Odahu Platform Tutorial

version release/1.6.1

Odahu Team

April 26, 2022

Contents

About ODAHU

The Open Data AI Hub (ODAHU) is an open source project that provides the building blocks for enterprise grade MLOps platforms.

Multi ML Frameworks

- Supporting major ML frameworks: Scikit-learn, Keras, Tensorflow, PyTorch, H2O (and more)
- Extends MLflow services with enterprise level features

Multi Clouds

- Kubernetes native system of services
- Deployment automation to Kubernetes cluster with Helm charts
- Supporting major Kubernetes platforms: AWS EKS, Azure AKS, GCP GKE, RedHat OpenShift

Secure

- Single sign-on (SSO) based on OAuth2
- RESTful API secured with SSL
- Role based access control based on Open Policy Agent
- Users activity audit
- Credentials and keys manager based on HashiCorp Vault
- Internal traffic encryption with Istio

Modular and Extensible

- Services for different ML phases: transform, train, validate, package, deploy, evaluate
- Services are extensible and manageable via REST APIs, SDK and CLI
- Functionality extensible with new services
- Connectors for data sources, package repositories, Docker container registries
- Plugins for data science IDEs
- Plugins for workflow engines like Airflow

Scalable

- Systems based on ODAHU components can be scaled from small to very large.
- Scalable ML model training, packaging, serving components

Manageable

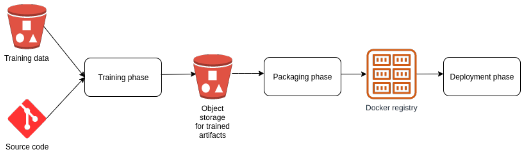
- Pre-build monitoring dashboards
- Configurable alerting rules
- Configurable logs collection
- Compatible with third party logs processing systems

Open

- It is free and open-source with the Apache2 License.
- Contribution to project is welcome!

Concepts

Phases



Odahu splits the ML/AI model lifecycle into three phases:

1. **Train**
2. **Package**
3. **Deploy**

Applications and tools can further automate each phase by implementing pluggable extensions *as*

1. **Trainer**
2. **Packager** or
3. **Deployer**

Trainers and Packagers can be registered as components of the Odahu Platform using:

1. **Trainer Extension**
2. **Packager Extension**

When registered, these components can use Odahu **Trainer Metrics** and **Trainer Tags**.

Users are encouraged to integrate third-party **Trainer Extensions** and **Packager Extensions**.

Toolchains

Taken together a Trainer, Packager, and Deployer comprise a *Toolchain* that automates an end-to-end machine learning pipeline.

Ready to use

Odahu provides a **Trainer Extension** and a **Packager Extension**

1. **MLflow Trainer**
2. **REST API Packager**

These power the default Toolchain.

Model storage

Odahu Platform stores models in **Trained Model Binaries** for different languages.

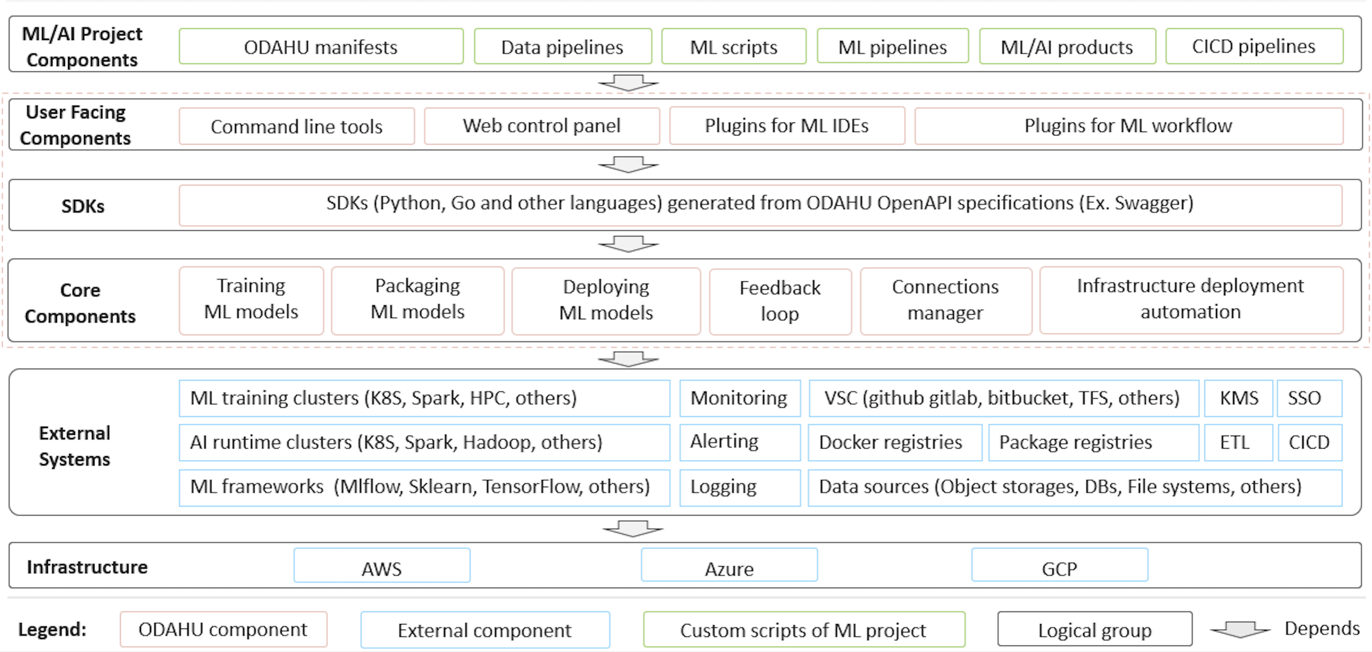
Presently, Odahu Platform supports only:

1. **General Python Prediction Interface**

Users are encouraged to provide additional formats.

Architecture

The components diagram below shows high level architecture of ODAHU project.



Core components:

- Training component for executing ML model training jobs in K8S.
- Packaging component for wrapping up ML model binary to an online service, batch job, library or command line tool.
- Deployment component for deploying ML model as a service or batch job.
- Feedback Loop component for collecting prediction feedback and linking it with actual prediction request and response.
- Connections component for managing credentials for external systems (data storages, code repositories, package repositories, docker registries, etc.) in a secure way. It uses HashiCorp Vault under the hood.
- Deployment automation scripts for deploying ODAHU components to major cloud providers AWS, Azure, GCP.

Interfaces:

- RESTful API
- SDK for ODAHU components API generated from OpenAPI/Swagger specification.
- Web control panel based on ODAHU SDK for interacting with ODAHU components via Web UI.
- Command line interface based on ODAHU SDK for interacting with ODAHU components via terminal commands.

Extensions for external tools:

- Argo Workflow templates based on ODAHU SDK and CLI provide Argo Workflow steps for ODAHU Training, Packaging and Deployment APIs Argo Workflow
- ODAHU Airflow plugin based on SDK provides Airflow operators for ODAHU Training, Packaging and Deployment APIs Apache Airflow
- JupyterLab extension adds UI features to JupyterLab for interacting with ODAHU components.

Distributions

HELM charts

- Release and pre-release **Helm charts** are in github.

