MAIA

Release 2.4.5-rc.6

Simone Bendazzoli

Mar 22, 2026

CONTENTS:

CHAPTER

ONE

MAIA TOOLKIT

MAIA Toolkit is the main tool for deploying and managing MAIA, a platform for collaborative research in medical AI.

MAIA is a Kubernetes-based platform designed to facilitate collaborative medical AI research. It supports the entire AI development lifecycle, including data preparation, model training, active learning, deployment, and evaluation. MAIA offers a secure and scalable environment with tools and services that are easy to use and flexible, enabling deployment and management of AI models across various environments, from local workstations to cloud-based clusters. Developed by the Biomedical Imaging Division at KTH Royal Institute of Technology in Stockholm, Sweden, MAIA aims to streamline the development and deployment of AI models in medical research.

MAIA serves two main purposes:

1. **Clinical Research Environment**: MAIA provides a standardized and scalable platform for developing, training, and deploying AI models in medical research. It offers a secure and collaborative environment for researchers to work on AI projects, share data, and collaborate on research projects. Furthermore, MAIA specifically focuses on the final model deployment, enabling researchers to deploy their models in real-world clinical settings.
2. **Educational Environment**: MAIA provides a platform for teaching and learning medical AI. It offers a hands-on learning experience for students and researchers to develop, train, and deploy AI models in a real-world setting.

The toolkit provides a set of scripts and tools to deploy and manage the MAIA platform as a Kubernetes cluster.

1.1 Citation

If you use the MAIA platform in your work, please cite the following paper:

```
Bendazzoli, S., Persson, S., Astaraki, M. et al. MAIA: a collaborative medical AI␣
↪platform for integrated healthcare innovation. npj Artif. Intell. 1, 45 (2025). https:/
↪/doi.org/10.1038/s44387-025-00042-6
```

1.2 Quickstart: Install MAIA in 10 Minutes

MAIA ships with a **one-command installer** that sets up everything you need: Kubernetes, its configuration, the MAIA Core and Admin layers, and the MAIA Dashboard so you can start building projects immediately.

The installer is powered by the **`MAIA.Installation` Ansible collection**, which provides roles and playbooks to install and configure the MAIA platform on a Kubernetes cluster.

For more details, please refer to the MAIA.Installation documentation.

To run the installer, you must prepare a **configuration folder** containing:

- **Inventory**: an Ansible inventory file (or folder) defining your hosts and their roles.
- **Configuration file**: a `config.yaml` file describing the installation steps and options.

1.2.1 Minimal Installation

Below is a minimal example that installs a full MAIA stack on a fresh **Ubuntu 24.04** server named `maia-node-0`.

Example `config.yaml`

```yaml
steps:
  - prepare_hosts
  - configure_hosts
  - install_microk8s
  - install_maia_core
  - install_maia_admin
  - configure_oidc_authentication
  - get_kubeconfig_from_rancher_local
  - configure_maia_dashboard

prepare_hosts:
  nvidia_drivers: true
  ufw: true
  nfs: false
  cifs: false

configure_hosts:
  auto_sync: true

install_microk8s:
  install_microk8s: true
  enable_oidc_microk8s: true
  enable_ca: true
  install_argocd: true
  connect_to_microk8s: false
  connect_to_argocd: false

install_maia_core:
  auto_sync: true

install_maia_admin:
```

(continues on next page)

(continued from previous page)

```
  auto_sync: true

configure_oidc_authentication:
  configure_rancher: true
  configure_harbor: true
  harbor_admin_user: admin
  harbor_admin_pass: Harbor12345

get_kubeconfig_from_rancher_local:
  kubeconfig_file: "local-from-rancher.yaml"

configure_maia_dashboard:
  auto_sync: true

env:
  MAIA_PRIVATE_REGISTRY: ""
  CLUSTER_DOMAIN: "example.maia.com"
  CLUSTER_NAME: "maia-cluster"
  INGRESS_RESOLVER_EMAIL: ""
  K8S_DISTRIBUTION: "microk8s"

cluster_config_extra_env:
  selfsigned: true
  shared_storage_class: microk8s-hostpath
```

Example `inventory`

```
[control-plane]
maia-dev-node-0 ansible_host=127.0.0.1 ansible_connection=local ansible_user=ansible-
↪user ansible_become_password=ansible ansible_become=true ansible_become_method=sudo
```

Run the installer

```
MAIA_Install --config-folder <CONFIG_FOLDER>
```

Replace <CONFIG_FOLDER> with the path to the folder containing your `config.yaml` and `inventory`.

Once the installation is complete, you can access the MAIA Dashboard at `https://maia.<cluster_domain>`. Wait for the dashboard to be ready by checking the `maia-dashboard` namespace:

```
export KUBECONFIG=<CONFIG_FOLDER>/<CLUSTER_NAME>-kubeconfig.yaml
kubectl get pod -n maia-dashboard
```

Output:

```
NAME                                          READY   STATUS    RESTARTS   AGE
admin-minio-tenant-pool-0-0                   2/2     Running   0          44m
maia-admin-maia-dashboard-b87475666-2vs77     1/1     Running   0          3m15s
maia-admin-maia-dashboard-mysql-5fffdd655c-5x92x   1/1     Running   0          3m57s
```

For first-access, you can use the following credentials:

```
username: admin@maia.io
password [Temporary]: admin
```

1.2.2 Installation on Linux and Windows Subsystem for Linux (WSL)

To install MAIA on Linux and Windows Subsystem for Linux (WSL), you can use the following one-command installer:

```
LATEST=$(curl -s https://api.github.com/repos/minnelab/MAIA/releases/latest | grep tag_
↪name | cut -d '"' -f4)
wget "https://github.com/minnelab/MAIA/releases/download/${LATEST}/install_MAIA.sh" &&␣
↪chmod +x install_MAIA.sh && ./install_MAIA.sh
```

To access all the features of MAIA, verify that all the subdomains are mapped in your Windows or Linux hosts files:

```
# Add the following lines to your Windows hosts file:
# C:\Windows\System32\drivers\etc\hosts
# Add the following lines to your Linux hosts file:
# /etc/hosts
<WSL_IP> <domain>
<WSL_IP> traefik.<domain>
<WSL_IP> dashboard.<domain>
<WSL_IP> grafana.<domain>
<WSL_IP> iam.<domain>
<WSL_IP> registry.<domain>
<WSL_IP> mgmt.<domain>
<WSL_IP> minio.<domain>
<WSL_IP> argocd.<domain>
<WSL_IP> maia.<domain>
<WSL_IP> test.<domain>
<WSL_IP> minio.test.<domain>
<WSL_IP> login.<domain>
```

1.3 MAIA Architecture

MAIA is built on top of Kubernetes, a popular open-source container orchestration platform. The platform is designed to be modular and extensible, allowing users to customize and extend its functionality to suit their needs. MAIA is composed of three different layers, each serving a specific purpose:

1.3.1 MAIA Core:

The `MAIA Core` layer includes the core components that provide the basic functionality of the platform.

The core components of MAIA include:

- **ArgoCD**: A GitOps continuous delivery tool for Kubernetes that allows users to deploy applications and manage the cluster's configuration using Git repositories.
- **Traefik**: A reverse proxy and load balancer that allows users to access the services deployed on the Kubernetes cluster.
- **Cert-Manager**: A Kubernetes add-on that automates the management and issuance of TLS certificates.

- **MetalLB**: A load balancer implementation for bare metal Kubernetes clusters.
- **Kubernetes Dashboard**: A web-based UI for managing the Kubernetes cluster, including viewing the cluster's status, deploying applications, and managing the cluster's configuration.
- **Rancher**: A Kubernetes management platform that allows users to manage the Kubernetes cluster, deploy applications, and monitor the cluster's status.
- **Grafana**: A monitoring and observability platform that allows users to monitor the cluster's status, including the CPU, Memory, and GPU usage.
- **Loki**: A log aggregation system that allows users to collect, store, and query logs from the Kubernetes cluster.
- **Prometheus**: A monitoring and alerting toolkit that allows users to monitor the cluster's status and set up alerts based on predefined rules.
- **Tempo**: A distributed tracing system that allows users to trace requests through the Kubernetes cluster.
- **NVIDIA GPU Operator**: A Kubernetes operator that allows users to deploy NVIDIA GPU drivers and device plugins on the Kubernetes cluster.

1.3.2 MAIA Admin:

The `MAIA Admin` layer includes the administrative components that provide the administrative functionality of the MAIA platform.

The admin components of MAIA include:

- **MinIO Operator**: A Kubernetes operator that allows users to deploy MinIO, a high-performance object storage server, on the Kubernetes cluster.
- **Login App**: A Django app that allows users to log in to the MAIA API using OpenID Connect authentication.
- **Keycloak**: An open-source identity and access management tool that allows users to manage the users and roles associated with the MAIA API.
- **Harbor**: A container image registry that allows users to store and distribute container images.
- **MAIA Dashboard**: A web-based dashboard that allows users to register projects, request resources, and access the different MAIA services deployed on the Kubernetes cluster.

1.3.3 MAIA Namespaces:

The `MAIA Namespaces` layer is designed to be project-specific, allowing users to create isolated environments for their projects. This layers is designed to provide the external interfaces for the users to interact with the platform, making the MAIA platform remotely accessible to the users.

The MAIA platform provides a range of applications and tools that you can use to develop your projects, grouped into a *MAIA Workspace*.

Workspace Credentials Configuration

The MAIA Workspace supports customizable user credentials. You can set your preferred username and password by creating a `.env` file in your home directory with `MAIA_USERNAME` and `MAIA_PASSWORD` variables. For detailed information, see the *Workspace Credentials Documentation*.

The MAIA Workspace includes:

- **Jupyter Notebook**: A web-based interactive development environment for Python, R, and other programming languages.
- **Remote Desktop**: A remote desktop to access your workspace.
- **SSH**: Secure Shell access to your workspace.
- **Visual Studio Code**: A powerful code editor with support for debugging, syntax highlighting, and more.
- **RStudio**: An integrated development environment for R.
- **3D Slicer**: A medical image analysis software for visualization and analysis of medical images.
- **FreeSurfer**: A software suite for the analysis and visualization of structural and functional neuroimaging data.
- **QuPath**: A software for digital pathology image analysis.
- **ITK-SNAP**: A software for segmentation of anatomical structures in medical images.
- **MatLab**: A high-level programming language and interactive environment for numerical computation, visualization, and programming.
- **Anaconda**: A distribution of Python and R programming languages for scientific computing.

Additionally, the MAIA platform provides access to a range of cloud services and tools, including:

- **MinIO**: An object storage server for storing large amounts of data.
- **MLFlow**: An open-source platform for managing the end-to-end machine learning lifecycle.
- **Orthanc**: An open-source DICOM server for medical imaging.
- **OHIF**: An open-source platform for viewing and annotating medical images.
- **XNAT [Experimental]** : An open-source platform for managing and sharing medical imaging data.
- **Label Studio**: An open-source platform for data labeling and annotation.
- **KubeFlow**: An open-source platform for deploying machine learning workflows on Kubernetes.
- **MONAI Deploy [Experimental]**: An open-source platform for deploying deep learning models for medical imaging in clinical production settings.

CHAPTER

TWO

MAIA PACKAGE

2.1 Submodules

2.1.1 MAIA.dashboard_utils module

MAIA.dashboard_utils.**decrypt_string**(*private_key*, *string*)

Decrypts an encrypted string using a given private key.

Parameters

- **private_key** (*str*) – Path to the private key file in PEM format.
- **string** (*bytes*) – The encrypted string to be decrypted.

Returns

str – The decrypted string.

Raises

ValueError – If the decryption process fails.

MAIA.dashboard_utils.**encrypt_string**(*public_key*, *string*)

Encrypts a given string using the provided public key.

Parameters

- **public_key** (*str*) – The file path to the public key in PEM format.
- **string** (*str*) – The string to be encrypted.

Returns

str – The encrypted string in hexadecimal format.

Raises

ValueError – If the public key file cannot be read or is invalid.

MAIA.dashboard_utils.**generate_encryption_keys**(*folder_path*)

Generate RSA encryption keys and save them to files.

Parameters

folder_path (*str*) – The path to the folder where the keys will be saved.

Returns

None

MAIA.dashboard_utils.**get_allocation_date_for_project**(*maia_project_model*, *group_id*, *is_namespace_style=False*)

Retrieves the allocation date for a project based on the given group ID.

Parameters

- **maia_project_model** (*Model*) – The Django model representing the MAIA project.
- **group_id** (*str*) – The group ID to match against the project's namespace.
- **is_namespace_style** (*bool, optional*) – If True, the group ID comparison will be done in a namespace style (lowercase and underscores replaced with hyphens). Default is False.

Returns

date or None – The allocation date of the project if a match is found, otherwise None.

MAIA.dashboard_utils.**get_argocd_project_status**(*argocd_namespace*, *project_id*)

async MAIA.dashboard_utils.**get_list_of_deployed_projects**()

Asynchronously retrieves a list of deployed projects from the Argo CD namespace. This function uses a Kubernetes client to list all releases in the "argocd" namespace and returns the names of these releases.

Returns

list of str – A list containing the names of the deployed projects.

Raises

- **KeyError** – If the "KUBECONFIG" environment variable is not set.
- **kubernetes.client.exceptions.ApiException** – If there is an error communicating with the Kubernetes API.

MAIA.dashboard_utils.**get_pending_projects**(*settings*, *maia_project_model*)

Retrieve a list of pending projects that are not in the active groups.

Parameters

- **settings** (*dict*) – Configuration settings required to access Keycloak.
- **maia_project_model** (*Django model*) – The Django model representing the MAIA projects.

Returns

list – A list of namespaces of pending projects.

MAIA.dashboard_utils.**get_project**(*group_id*, *settings*, *maia_project_model*, *is_namespace_style=False*)

Retrieve project details and associated cluster ID based on the group ID.

Parameters

- **group_id** (*str*) – The ID of the group to search for.
- **settings** (*object*) – The settings object containing OIDC configuration.
- **maia_project_model** (*Django model*) – The Django model representing MAIA projects.
- **is_namespace_style** (*bool, optional*) – Flag indicating whether the group ID is in namespace style (default is False).

Returns

tuple – A tuple containing: - namespace_form (dict): A dictionary with project details and resource limits. - cluster_id (str or None): The ID of the associated cluster, or None if not applicable.

MAIA.dashboard_utils.**get_project_argo_status_and_user_table**(*request*, *settings*, *maia_user_model*, *maia_project_model*)

Retrieves the Argo CD project status and user table information.

Parameters

- **request** (*HttpRequest*) – The HTTP request object containing session and user information.
- **settings** (*Settings*) – The settings object containing configuration values.
- **maia_user_model** (*Model*) – The Django model representing MAIA users.
- **maia_project_model** (*Model*) – The Django model representing MAIA projects.

Returns

tuple – A tuple containing: - user_table (dict): The user table information. - to_register_in_groups (list): List of users to register in groups. - to_register_in_keycloak (list): List of users to register in Keycloak. - maia_groups_dict (dict): Dictionary of MAIA groups. - project_argo_status (dict): Dictionary containing the Argo CD project status for each project.

MAIA.dashboard_utils.**get_user_table**(*settings*, *maia_user_model*, *maia_project_model*)

Retrieve user and project information from Keycloak and Minio, and organize it into a dictionary.

Parameters

- **settings** (*object*) – An object containing configuration settings such as OIDC and Minio credentials.
- **maia_user_model** (*Django model*) – The Django model representing MAIA users.
- **maia_project_model** (*Django model*) – The Django model representing MAIA projects.

Returns

tuple – A tuple containing: - users_to_register_in_group (dict): Users to be registered in Keycloak groups. - users_to_register_in_keycloak (list): Users to be registered in Keycloak. - maia_group_dict (dict): Dictionary containing group information including users, conda environments, and project details. - users_to_remove_from_group (dict): Users to be removed from Keycloak groups.

MAIA.dashboard_utils.**register_cluster_for_project_in_db**(*project_model*, *settings*, *namespace*, *cluster*)

Registers a cluster for a project in the database. This function connects to Keycloak to retrieve group information and associates a cluster with a project based on the provided namespace. If the project already exists, it updates the cluster information; otherwise, it creates a new project entry with the specified cluster.

Parameters

- **project_model** (*Django model*) – The Django model representing the project.
- **settings** (*object*) – An object containing configuration settings.
- **namespace** (*str*) – The namespace associated with the project.
- **cluster** (*str*) – The cluster to be registered for the project.

Returns

None

MAIA.dashboard_utils.**send_maia_info_email**(*receiver_email*, *register_project_url*, *register_user_url*, *support_link*)

Send an email with registration information for the MAIA platform. :type receiver_email: :param receiver_email: The email address of the recipient. :type receiver_email: str :type register_project_url: :param register_project_url: The URL for project registration. :type register_project_url: str :type register_user_url: :param register_user_url: The URL for user registration. :type register_user_url: str :type support_link: :param support_link: The URL for the MAIA support webhook. :type support_link: str

Returns

None

MAIA.dashboard_utils.**send_maia_message_email**(*receiver_emails*, *subject*, *message_body*)

Send an email with a custom message to multiple recipients with improved deliverability.

MAIA.dashboard_utils.**send_webhook_message**(*username*, *namespace*, *url*, *project_registration=False*)

Sends a message to a webhook to request a MAIA account.

Parameters

- **username** (*str*) – The username of the person requesting the account.
- **namespace** (*str*) – The project namespace for which the account is being requested.
- **url** (*str*) – The webhook URL to which the message will be sent.
- **project_registration** (*bool, optional*) – If True, indicates that a project registration is also being requested (default is False).

Raises

- **requests.exceptions.HTTPError** – If the HTTP request returned an unsuccessful status code.
- **Prints** –
- **------** –
- **str** – Success message with the HTTP status code if the payload is delivered successfully.
- **str** – Error message if the HTTP request fails.

MAIA.dashboard_utils.**update_user_table**(*form*, *user_model*, *maia_user_model*, *project_model*)

Updates user and project information based on the cleaned data from a form.

Parameters

- **form** (*Form*) – The form containing cleaned data to update the user and project models.
- **user_model** (*Model*) – The user model to query and update user information.
- **maia_user_model** (*Model*) – The MAIA user model to query and update namespace information.
- **project_model** (*Model*) – The project model to query and update project information.

Notes

- The function processes entries in the form's cleaned data to update user namespaces and project details.
- User namespaces are updated or created in the *maia_user_model* based on the user's email.
- Project details are updated or created in the *project_model* based on the namespace.

MAIA.dashboard_utils.**upload_env_file_to_minio**(*env_file*, *namespace*, *settings*)

MAIA.dashboard_utils.**verify_gpu_availability**(*global_existing_bookings*, *new_booking*, *gpu_specs*)

Verify GPU availability for a new booking.

Parameters

- **global_existing_bookings** (*list of dict*) – A list of existing bookings where each booking is represented as a dictionary with keys "gpu", "start_date", and "end_date".
- **new_booking** (*dict*) – A dictionary representing the new booking with keys "gpu", "start_date", and "end_date".

- **gpu_specs** (*list of dict*) – A list of GPU specifications where each specification is represented as a dictionary with keys "name", "replicas", and "count".

Returns

- **overlapping_time_points** (*list of datetime*) – A list of time points where bookings overlap.
- **gpu_availability_per_slot** (*list of int*) – A list of available GPU counts for each overlapping time slot.
- **total_gpus** (*int*) – The total number of GPUs available for the specified GPU type.

