PORTAINER.iO

PORTAINER TECHNICAL ARCHITECTURE

This document provides a tested model of the Portainer stack and a detailed design of each component within the architecture.

Mar 2026

1. Purpose and Intended Audience

The Portainer Technical Architecture Overview contains a validated model of the Portainer Stack and provides a detailed design of each Portainer component of the whole stack.

This document is intended for technical audiences; architects, administrators, and engineers, including OT engineers, automation specialists, and IT/OT convergence teams managing containerized applications at the industrial edge. It provides an understanding of the common architectural scenarios across different platforms, including a detailed design of each component of the Portainer stack.

2. Architecture Overview

Portainer is a centralized container management platform that accelerates container adoption, reduces operational complexity and addresses the challenges of running containers in Docker, Docker Swarm, Podman, and Kubernetes. It helps you get started and accelerate intelligent adoption with an innovative, centralized management interface.

Portainer has two main components to understand; Server Core and Agent:

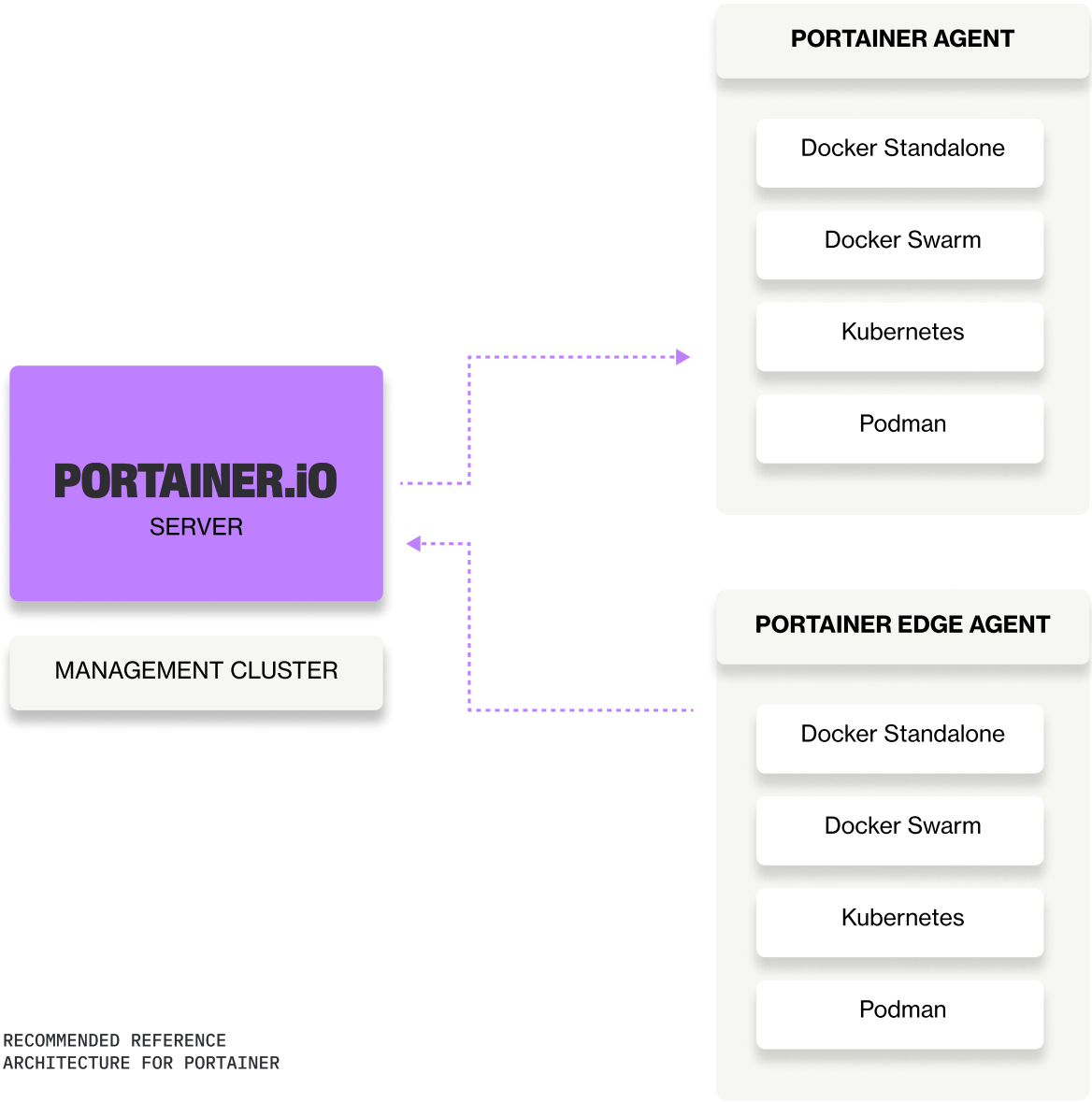
- The Server Core was designed to serve Portainer administrators and users to interact with the web application (UI and API) to set up, manage, and consume the underlying container orchestration platforms.

- The Agent was designed for users running multiple orchestration platforms to avoid running various Portainer Server core instances. Instead, run an agent in each container orchestration platform, and manage them from a single Portainer server core instance:

    - The Agent comes in two modes; Edge Agent and classic Agent. The Edge Agent mode is our recommendation for most scenarios, with the classic Agent mode available as a legacy option.
        - With the Edge Agent, the remote environments access the Portainer Server, i.e. Agents → Portainer. This communication is performed over an encrypted TLS tunnel. This is particularly important in Internet-connected configurations where there is no desire to expose the Portainer Agent to the Internet.
        - In classic Agent deployments, unlike the Edge Agent mode, the central Portainer Server accesses the environments, i.e. Portainer → Agents. As such, any environments it manages are assumed to be on the same network as the Portainer Server so it can securely communicate with Portainer Agents.

The Portainer server core and Agent run as lightweight containers on any containerized infrastructure. Ideally, the Agent will be deployed in each container orchestration and managed by the centralized Portainer Server container.

