How can I create secrets in self-managed camunda 8?

There is no built-in feature for managing secrets in self-managed Camunda 8. You could use external secret management tools like HashiCorp Vault, AWS Secrets Manager, Azure Key Vault, etc.

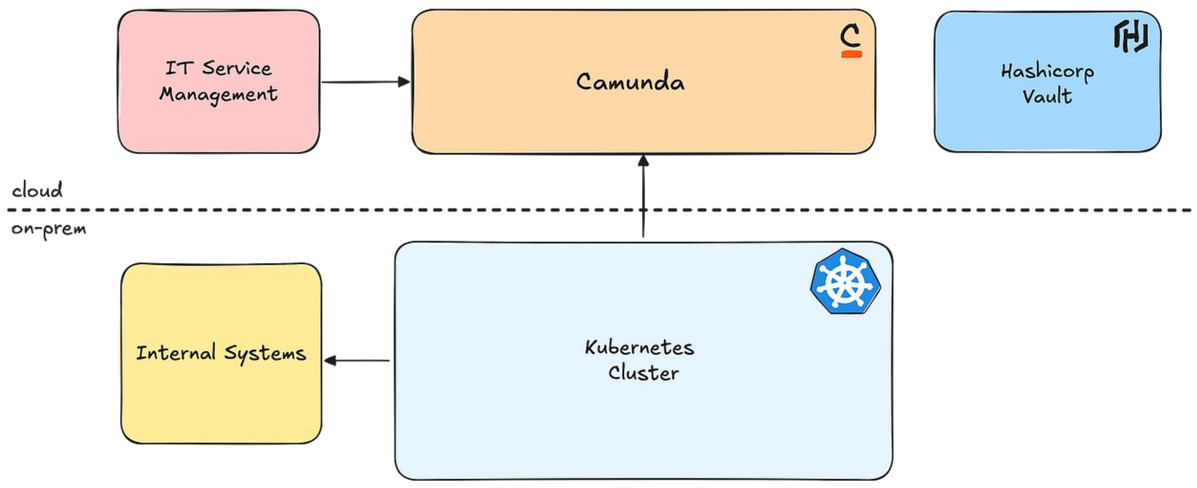
Secure Process Automation at Scale with Camunda and HashiCorp Vault

Automated processes typically connect multiple systems, pass data across service boundaries, and require credentials to access APIs or downstream tools. That makes secret management and secure architecture not just a best practice, but a necessity.

Need to develop a scalable target architecture for the new process platform. This architecture should support multiple JobWorkers and Connectors to integrate systems, as well as a third-party secret provider called HashiCorp Vault. We need to support multiple business units as well, which will all have their own connectors, secrets and processes.

Finding a Good Target Architecture

The initial landscape already has some great cornerstones in place. We're using *Kubernetes* as the runtime for applications, *Camunda* to orchestrate processes, and *Vault* to manage the credentials of the target systems.



There were three big questions we needed to address first:

1. How many runtime applications do we need?
2. How do we get secrets into the runtime(s)?
3. How can we support secret rotation?

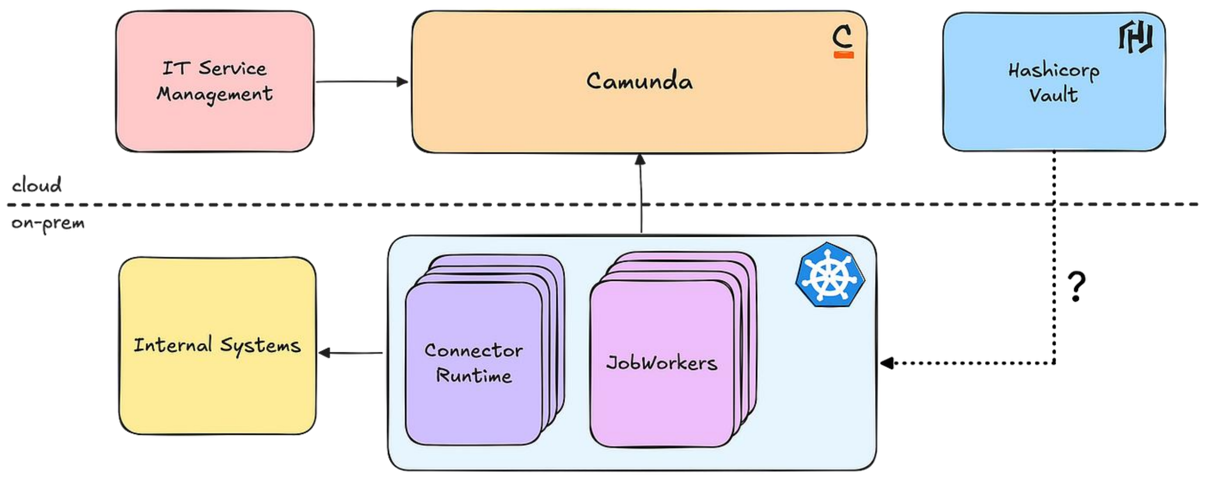
How Many Runtimes?