MAIA.dashboard_utils.**verify_gpu_booking_policy**(*existing_bookings*, *new_booking*, *global_existing_bookings*, *gpu_specs*)

Verify GPU booking policy to ensure the new booking does not exceed the allowed days and GPU availability.

Parameters

- **existing_bookings** (*list*) – A list of existing booking objects with *start_date* and *end_date* attributes.
- **new_booking** (*dict*) – A dictionary containing the *starting_time* and *ending_time* of the new booking in "%Y-%m-%d %H:%M:%S" format.
- **global_existing_bookings** (*list*) – A list of all existing bookings globally.
- **gpu_specs** (*dict*) – A dictionary containing the specifications of the GPUs.

Returns

- *bool* – True if the booking policy is verified, False otherwise.
- *str or None* – An error message if the booking policy is not verified, None otherwise.

MAIA.dashboard_utils.**verify_minio_availability**(*settings*)

Verifies the availability of a MinIO server.

Parameters

settings (*object*) – An object containing the MinIO configuration settings.

MINIO_URL
[str] The URL of the MinIO server.

MINIO_ACCESS_KEY
[str] The access key for the MinIO server.

MINIO_SECRET_KEY
[str] The secret key for the MinIO server.

BUCKET_NAME
[str] The name of the bucket to check for existence.

Returns

bool – True if the MinIO server is available and the bucket exists, False otherwise.

2.1.2 MAIA.helm_values module

MAIA.helm_values.**read_config_dict_and_generate_helm_values_dict**(*config_dict*, *kubeconfig_dict*)

Read Config Dict and generate helm-ready values dict.

Parameters

- **config_dict** (Dict[str, Any]) – Config Dict
- **kubeconfig_dict** (Dict[str, Any]) – Kubeconfig Dict

Return type
Dict[str, Any]

Returns
Helm values dict

2.1.3 MAIA.keycloak_utils module

MAIA.keycloak_utils.**delete_group_in_keycloak**(*group_id*, *settings*)

Delete a group in Keycloak

Parameters

- **group_id** (*str*) – The ID of the group to be deleted.
- **settings** (*object*) – An object containing the Keycloak server settings. It should have the following attributes: - OIDC_SERVER_URL: str, the URL of the Keycloak server. - OIDC_USERNAME: str, the username for Keycloak authentication. - OIDC_REALM_NAME: str, the realm name in Keycloak. - OIDC_RP_CLIENT_ID: str, the client ID for Keycloak. - OIDC_RP_CLIENT_SECRET: str, the client secret for Keycloak.

Return type
None

Returns
None

MAIA.keycloak_utils.**delete_user_in_keycloak**(*email*, *settings*)

Delete a user in Keycloak

Parameters

- **email** (*str*) – The email address of the user to be deleted.
- **settings** (*object*) – An object containing the Keycloak server settings. It should have the following attributes: - OIDC_SERVER_URL: str, the URL of the Keycloak server. - OIDC_USERNAME: str, the username for Keycloak authentication. - OIDC_REALM_NAME: str, the realm name in Keycloak. - OIDC_RP_CLIENT_ID: str, the client ID for Keycloak. - OIDC_RP_CLIENT_SECRET: str, the client secret for Keycloak.

Return type
None

Returns
None

MAIA.keycloak_utils.**get_access_token**(*keycloak_url*, *keycloak_client_secret*, *ca_cert*)

Get an access token from Keycloak.

Parameters

- **keycloak_url** (*str*) – The URL of the Keycloak server.
- **keycloak_client_secret** (*str*) – The client secret for the Keycloak client.
- **ca_cert** (*str*) – The path to the CA certificate.

Returns

dict – A dictionary containing the access token.

Raises

requests.exceptions.RequestException – If the request to Keycloak fails.

MAIA.keycloak_utils.**get_group_id_in_keycloak**(*group_name*, *settings*)

Retrieve the ID of a group in Keycloak.

Parameters

- **group_name** (*str*) – The name of the group to retrieve the ID of.
- **settings** (*object*) – An object containing the Keycloak server settings. It should have the following attributes: - OIDC_SERVER_URL: str, the URL of the Keycloak server.

Return type

str

MAIA.keycloak_utils.**get_groups_for_user**(*email*, *settings*)

Retrieve the MAIA groups associated with a user in Keycloak.

Parameters

- **email** (*str*) – The email address of the user to retrieve groups for.
- **settings** (*object*) – An object containing the Keycloak server settings. It should have the following attributes: - OIDC_SERVER_URL: str, the URL of the Keycloak server. - OIDC_USERNAME: str, the username for Keycloak authentication. - OIDC_REALM_NAME: str, the realm name in Keycloak. - OIDC_RP_CLIENT_ID: str, the client ID for Keycloak. - OIDC_RP_CLIENT_SECRET: str, the client secret for Keycloak.

Returns

list – A list of MAIA groups that the user is associated with.

MAIA.keycloak_utils.**get_groups_in_keycloak**(*settings*)

Retrieve groups from Keycloak that start with "MAIA:" and return them in a dictionary.

Parameters

- **settings** (*object*)
- **settings.** (*An object containing the Keycloak connection*)
- **attributes** (*It should have the following*)
- **OIDC_SERVER_URL** (-) – The URL of the Keycloak server.
- **OIDC_USERNAME** (-) – The username for Keycloak authentication.
- **OIDC_REALM_NAME** (-) – The name of the Keycloak realm.
- **OIDC_RP_CLIENT_ID** (-) – The client ID for Keycloak.

- **OIDC_RP_CLIENT_SECRET** (-) – The client secret for Keycloak.

Return type
dict[str, str]

Returns
dict – A dictionary where the keys are group IDs and the values are group names (with the "MAIA:" prefix removed) for groups that start with "MAIA:".

MAIA.keycloak_utils.**get_id_token**(*keycloak_url*, *keycloak_client_secret*, *username*, *password*, *ca_cert*, *realm='maia'*, *client_id='maia'*)

Get an ID token from Keycloak.

Parameters

- **keycloak_url** (*str*) – The URL of the Keycloak server.
- **keycloak_client_secret** (*str*) – The client secret for the Keycloak client.
- **username** (*str*) – The username for the Keycloak user.
- **password** (*str*) – The password for the Keycloak user.
- **ca_cert** (*str*) – The path to the CA certificate.
- **realm** (*str*) – The realm to use for the Keycloak client.
- **client_id** (*str*) – The client ID to use for the Keycloak client.

Returns
dict – A dictionary containing the ID token.

Raises
requests.exceptions.RequestException – If the request to Keycloak fails.

MAIA.keycloak_utils.**get_list_of_groups_requesting_a_user**(*email*, *user_model*)

Retrieves a list of groups (namespaces) that have requested a specific user based on their email.

Parameters

- **email** (*str*) – The email address of the user to search for.
- **user_model** (*object*) – The user model object to query for user information.

Return type
list[str]

Returns
list – A list of namespaces that have requested the user. Returns an empty list if no groups are found.

Raises

- **KeyError** – If environment variables 'DB_HOST', 'DB_USERNAME', or 'DB_PASS' are not set in non-debug mode.
- **Exception** – If there is an issue connecting to the database or executing the SQL queries.

MAIA.keycloak_utils.**get_list_of_users_requesting_a_group**(*maia_user_model*, *group_id*)

Retrieves a list of email addresses of users who have requested access to a specific group.

Parameters

- **group_id** (*str*) – The ID of the group to check for user requests.

- **settings** (*object*) – A settings object that contains configuration parameters, including DEBUG and LOCAL_DB_PATH.

Return type
list[str]

Returns
list – A list of email addresses of users who have requested access to the specified group.

Raises

- **KeyError** – If environment variables for database connection are not set when DEBUG is False.
- **Exception** – If there is an issue with database connection or query execution.

Notes

When settings.DEBUG is True, a local SQLite database is used. When settings.DEBUG is False, a MySQL database is used with connection parameters from environment variables.

MAIA.keycloak_utils.**get_maia_users_from_keycloak**(*settings*)

Retrieves all users from Keycloak who are members of any MAIA group.

Parameters

- **settings** (*An object containing Keycloak connection*)
- **settings**
- **including**
- **OIDC_SERVER_URL** (-) – The URL of the Keycloak server.
- **OIDC_USERNAME** (-) – The username for Keycloak authentication.
- **OIDC_REALM_NAME** (-) – The realm name in Keycloak.
- **OIDC_RP_CLIENT_ID** (-) – The client ID for Keycloak.
- **OIDC_RP_CLIENT_SECRET** (-) – The client secret for Keycloak.

Return type
list[dict[str, Any]]

Returns
list – A list of dictionaries containing user information for all users in MAIA groups. Each dictionary contains user details like email, username, and groups.

MAIA.keycloak_utils.**get_user_ids**(*settings*)

Retrieve user IDs and their associated MAIA groups from Keycloak.

Parameters
settings (*object*) – An object containing the Keycloak server settings. It should have the following attributes: - OIDC_SERVER_URL: str, the URL of the Keycloak server. - OIDC_USERNAME: str, the username for Keycloak authentication. - OIDC_REALM_NAME: str, the realm name in Keycloak. - OIDC_RP_CLIENT_ID: str, the client ID for Keycloak. - OIDC_RP_CLIENT_SECRET: str, the client secret for Keycloak.

Returns
dict – A dictionary where the keys are user email addresses and the values are lists of MAIA groups the user belongs to.

MAIA.keycloak_utils.**get_user_username_from_email**(*email*, *settings*)

Retrieve the username for a user from Keycloak.

MAIA.keycloak_utils.**get_users_in_group_in_keycloak**(*group_id*, *settings*)

Retrieve users in a group in Keycloak.

Parameters

- **group_id** (*str*) – The ID of the group to retrieve users from.
- **settings** (*object*) – An object containing the Keycloak server settings. It should have the following attributes: - OIDC_SERVER_URL: str, the URL of the Keycloak server. - OIDC_USERNAME: str, the username for Keycloak authentication. - OIDC_REALM_NAME: str, the realm name in Keycloak. - OIDC_RP_CLIENT_ID: str, the client ID for Keycloak. - OIDC_RP_CLIENT_SECRET: str, the client secret for Keycloak.

Return type
list[str]

Returns
list[str] – A list of email addresses of users in the group.

MAIA.keycloak_utils.**register_group_in_keycloak**(*group_id*, *settings*)

Registers a group in Keycloak with the specified group ID and settings.

Parameters

- **group_id** (*str*) – The ID of the group to be registered.
- **settings** (*An object containing the Keycloak server*)
- **settings**
- **including**
- **OIDC_SERVER_URL** (-) – The URL of the Keycloak server.
- **OIDC_USERNAME** (-) – The username for Keycloak authentication.
- **OIDC_REALM_NAME** (-) – The name of the Keycloak realm.
- **OIDC_RP_CLIENT_ID** (-) – The client ID for Keycloak.
- **OIDC_RP_CLIENT_SECRET** (-) – The client secret for Keycloak.

Return type
None

Returns
None

MAIA.keycloak_utils.**register_user_in_keycloak**(*email*, *settings*, *username=None*, *temp_password='Maia4YOU!'*)

Registers a user in Keycloak and sends an approved registration email.

Parameters

- **email** (*str*) – The email address of the user to be registered.
- **settings** (*object*) – An object containing the necessary settings for Keycloak connection and email sending.
- **username** (*str, optional*) – The Keycloak username. If not provided, email is used (username and email can differ).

- **temp_password** (*str, optional*) – The temporary password for the user. If not provided, "Maia4YOU!" is used.
- **Attributes** (*Settings*)
- **--------------------**
- **OIDC_SERVER_URL** (*str*) – The URL of the Keycloak server.
- **OIDC_USERNAME** (*str*) – The username for Keycloak authentication.
- **OIDC_REALM_NAME** (*str*) – The name of the Keycloak realm.
- **OIDC_RP_CLIENT_ID** (*str*) – The client ID for Keycloak.
- **OIDC_RP_CLIENT_SECRET** (*str*) – The client secret for Keycloak.
- **HOSTNAME** (*str*) – The hostname for generating the MAIA login URL.

Return type
None

Returns
None

MAIA.keycloak_utils.**register_users_in_group_in_keycloak**(*emails*, *group_id*, *settings*)

Registers users in a specified Keycloak group.

Parameters

- **emails** (*list*) – A list of email addresses of users to be added to the group.
- **group_id** (*str*) – The ID of the group to which users should be added.
- **settings** (*An object containing Keycloak server*)
- **settings**
- **including**
- **OIDC_SERVER_URL** (-) – The URL of the Keycloak server.
- **OIDC_USERNAME** (-) – The username for Keycloak authentication.
- **OIDC_REALM_NAME** (-) – The realm name in Keycloak.
- **OIDC_RP_CLIENT_ID** (-) – The client ID for Keycloak.
- **OIDC_RP_CLIENT_SECRET** (-) – The client secret for Keycloak.

Return type
None

Returns
None

MAIA.keycloak_utils.**remove_user_from_group_in_keycloak**(*email*, *group_id*, *settings*)

Remove a user from a group in Keycloak.

Parameters

- **email** (*str*) – The email address of the user to be removed from the group.
- **group_id** (*str*) – The ID of the group from which the user should be removed.
- **settings** (*object*) – An object containing the Keycloak server settings. It should have the following attributes: - OIDC_SERVER_URL: str, the URL of the Keycloak server. - OIDC_USERNAME: str, the username for Keycloak authentication. -

OIDC_REALM_NAME: str, the realm name in Keycloak. - OIDC_RP_CLIENT_ID: str, the client ID for Keycloak. - OIDC_RP_CLIENT_SECRET: str, the client secret for Keycloak.

Return type
None

Returns
None

2.1.4 MAIA.kubernetes_utils module

MAIA.kubernetes_utils.**create_cifs_secret**(*request*, *cluster_id*, *settings*, *namespace*, *user_id*, *username*, *password*, *public_key*)

Create a CIFS secret in the specified Kubernetes namespace.

Parameters

- **request** (*HttpRequest*) – The HTTP request object containing session and user information.
- **cluster_id** (*str*) – The ID of the Kubernetes cluster.
- **settings** (*dict*) – The settings dictionary containing configuration details.
- **namespace** (*str*) – The Kubernetes namespace where the secret will be created.
- **user_id** (*str*) – The user ID for the CIFS secret.
- **username** (*str*) – The username for the CIFS secret.
- **password** (*str*) – The password for the CIFS secret.
- **public_key** (*str*) – The public key for the CIFS secret.

Returns
None

MAIA.kubernetes_utils.**create_cifs_secret_from_context**(*namespace*, *user_id*, *username*, *password*, *public_key*)

Create a CIFS secret in the specified Kubernetes namespace.

Parameters

- **namespace** (*str*) – The Kubernetes namespace where the secret will be created.
- **user_id** (*str*) – The user ID to be used in the secret name.
- **username** (*str*) – The CIFS username to be encrypted and stored in the secret.
- **password** (*str*) – The CIFS password to be encrypted and stored in the secret.
- **public_key** (*str*) – The public key used to encrypt the username and password.

Returns
None

Raises
ApiException – If there is an error when calling the Kubernetes API to create the secret.

MAIA.kubernetes_utils.**create_docker_registry_secret_from_context**(*docker_credentials*, *namespace*, *secret_name*)

Creates a Kubernetes secret of type *kubernetes.io/dockerconfigjson* in the specified namespace using the provided Docker registry credentials.

Parameters

- **docker_credentials** (*dict*)
- **keys** (*A dictionary containing Docker registry credentials with the following*)
- **"registry"** (-) – The Docker registry URL (e.g., "https://index.docker.io/v1/").
- **"username"** (-) – The username for the Docker registry.
- **"password"** (-) – The password for the Docker registry.
- **namespace** (*str*) – The Kubernetes namespace where the secret will be created.
- **secret_name** (*str*) – The name of the Kubernetes secret to be created.

Raises

ApiException – If there is an error while creating the Kubernetes secret, an exception is raised with details about the failure.

Notes

This function uses the Kubernetes Python client to create the secret. Ensure that the Kubernetes client is properly configured to interact with the desired cluster.

MAIA.kubernetes_utils.**create_helm_repo_secret_from_context**(*repo_name*, *helm_repo_config*, *argocd_namespace='argocd'*)

Create a Helm repository secret in the specified Argo CD namespace using the provided Helm repository configuration.

Parameters

- **repo_name** (*str*) – The name of the Helm repository.
- **helm_repo_config** (*dict*) – A dictionary containing the Helm repository configuration with the following keys: - "username" (str): The username for the Helm repository. - "password" (str): The password for the Helm repository. - "project" (str): The project associated with the Helm repository. - "url" (str): The URL of the Helm repository. - "type" (str): The type of the Helm repository. - "name" (str): The name of the Helm repository. - "enableOCI" (str): A flag indicating whether OCI is enabled for the Helm repository.
- **argocd_namespace** (*str, optional*) – The namespace in which to create the secret (default is "argocd").

Returns

None

Raises

ApiException – If there is an error when calling the Kubernetes API to create the secret.

MAIA.kubernetes_utils.**create_kubeflow_profile**(*namespace*, *owner*)

Creates a Kubeflow Profile in the specified namespace.

Parameters

- **namespace** (*str*) – The namespace to create the Kubeflow Profile in.

- **owner** (*str*) – The owner of the Kubeflow Profile.

MAIA.kubernetes_utils.**create_kubeflow_profile_resources**(*namespace*, *owner*, *uid*)

Creates ServiceAccounts, RoleBindings, and an Istio AuthorizationPolicy for a Kubeflow profile namespace.

MAIA.kubernetes_utils.**create_maia_rbac**(*request*, *cluster_id*, *settings*, *namespace*)

MAIA.kubernetes_utils.**create_maia_rbac_from_context**(*namespace*)

MAIA.kubernetes_utils.**create_namespace**(*request*, *settings*, *namespace_id*, *cluster_id*, *kubeflow_namespace=False*, *owner_email=None*)

Creates a Kubernetes namespace using the provided request, settings, namespace ID, and cluster ID.

Parameters

- **request** (*HttpRequest*) – The HTTP request object containing session and user information.
- **settings** (*Settings*) – The settings object containing configuration details.
- **namespace_id** (*str*) – The ID of the namespace to be created.
- **cluster_id** (*str*) – The ID of the Kubernetes cluster where the namespace will be created.

Returns

None

Raises

ApiException – If an error occurs while creating the namespace using the Kubernetes API.

MAIA.kubernetes_utils.**create_namespace_from_context**(*namespace_id*, *kubeflow_namespace=False*, *owner_email=None*)

Create a Kubernetes namespace using the provided namespace ID.

Parameters

- **namespace_id** (*str*) – The ID of the namespace to be created.
- **kubeflow_namespace** (*bool, optional*) – Flag indicating if the namespace is a Kubeflow namespace. Defaults to False.
- **owner_email** (*str, optional*) – The email of the owner of the namespace. Defaults to None.

Returns

None – This function does not return any value. It prints the API response or an exception message.

Raises

ApiException – If there is an error when calling the Kubernetes CoreV1Api to create the namespace.

MAIA.kubernetes_utils.**generate_kubeconfig**(*id_token*, *user_id*, *namespace*, *cluster_id*, *settings*, *in_local_cluster_token=False*)

Generates a Kubernetes configuration dictionary for a given user and cluster.

Parameters

- **id_token** (*str*) – The ID token for the user.
- **user_id** (*str*) – The user ID.
- **namespace** (*str*) – The Kubernetes namespace.

