<https://docs.camunda.org/manual/latest/installation/docker/#community-edition>

<https://devapo.io/blog/process-automation/camunda-embedded-vs-camunda-standalone/>

<https://docs.camunda.org/manual/latest/introduction/architecture/>

<https://github.com/camunda/docker-camunda-bpm-platform#database-environment-variables>

Important - <https://blog.bernd-ruecker.com/moving-from-embedded-to-remote-workflow-engines-8472992cc371>

<https://docs.camunda.org/manual/latest/user-guide/camunda-bpm-run/>

<https://github.com/camunda-community-hub/camunda-platform-7-rest-client-java>

<https://github.com/camunda/camunda-bpm-platform/tree/master/spring-boot-starter/starter-client>

docker run -d --name camunda -p 8080:8080 camunda/camunda-bpm-platform: run-7.22.0

docker run --name postgres-db-local01 -p 5432:5432 -e POSTGRES\_PASSWORD=test123 --network bridge-network-demo --rm -v mydata01:/var/lib/postgresql/data postgres:16-alpine3.20

# start postgresql image with database and user configured

docker run -d --name postgresql ...

docker run -d --name camunda -p 8080:8080 --network bridge-network-demo \

-e DB\_DRIVER=org.postgresql.Driver \

-e DB\_URL=jdbc:postgresql:// postgres-db-local01:5432/process-engine \

-e DB\_USERNAME=postgres \

-e DB\_PASSWORD=test123 \

-e WAIT\_FOR= postgres-db-local01:5432 \

camunda/camunda-bpm-platform:run-7.22.0

**1. Centralized (Remote) Camunda Engine**

In a centralized setup, Camunda runs as a standalone service (on a separate server or container), and client applications communicate with it remotely over the network via REST API, Java API, or other protocols.

**Pros of Centralized Camunda Engine:**

* **Separation of Concerns**: The business process logic (BPMN workflows) and application logic are decoupled, which leads to easier maintenance and independent scaling of the process engine.
* **Centralized Monitoring and Management**: You have a single point to manage and monitor all process definitions, deployments, and workflows across multiple applications or microservices. This centralization is especially useful when many systems interact with the same process engine.
* **Multi-Application Support**: Since the process engine is standalone, multiple services or applications can use it without embedding Camunda in each of them. This avoids code duplication and provides consistency in how process definitions are managed across the organization.
* **Scalability**: The Camunda engine can be scaled independently of the client applications. You can horizontally scale the Camunda server and deploy it in a clustered environment to handle more workload.
* **Maintenance and Upgrades**: Since Camunda is running independently, upgrading or patching the process engine doesn't require modifying or redeploying your applications.
* **Resource Efficiency**: Running Camunda separately allows the application server or service to only focus on the application logic without worrying about executing BPMN processes, potentially leading to better resource utilization.

**Cons of Centralized Camunda Engine:**

* **Network Latency**: Remote interactions with the Camunda engine introduce network overhead. REST API calls or remote Java calls are slower compared to in-memory calls in an embedded setup.
* **Higher Complexity**: Setting up and managing a standalone process engine may introduce additional operational complexity (e.g., managing a separate server, dealing with networking, security, etc.).
* **Siloed Process Instances**: If one centralized Camunda engine fails, it can impact all applications depending on it. This could potentially cause single points of failure unless the architecture is resilient with proper clustering or failover mechanisms.

**2. Embedded Camunda Engine**

In an embedded setup, Camunda is included as a library inside the application (e.g., Spring Boot, Java EE). The process engine runs in the same application or microservice where the business logic resides.

**Pros of Embedded Camunda Engine:**

* **Tightly Coupled with Application**: Since the engine is embedded, business processes can be executed in the same JVM as the application. This reduces latency and allows direct access to application resources and objects (e.g., services, DAOs).
* **Simpler for Small Applications**: For smaller or less complex applications, embedding Camunda can be easier to set up and manage. It doesn't require a separate process engine server, simplifying the deployment.
* **In-Memory Execution**: All process execution happens in-memory, making it faster than remote API calls. There’s no need for remote communication when starting or managing processes.
* **No Network Overhead**: Because it’s embedded, there’s no risk of network latency, and process execution is highly performant, suitable for real-time systems or low-latency environments.
* **Simplified Transaction Management**: Transactional consistency between the business logic and the process engine is easier to manage since both run in the same context.

**Cons of Embedded Camunda Engine:**

* **Limited Scalability**: Embedded process engines are limited by the resources of the host application. Scaling the process engine requires scaling the entire application, which can lead to resource contention. It’s also more difficult to scale horizontally across distributed systems.
* **Process Management Overhead**: Each application that embeds Camunda becomes responsible for managing its own process definitions and workflow deployments. If multiple services use BPMN workflows, managing them becomes more complex without a centralized management interface like Cockpit.
* **Code Duplication**: Each application embedding Camunda needs to include the engine and its configurations. This can lead to code duplication, especially in microservice architectures, making maintenance more difficult.
* **Upgrading and Patching**: Any changes or upgrades to the Camunda engine require redeploying the application, as both are tightly coupled.
* **No Centralized Monitoring**: Without a centralized process engine, each application has its own process engine instance. Monitoring and managing workflows across applications can become fragmented unless additional tools are implemented.