Initially, we had to make a decision about the number of connector and job worker runtimes. We're going to automate several processes for each of the business units, provide shared and individual connectors as well as job workers, and have some shared processes, but also some individual processes.

Since scalability and fully separation of business units were the most important requirement, we decided to provide one connector runtime and one job worker runtime for **each** business unit. This approach initially resulted in two applications, as we started migrating processes from a single business unit. This would ensure, that units can scale out and prevent secrets being leaked to unintended eyes, since they are not shared in a runtime.

Getting Secrets Out of Vault

Since the runtime count matter is resolved, we can now concentrate on the next phase. **How can we transfer our secrets from Vault to our applications?**



About HashiCorp Vault

HashiCorp is the company behind some of the most widely used infrastructure tools out there. Their focus is on helping teams automate and secure infrastructure in cloud and hybrid environments.

One of their most important tools is Vault. It's a secrets management system that helps you store and control access to sensitive data like passwords, API keys, certificates, and tokens. Vault supports features like audit logging, encryption, access control, and even dynamic secrets, so credentials can expire and rotate automatically.

It’s made for setups where you’re working with a lot of different services and systems, and you want to manage secrets in a secure and centralized way.

Retrieval Phases

There are two main approaches to retrieving secrets: static at application **startup** or dynamically during **runtime**.

Both approaches have their advantages and disadvantages.

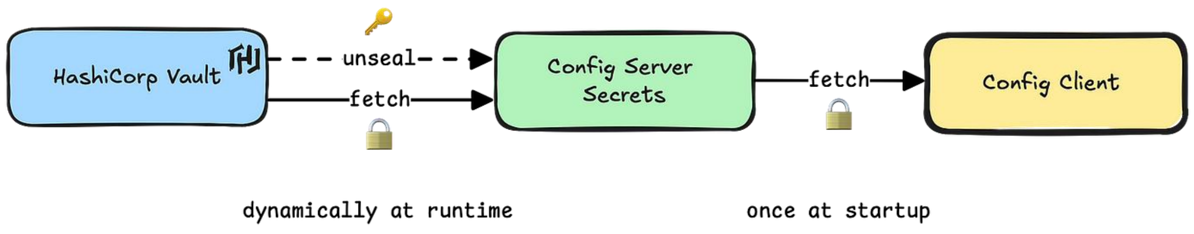
Loading secrets at **startup** is faster, but it makes it more challenging to reload changed secret values. You’d need to restart your application. Additionally, keeping secrets in-memory generally poses some security risks. Someone could obtain a memory dump or a corrupted library could access them.

Loading secrets at **runtime** ensures that your secrets are always up-to-date. However, it adds additional latency and complexity, especially when loading secrets for every execution. Caching can help mitigate this issue, but it also introduces a separate lifecycle for your application that requires management. So be careful when deciding that.

Ultimately, you must choose the approach that best suits your needs.

Our Hybrid Approach

We decided to follow a hybrid mode between both modes. At the customer, we weren’t able to access Vault directly but only had the possibility to read secrets from a central Spring Cloud Config Server instance, that was used to fetch secrets during runtime.



The server would unseal Vault and get the secrets every time a client would ask for them, and seal the Vault again, keeping secrets safe. The secrets are loaded during startup of our runtime, keeping them in-memory during its lifecycle. This means that we have to restart our applications if secrets change. But this is a compromise we’re ready to take on.

