQeQoKqFZuZq2weRbYoVxMH
```

You can also use `kubectl` to create the secret directly from the CLI:

```
kubectl create secret generic airbyte-config-secrets \
  --from-literal=license-key='' \
  --from-literal=database-host='' \
  --from-literal=database-port='' \
  --from-literal=database-name='' \
  --from-literal=database-user='' \
  --from-literal=database-password='' \
  --from-literal=instance-admin-email='' \
  --from-literal=instance-admin-password='' \
  --from-literal=s3-access-key-id='' \
  --from-literal=s3-secret-access-key='' \
  --from-literal=aws-secret-manager-access-key-id='' \
  --from-literal=aws-secret-manager-secret-access-key='' \
  --namespace airbyte
```

GCS

First, create a new file `gcp.json` containing the credentials JSON blob for the service account you are looking to assume.

```
apiVersion: v1
kind: Secret
metadata:
  name: airbyte-config-secrets
type: Opaque
stringData:
  # Enterprise License Key
  license-key: ## e.g. xxxxx.yyyyy.zzzzz

  # Database Secrets
```

```
  database-host: ## e.g. database.internal
  database-port: ## e.g. 5432
  database-name: ## e.g. airbyte
  database-user: ## e.g. airbyte
  database-password: ## e.g. password

  # Instance Admin Credentials
  instance-admin-email: ## e.g. admin@company.example
  instance-admin-password: ## e.g. password

  # SSO OIDC Credentials
  client-id: ## e.g. e83bbc57-1991-417f-8203-3affb47636cf
  client-secret: ## e.g. wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY

  # GCP Secrets
  gcp.json: <CREDENTIALS_JSON_BLOB>
```

Using `kubectl` to create the secret directly from the `gcp.json` file:

```
kubectl create secret generic airbyte-config-secrets \
  --from-literal=license-key='' \
  --from-literal=database-host='' \
  --from-literal=database-port='' \
  --from-literal=database-name='' \
  --from-literal=database-user='' \
  --from-literal=database-password='' \
  --from-literal=instance-admin-email='' \
  --from-literal=instance-admin-password='' \
  --from-file=gcp.json
  --namespace airbyte
```

Installation Steps

Step 1: Add Airbyte Helm Repository

Follow these instructions to add the Airbyte helm repository:

1. Run `helm repo add airbyte https://airbytehq.github.io/helm-charts`, where `airbyte` is the name of the repository that will be indexed locally.
2. Perform the repo indexing process, and ensure your helm repository is up-to-date by running `helm repo update`.
3. You can then browse all charts uploaded to your repository by running `helm search repo airbyte`.

Step 2: Configure your Deployment

1. Inside your `airbyte` directory, create an empty `values.yaml` file.
2. Paste the following into your newly created `values.yaml` file. This is required to deploy Airbyte Self-Managed Enterprise:

```yaml
global:
  edition: enterprise
```

3. To enable SSO authentication, add instance admin details SSO auth details to your `values.yaml` file, under `global`. See the following guide on how to collect this information for various IDPs, such as Okta and Azure Entra ID.

```yaml
global:
  auth:

    # -- Admin user configuration
    instanceAdmin:
      firstName: ""
      lastName:  ""
      emailSecretKey: "" # The key within `emailSecretName` where the
      ↪  initial user's email is stored
      passwordSecretKey: "" # The key within `passwordSecretName` where the
      ↪  initial user's password is stored

    # -- SSO Identify Provider configuration; (requires Enterprise)
    identityProvider:
        secretName: "" # Secret name where the OIDC configuration is stored
        type: "oidc"
        oidc:
          # -- OIDC application domain
          domain: ""
          # -- OIDC application name
          appName: ""
          # -- The key within `clientIdSecretName` where the OIDC client id
          ↪  is stored
          clientIdSecretKey: ""
          # -- The key within `clientSecretSecretName` where the OIDC
          ↪  client secret is stored
          clientSecretSecretKey: ""
```

```yaml
global:
  auth:

    # -- Admin user configuration
    instanceAdmin:
      firstName: ""
      lastName:  ""
      emailSecretKey: "" # The key within `emailSecretName` where the initial
      ↪  user's email is stored
      passwordSecretKey: "" # The key within `passwordSecretName` where the
      ↪  initial user's password is stored

    # -- SSO Identify Provider configuration; (requires Enterprise)
```

```
    identityProvider:
        secretName: "" # Secret name where the OIDC configuration is stored
        type: "generic-oidc"
        genericOidc:
          clientId: ""
          audience: ""
          extraScopes: ""
          issuer: ""
          endpoints:
            authorizationServerEndpoint: ""
            jwksEndpoint: ""
          fields:
            subject: sub
            email: email
            name: name
            issuer: iss
```

4. You must configure the public facing URL of your Airbyte instance to your `values.yaml` file, under `global`:

   ```
   global:
     airbyteUrl: # e.g. https://airbyte.company.example
   ```

5. Verify the configuration of your `values.yaml` so far. Ensure `license-key`, `instance-admin-email` and `instance-admin-password` are all available via Kubernetes Secrets (configured in prerequisites). It should appear as follows:

Sample initial values.yaml file

```
global:
  edition: enterprise
  airbyteUrl: # e.g. https://airbyte.company.example
  auth:
    instanceAdmin:
      firstName: ## First name of admin user.
      lastName: ## Last name of admin user.
    identityProvider:
      type: oidc
      secretName: airbyte-config-secrets ## Name of your Kubernetes secret.
      oidc:
        domain: ## e.g. company.example
        appName: ## e.g. airbyte
        clientIdSecretKey: client-id
        clientSecretSecretKey: client-secret
```

The following subsections help you customize your deployment to use an external database, log storage, dedicated ingress, and more. To skip this and deploy a minimal, local version of Self-Managed Enterprise, jump to Step 3.

Configuring the Airbyte Database

For Self-Managed Enterprise deployments, you must use a dedicated database instance for better reliability and backups, such as AWS RDS or GCP Cloud SQL. Don't use the default internal Postgres database, `airbyte/db`, that Airbyte spins up within the Kubernetes cluster.

We assume in the following that you've already configured a Postgres instance. Airbyte requires Postgres 13 or later, and tests its deployments through Postgres 17. See External Database for details.

External database setup steps

Add external database details to your `values.yaml` file. This disables the default internal Postgres database (`airbyte/db`), and configures the external Postgres database. You can override all of the values below by setting them in the airbyte-config-secrets or set them directly here. You must set the database password in the airbyte-config-secrets. Here is an example configuration:

```
postgresql:
  enabled: false

global:
  database:
    # -- Secret name where database credentials are stored
    secretName: "" # e.g. "airbyte-config-secrets"
    # -- The database host
    host: ""
    # -- The database port
    port:
    # -- The database name
    name: ""

    # Use EITHER user or userSecretKey, but not both
    # -- The database user
    user: ""
    # -- The key within `secretName` where the user is stored
    userSecretKey: "" # e.g. "database-user"

    # Use EITHER password or passwordSecretKey, but not both
    # -- The database password
    password: ""
    # -- The key within `secretName` where the password is stored
    passwordSecretKey: "" # e.g."database-password"
```

Configuring External Logging

For Self-Managed Enterprise deployments, spin up standalone log storage for additional reliability using tools such as S3 and GCS. Don't use the default internal MinIO storage, `airbyte/minio`. It's then a common practice to configure additional log forwarding from external log storage into your observability tool.

External log storage setup steps

Add external log storage details to your `values.yaml` file. This disables the default internal Minio instance (`airbyte/minio`), and configures the external log database:

S3

Ensure you've already created a Kubernetes secret containing both your S3 access key ID, and secret access key. By default, secrets are expected in the `airbyte-config-secrets` Kubernetes secret, under the `aws-s3-access-key-id` and `aws-s3-secret-access-key` keys. Steps to configure these are in the above prerequisites.

```
global:
  storage:
    secretName: ""
    type: minio # default storage is minio. Set to s3, gcs, or azure, according
    ↪  to what you use.

    bucket:
      log: airbyte-bucket
      state: airbyte-bucket
      workloadOutput: airbyte-bucket
      activityPayload: airbyte-bucket
    s3:
      region: "" ## e.g. us-east-1
      authenticationType: credentials ## Use "credentials" or "instanceProfile"
      accessKeyId: ""
      secretAccessKey: ""
```

Set `authenticationType` to `instanceProfile` if the compute infrastructure running Airbyte has pre-existing permissions (e.g. IAM role) to read and write from the appropriate buckets.

GCS

Ensure you've already created a Kubernetes secret containing the credentials blob for the service account to be assumed by the cluster. Steps to configure these are in the above prerequisites.