For the overall architecture design, the highly recommended approach for hosting Portainer in your environment is to run the Portainer server core in a dedicated environment, a management cluster outside your workloads clusters and manage them via Agents. Refer to the diagram below:



RECOMMENDED REFERENCE
ARCHITECTURE FOR PORTAINER

The benefits include but are not limited to the following:

- It becomes easier to protect your management cluster by applying hardened security rules to where Portainer is hosted.
- Portainer runs outside your workload clusters; hence a workload cluster outage does not impact the availability of the Portainer instance, and vice versa.
- Scaling in and out of the number of container runtime environments gets easier.
- Troubleshooting and maintenance get easier. For instance, apply the same RBAC rules to users and teams in various environments from a single Portainer instance.

With these details in mind, there are two primary solutions provided by Portainer; Edge Agent and classic Agent. It is absolutely fine to run a single Portainer instance to serve both; however, separating those two use cases is highly recommended.

2.1. Edge Agent

The Portainer Edge Agent is the solution recommended when deploying and configuring Portainer in most instances, both in datacenter environments and in edge computing / IoT / IIoT scenarios.

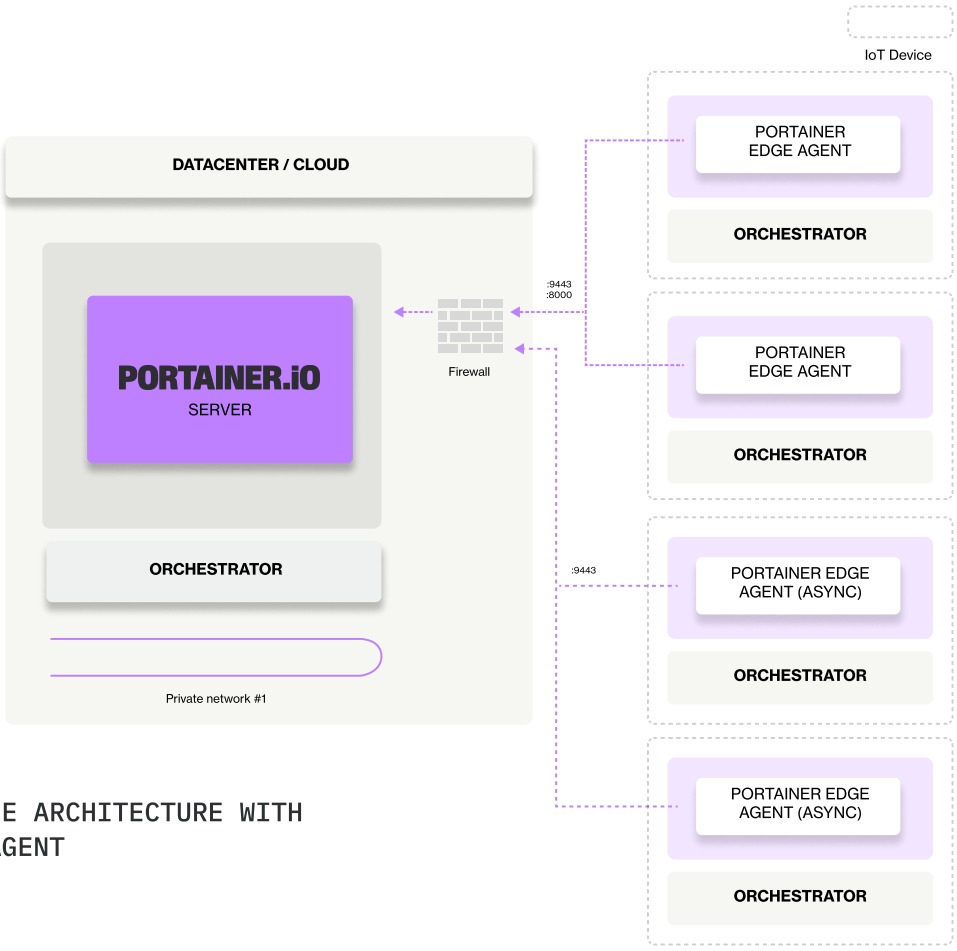
There are two types of Edge Agents for Portainer; **Standard** and **Async**:

- The **Standard mode** was designed for edge devices that must be managed in real-time by the Portainer Server, where it provides the ability to connect to the remote Edge Agent through a tunnel that is established on-demand from the Edge Agent to the Portainer Server
- The **Async mode** has been developed to use very small amounts of data and, as such, is suitable for environments with limited or intermittent connectivity and connections with limited data caps, such as a satellite network. To reduce data usage, the use of the tunnel is not available. Instead, Portainer allows users to browse snapshots of the remote environment, allowing users to see the state of the Edge Agent's environment based on a recent state capture sent to the Portainer Server.

For both types, the Edge Agents establish the connection to the Portainer Server. To facilitate this:

- A firewall will be leveraged to publicly host Portainer's HTTPS API (port 9443) and HTTP WebSocket Tunnel (port 8000) endpoints.
- The Async Edge agent type requires only the HTTPS API endpoint (port 9443).

The following diagram depicts how Portainer and Edge Agents should be deployed in your environment:

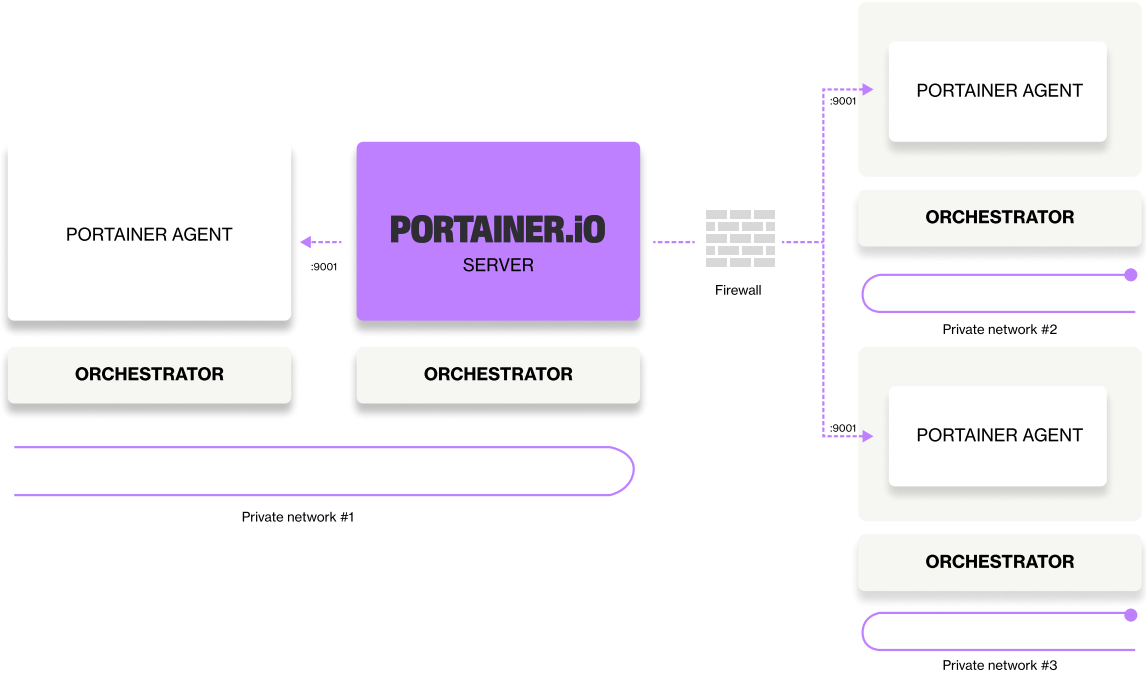


EXAMPLE ARCHITECTURE WITH EDGE AGENT

2.2. Classic Agent

The classic Portainer Agent is our original solution for managing environments, and works similarly to the standard Edge Agent with the major difference being in how the Agent connects to the Portainer Server core. When deploying the classic Agent, the Portainer Server core is the system that establishes the connection, and must be able to connect to the classic Agents on port 9001 (the port is customizable if required).

The following diagram depicts how Portainer and classic Agents would be deployed in your datacenter environment:



> EXAMPLE ARCHITECTURE WITH CLASSIC AGENT

In general, we recommend the use of the Edge Agent over the classic Agent in most scenarios.

3. Portainer Components

The Portainer Components section will dive into more technical details of the following components:

- How Portainer Works in Detail.
- How Portainer Works with Agents (Edge vs classic) in Detail.
- How Portainer Works with Swarm in Detail.
- How Portainer Works with Kubernetes in Detail.

3.1. How Portainer Works in Detail

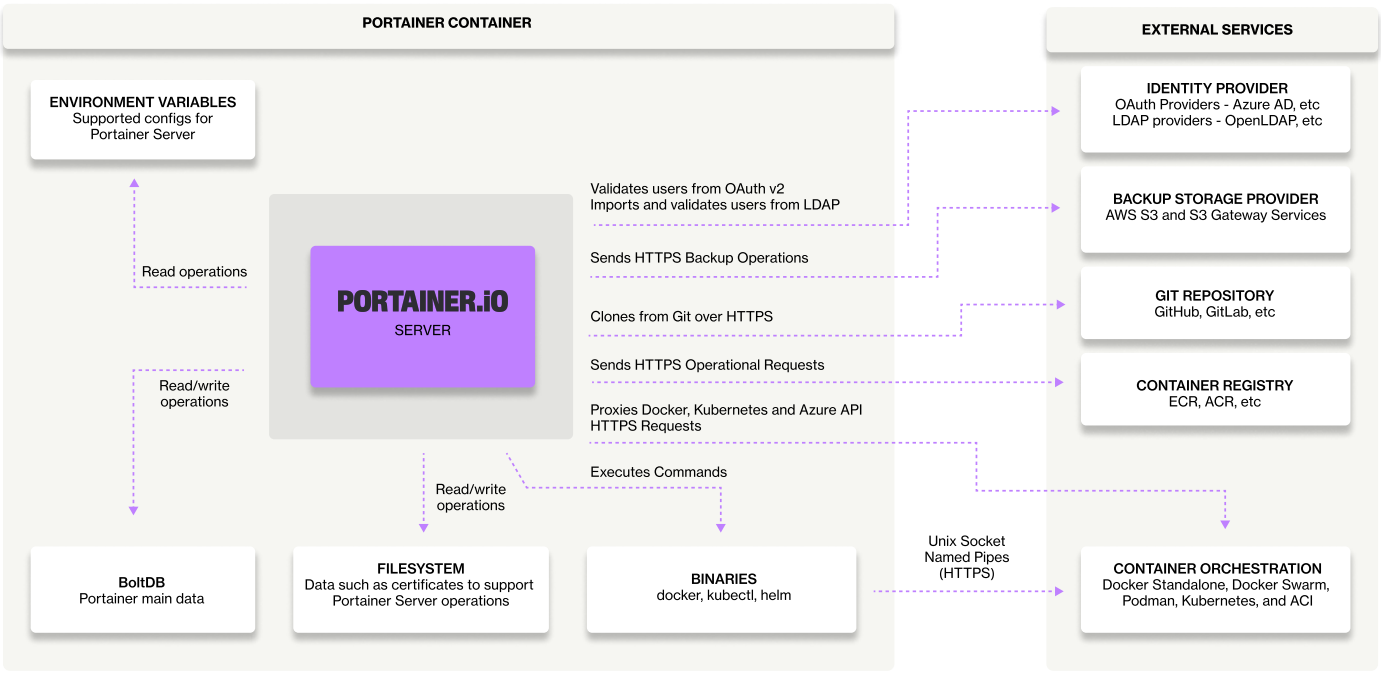
The Portainer server core was built as a lightweight container image to run on standard OCI-compatible container runtime environments. It is comprised of three main components:

- A Single Page Application (SPA) based Web Application with the following to serve the front-end UI experience:
  - AngularJS and React:
    - In the process of migrating AngularJS to React.
  - A set of assets (CSS, HTML and JavaScript).
- Core API is written in Golang to provide REST APIs for backend operations.
- Two BoltDB (a lightweight embedded key/value store) instances:
  - Main Portainer data:
    - Portainer Settings, Backup, Tunnel Server details, and Version.
    - Registries; Docker Hub, AWS ECR, Azure ACR, and so on.
    - Users, Teams, and Team Memberships.
    - Edge Groups, Stacks, and Jobs.
    - Environment details, Groups, Tags, and Relations.
    - Application Stacks and Webhooks.
    - Custom Templates.
  - Audit and Activity logs.

