Oracle® AI Database

Oracle Private AI Services Container User's Guide

26ai
G50009-03
February 2026

Oracle AI Database Oracle Private AI Services Container User's Guide, 26ai

G50009-03

Primary Author: Sarah Hirschfeld

Contributors: Aleksandra Czarlinska, Aurosish Mishra, Boriana Milenova, David Jiang, Doug Hood, Rohan Aggarwal, Sean Stacey, Shasank Chavan, Simeon Green, Tirthankar Lahiri, Weiwei Gong, Yuan Zhou

Contents

Preface

Audience i

1 About Oracle Private AI Services Container

2 Install the Private AI Services Container

3 Configure the Private AI Services Container

Install with HTTP and Default Models 2
Install with HTTP with Models and Advanced Options 4
Install with Self-Signed SSL Certificates 7
Install with HTTP/SSL with Models and Advanced Options 9
Use the OpenAI Python SDK Client with HTTP or HTTP/SSL 10

4 Considerations when Configuring the Private AI Services Container

Available Embedding Models 1
Private AI Environment Variables 3
Use API Keys in Clients 5
Container Input Validation 6
Container Automatic Image Conversion 8
Rate Limiting 8
Transport Layer Security 9
Container Log Files 11
Multi-threaded Scaling 11

A Private AI Services Container API Reference

Preface

Oracle Private AI Services Container User's Guide provides information about the Private AI Services Container, which provides a method for offloading expensive AI computation, such as vector embedding generation, outside of the database. This can free up database compute resources that can be used for indexing and similarity search.

- Audience

Audience

This guide is intended for application developers, database administrators, data users, and others who perform the following tasks:

- Implement artificial intelligence (AI) solutions for websites and unstructured or structured data
- Build query applications by using natural language processing and machine learning techniques
- Perform similarity searches on content, such as words, documents, audio tracks, or images

To use this document, you must have a basic familiarity with vector embedding and machine learning concepts, SQL, SQL*Plus, and PL/SQL.

1

About Oracle Private AI Services Container

The Private AI Services Container is a lightweight, containerized web service that provides an interface for performing inference on Oracle ONNX Pipeline format models using REST. This container can run in your data center or on compute nodes in the public cloud.

Using the Private AI Services Container allows you to offload expensive AI computation, such as vector embedding generation, outside of the database. This can free up database compute resources that can be used for indexing and similarity search.

Multiple containers can be run simultaneously using Podman. Oracle Database Kubernetes Operator can be used for container orchestration.

The management of the AI Services Container is handled by certain logical roles including the Container Admin, Model Creator, and Inference Client. These roles may be assumed by the same Linux user simultaneously.

- **Container Admin:** Acting as container admin, a user can configure, stop, start, and perform general management of a container.
- **Model Creator:** As model creator, a user performs actions that are related to the provision of model files. This includes tasks such as creating an ONNX pipeline and providing metadata and model specific configurations when necessary.
- **Inference Client:** As inference client, a user performs inference on data and queries available models using REST APIs.

The container enables multiple concurrent users. The effective number of concurrent users is determined by the number of CPU cores and the embedding model.

> **Note**
>
> User data used for inference is not stored and is only processed transiently. All requests to the container are stateless and the data is not stored.

The AI Services Container supports a fixed set of Oracle Machine Learning (OML) model types and mining functions. The model type is automatically determined by the container based on the data type and shape of the model inputs. The supported model types are as follows:

Model Type	Model Function	Goal
`ONNX_TXT`	`EMBEDDING`	Generate text embeddings. The model takes text as input and produces embeddings as output. Example models include sentence transformers and CLIP (text).
`ONNX_IMG`	`EMBEDDING`	Generate image embeddings. The model takes an image as input and produces embeddings as output. Example models include Vision Transformers and CLIP (image).

Only Oracle ONNX Pipeline formatted models that are produced by OML4Py are supported for deployment in the Private AI Services Container and in the database, as of Oracle AI Database 26ai. Both text and image embedding pipelines are supported. For more information about ONNX pipeline models, see *Oracle Machine Learning for Python User's Guide*. For more information about importing pretrained models in ONNX format, see *Oracle AI Database AI Vector Search User's Guide* .

The container can be called from the database using the `UTL_TO_EMBEDDING` and `UTL_TO_EMBEDDINGS` procedures of the `DBMS_VECTOR` PL/SQL package. The container can also be called by REST clients such as curl, or clients that use the OpenAI SDK. For information about the syntax and for examples using the PL/SQL procedures with the Private AI Services Container, see *Oracle AI Database AI Vector Search User's Guide*.

> **Note**
>
> Support may be available for ancillary use of this program in conjunction with a supported Oracle product, to the extent described in that product's documentation. If support is provided it will be in accordance with Oracle's technical support policies which may be found at https://www.oracle.com/support/policies/.
>
> For licensing information related to Oracle AI Private Services Container, see *Licensing Information User Manual for Oracle Private AI Services Container*.

2

Install the Private AI Services Container

Use the included steps to configure and install the container with best practices using bash scripts.

The Oracle Private AI Services Container is AI infrastructure designed to run on-premises, and optionally in air-gapped environments. The container can create low latency, free vector embeddings in a secure manner within the privacy of your own realm.

The container uses TLS 1.3 and API Keys for security. A PKCS12 keystore is used by default, but JKS is also supported. The container can also work with Security Enhanced Linux (SELinux) in enforcing mode.

The included scripts, provided for HTTP and HTTP/SSL, offer examples of how to use curl to create vector embeddings, list the loaded embedding models, check on the health of the container, and produce runtime metrics.

Prerequisites

The Oracle Private AI Services Container uses Oracle Linux 8 within the container and works with the following host Linux x86_64 distributions:

- Oracle Linux 8.6+
- Oracle Linux 9
- Oracle Linux 10

The AI Services Container uses TLS 1.3. This means that you need a version of OpenSSL that supports TLS 1.3 to create certificates, for example:

- OpenSSL 1.1.1k+
- OpenSSL 3.0+

The AI Services Container can be used with the following software:

- Podman

The chosen software must be one of the following versions:

- Podman 3.4.4+, 4.4+, 5+
- Kubernetes 1.31.1+
- Red Hat OpenShift 4.19+

The included examples use Podman with Oracle Linux 8 as the host operating system on OCI. The example OCI VM where the container is installed is called `privateaivm`.

There are also some system requirements to be aware of when using the Private AI Services Container:

- **Memory:** You must have at least 16 GB of free memory to effectively use the container to create vector embeddings. If you want to use large embedding models or create multiple different types of vectors at the same time, then you will need more memory. Insufficient memory can result in reduced performance.

- **CPU cores:** Although you can use a single CPU core with the ONNX Runtime to create vector embeddings, having more CPU cores will enhance performance. Multiple CPU cores enable either a single vector or many different vectors to be created at the same time using multi-threading.
- **Disk space:** You need at least 22 GB of disk space to effectively use the container. You will need additional disk space for each additional embedding model that you use. You will also need additional disk space depending on the log level for long running containers.