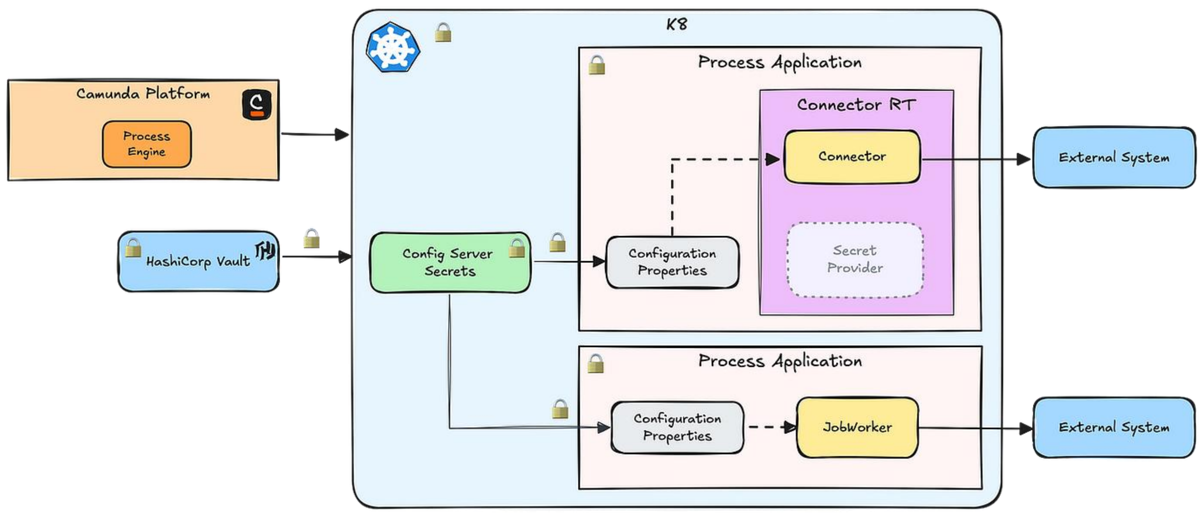
The communication between config server and client is also restricted. We’ll hand in those secrets in a separate phase, before we connect to the server to get the runtime secrets.

Security Aspects

- Communication between Vault and Kubernetes is secured by using *https*.

- The Vault backend encrypts secrets in transit and rest.
- Traffic is restricted from and to the config server.
- Spring Security is used to disable actuator endpoints that might leak configurations.
- Secret related properties are e.g. excluded from *toString()* calls.
- Additionally, we use a log interceptor to mask certain property values, if they contain *password* or *secret* etc. in the property name.

With this, the risk of an incident is quite low, making it a balance between security and pragmatism. Here you can see a schema of our technical design with all the places where we applied some additional security measures.



Technical design of secret management

In the process we also found out, that it is no longer necessary to use a secret provider to get secrets into the connector runtime. We were able to remove *all* secret-related fields from connector templates, making them a bit dumber. The runtime knows all the secrets and can use them to initialize clients, no need for modelers to care about this.

This solution provides sufficient flexibility to manage multiple runtimes while maintaining centralization and utilizing existing solutions in our landscape. However, it also introduces new challenges that necessitate our attention.

Open Issues

We now know the number of runtimes and how we get secrets in them securely. But we also have some issues left.

Given the existence of multiple connector runtimes, we now need a separate connector template for each runtime when sharing a connector over multiple units, even though the connector itself remains unchanged. The duplication of making changes across all templates is very unfortunate.

Furthermore, we aim to ensure that altering secret values does not result in incidents affecting job workers or the connector runtimes. This also has to be addressed.