Portainer also interacts with other external services to support managing container runtime environments:

- Microsoft Active Directory, LDAP and OAuth providers; Azure AD, GitHub, Google... etc.
- Backup Storage Provider; AWS S3 bucket or an S3 gateway.
- Git Repository, such as GitHub or GitLab.
- Container Registry; DockerHub, ACR, ECR, Quay, ProGet, GitLab, GitHub... etc.
- The following binaries are used to establish connections to the underlying container orchestrations over proxy:
  - docker Client to execute Docker APIs not supported by the SDK. For instance, Docker Stack management.
  - kubectl and helm to support interacting with Kubernetes clusters.
  - eksctl and awsAuth to bootstrap an EKS cluster in AWS, part of Portainer's deprecated Kubernetes as a Service (KaaS) feature.

The following diagram depicts the components described above:



> HOW PORTAINER WORKS IN DETAIL

3.2. How Portainer Works with Agents

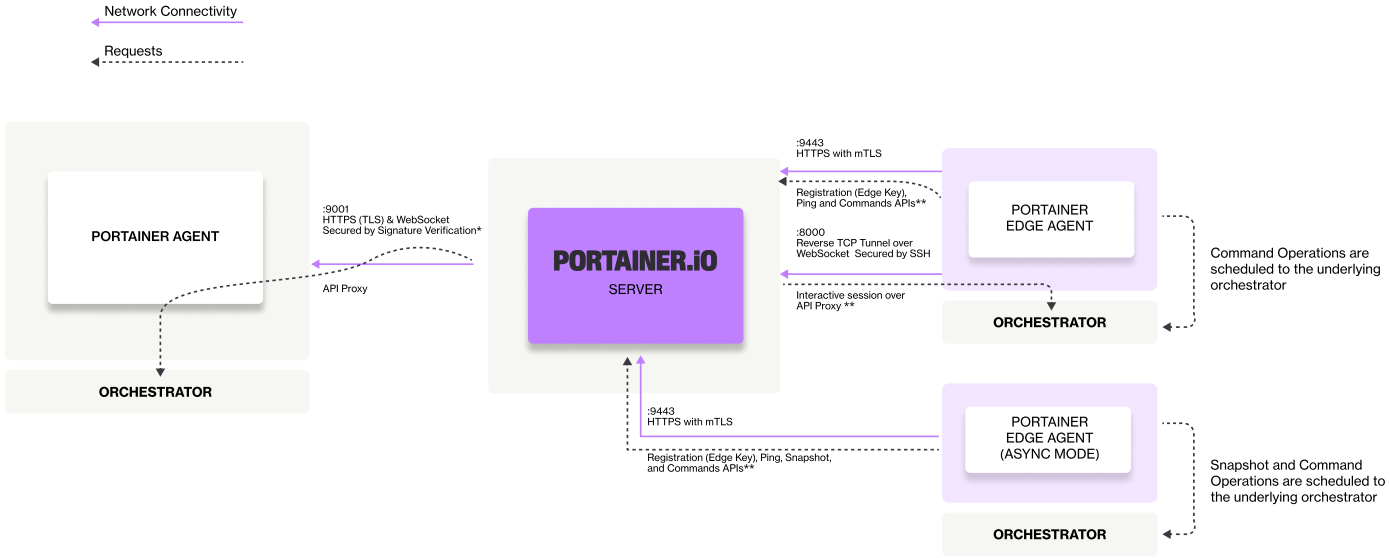
3.2.1. Overview

As reviewed in the earlier section, the Portainer Agent works in two modes; Edge Agent and classic Agent.

This section will dive deep into how Portainer works with Agent in different modes, including their core behaviors:

- Portainer interacts with the underlying container runtime where Agent runs as an API proxy:
  - Standalone/Podman: for instance, to list containers on a Docker Engine, Portainer will trigger a native Docker GET API call against /containers/json.
  - Swarm: different to Standalone that it executes aggregation, manager, and worker requests. Refer to the How Portainer Works with Agent on Swarm section for more details.
  - Kubernetes: the difference from the Swarm is that Kubernetes API Server handles all the operations requiring aggregation, manager, and worker requests. For instance, the API call /api/v1/pods is triggered in listing pods from all Kubernetes nodes without requiring aggregation requests like Swarm. Refer to the How Portainer Works with Agent on Kubernetes section for more details.
- The communication between Portainer and Agent for both types is protected by HTTPS:
  - For Edge Agents:
    - The Edge Agent connects back to the Portainer instance over HTTPS.
    - Once the header X-PortainerAgent-EdgeID is presented to Portainer, and the device with the identical EDGE ID is trusted by the admin, the ongoing communications will work. This means that there is no authentication in place. Hence, Portainer strongly recommends enabling mTLS to secure connectivity in production usage, especially in the IoT / IIoT contexts.
  - For classic Agents, the Agent generates a certificate for exposing its API server (the use of a custom SSL certificate is not supported):
    - Interactive API calls such as docker exec or docker logs are achieved using a protocol called WebSocket. Portainer uses a technique called Signature Verification to secure these communications.
- Establishing an interactive session for the standard edge mode differs from the standard agent. It instead creates a reverse SSH tunnel over WebSocket on port 8000:
  - Due to its nature, the edge async mode has no interactive session capability. For instance, IoT devices in the wild with limited internet access over the satellite. In this case, the Portainer Server schedules command operations based on the command frequency to reduce network usage as much as possible. For example, when a user deploys an Edge Stack to an Edge Group with Async devices, if the Command frequency is set to 60 minutes, the Agent will only send the command operation to the Edge Group when it hits 60 minutes. The default is set to 1 minute.
- For both Standard and Async Edge modes, there are additional operations Portainer triggers; Ping, Snapshot, and Commands. For the Standard mode, a Snapshot is taken after closing the interactive session, where the two's schedules are set by the Poll frequency. Whereas the ping, snapshot, and command frequencies can be set separately for the Async mode:
  - Edge Stacks and Edge Jobs work the same for Standard and Async Edge agents based on the command frequency set by Portainer.
- The agent contains binaries such as docker and kubectl to support certain API operations that cannot be achieved natively using the SDKs.