```
global:
  storage:
    secretName: ""
    type: minio # default storage is minio. Set to s3, gcs, or azure, according
    ↪  to what you use.
    bucket:
      log: airbyte-bucket
      state: airbyte-bucket
      workloadOutput: airbyte-bucket
      activityPayload: airbyte-bucket
    gcs:
      projectId: <project-id>
      credentialsJson:  <base64-encoded>
      credentialsJsonPath: /secrets/gcs-log-creds/gcp.json
```

Azure

```
global:
  storage:
    secretName: ""
    type: minio # default storage is minio. Set to s3, gcs, or azure, according
    ↪  to what you use.
    bucket:
      log: airbyte-bucket
      state: airbyte-bucket
      workloadOutput: airbyte-bucket
      activityPayload: airbyte-bucket
    azure:
      # one of the following: connectionString, connectionStringSecretKey
      connectionString: <azure storage connection string>
      connectionStringSecretKey: <secret coordinate containing an existing
      ↪  connection-string secret>
```

Configuring External Connector Secret Management

Airbyte's default behavior is to store encrypted connector secrets on your cluster as Kubernetes secrets. You may optionally opt to instead store connector secrets in an external secret manager such as AWS Secrets Manager, Google Secrets Manager or Hashicorp Vault. Upon creating a new connector, secrets (e.g. OAuth tokens, database passwords) will be written to, then read from the configured secrets manager.

Configuring external connector secret management

Modifying the configuration of connector secret storage will cause all existing connectors to fail. You will need to recreate these connectors to ensure they are reading from the appropriate secret store.

Amazon

If authenticating with credentials, ensure you've already created a Kubernetes secret containing both your AWS Secrets Manager access key ID, and secret access key. By default, secrets are expected in the `airbyte-config-secrets` Kubernetes secret, under the `aws-secret-manager-access-key-id` and `aws-secret-manager-secret-access-key` keys. Steps to configure these are in the above prerequisites.

```
global:
  secretsManager:
    enabled: false
    type: AWS_SECRET_MANAGER
    secretName: "airbyte-config-secrets"
    awsSecretManager:
      region: <aws-region>
      authenticationType: credentials ## Use "credentials" or "instanceProfile"
      tags: ## Optional - You may add tags to new secrets created by Airbyte.
      - key: ## e.g. team
          value: ## e.g. deployments
        - key: business-unit
```

```
        value: engineering
    kms: ## Optional - ARN for KMS Decryption.
```

Set `authenticationType` to `instanceProfile` if the compute infrastructure running Airbyte has pre-existing permissions (e.g. IAM role) to read and write from AWS Secrets Manager.

To decrypt secrets in the secret manager with AWS KMS, configure the `kms` field, and ensure your Kubernetes cluster has pre-existing permissions to read and decrypt secrets.

GCP

Ensure you've already created a Kubernetes secret containing the credentials blob for the service account to be assumed by the cluster. By default, secrets are expected in the `gcp-cred-secrets` Kubernetes secret, under a `gcp.json` file. Steps to configure these are in the above prerequisites. For simplicity, we recommend provisioning a single service account with access to both GCS and GSM.

```
global:
  secretsManager:
    enabled: false
    type: GOOGLE_SECRET_MANAGER
    secretName: gcp-cred-secrets
    googleSecretManager:
      projectId: <project-id>
      credentialsSecretKey: gcp.json
```

Azure Key Vault

```
global:
  secretsManager:
    enabled: true
    type: AZURE_KEY_VAULT
    secretName: "airbyte-config-secrets"
    azureKeyVault:
      tenantId: ""
      vaultUrl: ""
      clientId: ""
      clientIdSecretKey: ""
      clientSecret: ""
      clientSecretSecretKey: ""
      tags: ""
```

Configuring Ingress

To access the Airbyte UI, you need to configure ingress for your deployment. You have two options:

- **Use Airbyte's Helm chart ingress configuration** - Configure ingress through your `values.yaml` file.
- **Bring your own ingress** - Manually create and manage your own Kubernetes ingress resource.

Use the Helm chart ingress configuration if you want Airbyte to manage ingress creation and updates automatically. Use your own ingress if you need custom ingress configurations beyond what the Helm chart provides, or if you prefer to manage ingress independently.

Before enabling ingress You must have an ingress controller deployed in your Kubernetes cluster. Refer to ingress controller documentation for setup: NGINX, AWS ALB, or your controller's documentation. For TLS certificate management, refer to cert-manager or your cloud provider's certificate service.

Set appropriate backend timeout values for the Airbyte server ingress. Timeout values that are too short can lead to 504 errors in the UI when creating new sources or destinations.

Set up ingress in Airbyte Option 1: use Airbyte's Helm chart ingress configuration

You can configure ingress directly in your `values.yaml` file. Airbyte automatically creates and manages the ingress resource for you.

```
ingress:
  enabled: true
  className: "nginx"  # Specify your ingress class
  annotations: {}
    # Add any ingress-specific annotations here
  hosts:
    - host: airbyte.example.com  # Replace with your domain
      paths:
        - path: /auth
          pathType: Prefix
          backend: keycloak  # For Keycloak authentication (if using OIDC)
        - path: /
          pathType: Prefix
          backend: server  # Routes to airbyte-server
        - path: /connector-builder
          pathType: Prefix
          backend: connector-builder-server  # Required for connector builder
  tls: []
    # Optionally configure TLS
    # - secretName: airbyte-tls
    #   hosts:
    #     - airbyte.example.com
```

The `backend` field specifies which service to route to:

- `keycloak` - Routes to Keycloak service (required for OIDC authentication, omit if using generic OIDC)
- `server` - Routes to the main Airbyte server
- `connector-builder-server` - Routes to the connector builder service (required for Airbyte's connector builder to work)

Option 2: bring your own ingress

If you prefer to manage your own ingress resource, you can manually create a Kubernetes ingress resource.

NGINX

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: # ingress name, example: enterprise-demo
  annotations:
    nginx.ingress.kubernetes.io/ssl-redirect: "false"
spec:
  ingressClassName: nginx
  rules:
    - host: airbyte.example.com # replace with your host
      http:
        paths:
          - backend:
              service:
                # format is ${RELEASE_NAME}-airbyte-keycloak-svc
                name: airbyte-enterprise-airbyte-keycloak-svc
                port:
                  number: 8180
            path: /auth
            pathType: Prefix
          - backend:
              service:
                # format is ${RELEASE_NAME}-airbyte-connector-builder-server-svc
                name: airbyte-enterprise-airbyte-connector-builder-server-svc
                port:
                  number: 80 # service port, example: 8080
            path: /api/v1/connector_builder/
            pathType: Prefix
          - backend:
              service:
                # format is ${RELEASE_NAME}-airbyte-server-svc
                name: airbyte-enterprise-airbyte-server-svc
                port:
                  number: 8001 # service port, example: 8080
            path: /
            pathType: Prefix
```

Amazon ALB

If you are intending on using Amazon Application Load Balancer (ALB) for ingress, this ingress definition will be close to what's needed to get up and running:

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
```

```
  name: airbyte-ingress # ingress name, e.g. airbyte-production-ingress
  annotations:
    # Specifies that the Ingress should use an AWS ALB.
    kubernetes.io/ingress.class: "alb"
    # Redirects HTTP traffic to HTTPS.
    alb.ingress.kubernetes.io/ssl-redirect: "443"
    # Creates an internal ALB, which is only accessible within your VPC or
    ↪ through a VPN.
    alb.ingress.kubernetes.io/scheme: internal
    # Specifies the ARN of the SSL certificate managed by AWS ACM, essential for
    ↪ HTTPS.
    alb.ingress.kubernetes.io/certificate-arn: arn:aws:acm:us-east-x:xxxxxxxxx:c⌟
    ↪ ertificate/xxxxxxxxx-xxxxx-xxxx-xxxx-xxxxxxxxxxx
    # Sets the idle timeout value for the ALB.
    alb.ingress.kubernetes.io/load-balancer-attributes:
    ↪ idle_timeout.timeout_seconds=30
    # [If Applicable] Specifies the VPC subnets and security groups for the ALB
    # alb.ingress.kubernetes.io/subnets: '' e.g. 'subnet-12345, subnet-67890'
    # alb.ingress.kubernetes.io/security-groups: <SECURITY_GROUP>
spec:
  rules:
    - host: airbyte.example.com # replace with your host
      http:
        paths:
          - backend:
              service:
                name: airbyte-enterprise-airbyte-keycloak-svc
                port:
                  number: 8180
            path: /auth
            pathType: Prefix
          - backend:
              service:
                name: airbyte-enterprise-airbyte-connector-builder-server-svc
                port:
                  number: 80
            path: /api/v1/connector_builder/
            pathType: Prefix
          - backend:
              service:
                name: airbyte-enterprise-airbyte-server-svc
                port:
                  number: 8001
            path: /
            pathType: Prefix
```

The ALB controller will use a `ServiceAccount` that requires the following IAM policy to be attached.

Ensure your airbyte URL matches your ingress host Once you configure ingress, ensure that the value of `global.airbyteUrl` in your values.yaml matches the ingress URL.