Note that the container should not run on the same machine as the Oracle AI Database Server but on a Linux machine that is close to the database server. The container is designed to accelerate resource intensive tasks such as creating vectors or vector index offload of machines other than the Oracle AI Database. This enables the Oracle AI Database to have low latency and high throughput without being burdened with these resource-intensive AI infrastructure tasks.

1. Install the container.
   - If you are using Oracle Linux 8:

     ```
     sudo dnf module install -y container-tools:ol8
     ```

   - If you are using Oracle Linux 9 or 10:

     ```
     sudo dnf install -y container-tools
     ```

2. Verify that Podman is installed correctly.

   ```
   podman version
   ```

   ```
   podman images
   ```

3. Sign in to Oracle Container Registry (OCR).

   Go to Oracle Container Registry and click **Sign In**.

   You will be asked to enter your SSO user name and password. If you are not yet registered with container-registry.oracle.com, you will be prompted to do so. Registering simply saves your SSO user name with container-registry.oracle.com as an SSO-allowed user.

   After completing the one-time registration, clicking **Sign In** on future visits will prompt you to enter your SSO user name and password only.

4. Accept the License Agreement.

   You must click **Continue** to read and accept the license agreement. This action only needs to be completed once.

5. Generate an auth token to pull the container image.

   To use Podman to download the Private AI Services container image, you will need to use the Podman login with your SSO user name and a generated auth token. To generate an auth token, follow the steps at Generating an Oracle Container Registry Authentication Token in *Podman User's Guide*.

> **Note**
>
> Be sure to copy the generated auth token to a secure location before navigating from the generation page. The token will not be shown again and you will have to delete any existing token before creating another.

6. Pull the container image.

   Download (pull) the Private AI Services Container image to the host where you intend to run it.

   a. Run the following command to log in to OCR.

      ```
      podman login container-registry.oracle.com
      ```

      When prompted for a user name, enter your SSO user name. Enter the generated auth token from the previous step when prompted for a password.

   b. Once you have successfully logged in, pull the image.

      ```
      podman pull container-registry.oracle.com/database/private-ai:25.1.2.0.0
      ```

7. Verify the image has been pulled.

   Run the following command to show all images on your system, including `container-registry.oracle.com/database/private-ai:25.1.2.0.0`.

   ```
   podman images
   ```

8. (Optional) When using air-gapped environments, continue with the following steps so that you can access the Private AI Container Image offline.

   a. Save the Private AI Container Image to a tar file.

      The following command saves the container image to a file named `private-ai-25.1.2.0.0.tar` in the current directory:

      ```
      podman save -o private-ai-25.1.2.0.0.tar  container-registry.oracle.com/database/private-ai:25.1.2.0.0
      ```

   b. Load the tar file on the air-gapped machine.

      After copying the tar file to the air-gapped host, you can load it into the local podman image repository using the following command:

      ```
      podman load -i private-ai-25.1.2.0.0.tar
      ```

      Once this load operation is successful, verify that the Private AI Container Image is accessible on the air-gapped host using `podman images`. This should list all podman images on the host, including `container-registry.oracle.com/database/private-ai:25.1.2.0.0`.

3

Configure the Private AI Services Container

There are several different configurations with which the container can be installed, demonstrated in the following tutorials using scripts that are included in the container image.

The configuration tutorials are ordered from least to most complex. The simplest configurations are recommended as a starting point before trying out the more complex configurations. For security reasons, it is strongly recommended to use TLS 1.3 with an API Key for production deployments.

The configuration tutorials use bash scripts that are described in the following table, which are included in the container image as of version 25.1.2.0.0. These scripts simplify the configuration process for both the secure and non-secure modes of running the container.

Script Name	Purpose
quickStart.sh	This script automates the container setup and calls the other dependent scripts. It can be modified to use the paths that are needed for the different directories.
secretsSetup.sh	This script sets up secrets and certificates needed to launch the container and stores them in a user-specified folder. If the folder does not already exist, it will be created by the script. You can pass a subject distinguished name (DN) as a string and a password as a file in order to avoid user input during execution of the script.
configSetup.sh	You provide directories with models, secrets, and the configuration file, and the script copies them to another user-specified folder. If the directory does not already exist, it will be created. The owner is changed to the host UID of the specified container UID in order to permit logging by the container. A folder for log files is also created.
containerSetup.sh	You provide the directory with the copied files, which can be generated by `configSetup.sh`. This directory is owned by the host UID that corresponds to the container UID, allowing the container to write the necessary logs.
util.sh	This is a helper script with common functions that are called by the other scripts.

To use the scripts, you must first copy them from an image. The following commands will copy the scripts to the current directory. Note that you need to replace `<image version>` with the version of the image that you are using in each of the commands (as an explicit version number, such as `25.1.2.0.0`, not as `latest`).

```
IMAGEID=`podman create container-registry.oracle.com/database/private-
ai:<image version>`


podman cp $IMAGEID:/privateai/scripts/privateai-setup-<image version>.zip .
```

The scripts are designed to allow the container admins to quickly and easily set up a Private AI Services Container. Before you begin, verify that you have the following software installed and have followed the installation steps at Install the Private AI Services Container:

- Oracle Linux 8.6+, 9, or 10
- OpenSSL with TLS 1.3 support
- Podman 4.9.4+
- PrivateAI image loaded on podman

> **Note**
>
> The install scripts require the sudo privilege to run correctly.

The included tutorials using HTTP use `localhost`, as both the client and container run on the same host machine. The IP address or hostname where the container is running can also be used if desired. When the HTTP client is running on a different machine than the container, either the IP address or the hostname of the container must be specified.

There are a number of APIs that you can use to get information about the container. For example, you can verify that the container is running, get a list of currently deployed models, and get information about metrics exposed by the application. For more information, see Private AI Services Container API Reference.

- Install with HTTP and Default Models
  This is the simplest configuration. The API Key and SSL are not used. The default embedding models are used with the HTTP port 8080.
- Install with HTTP with Models and Advanced Options
  This tutorial is a superset of the HTTP with Configuration File tutorial that allows you to define vector embedding models that do not ship with the container. For a more advanced configuration, you can optionally choose to specify the HTTP port, container version, and or container name.
- Install with Self-Signed SSL Certificates
  This configuration is a superset of the configuration using HTTP with a configuration file that uses SSL with self-signed digital certificates.
- Install with HTTP/SSL with Models and Advanced Options
  This configuration combines an SSL configuration with additional embedding models and the configuration file. For a more advanced configuration, you can optionally choose to specify the HTTP port, container version, and or container name.
- Use the OpenAI Python SDK Client with HTTP or HTTP/SSL
  The OpenAI Python client works with the container when using HTTP or HTTP/SSL. You just need to specify the correct HTTP endpoint, along with a valid API KEY when using HTTP/SSL.

Install with HTTP and Default Models

This is the simplest configuration. The API Key and SSL are not used. The default embedding models are used with the HTTP port 8080.

1. Define the directory where you want the Private AI Services Container to be installed. A Linux user with the least privilege will own the subdirectories that the container uses.

