

**Collaboard Architecture**

KUBERNETES

|  |  |
| --- | --- |
| Title: | Collaboard Architecture on Kubernetes |
| Abstract: |  |
| Status\* | Approved |
| Document Owner: | IBV |
| Document Author: |  |
| \*Draft, In progress, Reviewed, Authorized, Approved, Completed | |

|  |  |  |
| --- | --- | --- |
| **Document Control** | | |
| Document Distribution: | Internal: | All dev Teams |
|  | External: |  |
| Document Authorization: | Authorizer(s): | GP Santopaolo |

|  |  |  |  |
| --- | --- | --- | --- |
| **Change History** | | | |
| Date | Ver | Changed By | Details |
| 09/08/2021 | 1.0 | N. Hermans | Initial draft |
| 10/08/2021 | 1.1 | GP Santopaolo | Review and approval |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

Index

[Introduction 3](#_Toc79479889)

[Purpose of the document 3](#_Toc79479890)

[OpenShift, Kubernetes, and Docker scenario 4](#_Toc79479891)

[Standard Web API request 5](#_Toc79479892)

[Real-time data exchange 5](#_Toc79479893)

[File upload download 5](#_Toc79479894)

[Telemetry data 5](#_Toc79479895)

[Web RTC 6](#_Toc79479896)

[gRPC 6](#_Toc79479897)

[Architecture layer explained 6](#_Toc79479898)

[Kubernetes internal network topology and flows 7](#_Toc79479899)

[Kubernetes infrastructure integration with external network 7](#_Toc79479900)

[Collaboard Ingress configuration details 8](#_Toc79479901)

# Introduction

## Purpose of the document

Collaboard, from an architectural point of view, is a highly flexible application. It can run on the Cloud by using only containers, or when requested, it is able to take native cloud technologies and integrate several PaaS services.

Collaboard can run on the Cloud: AWS EKS, Azure AKS, Google GKE. Or on-premises within the customer's boundaries: Kubernetes, OpenShift, or even Docker-compose (for very small POCs) are all supported scenarios where Collaboard can be installed.

This document will describe the various possible scenarios in different environments

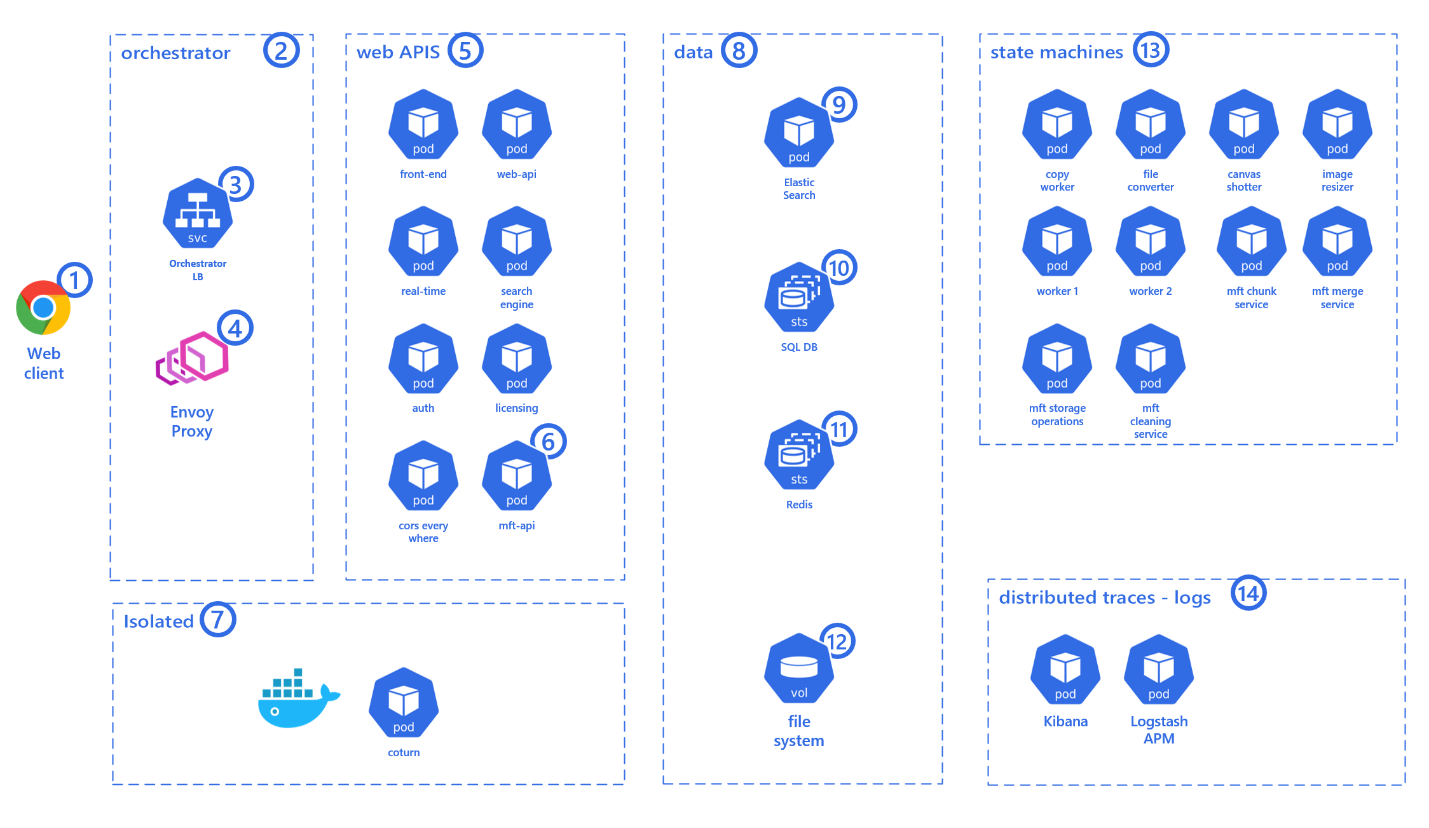
All connections described in this document are secure (https)