```
global:
  airbyteUrl: # e.g. https://airbyte.example.com
```

You may configure ingress using a load balancer or an API Gateway. We do not currently support most service meshes (such as Istio). If you are having networking issues after fully deploying Airbyte, please verify that firewalls or lacking permissions are not interfering with pod-pod communication. Please also verify that deployed pods have the right permissions to make requests to your external database.

Step 3: Deploy Self-Managed Enterprise

Install Airbyte Self-Managed Enterprise on helm using the following command:

1. Identify the Helm chart version that corresponds to the platform version you want to run. Most Helm chart versions are designed to work with one Airbyte version, and they don't necessarily have the same version number.

   ```
   helm search repo airbyte --versions
   ```

2. Install Airbyte Self-Managed Enterprise.

   ```
   helm install airbyte airbyte/airbyte \
     --namespace airbyte \           # Target Kubernetes namespace
     --values ./values.yaml \       # Custom configuration values
     --version 2.x.x                # Helm chart version to use
   ```

To uninstall Self-Managed Enterprise, run `helm uninstall airbyte-enterprise`.

Updating Self-Managed Enterprise

Upgrade Airbyte Self-Managed Enterprise by:

1. Running `helm repo update`. This pulls an up-to-date version of our helm charts, which is tied to a version of the Airbyte platform.
2. Re-installing Airbyte Self-Managed Enterprise:

   ```
   helm upgrade airbyte airbyte/airbyte \
     --namespace airbyte \           # Target Kubernetes namespace
     --values ./values.yaml \       # Custom configuration values
     --version 2.x.x                # Helm chart version to use
   ```

Customizing your Deployment

In order to customize your deployment, you need to create an additional `values.yaml` file in your `airbyte` directory, and populate it with configuration override values. A thorough `values.yaml` example including many configurations can be located in Values.yaml reference folder of the Airbyte repository.

After specifying your own configuration, run the following command:

```
helm upgrade airbyte airbyte/airbyte \
  --namespace airbyte \           # Target Kubernetes namespace
  --values ./values.yaml \       # Custom configuration values
  --version 2.x.x                # Helm chart version to use
```

Configure a custom image registry

You can optionally configure Airbyte to pull Docker images from a custom image registry rather than Airbyte's public Docker repository. In this case, Airbyte pulls both platform images (e.g. `server`, `workload-launcher`, etc.) and connector images (e.g. Postgres Source, S3 Destination, etc.) from the configured registry.

Implementing Airbyte this way has several advantages.

- **Security**: Private custom image registries keep images in your network, reducing the risk of external threats.
- **Access control**: You have more control over who can access and modify images.
- **Compliance**: By keeping images in a controlled environment, it's easier to prove compliance with regulatory requirements for data storage and handling.

Custom Docker connectors in your workspace that specify an image using a fully qualified domain name (for example, `example.com/airbyte/your-custom-source`) ignore your configured custom image registry and pull images from the domain specified by that connector.

Before you start

1. Set up your custom image registry. The examples in this article use GitHub, but you have many options. Here are some popular ones:
2. Install `abctl`. Although abctl is typically only used to manage local installations of Airbyte, it has some helpful commands for this process.

Homebrew

    ```bash
    brew tap airbytehq/tap
    brew install abctl
    ```

Go

    ```bash
    go install github.com/airbytehq/abctl@latest
    ```

GitHub

    See [GitHub releases](https://github.com/airbytehq/abctl/releases/latest).

Get a list of all Airbyte images

To get a list of Airbyte images for the latest version, use abctl.

```
abctl images manifest
```

You should see something like this:

```
airbyte/bootloader:1.8.0
airbyte/connector-builder-server:1.8.0
airbyte/connector-sidecar:1.8.0
airbyte/container-orchestrator:1.8.0
airbyte/cron:1.8.0
airbyte/db:1.8.0
airbyte/mc:latest
airbyte/server:1.8.0
airbyte/worker:1.8.0
airbyte/workload-api-server:1.8.0
airbyte/workload-init-container:1.8.0
airbyte/workload-launcher:1.8.0
bitnami/kubectl:1.28.9
busybox:1.35
busybox:latest
curlimages/curl:8.1.1
minio/minio:RELEASE.2023-11-20T22-40-07Z
temporalio/auto-setup:1.23.0
```

Step 1: Customize Airbyte to use your image registry

To pull all platform and connector images from a custom image registry, add the following customization to Airbyte's `values.yaml` file, replacing the `registry` value with your own registry location.

```
global:
  image:
    registry: ghcr.io/NAMESPACE
```

If your registry requires authentication, you can create a Kubernetes secret and reference it in the Airbyte config:

1. Create a Kubernetes secret. In this example, you create a secret called `regcred` from a config file. That file contains authentication information for a private custom image registry. Learn more about Kubernetes secrets.

   ```
   kubectl create secret generic regcred \
   --from-file=.dockerconfigjson=<path/to/.docker/config.json> \
   --type=kubernetes.io/dockerconfigjson
   ```

2. Add the secret you created to your `values.yaml` file. In this example, you use your `regcred` secret to authenticate.

   ```
   global:
     image:
       registry: ghcr.io/NAMESPACE
     // highlight-start
     imagePullSecrets:
   ```

```
        - name: regcred
      // highlight-end
```

Step 2: Tag and push Airbyte images

Tag and push Airbyte's images to your custom image registry.

In this example, you tag all platform images and push them all to GitHub.

```
abctl images manifest | xargs -L1 -I{} docker tag {} ghcr.io/NAMESPACE/{} &&
↪  docker push ghcr.io/NAMESPACE/{}
```

You can also pull Airbyte's connector images from Docker, tag them, and push them to your custom image registry. You must do this prior to adding a source or destination.

In this example, you pull a connector from Docker, tag it, and push it to GitHub.

```
docker pull airbyte/destination-google-sheets:latest
docker tag airbyte/desination-google-sheets:latest
↪  ghcr.io/NAMESPACE/desination-google-sheets:latest
docker push ghcr.io/NAMESPACE/destination-google-sheets:latest
```

Now, when you install Airbyte, images will come from the custom image registry you configured.

Customizing your Service Account

You may choose to use your own service account instead of the Airbyte default, `airbyte-sa`. This may allow for better audit trails and resource management specific to your organizational policies and requirements.

To do this, add the following to your `values.yaml`:

```
serviceAccount:
  name:
```

Deploying to multiple regions

See Multiple region deployments.

Enabling audit logs

See Audit logging.

AWS Policies Appendix

Ensure your access key is tied to an IAM user or you are using a Role with the following policies.

AWS S3 Policy

The following policies, allow the cluster to communicate with S3 storage