Helm chart name	Repository	Description
odahu-flow-fluentd	odahu/odahu-infra	Fluentd with gcp, s3 and abs plugins
odahu-flow-k8s-gke-saa	odahu/odahu-infra	GKE role assigner
odahu-flow-knative	odahu/odahu-infra	Custom knative chart
odahu-flow-monitoring	odahu/odahu-infra	Prometheus, grafana and alertmanager
odahu-flow-opa	odahu/odahu-infra	Open Policy Agent
odahu-flow-tekton	odahu/odahu-infra	Custom tekton chart
odahu-flow-core	odahu/odahu-flow	Core Odahu-flow services
odahu-flow-mlflow	odahu/odahu-trainer	Odahu-flow mlflow toolchain
odahu-flow-packagers	odahu/odahu-packager	Odahu-flow REST packager

Docker Images

Release versions of images are on Docker Hub in the odahu team.

Image name	Repository	Description
odahu-flow-fluentd	odahu/odahu-infra	Fluentd with gcp, s3 and abs plugins
odahu-flow-api	odahu/odahu-flow	Odahu-flow API service
odahu-flow-model-cli	odahu/odahu-flow	Odahu-flow CLI
odahu-flow-model-trainer	odahu/odahu-flow	Trainer helper
odahu-flow-model-packager	odahu/odahu-flow	Packager helper
odahu-flow-service-catalog	odahu/odahu-flow	Swagger for model deployments

odahu-flow-operator	odahu/odahu-flow	Odahu-flow kubernetes orchestrator
odahu-flow-feedback-collector	odahu/odahu-flow	REST API for user feedback service
odahu-flow-feedback-rq-catcher	odahu/odahu-flow	Model deployment request-response catcher
odahu-flow-mlflow-toolchain	odahu/odahu-trainer	Odahu-flow mlflow toolchain
odahu-flow-mlflow-toolchain-gpu	odahu/odahu-trainer	Odahu-flow mlflow toolchain with NVIDIA GPU
odahu-flow-mlflow-tracking-server	odahu/odahu-trainer	MLflow tracking service
odahu-flow-packagers	odahu/odahu-packager	Odahu-flow packagers
base-notebook	odahu/odahu-flow-jupyterlab-plugin	Image with JupyterLab extension based on jupyter/base-notebook
datascience-notebook	odahu/odahu-flow-jupyterlab-plugin	Image with JupyterLab extension based on jupyter/datascience-notebook
tensorflow-notebook	odahu/odahu-flow-jupyterlab-plugin	Image with JupyterLab extension based on jupyter/tensorflow-notebook

Python packages

- Release versions of Python packages are on PyPi: odahu.

Package name	Repository	Description
odahu-flow-cli	odahu/odahu-flow	Odahu-flow CLI
odahu-flow-sdk	odahu/odahu-flow	Odahu-flow SDK
odahu-flow-jupyterlab-plugin	odahu/odahu-flow-jupyterlab-plugin	Jupyterlab with the Odahu-flow plugin
odahu-flow-airflow-plugin	odahu/odahu-airflow-plugin	Odahu-flow Airflow plugin(operators, hooks and so on)

NPM packages

- Release versions of Python packages are on npm in project odahu.

Package name	Repository	Description

odahu-flow-jupyterlab-plugin	odahu/odahu-flow-jupyterlab-plugin	Jupyterlab with the Odahu-flow plugin

Security

Prime goals of ODAHU Security system is to provide authentication, authorization and give users a flexible access control management solution.

The first section Security overview shows the general design of authentication and authorization is described. Look at this section to have a deep understanding of how ODAHU security works under the hood or to learn basic concepts.

The second section Policies describes default security policies for different ODAHU services and how to configure them

Implementation details of ODAHU Security system could be found here

Security overview

Component roles

There are some common terms related to access control management systems. In this documentation, the next ones are commonly used.

Identity Provider (idP)

: A component that provides information about an entity (user or service). In ODAHU the role of idP can do any OpenID Connect compatible provider.

Policy Enforcement Point (PEP)

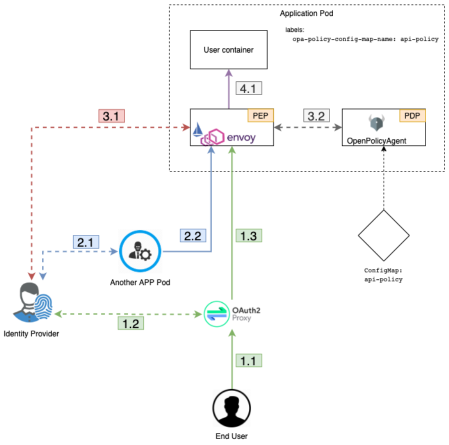
: A component that enforces security policies against each request to API or other protected resources. In ODAHU the role of PEP plays Envoy proxy.

Policy Decision Point (PDP)

: A component that decides whether the request (action in the system) should be permitted or not. In ODAHU role of PDP plays OpenPolicyAgent.

API Request lifecycle

To have a better understanding of how all ODAHU security components work together let’s review the API request lifecycle and describe what is happened for each HTTP request.



1. HTTP Request could be made from the outside of the cluster perimeter.
    1. In this case, the request is handled by OAuth2Proxy via Kubernetes ingress controller
    2. OAuth2Proxy looks up cookies that contain JWT Token issued by idP. If there are no such cookies it redirects the request to idP. After successful login OAuth2Proxy set issued token to cookies (and also to) and send the request to upstream. Before proxying requests to upstream OAuth2Proxy add Authorization Request Header Field from the cookie automatically by setting it from the cookie.
    3. OAuth2Proxy send request to upstream.
2. HTTP Request from inside the cluster perimeter. Such requests usually made by background processes inside the cluster on behalf of service accounts.
    1. Service should previously authenticate in idP using OpenID Connect protocol. The most suitable way to authenticate services is OAuth2 Client Credentials Grant
    2. Service makes a request to API using issued JWT token as Authorization Request Header Field
3. Envoy proxy as PEP that is configured as sidecar container by Istio Pilot for those ODAHU components that must be protected ensures that security policy allows making this request to ODAHU API
    1. Envoy verifies JWT token in Authorization Request Header Field using JSON Web Token (JWT) Authentication filter
    2. Envoy makes a query to OpenPolicyAgent sidecar as PDP using External Authorization filter passing parsed JWT token from the previous step. OpenPolicyAgent sidecars are injected for all pods that have *odahu-flow-authorization=enabled* label
4. If a request is permitted by OpenPolicyAgent, it is sent to upstream (ODAHU API)

UML sequence diagram of a successful API request described above is shown in the image:

Policies

ODAHU is distributed with build-in policies that are written on Rego policy language and included into helm charts of appropriate services.

`Role-based access control`_ is implemented by default for next services

- API
- Feedback aggregator
- *ODAHU deployed ML Models*

ODAHU API and Feedback aggregator policies

Overview

API and Feedback aggregator are distributed with a pre-defined set of OpenPolicyAgent policies. These policies implement simple `Role-based access control`_ (RBAC).