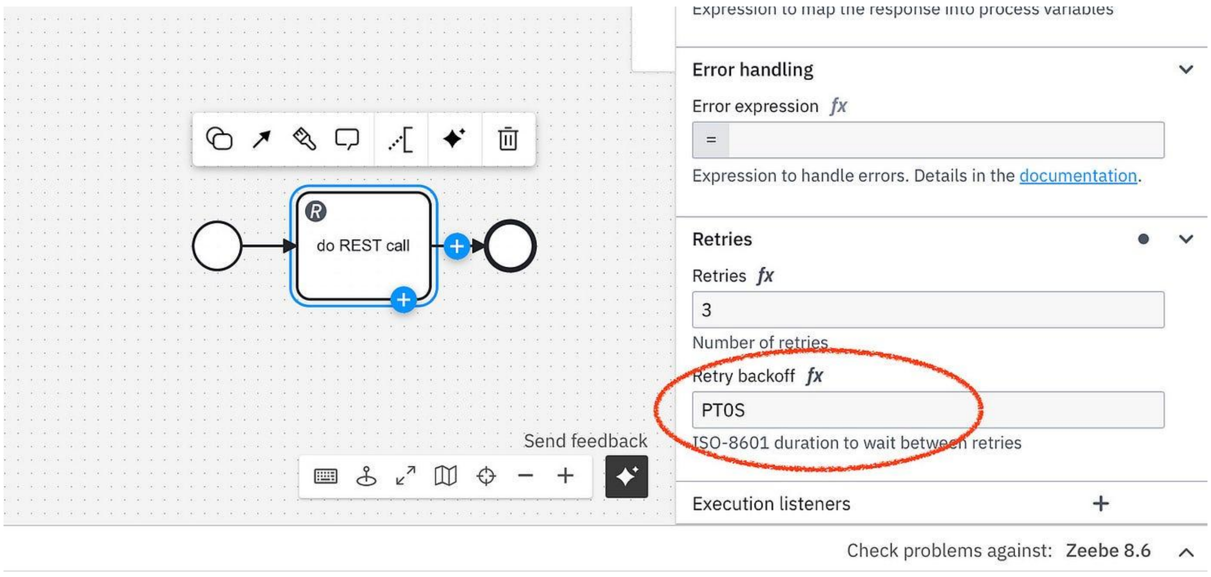
Support of Secret Rotation

The easy one first. Camunda offers several mechanisms to avoid incidents due to changing secret values. Every time a task fails, e.g. due to invalid secrets, the process engine uses a number of retry attempts before an incident is created.

If credentials are invalid, it will repeat the call until it either works, or the amount of retries is exhausted. By default, Camunda doesn't wait between retries, giving us a relatively small timeframe to fix the issue. For that we can use a mechanism called *retry backoff*.

In BPMN

A connector template can have a specific field where you can set a backoff time. The value is sent as a task header to the engine. The engine will wait this amount of time before a new retry is attempted. In the meantime we can e.g. restart our connector runtime to refresh the changed value.



Expression to map the response into process variables

Error handling

Error expression fx

=

Expression to handle errors. Details in the documentation.

Retries

Retries fx

3

Number of retries

Retry backoff fx

PT0S

ISO-8601 duration to wait between retries

Execution listeners

Reetry backoff settings in the web modeler for a connector

In Java Code

We can also use the Java client or Camunda's SDK to handle retry backoff in workers and connectors.

In the Zeebe client, the *FailCommand* has a property that can be set with a custom backoff duration. This leads to a retry backoff for **job workers** giving us enough time to restart the runtime.

```
try {
  authenticate();
  // do worker logic here...
} catch (AuthException e) {
  // fail job with custom retry back-off
  client.newFailCommand(job)
          .retries(job.getRetries() - 1)
          .retryBackoff(Duration.ofSeconds(60)) // wait 60s!
          .errorMessage("Authentication failed!")
          .send()
          .join();
}
```

For **connectors** we can use a *ConnectorRetryExceptionBuilder* from the Java SDK, to do the same thing.

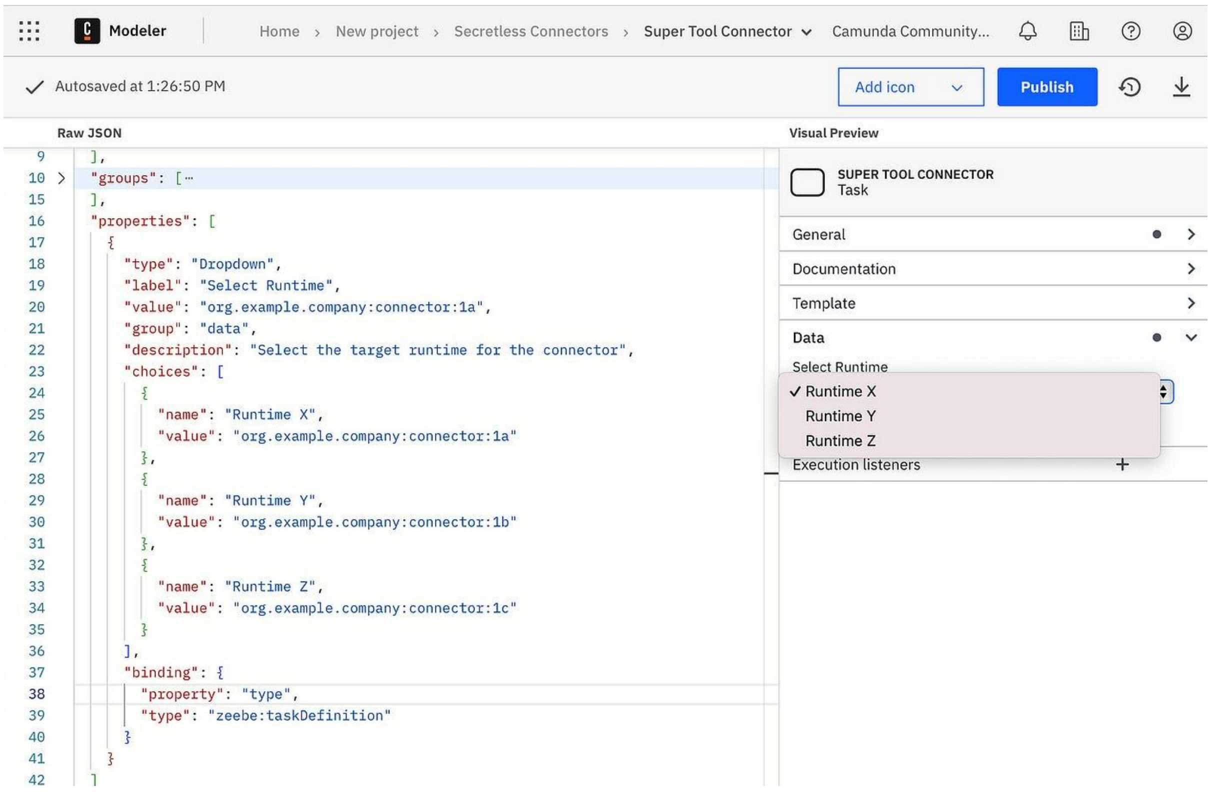
```
try {
  authenticate();
  // do connector logic here...
} catch (AuthException e) {
  // fail connector with custom retry back-off
  throw new ConnectorRetryExceptionBuilder()
          .retries(retries - 1)
          .backoffDuration(Duration.ofSeconds(60)) // wait 60s!
          .errorCode("AUTH_EXCEPTION")
          .message("Authentication failed!")
          .build();
}
```