The following diagram depicts the components described previously:



HOW PORTAINER WORKS WITH AGENTS

3.2.2. Edge Agent

3.2.2.1. Registration

To start an agent in Edge mode, the EDGE=1 environment variable must be set.

Upon startup, the agent will try to retrieve an existing Edge key in the following order:

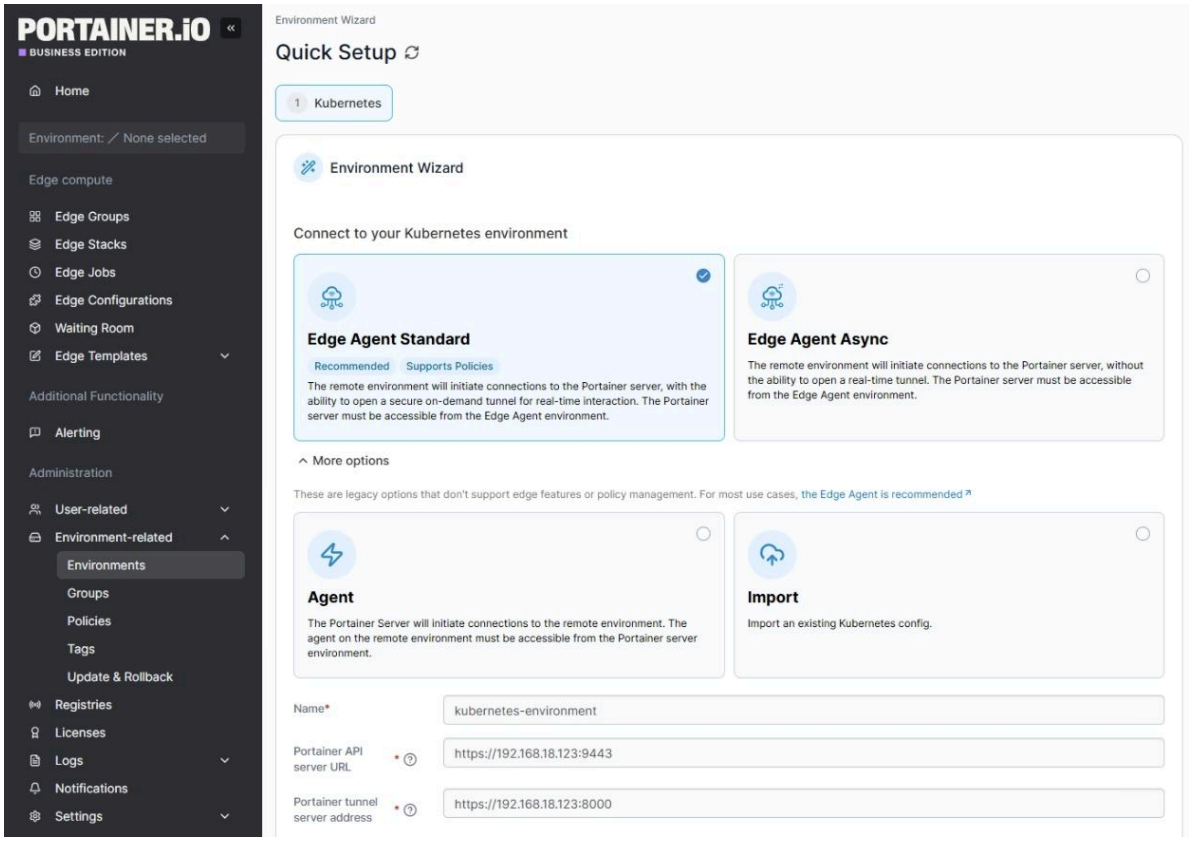
- From the environment variables via the EDGE_KEY environment variable.
- From the filesystem (see the Edge key section below for more information about key persistence on disk).
- From the cluster (if joining an existing Edge agent cluster).

If there is no Edge key provided, the agent will start an HTTP server exposing a UI for users to to associate an Edge key. The UI server will shut down after a valid key is provided.

For security reasons, identical to the Portainer server, the Edge server UI will shut down after 15 minutes if no key has been specified. The agent will require a restart to access the Edge UI again.

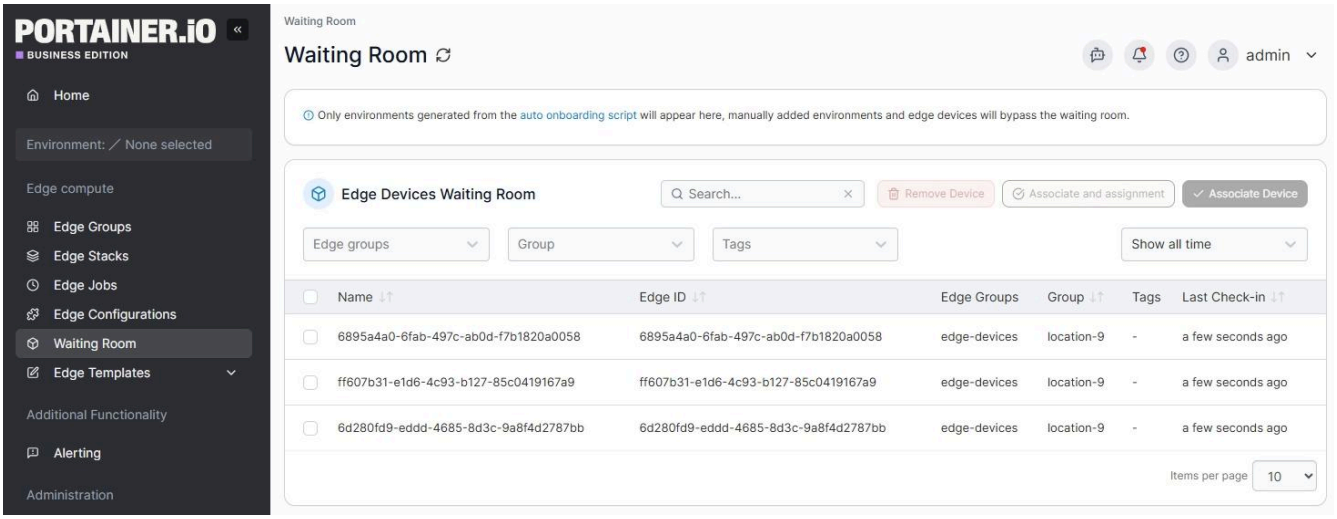
The final process is to onboard it to Portainer, and there are two scenarios:

- Manual environment creation:
- This is a method to on-board edge agents one-by-one. Creation of an environment will show the steps on how to deploy the edge agent as a container based on the container runtime type:



Auto-onboarding:

- In this case, the device will be in the waiting room for the administrator to mark it as trusted, and the on-boarding will be finished. This is the feature to be considered for on-boarding a large amount of edge devices:



3.2.2.2. Edge key

The agent uses the Edge key to connect to a specific Portainer instance. It is encoded using base64 format (without the padding characters) and contains the following information:

- Portainer instance API URL.
- Portainer instance tunnel server address.
- Portainer instance tunnel server fingerprint.
- Endpoint (Edge Device) identifier.

This information is represented in the following format before encoding (single string using the | character as a separator):

```
portainer_instance_url|tunnel_server_addr|tunnel_server_fingerprint|endpoint_ID
```

The Edge key associated to an agent will be persisted on disk after association under /data/agent_edge_key.

3.2.2.3. Reverse tunnel

The reverse tunnel is established by the Edge Agent in the Standard mode only. The permissions associated to the credentials are set on the Portainer instance, where the credentials are valid for a management session, and can only be used to create a reverse tunnel on a specific port (the one that is specified in the poll response).

The agent will monitor the usage of the tunnel. The tunnel will be closed in any of the following cases:

1. The status of the tunnel specified in the poll response is equal to IDLE.
2. If no activity has been registered on the tunnel (no requests executed against the agent API) after a specific amount of time (can be configured via EDGE_INACTIVITY_TIMEOUT, default to 5 minutes).

3.2.2.4. Polling

After associating an Edge key to an Edge Agent, the agent will start polling the associated Portainer instance. Defaults to 5 seconds frequency.

It will use the Portainer instance API URL and the endpoint identifier available in the Edge key to build the poll request URL: http(s)://API_URL/api/endpoints/ENDPOINT_ID/status

The response of the poll request contains the following information:

- Tunnel status.
- Poll frequency.
- Tunnel port.
- Encrypted credentials.
- Schedules.

The tunnel status property can take one of the following values: IDLE, REQUIRED, ACTIVE. When this property is set to REQUIRED, the agent will create a reverse tunnel to the Portainer instance using the port specified in the response as well as the credentials.

Each poll request sent to the Portainer instance contains the X-PortainerAgent-EdgeID header (with the value set to the Edge ID associated to the agent). This is used by the Portainer instance to associate an Edge ID to an endpoint so that an agent won't be able to poll information and join an Edge cluster by re-using an existing key without knowing the Edge ID.

To allow for pre-staged environments, this Edge ID is associated to an endpoint by Portainer after receiving the first poll request from an agent.

3.2.2.5. Snapshot

The following details are captured by a Snapshot in the Edge mode:

- docker info.
- docker containers.
- docker images.
- docker volumes.
- docker networks.
- Snapshot version.

Though, the process of taking a snapshot differs between Standard and Async:

- For the Standard mode, a snapshot is only taken when user interacts with the device:
  - Live connect to a standard edge agent.
  - Disconnect from it after the usage.
  - After 5 mins, the tunnel will become idle.
  - Right before closing the tunnel, a snapshot will be taken.
- For the Async mode, a snapshot is taken based on the frequency set by the Portainer server.

3.2.2.6. Command APIs

Command APIs were specifically designed for Edge Stacks and Edge Jobs. The idea is to send a set of instructions to the Edge Agents to execute these commands. The way Standard and Async mode works for Command APIs is different:

Standard

- The instructions are executed by the Portainer server after establishing the reverse tunnel to the Edge Device on port 8000.

Async

- Only grabs the instructions from the Portainer server based on the command frequency, and execute it locally at the Edge Device level.

3.2.2.7. API server

When deployed in Edge mode, the agent API is not exposed over HTTPS anymore (see Using the agent non Edge section below) because we're using SSH to setup an encrypted tunnel. In order to avoid potential security issues with agent deployment exposing the API port on their host, the agent won't expose the API server under 0.0.0.0. Instead, it will expose the API server on the same IP address that is used to advertise the cluster (usually, the container IP in the overlay network).

This means that only a container deployed in the same overlay network as the agent will be able to query it.

3.2.3. Classic Agent

3.2.3.1. Startup

The classic agent works as a proxy to communicate with the underlying container runtime via Docker API (both Standalone and Swarm) or the Kubernetes API Server (Swarm will talk with other agents running as a global mode deployment).

First, the Agent will start an API server on port 9001 (the port is customizable). The Agent will then determine which container runtime the environment is; Podman, Kubernetes, or Docker Engine. If Docker, it will check if it is running a Swarm mode by checking the node's role (docker node inspect). The agent API server's advertised address will be decided by the Portainer Server's pod or container IP address.

The TLS certificate for exposing its API service gets generated automatically. The use of mTLS is not configurable for the classic Agent.

3.2.3.2. Docker API compliance

When communicating with a Portainer Agent instead of using the Docker API directly, the only difference is the additional X-PortainerAgent-Target header to each request is added to execute actions against a specific node in the Swarm cluster.

The fact that the agent's final proxy target is always the Docker API means that we keep the Docker original response format. The only difference in the response is that the agent will automatically add the Portainer-Agent header to each response using the version of the Portainer agent as a value.