Next features are implemented using Rego policy language:

1. Set of predefined roles with assigned permissions
2. Default mapper that match JWT Claims to attributes that ODAHU RBAC policy expects
3. ODAHU RBAC core policy

These features are implemented in the next files:

- roles.rego – all odahu roles are listed here
- permissions.rego – permissions for roles
- input_mapper.rego – mapper to match JWT Claims to attributes ODAHU RBAC rely on. These attributes include:
  - user – info about user or service who makes the request (this property contains roles attribute with a list of roles)
  - action – HTTP verb of the request
  - resource – URL of the request
- core.rego – core implementation of Role based access control.

All policies customization can be done on the stage of system configuration as described in installation guide

Customize

In this section, different ways to customize pre-defined policies

Extend roles

To define new custom roles, you should add them as a variable in the *roles.rego* file

roles.rego

```
package odahu.roles

admin := "admin"
data_scientist := "data_scientist"
viewer := "viewer"

# new role
connection_manager := "connection_manager"
```

Then you need to set permissions to that role in file *permissions.rego*

permissions.rego

```
1  package odahu.permissions
2
3  import data.odahu.roles
4
5  permissions := {
6    roles.data_scientist: [
7        [".*", "api/v1/model/deployment.*"],
8        [".*", "api/v1/model/packaging.*"],
9        [".*", "api/v1/model/training.*"],
10       ["GET", "api/v1/connection.*"],
11       ["GET", "api/v1/packaging/integration.*"],
12       ["GET", "api/v1/toolchain/integration.*"]
13     ],
14   roles.admin : [
15       [".*", ".*"]
16   ],
17   roles.viewer : [
18       ["GET", ".*"]
19   ],
20   roles.connection_manager : [
21       [".*", "api/v1/connection.*"]
22   ],
23 }
```

In this file, we:

- lines 20-22: add permissions to any request to api/v1/connection.* URL for a new role

Customize default mapper

You can configure *mapper.rego* to extend input that is passed to *core.rego* file with RBAC implementation

mapper.rego

```
package odahu.mapper

import data.odahu.roles

roles_map = {
  "odahu_admin": roles.admin,
  "odahu_data_scientist": roles.data_scientist,
```

```
 8   "odahu_viewer": roles.viewer
 9 }
10
11 jwt = input.attributes.metadata_context.filter_metadata["envoy.filters.http.jwt_authn"].fields.jwt_payload
12
13 keycloak_user_roles[role]{
14   role = jwt.Kind.StructValue.fields.realm_access.Kind.StructValue.fields.roles.Kind.ListValue.values[_].Kind.StringValue
15 }
16
17 user_roles[role]{
18   role = roles_map[keycloak_user_roles[_]]
19 }
20
21 parsed_input = {
22   "action": input.attributes.request.http.method,
23   "resource": input.attributes.request.http.path,
24   "user": {
25     "roles": user_roles
26   }
27 }
```

In this file, we:

- lines 5-9: map roles from jwt claims to policies roles from *roles.rego*
- lines 11-19: extract roles from claims and match them to policies roles
- lines 21-26: create input that is expected by file *core.rego* that contains resource, action and user's roles

Create custom policies

If `Role-based access control`_ is not enough for your purposes you can customize policies to use more general `Attribute-based access control`_. For this purpose, rewrite *core.rego* file or create your own rego policies from scratch

core.rego

```
package odahu.core

import data.odahu.mapper.parsed_input
import data.odahu.permissions.permissions

default allow = false

allow {
  any_user_role := parsed_input.user.roles[_]
    any_permission_of_user_role := permissions[any_user_role][_]
    action := any_permission_of_user_role[0]
    resource := any_permission_of_user_role[1]

    re_match(action, parsed_input.action)
    re_match(resource, parsed_input.resource)
}

allow {
    parsed_input.action == "GET"
    parsed_input.resource == "/"
}

allow {
    parsed_input.action == "GET"
    re_match("/swagger*", parsed_input.resource)
}
```

In this file, we:

- lines 8-16: allow access if there are required permissions for action and resource for at least one user’s roles
- lines 18-21: allow access to root for any user
- lines 23-26: allow access to swagger docs to any user

ODAHU ML Models pods policies

All deployed models contain default policies that permit requests to them for all users that have **Model Deployment Access Role Name**. This role can be set at the model deployment stage using .Spec.roleName key of the ModelDeployment manifest and also statically configured in policies during ODAHU deployment.

Installation

To install ODAHU services, you need to provide a number of preliminary requirements for it.

In particular:

- Python 3.6 or higher; to install JupyterLab extension or Odahuflowctl which are interfaces for interacting with Odahu-flow cluster.
- Kubernetes cluster to perform base and accessory ODAHU services in it, as well as models training, packaging and deployment processes. To be able to use ODAHU services, minimum version of your Kubernetes cluster must be at least 1.16.
- object storage to store models training artifacts and get input data for models (S3, Google Cloud Storage, Azure Blob storage are supported)
- Docker registry (to store resulting Docker images from packagers)

Kubernetes setup

Deploy Kubernetes cluster in Google Compute Platform (GKE)

Prerequisites:

- GCP service account to deploy Kubernetes cluster with and use its credentials for access to object storage and Google Cloud Registry
- Google Cloud Storage bucket (`odahu-flow-test-store` in examples below) to store models output data

Run deploy of a new Kubernetes cluster:

```
$ gcloud container clusters create <cluster-name> \
    --cluster-version 1.13 \
    --machine-type=n1-standard-2 \
    --disk-size=100GB \
    --disk-type=pd-ssd \
    --num-nodes 4 \
    --zone <cluster-region> \
    --project <project-id>
```

Note

Make sure that the disk size on the cluster nodes is sufficient to store images for all services and packaged models. We recommend using a disk size of at least 100 GiB.

You can enable the GPU on your Kubernetes cluster, follow the instructions on how to use GPU hardware accelerators in your GKE clusters’ nodes.

Fetch your Kubernetes credentials for kubectl after cluster is successfully deployed:

```
$ gcloud container clusters get-credentials <cluster-name> \
    --zone <cluster-region> \
    --project <project-id>
```

Deploy Kubernetes cluster in Amazon Web Services (EKS)

Prerequisites

- Resources that are described in AWS documentation
- AWS S3 bucket (`odahu-flow-test-store` in examples below) to store models output data

After you've created VPC and a dedicated security group for it along with Amazon EKS service role to apply to your cluster, you can create a Kubernetes cluster with following command:

```
$ aws eks --region <cluster-region> create-cluster \
    --name <cluster-name> --kubernetes-version 1.13 \
    --role-arn arn:aws:iam::111122223333:role/eks-service-role-AWSServiceRoleForAmazonEKS-EXAMPLEBKZRQR \
    --resources-vpc-config subnetIds=subnet-a9189fe2,subnet-50432629,securityGroupIds=sg-f5c54184
```

Use the AWS CLI `update-kubeconfig` command to create or update `kubeconfig` for your cluster:

```
$ aws eks --region <cluster-region> update-kubeconfig --name <cluster-name>
```

Deploy Kubernetes cluster in Microsoft Azure (AKS)

Prerequisites

- Azure AD Service Principal to interact with Azure APIs and create dynamic resources for an AKS cluster
- Azure Storage account with Blob container (`odahu-flow-test-store` in examples below) to store models output data

First, create a resource group in which all created resources will be placed:

```
$ az group create --location <cluster-region> \
    --name <resource-group-name>
```