# OpenShift, Kubernetes, and Docker scenario

Our infrastructure consists of several layers (Diagram 1):

* The orchestrator (2) and Web APIs (5) layers are the only ones available for the client.
* The data layer (8) is only accessible internally and contains all our repositories
* The state machines (13) are only accessible internally, and they are responsible for running all our worker processes
* The distributed traces (14) are only accessible internally. Optionally the customer can decide if to expose Kibana publically. Kibana is the web UI showing all the traces and logs.
* The isolated layer is publically visible, but it runs outside the architecture. It runs in an isolated docker-compose instance.

The isolation level is also essential to avoid exposing the application to any potential security risk.



*Diagram 1 Collaboard, complete architecture*

When a web client (1) connects to the online whiteboard URL (https://web.collaboard.app on our public environment, it can be any URL), it requests a connection to orchestrator (2), the load balancer (3) together with the proxy (4) decide to which pod (container instance) to route the request.

When a Collaboard client deals with services (5), it can perform several service requests that we can categorize as:

1. Standard Web API request
2. Real-time data exchange (web socket)
3. File upload and download
4. Telemetry data
5. Web RTC
6. gRPC

## Standard Web API request

The client (1) performs a service request to the Web API (5). The business layer processes the request and, data are collected or stored in the data repositories (8). The response is sent back to the client (1).

## Real-time data exchange

When the client (1) runs the web application, it is always connected to a Real-Time service (real-time pod in 5) based on WebSockets. Thanks to this approach, all the clients participating in the same projects can send and receive data from each other.

The same data, when needed and when the business layer decides that is appropriate, are also stored in the database (8) or sent to the state machines (13) for offline processing

## File upload download

The client (1) can upload and download files of any type and size. Collaboard can achieve the same result by using IBV’s MFT, Azure Blob Storage or AWS S3 with CloudFront.

Using native Cloud technology (Blob Storage or S3 with CF), it's effortless to transfer terabytes of data across clients and servers all over the World.

No known technology allows files of any size to be transferred over a network when it comes to on-premises. No matter if it is a small local network or a wide area network. Transfer a file of any given size in a reliable way it is not possible.