3.2.3.3. Agent specific API

The agent exposes the following endpoints:

- /v2/agents (GET): List all the available agents in the cluster.
- /v2/browse/ls (GET): List the files available under a specific path on the filesystem.
- /v2/browse/get (GET): Retrieve a file available under a specific path on the filesystem.
- /v2/browse/delete (DELETE): Delete an existing file under a specific path on the filesystem.
- /v2/browse/rename (PUT): Rename an existing file under a specific path on the filesystem.
- /v2/browse/put (POST): Upload a file under a specific path on the filesystem.
- /v2/host/info (GET): Get information about the underlying host system.
- /v2/ping (GET): Returns a 204. A public endpoint that does not require any form of authentication.
- /v2/key (GET): Returns the Edge key associated with the agent, only available when the agent is started in the Edge mode.
- /v2/key (POST): Set the Edge key on this agent only available when the agent is started in Edge mode.
- /v2/websocket/attach (GET): WebSocket attach endpoint (for container console usage).
- /v2/websocket/exec (GET): WebSocket exec endpoint (for container console usage).
- /v2/kubernetes (POST): Kubernetes deploy endpoint (equivalent to kubectl apply -f).
- /v2/dockerhub (POST): Docker Hub endpoint for checking the status of its token and rate limits.

Note: The /v2/browse/* endpoints can manage a filesystem. By default, it allows manipulation of files in Docker volumes (available under /var/run/docker/volumes when bind-mounted in the agent container).

3.2.3.4. Agent API version

The agent API version is exposed via the Portainer-Agent-API-Version in each response of the agent. For instance:

```
HTTP/2 200
< content-type: application/json
< portainer-agent: 2.39.0
< portainer-agent-api-version: 2
< portainer-agent-platform: 2
```

3.3. How Portainer Works with Agent on Swarm

For Swarm, deploying agents on each node is mandatory (global mode) where all agents form a cluster using the serf library to communicate each other over an overlay network. Refer to the following requirements for an overlay network:

Component	Requirements
Firewall	The following ports must be available. On some systems, these ports are open by default: - Port 2377 TCP for communication with and between manager nodes - Port 7946 TCP/UDP for overlay network node discovery - Port 4789 UDP (configurable) for overlay network traffic
Network	Docker Swarm's default network MTU is 1500, and if the underlying network has a lower MTU than this, then any containers on the overlay network will fail to communicate with each other. In case of the MTU size is less than 1500, recreate the primary ingress overlay network with MTU specified: docker network rm ingress && docker network create -d overlay --ingress --opt com.docker.network.driver.mtu=1450 ingress
Network	**For VMware** Communication issues over the swarm node routing mesh might occur when running Docker Swarm under VMware. This is due to UDP packets being dropped by the source node. Disabling checksum offloading appears to resolve this issue. Run the following on all the VMs in your cluster: ethtool -K [network] tx-checksum-ip-generic off (Replace [network] with the name of your network adapter)

Since there are multiple agents in the cluster, in order to proxy the requests to the other agents inside the cluster, it introduces a header called X-PortainerAgent-Target which can have the name of any node in the cluster as a value. When this header is specified, the Portainer agent receiving the request will extract its value, retrieve the agent's address located on the node specified using this header value and proxy the request to it. If no header X-PortainerAgent-Target is present, Portainer assumes that the agent receiving the request is the target of the request, and it will be proxied to the local Docker API. Now, this is called Worker requests.

There are requests specifically marked to be executed against a manager node inside the cluster, and this is called Manager requests. The agent will inspect any requests and search for the X-PortainerAgent-ManagerOperation header and if it is found, then the agent will proxy that request to an agent located on any manager node. For instance, /services/, /tasks/, /nodes/**, where all the Docker API operations can only be executed on a Swarm manager node. This means that you can execute these requests against any agent in the cluster, and they will be proxied to an agent (and to the associated Docker API) located on a Swarm manager node.

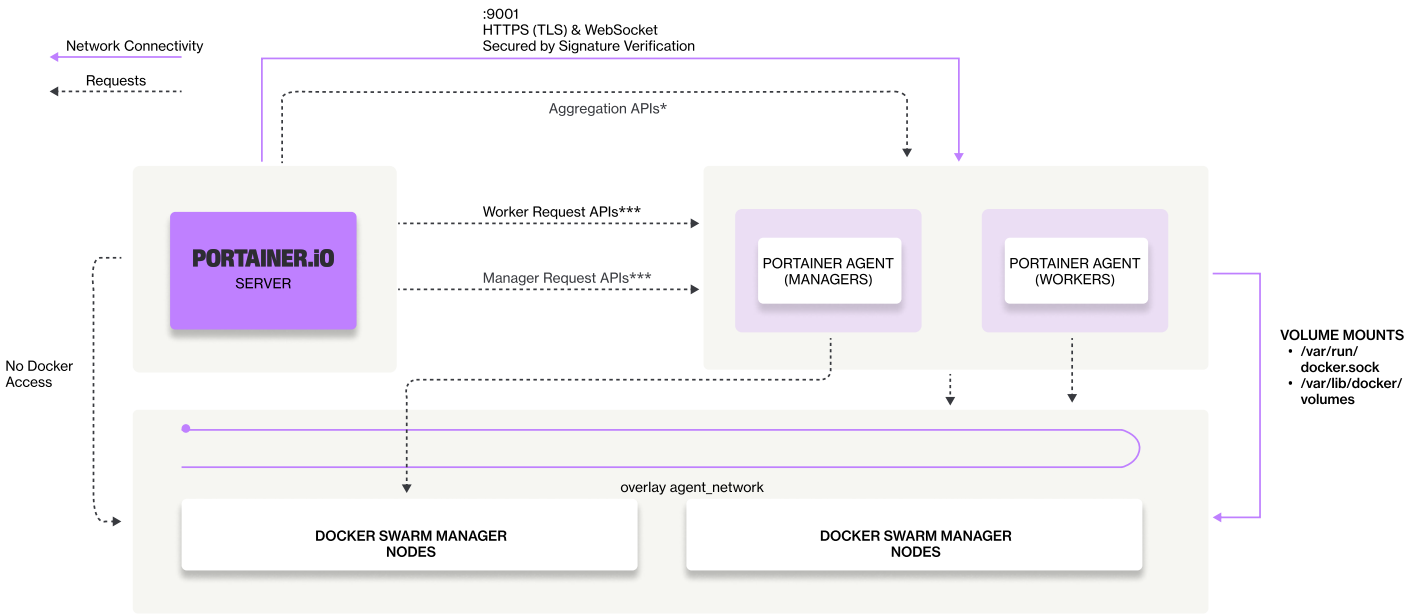
The X-PortainerAgent-ManagerOperation header was introduced to support managing Docker Swarm Stacks using the Docker binary. It MUST always target a manager node when executing any command.