**When to Use Centralized Camunda (Remote):**

* **Microservices Architecture**: If you have multiple microservices and need a consistent BPMN process engine across all of them, a centralized engine reduces duplication and centralizes management.
* **Large-Scale Systems**: If you require horizontal scalability and process engines that can handle high loads, a centralized engine in a clustered environment will be easier to scale.
* **Multi-Application Use Cases**: If multiple applications need to share processes or workflows, a centralized engine helps manage those processes consistently across systems.
* **Centralized Monitoring and Administration**: If monitoring process instances centrally is critical (for example, in enterprise environments), having a central engine simplifies this.

**When to Use Embedded Camunda:**

* **Simple or Monolithic Applications**: If your application is monolithic and self-contained, embedding the engine can reduce the complexity of managing a separate server and networking overhead.
* **Low-Latency Requirements**: If you need low-latency, in-memory process execution, embedding Camunda in the same application or microservice can be beneficial.
* **Single-Application Use Case**: If only one application is using Camunda and no other system needs to share the workflows, embedding it simplifies the architecture.
* **Tightly Coupled Business Logic and Processes**: If your business logic and process flows are tightly coupled and you need to keep them within the same codebase for transactional consistency, embedding is easier to manage.

**Conclusion:**

* **Centralized Camunda (Remote)** is better suited for **distributed systems, microservices architectures**, or environments where multiple applications need to share the same processes and workflows, and centralized monitoring is important. It offers **better scalability, maintainability, and separation of concerns**.
* **Embedded Camunda** is ideal for **smaller, monolithic applications** or situations where you need **low-latency, in-memory process execution**, and don’t require the engine to be shared across multiple applications.

Running Camunda as a **standalone engine on OpenShift** with scalability, reliability, and performance in mind requires careful architectural choices and leveraging OpenShift's capabilities. Below are the key approaches and considerations for deploying and managing Camunda on OpenShift in a scalable, reliable, and performant manner.

**1. Containerization of Camunda**

To run Camunda on OpenShift, you’ll first need to package it as a container. OpenShift is Kubernetes-based, so you can use Docker containers to package Camunda along with any other dependencies (like databases or message brokers).

**Key Considerations:**

* **Use Official Camunda Docker Images**: You can use or customize the official Docker images provided by Camunda. For example:
  + camunda/camunda-bpm-platform:tomcat-latest
  + camunda/camunda-bpm-platform:wildfly-latest

Ensure that the images are properly configured with the necessary external resources (e.g., databases, message brokers).

* **Stateful and Stateless Services**: Decide whether you want to run Camunda with a **stateful** or **stateless** configuration. For most enterprise applications, the state is stored in a database, and Camunda itself can be deployed in a stateless mode where it doesn’t store state in memory between requests.

**2. Deployment Strategies for Scalability and Reliability**

OpenShift allows multiple deployment strategies. You can scale and manage Camunda effectively by leveraging OpenShift's features for **clustering, scaling, and high availability**.

**Horizontal Pod Autoscaling (HPA):**

* **Horizontal Scaling**: Set up autoscaling policies to automatically scale the number of Camunda instances (pods) based on resource utilization such as CPU or memory usage. In OpenShift, this is known as **Horizontal Pod Autoscaling** (HPA).

Example:

yaml

Copy code

apiVersion: autoscaling/v1

kind: HorizontalPodAutoscaler

metadata:

name: camunda-hpa

spec:

scaleTargetRef:

apiVersion: apps/v1

kind: Deployment

name: camunda

minReplicas: 2

maxReplicas: 10

targetCPUUtilizationPercentage: 70

This ensures that if the load increases, more Camunda instances (pods) will be automatically launched, and when the load decreases, they will be terminated.

**Rolling Updates:**

* **Rolling Updates for Reliability**: Use rolling updates to ensure **zero downtime deployments** when new versions of the Camunda engine are released or when scaling up/down. This ensures that there's no interruption in service during upgrades or changes to the application.

**Failover and High Availability:**

* **HAProxy or OpenShift Router**: OpenShift uses the OpenShift Router (built on HAProxy) to provide **load balancing** and **failover capabilities**. This ensures that incoming traffic to Camunda is balanced across multiple instances, and if one instance goes down, the traffic is routed to healthy instances.
  + Configure your OpenShift Router to route requests to multiple Camunda instances.
  + Ensure that Camunda is stateless and relies on an external, highly available database to avoid session stickiness.

**3. Persistence and Database Considerations**

The process data and execution state of the Camunda BPM engine are typically stored in a relational database. The choice and configuration of the database play a critical role in scalability, reliability, and performance.