That’s why we developed MFT (managed file transfer), our custom technology that allows Collaboard to handle files of any size, no matter how many parallel requests we get.

In diagram 1 above, the application uses our MFT (Managed File Transfer) component to achieve the result of uploading and downloading files of any size from a remote client to the collaboard infrastructure.

## Telemetry data

Each action performed on the client (1) and server are stored on the Elastic Search (7). We have two different Telemetry categories:

1. Application logs, including exceptions
2. Performed actions stored in an anonymous form (e.g., upload action, project creation action, and so on)

If the customer decides to store client (1) telemetry data, then the Logstash APM service (14) needs to be reachable by the client.

If there isn’t a need to store client telemetry data, then the Logstash APM service can be set to be reached internally only.

The same applies when using Application Insight on the *Running on Azure without PaaS Services* or Running on AWS without PaaS services scenarios.

Any Business Intelligence can later analyze these data.

## Web RTC

The client (1) failed to establish a peer-to-peer connection with other clients; it backs up using the TURN and STUN server (coturn) (7).

## gRPC

The client (1) establishes a direct communication channel, unidirectional from the web APIs (5) layer, to receive notifications.

gRRC will also be used as a bidirectional communication channel for server-to-server communication (internal only).

This functionality will be GA (generally available) in a future release.

## Architecture layer explained

The **orchestrator layer** (2) is the entry point of our application for all TCP traffic coming from clients (1) and is analyzed by the orchestrator load balancer (3) and the application proxy (4).

They decide which web APIs (5) will be responsible for serving the particular web request.

The orchestrator LB (3) and proxy (4) are also responsible for scaling up and down all the pods (5-8-13 and 14) based on their internal rules.

When the application runs in the customer's environment, the customer can work with our engineers to determine the best rules for the environment The **web APIs** (5) are responsible for serving the various client requests. They contain part of the business logic of the application. They are responsible for storing data in our data repositories (8), broadcasting messages to clients, and to create queues for the worker services (13).

The **data layer** is where we store **data** (8) in various repositories:

· On SQL Server (10), we store all the information related to canvas, users, and projects

· On Elastic Search, we store all the metadata needed for the client to perform fast and efficient searches

· We use Redis as a backplane for the real-time web socket connections to scale-out. In the future, we will also use it for other features

· We use the file system (12) mounts to store and retrieve files stored in our canvases (projects)

The Application is storing few kinds of information :

**User information**

- Stored in the SQL database.

- Application logs can also contain username (can be email, but not necessarily) mentioned

as an initiator of action.

- Required:

- First and Last name - Used for identification, displayed across board.

- Username (= email) - Unique identification, used for newsletter and other communication.

- Optional:

- Profile picture, company name, industry, user's role in company, company size, phone

number, country, and language.

- These extend the information about user.

- Only profile picture can be visible by other users.

- All personal information are visible to its owner.

- Following information are visible to User Manager role in Backoffice:

- Username, email, first name, last name.

**Project information**

- Project metadata and information are stored in database.

- Documents, pictures, and other attachments are stored in mounted storage.

**Telemetry data**

- Depends on chosen logging solution.

- In case of centralized logging like EFK, logs and telemetry are being sent to elastic search and

stored in mounted storage.

- In cloud solution, logs are being sent to Application Insights, CloudWatch or other cloud

resource.

**Personal data security:**

- All user data are stored in the database. Username is the only information that is also appearing in logs.

- Personal data are uploaded/downloaded via API requests over HTTPS. SSL protocol being used is TLS 1.2 or higher.

- Username/email may also be part of project invitation link and similar. In that case, data are encrypted using asymmetric encryption. The keys for the encryption are stored in the database.

- Database needs to be encrypted and protected with controlled access. Encryption and its parameters depend on selected database technology and its options.

- Network/local storage that is used to store project data also needs to be sufficiently secured. Access to the storage is performed by mounting it to pods. Stored data are NOT encrypted by the application, it relies on external security of selected storage.

# Kubernetes internal network topology and flows

Diagram

Description automatically generated

# Kubernetes infrastructure integration with external network

Diagram

Description automatically generated

# Collaboard Ingress configuration details

Diagram

Description automatically generated