```
{
  "Version": "2012-10-17",
  "Statement":
```

```
    [
      { "Effect": "Allow", "Action": "s3:ListAllMyBuckets", "Resource": "*" },
      {
        "Effect": "Allow",
        "Action": ["s3:ListBucket", "s3:GetBucketLocation"],
        "Resource": "arn:aws:s3
> **Your: -S3-BUCKET-NAME"**
>
> },
      {
        "Effect": "Allow",
        "Action":
          [
            "s3:PutObject",
            "s3:PutObjectAcl",
            "s3:GetObject",
            "s3:GetObjectAcl",
            "s3:DeleteObject"
          ],
        "Resource": "arn:aws:s3
YOUR-S3-BUCKET-NAME/*"
      }
    ]
}
```

AWS Secret Manager Policy

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "secretsmanager:GetSecretValue",
                "secretsmanager:CreateSecret",
                "secretsmanager:ListSecrets",
                "secretsmanager:DescribeSecret",
                "secretsmanager:TagResource",
                "secretsmanager:UpdateSecret"
            ],
            "Resource": [
                "*"
            ],
            "Condition": {
                "ForAllValues:StringEquals": {
                    "secretsmanager:ResourceTag/AirbyteManaged": "true"
                }
            }
```

```
        }
    ]
}
```

Azure Policies Appendix

Azure Key Vault Policy

Airbyte requires the ability to write and read secrets in an Azure Key Vault. The built-in role that supports this is the Key Vault Secrets Officer role, whose JSON configuration can be viewed below to understand the specific permissions needed.

```
{
    "id": "/providers/Microsoft.Authorization/roleDefinitions/b86a8fe4-44ce-4948
    ↪ -aee5-eccb2c155cd7",
    "properties": {
        "roleName": "Key Vault Secrets Officer",
        "description": "Perform any action on the secrets of a key vault, except
        ↪ manage permissions. Only works for key vaults that use the 'Azure
        ↪ role-based access control' permission model.",
        "assignableScopes": [
            "/"
        ],
        "permissions": [
            {
                "actions": [
                    "Microsoft.Authorization/*/read",
                    "Microsoft.Insights/alertRules/*",
                    "Microsoft.Resources/deployments/*",
                    "Microsoft.Resources/subscriptions/resourceGroups/read",
                    "Microsoft.Support/*",
                    "Microsoft.KeyVault/checkNameAvailability/read",
                    "Microsoft.KeyVault/deletedVaults/read",
                    "Microsoft.KeyVault/locations/*/read",
                    "Microsoft.KeyVault/vaults/*/read",
                    "Microsoft.KeyVault/operations/read"
                ],
                "notActions": [],
                "dataActions": [
                    "Microsoft.KeyVault/vaults/secrets/*"
                ],
                "notDataActions": []
            }
        ]
    }
}
```

Multiple region deployments (Self-Managed Enterprise)

Self-Managed Enterprise customers can use Airbyte's public API to define regions and create independent data planes that operate in those regions. This ensures you're satisfying your data residency and governance requirements with a single Airbyte deployment, and it can help you reduce data egress costs with cloud providers.

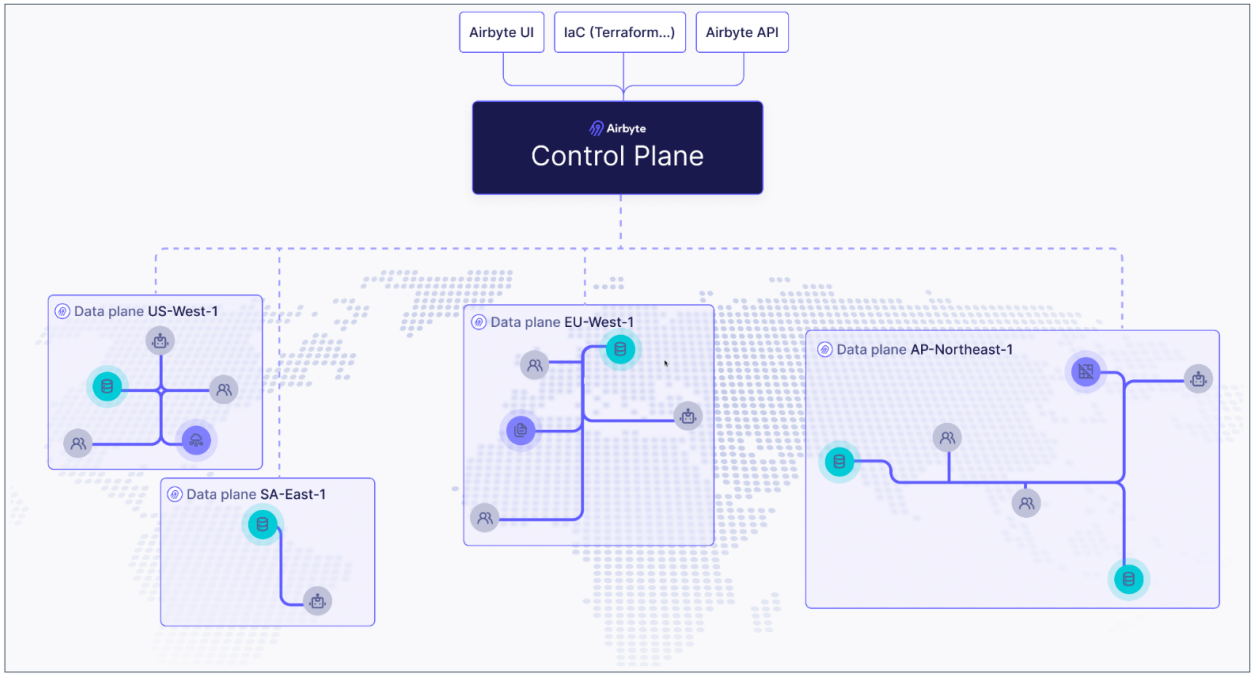

Figure 3: Stylized diagram showing a control plane above multiple data planes in different global regions

How it works

If you're not familiar with Kubernetes, think of the control plane as the brain and data planes as the muscles doing work the brain tells them to do.

- The control plane is responsible for Airbyte's user interface, APIs, Terraform provider, and

orchestrating work.
- The data plane initiates jobs, syncs data, completes jobs, and reports its status back to the control plane.

This separation of duties is what allows a single Airbyte deployment to ensure your data remains segregated and compliant.

By default, Airbyte has a single data plane that any workspace in the organization can access, and it's automatically tied to the default workspace when Airbyte first starts. To configure additional data planes and regions, complete these steps.

1. Create a region.
2. Create a data plane in that region.
3. Associate your data plane to an Airbyte workspace. You can tie each workspace to exactly one region.
4. Configure Kubernetes secrets.
5. Create your values.yaml file.
6. Deploy your data plane.

Once you complete these steps, jobs in your workspace move data exclusively using the data plane mapped to that region.

Limitations and considerations

While data planes process data in their respective regions, some metadata remains in the control plane.

- Airbyte stores Cursor and Primary Key data in the control plane regardless of data plane location. If you have data that you can't store in the control plane, don't use it as a cursor or primary key.
- The Connector Builder processes all data through the control plane, regardless of workspace settings. This limitation applies to the development and testing phase only; published connectors respect workspace data residency settings during syncs.
- If you're using a secrets manager, the secrets manager used by the control plane is the one used by the data plane.

Prerequisites

Before you begin, make sure you've completed the following.

- Deploy your Self-Managed Enterprise version of Airbyte as described in the implementation guide.
- You must be an Instance Administrator to manage regions and data planes.
- You need a Kubernetes cluster on which your data plane can run. For example, if your Airbyte control plane already runs on an EKS cluster on `us-west-2`, and you want your data plane to run on `eu-west-1`, create an EKS cluster on `eu-west-1`.
- If you haven't already, get access to Airbyte's API by creating an application and generating an access token. For help, see Configuring API access.

1. Create a region

The first step is to create a region. Regions are objects that contain data planes, and which you associate to workspaces.

Request

Send a POST request to /v1/regions/.

```
curl --request POST \
  --url https://example.com/api/public/v1/regions \
  --header "Authorization: Bearer $TOKEN" \
  --header "Content-Type: application/json" \
  --data '{
  "name": "aws-us-east-1",
  "organizationId": "00000000-0000-0000-0000-000000000000"
}'
```

Include the following parameters in your request.

Body parameter	Required?	Description
`name`	Required	The name of your region in Airbyte. For simplicity, you might want to make this the same as the actual cloud region this region runs on.
`organizationId`	Required	Your Airbyte organization ID. In most cases, this is `00000000-0000-0000-0000-000000000000`.
`enabled`	Optional	Defaults to true. Set this to `false` if you don't want this region enabled.

For additional request examples, see the API reference.