- **cluster_id** (*str*) – The cluster ID.
- **in_local_cluster_token** (*bool, optional*) – Whether to use the local cluster token instead of the ID token.
- **settings** (*object*) – An object containing various settings, including: - CLUSTER_NAMES (dict): A dictionary mapping cluster names to their IDs. - PRIVATE_CLUSTERS (dict): A dictionary of private clusters with their tokens. - OIDC_ISSUER_URL (str): The OIDC issuer URL. - OIDC_RP_CLIENT_ID (str): The OIDC client ID. - OIDC_RP_CLIENT_SECRET (str): The OIDC client secret.

Returns

dict – A dictionary representing the Kubernetes configuration.

MAIA.kubernetes_utils.**get_available_resources**(*id_token*, *api_urls*, *cluster_names*, *private_clusters=None*)

Retrieves available GPU, CPU, and RAM resources from multiple Kubernetes clusters.

Parameters

- **id_token** (*str*) – The ID token for authentication.
- **api_urls** (*list*) – List of API URLs for the Kubernetes clusters.
- **cluster_names** (*dict*) – Dictionary mapping API URLs to cluster names.
- **private_clusters** (*list, optional*) – List of private clusters with their tokens. Defaults to {}.

Returns

tuple –

A tuple containing:

- gpu_dict (dict): Dictionary with GPU availability information for each node.
- cpu_dict (dict): Dictionary with CPU availability information for each node.
- ram_dict (dict): Dictionary with RAM availability information for each node.
- gpu_allocations (dict): Dictionary with GPU allocation details for each pod.

MAIA.kubernetes_utils.**get_cluster_status**(*id_token*, *api_urls*, *cluster_names*, *private_clusters=None*)

Retrieve the status of clusters and their nodes.

Parameters

- **id_token** (*str*) – The ID token for authentication.
- **api_urls** (*list*) – A list of API URLs for the clusters.
- **cluster_names** (*dict*) – A dictionary mapping API URLs to cluster names.
- **private_clusters** (*dict, optional*) – A dictionary mapping private cluster API URLs to their tokens. Defaults to {}.

Returns

tuple –

A tuple containing:

- node_status_dict (dict): A dictionary mapping node names to their status and schedulability.
- cluster_dict (dict): A dictionary mapping cluster names to their node names.

MAIA.kubernetes_utils.**get_filtered_available_nodes**(*gpu_dict*, *cpu_dict*, *ram_dict*, *gpu_request*, *cpu_request*, *memory_request*)

Filters and returns nodes that meet the specified GPU, CPU, and memory requirements.

Parameters

- **gpu_dict** (*dict*) – A dictionary where keys are node names and values are lists containing GPU information.
- **cpu_dict** (*dict*) – A dictionary where keys are node names and values are lists containing CPU information.
- **ram_dict** (*dict*) – A dictionary where keys are node names and values are lists containing RAM information.
- **gpu_request** (*int*) – The minimum number of GPUs required.
- **cpu_request** (*float*) – The minimum amount of CPU required.
- **memory_request** (*float*) – The minimum amount of memory required.

Returns

tuple – Three dictionaries containing the filtered nodes and their respective GPU, CPU, and RAM information.

MAIA.kubernetes_utils.**get_minio_shareable_link**(*object_name*, *bucket_name*, *settings*)

MAIA.kubernetes_utils.**get_namespace_details**(*settings*, *id_token*, *namespace*, *user_id*, *is_admin=False*)

Retrieve details about the namespace including workspace applications, remote desktops, SSH ports, MONAI models, Orthanc instances and deployed clusters.

Parameters

- **settings** (*object*) – Configuration settings containing API URLs and private cluster tokens.
- **id_token** (*str*) – Identity token for authentication.
- **namespace** (*str*) – The namespace to retrieve details for.
- **user_id** (*str*) – The user ID to filter resources.
- **is_admin** (*bool, optional*) – Flag indicating if the user has admin privileges. Defaults to False.

Returns

tuple – A tuple containing: - maia_workspace_apps (dict): Dictionary of workspace applications with their URLs. - remote_desktop_dict (dict): Dictionary of remote desktop URLs for users. - ssh_ports (dict): Dictionary of SSH ports for users. - monai_models (dict): Dictionary of MONAI models. - orthanc_list (dict): Dictionary of Orthanc instances. - deployed_clusters (list): List of clusters where the namespace is deployed.

MAIA.kubernetes_utils.**get_namespaces**(*id_token*, *api_urls*, *private_clusters=None*)

Retrieves a list of unique namespaces from multiple API URLs.

Parameters

- **id_token** (*str*) – The ID token used for authorization when accessing public clusters.
- **api_urls** (*list*) – A list of API URLs to query for namespaces.
- **private_clusters** (*dict, optional*) – A dictionary where keys are API URLs of private clusters and values are their respective tokens. Defaults to an empty dict.

Returns
list – A list of unique namespace names retrieved from the provided API URLs.

MAIA.kubernetes_utils.**get_profile_uid**(*profile_name*)

Reads the UID of a Kubeflow Profile given its name.

Parameters
profile_name (*str*) – The name of the Profile resource.

Return type
str

Returns
str – The UID of the resource, or None if not found/error.

MAIA.kubernetes_utils.**label_pod_for_deletion**(*namespace*, *pod_name*)

Label a Kubernetes pod for deletion by adding a 'terminate-at' annotation.

Parameters

- **namespace** (*str*) – The namespace of the pod.
- **pod_name** (*str*) – The name of the pod to be labeled for deletion.

Raises
Exception – If there is an error labeling the pod for deletion.

MAIA.kubernetes_utils.**retrieve_json_key_for_maia_registry_authentication**(*request*, *cluster_id*, *settings*, *namespace*, *secret_name*, *registry_url*)

Retrieves the JSON key for MAIA registry authentication.

This function generates a kubeconfig dictionary using the provided OpenID Connect (OIDC) ID token and user information, writes it to a temporary kubeconfig file, and sets the *KUBECONFIG* environment variable. It then delegates the retrieval of the JSON key to another function.

Parameters

- **request** (*HttpRequest*) – The HTTP request object containing the session and user information.
- **cluster_id** (*str*) – The ID of the Kubernetes cluster.
- **settings** (*dict*) – The settings dictionary containing configuration details.
- **namespace** (*str*) – The Kubernetes namespace where the secret is located.
- **secret_name** (*str*) – The name of the Kubernetes secret containing the registry credentials.
- **registry_url** (*str*) – The URL of the container registry.

Returns
dict – A dictionary containing the JSON key for MAIA registry authentication.

Raises

- **KeyError** – If the *oidc_id_token* is not found in the session.
- **FileNotFoundError** – If there is an issue writing the kubeconfig file to the temporary directory.

MAIA.kubernetes_utils.**retrieve_json_key_for_maia_registry_authentication_from_context**(*namespace*, *secret_name*, *registry_url*)

Retrieve the JSON key for MAIA registry authentication from a Kubernetes secret. This function reads a Kubernetes secret in the specified namespace, decodes the *.dockerconfigjson* field, and extracts the password for the given registry URL.

Parameters

- **namespace** (*str*) – The namespace in which the Kubernetes secret is located.
- **secret_name** (*str*) – The name of the Kubernetes secret containing the *.dockerconfigjson*.
- **registry_url** (*str*) – The URL of the container registry for which the authentication key is required.

Returns

str – The password associated with the specified registry URL in the *.dockerconfigjson*. Returns an empty dictionary if an exception occurs.

Raises

kubernetes.client.exceptions.ApiException – If there is an error while reading the Kubernetes secret.

2.1.5 MAIA.maia_admin module

MAIA.maia_admin.**create_harbor_values**(*config_folder*, *project_id*, *cluster_config_dict*)

Create and save Harbor values configuration for a given project and cluster configuration.

Parameters

- **config_folder** (*str*) – The path to the configuration folder where the Harbor values file will be saved.
- **project_id** (*str*) – The unique identifier for the project.
- **cluster_config_dict** (*dict*) –

  A dictionary containing cluster configuration details, including:

  - domain (str): The domain name for the Harbor registry.
  - ingress_class (str): The ingress class to be used (e.g., “maia-core-traefik”, “nginx”).
  - traefik_resolver (str, optional): The Traefik resolver to be used if ingress_class is “maia-core-traefik”.

Returns

dict – A dictionary containing the following keys: - namespace (str): The Kubernetes namespace for Harbor. - release (str): The release name for the Harbor Helm chart. - chart (str): The name of the Harbor Helm chart. - repo (str): The URL of the Harbor Helm chart repository. - version (str): The version of the Harbor Helm chart. - values (str): The path to the generated Harbor values YAML file.

MAIA.maia_admin.**create_keycloak_values**(*config_folder*, *project_id*, *cluster_config_dict*)

Generates Keycloak Helm chart values and writes them to a YAML file.

Parameters

- **config_folder** (*str*) – The path to the configuration folder where the YAML file will be saved.
- **project_id** (*str*) – The project identifier used to create a unique namespace and release name.
- **cluster_config_dict** (*dict*) – A dictionary containing cluster configuration details such as domain, ingress class, and traefik resolver.

Returns

dict – A dictionary containing the namespace, release name, chart name, repository URL, chart version, and the path to the generated values YAML file.

MAIA.maia_admin.**create_maia_admin_toolkit_values**(*config_folder*, *project_id*, *cluster_config_dict*)

Creates and writes the MAIA admin toolkit values to a YAML file.

Parameters

- **config_folder** (*str*) – The path to the configuration folder.
- **project_id** (*str*) – The project identifier.
- **cluster_config_dict** (*dict*) – Dictionary containing cluster configuration values.

Returns

dict – A dictionary containing the namespace, release name, chart name, repository URL, chart version, and the path to the generated values YAML file.

MAIA.maia_admin.**create_maia_dashboard_values**(*config_folder*, *project_id*, *cluster_config_dict*)

Create MAIA dashboard values for Helm chart deployment.

Parameters

- **config_folder** (*str*) – The path to the configuration folder.
- **project_id** (*str*) – The project identifier.
- **cluster_config_dict** (*dict*) – Dictionary containing cluster configuration details.

Returns

dict – A dictionary containing the namespace, release name, chart name, repository URL, chart version, and the path to the generated values YAML file.

MAIA.maia_admin.**create_rancher_values**(*config_folder*, *project_id*, *cluster_config_dict*)

Generates Rancher values configuration and writes it to a YAML file.

Parameters

- **config_folder** (*str*) – The path to the configuration folder.
- **project_id** (*str*) – The project identifier.
- **cluster_config_dict** (*dict*) – A dictionary containing cluster configuration details.

Returns

dict – A dictionary containing Rancher deployment details including namespace, repo URL, chart version, values file path, release name, and chart name.

MAIA.maia_admin.**get_maia_toolkit_apps**(*group_id*, *password*, *argo_cd_host*)

Retrieve and print information about a specific project and its associated applications from Argo CD.

Parameters

- **group_id** (*str*) – The group identifier used to construct project and application names.

- **password** (*str*) – The authorization token for accessing the Argo CD API.
- **argo_cd_host** (*str*) – The host URL of the Argo CD server.

Returns

list – A list of dictionaries containing the name and version of each application. Each dictionary has the following keys: - name (str): The name of the application. - version (str): The version of the application.

Example

apps = get_maia_toolkit_apps("maia-core", "password", "http://localhost:8080") logger.info(f"Apps: {apps}")

async MAIA.maia_admin.**install_maia_project**(*group_id*, *values_file*, *argo_cd_namespace*, *project_chart*, *project_repo=None*, *project_version=None*, *json_key_path=None*)

Installs or upgrades a MAIA project using the specified Helm chart and values file.

Parameters

- **group_id** (*str*) – The group ID for the project. This will be used as the release name.
- **values_file** (*str*) – Path to the YAML file containing the values for the Helm chart.
- **argo_cd_namespace** (*str*) – The namespace in which to install the project.
- **project_chart** (*str*) – The name of the Helm chart to use for the project.
- **project_repo** (*str, optional*) – The repository URL where the Helm chart is located. Defaults to None.
- **project_version** (*str, optional*) – The version of the Helm chart to use. Defaults to None.
- **json_key_path** (*str, optional*) – Path to the JSON key file for authentication with the Helm registry. Defaults to None.

Returns

None

Raises

- **FileNotFoundError** – If the values file does not exist.
- **yaml.YAMLError** – If there is an error parsing the values file.
- **Exception** – If there is an error during the installation or upgrade process.

2.1.6 MAIA.maia_core module

MAIA.maia_core.**create_cert_manager_values**(*config_folder*, *project_id*)

Creates a dictionary of values for configuring cert-manager and writes it to a YAML file.

Parameters

- **config_folder** (*str*) – The path to the configuration folder.
- **project_id** (*str*) – The project identifier.

Returns

dict – A dictionary containing the namespace, repository URL, chart version, path to the values file, release name, and chart name.

MAIA.maia_core.**create_core_toolkit_values**(*config_folder*, *project_id*, *cluster_config_dict*)

Creates and saves the core toolkit values for a Kubernetes cluster. This function generates a dictionary of core toolkit values based on the provided configuration folder, project ID, and cluster configuration dictionary. It retrieves the internal IP addresses of the nodes in the Kubernetes cluster and uses them to configure the MetalLB addresses. The generated values are saved to a YAML file.

Parameters

- **config_folder** (*str*) – The path to the configuration folder.
- **project_id** (*str*) – The project identifier.
- **cluster_config_dict** (*dict*) – A dictionary containing cluster configuration details, including 'ingress_class' and 'ingress_resolver_email'.

Returns

dict – A dictionary containing the namespace, repository URL, chart version, path to the values YAML file, release name, and chart name.

MAIA.maia_core.**create_gpu_booking_values**(*config_folder*, *project_id*)

Creates and writes GPU booking Helm chart values to a YAML file for a given project and cluster configuration.

This function prepares a dictionary of values required for deploying the GPU booking Helm chart, including image repositories, API URLs, and authentication tokens. It writes these values to a YAML file in a structured directory under the specified configuration folder.

Parameters

- **config_folder** (*str or Path*) – The base directory where the configuration files should be stored.
- **project_id** (*str*) – The unique identifier for the project, used to create a subdirectory.

Returns

dict –

A dictionary containing:

- "namespace": The Kubernetes namespace for the deployment.
- "repo": The Helm chart repository URL.
- "version": The Helm chart version.
- "values": The path to the generated YAML values file.
- "release": The Helm release name.
- "chart": The Helm chart name.

Raises

- **KeyError** – If required keys are missing from *cluster_config_dict*.
- **OSError** – If there is an error creating directories or writing the YAML file.

MAIA.maia_core.**create_gpu_operator_values**(*config_folder*, *project_id*, *cluster_config_dict*)

Creates GPU operator values configuration for a Kubernetes cluster and writes it to a YAML file.

Parameters

- **config_folder** (*str*) – The folder path where the configuration will be saved.
- **project_id** (*str*) – The project identifier used to create a unique directory for the configuration.

- **cluster_config_dict** (*dict*) – A dictionary containing cluster configuration details, including the Kubernetes distribution.

Returns

dict – A dictionary containing the namespace, repository URL, chart version, path to the values file, release name, and chart name.

MAIA.maia_core.**create_ingress_nginx_values**(*config_folder*, *project_id*)

Creates and writes the ingress-nginx Helm chart values to a YAML file.

Parameters

- **config_folder** (*str*) – The path to the configuration folder.
- **project_id** (*str*) – The unique identifier for the project.

Returns

dict – A dictionary containing the following keys: - namespace (str): The namespace for the ingress-nginx. - repo (str): The repository URL for the ingress-nginx chart. - version (str): The version of the ingress-nginx chart. - values (str): The file path to the generated ingress-nginx values YAML file. - release (str): The release name for the ingress-nginx chart. - chart (str): The name of the ingress-nginx chart.

MAIA.maia_core.**create_kubeflow_values**(*config_folder*, *project_id*, *cluster_config_dict*)

Creates and writes Kubeflow values to a YAML file and returns a dictionary with deployment details.

Parameters

- **config_folder** (*str*) – The path to the configuration folder.
- **project_id** (*str*) – The unique identifier for the project.
- **cluster_config_dict** (*dict*) –

  A dictionary containing cluster configuration details, including:

  - domain (str): The domain name for the cluster.
  - ingress_class (str): The ingress class to be used (e.g., "maia-core-traefik" or "nginx").
  - traefik_resolver (str, optional): The Traefik resolver to be used if ingress_class is "maia-core-traefik".

MAIA.maia_core.**create_local_path_values**(*config_folder*, *project_id*, *cluster_config_dict*)

Creates and saves the local path values for a Kubernetes cluster. This function generates a dictionary of local path values based on the provided configuration folder, project ID, and cluster configuration dictionary. It retrieves the internal IP addresses of the nodes in the Kubernetes cluster and uses them to configure the MetalLB addresses. The generated values are saved to a YAML file.

MAIA.maia_core.**create_loginapp_values**(*config_folder*, *project_id*, *cluster_config_dict*)

Creates and writes the loginapp values configuration file for a given project and cluster configuration.

Parameters

- **config_folder** (*str*) – The base directory where the configuration files will be stored.
- **project_id** (*str*) – The unique identifier for the project.
- **cluster_config_dict** (*dict*) –

  A dictionary containing cluster configuration details, including:

  - keycloak.client_secret (str): The client secret for Keycloak.
  - domain (str): The domain name for the cluster.

- ingress_class (str): The ingress class to be used (e.g., "maia-core-traefik" or "nginx").
- traefik_resolver (str, optional): The Traefik resolver to be used if ingress_class is "maia-core-traefik".

Returns

dict – A dictionary containing the namespace, release name, chart name, repository URL, chart version, and the path to the generated values file.

Raises

- **KeyError** – If required keys are missing from the cluster_config_dict.
- **OSError** – If there is an error creating directories or writing the configuration file.

MAIA.maia_core.**create_loki_values**(*config_folder*, *project_id*)

Creates and writes Loki values configuration to a YAML file and returns deployment details.

Parameters

- **config_folder** (*str*) – The path to the configuration folder.
- **project_id** (*str*) – The project identifier.

Returns

dict – A dictionary containing deployment details including namespace, repo URL, chart version, values file path, release name, and chart name.

MAIA.maia_core.**create_metallb_values**(*config_folder*, *project_id*)

Creates and writes MetalLB Helm chart values to a YAML file and returns a dictionary with deployment details.

Parameters

- **config_folder** (*str*) – The path to the configuration folder where the YAML file will be created.
- **project_id** (*str*) – The project identifier used to create a unique directory and release name.

Returns

dict – A dictionary containing the following keys: - namespace (str): The Kubernetes namespace for MetalLB. - repo (str): The URL of the MetalLB Helm chart repository. - version (str): The version of the MetalLB Helm chart. - values (str): The file path to the generated YAML values file. - release (str): The release name for the MetalLB deployment. - chart (str): The name of the MetalLB Helm chart.

MAIA.maia_core.**create_metrics_server_values**(*config_folder*, *project_id*)

Creates and writes Metrics server values to a YAML file.

Parameters

- **config_folder** (*str*) – The path to the configuration folder.
- **project_id** (*str*) – The unique identifier for the project.

Returns

dict – A dictionary containing the namespace, repository URL, chart version, path to the values file, release name, and chart name.

MAIA.maia_core.**create_minio_operator_values**(*config_folder*, *project_id*)

Creates and writes MinIO operator values to a YAML file and returns a dictionary with deployment details.

Parameters

- **config_folder** (*str*) – The path to the configuration folder.
- **project_id** (*str*) – The unique identifier for the project.