Run deploy of a new Kubernetes cluster:

```
$ az aks create --name <cluster-name> \
    --resource-group <resource-group-name>
    --node-vm-size Standard_DS2_v2 \
    --node-osdisk-size 100GB \
    --node-count 4 \
```

```
    --service-principal <service-principal-appid> \
    --client-secret <service-principal-password>
```

Fetch your Kubernetes credentials for kubectl after cluster is successfully deployed:

```
$ az aks get-credentials --name <cluster-name> \
    --resource-group <resource-group-name>
```

Install base Kubernetes services

Install Helm (version 3.1.2)

Install Nginx Ingress

Install `nginx-ingress` Helm chart:

```
$ helm install stable/nginx-ingress --name nginx-ingress --namespace kube-system
```

Get external LoadBalancer IP assigned to `nginx-ingress` service:

```
$ kubectl get -n kube-system svc nginx-ingress-controller \
    -o=jsonpath='{.status.loadBalancer.ingress[*].ip}{"\n"}'
```

Install Istio (with Helm and Tiller)

Note

ODAHU services uses number of Istio custom resources actively, so Istio installation is mandatory.

Add Helm repository for Istio charts

```
$ helm repo add istio https://storage.googleapis.com/istio-release/releases/1.4.2/charts/
```

Crate a namespace for the istio-system components

```
$ kubectl create namespace istio-system
```

Install the `istio-init` chart to bootstrap all the Istio's CustomResourceDefinitions

```
$ helm install istio/istio-init --name istio-init --namespace istio-system
```

Ensure that all `istio-init` jobs have been completed:

```
$ kubectl -n istio-system get job \
    -o=jsonpath='{range.items[?(@.status.succeeded==1)]}{.metadata.name}{"\n"}{end}'
```

Install Istio Helm chart with provided values.

Example:

```
$ cat << EOF > istio_values.yaml
global:
  proxy:
    accessLogFile: "/dev/stdout"
  disablePolicyChecks: false
sidecarInjectorWebhook:
  enabled: true
pilot:
  enabled: true
mixer:
  policy:
    enabled: true
  telemetry:
    enabled: true
  adapters:
    stdio:
      enabled: true
gateways:
  istio-ingressgateway:
    enabled: true
    type: ClusterIP
    meshExpansionPorts: []
    ports:
      - port: 80
        targetPort: 80
        name: http
      - port: 443
        name: https
      - port: 15000
        name: administration
  istio-egressgateway:
    enabled: true
prometheus:
  enabled: false
EOF

$ helm install istio/istio --name istio --namespace istio-system --values ./istio_values.yaml
```

Add ODAHU Helm charts repository

```
$ helm repo add odahu https://raw.githubusercontent.com/odahu/odahu-helm/master
```

Install Knative

Create namespace for Knative and label it for Istio injection:

```
$ kubectl create namespace knative-serving && \
    kubectl label namespace knative-serving istio-injection=enabled
```

Install Knative with Helm chart provided by ODAHU team:

```
$ helm install odahu/odahu-flow-knative --name knative --namespace knative-serving
```

Install Tekton Pipelines

Create namespace for Tekton:

```
$ kubectl create namespace tekton-pipelines
```

Install Tekton Pipelines with Helm chart provided by ODAHU team:

```
$ helm install odahu/odahu-flow-tekton --name tekton --namespace tekton-pipelines
```

Install Fluentd with set of cloud object storage plugins

In order to save models training logs to object storage of cloud provider you use, a container with `fluentd` is used, in which a set of plugins for popular cloud providers' object storages (AWS S3, Google Storage, Azure Blob) is added. Installation is done using a fluentd Helm chart provided by ODAHU team.

First, create an object storage bucket:

```
$ gsutil mb gs://odahu-flow-test-store/
```

Create namespace for Fluentd:

```
$ kubectl create namespace fluentd
```

Install Fluentd with specified values. Full list of values you can see in chart's values.yaml.

Example:

```
$ cat << EOF > fluentd_values.yaml
output:
  target: gcs
  gcs:
    authorization: keyfile
    bucket: odahu-flow-test-store
    project: my-gcp-project-id-zzzzz
    private_key_id: 0bacc0b0caa0a0aacabcacbab0a0b00ababacaab
    private_key: -----BEGIN PRIVATE KEY-----\nprivate-key-here\n-----END PRIVATE KEY-----\n
    client_email: service-account@my-gcp-project-id-zzzzz.iam.gserviceaccount.com
    client_id: 000000000000000000000
    auth_uri: https://accounts.google.com/o/oauth2/auth
    token_uri: https://oauth2.googleapis.com/token
    auth_provider_x509_cert_url: https://www.googleapis.com/oauth2/v1/certs
    client_x509_cert_url: https://www.googleapis.com/robot/v1/metadata/x509/service-account%40my-gcp-project-id-zzzzz.iam.gserviceaccount.com
EOF
```

Install base Kubernetes services

```
$ helm install odahu/odahu-flow-fluentd --name fluentd --namespace fluentd --values ./fluentd_values.yaml
```

Install PostgreSQL (optional)

Create namespace for PostgreSQL:

```
$ kubectl create namespace postgresql
```

Install PostgreSQL Operator with Helm chart:

```
$ helm install postgres-operator/postgres-operator --name odahu-db --namespace postgresql
```

You must configure your PostgreSQL operator using next values

Parameters to configure PostgreSQL Provider:

```
$  cat << EOF > postgres.yaml
apiVersion: "acid.zalan.do/v1"
kind: postgresql
metadata:
  name: odahu
  namespace: postgres
spec:
  teamId: "postgres"
  volume:
    size: 10Gi
  numberOfInstances: 2
  users:
    mlflow: []
    jupyterhub: []
    odahu: []
  databases:
    mlflow: mlflow,
    jupyterhub: jupyterhub,
    odahu: odahu
  postgresql:
    version: "12"
  ---
  apiVersion: v1
  kind: Secret
  metadata:
    name: jupyterhub.odahu-db.credentials.postgresql.acid.zalan.do
    namespace: postgres
  type: Opaque
  ---
  apiVersion: v1
  kind: Secret
  metadata:
    name: mlflow.odahu-db.credentials.postgresql.acid.zalan.do
    namespace: postgres
  type: Opaque
  ---
  apiVersion: v1
  kind: Secret
  metadata:
    name: odahu.odahu-db.credentials.postgresql.acid.zalan.do
    namespace: postgres
  type: Opaque
```

Apply configuration to kubernetes: .. code:: bash kubectl apply -f postgres.yaml

Install Open Policy Agent (optional)

To activate API authentication and authorization using Security install OpenPolicyAgent (OPA) helm chart with ODAHU built-in policies.