With the backoff in place, it is possible to restart the application without Camunda raising an incident due to exhausted retries.

Avoiding Duplicate Connector Templates

The solution to this was also relatively easy to execute. The only thing that has to change is the id in the connector runtime and the template. This id is used by the engine to assign a certain job to a connector runtime.

In the template, this value is normally *hidden* for users, but one can also just set it as a dynamic value by making the field visible in the modeler. Since we want to make sure that the correct value is set, we decided to offer a simple drop-down menu in our templates, with all existing runtimes.

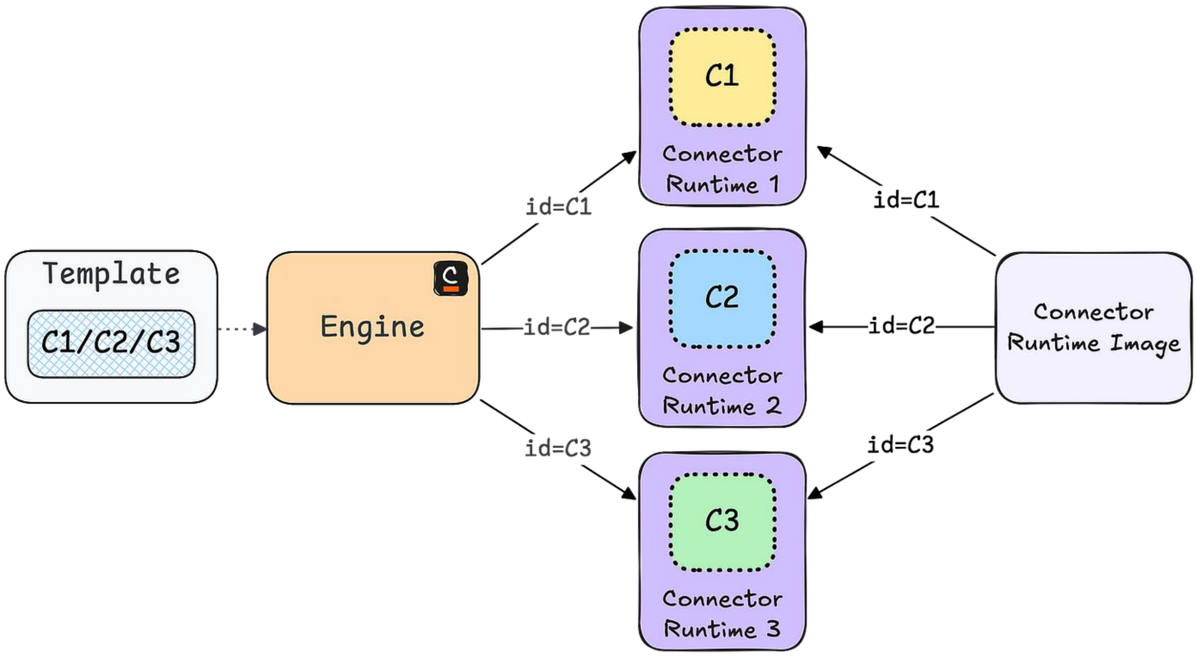


```
  ],
  "groups": [⋯
  ],
  "properties": [
    {
      "type": "Dropdown",
      "label": "Select Runtime",
      "value": "org.example.company:connector:1a",
      "group": "data",
      "description": "Select the target runtime for the connector",
      "choices": [
        {
          "name": "Runtime X",
          "value": "org.example.company:connector:1a"
        },
        {
          "name": "Runtime Y",
          "value": "org.example.company:connector:1b"
        },
        {
          "name": "Runtime Z",
          "value": "org.example.company:connector:1c"
        }
      ],
      "binding": {
        "property": "type",
        "type": "zeebe:taskDefinition"
      }
    }
  ]
```