**Database Scaling:**

* **External Database**: Use an external database (such as PostgreSQL, MySQL, or Oracle) that is separate from the Camunda containers. This ensures that Camunda pods can scale independently without impacting the database.
* **Database Clustering**: Use a **clustered database** setup with replication, load balancing, and failover to ensure high availability. PostgreSQL and MySQL both support **replication and clustering** (e.g., Galera for MySQL or Patroni for PostgreSQL).
* **Connection Pooling**: Properly configure the connection pool for database connections within the Camunda engine to ensure efficient usage of database resources under high load.

Example configuration in server.xml (for Tomcat):

xml

Copy code

<Resource name="jdbc/ProcessEngine"

auth="Container"

type="javax.sql.DataSource"

username="camunda"

password="password"

driverClassName="org.postgresql.Driver"

url="jdbc:postgresql://postgres-db:5432/camunda"

maxActive="100"

maxIdle="30"

maxWait="10000"

/>

**Database Backups and Failover:**

* **Automated Backups**: Implement automated backup strategies to regularly back up the database.
* **Active-Passive Failover**: Ensure the database has failover mechanisms such as **active-passive** or **active-active replication** setups for disaster recovery.

**4. Service Discovery and Load Balancing**

OpenShift provides **Service Discovery** using DNS-based mechanisms and **load balancing** using Kubernetes services. Each instance of Camunda can be registered as a service, and OpenShift/Kubernetes can automatically load-balance between them.

**Headless Service:**

* **Headless Services for Stateful Workloads**: If you need to deploy Camunda in a **stateful** configuration (e.g., long-running processes), you might need to use a **headless service** for proper service discovery and persistent connections.

**Load-Balancing Mechanisms:**

* Ensure that OpenShift’s load-balancing service routes traffic efficiently to your Camunda instances. This can be configured to balance traffic based on resource usage or round-robin.

**5. Distributed Execution**

For highly scalable and reliable deployments, it’s important to consider the **distributed execution** of process instances and jobs within Camunda.

**Job Executor Scaling:**

* **External Job Executors**: Use **Camunda's external job executor pattern**, which allows job execution to be offloaded to dedicated workers (using REST or message-based communication like Kafka). This allows process definitions to be lightweight and distributed across multiple nodes.
* **Distributed Job Executors**: Use the distributed job execution mechanism where multiple Camunda instances share the same job pool, and jobs are executed across the nodes in the cluster. This is particularly useful when handling high workloads or scaling out.

**6. Persistent Volumes and ConfigMaps**

* **Persistent Volumes**: If Camunda requires persistent storage for certain operations (e.g., file uploads), ensure that you use **Persistent Volume Claims (PVCs)** in OpenShift to handle persistent data across pod restarts.
* **ConfigMaps and Secrets**: Use OpenShift **ConfigMaps** and **Secrets** to manage configuration settings and sensitive data (e.g., database credentials) securely. This avoids hardcoding environment-specific data into the Docker images.

Example using ConfigMaps:

yaml

Copy code

apiVersion: v1

kind: ConfigMap

metadata:

name: camunda-config

data:

DB\_URL: "jdbc:postgresql://db-server/camunda"

DB\_USERNAME: "camunda"

DB\_PASSWORD: "password"

**7. Monitoring and Logging**

Monitoring the health of Camunda is critical for ensuring performance and reliability.

**Monitoring with Prometheus and Grafana:**

* **Prometheus**: Camunda can expose metrics using **Micrometer** or **Prometheus JMX exporter** to send metrics to **Prometheus**. These metrics can then be visualized in **Grafana** to monitor process engine performance.

Example metrics include job execution times, active process instances, failed jobs, etc.

**Centralized Logging:**

* **ELK Stack or OpenShift Logging**: Ensure that logs from Camunda instances are centrally collected and monitored using OpenShift’s logging stack (Elasticsearch, Fluentd, and Kibana) or using other solutions like the **ELK stack**. This allows you to track issues, errors, and performance bottlenecks in real-time.

**Summary of Key Approaches:**

1. **Containerization**: Use Camunda’s official Docker image and run it in OpenShift as a stateless or stateful service.
2. **Horizontal Pod Autoscaling (HPA)**: Automatically scale Camunda pods based on load using OpenShift’s Horizontal Pod Autoscaler.
3. **Persistent Databases**: Ensure the use of a highly available external database (PostgreSQL/MySQL) with clustering, replication, and failover support.
4. **Distributed Job Execution**: Utilize distributed job execution and external job workers to improve scalability and distribute the workload.
5. **Service Discovery and Load Balancing**: Use OpenShift’s built-in service discovery and load-balancing features to ensure requests are efficiently routed.
6. **Monitoring and Logging**: Integrate with Prometheus and Grafana for monitoring and set up centralized logging using OpenShift’s logging solutions.
7. **Resilience and Failover**: Design the architecture with resilience (rolling updates, replication, and failover mechanisms) to ensure high availability and reliability.