Note that the `http` parameter includes two dashes.

```
cd setup
mkdir /home/opc/privateai
export PRIVATE_DIR=/home/opc/privateai
./configSetup.sh -d $PRIVATE_DIR
./containerSetup.sh -d $PRIVATE_DIR --http
```

2. Verify that the container is running.

```
podman ps
```

After the container is running, it will take a few seconds for the container to load the default embedding models. You can verify that the container is ready using the `/health` endpoint:

```
curl -i http://localhost:8080/health
```

3. Now that the container is ready, determine which vector embedding models are available without any explicit model configuration:
   - clip-vit-base-patch32-txt
   - clip-vit-base-patch32-img
   - all-mpnet-base-v2
   - all-MiniLM-L12-v2
   - multilingual-e5-base
   - multilingual-e5-large

```
curl http://localhost:8080/v1/models
```

4. You can now, for example, create text vector embeddings and explicitly define the embedding model to use:
   - Using `multilingual-e5-base`:

     ```
     curl -X POST -H "Content-Type: application/json" -d '{"model":
     "multilingual-e5-base", "input":["This is a phrase to vectorize"]}'
     http://localhost:8080/v1/embeddings
     ```

   - Using `clip-vit-base-patch32-txt`:

     ```
     curl -X POST -H "Content-Type: application/json" -d '{"model": "clip-
     vit-base-patch32-txt", "input":["This is a phrase to vectorize"]}'
     http://localhost:8080/v1/embeddings
     ```

   `/v1/embeddings` and `/v1/models` are examples of using the REST endpoints as defined by the OpenAI API. This container implements these OpenAI API REST endpoints without requiring an OpenAI account or a connection to the public internet.

5. You can gather performance metrics using the `/metrics` endpoint.

```
curl http://localhost:8080/metrics/embeddings_call_latency
```

After following these steps, you now know how to do the following with HTTP:

- List the available embeddings models
- Create vectors based on an embedding model
- Check the health of the container with the `/health` endpoint
- Check the performance metrics with the `/metrics` endpoint

Install with HTTP with Models and Advanced Options

This tutorial is a superset of the HTTP with Configuration File tutorial that allows you to define vector embedding models that do not ship with the container. For a more advanced configuration, you can optionally choose to specify the HTTP port, container version, and or container name.

The API Key and SSL are neither configured nor used in this tutorial.

The default configuration is used with HTTP port 8080.

Choose Your ONNX Pipeline Models

There are many possible embedding models that can be used. Along with models that are shipped with the container, a more extensive list of embedding models that are known to work with the container can be found in Available Embedding Models.

Once you have chosen your desired ONNX Pipeline model, and built it with Oracle Machine Learning Client 2.1 (if you choose a model other than the pre-built options), you need to copy that ONNX file into a directory on the host machine in which the container will run.

You must also have a JSON configuration file that lists the desired ONNX Pipeline models to be used in the container.

In this tutorial, both the ONNX models and the config file are copied into their own directories and run as the container user with least privilege.

1. Set up the JSON configuration file.

   In this example, your desired ONNX Pipeline models are in `/home/opc/models` and your config file is `/home/opc/config/config.json`. The contents of the example `config.json` file are as follows:

```
{
    "environment":{
       "PRIVATE_AI_LOG_LEVEL":"INFO"
    },
    "ratelimiter": {
       "service_requests_per_min": 3000,
       "monitor_requests_per_min": 60
  },
    "models": [
        {
            "modelname":"tinybert",
            "modelfile":"tinybert.onnx",
            "modelfunction":"EMBEDDING",
            "cache_on_startup":true
        },
        {
            "modelname":"snowflake-arctic-embed-s",
            "modelfile":"snowflake-arctic-embed-s.onnx",
```

```
            "modelfunction":"EMBEDDING",
            "cache_on_startup":true
        },
        {
            "modelname":"all-MiniLM-L12-v2",
            "modelfile":"all-MiniLM-L12-v2.onnx",
            "modelfunction":"EMBEDDING",
            "cache_on_startup":true
        }
    ]
}
```

In this configuration file, the `modelfile` (filename) and `modelname` of the two new ONNX Pipeline embedding models are defined. The `modelname` does not need to be related to the ONNX filename, but it makes things simpler for you to manage when you have many embedding models if they are the same name. The `modelname` is an identifier and cannot have a file extension such as `.onnx` or `.zip`.

An existing model that ships with the container is also defined. If no `config.json` file is used, all models that ship with the container are loaded. If a `config.json` file is used, only the models that are explicitly defined in the file are loaded. Minimizing the number of explicitly defined models reduces the memory required by the container, as each model needs to be loaded into memory.

2. Configure and start the container with the three models provided in the configuration file.

```
mkdir /home/opc/privateai
mkdir /home/opc/models
mkdir /home/opc/config
export PRIVATE_DIR=/home/opc/privateai
./configSetup.sh -d $PRIVATE_DIR -m /home/opc/models -c /home/opc/config/
config.json
./containerSetup.sh -d $PRIVATE_DIR --http
podman ps
```

The time it takes to start the container depends on the number and size of ONNX Pipeline models that are in your `models` directory. It may take minutes if you have many ONNX files.

3. Optionally specify the container version.

If multiple versions of the container are available, the latest version will be run by default. If you want to run an older version, you can specify the desired version, for example 25.1.2.0.0.

```
export PRIVATE_DIR=/home/opc/privateai
./configSetup.sh -d $PRIVATE_DIR -m /home/opc/models -c /home/opc/config/
config.json
./containerSetup.sh -d $PRIVATE_DIR --http -v 25.1.2.0.0
```

4. Optionally name the container instance.

You can stop a container by using either its container ID or its name. For example, `podman stop privateai`.

If you would like to give the container instance a specific name, use the `-n` parameter. In this example, `container5` is used:

```
export PRIVATE_DIR=/home/opc/privateai
./configSetup.sh -d $PRIVATE_DIR -m /home/opc/models -c /home/opc/config/
config.json
./containerSetup.sh -d $PRIVATE_DIR --http -n container5
```

5. Combine previous steps to specify multiple parameters.

   First, stop running the container:

   ```
   podman stop container5
   ```

   The commands in this step specify the port, container version, and container name:

   ```
   export PRIVATE_DIR=/home/opc/privateai
   ./configSetup.sh -d $PRIVATE_DIR -m /home/opc/models -c /home/opc/config/
   config.json
   ./containerSetup.sh -d $PRIVATE_DIR --http -p 9000 -v 25.1.2.0.0  -n
   container5
   ```

6. The curl commands to create vectors and to check the `/health`, `/models`, and `/metrics` are the same here as for the installation using HTTP with a configuration file.

   The difference is that you will now see the four new models that you configured in the `/models` endpoint.

   > **Note**
   >
   > HTTP port 9000 is used rather than 8080 because that's how the container was configured.