Response

Make note of your `regionId`. You need it to create a data plane.

```
{
  //highlight-next-line
  "regionId": "uuid-string",
  "name": "region-name",
  "organizationId": "org-uuid-string",
  "enabled": true,
  "createdAt": "timestamp-string",
  "updatedAt": "timestamp-string"
}
```

2. Create a data plane

Once you have a region, you create a data plane within it.

Request

Send a POST request to /v1/dataplanes.

```
curl -X POST https://example.com/api/public/v1/dataplanes \
  --header "Authorization: Bearer $TOKEN" \
  --header "Content-Type: application/json" \
  -d '{
    "name": "My Data Plane",
    "regionId": "780d5bd9-a8a0-43cf-8b35-cc2061ad8319"
  }'
```

Include the following parameters in your request.

Body parameter	Required?	Description
`name`	Required	The name of your data plane. For simplicity, you might want to name it based on the region in which you created it.
`regionId`	Optional	The region this data plane belongs to.

For additional request examples, see the API reference.

Response

Make note of your `dataplaneId`, `clientId` and `clientSecret`. You need these values later to deploy your data plane on Kubernetes.

```
json
{
  "dataplaneId": "uuid-string",
  "clientId": "client-id-string",
  "clientSecret": "client-secret-string"
}
```

3. Associate a region to a workspace

One you have a region and a data plane, you need to associate that region to your workspace. You can associate a workspace with a region when you create that workspace or later, after it exists.

> **Note: You can only associate each workspace with one region.**

UI

Follow these steps to associate your region to your current workspace using Airbyte's user interface.

1. In the navigation panel, click **Workspace settings** > **General**.
2. Under **Region**, select your region.
3. Click **Save changes**.

API

When creating a new workspace:

Request

Send a POST request to /v1/workspaces/

```
curl -X POST "https://example.com/api/public/v1/workspaces" \
  --header "Authorization: Bearer $TOKEN" \
  --header "Content-Type: application/json" \
  -d '{
    "name": "My New Workspace",
    "dataResidency": "auto"
  }'
```

Include the following parameters in your request.

Body parameter	Description
`name`	The name of your workspace in Airbyte.
`dataResidency`	A string with a region identifier you received in step 1.

For additional request examples, see the API reference.

Response

```
{
  "workspaceId": "uuid-string",
  "name": "workspace-name",
  "dataResidency": "auto",
  "notifications": {
    "failure": {},
    "success": {}
  }
}
```

When updating a workspace:

Request

Send a PATCH request to /v1/workspaces/`{workspaceId}`.

```
curl -X PATCH "https://example.com/api/public/v1/workspaces/{workspaceId}" \
  --header "Authorization: Bearer $TOKEN" \
  --header "Content-Type: application/json" \
  -d '{
    "name": "Updated Workspace Name",
    "dataResidency": "us-west"
  }'
```

Include the following parameters in your request.

Body parameter	Description
`name`	The name of your workspace in Airbyte.
`dataResidency`	A string with a region identifier you received in step 1.

For additional request examples, see the API reference.

Response

```
{
  "workspaceId": "uuid-string",
  "name": "updated-workspace-name",
  "dataResidency": "region-identifier",
  "notifications": {
    "failure": {},
    "success": {}
  }
}
```

4. Configure Kubernetes Secrets

Your data plane relies on Kubernetes secrets to identify itself with the control plane.

In step 5, you create a values.yaml file that references this Kubernetes secret store and these secret keys. Configure all required secrets before deploying your data plane.

You may apply your Kubernetes secrets by applying the example manifests below to your cluster, or using kubectl directly. If your Kubernetes cluster already has permissions to make requests to an external entity via an instance profile, credentials aren't required. For example, if your Amazon EKS cluster has a sufficient AWS IAM role to make requests to AWS S3, you don't need to specify access keys.

While you can set the name of the secret to whatever you prefer, you need to set that name in your values.yaml file. For this reason it's easiest to keep the name of airbyte-config-secrets unless you have a reason to change it.

airbyte-config-secrets

S3

```
apiVersion: v1
kind: Secret
metadata:
  name: airbyte-config-secrets
type: Opaque
data:
  # Enterprise License Key
  license-key: your-airbyte-license-key

  # Insert the data plane credentials received in step 2
  DATA_PLANE_CLIENT_ID: your-data-plane-client-id
  DATA_PLANE_CLIENT_SECRET: your-data-plane-client-id

  # Only set these values if they are also set on your control plane
  AWS_SECRET_MANAGER_ACCESS_KEY_ID: your-aws-secret-manager-access-key
  AWS_SECRET_MANAGER_SECRET_ACCESS_KEY: your-aws-secret-manager-secret-key
  S3_ACCESS_KEY_ID: your-s3-access-key
  S3_SECRET_ACCESS_KEY: your-s3-secret-key
```

Apply your secrets manifest in your command-line tool with `kubectl`: `kubectl apply -f <file>.yaml -n <namespace>`.

You can also use `kubectl` to create the secret directly from the command-line tool:

```
kubectl create secret generic airbyte-config-secrets \
  --from-literal=license-key='' \
  --from-literal=DATA_PLANE_CLIENT_ID='' \
  --from-literal=DATA_PLANE_CLIENT_SECRET='' \
  --from-literal=s3-access-key-id='' \
  --from-literal=s3-secret-access-key='' \
  --from-literal=aws-secret-manager-access-key-id='' \
  --from-literal=aws-secret-manager-secret-access-key='' \
  --namespace airbyte
```

GCS

First, create a new file `gcp.json` containing the credentials JSON blob for the service account you are looking to assume.

```
apiVersion: v1
kind: Secret
metadata:
  name: airbyte-config-secrets
type: Opaque
stringData:
  # Enterprise License Key
  license-key: your-airbyte-license-key

  # Insert the data plane credentials received in step 2
  DATA_PLANE_CLIENT_ID: your-data-plane-client-id
  DATA_PLANE_CLIENT_SECRET: your-data-plane-client-id

  # Only set these values if they are also set on your control plane
  AWS_SECRET_MANAGER_ACCESS_KEY_ID: your-aws-secret-manager-access-key
  AWS_SECRET_MANAGER_SECRET_ACCESS_KEY: your-aws-secret-manager-secret-key
  S3_ACCESS_KEY_ID: your-s3-access-key
  S3_SECRET_ACCESS_KEY: your-s3-secret-key

  # GCP Secrets
  gcp.json: <CREDENTIALS_JSON_BLOB>
```

Apply your secrets manifest in your command-line tool with `kubectl`: `kubectl apply -f <file>.yaml -n <namespace>`.

You can also use `kubectl` to create the secret directly from the command-line tool:

```
kubectl create secret generic airbyte-config-secrets \
  --from-literal=license-key='' \
  --from-literal=DATA_PLANE_CLIENT_ID='' \
  --from-literal=DATA_PLANE_CLIENT_SECRET='' \
  --from-literal=s3-access-key-id='' \
```

```
  --from-literal=s3-secret-access-key='' \
  --from-literal=aws-secret-manager-access-key-id='' \
  --from-literal=aws-secret-manager-secret-access-key='' \
  --from-file=gcp.json
  --namespace airbyte
```

5. Create your deployment values

Add the following overrides to a new `values.yaml` file.

```
airbyteUrl: https://airbyte.example.com # Base URL for the control plane so
↪  Airbyte knows where to authenticate

edition: enterprise # Required for Self-Managed Enterprise

# Logging:
#  level: DEBUG

dataPlane:
  # Used to render the data plane creds secret into the Helm chart.
  secretName: airbyte-config-secrets
  id: "preview-data-plane"

  # Describe secret name and key where each of the client ID and secret are
  ↪  stored
  clientIdSecretName: airbyte-config-secrets
  clientIdSecretKey: "DATA_PLANE_CLIENT_ID"
  clientSecretSecretName: airbyte-config-secrets
  clientSecretSecretKey: "DATA_PLANE_CLIENT_SECRET"

# Describe the secret name and key where the Airbyte license key is found
enterprise:
  secretName: airbyte-config-secrets
  licenseKeySecretKey: AIRBYTE_LICENSE_KEY

# S3 bucket secrets/config
# Only set this section if the control plane has also set these values.
storage:
  secretName: airbyte-config-secrets
  type: "s3"
  bucket:
    log: my-bucket-name
    state: my-bucket-name
    workloadOutput: my-bucket-name
  s3:
    region: "us-west-2"
    authenticationType: credentials
    accessKeyIdSecretKey: S3_ACCESS_KEY_ID
```

```
    secretAccessKeySecretKey: S3_SECRET_ACCESS_KEY

# Secret manager secrets/config
# Only set this section if the control plane has also set these values.
secretsManager:
  secretName: airbyte-config-secrets
  type: AWS_SECRET_MANAGER
  awsSecretManager:
    region: us-west-2
    authenticationType: credentials
    accessKeyIdSecretKey: AWS_SECRET_MANAGER_ACCESS_KEY_ID
    secretAccessKeySecretKey: AWS_SECRET_MANAGER_SECRET_ACCESS_KEY
```