Returns

dict – A dictionary containing the namespace, release name, chart name, repository URL, chart version, and the path to the generated YAML values file.

MAIA.maia_core.**create_nfs_server_provisioner_values**(*config_folder*, *project_id*, *cluster_config_dict*)

Creates and writes the NFS server provisioner Helm chart values to a YAML file.

Parameters

- **config_folder** (*str*) – The path to the configuration folder.
- **project_id** (*str*) – The unique identifier for the project.
- **cluster_config_dict** (*dict*) – A dictionary containing cluster configuration details, including the NFS server and path.

Returns

dict – A dictionary containing the following keys: - namespace (str): The namespace for the NFS server provisioner. - repo (str): The repository URL for the NFS server provisioner chart. - version (str): The version of the NFS server provisioner chart. - values (str): The file path to the generated NFS server provisioner values YAML file. - release (str): The release name for the NFS server provisioner chart. - chart (str): The name of the NFS server provisioner chart.

MAIA.maia_core.**create_prometheus_values**(*config_folder*, *project_id*, *cluster_config_dict*)

Generates Prometheus values configuration for a Kubernetes cluster and writes it to a YAML file.

Parameters

- **config_folder** (*str*) – The folder where the configuration files will be stored.
- **project_id** (*str*) – The project identifier.
- **cluster_config_dict** (*dict*) – Dictionary containing cluster configuration details.

Returns

dict – A dictionary containing the namespace, repository URL, chart version, path to the values file, release name, and chart name.

MAIA.maia_core.**create_tempo_values**(*config_folder*, *project_id*)

Creates a set of tempo values and writes them to a YAML file in the specified configuration folder.

Parameters

- **config_folder** (*str*) – The path to the configuration folder where the tempo values will be stored.
- **project_id** (*str*) – The project identifier used to create a subdirectory and name the YAML file.

Returns

dict – A dictionary containing the following keys: - namespace (str): The namespace for the tempo values. - repo (str): The repository URL for the Helm chart. - version (str): The version of the Helm chart. - values (str): The path to the generated tempo values YAML file. - release (str): The release name for the Helm chart. - chart (str): The name of the Helm chart.

MAIA.maia_core.**create_traefik_values**(*config_folder*, *project_id*, *cluster_config_dict*)

Creates the Traefik values configuration file for a given project and cluster configuration.

Parameters

- **config_folder** (*str*) – The path to the configuration folder.
- **project_id** (*str*) – The unique identifier for the project.
- **cluster_config_dict** (*dict*) – A dictionary containing the cluster configuration.

Returns

dict – A dictionary containing the namespace, repository URL, chart version, values file path, release name, and chart name for the Traefik deployment.

Raises

OSError – If there is an error creating the directory or writing the file.

MAIA.maia_core.**sync_argocd_app**(*project_name*, *app_name*, *chart_version*, *argo_cd_host*, *password*)

2.1.7 MAIA.maia_docker_images module

MAIA.maia_docker_images.**deploy_maia_kaniko**(*namespace*, *config_folder*, *cluster_config_dict*, *release_name*, *project_id*, *registry_url*, *registry_secret_name*, *image_name*, *image_tag*, *subpath*, *build_args=None*, *registry_credentials=None*, *git_repo_url=None*)

Deploys a Kaniko job for building and pushing Docker images to a specified registry.

Parameters

- **namespace** (*str*) – The Kubernetes namespace where the Kaniko job will be deployed.
- **config_folder** (*str*) – The folder path where the configuration files will be stored.
- **cluster_config_dict** (*dict*) – Dictionary containing cluster configuration details, including storage class.
- **release_name** (*str*) – The release name for the Kaniko job.
- **project_id** (*str*) – The project identifier.
- **registry_url** (*str*) – The URL of the Docker registry where the image will be pushed.
- **registry_secret_name** (*str*) – The name of the Kubernetes secret for accessing the Docker registry.
- **image_name** (*str*) – The name of the Docker image to be built.
- **image_tag** (*str*) – The tag of the Docker image to be built.
- **subpath** (*str*) – The subpath of the repository where the Dockerfile is located.
- **build_args** (*list, optional*) – A list of build arguments to be passed to the Kaniko job.
- **registry_credentials** (*dict, optional*) – A dictionary containing registry credentials with keys 'username', 'password', 'server', and 'email'.
- **custom_git_repo_url** (*str, optional*) – The URL of the Git repository where the Dockerfile is located.

Returns

dict – A dictionary containing deployment details including namespace, release name, chart name, repo URL, chart version, and values file path.

2.1.8 MAIA.maia_fn module

MAIA.maia_fn.**convert_username_to_jupyterhub_username**(*username*)

Convert a username to a JupyterHub-compatible username.

Parameters
username (*str*) – The original username.

Returns
str – The JupyterHub-compatible username.

MAIA.maia_fn.**copy_certificate_authority_secret**(*namespace*, *source_secret_name='kubernetes-ca'*, *target_secret_name='kubernetes-ca'*, *source_namespace='cert-manager'*, *opaque=False*, *cert_name='tls.crt'*, *key_name='tls.key'*)

MAIA.maia_fn.**create_config_map_from_data**(*data*, *config_map_name*, *namespace*, *kubeconfig_dict*, *data_key='values.yaml'*)

Create a ConfigMap on a Kubernetes Cluster.

Parameters

- **data** (*str*) – String containing the content of the ConfigMap to dump.
- **config_map_name** (*str*) – ConfigMap name.
- **namespace** (*str*) – Namespace where to create the ConfigMap.
- **data_key** (*str, optional*) – Value to use as the filename for the content in the ConfigMap.
- **kubeconfig_dict** (*dict*) – Kube Configuration dictionary for Kubernetes cluster authentication.

MAIA.maia_fn.**create_filebrowser_values**(*namespace_config*, *cluster_config*, *config_folder*, *mlflow_configs=None*, *mount_cifs=True*)

Create and write configuration values for deploying the MAIA Filebrowser Helm chart. This function generates a dictionary of configuration values required to deploy the MAIA Filebrowser application in a Kubernetes namespace. It handles image configuration, environment variables, volume mounts, CIFS volume setup, and ingress settings for both NGINX and Traefik ingress controllers. The resulting configuration is written to a YAML file in the specified config folder.

Parameters

- **namespace_config** (*dict*) – Dictionary containing namespace-specific configuration, including group ID, subdomain, and users.
- **cluster_config** (*dict*) – Dictionary containing cluster-specific configuration, such as docker server, image pull secrets, domain, and optional ingress settings.
- **config_folder** (*str or Path*) – Path to the folder where the generated configuration YAML file will be saved.
- **mlflow_configs** (*dict, optional*) – Optional dictionary containing MLflow configuration, specifically the base64-encoded 'mlflow_password'. If not provided, a new human-memorable password is generated.

Returns

dict –

A dictionary containing:

- 'namespace': The Kubernetes namespace for deployment.
- 'release': The Helm release name.
- 'chart': The Helm chart name.
- 'repo': The Helm chart repository URL.
- 'version': The Helm chart version.
- 'values': Path to the generated YAML values file.

Notes

- The function expects certain helper functions and environment variables to be available, such as *generate_human_memorable_password*, *convert_username_to_jupyterhub_username*, and *OmegaConf*.
- The CIFS server address is read from the 'CIFS_SERVER' environment variable.

MAIA.maia_fn.**create_maia_namespace_values**(*namespace_config*, *cluster_config*, *config_folder*, *minio_configs=None*, *mlflow_configs=None*)

Create MAIA namespace values for deployment.

Parameters

- **namespace_config** (*dict*) – Configuration for the namespace, including group ID and users.
- **cluster_config** (*dict*) – Configuration for the cluster, including SSH port type, port range, and storage class.
- **config_folder** (*str*) – Path to the folder where configuration files will be saved.
- **minio_configs** (*dict, optional*) – Configuration for MinIO, including access keys and console keys. Defaults to None.
- **mlflow_configs** (*dict, optional*) – Configuration for MLflow, including user and password. Defaults to None.

Returns

dict – A dictionary containing the namespace, release name, chart name, repository URL, chart version, and the path to the generated values file.

MAIA.maia_fn.**create_nvflare_dashboard_values**(*namespace_config*, *cluster_config*, *config_folder*)

Create and write configuration values for deploying the MAIA NVFlare Dashboard Helm chart. This function generates a dictionary of configuration values required to deploy the MAIA NVFlare Dashboard application in a Kubernetes namespace. It handles image configuration, environment variables, volume mounts, CIFS volume setup, and ingress settings for both NGINX and Traefik ingress controllers. The resulting configuration is written to a YAML file in the specified config folder.

MAIA.maia_fn.**deploy_kubeflow_project**(*cluster_config*, *user_config*, *config_folder*, *project_config_dict=None*, *minimal=True*)

Deploy a Kubeflow project using the provided configuration. :type cluster_config: :param cluster_config: Dictionary containing the cluster configuration. :type cluster_config: dict :type user_config: :param user_config: Dictionary containing the user configuration. :type user_config: dict :type config_folder: :param config_folder: Path to the configuration folder. :type config_folder: str or Path :type project_config_dict: :param project_config_dict: Dictionary containing the project configuration. :type project_config_dict: dict, optional

Returns

dict – A dictionary containing deployment details such as namespace, release, chart, repo, version, and values file path.

MAIA.maia_fn.**deploy_mlflow**(*cluster_config*, *user_config*, *config_folder*, *mysql_config=None*, *minio_config=None*)

Deploy an MLflow instance on a Kubernetes cluster using Helm.

Parameters

- **cluster_config** (*dict*) – Configuration dictionary for the Kubernetes cluster.
- **user_config** (*dict*) – Configuration dictionary for the user, including group_ID.
- **config_folder** (*str*) – Path to the folder where configuration files will be stored.
- **mysql_config** (*dict, optional*) – Configuration dictionary for MySQL, including mysql_user and mysql_password. Defaults to None.
- **minio_config** (*dict, optional*) – Configuration dictionary for MinIO, including console_access_key and console_secret_key. Defaults to None.

Returns

dict – A dictionary containing deployment details such as namespace, release name, chart name, repository URL, chart version, and path to the values file.

MAIA.maia_fn.**deploy_mysql**(*cluster_config*, *user_config*, *config_folder*, *mysql_configs*)

Deploy a MySQL instance on a Kubernetes cluster using Helm.

Parameters

- **cluster_config** (*dict*) – Configuration dictionary for the cluster, including storage class.
- **user_config** (*dict*) – Configuration dictionary for the user, including group ID.
- **config_folder** (*str*) – Path to the folder where configuration files will be stored.
- **mysql_configs** (*dict*) – Configuration dictionary for MySQL, including user, password, and other settings.

Returns

dict – A dictionary containing deployment details such as namespace, release name, chart name, repository URL, version, and values file path.

MAIA.maia_fn.**deploy_oauth2_proxy**(*cluster_config*, *user_config*, *config_folder=None*)

Deploy an OAuth2 Proxy using the provided cluster and user configurations.

Parameters

- **cluster_config** (*dict*) –

  Configuration dictionary for the cluster. Expected keys include:

  - "keycloak": A dictionary with "issuer_url", "client_id", and "client_secret".
  - "domain": The domain name for the cluster.
  - "url_type": The type of URL, either "subpath" or other.
  - "storage_class": The storage class for Redis.
  - "nginx_cluster_issuer" (optional): The cluster issuer for NGINX.
  - "traefik_resolver" (optional): The resolver for Traefik.

- **user_config** (*dict*) –

  Configuration dictionary for the user. Expected keys include:

  - "group_ID": The group ID for the user.

- ”group_subdomain”: The subdomain for the user’s group.

- **config_folder** (*str, optional*) – The folder path where the configuration files will be saved. Defaults to None.

Returns

dict –

A dictionary containing deployment details:

- ”namespace”: The namespace for the deployment.
- ”release”: The release name for the deployment.
- ”chart”: The chart name for the deployment.
- ”repo”: The repository URL for the chart.
- ”version”: The chart version.
- ”values”: The path to the generated values YAML file.

MAIA.maia_fn.**deploy_orthanc**(*cluster_config*, *user_config*, *config_folder*, *project_config_dict=None*)

Deploys Orthanc using the provided configuration. :type cluster_config: :param cluster_config: Dictionary containing the cluster configuration. :type cluster_config: dict :type user_config: :param user_config: Dictionary containing the user configuration. :type user_config: dict :type config_folder: :param config_folder: Path to the configuration folder. :type config_folder: str or Path

Returns

dict – A dictionary containing deployment details such as namespace, release, chart, repo, version, and values file path.

MAIA.maia_fn.**edit_orthanc_configuration**(*orthanc_config_template*, *orthanc_edit_dict*)

MAIA.maia_fn.**encode_docker_registry_secret**(*registry_server*, *registry_username*, *registry_password*)

Encode Docker registry credentials into a base64-encoded string.

Parameters

- **registry_server** (*str*) – The Docker registry server.
- **registry_username** (*str*) – The Docker registry username.
- **registry_password** (*str*) – The Docker registry password.

Returns

str – The base64-encoded Docker registry credentials.

MAIA.maia_fn.**generate_human_memorable_password**(*length=12*)

MAIA.maia_fn.**generate_minio_configs**(*namespace*, *project_config_dict=None*)

Generate configuration settings for MinIO.

Parameters

- **namespace** (*int or str*) – The unique identifier for the project.
- **project_config_dict** (*dict, optional*) – A dictionary containing the custom configuration for the MinIO.

Returns

dict – A dictionary with the following keys: - access_key (str): The access key for MinIO. - secret_key (str): A randomly generated secret key for MinIO. - console_access_key (str): A

base64 encoded access key for console access. - console_secret_key (str): A base64 encoded secret key for console access.

MAIA.maia_fn.**generate_mlflow_configs**(*namespace*, *project_config_dict=None*)

Generate MLflow configuration dictionary with encoded user and password.

Parameters

- **namespace** (*str*) – The namespace to be encoded as the MLflow user.
- **project_config_dict** (*dict, optional*) – A dictionary containing the custom configuration for the MLflow.

Returns

dict – A dictionary containing the encoded MLflow user and password.

MAIA.maia_fn.**generate_mysql_configs**(*namespace*, *project_config_dict=None*)

Generate MySQL configuration dictionary.

Parameters

- **namespace** (*str*) – The namespace to be used as the MySQL user.
- **project_config_dict** (*dict, optional*) – A dictionary containing the custom configuration for the MySQL.

Returns

dict – A dictionary containing MySQL user and password.

MAIA.maia_fn.**generate_nvflare_dashboard_configs**(*project_id*, *project_config_dict=None*)

Generates NVFlare Dashboard configuration dictionary.

MAIA.maia_fn.**generate_orthanc_configs**(*project_id*, *project_config_dict=None*)

Generates Orthanc configuration dictionary.

MAIA.maia_fn.**generate_random_password**(*length=12*)

MAIA.maia_fn.**get_minio_config_if_exists**(*project_id*)

Retrieves MinIO configuration if it exists for the given project ID. This function loads the Kubernetes configuration from the environment, accesses the Kubernetes API to list secrets in the specified namespace, and extracts MinIO-related configuration from the secrets.

Parameters

project_id (*str*) – The ID of the project for which to retrieve the MinIO configuration.

Returns

dict – A dictionary containing MinIO configuration keys and their corresponding values. The dictionary may contain the following keys: - “access_key”: The default access key (always “admin”). - “console_access_key”: The console access key, if found. - “console_secret_key”: The console secret key, if found. - “secret_key”: The MinIO root password, if found.

MAIA.maia_fn.**get_mlflow_config_if_exists**(*project_id*)

Retrieve MLflow configuration from Kubernetes secrets if they exist.

Parameters

project_id (*str*) – The ID of the project for which to retrieve the MLflow configuration. This ID is used to locate the corresponding Kubernetes namespace and secrets.

Returns

dict – A dictionary containing the MLflow configuration with keys “mlflow_user” and “mlflow_password” if they exist in the Kubernetes secrets. If the secrets are not found, an empty dictionary is returned.

Raises

- **KeyError** – If the "KUBECONFIG" environment variable is not set.
- **yaml.YAMLError** – If there is an error parsing the Kubernetes configuration file.
- **kubernetes.client.exceptions.ApiException** – If there is an error communicating with the Kubernetes API.

MAIA.maia_fn.**get_mysql_config_if_exists**(*project_id*)

Retrieves MySQL configuration from Kubernetes environment variables if they exist.

Parameters

project_id (*str*) – The ID of the project for which to retrieve the MySQL configuration. This ID is used to identify the namespace and the MySQL deployment within the Kubernetes cluster.

Returns

dict – A dictionary containing the MySQL user and password if they exist in the environment variables of the MySQL deployment. The dictionary keys are: - "mysql_user": The MySQL user. - "mysql_password": The MySQL password.

Notes

This function assumes that the Kubernetes configuration file is specified in the environment variable "KUBECONFIG" and that the MySQL deployment name starts with the project ID followed by "-mysql-mkg".

MAIA.maia_fn.**get_nvflare_dashboard_config_if_exists**(*project_id*)

Retrieves NVFlare Dashboard configuration from Kubernetes environment variables if they exist.

MAIA.maia_fn.**get_orthanc_config_if_exists**(*project_id*)

Retrieves Orthanc configuration from Kubernetes environment variables if they exist.

MAIA.maia_fn.**get_ssh_port_dict**(*port_type*, *namespace*, *port_range*, *maia_metallb_ip=None*)

Retrieve a dictionary of used SSH ports for services in a Kubernetes cluster.

Parameters

- **port_type** (*str*) – The type of port to check ('LoadBalancer' or 'NodePort').
- **namespace** (*str*) – The namespace to filter services by.
- **port_range** (*tuple*) – A tuple specifying the range of ports to check (start, end).
- **maia_metallb_ip** (*str, optional*) – The IP address of the MetalLB load balancer (default is None).

Returns

list of dict – A list of dictionaries with service names as keys and their corresponding used SSH ports as values. Returns None if an exception occurs.

MAIA.maia_fn.**get_ssh_ports**(*n_requested_ports*, *port_type*, *ip_range*, *maia_metallb_ip=None*)

Retrieve a list of available SSH ports based on the specified criteria.

Parameters

- **n_requested_ports** (*int*) – The number of SSH ports requested.
- **port_type** (*str*) – The type of port to search for ('LoadBalancer' or 'NodePort').
- **ip_range** (*tuple*) – A tuple specifying the range of IPs to search within (start, end).
- **maia_metallb_ip** (*str, optional*) – The specific IP address to match for 'LoadBalancer' type. Defaults to None.

Returns

- *list* – A list of available SSH ports that meet the specified criteria.
- *None* – If an error occurs during the process.

MAIA.maia_fn.**gpu_list_from_nodes**()

Retrieves a list of GPUs from the nodes in a Kubernetes cluster.

This function loads the Kubernetes configuration from the environment, initializes the Kubernetes client, and retrieves the list of nodes. It then checks each node to see if it is ready and has GPU labels, and constructs a dictionary with the node names as keys and a list containing the GPU product and count as values.

Returns

dict – A dictionary where the keys are node names and the values are lists containing the GPU product and GPU count.