   ```
   curl http://localhost:9000/v1/models
   ```

7. Specify the embedding model.

   This example vectorizes two input strings with the `snowflake-arctic-embed-s` embedding model:

   ```
   curl -X POST -H "Content-Type: application/json" -d '{"model": "snowflake-
   arctic-embed-s", "input":["A siamese cat","Standard Poodle running"]}'
   http://localhost:9000/v1/embeddings
   ```

   This example vectorizes a single input string for the `tinybert` embedding model:

   ```
   curl -X POST -H "Content-Type: application/json" -d '{"model": "tinybert",
   "input":["Standard Poodle running"]}' http://localhost:9000/v1/embeddings
   ```

Install with Self-Signed SSL Certificates

This configuration is a superset of the configuration using HTTP with a configuration file that uses SSL with self-signed digital certificates.

Self-signed digital certificates are free and can be appropriate for internal deployments and air-gapped systems.

The `secretsSetup.sh` script is used in this tutorial with OpenSSL to create the public and private keys, self-signed digital certificate, API Key, and the Podman secrets. For information about the `secretsSetup.sh` script and where to download the file, see Configure the Private AI Services Container.

TLS 1.3 will be used for SSL for the container's listener. This means that any HTTPS clients also must support TLS 1.3, for example SSL libraries like OpenSSL 1.1.1k+ or equivalent.

In this example, your configuration file is `/home/opc/config/config.json`.

When the `secretsSetup.sh` script is run, the following files are created in the `$SECRETS_DIR` directory:

Filename	Description
api-key	A random string used for authentication. The API Key is a shared secret that is needed by the clients.
cert.pem	The self-signed digital certificate
key.pem	The generated private key
key.pub	The generated public key
keystore	A PKCS12 keystore used to store the certificate password

These files are copied to the `$PRIVATE_DIR/secrets` directory to enable the container to run with least privilege.

1. Determine the fully qualified hostname where the container will run.

   ```
   export HOST=$(hostname -f)
   echo $HOST
   ```

2. Define the directory where you want the container's secrets to be created.

   Make sure to use the fully qualified hostname for the OpenSSL Common name/ hostname question and remember the password for the keystore that is created.

   ```
   mkdir /home/opc/privateai
   mkdir /home/opc/secrets
   export PRIVATE_DIR=/home/opc/privateai
   export SECRETS_DIR=/home/opc/secrets
   ./secretsSetup.sh -s $SECRETS_DIR
   ./configSetup.sh -d $PRIVATE_DIR -s $SECRETS_DIR
   ./containerSetup.sh -d $PRIVATE_DIR
   ```

   The container is now running using HTTPS port 8443.

3. In order to use curl with HTTP/SSL, provide the digital certificate as a parameter.

Now that the container is running in HTTP/SSL mode, the curl commands used in previous tutorials for HTTP will no longer work.

```
curl -i --cacert $SECRETS_DIR/cert.pem https://$HOST:8443/health
```

The `/health` endpoint did not require the API Key but all of the other endpoints will need it.

4. Define the value of the `API_KEY` to make the curl SSL commands easier.

```
export API_KEY=$(cat $SECRETS_DIR/api-key)
```

5. List the loaded models with SSL.

```
curl --header "Authorization: Bearer $API_KEY" --cacert $SECRETS_DIR/
cert.pem  https://$HOST:8443/v1/models
```

6. Create a vector embedding with SSL.

```
curl -X POST --header "Authorization: Bearer $API_KEY" -H "Content-Type:
application/json" -d '{"model": "multilingual-e5-base", "input":["This is
a phrase to vectorize"]}' --cacert $SECRETS_DIR/cert.pem
https://$HOST:8443/v1/embeddings
```

7. Show the runtime metrics with SSL.

```
curl --header "Authorization: Bearer $API_KEY" --cacert $SECRETS_DIR/
cert.pem https://$HOST:8443/metrics/embeddings_call_latency
```

8. Optionally run commands with a different host from the container.

   In the case that you want to connect from a remote host, you will want to use the fully qualified hostname of the container to create the self-signed digital certificate. Use the fully qualified hostname for the container from the client machine.

   The fully qualified hostname for the container must be resolvable from the client machine. You may need to add an entry in the client's `/etc/hosts` file for the container's fully qualified hostname.

   The client machine needs a local copy of the container's self-signed digital certificate and `API_KEY`. Copying the contents of the `$SECRETS_DIR` from the container machine to the client's `$SECRETS_DIR` is the simplest way to do this.

```
curl -i --cacert $SECRETS_DIR/cert.pem https://$HOST:8443/health
curl --header "Authorization: Bearer $API_KEY" --cacert $SECRETS_DIR/
cert.pem  https://$HOST:8443/v1/models
curl -X POST --header "Authorization: Bearer $API_KEY" -H "Content-Type:
application/json" -d '{"model": "multilingual-e5-base", "input":["This is
a phrase to vectorize"]}' --cacert $SECRETS_DIR/cert.pem
https://$HOST:8443/v1/embeddings
curl --header "Authorization: Bearer $API_KEY" --cacert $SECRETS_DIR/
cert.pem https://$HOST:8443/metrics/embeddings_call_latency
```

Install with HTTP/SSL with Models and Advanced Options

This configuration combines an SSL configuration with additional embedding models and the configuration file. For a more advanced configuration, you can optionally choose to specify the HTTP port, container version, and or container name.

In this example, your desired ONNX Pipeline models are in `/home/opc/models` and your config file is `/home/opc/config/config.json`.

1. Configure and start the container with the models provided in the configuration file. Notice the file locations specified when running the `configSetup` script.

   > **Note**
   >
   > If a container is already running and you try to start a new container with the same configuration, then the old container will automatically be stopped and replaced with the new container.
   >
   > If you try to start a new container with a different configuration than that of the running container, then the container start will likely fail, and you will need to stop the existing container.

   ```
   mkdir /home/opc/privateai
   mkdir /home/opc/secrets
   mkdir /home/opc/models
   export PRIVATE_DIR=/home/opc/privateai
   export SECRETS_DIR=/home/opc/secrets
   ./secretsSetup.sh -s $SECRETS_DIR
   ./configSetup.sh -d $PRIVATE_DIR -s $SECRETS_DIR -m /home/opc/models -c /
   home/opc/config/config.json
   ./containerSetup.sh -d $PRIVATE_DIR
   ```

   You can optionally specify the port, version, and container name when running the `containerSetup` script by replacing the last line, as in the following:

   ```
   podman stop privateai
   ./containerSetup.sh -d $PRIVATE_DIR -p 9443 -v 25.1.2.0.0 -n everything4
   podman ps
   ```

2. Define the value of the `API_KEY` to make the curl SSL commands easier and define the `API_KEY` value:

   ```
   export PORT=9443
   export API_KEY=$(cat $SECRETS_DIR/api-key)
   ```

3. Provide the digital certificate as a parameter.

   The same SSL curl commands work for both local and remote client machines.