Finally, if no X-PortainerAgent-Target is found, it will automatically execute the request against each node in the cluster in a concurrent way. Behind the scenes, it retrieves the IP address of each node, create a copy of the request, decorate each request with the X-PortainerAgent-Target header and aggregate the response of each request into a single one (reproducing the Docker API response format). This is called Aggregation Requests:

- GET /containers/json
- GET /images/json
- GET /volumes
- GET /networks

The agent handles these requests using the same header mechanism.

The following diagram depicts the components described above:



> HOW PORTAINER WORKS
> WITH SWARM

3.4. How Portainer Works with Agent on Kubernetes

Unlike Docker Swarm, Kubernetes has a control plane object called Kubernetes API Server, which can handle any API operations to manage the underlying Kubernetes cluster. For instance, listing all containers can be handled where aggregation requests are not required. But, in order to execute these API requests, it must go through Kubernetes built-in authentication and authorization process, which is called Kubernetes RBAC.

Due to the nature of Portainer requiring full access to the cluster, as part of the deployment via Kubernetes manifests or Helm, it creates a Service Account called portainer-sa-clusteradmin and bind it with the cluster-admin ClusterRole. This gets used by any administrators within Portainer to manage the underlying cluster. Now, this is called Cluster Requests that allows privileged operations by using the token of this service account to interact with the Kubernetes API server kubernetes.default.svc.

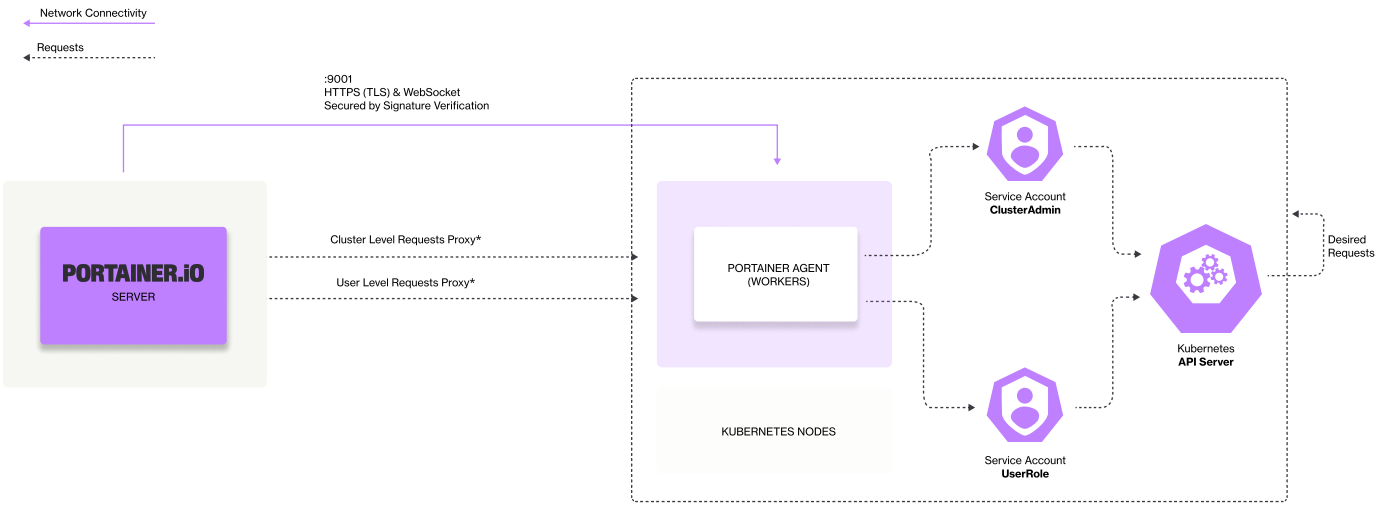
However, for standard users, the process is slightly different. Portainer has the roles pre-built for Kubernetes users; Environment Administrator, Operator, Helpdesk, Standard User, and Read-only User.

For these users to be able to communicate with the Kubernetes cluster, Portainer creates a Kubernetes Service Account, and Roles/ClusterRoles and its RoleBindings/ClusterRoleBindings behind the scenes. Be noted that these objects creations are done only when the user logs in and interacts with the Kubernetes environment. This is to ensure that either the role or ClusterRole is always consistent, i.e. users won't be able to override them manually. For example, for a Read-only user, Portainer creates does the following in order:

1. Creates a ServiceAccount - portainer-sa-user-${UUID}-${Environment_ID}
2. Creates a ClusterRole called portainer-view
3. Bind the ClusterRole called portainer-crb-${UUID}-portainer-view with the ServiceAccount in step 1

Now, this is called User Requests, where the user only has access to the Kubernetes with get and list operations. The details are below:

```
rules:
- apiGroups:
  - "" resources:
  - namespaces
  - nodes
  verbs:
  - list
  - get
- apiGroups:
  - storage.k8s.io
  resources:
  - storageclasses
  verbs:
  - list
- apiGroups:
- metrics.k8s.io
  resources:
  - namespaces
  - pods
  - nodes
  verbs:
  - list
  - get
- apiGroups:
- networking.k8s.io
  resources:
  - ingressclasses
  verbs:
  - list
```



HOW PORTAINER WORKS WITH KUBERNETES

PORTAINER.iO