2.1.9 MAIA.maia_k8s_distros module

MAIA.maia_k8s_distros.**get_api_port**(*k8s_distribution*)

MAIA.maia_k8s_distros.**get_gpu_operator_toolkit**(*k8s_distribution*)

MAIA.maia_k8s_distros.**get_ingress_class**(*k8s_distribution*)

MAIA.maia_k8s_distros.**get_storage_class**(*k8s_distribution*)

2.1.10 MAIA.notifications module

MAIA.notifications.**confirm_request_registration_for_group**(*group_name*, *user_email*, *support_link*, *dashboard_url*, *smtp_sender_email*, *smtp_server*, *smtp_port*, *smtp_password*)

MAIA.notifications.**confirm_request_registration_to_project**(*project_name*, *user_email*, *support_link*, *dashboard_url*, *smtp_sender_email*, *smtp_server*, *smtp_port*, *smtp_password*)

MAIA.notifications.**send_email_approved_project_registration**(*project_name*, *project_owner*, *support_link*, *dashboard_url*, *smtp_sender_email*, *smtp_server*, *smtp_port*, *smtp_password*)

MAIA.notifications.**send_email_approved_registration_email**(*email*, *temp_password*, *login_url*, *smtp_sender_email*, *smtp_server*, *smtp_port*, *smtp_password*)

MAIA.notifications.**send_email_user_registration_to_group**(*project_name*, *user_email*, *support_link*, *dashboard_url*, *smtp_sender_email*, *smtp_server*, *smtp_port*, *smtp_password*)

2.1.11 MAIA.versions module

MAIA.versions.**define_docker_image_versions**()

MAIA.versions.**define_maia_admin_versions**()

MAIA.versions.**define_maia_core_versions**()

MAIA.versions.**define_maia_docker_versions**()

MAIA.versions.**define_maia_project_versions**()

2.2 Module contents

CHAPTER

THREE

SCRIPTS

3.1 MAIA Scripts

3.1.1 MAIA_Install script

Script to install MAIA on a Kubernetes cluster. This script: 1. Installs the MAIA Ansible collection 2. Runs MAIA_Configure_Installation.sh to configure the installation 3. Executes the MAIA installation playbooks in sequence

```
usage: MAIA_Install [-h] --config-folder CONFIG_FOLDER
                    [--ansible-collection-path ANSIBLE_COLLECTION_PATH]
                    [--inventory-path INVENTORY_PATH] [--skip-configure]
                    [--configure-local-host] [--configure-no-prompt]
                    [--steps STEPS [STEPS ...]] [--upgrade-maia-core]
                    [--upgrade-maia-admin] [--upgrade-maia-dashboard] [-v]
```

Named Arguments

--config-folder Configuration folder where MAIA configuration files will be stored.

--ansible-collection-path Path to the MAIA Ansible collection directory. Default: git+https://github.com/minnelab/MAIA.git#/ansible/MAIA/Installation

Default: "git+https://github.com/minnelab/MAIA.git#/ansible/MAIA/Installation"

--inventory-path Path where the inventory file will be created. If not provided, defaults to <config-folder>/inventory

--skip-configure Skip running MAIA_Configure_Installation.sh. Use this if configuration is already done.

Default: False

--configure-local-host Configure the local host with the subdomain and IP address for self-signed certificates.

Default: False

--configure-no-prompt Run MAIA_Configure_Installation.sh without prompts (requires env.json to exist).

Default: False

--steps Steps to run. Default: None. If provided, only the specified steps will be run, overriding the steps in the config.yaml file.

--upgrade-maia-core Upgrade MAIA core. Default: False. If provided, the MAIA core will be upgraded.

Default: False

--upgrade-maia-admin Upgrade MAIA admin. Default: False. If provided, the MAIA admin will be upgraded.

Default: False

--upgrade-maia-dashboard Upgrade MAIA dashboard. Default: False. If provided, the MAIA dashboard will be upgraded.

Default: False

-v, --version show program's version number and exit

Example call:

```
MAIA_Install --config-folder /path/to/config
```

3.1.2 MAIA_build_images script

Script to Build MAIA Docker Images using Kaniko. The specific MAIA configuration is specified by setting the corresponding `--maia-config-file`, and the cluster configuration is specified by setting the corresponding `--cluster-config`.

```
usage: MAIA_build_images [-h] --cluster-config CLUSTER_CONFIG --config-folder
                         CONFIG_FOLDER [--registry-path REGISTRY_PATH]
                         --project-id PROJECT_ID
                         [--cluster-address CLUSTER_ADDRESS]
                         [--build-custom-images BUILD_CUSTOM_IMAGES] [-v]
```

Named Arguments

--cluster-config YAML configuration file used to extract the cluster configuration.

--config-folder Configuration Folder where to locate (and temporarily store) the MAIA configuration files.

--registry-path Optional path to the Docker registry. If not provided, the default is empty.

Default: ""

--project-id Project ID to use for ArgoCD. This is used to identify the project in the cluster.

--cluster-address Optional address of the cluster. If not provided, the default is https://kubernetes.default.svc

Default: "https://kubernetes.default.svc"

--build-custom-images Build the custom images from the given YAML file.

-v, --version show program's version number and exit

Example call:

```
MAIA_build_images --cluster-config /PATH/TO/cluster_config.yaml --config-folder /PATH/TO/
↪config_folder
```

3.1.3 MAIA_change_keycloak_client_secret script

Change Keycloak Client Secret

```
usage: MAIA_change_keycloak_client_secret [-h] --client_secret CLIENT_SECRET
                                          --realm_file REALM_FILE
```

Named Arguments

--client_secret	The client secret to change
--realm_file	The realm file to change

Example call:

```
MAIA_change_keycloak_client_secret --client_secret <client_secret> --realm_file <realm_
↪file>
```

3.1.4 MAIA_configure_keycloak script

Configure Keycloak

```
usage: MAIA_configure_keycloak [-h] --client_secret CLIENT_SECRET --server_url
                               SERVER_URL --admin_email ADMIN_EMAIL
                               [--admin_password ADMIN_PASSWORD]
                               [--admin_group_id ADMIN_GROUP_ID]
```

Named Arguments

--client_secret	The client secret to use
--server_url	The server URL to configure
--admin_email	The admin email to use
--admin_password	The admin password to use Default: “admin”
--admin_group_id	The admin group ID to use Default: “admin”

Example call:

```
MAIA_configure_keycloak --client_secret <client_secret> --server_url <server_url>
```

3.1.5 MAIA_create_JupyterHub_config script

Script to deploy the JupyterHub helm chart to a Kubernetes cluster. The target cluster is specified by setting the correspondin `--cluster--config-file`, while the namespace-related configuration is specified with `--form`.

```
usage: MAIA_create_JupyterHub_config [-h] --form FORM --cluster-config-file
                                     CLUSTER_CONFIG_FILE [-v]
```

Named Arguments

--form	YAML configuration file used to extract the namespace configuration.
--cluster-config-file	YAML configuration file used to extract the cluster configuration.
-v, --version	show program's version number and exit

Example call:

```
MAIA_create_JupyterHub_config  --form /PATH/TO/form.yaml --cluster-config-file /PATH/TO/
↪cluster.yaml
```

3.1.6 MAIA_deploy_helm_chart script

Script to deploy the MAIAKubeGate helm chart to a Kubernetes cluster. The target cluster is specified by setting `KUBECONFIG` as an environment variable, while the configuration file for the chart is specified with `config_file`.

```
usage: MAIA_deploy_helm_chart [-h] --config-file CONFIG_FILE
                              [--print-helm-values-only PRINT_HELM_VALUES_ONLY]
                              [-v]
```

Named Arguments

--config-file	JSON configuration file used to generate the custom values used in the Helm chart.
--print-helm-values-only	Flag to save the generated Helm values on the specified file and exit.
-v, --version	show program's version number and exit

Example call:

```
MAIA_deploy_helm_chart  --config-file /PATH/TO/config_file.json
```

3.1.7 MAIA_deploy_project script

Script to deploy a MAIA Project through the MAIA Dashboard API.

```
usage: MAIA_deploy_project [-h] --project-config-file PROJECT_CONFIG_FILE
```

Named Arguments

--project-config-file JSON configuration file used to extract the project configuration.

Example call:

```
MAIA_deploy_project  --project-config-file /PATH/TO/project_config_file.json
```

3.1.8 MAIA_install_admin_toolkit script

Script to Install MAIA Admin Toolkit to a Kubernetes cluster from ArgoCD. The specific MAIA configuration is specified by setting the corresponding `--maia-config-file`, and the cluster configuration is specified by setting the corresponding `--cluster-config`.

```
usage: MAIA_install_admin_toolkit [-h] --cluster-config CLUSTER_CONFIG
                                  --config-folder CONFIG_FOLDER [-v]
```

Named Arguments

--cluster-config YAML configuration file used to extract the cluster configuration.

--config-folder Configuration Folder where to locate (and temporarily store) the MAIA configuration files.

-v, --version show program's version number and exit

Example call:

```
MAIA_install_admin_toolkit --cluster-config /PATH/TO/cluster_config.yaml --config-folder␣
↪/PATH/TO/config_folder
```

3.1.9 MAIA_install_core_toolkit script

Script to Install MAIA Core Toolkit to a Kubernetes cluster from ArgoCD. The specific MAIA configuration is specified by setting the corresponding `--maia-config-file`, and the cluster configuration is specified by setting the corresponding `--cluster-config`.

```
usage: MAIA_install_core_toolkit [-h] --cluster-config CLUSTER_CONFIG
                                 --config-folder CONFIG_FOLDER [-v]
```

Named Arguments

--cluster-config	YAML configuration file used to extract the cluster configuration.
--config-folder	Configuration Folder where to locate (and temporarily store) the MAIA configuration files.
-v, --version	show program's version number and exit

Example call:

```
MAIA_install_core_toolkit --cluster-config /PATH/TO/cluster_config.yaml --config-folder /
↪PATH/TO/config_folder
```

3.1.10 MAIA_install_project_toolkit script

Script to deploy a MAIA Project Toolkit to a Kubernetes cluster. The target cluster is specified by setting the corresponding `--cluster-config`, while the project-related configuration is specified with `--project-config-file`. The necessary MAIA Configuration files should be found in `--config-folder`.

```
usage: MAIA_install_project_toolkit [-h] --project-config-file
                                    PROJECT_CONFIG_FILE --cluster-config
                                    CLUSTER_CONFIG --config-folder
                                    CONFIG_FOLDER [--minimal MINIMAL]
                                    [--no-argocd] [-v]
```

Named Arguments

--project-config-file	YAML configuration file used to extract the project configuration.
--cluster-config	YAML configuration file used to extract the cluster configuration.
--config-folder	Configuration Folder where to locate (and temporarily store) the MAIA configuration files.
--minimal	Optional flag to only deploy JupyterHub in the MAIA namespace.
--no-argocd	Do not deploy with ArgoCD. Default: False
-v, --version	show program's version number and exit

Example call:

```
MAIA_install_project_toolkit --project-config-file /PATH/TO/form.yaml --cluster-config /
↪PATH/TO/cluster.yaml
--config-folder /PATH/TO/config_folder
```

3.1.11 MAIA_send_all_user_email script

Send reminder email to all MAIA users

```
usage: MAIA_send_all_user_email [-h] [--email-list EMAIL_LIST]
```

Named Arguments

--email-list List of emails to send the reminder email to

Example call:

```
MAIA_send_all_user_email --email-list <email_list>
```

3.1.12 MAIA_send_welcome_user_mail script

Send welcome email to new MAIA users

```
usage: MAIA_send_welcome_user_mail [-h] --email EMAIL --url URL [-v]
```

Named Arguments

--email Recipient email address

--url MAIA platform URL

-v, --version show program's version number and exit

Example call:

```
MAIA_send_welcome_user_mail --email <recipient_email> --url <maia_platform_url>
```

CHAPTER

FOUR

TUTORIALS

4.1 Build and Push Docker Image

This tutorial will guide you through the process of building and pushing a Docker image to the MAIA Docker registry. The Docker image will be remotely built on the cloud and pushed to the MAIA Docker registry, which is hosted at `registry.cloud.cbh.kth.se`. The Docker Context (Dockerfile and other necessary files) need to be located in a Git repository, which will be accessed by KANIKO to build the Docker image.

The configuration parameters for building and pushing the Docker image are set in a JSON configuration file, following the conventions described below.

```
git_path: "<GIT_PATH>"              # The path to the Git repository where the Docker␣
↪Context is located
git_subpath: "<GIT_PATH>"           # The subpath to the Docker Context in the Git␣
↪repository
git_username: "<GIT_USERNAME>"    # The username to access the Git repository
git_token: "<GIT_TOKEN>"            # The token to access the Git repository
docker_registry_secret: "<SECRET>" # Kubernetes Secret to Authenticate in the private␣
↪Registry
docker_image: "<IMAGE_NAME>"        # The name of the Docker image to build
buildArgs:                          # Optional build arguments
    OPTIONAL_BUILD_ARG: "<VALUE>"
    OPTIONAL_BUILD_ARG: "<VALUE>"
```

4.1.1 Set Git Specifications

In this example, we will build our custom Jupyter Scipy Image, where it will be possible to set the Jupyter Token as an env variable `PW`. For more details, check the Dockerfile

```
[ ]: %%writefile kaniko_values.yaml

docker_registry_secret: <SECRET>
pvc:
  pvc_type: <STORAGE_CLASS>
  size: "10Gi"

args:
  - "--dockerfile=Dockerfile"
  - "--context=git://github.com/minnelab/MAIA.git"
  - "--context-sub-path=docker/demo"
```

(continues on next page)

(continued from previous page)

```
  - "--destination=registry.cloud.cbh.kth.se/maia/jupyter-demo:1.2"
  - "--cache=true"
  - "--cache-dir=/cache"
  #- "--context=s3://docker-contexts/{{.Values.context_file}}"
```

4.1.2 Install Helm Chart

[ ]:
```
%%bash

helm repo add maia https://minnelab.github.io/MAIA/
helm repo update
export KUBECONFIG=<KUBECONFIG>

#Optional: delete the previous deployment
# helm delete -n maia-admin demo-docker

helm install -n maia-admin demo-docker maia/mkg-kaniko --values kaniko_values.yaml --
↪version 1.0.3
```

4.1.3 Log the Docker Build Job

[ ]:
```
%%bash

export KUBECONFIG=<KUBECONFIG>

kubectl logs -f -n maia-admin job/demo-docker-mkg-kaniko
```

4.2 MAIA User Registration

Instructions to read the MAIA User Registration requests from the MySQL database, register the users in Keycloak, create a namespace in the Kubernetes cluster, and store the user information in Vault.

[ ]:
```
!pip install pymysql pandas python-keycloak kubernetes sqlalchemy itables
```

4.2.1 Login to the MySQL database

The MySQL database contains the information about the users, the groups they belong to, and the namespaces they requested to be registered in. This Database is directly connected to the Sign-Up form in the MAIA website .

[ ]:
```
from sqlalchemy import create_engine

DB_URL = ""
DB_PORT = ""
DB_USERNAME = ""
```

(continues on next page)

(continued from previous page)

```
DB_PASSWORD = ""

engine = create_engine(f"mysql+pymysql://{DB_USERNAME}:{DB_PASSWORD}@{DB_URL}:{DB_PORT}/
↪mysql")
cnx = engine.raw_connection()
```

[ ]:
```
import pandas as pd
```

[ ]:
```
pd.read_sql_query("SHOW TABLES", con=cnx)
```

4.2.2 Read the tables

[ ]:
```
auth_user = pd.read_sql_query("SELECT * FROM auth_user", con=cnx)
auth_group = pd.read_sql_query("SELECT * FROM auth_group", con=cnx)
auth_user_groups = pd.read_sql_query("SELECT * FROM auth_user_groups", con=cnx)
authentication_maiauser = pd.read_sql_query("SELECT * FROM authentication_maiauser",␣
↪con=cnx)
```

[ ]:
```
from itables import init_notebook_mode

init_notebook_mode(all_interactive=True)

from itables import show
```

[ ]:
```
show(auth_user,buttons=["copyHtml5", "csvHtml5", "excelHtml5"],
     layout={"top1": "searchPanes"},
    searchPanes={"layout": "columns-3", "cascadePanes": True},)
```

[ ]:
```
from keycloak import KeycloakAdmin
from keycloak import KeycloakOpenIDConnection


keycloak_connection = KeycloakOpenIDConnection(

                        server_url="",
                        server_url="",
                        username='',
                        password='',
                        realm_name="",
                        client_id="",
                        client_secret_key="",
                        verify=False)


keycloak_admin = KeycloakAdmin(connection=keycloak_connection)
```

[ ]:
```
users_to_register = {}
```

(continues on next page)

(continued from previous page)

```
for user in auth_user.iterrows():
    uid = user[1]['id']
    username = user[1]['username']
    email = user[1]['email']
    user_groups = []
    for keycloack_user in keycloak_admin.get_users():

        if 'email' in keycloack_user:
            if keycloack_user['email'] == email:
                user_keycloack_groups = keycloak_admin.get_user_groups(user_id=keycloack_
↪user['id'])
                for user_keycloack_group in user_keycloack_groups:
                    user_groups.append(user_keycloack_group['name'][len("MAIA:"):])

    if uid in authentication_maiauser['user_ptr_id'].values:
        requested_namespaces = authentication_maiauser[authentication_maiauser['user_ptr_
↪id'] == uid][
            'namespace'].values
        print(
            f"User {username} requested to be registered in MAIA in {authentication_
↪maiauser[authentication_maiauser['user_ptr_id'] == uid]['namespace'].values[0]}␣
↪namespace")
        print(authentication_maiauser[authentication_maiauser['user_ptr_id'] == uid])
        for requested_namespace in requested_namespaces:
            if requested_namespace not in user_groups:
                print(f"User {username} is not in the group {requested_namespace}")
                #print(user_groups)
                if email not in users_to_register:
                    users_to_register[email] = [requested_namespace]
                else:
                    users_to_register[email].append(requested_namespace)
```

4.2.3 Register the users in Keycloak

[ ]:
```
users = keycloak_admin.get_users()
keycloak_admin.get_realms()
usernames = [user['username'] for user in users]

groups = keycloak_admin.get_groups()
```

[ ]:
```
users_to_register
```

[ ]:
```
ids_to_register = {}
for user in users_to_register:
    if user not in usernames:
        print(f"{user} is not registered in Keycloack!")
    else:
        for uid in users:
            if uid['username'] == user:
                ids_to_register[uid['id']]={}
```

(continues on next page)

(continued from previous page)

```
                ids_to_register[uid['id']]['username'] = uid['username']
                ids_to_register[uid['id']]['groups'] = []
```

[ ]:
```
ids_to_register
```

[ ]:
```
existing_groups_to_register = ["MAIA:users"]

for group in groups:
    if group["name"] in existing_groups_to_register:
        for id_to_register in ids_to_register:
            ids_to_register[id_to_register]['groups'].append(group["id"])
```

[ ]:
```
for id_to_register in ids_to_register:
    namespaces = users_to_register[ids_to_register[id_to_register]['username']]
    for namespace in namespaces:
        payload = {
        "name": f"MAIA:{namespace}",
        "path": f"/MAIA:{namespace}",
        "attributes":

        { }
        ,
        "realmRoles": [],
        "clientRoles":

        {}
        ,
        "subGroups": [],
        "access":

        { "view": True, "manage": True, "manageMembership": True }
        }
        try:
            group_id = keycloak_admin.create_group(payload)
            ids_to_register[id_to_register]['groups'].append(group_id)
        except:
            for group in groups:
                if group["name"] == f"MAIA:{namespace}":
                    ids_to_register[id_to_register]['groups'].append(group["id"])
```

[ ]:
```
ids_to_register
```

4.2.4 Add the users to the groups

```
[ ]: groups = keycloak_admin.get_groups()

for uid in ids_to_register:
    for user in users:

        if user["id"] == uid:
            print(f"Registering user {user['username']} with id {uid}")
            for gid in ids_to_register[uid]['groups']:
                print(gid)
                for group in groups:
                    if group["id"] == gid:
                        print(f"Adding user {user['username']} to group {group['name']}")
```

```
[ ]: for uid in ids_to_register:
    for gid in ids_to_register[uid]['groups']:
        keycloak_admin.group_user_add(uid, gid)
```

```
[ ]:
```

4.3 Deploy a Jupyter Environment on a Kubernetes Cluster

In this tutorial we will go through the required steps to deploy a Jupyter environment as a Job on a Kubernetes cluster. We will use the MAIA/mkg Helm Chart to deploy the PyTorch Application. For correctly deploying the application from the Helm Chart, we first need to install Helm:

```
curl https://raw.githubusercontent.com/helm/helm/master/scripts/get-helm-3 | bash
```

We will also need to install the MAIA Helm Chart Repository:

```
helm repo add maia https://github.com/minnelab/MAIA
helm repo update
```

We need also to set the `KUBECONFIG` environment variable to the path of the kubeconfig file of the cluster we want to deploy the application on.