Create namespace for OPA

```
$ kubectl create namespace odahu-flow-opa
```

Install OpenPolicyAgent with Helm chart provided by ODAHU team:

```
$ helm install odahu/odahu-flow-opa --name odahu-flow-opa --namespace odahu-flow-opa
```

You must configure your OpenID provider (to allow envoy JWT token verifying) using next Helm values

Parameters to configure OpenID provider

```
# authn overrides configuration of envoy.filters.http.jwt_authn http filter
authn:
  # enabled activate envoy authn filter that verify jwt token and pass parsed data
  # to next filters (particularly to authz)
  oidcIssuer: ""
  oidcJwks: ""
  oidcHost: ""
  localJwks: ""
```

For information about *authn* section parameters see docs for envoy authentication filter

By default chart is delivered with **built-in policies** that implements Role based access system and set of pre-defined roles. To customize some of built-in policies files or define new ones use next Helm values

Parameters to configure built-in policies

```
opa:
  policies: {}
  #  policies:
    #  file1: ".rego policy content encoded as base64"
    #  file2: ".rego policy content encoded as base64"
```

> ***Warning***
>
> Content of rego files defined in values.yaml should be base64 encoded

Install ODAHU

Install core ODAHU services

Create namespace for core ODAHU service:

```
$ kubectl create namespace odahu-flow &&\
    kubectl label namespace odahu-flow project=odahu-flow
```

Create namespaces for ODAHU training, packaging and deployment.

```
$ for i in training packaging deployment; do \
    kubectl create namespace odahu-flow-${i} &&\
    kubectl label namespace odahu-flow-${i} project=odahu-flow; done
```

To provision pods in the deployment namespace according to node selectors and toleration from the config you need to label the namespace so the model deployment webhook use it as a target

```
$ kubectl label namespace odahu-flow-deployment odahu/node-selector-webhook=enabled
```

Deployment namespace should be also labeled for Istio injection.

```
$ kubectl label namespace odahu-flow-deployment istio-injection=enabled
```

Prepare YAML config with values for odahu-flow-core Helm chart.

Example:

```
$  cat << EOF > odahuflow_values.yaml
logLevel: debug
ingress:
  enabled: true
  globalDomain: odahu.example.com
edge:
  ingress:
    enabled: true
    domain: odahu.example.com
feedback:
  enabled: true
config:
  common:
    external_urls:
    - name: Documentation
      url: https://docs.odahu.epam.com
    databaseConnectionString: postgresql://odahu:PASSWORD@odahu-db.postgresql/odahu
  connection:
    enabled: true
    decrypt_token: somenotemptystring
    repository_type: kubernetes
  deployment:
    edge:
      host: http://odahu.example.com
EOF
```

> ***Note***
>
> This example uses hostname `odahu.example.com` as entrypoint for cluster services. It should point to LoadBalancer IP got from Nginx Ingress section.

In order to setup ODAHU services along with ready-to-use **connections**, you may add according section to values YAML in advance.

To support training on GPU, you should provide the GPU node selectors and tolerations:

Example:

Example of Connection GCS:

```yaml
config:
  training:
    gpu_toleration:
      Key: dedicated
      Operator: Equal
      Value: training-gpu
      Effect: NO_SCHEDULE
    gpu_node_selector:
      mode: odahu-flow-training-gpu
```

Examples:

a. Docker registry connection is used to pull/push Odahu packager resulting Docker images to a Docker registry

```
connections:
- id: docker-hub
  spec:
    description: Docker registry for model packaging
    username: "user"
    password: "supersecure"
    type: docker
    uri: docker.io/odahu-models-repo
    webUILink: https://hub.docker.com/r/odahu-models-repo
    vital: true
```

b. Google Cloud Storage connection is used to store model trained artifacts and

```
connections:
- id: models-output
  spec:
    description: Object storage for trained artifacts
    keySecret: '{ "type": "service_account", "project_id": "my-gcp-project-id-zzzzz", "private_key_id": "0bacc0b0caa0a0aacabcacbab0a0b00ababacaab", "private_key": "-----BEGIN PRIVATE KEY-----\nprivate-key-here\n-----END PRIVATE KEY-----\n", "client_email": "service-account@my-gcp-project-id-zzzzz.iam.gserviceaccount.com", "client_id": "000000000000000000000", "auth_uri": "https://accounts.google.com/o/oauth2/auth", "token_uri": "https://oauth2.googleapis.com/token", "auth_provider_x509_cert_url": "https://www.googleapis.com/oauth2/v1/certs", "client_x509_cert_url": "https://www.googleapis.com/robot/v1/metadata/x509/service-account%40my-gcp-project-id-zzzzz.iam.gserviceaccount.com" }'
    region: my-gcp-project-id-zzzzz
    type: gcs
    uri: gs://odahu-flow-test-store/output
    webUILink: https://console.cloud.google.com/storage/browser/odahu-flow-test-store/output?project=my-gcp-project-id-zzzzz
    vital: true
```

If you install **Open Policy Agent** for ODAHU then you will need to configure service accounts which will be used by ODAHU core background services such as **<Trainer>** or **<Packager>**.

All service accounts below require *odahu-admin* ODAHU built-in role. (see more about built-in roles in **security docs**)

Next values with service account credentials are required :

values.yaml

```yaml
config:
```

```
2   operator:
3     # OpenId Provider token url
4     oauth_oidc_token_endpoint: https://oauth2.googleapis.com/token
5     # Credentials from OAuth2 client with Client Credentials Grant
6     client_id: client-id
7     client_secret: client-secret
8
9   trainer:
10    # OpenId Provider token url
11    oauth_oidc_token_endpoint: https://oauth2.googleapis.com/token
12    # Credentials from OAuth2 client with Client Credentials Grant
13    client_id: client-id
14    client_secret: client-secret
15
16  packager:
17    # OpenId Provider token url
18    oauth_oidc_token_endpoint: https://oauth2.googleapis.com/token
19    # Credentials from OAuth2 client with Client Credentials Grant
20    client_id: client-id
21    client_secret: client-secret
22
23 # Service account used to upload odahu resources via odahuflowctl
24 resource_uploader_sa:
25   client_id: some-client-id
26   client_secret: client-secret
27
28 # OpenID provider url
29 oauth_oidc_issuer_url: ""
```

In this file, we:

- lines 2-7: configure service account for **Operator**
- lines 9-14: configure service account for **Trainer**
- lines 16-21: configure service account for **Packager**
- lines 24-29: configure service account Kubernetes Job that install some ODAHU Manifests using ODAHU API

Install odahu-flow core services:

```
$ helm install odahu/odahu-flow-core --name odahu-flow --namespace odahu-flow --values ./odahuflow_values.yaml
```

Training service (MLFlow)

Prepare YAML config with values for odahu-flow-mlflow Helm chart.

```
$ cat << EOF > mlflow_values.yaml
logLevel: debug
ingress:
  globalDomain: example.com
  enabled: true
tracking_server:
  annotations:
    sidecar.istio.io/inject: "false"
toolchain_integration:
  enabled: true
EOF
```

If you install **Open Policy Agent** for ODAHU then you will need to configure service account for a Kubernetes Job that install some ODAHU Manifests using ODAHU API. This Service account should have role *odahu-admin*.

Next values with service account credentials are required :

values.yaml

```yaml
# Service account used to upload odahu resources via odahuflowctl
resource_uploader_sa:
  client_id: some-client-id
  client_secret: client-secret

# OpenID provider url
oauth_oidc_issuer_url: ""
```

Install Helm chart:

```
$ helm install odahu/odahu-flow-mlflow --name odahu-flow-mlflow --namespace odahu-flow \
    --values ./mlflow_values.yaml
```

Packaging service

If you install **Open Policy Agent** for ODAHU then you will need to configure service account for a Kubernetes Job that install some ODAHU Manifests using ODAHU API. This Service account should have role *odahu-admin*.