   ```
   curl -i --cacert $SECRETS_DIR/cert.pem https://$HOST:$PORT/health

   curl --header "Authorization: Bearer $API_KEY" --cacert $SECRETS_DIR/
   ```

```
cert.pem  https://$HOST:$PORT/v1/models

curl -X POST --header "Authorization: Bearer $API_KEY" -H "Content-Type:
application/json" -d '{"model": "tinybert", "input":["This is a phrase to
vectorize"]}' --cacert $SECRETS_DIR/cert.pem https://$HOST:$PORT/v1/
embeddings

curl --header "Authorization: Bearer $API_KEY" --cacert $SECRETS_DIR/
cert.pem https://$HOST:$PORT/metrics/embeddings_call_latency
```

Use the OpenAI Python SDK Client with HTTP or HTTP/SSL

The OpenAI Python client works with the container when using HTTP or HTTP/SSL. You just need to specify the correct HTTP endpoint, along with a valid API KEY when using HTTP/SSL.

The OpenAI Python client requires Python 3.8 or greater.

1. Install Python 3.12 using the `uv` or equivalent Python version manager.

   ```
   cd ~

   curl -LsSf https://astral.sh/uv/install.sh | sh

   uv

   cd ~

   uv venv --python 3.12

   source .venv/bin/activate

   python -V

   mkdir /home/opc/python
   cd /home/opc/python
   ```

2. Install the OpenAI Python client.

   ```
   uv pip install openai
   ```

3. You can now create a Python program to list the available embedding models using HTTP. The `API_KEY` value is not checked with HTTP, but the `API_KEY` parameter is still needed.

   ```
   from openai import OpenAI

   my_url = "http://localhost:8080/v1"
   my_key = "Any string will do"

   client = OpenAI(base_url=my_url, api_key=my_key)

   models = client.models.list()
   for model in models:
      print(f"- {model.id}, {model.modelSize}, {model.modelCapabilities}")
   ```

You can also create a Python program to create a vector using HTTP.

```
from openai import OpenAI

my_endpoint = "http://localhost:8080/v1"
my_api_key  = "Any string will do"
my_sentence = "Just some sample text."
my_model    = "clip-vit-base-patch32-txt"

client = OpenAI(base_url=my_endpoint, api_key=my_api_key)

embeddings = client.embeddings.create(model=my_model,input=my_sentence)
print(embeddings.data[0].embedding)
```

> **Note**
>
> The `/health` and `/metrics` endpoints are helper endpoints and are not part of the OpenAI API.

4. You can now create a Python program to list the embedding models using HTTP/SSL.

> **Note**
>
> Keep in mind, the following conditions must be true:
>
> - The protocol must be HTTPS.
> - The HTTPS Port must be correct, for example 8443.
> - The `API_KEY` must match the value of `$SECRETS_DIR/api-key` on the container machine.
> - The certificate must match the certificate in the `$SECRETS_DIR/cert.pem` file.
> - The OpenAI function must use the valid `API_KEY` and certificate values.

```
from openai import OpenAI
import httpx

my_url = "https://your_FQDN:8443/v1"
my_key = "Your_API_KEY_value"
my_cert = "/home/opc/secrets/cert.pem"
my_string = "Just some sample text."

client = OpenAI(base_url=my_url,
                api_key=my_key,
                http_client=httpx.Client(verify=my_cert))

models = client.models.list()
for model in models:
   print(f"- {model.id}, {model.modelSize}, {model.modelCapabilities}")
```

You can also create a Python program to create a vector embedding using HTTP/SSL.

```
from openai import OpenAI
import httpx

my_url = "https://your_FQDN:9443/v1"
my_key = "Your_API_KEY_value"
my_cert = "/home/opc/secrets/cert.pem"
my_model = "tinybert"
my_string = "Just some sample text."

client = OpenAI(base_url=my_url,
                api_key=my_key,
                http_client=httpx.Client(verify=my_cert))

embeddings = client.embeddings.create(model=my_model,input=my_string)
print(embeddings.data[0].embedding)
```

4

Considerations when Configuring the Private AI Services Container

When you configure the Private AI Services Container, there are a number of options and information to consider that can help you get the most out of this service.

- Available Embedding Models
  A number of embedding models are available to be used with the Private AI Services Container, including some that are shipped with the container. There are also pre-built models available for download as well as models that are known to work with the container but must be built using Oracle Machine Learning for Python (OML4Py) Client 2.1.
- Private AI Environment Variables
  Environment variables can be used to set configuration properties, such as file locations, logging settings, and others. They can be passed to the container when running the container using the standard `-e` flag.
- Use API Keys in Clients
  Clients to the Private AI Services Container must send a valid API key if the service has `PRIVATE_AI_AUTHENTICATION_ENABLED` set to `true`.
- Container Input Validation
  Inputs to the Private AI Services Container can come from a variety of sources, each of which are validated using different methods.
- Container Automatic Image Conversion
  The request header `x-convert-images` can be used to instruct the container to examine the format of input images and convert them as necessary. The JPEG format is supported by default.
- Rate Limiting
  The Private AI Services Container provides a configurable method to control the number of requests per minute that the container can handle for different categories of endpoints. This improves scalability and helps prevent abuse, including denial-of-service (DOS) attacks.
- Transport Layer Security
  The container admin can manage transport layer security using environment variables and a keystore file, along with a password file to access the keystore.
- Container Log Files
  Diagnostic information about the Private AI Services Container is available and stored in log messages.
- Multi-threaded Scaling
  The ONNX Runtime enables multi-threading and can benefit from multiple CPU cores.

Available Embedding Models

A number of embedding models are available to be used with the Private AI Services Container, including some that are shipped with the container. There are also pre-built models available for download as well as models that are known to work with the container but must be built using Oracle Machine Learning for Python (OML4Py) Client 2.1.

The following models are included with the container and are available without any additional download or build steps:

- all-mpnet-base-v2
- all-MiniLM-L12-v2
- multilingual-e5-base
- multilingual-e5-large
- clip-vit-base-patch32-txt
- clip-vit-base-patch32-img

The following ONNX Pipeline Model is pre-built and available for download. You can find the link for downloading the model at *Oracle Machine Learning for Python User's Guide*:

- multilingual-e5-small

The following lists of models are ordered by popularity based on download frequency. They are known to work with the container but must be built using OML4Py Client 2.1. Once you have built your desired ONNX Pipeline model, you must copy that ONNX file into a directory on the host machine in which the container will run. These models can be found for download at https://huggingface.co/models.

English Vector Embedding Models:

- sentence-transformers/all-MiniLM-L6-v2
- sentence-transformers/all-mpnet-base-v2
- prajjwal1/bert-tiny
- BAAI/bge-base-en-v1.5
- BAAI/bge-small-en-v1.5
- colbert-ir/colbertv2.0
- sentence-transformers/all-MiniLM-L12-v2
- BAAI/bge-large-en-v1.5
- nomic-ai/nomic-embed-text-v1.5
- mixedbread-ai/mxbai-embed-large-v1
- sentence-transformers/multi-qa-MiniLM-L6-cos-v1
- thenlper/gte-large
- intfloat/e5-base-v2
- intfloat/e5-large-v2
- WhereIsAI/UAE-Large-V1
- Snowflake/snowflake-arctic-embed-xs
- thenlper/gte-base
- Snowflake/snowflake-arctic-embed-m
- thenlper/gte-small
- intfloat/e5-small-v2
- Snowflake/snowflake-arctic-embed-s
- jinaai/jina-embeddings-v2-small-en