4.3.1 Generate Configuration

Next, we need to create the configuration dictionary, containing the required information for the deployment of the application.

We will use the custom Jupyter Image generated in the *previous tutorial*, requesting a deployment for 10 minutes, exposing the Jupyter port 888 to test-jupyter.app.cloud.cbh.kth.se

```
[ ]: %%writefile jupyter.json
{
  "namespace": "<NAMESPACE>",
  "chart_name": "demo-jupyter-v1",
  "docker_image": "registry.cloud.cbh.kth.se/maia/jupyter-demo",
```

(continues on next page)

(continued from previous page)

```
  "tag": "1.2",
  "allocationTime": "10m",
    "env_variables": {
    "PW": "MAIA"
  },
    "ports": {
    "jupyter": [
      8888
    ]
  },
      "ingress": {
    "host": "test-jupyter.app.cloud.cbh.kth.se",
    "port": "8888",
    "path": "",
    "nginx_issuer": "<ISSUER>"
  },
    "image_pull_secret": "<SECRET>"
}
```

[ ]:
```
%%bash
export KUBECONFIG=<KUBECONFIG>
MAIA_deploy_helm_chart --config-file jupyter.json
```

4.3.2 Check Application

After completing the Chart deployment, the app should be available at: test-jupyter.app.cloud.cbh.kth.se.

Use the password set on the configuration (`PW`) to authenticate.

4.4 Installation

4.4.1 Requirements

Before running the Ansible playbooks, you must configure the installation environment by executing the `MAIA_Configure_Installation.sh` script. This script comes with the `maia-toolkit` package and is required to generate the necessary cluster parameters and configuration files for the Ansible deployment.

Running the Configuration Script

The script can be run with an optional `env.json` file as an argument:

```
MAIA_Configure_Installation.sh [path/to/env.json]
```

If an `env.json` file is provided, the script will load existing values from it. Otherwise, it will prompt for required values.

Required Values

The script requires the following environment variables (will prompt if not set):

- **MAIA_PRIVATE_REGISTRY**: The URL of the private MAIA Docker/Helm registry (e.g., `https://harbor.example.com`). If left empty, the script will set PUBLIC_REGISTRY=1 and skip prompting for REGISTRY_USERNAME and REGISTRY_PASSWORD
- **REGISTRY_USERNAME**: Username for authenticating with a private registry. Only required if `MAIA_PRIVATE_REGISTRY` is set.
- **REGISTRY_PASSWORD**: Password for authenticating with a private registry. Only required if `MAIA_PRIVATE_REGISTRY` is set.
- **CLUSTER_DOMAIN**: The public domain or base domain for the MAIA cluster (e.g., `maia.example.com`).
- **CLUSTER_NAME**: Name to assign to the MAIA Kubernetes cluster (e.g., `maia-cluster`).
- **CONFIG_FOLDER**: Directory path to store MAIA/cluster configuration files (e.g., `/opt/maia/config`).
- **INGRESS_RESOLVER_EMAIL**: Email for Let's Encrypt certificate for ingress (e.g., `admin@example.com`). It can be left empty if using self-signed certificates.
- **K8S_DISTRIBUTION**: Chosen Kubernetes distribution - must be one of: `microk8s`, `rke2`, `k3s`, `k0s`.

Optional Values

- **PUBLIC_REGISTRY**: Set to 1 to skip prompting for `REGISTRY_USERNAME` and `REGISTRY_PASSWORD` (uses public registry)
- **JSON_KEY_PATH**: Path to a JSON file containing Harbor credentials in the format:

```
{
  "username": "username",
  "password": "password"
}
```

- **CIFS Key Generation**: The script will prompt to generate a CIFS public-private key pair for CIFS shared storage support

Generated Files

The script generates all necessary configuration files and stores them in the `CONFIG_FOLDER` directory:

- **env.json**: Contains all environment variables and cluster parameters (cluster_name, DEPLOY_KUBECONFIG, passwords, secrets, etc.)
- **<CLUSTER_NAME>.yaml**: Cluster configuration YAML file with domain, ingress settings, passwords, and cluster-specific parameters
- **microk8s-config.yaml**: MicroK8s-specific configuration file
- **maia-registry-credentials.json**: Harbor registry credentials in JSON format
- **MAIA_realm.json**: Keycloak realm configuration template
- **<CLUSTER_NAME>-kubeconfig.yaml**: Kubernetes kubeconfig file (generated after cluster installation)
- **ca.crt**: Certificate Authority certificate (generated during installation)
- **cifs_key** and **cifs_key.pub**: CIFS public-private key pair (if generated)

All files generated by the Ansible scripts, including kubeconfig files, CA certificates, and other cluster-specific files, are stored in the `CONFIG_FOLDER`. This makes the configuration self-contained and portable.

Note: The `CONFIG_FOLDER` must be accessible to the Ansible control node and should have appropriate permissions for file creation and modification.

4.4.2 Inventory

The Ansible inventory file defines the hosts and groups that will be managed during the MAIA installation. The inventory structure is based on the roles that will be applied to different host groups.

Inventory Groups

control-plane

The `control-plane` group contains hosts that will run the Kubernetes control plane (API server, etcd, scheduler, controller manager). For MicroK8s installations, this typically includes the primary node where MicroK8s is installed.

```
[control-plane]
maia-node-0 ansible_user=root ansible_become=true
maia-node-1 ansible_user=ansible-user ansible_become_password=ansible ansible_
↪become=true ansible_become_method=sudo
```

Roles applied: `microk8s`, `oidc_microk8s`, `ca_microk8s`, `argocd`, `maia_core_layer`

nfs_server

The `nfs_server` group contains hosts that will act as NFS servers, providing shared storage for the cluster. Only one host should typically be in this group.

```
[nfs_server]
maia-node-0 ansible_user=root ansible_become=true
```

Roles applied: `nfs` (server tasks), `lvm` (with NFS storage volume)

Note: Hosts in the `nfs_server` group will have an additional NFS storage logical volume (`maia_0`) created by the `lvm` role, in addition to the local storage volume.

nfs_clients

The `nfs_clients` group contains hosts that will mount the NFS share from the NFS server. These hosts need NFS client packages installed and will mount the shared storage.

```
[nfs_clients]
maia-node-1 ansible_user=ansible-user ansible_become_password=ansible ansible_
↪become=true ansible_become_method=sudo
maia-node-2 ansible_user=ansible-user ansible_become_password=ansible ansible_
↪become=true ansible_become_method=sudo
```

Roles applied: `nfs` (client tasks)

Note: The NFS server automatically configures firewall rules to allow access from hosts in the `nfs_clients` group.

Host Variables for LVM

The `lvm` role requires host-specific variables to be defined for each host that will have LVM volumes created. These variables should be defined in `host_vars/<hostname>.yml` files.

Required Host Variable

- **device_list**: List of physical volumes (disk devices) to use for creating the volume group. Each device should be a block device path.

Example (`host_vars/maia-node-0.yml`):

```
device_list:
  - /dev/sda1
  - /dev/sdc2
```

Optional Host Variables

- **local_storage_size**: Size of the local storage logical volume (`maia_0_local`). Default: `100%FREE`
  - Can be specified as absolute size: `300g`, `500m`, `1t`
  - Or as percentage: `100%FREE`, `50%FREE`, `50%VG`
- **nfs_storage_size**: Size of the NFS storage logical volume (`maia_0`). Default: `100%FREE`
  - Only used for hosts in the `nfs_server` group
  - Same format as `local_storage_size`

Example (`host_vars/maia-node-0.yml` for NFS server):

```
device_list:
  - /dev/sda1
  - /dev/sdc2
local_storage_size: 300g
nfs_storage_size: 1.8t
```

Example (`host_vars/maia-node-1.yml` for regular node):

```
device_list:
  - /dev/sda1
local_storage_size: 500g
```

4.4.3 Ansible Collection - MAIA.Installation

Documentation for the MAIA.Installation Ansible collection, which provides roles and playbooks for installing and configuring MAIA (Medical AI Assistant) infrastructure on Kubernetes clusters.

Requirements

The following packages and tools are required before using this collection:

Python packages:

```
pip install maia-toolkit ansible jmespath
```

System packages:

```
apt install jq yq apache2-utils
```

Kubernetes tools:

```
curl -LO "https://dl.k8s.io/release/$(curl -L -s https://dl.k8s.io/release/stable.txt)/
↪bin/linux/amd64/kubectl"
sudo install -o root -g root -m 0755 kubectl /usr/local/bin/kubectl

curl -fsSL -o get_helm.sh https://raw.githubusercontent.com/helm/helm/main/scripts/get-
↪helm-4
chmod 700 get_helm.sh
./get_helm.sh

VERSION=$(curl -s https://api.github.com/repos/argoproj/argo-cd/releases/latest | grep␣
↪tag_name | cut -d '"' -f 4)
curl -sSL -o /usr/local/bin/argocd https://github.com/argoproj/argo-cd/releases/download/
↪$VERSION/argocd-linux-amd64
chmod +x /usr/local/bin/argocd
```

And verify the installations:

```
kubectl version
helm version
argocd version
```

Minimum Hardware Requirements

To successfully deploy a minimal MAIA environment, your host should meet at least the following hardware specifications:

- **Memory:** 8 GB RAM
- **CPU:** 4 CPU cores
- **Disk:** 20 GB available storage

Operating System:
MAIA installation has been fully tested on Ubuntu 22.04 and 24.04 LTS.

> **Note:**
> These requirements are **ONLY** for deploying and running the MAIA platform and a basic Kubernetes cluster. If you plan to run large projects or host resource-intensive workloads, you should scale up CPU, memory, disk space, and add GPUs as needed for your use case.

Installation

To install the MAIA.Installation collection, run the following command:

```
ansible-galaxy collection install MAIA.Installation
```

Quickstart: Full MAIA Installation in 10 Minutes

Use the MAIA Toolkit's one-command installer to deploy the `MAIA.Installation` collection and perform a full installation in about 10 minutes.

To run this installer, you must provide a **configuration folder** containing:

- **Inventory**: An Ansible inventory file or folder defining your hosts and their roles.
- **Configuration file**: A `config.yaml` file describing the installation steps and options.

Example `inventory` file:

```
[control-plane]
maia-dev-node-0 ansible_host=127.0.0.1 ansible_connection=local ansible_user=ansible-
↪user ansible_become_password=ansible ansible_become=true ansible_become_method=sudo
```

Example `config.yaml`

```yaml
# List of steps to execute in sequence, each of them is a playbook
steps:
  - prepare_hosts
  - configure_hosts
  - install_microk8s
  - install_maia_core
  - install_maia_admin
  - configure_oidc_authentication
  - get_kubeconfig_from_rancher_local
  - configure_maia_dashboard

# Playbook-specific configuration options

prepare_hosts:
  nvidia_drivers: false
  ufw: true
  nfs: false
  cifs: false

install_microk8s:
  install_microk8s: true
  enable_oidc_microk8s: true
  enable_ca: true
  install_argocd: true
  connect_to_microk8s: false
  connect_to_argocd: false
```

(continues on next page)

(continued from previous page)

```
install_maia_core:
  auto_sync: true

install_maia_admin:
  auto_sync: true

configure_oidc_authentication:
  configure_rancher: true
  configure_harbor: true
  harbor_admin_user: admin
  harbor_admin_pass: Harbor12345

get_kubeconfig_from_rancher_local:
  kubeconfig_file: "local.yaml" # kubeconfig from the Rancher local cluster, stored in␣
↪the config folder

configure_maia_dashboard:
  auto_sync: true

# Environment variables used during the execution of the script MAIA_Configure_
↪Installation.sh
env:
  MAIA_PRIVATE_REGISTRY: ""
  CLUSTER_DOMAIN: "example.maia.com"
  CLUSTER_NAME: "maia-cluster"
  INGRESS_RESOLVER_EMAIL: ""
  K8S_DISTRIBUTION: "microk8s"

# Additional cluster configuration options
cluster_config_extra_env:
  selfsigned: true
  coredns_mappings:
    - subdomain: "iam"
      coredns_ip: "<IP_ADDRESS>"
    - subdomain: "registry"
      coredns_ip: "<IP_ADDRESS>"
    - subdomain: "mgmt"
      coredns_ip: "<IP_ADDRESS>"
    - subdomain: "kubeflow"
      coredns_ip: "<IP_ADDRESS>"
  externalCA:
    name: "external-ca-secret"
    cert: "<PATH_TO_CERTIFICATE>"
  shared_storage_class: microk8s-hostpath
```

Run the Installation

From your terminal, execute:

```
MAIA_install --config-folder /path/to/config_folder
```

Once the installation is complete, you can access the MAIA Dashboard at `https://maia.<cluster_domain>`. Wait for the dashboard to be ready by checking the `maia-dashboard` namespace:

```
export KUBECONFIG=<CONFIG_FOLDER>/<CLUSTER_NAME>-kubeconfig.yaml
kubectl get pod -n maia-dashboard
```

Output:

```
NAME                                              READY   STATUS    RESTARTS   AGE
admin-minio-tenant-pool-0-0                       2/2     Running   0          44m
maia-admin-maia-dashboard-b87475666-2vs77         1/1     Running   0          3m15s
maia-admin-maia-dashboard-mysql-5fffdd655c-5x92x  1/1     Running   0          3m57s
```

For first-access, you can use the following credentials:

```
username: admin@maia.io
password [Temporary]: admin
```

Installation on Linux and Windows Subsystem for Linux (WSL)

To install MAIA on Linux and Windows Subsystem for Linux (WSL), you can use the following one-command installer:

```
# To fetch and use the latest release of the installer automatically, you can use the␣
↪following command:
LATEST=$(curl -s https://api.github.com/repos/minnelab/MAIA/releases/latest | grep tag_
↪name | cut -d '"' -f4)
wget "https://github.com/minnelab/MAIA/releases/download/${LATEST}/install_MAIA.sh" &&␣
↪chmod +x install_MAIA.sh && ./install_MAIA.sh
```

To access all the features of MAIA, verify that all the subdomains are mapped in your Windows or Linux hosts files:

```
# Add the following lines to your Windows hosts file:
# C:\Windows\System32\drivers\etc\hosts
# Add the following lines to your Linux hosts file:
# /etc/hosts
<WSL_IP> <domain>
<WSL_IP> traefik.<domain>
<WSL_IP> dashboard.<domain>
<WSL_IP> grafana.<domain>
<WSL_IP> iam.<domain>
<WSL_IP> registry.<domain>
<WSL_IP> mgmt.<domain>
<WSL_IP> minio.<domain>
<WSL_IP> argocd.<domain>
<WSL_IP> maia.<domain>
<WSL_IP> test.<domain>
```

(continues on next page)

(continued from previous page)

```
<WSL_IP> minio.test.<domain>
<WSL_IP> login.<domain>
```

MAIA Dashboard and MAIA Workspace Dev Environment

To deploy the dev environment for the MAIA Dashboard and MAIA Workspace, you can set the following environment variables before running the installer:

```
export DEV_BRANCH=<branch_name>
export GIT_EMAIL=<email>
export GIT_NAME=<name>
export GPG_KEY=<path/to/gpg.key>
```

The DEV_BRANCH is the branch that will be used to deploy the MAIA Dashboard and install the `maia-toolkit` package. The GIT_EMAIL and GIT_NAME are the email and name of the user that will be used to commit the changes to the repository. The GPG_KEY is the path to the GPG key that will be used to sign the commits.

Extra Configuration

Deploy MAIA-Admin without Rancher

If you want to deploy MAIA-Admin without Rancher, you can set in your config.yaml file the following variable, to force the user OIDC authentication for the MAIA Dashboard to connect with the k8s cluster (by default, the connection is done through Rancher).

```
env:
  MAIA_DASHBOARD_OIDC_AUTHENTICATION: true
```

Move Let's Encrypt staging to production

After validating the Ingress certificate with the staging CA, you can move the Let's Encrypt staging to production by performing the following steps:

1. Set the OIDC_CA_BUNDLE environment variable to True in the maia-admin-maia-dashboard Application from ArgoCD:

```
env:
  - name: OIDC_CA_BUNDLE
    value: True
```

2. Remove the rootCA environment variable from the ArgoCD `argocd-cm` ConfigMap, entry `oidc.config`.
3. Remove the `--oidc-ca-file` configuration from the apiserver args (kube-apiserver) and restart the kubernetes service. In microk8s, you can find the apiserver-args in the `/var/snap/microk8s/current/args/kube-apiserver` file, while in k0s, you can find it in the `/etc/k0s/k0s.yaml` file, in k3s, you can find it in the `/etc/rancher/k3s/config.yaml` file.
4. From the `maia-admin-maia-dashboard` ConfigMap in the `maia-dashboard` namespace, edit the <cluster_name>.yaml file and set the selfsigned variable to False:

```
selfsigned: false
```

From the <cluster_name>.yaml file in the config folder, set the staging_certificates variable to False and delete the externalCA variable:

```
staging_certificates: false
externalCA: null
```

5. Restart the kubernetes service.
6. From the maia-admin-maia-dashboard ConfigMap in the maia-dashboard namespace, edit the <cluster_name>.yaml file and set the selfsigned variable to False:

```
selfsigned: false
```

Set MAIA Registry

The default MAIA Registry is ghcr.io/minnelab, but you can set it to any other registry by setting the MAIA_REGISTRY environment variable in your config.yaml file:

```
env:
  MAIA_REGISTRY: maiacloudai
```

Set Dev Environment

To enable the dev environment for the MAIA Dashboard, allowing to install, deploy and update the MAIA ecosystem with the latest changes from the dev branch, and also contributing to the MAIA Github repository, you can set the following environment variables in your config.yaml file:

```
env:
  DEV_BRANCH: master
  GIT_EMAIL: <email>
  GIT_NAME: <name>
  GPG_KEY: <path/to/gpg.key>
```

Set Webhook and Support URL

The WEBHOOK_URL is the URL of the webhook that will be used to send notifications to the admin channel, notifying the administrators about the new project registrations and user registrations. The SUPPORT_URL is the URL for registering to the support channels, such as Discord, Slack, Mattermost, etc.

```
env:
  WEBHOOK_URL: <WEBHOOK_URL>
  SUPPORT_URL: <SUPPORT_URL>
```

Set OpenWebAI API Key and URL

To enable the OpenWebAI API for the MAIA Chatbot, you can set the following environment variables in your config.yaml file:

```
env:
  OPENWEBAI_API_KEY: <OPENWEBAI_API_KEY>
  OPENWEBAI_URL: <OPENWEBAI_URL>
  OPENWEBAI_MODEL: <OPENWEBAI_MODEL>
```

The OPENWEBAI_API_KEY is the API key for the OpenWebAI API. The OPENWEBAI_URL is the URL for the OpenWebAI API. The OPENWEBAI_MODEL is the model to use for the OpenWebAI API.