Next values with service account credentials are required :

values.yaml

```yaml
# Service account used to upload odahu resources via odahuflowctl
resource_uploader_sa:
  client_id: some-client-id
  client_secret: client-secret

# OpenID provider url
oauth_oidc_issuer_url: ""
```

Install odahu-flow-packagers Helm chart:

```
$ helm install odahu/odahu-flow-packagers --name odahu-flow-packagers --namespace odahu-flow
```

Install additional services (optional)

In order to provide additional functionality, ODAHU team also developed several Helm charts to install them into Kubernetes cluster. These are:

- odahu-flow-monitoring - Helm chart providing installation and setup of
  - Prometheus operator - to collect various metrics from models trainings
  - Grafana with set of custom dashboards - to visualize these metrics

- odahu-flow-k8s-gke-saa - Helm chart providing installation and setup of k8s-gke-service-account-assigner service.

Delete ODAHU services

To delete and purge Helm chart run:

```
$ helm delete --purge odahu-flow
```

To clean up remaining `CustomResourceDefinitions` execute following command:

```
$ kubectl get crd | awk '/odahuflow/ {print $1}' | xargs -n1 kubectl delete crd
```

To purge everything installed in previous steps with single command, run

```
$ helm delete --purge odahu-flow-packagers odahu-flow-mlflow odahu-flow &&\
  kubectl delete namespace odahu-flow &&\
  for i in training packaging deployment; do \
    kubectl delete namespace odahu-flow-${i} || true; done &&\
  kubectl get crd | awk '/odahuflow/ {print $1}' | xargs -n1 kubectl delete crd &&\
  kubectl -n istio-system delete job.batch/odahu-flow-feedback-rq-catcher-patcher &&\
  kubectl -n istio-system delete sa/odahu-flow-feedback-rq-catcher-patcher &&\
  kubectl -n istio-system delete cm/odahu-flow-feedback-rq-catcher-patch
```

Conclusion

After successful deployment of a cluster, you may proceed to **Quickstart section** and learn how to perform base ML operations such as **train**, **package** and **deploy** steps.

Cluster Quickstart

In this tutorial you will learn how to Train, Package and Deploy a model from scratch on Odahu. Once deployed, the model serves RESTful requests, and makes a prediction when provided user input.

Odahu's API server performs Train, Package, and Deploy operations for you, using its REST API.

Prerequisites

- Odahu cluster
- MLFlow and **REST API Packager** (installed by default)
- **Odahu-flow CLI** or **Plugin for JupyterLab** (installation instructions: CLI, Plugin)
- JWT token from API (**instructions**)
- Google Cloud Storage bucket on Google Compute Platform
- GitHub repository and an ssh key to connect to it

Tutorial

In this tutorial, you will learn how to:

1. Create an MLFlow project
2. Setup Connections
3. Train a model
4. Package the model
5. Deploy the packaged model
6. Use the deployed model

This tutorial uses a dataset to predict the quality of the wine based on quantitative features like the wine's *fixed acidity*, *pH*, *residual sugar*, and so on.

Code for the tutorial is available on GitHub.

Create MLFlow project

Before	Odahu cluster that meets prerequisites
After	Model code that predicts wine quality

Create a new project folder:

```
$ mkdir wine && cd wine
```

Create a training script:

```
$ touch train.py
```

Paste code into the file:

train.py

```python
import os
import warnings
import sys
import argparse

import pandas as pd
import numpy as np
from sklearn.metrics import mean_squared_error, mean_absolute_error, r2_score
from sklearn.model_selection import train_test_split
from sklearn.linear_model import ElasticNet

import mlflow
import mlflow.sklearn

def eval_metrics(actual, pred):
    rmse = np.sqrt(mean_squared_error(actual, pred))
    mae = mean_absolute_error(actual, pred)
    r2 = r2_score(actual, pred)
    return rmse, mae, r2

if __name__ == "__main__":
    warnings.filterwarnings("ignore")
    np.random.seed(40)

    parser = argparse.ArgumentParser()
    parser.add_argument('--alpha')
    parser.add_argument('--l1-ratio')
    args = parser.parse_args()

    # Read the wine-quality csv file (make sure you're running this from the root of MLflow!)
    wine_path = os.path.join(os.path.dirname(os.path.abspath(__file__)), "wine-quality.csv")
    data = pd.read_csv(wine_path)

    # Split the data into training and test sets. (0.75, 0.25) split.
    train, test = train_test_split(data)

    # The predicted column is "quality" which is a scalar from [3, 9]
    train_x = train.drop(["quality"], axis=1)
    test_x = test.drop(["quality"], axis=1)
    train_y = train[["quality"]]
    test_y = test[["quality"]]

    alpha = float(args.alpha)
    l1_ratio = float(args.l1_ratio)

    with mlflow.start_run():
        lr = ElasticNet(alpha=alpha, l1_ratio=l1_ratio, random_state=42)
        lr.fit(train_x, train_y)

        predicted_qualities = lr.predict(test_x)

        (rmse, mae, r2) = eval_metrics(test_y, predicted_qualities)

        print("Elasticnet model (alpha=%f, l1_ratio=%f):" % (alpha, l1_ratio))
        print("  RMSE: %s" % rmse)
        print("  MAE: %s" % mae)
        print("  R2: %s" % r2)

        mlflow.log_param("alpha", alpha)
        mlflow.log_param("l1_ratio", l1_ratio)
        mlflow.log_metric("rmse", rmse)
        mlflow.log_metric("r2", r2)
```

```
63        mlflow.log_metric("mae", mae)
64        mlflow.set_tag("test", '13')
65
66        mlflow.sklearn.log_model(lr, "model")
67
68        # Persist samples (input and output)
69        train_x.head().to_pickle('head_input.pkl')
70        mlflow.log_artifact('head_input.pkl', 'model')
71        train_y.head().to_pickle('head_output.pkl')
72        mlflow.log_artifact('head_output.pkl', 'model')
```

In this file, we:

- Start MLflow context on line 46
- Train `ElasticNet` model on line 48
- Set metrics, parameters and tags on lines 59-64
- Save model with name `model` (model is serialized and sent to the MLflow engine) on line 66
- Save input and output samples (for persisting information about input and output column names) on lines 69-72

Create an MLproject file:

```
$ touch MLproject
```

Paste code into the file:

MLproject

```
name: wine-quality-example
conda_env: conda.yaml
entry_points:
    main:
        parameters:
            alpha: float
            l1_ratio: {type: float, default: 0.1}
        command: "python train.py --alpha {alpha} --l1-ratio {l1_ratio}"
```

> **Note**
>
> *Read more about MLproject structure on the* official MLFlow docs.

Create a conda environment file:

```
$ touch conda.yaml
```

Paste code to the created file:

conda.yaml

```
name: example
channels:
```

```
  - defaults
dependencies:
  - python=3.6
  - numpy=1.14.3
  - pandas=0.22.0
  - scikit-learn=0.19.1
  - pip:
    - mlflow==1.0.0
```

> **Note**
>
> All python packages that are used in training script must be listed in the conda.yaml file.
>
> *Read more about conda environment on the* official conda docs.