6. Deploy your data plane

In your command-line tool, deploy the data plane using `helm upgrade`. The examples here may not reflect your actual Airbyte version and namespace conventions, so make sure you use the settings that are appropriate for your environment.

```
helm upgrade --install airbyte-enterprise airbyte/airbyte-data-plane --version
↪  1.6.0 --values values.yaml
```

```
helm upgrade --install airbyte-enterprise airbyte/airbyte-data-plane --version
↪  1.6.0 -n airbyte-dataplane --create-namespace --values values.yaml
```

Check which region your workspaces use

UI

You can see a list of your workspaces and the region associated to each from Airbyte's organization settings.

1. In Airbyte's user interface, click **Workspace settings** > **General**. Airbyte displays your workspaces and each workspace region under **Regions**.

API

Request:

```
bash
curl -X GET "https://example.com/api/public/v1/workspaces/{workspaceId}" \
  -H "Authorization: Bearer YOUR_ACCESS_TOKEN" \
  -H "Content-Type: application/json"
```

Response:

```
{
  "workspaceId": "18dccc91-0ab1-4f72-9ed7-0b8fc27c5826",
  "name": "Acme Company",
  //highlight-next-line
  "dataResidency": "auto",
}
```

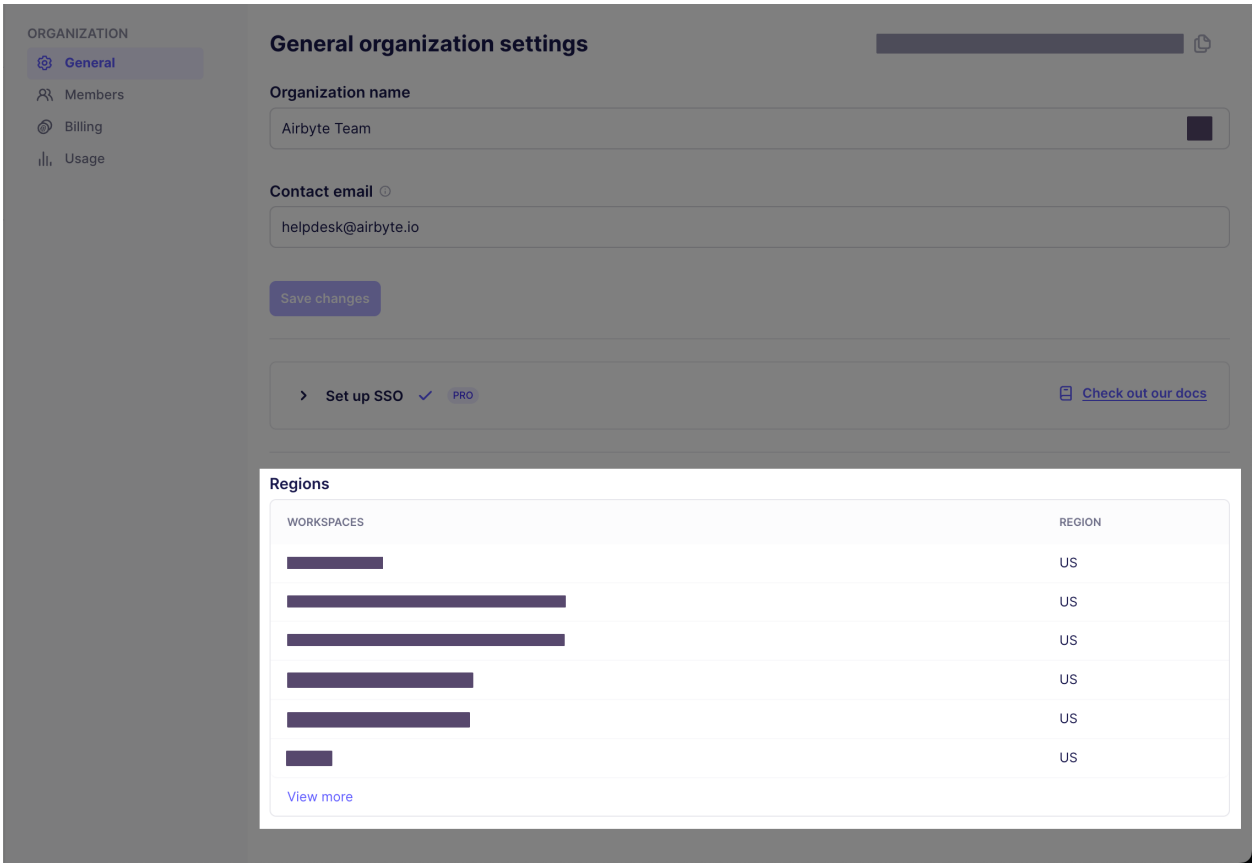

Figure 4: Multiple regions displayed in Airbyte's General Organization settings

Audit logging

Audit logging gives you visibility into data, environment, user, and permission changes. This data ensures you have records of any unauthorized changes and insider threats, making it easy to continue meeting your compliance obligations while using Airbyte.

What Airbyte logs

Event	Logged operations
Connections	Create, Update, Delete
Sources	Create, Update, Delete, Partial Update, Upgrade Version
Destinations	Create, Update, Delete, Partial Update, Upgrade Version
Organizations	Create, Update, Delete
Workspaces	Create, Update, Delete, Update Name, Update Organization
Permissions	Create, Update, Delete
Users	Create, Update, Delete

How it works

You set up a blob storage solution to store audit logs. Then, you use Airbyte's `values.yaml` file to configure that storage solution in Airbyte. Once enabled, Airbyte writes audit logs to the `/audit-logging/` directory as JSON files. These files have the following naming convention: `<yyyyMMddHHmmss>_<hostname>_<random UUID>`. You can process these logs outside of Airbyte using any tool you like.

Log format

Here is an example log for a deleted workspace. Logs for all events follow this same basic format.

```
{
  "id": "8478fcbd-d369-4bda-8d9b-b782cea5ad40",
  "timestamp": 1746724563299,
  "actor": {
    "actorId": "1c26c465-58h8-43e6-8jko-2252b7g8a9e2",
    "email": "user@airbyte.io",
    "ipAddress": "192.000.000.0",
```

```
    "userAgent": "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_7)
    ↪  AppleWebKit/537.36 (KHTML, like Gecko) Chrome/136.0.0.0 Safari/537.36"
  },
  "operation": "deleteWorkspace",
  "request": "<request object>",
  "response": "<response object>",
  "success": true,
  "errorMessage": null
}
```

Enable logging

Self-Managed Enterprise

To enable logging, set up a blob storage solution to store audit logs. Then, you use Airbyte's `values.yaml` file to configure that storage solution in Airbyte.

Step 1: Configure blob storage

Choose a new blob storage bucket with your chosen cloud provider (for example, AWS S3, GCS, or Azure Blob Storage). This bucket must be accessible by Airbyte's pods using the same credentials it uses to access your log and state storage.

Step 2: Configure audit logging in Airbyte

1. Configure Airbyte to read from and write to that bucket by modifying your values.yaml file.

```
server:
  auditLoggingEnabled: true

storage:
  bucket:
    auditLogging: your-audit-logging-bucket-name-here
```

2. Redeploy Airbyte.

```
helm upgrade airbyte airbyte/airbyte \
  --namespace airbyte \           # Target Kubernetes namespace
  --values ./values.yaml \       # Custom configuration values
  --version 2.x.x                # Helm chart version to use
```

Cloud

Airbyte implements audit logging on all Airbyte Cloud instances. You don't have access to these logs, but the Airbyte team can reference them if you need assistance with a security investigation.