Set Email Notification System

The email notification system is used to send notifications to users about the user and project notifications in MAIA, such as project approval, user approval, project registration, user registration, etc. To enable the email notification system for the MAIA Dashboard, you can set the following environment variables in your config.yaml file:

```
env:
  SMTP_SENDER_EMAIL: <SMTP_SENDER_EMAIL>
  SMTP_SERVER: <SMTP_SERVER>
  SMTP_PORT: <SMTP_PORT>
  SMTP_PASSWORD: <SMTP_PASSWORD>
```

The SMTP_SENDER_EMAIL is the email address that will be used to send the notifications. The SMTP_SERVER is the SMTP server address. The SMTP_PORT is the SMTP server port. The SMTP_PASSWORD is the SMTP server password.

Select which ArgoCD applications to automatically synchronize

For the MAIA core and admin layers, you can select which ArgoCD applications to automatically synchronize by setting the following environment variables in your config.yaml file:

```
install_maia_core:
  auto_sync: true
  maia_core_argocd_applications: ["maia-core-traefik", "maia-core-local-path", "maia-
↪core-cert-manager","maia-core-metallb",  "maia-core-toolkit", "maia-core-minio-operator
↪", "maia-core-kubeflow"]

install_maia_admin:
  auto_sync: true
  maia_admin_argocd_applications: ["maia-core-loginapp", "maia-admin-keycloak", "maia-
↪admin-admin-toolkit"]
```

Override default cluster configuration

To override the default cluster configuration, you can set the following variables in your config.yaml file (e.g., to set the shared storage class to microk8s-hostpath and the port range to 32000-32767):

```
cluster_config_extra_env:
  shared_storage_class: microk8s-hostpath
  jupyterhub_username_claim: "email"
  port_range:
  - 32000
  - 32767
```

Project Configuration:

To add a project to the MAIA Dashboard, you can set the following variables in your <project_id>.yaml file (or JSON file):

```
name: <project_id>
  <key>: <value>
  env:
    <env_key>: <env_value>
```

and the following variables in your config.yaml file:

```
env:
  maia_projects: "<path/to/project_id_1>.yaml,<path/to/project_id_2>.yaml,<path/to/
↪project_id_3>.yaml"
```

Enable CIFS shared storage support

Use to enable the CIFS shared storage support for the project, creating a CIFS secret in the namespace for the CIFS encryption public key, CIFS volumes will be mounted to the JupyterHub and FileBrowser Apps.

```
env:
  enable_cifs: true
```

MLflow and FileBrowser App Credentials

```
mlflow_user: "<mlflow_user>"
mlflow_password: "<mlflow_password>"
```

MinIO App Credentials

```
minio_user: "<minio_user>"
minio_password: "<minio_password>"
minio_console_access_key: "<minio_console_access_key>"
minio_console_secret_key: "<minio_console_secret_key>"
```

MySQL App Credentials

For both the MySQL deployments associated with MLFlow and Orthanc. For the Orthanc MySQL deployment, the user is `maia-admin` and the password is the same as the one for the MLFlow MySQL database.

```
mysql_user: "<mysql_user>"
mysql_password: "<mysql_password>"
```

Load Balancer IP Whitelist

Use to whitelist the IP addresses that will be allowed to access the project services. Specifically, if the Orthanc and SSH Services are deployed as LoadBalancer, the IP addresses will be allowed to access the Orthanc and SSH Services. The IP addresses will be also allowed to access the Kubeflow project through the Authorization Policies, if Kubeflow is deployed.

```
ip_whitelist:
  - "<ip_address>"
  - "<ip_address>"
```

Override default GPU request

Use to override the default GPU request for the project.

```
gpu_request: <gpu_request>
```

JupyterHub Configuration

```
admins:  # List of admin emails for the JupyterHub Environment
  - "<admin_email>"
  - "<admin_email>"

password: "<password>" # Default assigned password set for all the users in the␣
↪JupyterHub Environment
allow_ssh_password_authentication: "True" # Allow SSH password authentication for the␣
↪JupyterHub Environment
active_server_limit: 1 # Active server limit for the JupyterHub Environment
concurrent_spawn_limit: 1 # Concurrent spawn limit for the JupyterHub Environment
shared_server_user: "<shared_server_user>" # Shared server user for the JupyterHub␣
↪Environment
jupyterhub_extraEnv:
  INSTALL_SLICER: "1" # Install Slicer for the JupyterHub Environment
```

Resources Limits and Requests

```
env:
  ORTHANC_CPU_REQUEST: "4000m"
  ORTHANC_CPU_LIMIT: "4000m"
  ORTHANC_MEMORY_REQUEST: "4Gi"
  ORTHANC_MEMORY_LIMIT: "4Gi"
  ORTHANC_MYSQL_CPU_REQUEST: "500m"
  ORTHANC_MYSQL_CPU_LIMIT: "500m"
  ORTHANC_MYSQL_MEMORY_REQUEST: "2Gi"
  ORTHANC_MYSQL_MEMORY_LIMIT: "2Gi"
  MLFLOW_CPU_REQUEST: "500m"
  MLFLOW_CPU_LIMIT: "500m"
  MLFLOW_MEMORY_REQUEST: "2Gi"
  MLFLOW_MEMORY_LIMIT: "2Gi"
  MYSQL_CPU_REQUEST: "500m"
  MYSQL_CPU_LIMIT: "500m"
  MYSQL_MEMORY_REQUEST: "2Gi"
  MYSQL_MEMORY_LIMIT: "2Gi"
```

MONAI Label Authentication for Orthanc

To be set only if the MONAI Label servers linked to the Orthanc deployment are protected by authentication.

```
env:
  MONAI_LABEL_AUTH_USERNAME: <username>
  MONAI_LABEL_AUTH_PASSWORD: <password>
```

MONAI Label Models for Orthanc

To set the MONAI Label models for the Orthanc deployment, you can set the following environment variable in your config.yaml file:

```
monai_label_models:
  <model_name>:
    label_info:
      params:
        label_info:
          - name: <label_1_name>
            model_name: <model_name>
          - name: <label_2_name>
            model_name: <model_name>
    host: <host>
```

NVFlare Dashboard

To enable the NVFlare Dashboard for the project, you can set the following environment variables in your config.yaml file:

```
env:
  DEPLOY_NVFLARE_DASHBOARD: "True"
  nvflare_dashboard_admin_username: <username>
  nvflare_dashboard_admin_password: <password>
```

Kubeflow Project

To deploy the project as a Kubeflow project, you can set the following environment variable in your config.yaml file:

```
env:
  DEPLOY_KUBEFLOW: "True"
```

Email to Username Map

Used to set the username when registering a new user in Keycloak.

```
email_to_username_map:
  "email_1": "username_1"
  "email_2": "username_2"
  "email_3": "username_3"
```

4.4.4 Example: Complete Cluster Setup – Components and Roles

```
[control-plane]
maia-node-0 ansible_user=root ansible_become=true

[nfs_server]
maia-node-0 ansible_user=root ansible_become=true

[nfs_clients]
maia-node-1 ansible_user=root ansible_become=true
maia-node-2 ansible_user=root ansible_become=true
```

With corresponding host variable files:

host_vars/maia-node-0.yml (control-plane and NFS server):

```
device_list:
  - /dev/sda1
  - /dev/sdc2
local_storage_size: 300g
nfs_storage_size: 1.8t
```

host_vars/maia-node-1.yml (NFS client):

```
device_list:
  - /dev/sda1
local_storage_size: 500g
```

host_vars/maia-node-2.yml (NFS client):

```
device_list:
  - /dev/sda1
local_storage_size: 500g
```

Usage

The MAIA installation with MicroK8s consists of three main playbooks that should be executed in sequence:

1. Prepare Hosts

```
ansible-playbook -i inventory playbooks/prepare_hosts.yaml -e "config_folder=<CONFIG_
↪FOLDER>" #Options -e nvidia_drivers=false -e lvm=false -e ufw=false -e nfs=false -e␣
↪cifs=false
```

The *prepare_hosts.yaml* playbook configures the base system requirements for MAIA installation. Each component can be enabled or disabled based on your specific needs:

Optional flags to skip specific components:

- *-e nvidia_drivers=false* - Skip NVIDIA driver installation
- *-e lvm=false* - Skip LVM volume creation
- *-e ufw=false* - Skip UFW firewall configuration
- *-e nfs=false* - Skip NFS server/client setup
- *-e cifs=false* - Skip CIFS plugin installation

This playbook installs NVIDIA drivers (if GPUs are present), configures LVM storage, sets up firewall rules, configures NFS for shared storage, and installs the CIFS plugin for Windows-compatible storage.

Install NVIDIA Drivers

Description: Installs and configures NVIDIA GPU drivers on Ubuntu/Debian systems using the apt package manager. The role handles driver installation and optionally reboots the system to activate the newly installed drivers.

Use Cases: - Clusters with NVIDIA GPUs requiring GPU support for workloads - GPU-accelerated machine learning, deep learning, or scientific computing workloads - Kubernetes deployments using GPU operators or GPU scheduling features

Benefits: - Enables GPU access for containerized workloads in Kubernetes - Required for GPU resource allocation and scheduling - Supports GPU monitoring and management tools - Can be skipped if your cluster doesn't have GPUs or doesn't need GPU support

Configuration: Optional - can be disabled with *-e nvidia_drivers=false*

Install LVM

Description: Configures LVM (Logical Volume Manager) to create volume groups and logical volumes for local and NFS storage. It creates a volume group (*MAIA_Storage*) from specified physical volumes, creates logical volumes for local storage (*maia_0_local*) and optionally NFS storage (*maia_0*), formats them, and configures automatic mounting.

Use Cases: - Environments requiring flexible storage management with the ability to resize volumes later - Deployments needing to partition disk space between local storage and NFS storage - Systems where storage requirements may change over time - Scenarios where you want to avoid directly assigning all available disk space to the cluster

Benefits: - Provides storage flexibility - volumes can be resized without reformatting - Allows better disk space management and allocation - Enables separation of local storage (for local-path-provisioner) and NFS storage - Supports dynamic storage expansion as needs grow - Prevents locking all disk space to a single purpose

Configuration: Recommended for production deployments - can be disabled with *-e lvm=false*

Install UFW

Description: Configures UFW (Uncomplicated Firewall) to allow SSH access and enable bidirectional communication between all nodes in the inventory. It automatically discovers all hosts and creates firewall rules to allow inter-node traffic, which is essential for Kubernetes cluster communication.

Use Cases: - Hosts with UFW firewall enabled (default on Ubuntu) - Kubernetes clusters where nodes need to communicate with each other - Deployments requiring proper network connectivity between cluster nodes - Security-conscious environments that require firewall configuration

Benefits: - Ensures Kubernetes nodes can communicate with each other (required for cluster functionality) - Maintains SSH access while configuring firewall rules - Automatically configures rules for all nodes in the inventory - Prevents firewall from blocking essential Kubernetes traffic (API server, kubelet, etc.) - Can be configured to allow additional ports for specific services

Configuration: Recommended for most deployments - can be disabled with *-e ufw=false* if firewall is managed differently

Install NFS

Description: Configures NFS (Network File System) server and client components. For hosts in the *nfs_server* group, it installs NFS server packages, creates export directories, configures exports, and sets up firewall rules. For hosts in the *nfs_clients* group, it installs NFS client packages, creates mount points, and mounts the NFS share persistently.

Use Cases: - Deployments requiring shared storage across multiple Kubernetes nodes - Persistent volumes that need to be accessible from any node - Kubernetes clusters using NFS-based storage classes - Applications that require shared filesystem access

Benefits: - Provides shared storage that can be accessed from any node in the cluster - Enables persistent volumes that survive pod migrations - Supports ReadWriteMany (RWX) access mode for Kubernetes volumes - Useful for shared data, logs, or application state - Works well with the LVM role to provide the underlying storage

Configuration: Required if using NFS storage - can be disabled with *-e nfs=false*

Install CIFS

Description: Installs and configures CIFS (Common Internet File System) support for Kubernetes. It sets up the CIFS volume plugin that allows Kubernetes to mount CIFS/SMB shares as persistent volumes using flexVolume drivers. The role installs required packages, downloads the CIFS plugin scripts, and configures the plugin with a private key for credential decryption.

Use Cases: - Deployments requiring Windows SMB/CIFS shares mounted in Kubernetes pods - Accessing network-attached storage (NAS) devices that use SMB/CIFS protocol - Integration with existing Windows file servers or Samba shares - Workloads that require access to CIFS-based storage systems

Benefits: - Enables Kubernetes to use existing CIFS/SMB shares as persistent volumes - Supports encrypted credentials for secure access to CIFS shares - Allows integration with Windows-based storage infrastructure - Provides persistent storage option when NFS is not available - Useful for hybrid environments with mixed storage protocols

Configuration: Optional - only needed if using CIFS storage. Can be disabled with *-e cifs=false* or by not providing the *cifs_private_key* variable

2. Install MicroK8s

```
ansible-playbook -i inventory playbooks/install_microk8s.yaml -e "config_folder=<CONFIG_
↪FOLDER>"
```

The *install_microk8s.yaml* playbook installs and configures MicroK8s, a lightweight Kubernetes distribution, along with essential authentication and tooling components. Each component is executed in sequence to set up a fully functional Kubernetes cluster:

This playbook installs MicroK8s, enables OIDC authentication, creates the CA certificate in cert-manager, and installs ArgoCD for GitOps-based application management.

Install MicroK8s

Description: Installs and configures MicroK8s, a lightweight, single-package Kubernetes distribution. The role installs MicroK8s via snap, enables essential addons (hostpath-storage, rbac), configures kubeconfig for local access, sets up firewall rules, and establishes SSH port forwarding for API server access.

Use Cases: - Single-node or small multi-node Kubernetes clusters - Development and testing environments requiring quick Kubernetes setup - Edge computing deployments needing lightweight Kubernetes - Local development clusters for application testing

Benefits: - Fast and simple Kubernetes installation via snap package manager - Minimal resource footprint compared to full Kubernetes distributions - Includes essential addons for storage and RBAC out of the box - Automatic kubeconfig generation and configuration - Built-in port forwarding for easy local access - Supports standard Kubernetes APIs and tools

Configuration: Required - this is the core Kubernetes installation step

Enable OIDC

Description: Configures OIDC (OpenID Connect) authentication for MicroK8s, enabling integration with identity providers like Keycloak. The role configures the Kubernetes API server with OIDC parameters, allowing users to authenticate using JWT tokens from the OIDC provider.

Use Cases: - Clusters requiring centralized identity management - Integration with existing identity providers (Keycloak, Okta, etc.) - Multi-user environments needing role-based access control - Enterprise deployments with SSO requirements

Benefits: - Enables single sign-on (SSO) for Kubernetes cluster access - Integrates with existing identity management infrastructure - Supports group-based authorization for RBAC - Allows users to authenticate without managing separate Kubernetes credentials - Provides audit trail through identity provider logs

Configuration: Required for OIDC-enabled clusters - ensures users can authenticate via Keycloak

Create CA in Cert-manager

Description: Configures the Kubernetes CA (Certificate Authority) certificate and key from MicroK8s for use with cert-manager. Creates a TLS secret in the cert-manager namespace containing the MicroK8s CA certificate and private key, enabling cert-manager to issue certificates signed by the Kubernetes CA.

Use Cases: - Clusters using cert-manager for certificate management - Applications requiring TLS certificates signed by the cluster CA - Secure service-to-service communication within the cluster - Automated certificate provisioning and renewal

Benefits: - Enables cert-manager to issue cluster-signed certificates - Automates TLS certificate management for applications - Provides secure communication between services - Supports automatic certificate renewal - Integrates with Let's Encrypt and other certificate authorities

Configuration: Required if using cert-manager - enables automated certificate management

Install ArgoCD

Description: Installs and configures ArgoCD (Argo Continuous Delivery), a declarative GitOps continuous delivery tool for Kubernetes. The role installs ArgoCD manifests, configures the admin password, creates service accounts for automation, and sets up port forwarding for UI access.

Use Cases: - GitOps-based deployment workflows - Automated application deployment from Git repositories - Multi-environment deployment management (dev, staging, production) - Continuous delivery pipelines requiring declarative configuration

Benefits: - Enables GitOps workflows for application deployment - Provides declarative application management - Supports automatic synchronization from Git repositories - Offers web UI and CLI for application management - Tracks application state and provides rollback capabilities - Supports multi-cluster deployments

Configuration: Required for GitOps workflows - essential for MAIA's application deployment model

3. Install MAIA-Core Layer

```
ansible-playbook -i inventory playbooks/install_maia_core.yaml -e "config_folder=<CONFIG_
↪FOLDER>"
```

This playbook installs the MAIA Core Toolkit, Prometheus stack for observability, and synchronizes all ArgoCD applications for core components including Traefik, cert-manager, GPU operator, MetalLB, Loki, Tempo, metrics server, GPU booking system, MinIO operator, and NFS provisioner.

Note: All playbooks require the *config_folder* variable to be set, which should point to the directory created by *MAIA_Configure_Installation.sh* containing *env.json* and the cluster configuration files.

The *install_maia_core.yaml* playbook installs and configures the MAIA-Core layer components on a Kubernetes cluster. This playbook deploys essential infrastructure components including ingress controllers, storage solutions, observability stack, GPU support, and core toolkit applications via ArgoCD. Each component is deployed as an ArgoCD application for GitOps-based management:

MAIA Core Toolkit Installer

Description: Executes the *MAIA_install_core_toolkit* command which creates and configures ArgoCD applications for all MAIA Core components. The installer reads cluster configuration and environment variables to deploy core infrastructure applications including Traefik, cert-manager, GPU operator, MetalLB, observability stack (Loki, Tempo), metrics server, GPU booking system, MinIO operator, and NFS provisioner. It also creates the necessary namespaces and configures ArgoCD project settings.

Use Cases: - Initial MAIA Core infrastructure deployment - GitOps-based infrastructure management - Automated application lifecycle management - Multi-component infrastructure orchestration

Benefits: - Centralized deployment of all core components - GitOps-based configuration management - Declarative infrastructure as code - Automated synchronization from Git repositories - Consistent deployment across environments

Configuration: Required - this is the core deployment mechanism that creates all ArgoCD applications

Traefik (maia-core-traefik)

Description: Deploys Traefik, a modern HTTP reverse proxy and load balancer that serves as the primary ingress controller for MAIA. Traefik automatically discovers services and routes traffic based on Ingress resources, providing TLS termination, load balancing, and advanced routing capabilities. It integrates with cert-manager for automatic SSL certificate management.

Use Cases: - Ingress controller for Kubernetes services - TLS/SSL termination for HTTPS traffic - Load balancing across service instances - Path-based and host-based routing - Integration with Let's Encrypt for automatic certificates

Benefits: - Automatic service discovery and routing - Dynamic configuration updates without restarts - Built-in support for Let's Encrypt certificates - Web dashboard for monitoring and debugging - Support for multiple protocols (HTTP, HTTPS, TCP, UDP)

Configuration: Required for ingress - provides external access to MAIA services

Cert Manager (maia-core-cert-manager)

Description: Installs cert-manager, a Kubernetes add-on that automates the management and issuance of TLS certificates from various issuing sources. It integrates with Let's Encrypt, HashiCorp Vault, and other certificate authorities to automatically provision and renew certificates for services in the cluster.