Make directory “data” and download the wine data set:

```
$ mkdir ./data
$ wget https://raw.githubusercontent.com/odahu/odahu-examples/develop/mlflow/sklearn/wine/data/wine-quality.csv -O ./data/wine-quality.csv
```

After this step the project folder should look like this:

```
.
├── MLproject
├── conda.yaml
├── data
│   └── wine-quality.csv
└── train.py
```

Setup connections

Before	Odahu cluster that meets prerequisites
After	Odahu cluster with **Connections**

Odahu Platform uses the concept of **Connections** to manage authorizations to external services and data.

This tutorial requires three Connections:

- A GitHub repository, where the code is located
- A Google Cloud Storage folder, where input data is located (wine-quality.csv)
- A Docker registry, where the trained and packaged model will be stored for later use

You can find more detailed documentation about a connection configuration here.

Create a Connection to GitHub repository

Because odahu-examples repository already contains the required code we will just use this repository. But feel free to create and use a new repository if you want.

Odahu is REST-powered, and so we encode the REST “payloads” in this tutorial in YAML files. Create a directory where payloads files will be staged:

```
$ mkdir ./odahu-flow
```

Create payload:

```
$ touch ./odahu-flow/vcs_connection.odahu.yaml
```

Paste code into the created file:

vcs_connection.odahu.yaml

```yaml
kind: Connection
id: odahu-flow-tutorial
spec:
  type: git
  uri: git@github.com:odahu/odahu-examples.git
  reference: origin/master
  keySecret: <paste here your base64-encoded key github ssh key>
  description: Git repository with odahu-flow-examples
  webUILink: https://github.com/odahu/odahu-examples
```

> ***Note***
>
> Read more about GitHub ssh keys

Create a Connection using the **Odahu-flow CLI**:

```
$ odahuflowctl conn create -f ./odahu-flow/vcs_connection.odahu.yaml
```

Or create a Connection using **Plugin for JupyterLab**:

1. Open jupyterlab (available by <your.cluster.base.address>/jupyterhub);
2. Navigate to ‘File Browser’ (folder icon)
3. Select file `./odahu-flow/vcs_connection.odahu.yaml` and in context menu press `submit` button;

Create **Connection** *to wine-quality.csv object storage*

Create payload:

```
$ touch ./odahu-flow/wine_connection.odahu.yaml
```

Paste this code into the file:

wine_connection.odahu.yaml

```yaml
kind: Connection
id: wine-tutorial
spec:
  type: gcs
  uri: gs://<paste your bucket address here>/data-tutorial/wine-quality.csv
  region: <paste region here>
  keySecret: <paste base64-encoded key secret here>  # should be enclosed in single quotes
  description: Wine dataset
```

Create a connection using the **Odahu-flow CLI** or **Plugin for JupyterLab**, as in the previous example.

If wine-quality.csv is not in the GCS bucket yet, use this command:

```
$ gsutil cp ./data/wine-quality.csv gs://<bucket-name>/data-tutorial/
```

Create a Connection *to a docker registry*

Create payload:

```
$ touch ./odahu-flow/docker_connection.odahu.yaml
```

Paste this code into the file:

docker_connection.odahu.yaml

```yaml
kind: Connection  # type of payload
id: docker-tutorial
spec:
  type: docker
  uri: <past uri of your registry here>  # uri to docker image registry
  username: <paste your username here>
  password: <paste your base64-encoded password here>
  description: Docker registry for model packaging
```

Create the connection using **Odahu-flow CLI** or **Plugin for JupyterLab**, as in the previous example.

Check that all Connections were created successfully:

```
- id: docker-tutorial
    description: Docker repository for model packaging
    type: docker
- id: odahu-flow-tutorial
    description: Git repository with odahu-flow-tutorial
    type: git
- id: models-output
    description: Storage for trainined artifacts
    type: gcs
- id: wine-tutorial
    description: Wine dataset
    type: gcs
```

Congrats! You are now ready to train the model.

Train the model

Before	Project code, hosted on GitHub
After	Trained **GPPI** model (a **Trained Model Binary**)

Create payload:

```
$ touch ./odahu-flow/training.odahu.yaml
```

Paste code into the file:

./odahu-flow/training.odahu.yaml

```yaml
1 kind: ModelTraining
2 id: wine-tutorial
3 spec:
4   model:
5     name: wine
6     version: 1.0
7   toolchain: mlflow  # MLFlow training toolchain integration
8   entrypoint: main
9   workDir: mlflow/sklearn/wine  # MLproject location (in GitHub)
10  data:
11    - connection: wine-tutorial
12      # Where to save a local copy of wine-quality.csv from wine-tutorial GCP connection
13      localPath: mlflow/sklearn/wine/wine-quality.csv
14  hyperParameters:
15    alpha: "1.0"
16  resources:
17    limits:
18       cpu: 4
19       memory: 4Gi
20    requests:
21       cpu: 2
22       memory: 2Gi
23   algorithmSource:
24     vcs:
25       connection: odahu-flow-tutorial
```

In this file, we:

- line 7: Set Odahu toolchain's name to mlflow
- line 8: Reference `main` method in `entry_points` (which is defined for MLproject files)
- line 9: Point `workDir` to the MLFlow project directory. (This is the directory that has the MLproject in it.)
- line 10: A section defining input data
- line 11: `connection` id of the wine_connection.odahu.yaml (created in the previous step)
- line 13: `localPath` relative (to Git repository root) path of the data file at the training (docker) container where data were put
- lines 14-15: Input hyperparameters, defined in MLProject file, and passed to `main` method
- line 23: A section defining training source code

- line 24: `vcs` if source code located in a repository and `objectStorage` if in a storage. Should not use both
- line 25: id of the vcs_connection.odahu.yaml (created in the previous step)

Train using **Odahu-flow CLI**:

```
$ odahuflowctl training create -f ./odahu-flow/training.odahu.yaml
```

Check **Train** logs:

```
$ odahuflowctl training logs --id wine-tutorial
```

The **Train** process will finish after some time.

To check the status run:

```
$ odahuflowctl training get --id wine-tutorial
```

When the Train process finishes, the command will output this YAML:

- `state` succeeded
- `artifactName` (filename of **Trained Model Binary**)

Or *Train* using the **Plugin for JupyterLab**:

1. Open jupyterlab
2. Open cloned repo, and then the folder with the project
3. Select file `./odahu-flow/training.odahu.yaml` and in context menu press `submit` button

You can see model logs using `Odahu cloud mode` in the left side tab (cloud icon) in Jupyterlab

1. Open `Odahu cloud mode` tab
2. Look for `TRAINING` section
3. Press on the row with *ID=wine*
4. Press button `LOGS` to connect to **Train** logs

After some time, the **Train** process will finish. Train status is updated in column `status` of the *TRAINING* section in the `Odahu cloud mode` tab. If the model training finishes with success, you will see *status=succeeded*.

Then open **Train** again by pressing the appropriate row. Look at the *Results* section. You should see:

- `artifactName` (filename of **Trained Model Binary**)

`artifactName` is the filename of the trained model. This model is in **GPPI** format. We can download it from storage defined in the `models-output` Connection. (This connection is created during Odahu Platform installation, so we were not required to create this Connection as part of this tutorial.)