- jinaai/jina-embeddings-v2-base-en
- TaylorAI/bge-micro-v2
- Snowflake/snowflake-arctic-embed-m-v1.5
- TaylorAI/gte-tiny
- Snowflake/snowflake-arctic-embed-l

Multilingual Text Embedding Models:

- sentence-transformers/paraphrase-multilingual-mpnet-base-v2
- intfloat/multilingual-e5-small
- sentence-transformers/distiluse-base-multilingual-cased-v2
- intfloat/multilingual-e5-base
- sentence-transformers/multi-qa-MiniLM-L6-cos-v1
- sentence-transformers/stsb-xlm-r-multilingual
- ibm-granite/granite-embedding-278m-multilingual
- ibm-granite/granite-embedding-107m-multilingual

Vision Embedding Models:

- openai/clip-vit-base-patch32
- google/vit-base-patch16-224
- facebook/deit-tiny-patch16-224
- microsoft/resnet-50
- microsoft/resnet-18
- WinKawaks/vit-small-patch16-224
- WinKawaks/vit-tiny-patch16-224

Cross-Encoder Embedding Models:

These models can be used for reranking.

- cross-encoder/ms-marco-MiniLM-L6-v2
- cross-encoder/ms-marco-MiniLM-L12-v2
- BAAI/bge-reranker-base

Private AI Environment Variables

Environment variables can be used to set configuration properties, such as file locations, logging settings, and others. They can be passed to the container when running the container using the standard `-e` flag.

Typically, the JSON configuration file is the preferred method to specify most configuration settings, as it centralizes the settings and simplifies management. However, certain properties can only be set as environment variables using the `-e` flag and cannot be specified in the configuration file. This is because these environment variables are processed before the JSON file is loaded. The exceptions are listed in the following table:

Variable Name	Type	Default Value	Value Range	Description
PRIVATE_AI_CONFIG_FILE	String	None	None	Specifies the location of a config.json file (may have a different file name) that will be used as the configuration file. This variable is required if PRIVATE_AI_MODELFILE is not specified.
PRIVATE_AI_DRYRUN	Boolean	FALSE	TRUE FALSE	This optional variable specifies whether to run the container in dry run mode. Information about dry run mode can be found at the end of this section.
PRIVATE_AI_LOG_STDOUT_ENABLED	Boolean	FALSE	TRUE FALSE	This optional variable specifies whether to display logs in the console or write to the log directory.

The following list of environment variables (all optional) can be set either in the JSON configuration file or using the `-e` flag when running the container.

Variable Name	Type	Default Value	Value Range	Description
PRIVATE_AI_ONNX_INTRA_OP_NUM_THREADS	Number	None	From 1 to the number of available processors on the host	The number of intra-op threads for the ONNX Runtime. This environment variable is used as the global default when using multiple models. The property can be set on individual models using the configuration file.
PRIVATE_AI_MODEL_SCORING_TIMEOUT	Number	120	1 to 3600	The maximum time (in seconds) that an ONNX model scoring thread may execute before an attempt is made to cancel it.
PRIVATE_AI_MODEL_CACHE_TTL	Number	5400	1 to 7200	The maximum idle time (in seconds) before a cache entry is expired and removed from the model cache.
PRIVATE_AI_MODEL_CACHE_MAXENTRIES	Number	16	1 to 64	The maximum number of entries that may exist in the model cache before the LRU entry is evicted.

Variable Name	Type	Default Value	Value Range	Description
PRIVATE_AI_MAX_FILE_SIZE	Number	800,000,000	1 or greater	The maximum model input byte length
PRIVATE_AI_MAX_IMG_SIZE	Number	20,000,000	1 or greater	The maximum image byte length for an ONNX model.
PRIVATE_AI_LOG_LEVEL	String	INFO	INFO, OFF, SEVERE, WARNING, FINE, FINER, FINEST, ALL	The logging level.
PRIVATE_AI_MAX_MODEL_COUNT	Number	64	1 or greater	The maximum number of models allowed in one configuration file.
PRIVATE_AI_MAX_SCORE_PAYLOAD	Number	20,000,000	1 or greater	The maximum payload for scoring requests.
PRIVATE_AI_MAX_BATCHSIZE	Number	256	1 or greater	The maximum batch size.

The PRIVATE_AI_DRYRUN environment variable enables dry run mode, which allows Kubernetes operators and automation tools to verify container readiness and environmental correctness without launching the full application server. Dry run mode is especially useful for validating dynamic secrets, environment variables, and configuration files prior to exposing application endpoints or initiating expensive processes.

In dry run mode, the container performs all validation and initialization steps normally performed during setup, except it does not start the Micronaut server. This approach helps detect misconfigurations early and integrates seamlessly into DevOps and Kubernetes workflows. The container process will exit immediately after successful validation or upon first failure.

The result of a dry run is indicated by one of two files in the container at /privateai/app/:

- dryrun.suc: indicates validation was successful.
- dryrun.dif: indicates validation failed; the file contains the reason(s) for the failure.

Use API Keys in Clients

Clients to the Private AI Services Container must send a valid API key if the service has PRIVATE_AI_AUTHENTICATION_ENABLED set to true.

The following command shows how clients can make API requests to prevent an "invalid API key" error:

```
curl -X POST --header 'Content-Type: application/json' --header 'Accept:
application/json' --header 'Authorization: Bearer [API_KEY]' -d '{ \
  "model": "all_minilm_v6",
  "input": [ \
      "The quick brown fox jumped over the fence.", \
      "Another test sentence" \
  ] \
}' http://localhost:9091/v1/embeddings
```

You can then configure the database to use the API key to communicate with the Private AI Services Container. A user with the `CREATE CREDENTIAL` privilege can create a credential using the `CREATE CREDENTIAL` helper procedure of the `DBMS_VECTOR` PL/SQL package.

Note that the value of `access_token` in the following command is for example purposes. In practice the value should exactly match the API Key used by the container.

```
declare
  jo json_object_t;
begin
  jo := json_object_t();
  jo.put('access_token', 'A1Aa0abA1AB1a1Abc123ab1A123ab123AbcA12a');
  dbms_vector.create_credential(
    credential_name   => 'PRIVATEAI_CRED',
    params            => json(jo.to_string));
end;
/
```

Once the credential is created, it can be added as part of the call to `UTL_TO_EMBEDDING`.

```
-- declare embedding parameters

var params clob;

begin
 :params := '
{
  "provider": "privateai",
  "credential_name": "PRIVATEAI_CRED",
  "url": "https://hostname:8443/v1/embeddings",
  "model": "all-mpnet-base-v2"
}';
end;
/

-- Get the embeddings
select dbms_vector.utl_to_embedding('Hello world', json(:params)) from dual;
```

Container Input Validation

Inputs to the Private AI Services Container can come from a variety of sources, each of which are validated using different methods.

Inputs to the container include HTTP request input from clients, files from the host system provided by the container admin, and environment variables set by the container admin when starting the container.