Scaling Airbyte After Installation

Once you've completed the initial installation of Airbyte Self-Managed Enterprise, the next crucial step is scaling your setup as needed to ensure optimal performance and reliability as your data integration needs grow. This guide walks you through best practices and strategies for scaling Airbyte in an enterprise environment.

Concurrent Syncs

The primary driver of increased resource usage in Airbyte is the number of concurrent syncs running at any given time. Each concurrent sync requires at least 3 additional connector pods to be running at once (`orchestrator`, `read`, `write`). For example, 10 concurrent syncs require 30 additional pods in your namespace. Connector pods last only for the duration of a sync, and will be appended by the ID of the ongoing job.

If your deployment of Airbyte is intended to run many concurrent syncs at once (e.g. an overnight backfill), you are likely to require an increased number of instances to run all syncs.

Connector CPU & Memory Settings

Some connectors are memory and CPU intensive, while others are not. Using an infrastructure monitoring tool, we recommend measuring the following at all times:

- Requested CPU %
- CPU Usage %
- Requested Memory %
- Memory Usage %

If your nodes are under high CPU or Memory usage, we recommend scaling up your Airbyte deployment to a larger number of nodes, or reducing the maximum resource usage by any given connector pod. If high *requested* CPU or memory usage is blocking new pods from being scheduled, while *used* CPU or memory is low, you may modify connector pod provisioning defaults in your `values.yaml` file:

```
global:
  edition: "enterprise"
  # ...
  jobs:
    resources:
      limits:
        cpu: ## e.g. 250m
```

```
        memory: ## e.g. 500m
      requests:
        cpu: ## e.g. 75m
        memory: ## e.g. 150m
```

If your Airbyte deployment is under-provisioned, you may notice occasional 'stuck jobs' that remain in-progress for long periods, with eventual failures related to unavailable pods. Increasing job CPU and memory limits may also allow for increased sync speeds. For help and best practices, see Configuring connector resources.

Concurrent Sync Limits

To help rightsize Airbyte deployments and reduce the likelihood of stuck syncs, there are configurable limits to the number of syncs that can be run at once:

```
worker:
  maxSyncWorkers: ## e.g. 5
  maxCheckWorkers: ## e.g. 5
```

If you intend to run many syncs at the same time, you may also want to increase the number of worker replicas that run in your Airbyte instance:

```
worker:
  replicaCount: ## e.g. 2
```

Multiple Node Groups

To reduce the blast radius of an underprovisioned Airbyte deployment, place 'static' workloads (`server`, etc.) on one Kubernetes node group, while placing job-related workloads (connector pods) on a different Kubernetes node group. This ensures that UI or API availability is unlikely to be impacted by the number of concurrent syncs.

Configure Airbyte Self-Managed Enterprise to run in two node groups

```
global:
  jobs:
    kube:
      nodeSelector:
        type: jobs
      scheduling:
        check:
          nodeSelectors:
            type: jobs
        discover:
          nodeSelectors:
            type: jobs
        spec:
          nodeSelectors:
            type: jobs

airbyteBootloader:
```

```
  nodeSelector:
    type: static

server:
  nodeSelector:
    type: static

keycloak:
  nodeSelector:
    type: static

keycloakSetup:
  nodeSelector:
    type: static

temporal:
  nodeSelector:
    type: static

workloadLauncher:
  nodeSelector:
    type: static

worker:
  nodeSelector:
    type: jobs

workloadApiServer:
  nodeSelector:
    type: jobs
```

High Availability

You may wish to implement high availability (HA) to minimize downtime and ensure continuous data integration processes. Please note that this requires provisioning Airbyte on a larger number of Nodes, which may increase your licensing fees. For a typical HA deployment, you will want a VPC with subnets in at least two (and preferably three) availability zones (AZs).

We particularly recommend having multiple instances of `worker` and `server` pods:

```
worker:
  replicaCount: 2

server:
  replicaCount: 2
```

Furthermore, you may want to implement a primary-replica setup for the database (e.g., PostgreSQL) used by Airbyte. The primary database handles write operations, while replicas handle read operations, ensuring data availability even if the primary fails.

Disaster Recovery (DR) Regions

For business-critical applications of Airbyte, you may want to configure a Disaster Recovery (DR) cluster for Airbyte. We do not support assisting customers with DR deployments at this time. However, we offer a few high level suggestions:

1. Airbyte strongly recommends configuring an external database, external log storage and external connector secret management.
2. Airbyte strongly recommends that your DR cluster is also an instance of Self-Managed Enterprise, kept at the same version as your prod instance.

DEBUG Logs

We recommend turning off `DEBUG` logs for any non-testing use of Self-Managed Airbyte. Failing to do while running at-scale syncs may result in the `server` pod being overloaded, preventing most of the deployment for operating as normal.

Schema Discovery Timeouts

While configuring a database source connector with hundreds to thousands of tables, each with many columns, the one-time `discover` mechanism - by which we discover the topology of your source - may run for a long time and exceed Airbyte's timeout duration. Should this be the case, you may increase Airbyte's timeout limit as follows:

```
server:
  httpIdleTimeout: 20m
  extraEnv:
    - name: READ_TIMEOUT
      value: 30m
```

Update service account for 1.6

Airbyte version 1.6 introduced a breaking change for service account permissions. If you're a Self-Managed Enterprise customer upgrading from 1.5.1 or earlier to 1.6 or later, follow the directions in this article before you upgrade to 1.6. If you're a Core user, this information isn't relevant to you. Learn more about service accounts.

Upgrading without updating service account permissions

If you try to upgrade from 1.5.1 or earlier to 1.6 or later without updating your service account permissions, the Bootloader pod fails and you see the following error in the logs.

```
2025-04-08 21:40:25,960 [main]  INFO    i.a.b.Bootloader(load):73 - Initializing
↪  auth secrets...
2025-04-08 21:40:26,880 [main]  ERROR   i.a.b.ApplicationKt(main):28 - Unable to
↪  bootstrap Airbyte environment.
java.lang.IllegalStateException: Upgrade to version
↪  AirbyteVersion{version='1.6.0-alpha-35dc75a5941', major='1', minor='6',
↪  patch='0'} failed. As of version 1.6 of the Airbyte Platform, we require
↪  your Service Account permissions to include access to the "secrets"
↪  resource. To learn more, please visit our documentation page at
↪  https://docs.airbyte.com/enterprise-setup/upgrade-service-account.
        at io.airbyte.bootloader.AuthKubernetesSecretInitializer.checkAccessToSe
        ↪  crets(AuthKubernetesSecretInitializer.kt:57)
        at io.airbyte.bootloader.Bootloader.load(Bootloader.kt:74)
        at io.airbyte.bootloader.ApplicationKt.main(Application.kt:25)
```

Airbyte doesn't begin the upgrade process, and you can continue using your old version until you're ready to update permissions. Until you do, the Bootloader pod shows that it's in a failed state, but this doesn't prevent Airbyte from working.

Update permissions and begin the upgrade

The process you follow depends on whether you're using Airbyte's default service account or your own.

Which one you're using

If you've defined a service account in the `values.yaml` file you use to deploy Airbyte, you're using your own.

```
global:
  serviceAccount:
```

If you haven't, you're using Airbyte's default service account, `airbyte-sa`.

Using Airbyte's default service account

1. Run the following kubectl command.

   ```
   kubectl -n <namespace> patch role airbyte-admin-role --type='json'
   ↪  -p='[{"op": "replace", "path": "/rules/0/resources", "value": ["jobs",
   ↪  "pods", "pods/log", "pods/exec", "pods/attach", "secrets"]}]'
   ```

2. Upgrade Airbyte as you normally would.

Using your own service account

1. Upgrade your service account/role to grant it **secrets** access. The role should look approximately like this:

   ```
   apiVersion: rbac.authorization.k8s.io/v1
   kind: Role
   metadata:
     name: roleName    app.kubernetes.io/managed-by: Helm
     annotations:
       helm.sh/hook: pre-install
       helm.sh/hook-weight: "-5"
   rules:
     - apiGroups: ["*"]
     //highlight-next-line
       resources: ["jobs", "pods", "pods/log", "pods/exec", "pods/attach",
       ↪  "secrets"]
       verbs: ["get", "list", "watch", "create", "update", "patch", "delete"]
   ```

2. Upgrade Airbyte as you normally would.

Existing Instance Upgrades

This page supplements the Self-Managed Enterprise implementation guide. It highlights the steps to take if you are currently using Airbyte Core, our free open source offering, and are ready to upgrade to Airbyte Self-Managed Enterprise.