Use Cases: - Automatic TLS certificate provisioning - Certificate renewal management - Integration with Let's Encrypt for free certificates - Secure service-to-service communication - Certificate management for ingress controllers

Benefits: - Automated certificate lifecycle management - Reduces manual certificate management overhead - Supports multiple certificate authorities - Automatic renewal before expiration - Kubernetes-native certificate management

Configuration: Required for TLS - enables automatic certificate management for secure communications

GPU Operator (maia-core-gpu-operator)

Description: Deploys the NVIDIA GPU Operator, which automates the management of all NVIDIA software components needed to provision GPUs in Kubernetes. This includes the NVIDIA device plugin for Kubernetes, NVIDIA Container Runtime, NVIDIA driver, GPU Feature Discovery, and Data Center GPU Manager (DCGM) exporter. The operator supports both MicroK8s and RKE2 Kubernetes distributions.

Use Cases: - GPU-accelerated workloads in Kubernetes - Machine learning and deep learning training - High-performance computing (HPC) applications - GPU resource scheduling and management - Multi-GPU node support

Benefits: - Automated GPU driver and runtime installation - Dynamic GPU resource allocation - Support for multiple GPU models and architectures - Integration with Kubernetes device plugin framework - Monitoring and metrics via DCGM

Configuration: Required for GPU support - enables GPU workloads in the cluster

MetalLB (maia-core-metallb)

Description: Installs MetalLB, a load-balancer implementation for bare metal Kubernetes clusters. MetalLB provides a network load balancer that works with standard network equipment, allowing services of type LoadBalancer to receive external IP addresses. It supports both Layer 2 (ARP/NDP) and BGP modes for IP address assignment.

Use Cases: - LoadBalancer services in bare metal environments - On-premises Kubernetes deployments - Cloud-like load balancing without cloud provider - External IP assignment for services - Integration with existing network infrastructure

Benefits: - Enables LoadBalancer services in bare metal - No cloud provider dependency required - Supports both Layer 2 and BGP protocols - Automatic IP address management - Works with standard networking equipment

Configuration: Required for LoadBalancer services - provides external IPs for services

Loki (maia-core-loki)

Description: Deploys Grafana Loki, a horizontally-scalable, highly-available log aggregation system inspired by Prometheus. Loki is designed to be very cost-effective and easy to operate, indexing only metadata (labels) while storing log content separately. It integrates seamlessly with Grafana for log visualization and querying.

Use Cases: - Centralized log aggregation from all pods - Log querying and analysis - Integration with Grafana for log visualization - Cost-effective log storage - Application debugging and troubleshooting

Benefits: - Efficient log storage and indexing - Prometheus-like labeling system - Seamless Grafana integration - Horizontal scalability - Lower storage costs compared to full-text indexing

Configuration: Required for observability - provides centralized log aggregation

Tempo (maia-core-tempo)

Description: Installs Grafana Tempo, a high-volume, minimal-dependency distributed tracing backend. Tempo is cost-efficient, requiring only object storage to operate, and can be used with any of the open source tracing protocols, including Jaeger, Zipkin, and OpenTelemetry. It integrates with Loki and Prometheus for correlation of traces, logs, and metrics.

Use Cases: - Distributed tracing for microservices - Performance analysis and optimization - Request flow visualization across services - Integration with OpenTelemetry, Jaeger, Zipkin - Correlation of traces with logs and metrics

Benefits: - Cost-effective distributed tracing - Minimal infrastructure requirements - Support for multiple tracing protocols - Integration with Grafana, Loki, and Prometheus - High-volume trace ingestion

Configuration: Required for observability - provides distributed tracing capabilities

Metrics Server (maia-core-metrics-server)

Description: Deploys Kubernetes Metrics Server, a scalable, efficient source of container resource metrics for Kubernetes built-in autoscaling pipelines. The Metrics Server collects resource usage data (CPU and memory) from each node's kubelet and makes it available via the Kubernetes Metrics API for use by Horizontal Pod Autoscaler (HPA) and Vertical Pod Autoscaler (VPA).

Use Cases: - Resource metrics collection for autoscaling - Horizontal Pod Autoscaler (HPA) support - Vertical Pod Autoscaler (VPA) support - Kubernetes dashboard metrics - Resource usage monitoring

Benefits: - Enables Kubernetes autoscaling features - Efficient resource metrics collection - Standard Kubernetes Metrics API - Required for HPA and VPA functionality - Low overhead metrics collection

Configuration: Required for autoscaling - enables HPA and VPA functionality

Prometheus Stack

Description: Installs the kube-prometheus-stack Helm chart, which includes Prometheus for metrics collection, Grafana for visualization, Alertmanager for alerting, and various exporters. This provides a complete observability solution for monitoring cluster health, application metrics, and infrastructure performance. The stack is installed in the observability namespace.

Use Cases: - Cluster and application metrics collection - Infrastructure monitoring and alerting - Performance analysis and capacity planning - Service level objective (SLO) monitoring - Integration with Grafana dashboards

Benefits: - Comprehensive monitoring solution - Pre-configured Grafana dashboards - Alertmanager for alert routing - Prometheus Operator for easy management - Integration with Loki and Tempo

Configuration: Required for observability - provides metrics collection and visualization

MAIA Core Toolkit (maia-core-toolkit)

Description: Deploys the MAIA Core Toolkit, which includes essential MAIA components and utilities. This includes webhooks for admission control, authentication components, and core MAIA services that provide the foundation for the MAIA platform. The toolkit provides core functionality required by other MAIA components.

Use Cases: - Core MAIA platform functionality - Admission control webhooks - Authentication and authorization components - Core MAIA services and APIs - Foundation for MAIA applications

Benefits: - Centralized core MAIA functionality - Reusable components across MAIA - Consistent authentication and authorization - Admission control for security - Foundation for higher-level MAIA features

Configuration: Required - provides core MAIA platform functionality

GPU Booking System (maia-core-gpu-booking)

Description: Deploys the MAIA GPU Booking System, which enables users to reserve and schedule GPU resources in the cluster. The system provides a web interface and API for booking GPUs for specific time periods, managing reservations, and tracking GPU usage. It integrates with the GPU operator and Kubernetes scheduling to allocate GPU resources.

Use Cases: - GPU resource reservation and scheduling - Multi-user GPU access management - Time-based GPU allocation - GPU usage tracking and reporting - Fair resource distribution

Benefits: - Prevents GPU resource conflicts - Enables fair GPU sharing - Time-based reservation system - Usage tracking and reporting - Web-based booking interface

Configuration: Required for GPU scheduling - enables GPU resource booking

MinIO Operator (maia-core-minio-operator)

Description: Installs the MinIO Operator, which manages MinIO object storage instances in Kubernetes. MinIO is a high-performance, S3-compatible object storage service that can be used for storing datasets, model artifacts, backups, and other unstructured data. The operator simplifies deployment and management of MinIO instances.

Use Cases: - S3-compatible object storage - Dataset and model artifact storage - Backup and archival storage - Data lake infrastructure - Integration with ML/AI workflows

Benefits: - S3-compatible API - High-performance object storage - Kubernetes-native operator - Simplified deployment and management - Cost-effective storage solution

Configuration: Required for object storage - provides S3-compatible storage for MAIA

NFS Provisioner (maia-core-nfs-provisioner)

Description: Deploys an NFS server provisioner that dynamically creates PersistentVolumes for Kubernetes using an existing NFS server. The provisioner automatically creates PersistentVolumeClaims and binds them to NFS-backed PersistentVolumes, enabling dynamic storage provisioning for applications that require shared file storage.

Use Cases: - Dynamic NFS storage provisioning - Shared file storage for applications - PersistentVolumeClaim automation - Multi-pod shared storage - Integration with existing NFS infrastructure

Benefits: - Automatic storage provisioning - Shared storage across pods - Dynamic volume creation - Integration with existing NFS servers - Simplifies storage management

Configuration: Required for NFS storage - enables dynamic NFS volume provisioning

4. Install MAIA Admin Layer

```
ansible-playbook -i inventory playbooks/install_maia_admin.yaml -e "config_folder=
↪<CONFIG_FOLDER>"
```

This playbook installs the MAIA Admin layer, deploying identity, registry, dashboards, and supporting services through the MAIA Admin Toolkit and ArgoCD applications.

Note: All playbooks require the *config_folder* variable to be set, pointing to the directory created by *MAIA_Configure_Installation.sh* containing *env.json* and the cluster configuration files.

The *install_maia_admin.yaml* playbook configures admin-facing components using the MAIA Admin Toolkit and ArgoCD synchronization:

MAIA Admin Toolkit Installer

Description: Executes the *MAIA_install_admin_toolkit* command using cluster configuration and required secrets to create ArgoCD applications and supporting resources for the admin stack.

Use Cases: - Initial deployment of admin-facing MAIA services - GitOps-based lifecycle management of admin apps - Automated provisioning of identity, registry, and dashboards

Benefits: - Centralized deployment of admin services - Declarative configuration via ArgoCD - Consistent, repeatable rollouts across environments

Configuration: Required — drives creation of admin ArgoCD applications and supporting resources.

Harbor (maia-admin-harbor)

Description: Deploys Harbor registry for container images and Helm charts, including project bootstrap performed by the Admin Toolkit.

Use Cases: - Private image and chart registry for MAIA workloads - Image scanning and access control - Registry replication and retention policies

Benefits: - Secure, role-based registry with scanning - Central artifact storage for MAIA deployments - Supports Helm/OIDC integrations

Configuration: Required for private registry needs and MAIA image distribution.

Keycloak (maia-admin-keycloak)

Description: Deploys Keycloak for identity and access management. The playbook can patch images, import the MAIA realm config map, and run *MAIA_configure_keycloak*.

Use Cases: - SSO and OIDC provider for the MAIA platform - Group-based RBAC across MAIA services - Credential and client management

Benefits: - Centralized identity with OIDC/SAML - Realm bootstrap and automated configuration - Integration with ArgoCD, dashboard, and other apps

Configuration: Required for authentication/authorization across MAIA components.

Rancher (maia-admin-rancher)

Description: Deploys Rancher for cluster and application lifecycle management, bootstrapped through ArgoCD.

Use Cases: - Multi-cluster operations and UI - Centralized Kubernetes management - RBAC and policy control

Benefits: - GUI-driven cluster administration - GitOps-friendly management - Integrated auth with Keycloak

Configuration: Optional but recommended for graphical cluster operations.

MAIA Admin Toolkit (maia-admin-admin-toolkit)

Description: Deploys admin-specific toolkit components (projects, policies, supporting services) used by the admin layer.

Use Cases: - Bootstrap of admin projects and resources - Helper services consumed by other admin apps

Benefits: - Consistent baseline for admin workloads - Preconfigured policies and resources

Configuration: Required — foundational pieces for other admin apps.

Login App (maia-core-loginapp)

Description: Synchronizes the MAIA login application providing user-facing authentication UI integrated with Keycloak.

Use Cases: - End-user login portal - Integration with MAIA dashboards and services

Benefits: - Unified entrypoint for MAIA users - Works with Keycloak and ArgoCD-managed configs

Configuration: Recommended for user-facing access to MAIA services.

MAIA Dashboard (maia-dashboard)

Description: Deploys the MAIA Dashboard application and supporting services for admin-facing UI and APIs (including database and secrets such as *dashboard_api_secret* and *mysql_dashboard_password* provided via the Admin Toolkit).

Use Cases: - Central UI for administrators to monitor and manage MAIA resources - Surfacing status of admin services (Harbor, Keycloak, Rancher) - Entry point for admin workflows that rely on Keycloak authentication

Benefits: - Consolidated admin experience - Integrates with Keycloak/SSO and ArgoCD-managed configuration - Uses GitOps to keep dashboard components in sync

Configuration: Recommended for operational visibility of MAIA Admin components.

MinIO Admin Tenant (maia-dashboard namespace)

Description: Ensures the MinIO tenant pod is running, creates the *maia-envs* bucket, and applies an admin policy via *mc* commands.

Use Cases: - Object storage for environment artifacts and configs - Admin-level access for MAIA ops tasks

Benefits: - S3-compatible storage scoped for admin workflows - Automated bucket creation and policy setup

Configuration: Required when admin components rely on MinIO-backed storage.

5. Configure OIDC Authentication

```
ansible-playbook -i inventory playbooks/configure_oidc_authentication.yaml -e "config_
↪folder=<CONFIG_FOLDER>"
```

This playbook configures OIDC-related settings for the MAIA environment, primarily by preparing Rancher (and optionally Harbor) using values from the cluster configuration.

Note: As with other playbooks, *config_folder* must point to the directory created by *MAIA_Configure_Installation.sh* that contains *env.json* and the cluster configuration files.

The *configure_oidc_authentication.yaml* playbook runs the *configure_oidc_authentication* role, which performs the following:

Rancher OIDC Preparation

Description: Reads the cluster configuration, extracts the *domain*, logs into Rancher at *https://mgmt.domain*, obtains a token, and accepts the Rancher EULA. This is a prerequisite step before enabling full OIDC integration with Keycloak.

Use Cases: - Initial Rancher bootstrap for MAIA clusters - Preparing Rancher for later OIDC configuration and SSO

Benefits: - Automates Rancher login and EULA acceptance - Ensures Rancher is in a known good state before further configuration

Configuration: Controlled by *configure_rancher* (default *true*) in the role; when set to *false*, Rancher configuration is skipped.

Harbor OIDC Preparation

Description: Optionally configures Harbor for OIDC authentication using the provided admin credentials and cluster configuration.

Use Cases: - Preparing Harbor to use Keycloak/OIDC for authentication - Aligning registry access control with the rest of the MAIA platform

Benefits: - Centralized identity via OIDC for the registry - Consistent access control model across MAIA components

Configuration: Controlled by *configure_harbor* (default *true*). Admin credentials are provided via *harbor_admin_user* and *harbor_admin_pass*.

6. Get Kubeconfig from Rancher Local

```
ansible-playbook -i inventory playbooks/get_kubeconfig_from_rancher_local.yaml -e
↪"config_folder=<CONFIG_FOLDER>"
```

This playbook retrieves a kubeconfig for the local Rancher-managed cluster using the Rancher API and stores it in the MAIA configuration folder.

Note: *config_folder* must point to the directory created by *MAIA_Configure_Installation.sh* and contain *env.json* and the cluster configuration YAML (*{{ cluster_name }}.yaml*).

The *get_kubeconfig_from_rancher_local.yaml* playbook performs the following steps:

Obtain Rancher API Token

Description: Reads the cluster configuration, extracts *domain*, and uses them to call the Rancher public login endpoint at *https://mgmt.domain* to obtain an authentication token. The token is stored in the env.json file as *rancher_token*.

Use Cases: - Authenticating to Rancher without manual UI interaction - Automating API access for subsequent operations

Benefits: - Eliminates manual login for CLI/API usage - Provides a reusable token for further Rancher API calls within the play

Create Rancher API Key

Description: Uses the login token to create a Rancher API key (*type: token*) and extracts the secret key from the response.

Use Cases: - Issuing a dedicated API token for kubeconfig generation - Enabling scripted Rancher API calls

Benefits: - Scoped API access token managed by Rancher - Clear separation between login credentials and API usage

Generate and Save Kubeconfig

Description: Calls the Rancher *generateKubeconfig* action on the *local* cluster using the API key, extracts the kubeconfig YAML from the response, and writes it to *{{ config_folder }}/local.yaml* (or the file specified by *kubeconfig_file*).

Use Cases: - Producing a kubeconfig for local development or automation - Feeding kubeconfig into subsequent MAIA tooling or playbooks

Benefits: - Fully automated kubeconfig retrieval - Ensures kubeconfig is stored alongside other MAIA configuration artifacts

Write Rancher Token to Cluster Config

Description: Writes the Rancher token to the env.json file.

Use Cases: - Providing a Rancher token for subsequent operations - Ensuring the Rancher token is available for other playbooks

Benefits: - Provides a reusable Rancher token for further Rancher API calls within the play - Ensures the Rancher token is stored alongside other MAIA configuration artifacts

7. Configure MAIA Dashboard

```
ansible-playbook -i inventory playbooks/configure_maia_dashboard.yaml -e "config_folder=
↪<CONFIG_FOLDER>"
```

This playbook configures the MAIA Dashboard by running the Admin Toolkit installer and synchronizing the dashboard ArgoCD application.

Note: *config_folder* must point to the directory created by *MAIA_Configure_Installation.sh* and contain *env.json* and the cluster configuration YAML (*{{ cluster_name }}.yaml*). The playbook requires various environment variables to be set in *env.json*, including *DEPLOY_KUBECONFIG*, *argocd_namespace*, *admin_group_ID*, *admin_project_chart*, *admin_project_repo*, *admin_project_version*, *keycloak_client_secret*, *minio_admin_password*, *minio_root_password*, *dashboard_api_secret*, *mysql_dashboard_password*, and *ARGOCD_PASSWORD*.

The *configure_maia_dashboard.yaml* playbook performs the following steps:

Run MAIA Admin Toolkit Installer

Description: Executes *MAIA_install_admin_toolkit* with the cluster configuration and required environment variables to configure the MAIA Dashboard and related admin components. The installer reads the cluster configuration YAML and uses environment variables for secrets and configuration values.

Use Cases: - Initial configuration of the MAIA Dashboard after admin layer installation - Updating dashboard configuration with new settings or secrets - Ensuring dashboard components are properly configured with Keycloak, MinIO, and database credentials

Benefits: - Centralized dashboard configuration through the Admin Toolkit - Ensures all required secrets and environment variables are properly set - Configures dashboard integration with Keycloak, MinIO, and MySQL

Login to ArgoCD

Description: Logs into ArgoCD using the CLI, attempting first with *localhost:8080* and falling back to *argocd.<domain>* if the local connection fails. Uses the *ARGOCD_PASSWORD* from environment variables.

Use Cases: - Authenticating to ArgoCD for application synchronization - Enabling automated ArgoCD operations without manual login

Benefits: - Automated ArgoCD authentication for subsequent operations - Fallback mechanism ensures connection even if port forwarding is not active - Enables scripted ArgoCD application management

Synchronize MAIA Dashboard Application

Description: Synchronizes the *maia-admin-maia-dashboard* ArgoCD application to ensure the dashboard deployment matches the desired state defined in Git. This step is conditional on *auto_sync* being enabled (default: *true*).

Use Cases: - Ensuring dashboard is deployed and up-to-date with Git configuration - Applying configuration changes from Git repositories - Recovering from deployment drift or failures

Benefits: - GitOps-based dashboard deployment management - Ensures dashboard matches declared configuration - Automatic synchronization from Git repositories

Restart Dashboard Deployment

Description: Performs a rollout restart of the *maia-admin-maia-dashboard* deployment in the *maia-dashboard* namespace to apply configuration changes and ensure pods are running with the latest settings. This step is conditional on *auto_sync* being enabled (default: *true*).

Use Cases: - Applying configuration changes that require pod restart - Refreshing dashboard pods after secret or config updates - Ensuring dashboard is running with latest configuration

Benefits: - Ensures configuration changes are applied immediately - Refreshes dashboard pods to pick up new environment variables or secrets - Maintains dashboard availability during restart process

CHAPTER

FIVE

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

m

INDEX

C

D

E

G

I

L

M

R

S

U

V