HTTP Request Input (REST API)

The REST API accepts the media type `application/json` for all requests that have a body (payload), and returns `application/json` for all responses. Input is parsed and deserialized to an Object using the serialization framework provided by Micronaut. Any errors during this process will be caught and returned as HTTP 400 to the client along with a message describing the problem. After deserialization, the Request Objects are validated against rules that check for the following:

- Null or empty
- That numeric parameters are numbers of the required type and within the required range
- That text, URI, and UUID parameters match expected values or regular expressions corresponding to the required type

Any failures in validation at this point will also be handled and an HTTP 400 error is returned with a message describing the problem. All regular expressions used in the validation of input data are evaluated to confirm that they are not susceptible to Regular Expression denial-of-service (DOS) vulnerabilities.

User Models

The container admin can specify user models as part of the `config.json` file. These models can be on the host file system or specified using a Pre-Authenticated Request (PAR) link. Model files can be `.onnx` files or `.zip` files that contain a single `.onnx` file and metadata. In all cases, the container expects the ONNX format model to be a pipeline that matches a specific signature based on metadata. The metadata attributes used to determine the signature of the ONNX format model are `modelType` and `function`.

Given the metadata, the container will introspect the model to ensure that the inputs and outputs of the ONNX graph match the expected signature. The following table shows the relationship between metadata, signature, and input/output shape:

Signature	Model Type	Model Function	Input Shape	Output Shape
Image Embedding	`ONNX_IMG`	embedding	Single-dimensional array of UINT8	Single-dimensional array of embeddings
Text Embedding	`ONNX_TXT`	embedding	2-dimensional array of String	2-dimensional array of embeddings

ONNX Files

The container checks the validity of ONNX files by loading them with the onnx library and verifying their signatures using the methods described previously. For standalone ONNX files, you specify the location of the ONNX file in the `config.json` file along with the metadata, for example:

```
{
    "modelname":"bert_tiny",
    "modelfile":"bert_tiny.onnx",
    "modelfunction": "EMBEDDING",
    "properties":{
        "PRIVATE_AI_ONNX_INTRA_OP_NUM_THREADS":4
    }
}
```

In this example, the ONNX file is `bert_tiny.onnx`, which will be loaded from `/privateai/models/bert_tiny.onnx` on the container file system. This path *must* be mounted to a directory on the host system that contains the ONNX file. The property `modelfunction`, along with the input shape and data type from the ONNX model file, are used to validate the model before deployment.

Zip Files

For large models, typically those over 2GB, you may use OML4Py to generate a compatible ONNX format model. Because ONNX relies on a protobuffer that does not support files larger

than 2GB, the ONNX Runtime supports loading such models using external data. The OML4Py tool has built in support for generating external data for such models that result in multiple files, including a metadata file. For portability and convenience, these files are zipped and the `zip` file can be provided to the container for loading.

In addition to the validation of ONNX files as described previously, `zip` files go through specific validation to prevent malicious Zip Bomb attacks. The files that are expected to be in the `zip` are fixed and each have maximum allowable sizes that are checked. The container determines a `zip` file by checking its binary signature.

Container Automatic Image Conversion

The request header `x-convert-images` can be used to instruct the container to examine the format of input images and convert them as necessary. The JPEG format is supported by default.

The ONNX models used with the Private AI Services Container include operations, such as resizing and channel reordering, that process images. This processing enables the subsequent nodes of the model graph to produce embeddings or probabilities for image classification. However, these operations are intended to work only with the JPEG image format and will produce errors when the ONNX model is supplied with other formats.

When the request header `x-convert-images` is set to true, the container will examine the format of the images sent in the request and, if the input image format is supported, will convert them to JPEG. The following input formats are supported:

- `.bmp`
- `.gif`
- `.odd`
- `.png`
- `.tiff`

The `x-convert-images` header is mapped to a boolean variable and when not included, defaults to `false`. The following is an example of a request that includes the image conversion header:

```
curl -X POST -H "Content-Type: application/json" -H "x-convert-images: true" -
d "{\"model\": \"image_model\", \"input\":\"$IMAGE_TXT\"}" http://
localhost:9091/v1/embeddings
```

Rate Limiting

The Private AI Services Container provides a configurable method to control the number of requests per minute that the container can handle for different categories of endpoints. This improves scalability and helps prevent abuse, including denial-of-service (DOS) attacks.

The container distinguishes between two types of endpoints, each with an independently configurable rate limit:

- Monitor endpoints: These are used for operational monitoring and health checks, including `/health` and `/metrics`, and are controlled by `monitor_requests_per_min` (default value 60) in the configuration file.

- Service endpoints: These include all of the other API functions such as `/v1/embeddings` and `/v1/models`. These are controlled by `service_requests_per_min` (default value 3000) in the configuration file.

This separation ensures that health and metrics scraping, as well as monitoring, does not interfere with the ability to serve regular API traffic from users or client applications.

Container admins can start the container with rate limiting by adding a `"ratelimiter"` session in the configuration JSON file with `"monitor_requests_per_min"` and `"service_requests_per_min"`, as in the following:

```
{
  "ratelimiter": {
    "service_requests_per_min": 3000,
    "monitor_requests_per_min": 60
  }
}
```

This example configuration file would set a limit of 3000 API (service) requests per minute per IP address and 60 monitor (health, metrics) requests per minute per IP address.

Each IP address is tracked independently for both service and monitor groups and counters reset every minute. If a client exceeds its assigned quota within a 60-second window, the server responds with HTTP 429 (Too Many Requests). This logic protects the core APIs from being overwhelmed while ensuring observability endpoints remain responsive.

Several response headers are provided to help monitor the requests usage:

- `x-ratelimit-limit-requests`: This is the same as `"service_requests_per_min"` or `"monitor_requests_per_min"` defined in the configuration file, depending on the endpoints. It shows the maximum requests allowed per minute per IP address.
- `x-ratelimit-remaining-requests`: This shows the number of requests remaining before the total request resets.
- `x-ratelimit-reset-requests`: This indicates the number of seconds left until the total request is refilled.

If, for example, `service_requests_per_min` has a value of 500 and a client sends 400 scoring or inference requests in the first 25 seconds, the response will include: `x-ratelimit-limit-requests = 500`, `x-ratelimit-remaining-requests = 100` (500 - 400), and `x-ratelimit-reset-requests = 35s` (60 seconds - 25 seconds).

With a `monitor_requests_per_minute` value of 120, if a client sends 60 health check requests in the first 15 seconds, the response will include: `x-ratelimit-limit-requests = 120`, `x-ratelimit-remaining-requests = 60` (120-60), and `x-ratelimit-reset-requests = 45s` (60 seconds - 15 seconds).

Transport Layer Security

The container admin can manage transport layer security using environment variables and a keystore file, along with a password file to access the keystore.

By default, the container provides access using the HTTPS protocol while HTTP is disabled. When the container admin starts a container, the following environment variables must be set:

Variable Name	Default Value	Description
PRIVATE_AI_HTTPS_ENABLED	True	When set to true, HTTPS will be the only supported protocol. If set to false, HTTP will be enabled and the other SSL environments will be ignored. If omitted, the default value of true is used.
PRIVATE_AI_SSL_CERT_TYPE	"PKCS12"	The certificate format for the SSL key store. Valid values are "PKCS12" and "JKS".