Select Runtime
✓ Runtime X
Runtime Y
Runtime Z

A connector template with several possible ids

We can still use the same connector runtime image for each business unit, since we are able to overwrite the connector id in each container as well. This is called *hybrid mode*, since you can use one template for several runtimes.



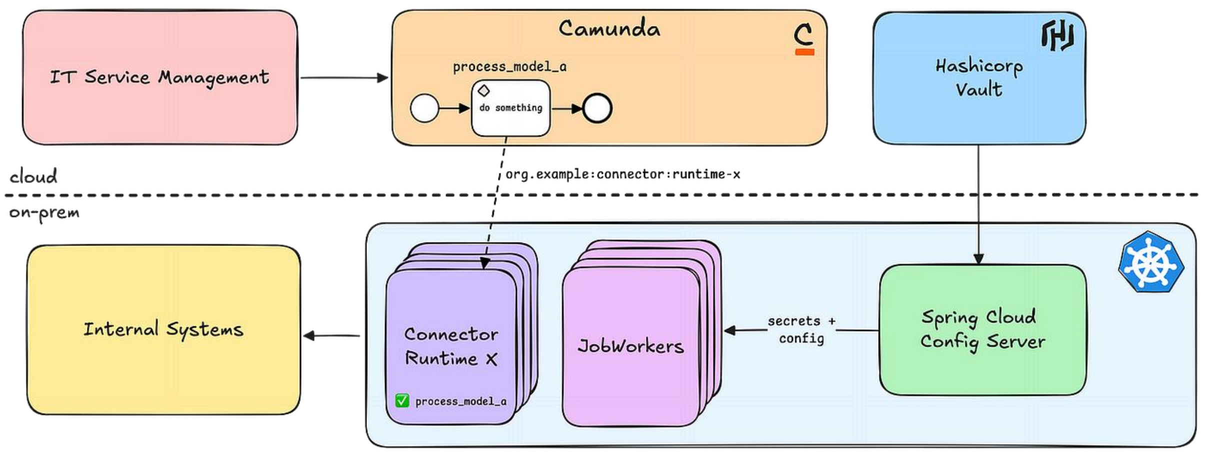
Hybrid mode for shared connectors in different runtimes

To prevent one business unit from connecting to an incorrect runtime, we introduce an allowlist for processes per runtime. This list contains all the process IDs that the runtime is intended to work with. Any calls from other processes are rejected.

The Final Target Architecture

All these decisions lead us to the following architecture, which provides secrets for all runtimes and ensures flexibility under load.

We ensured that components, that would normally scale with the number of runtimes, stay constant, while still ensuring that business units can scale. The only downside to this setup is, that our ops team has to monitor multiple applications now instead of one big runtime. We will monitor this and see if we have to adapt in the future with more business units joining the platform.



Target architecture

How Scalable and Secure is This Architecture?

We separated business units from each other by providing separate runtimes for them. With Kubernetes we are able to scale each runtime individually based on usage. Connector templates provide a list of possible runtimes, with each runtime guards their workers from unintended process access.

Secrets are loaded once at startup. We keep them outside of the engine and connector templates, ensuring they are not accidentally leaked. We support secret rotation by using Camunda’s retry with backoff mechanism for both workers and connectors. This way we keep secrets save and still offer flexibility to rotate them regularly without affecting customers.

Secrets are encrypted during transport and rest, and we have precautions in place to prevent accidental logging.

We use modern mechanisms of a distributed system to ensure operability even though some components are not available due to restarts.