A valid license key is required to get started with Airbyte Enterprise. Talk to sales to receive your license key.

These instructions are for you if:

- You want your Self-Managed Enterprise instance to inherit state from your existing deployment.
- You are currently deploying Airbyte on Kubernetes.
- You are comfortable with an in-place upgrade. This guide does not dual-write to a new Airbyte deployment.

Step 1: Update Airbyte Open Source

You must first update to the latest Open Source Community release. We assume you are running the following steps from the root of the `airbytehq/airbyte-platform` cloned repo.

1. Determine your current helm release name by running `helm list`. This will now be referred to as `[RELEASE_NAME]` for the rest of this guide.
2. Upgrade to the latest Open Source Community release. The output will now be referred to as `[RELEASE_VERSION]` for the rest of this guide:

```
helm upgrade [RELEASE_NAME] airbyte/airbyte
```

Step 2: Configure Self-Managed Enterprise

Update your `values.yaml` file as explained in the Self-Managed Enterprise implementation guide. Avoid making any changes to your external database or log storage configuration at this time.

Step 3: Deploy Self-Managed Enterprise

1. You can now run the following command to upgrade your instance to Self-Managed Enterprise. If you previously included additional `values` files on your existing deployment, be sure to add these here as well:

```
helm upgrade airbyte airbyte/airbyte \
  --namespace airbyte \           # Target Kubernetes namespace
```

```
  --values ./values.yaml \       # Custom configuration values
  --version 2.x.x                # Helm chart version to use
```

2. Once this is complete, you need to upgrade your ingress to include the new `/auth` path. The following is a skimmed down definition of an ingress resource you could use for Self-Managed Enterprise:

Configuring your Airbyte ingress

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: # ingress name, example: enterprise-demo
  annotations:
    nginx.ingress.kubernetes.io/ssl-redirect: "false"
spec:
  ingressClassName: nginx
  rules:
    - host: airbyte.example.com # replace with your host
      http:
        paths:
          - backend:
              service:
                # format is ${RELEASE_NAME}-airbyte-connector-builder-server-svc
                name: airbyte-enterprise-airbyte-connector-builder-server-svc
                port:
                  number: 80 # service port, example: 8080
            path: /api/v1/connector_builder/
            pathType: Prefix
          - backend:
              service:
                # format is ${RELEASE_NAME}-airbyte-server-svc
                name: airbyte-enterprise-airbyte-server-svc
                port:
                  number: 8001 # service port, example: 8080
            path: /
            pathType: Prefix
```

All set! When you log in, you should expect all connections, sources and destinations to be present, and configured as prior.

Upgrade to Helm chart V2 (Self-Managed Enterprise)

Do not upgrade to Helm chart V2 or any Airbyte version that requires it. Speak to your Airbyte representative about switching to one of Airbyte's Cloud plans, which always have the latest capabilities and security improvements. Private Link is available if you require it.

Configuring API Access

The Airbyte API enables you to programmatically interact with Airbyte: create sources, destinations, run syncs, list workspaces, and much more.

Access to the API in Self-Managed Enterprise deployments is controlled via application keys. Applications keys are tied to individual users and their respective permissions. Prior to configuring API access, ensure you have an up and running deployment of Airbyte Self-Managed Enterprise that exposes the `airbyte-server` service. To do this, follow the steps in the implementation guide.

Step 1: Create an Application

1. In the navigation bar, click your user name > **User settings** > **Applications**.
2. Click **Create an application**.
3. Give you application a descriptive name.
4. Copy your `client_id` and `client_secret` credentials. You can exchange them anytime to get an access token to make requests to the API. These credentials don't expire, but you can delete them at any time.

Step 2: Obtain an Access Token

With your `client_id` and `client_secret` in hand, make the following API request, replacing `<YOUR_WEBAPP_URL>` with the URL you use to access the Airbyte UI:

```
POST <YOUR_WEBAPP_URL>/api/v1/applications/token
```

Ensure the following JSON Body is attached to the request, populated with your `client_id` and `client_secret`:

```
{ "client_id": "", "client_secret": "" }
```

The API response should provide an `access_token` which is a Bearer Token valid for 60 minutes that can be used to make requests to the API. Once your `access_token` expires, you may make a new request to the `applications/token` endpoint to get a new token.

Step 3: Operate Airbyte via API

You may now make requests to any endpoint documented in our Airbyte API Reference. For example, you may use the List workspaces endpoint to verify the list of workspaces in your organization.

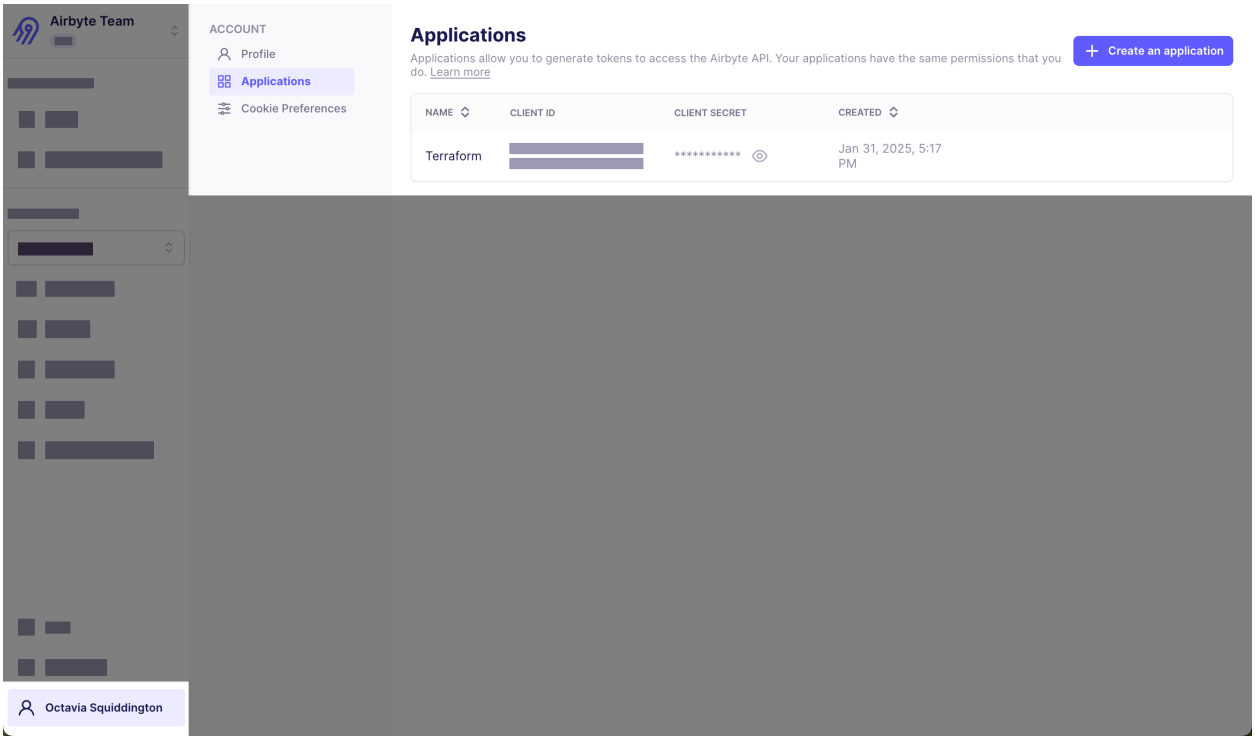

Figure 5: Create an Application

Ensure to include your `access_token` as a `Bearer Token` in your request.:

```
GET <YOUR_WEBAPP_URL>/api/public/v1/workspaces
```

Expect a response like the following:

```
{
  "data": [
    {
      "workspaceId": "b5367aab-9d68-4fea-800f-0000000000",
      "name": "Finance Team",
      "dataResidency": "auto"
    },
    {
      "workspaceId": "b5367aab-9d68-4fea-800f-0000000001",
      "name": "Analytics Team",
      "dataResidency": "auto"
    }
  ]
}
```

To go further, you may use our Python and Java SDKs to make API requests directly in code, or our Terraform Provider (which uses the Airbyte API) to declare your Airbyte configuration as infrastructure.