The container uses TLS 1.3; other TLS versions are not accepted. The container *only* uses the following ciphers:

- TLS_AES_128_GCM_SHA256
- TLS_AES_256_GCM_SHA384
- TLS_CHACHA20_POLY1305_SHA256

Clients that attempt to connect with ciphers that are not included in the list will be rejected with an SSL error.

In addition to setting the environment variables, the container admin must also create a keystore file. The keystore must be named *"keystore"* and contain the private key and any certificates required to be presented to clients. The format of this file should be consistent with the format specified in the PRIVATE_AI_SSL_CERT_TYPE variable. An extension to the file is not required but if one exists, It must be .pk12 or .pfx for the PKCS12 format or .jks for the JKS format.

Because these files exist on the host system, they must be mounted onto the container file system. The container will only look for these files under /privateai/ssl in the container file system, therefore, the host directory containing the keystore should be mounted by the container admin at that location.

For example, if the keystore is stored at /opt/security/keystore.pk12, the container admin could use the following commands to start the container with SSL support (this example assumes Podman is the container engine):

```
podman run -e PRIVATE_AI_CONFIG_FILE=/privateai/config/config.json
  -e PRIVATE_AI_ENABLE_SSL=true
  --secret keystore --secret key_pwd
  -v /opt/security:/privateai/ssl -v /opt/models:/privateai/models
  -p 9090:8443 <container_name>
```

The container admin also must generate a password file so that the container can access the keystore. For example:

```
cd /opt/.secrets
openssl genrsa -out key.pem
openssl rsa -in key.pem -out key.pub -pubout
openssl pkeyutl -in pwdfile.txt -out pwdfile.enc -pubin -inkey key.pub -
encrypt
rm -rf /opt/.secrets/pwdfile.txt
```

The container admin can then create secrets in the container. The following example uses Podman but similar commands are available for Docker:

```
podman secret create keystore /path/to/ssl/myKeystore.p12
podman secret create key_pwd -
```

The `secretsSetup.sh` script helps in the creation of secrets. For more information about the script and where to download the file, see Configure the Private AI Services Container.

Container Log Files

Diagnostic information about the Private AI Services Container is available and stored in log messages.

An external log folder can be mounted to collect the logs by adding, for example, the `-v /scratch/$USER/logs:/privateai/logs` option. In this example, `/scratch/$USER/logs` is your local log folder.

Both Logback and java.util.logging (JUL) frameworks are used for logging, with Logback handling the logs from the container and JUL handling the logs from java-core-api. The log messages from the two frameworks are written to separate log files with prefixes `privateai-log-logback` and `privateai-mod-jul`.

> **Note**
>
> In order for logs to be displayed in the console, `-e PRIVATE_AI_LOG_STDOUT_ENABLED=true` must be set when starting a container. If not set, log files can be found under `/privateai/logs` in the container path.

The logging level is determined by the `PRIVATE_AI_LOG_LEVEL` environment variable that can be specified in the configuration file. The possible values are as follows:

- `INFO` (default)
- `OFF`
- `SEVERE`
- `WARNING`
- `FINE`
- `FINER`
- `FINEST`
- `ALL`

Multi-threaded Scaling

The ONNX Runtime enables multi-threading and can benefit from multiple CPU cores.

Using multiple threads on a multi-code CPU can reduce the latency for creating a vector for most embedding models. It can also increase the throughput by parallelizing vector creation across requests. The ONNX Runtime automatically sizes thread pools for intra-op and inter-op parallelism based on your workload.

A

Private AI Services Container API Reference

Learn about the REST APIs available to use with Oracle Private AI Services Container.

- models
- models/{id}
- embeddings
- health
- metrics
- metrics/{metricName}
- api

models

Use GET requests to print a list of all currently deployed models.

Syntax

`/v1/models`

Returns

A list of deployed models.

Example Output

```
{
  "data": [
    {
      "id": "h126414603234059290",
      "modelSize": "string",
      "modelDeployedTime": "2025-12-22T15:49:11.745Z",
      "modelCapabilities": [
        "TEXT_EMBEDDINGS"
      ]
    }
  ]
}
```

models/{id}

Use GET requests to print information about a specific model.

Syntax

`/v1/models/{id}`

Parameters

`id` (*string*): A unique model name. This parameter is required.

Returns

Information about the model specified by model ID.

Example Output

```
{
  "id": "L55808652807957200809612118083123839056757756025",
  "modelSize": "string",
  "modelDeployedTime": "2025-12-22T16:52:28.365Z",
  "modelCapabilities": [
    "TEXT_EMBEDDINGS"
  ]
}
```

embeddings

Use POST requests to get embeddings against a model.

Syntax

`/v1/embeddings`

Parameters

`x-convert-images` (*boolean*): Indicates whether images in the input list require conversion to JPG. The default value is `false`.

Example Input

Note that `input` can be a string or an array of strings.

```
{
  "input": "string",
  "model": "string"
}
```

Example Output

- Embedding results:

  ```
  {
    "data": [
      {
        "embedding": [
          0
        ],
        "index": 0
      }
    ],
    "model": "string"
  }
  ```

- 400: Error processing the request data.
- 404: Model not found.
- 500: An error occurred during the score operation for this model.

health

Use GET requests to verify that the container is ready to use.

Syntax

```
/health
```

Example Output

- 200: Private AI Services Container is up and running.
- 401: Unauthorized
- 500: Internal server error

metrics

Use GET requests to return a list of metric names exposed by the application.

Syntax

```
/metrics
```

Returns

Returns a list of metric names exposed by the application.

Example Output

- Successful response with metric names:

```
{
  "names": [
    "embeddings_call_error_total",
    "embeddings_call_latency",
    "embeddings_call_success_total",
    "embeddings_call_total",
    "embeddings_last_latency",
    "http.server.requests",
    "jvm.memory.used",
    "process.cpu.usage",
    "system.cpu.usage"
  ]
}
```

- 401: Unauthorized
- 500: Internal server error

metrics/{metricName}

Use GET requests to return detailed information for a metric, including measurements and available tags. Supports optional tag filters using repeated `tag` query params in the form `key:value`.

Syntax

```
/metrics/{metricName}
```

Parameters

`metricName` (*string*): Metric name as returned by GET /metrics. This parameter is required.

`tag` (*array<string>*): Tag filter(s) in the form `key:value`. Repeat for multiple tags.

Example Output

- Metric details:

```
{
  "name": "embeddings_call_error_total",
  "description": "Total number of errors from embeddings calls.",
  "baseUnit": "count",
  "measurements": [
    {
      "statistic": "COUNT",
      "value": 3
    }
  ],
  "availableTags": [
    {
      "tag": "model"
    },
    {
      "tag": "status",
      "values": [
        "success",
        "error"
      ]
    }
  ]
}
```

- 400: Invalid tag filter
- 404: Metric not found
- 500: Internal server error

api

Use GET requests to return the OpenAI specification for this API in YAML format.

Syntax

`/v1/api`

Returns

Returns the OpenAPI (YAML) document as a string.