Package the model

Before	The trained model in **GPPI** **Trained Model Binary**
After	Docker image for the packaged model, including a model REST API

Create payload:

```
$ touch ./odahu-flow/packaging.odahu.yaml
```

Paste code into the file:

./odahu-flow/packaging.odahu.yaml

```
1 kind: ModelPackaging
2 id: wine-tutorial
3 spec:
4   artifactName: "<fill-in>"  # Use artifact name from Train step
5   targets:
6     - connectionName: docker-tutorial  # Docker registry when output image will be stored
7       name: docker-push
8   integrationName: docker-rest  # REST API Packager
```

In this file, we:

- line 4: Set to artifact name from the Train step
- line 6: Set to docker registry, where output will be staged
- line 7: Specify the docker command
- line 8: id of the **REST API Packager**

Create a **Package** using **Odahu-flow CLI**:

```
$ odahuflowctl packaging create -f ./odahu-flow/packaging.odahu.yaml
```

Check the **Package** logs:

```
$ odahuflowctl packaging logs --id wine-tutorial
```

After some time, the **Package** process will finish.

To check the status, run:

```
$ odahuflowctl packaging get --id wine-tutorial
```

You will see YAML with updated **Package** resource. Look at the status section. You can see:

- `image` # This is the filename of the Docker image in the registry with the trained model prediction, served via REST`.

Or run Package using the **Plugin for JupyterLab**:

1. Open jupyterlab
2. Open the repository that has the source code, and navigate to the folder with the MLProject file
3. Select file `./odahu-flow/packaging.odahu.yaml` and in the context menu press the `submit` button

To view Package logs, use `Odahu cloud mode` in the side tab of your Jupyterlab

1. Open `Odahu cloud mode` tab
2. Look for `PACKAGING` section
3. Click on the row with *ID=wine*
4. Click the button for `LOGS` and view the `Packaging` logs

After some time, the **Package** process will finish. The status of training is updated in column `status` of the *PACKAGING* section in the `Odahu cloud mode` tab. You should see *status=succeeded*.

Then open PACKAGING again by pressing the appropriate row. Look at the *Results* section. You should see:

- `image` (this is the filename of docker image in the registry with the trained model as a REST service`);

Deploy the model

Before	Model is packaged as image in the Docker registry
After	Model is served via REST API from the Odahu cluster

Create payload:

```
$ touch ./odahu-flow/deployment.odahu.yaml
```

Paste code into the file:

./odahu-flow/deployment.odahu.yaml

```
1 kind: ModelDeployment
2 id: wine-tutorial
3 spec:
4   image: "<fill-in>"
5   predictor: odahu-ml-server
6   minReplicas: 1
7   imagePullConnectionID: docker-tutorial
```

In this file, we:

- line 4: Set the `image` that was created in the Package step
- line 5: Set the `predictor` that indicates what Inference Server is used in the image; Check `Predictors`_ for more;

- line 7: Set the connection ID to access the container registry where the image lives

Create a **Deploy** using the **Odahu-flow CLI**:

```
$ odahuflowctl deployment create -f ./odahu-flow/deployment.odahu.yaml
```

After some time, the **Deploy** process will finish.

To check its status, run:

```
$ odahuflowctl deployment get --id wine-tutorial
```

Or create a *Deploy* using the **Plugin for JupyterLab**:

1. Open jupyterlab
2. Open the cloned repo, and then the folder with the MLProject file
3. Select file `./odahu-flow/deployment.odahu.yaml`. In context menu press the `submit` button

You can see Deploy logs using the `Odahu cloud mode` side tab in your Jupyterlab

1. Open the `Odahu cloud mode` tab
2. Look for the `DEPLOYMENT` section
3. Click the row with *ID=wine*

After some time, the **Deploy** process will finish. The status of Deploy is updated in column `status` of the *DEPLOYMENT* section in the `Odahu cloud mode` tab. You should see *status=Ready*.

Use the deployed model

Step input data	The deployed model

After the model is deployed, you can check its API in Swagger:

Open `<your-odahu-platform-host>/service-catalog/swagger/index.html` and look and the endpoints:

1. `GET /model/wine-tutorial/api/model/info` – OpenAPI model specification;
2. `POST /model/wine-tutorial/api/model/invoke` – Endpoint to do predictions;

But you can also do predictions using the **Odahu-flow CLI**.

Create a payload file:

```
$ touch ./odahu-flow/r.json
```

Add payload for `/model/wine-tutorial/api/model/invoke` according to the OpenAPI schema. In this payload we provide values for model input variables:

./odahu-flow/r.json

```
{
  "columns": [
    "fixed acidity",
    "volatile acidity",
    "citric acid",
    "residual sugar",
    "chlorides",
    "free sulfur dioxide",
    "total sulfur dioxide",
    "density",
    "pH",
    "sulphates",
    "alcohol"
  ],
  "data": [
    [
      7,
      0.27,
      0.36,
      20.7,
      0.045,
      45,
      170,
      1.001,
      3,
      0.45,
      8.8
    ]
  ]
}
```

Invoke the model to make a prediction:

```
$ odahuflowctl model invoke --mr wine-tutorial --json-file r.json
```

./odahu-flow/r.json

```
{"prediction": [6.0], "columns": ["quality"]}
```

Congrats! You have completed the tutorial.

Local Quickstart

In this tutorial, we will walk through the training, packaging and serving of a machine learning model locally by leveraging ODAHUFlow's main components.

Prerequisites

- Docker engine (at least version 17.0) with access from current user (*docker ps* should executes without errors)
- **Odahu-flow CLI**
- git
- wget

Tutorial

We will consider the wine model from Cluster Quickstart. But now, we will train, package and deploy the model locally.

> ***Note***
>
> Code for the tutorial is available on GitHub.

odahuflowctl has commands for local training and packaging.

```
$ odahuflowctl local --help
```

To train a model locally, you have to provide an ODAHU model training manifest and training toolchain. *odahuflowctl* tries to find them on your local filesystem. If it can not do it, then the CLI requests to ODAHU API.

Local training arguments:

```
  --train-id, --id TEXT       Model training ID  [required]
  -f, --manifest-file PATH    Path to a ODAHU-flow manifest file
  -d, --manifest-dir PATH     Path to a directory with ODAHU-flow manifests
```

The *mlflow/sklearn/wine/odahuflow* directory already contains training manifest file for wine model. If we don't have a running ODAHUFlow API server, we should create toolchain manifest manually.

Paste the toolchain manifest into the *mlflow/sklearn/wine/odahuflow/toolchain.yaml* file:

```
kind: ToolchainIntegration
id: mlflow
```

```yaml
spec:
  defaultImage: "odahu/odahu-flow-mlflow-toolchain:1.1.0-rc11"
  entrypoint: /opt/conda/bin/odahu-flow-mlflow-runner
```

We are ready to launch the local training. Copy, past and execute the following command.

```
$ odahuflowctl local train run -d mlflow/sklearn/wine/odahuflow --id wine
```

> **Warning**
>
> MLFlow metrics does not propagate to the tracking server during training. This will be implemented in the near future.

odahuflowctl trains the model, verify that it satisfy the GPPI spec and save GPPI binary in the host filesystem. Execute the following command to take a look at all trained models in the default output directory.

```
$ odahuflowctl local train list
```

Our next step is to package the trained model to a REST service. Like for local training, local packaging requires a model packaging and packaging integration manifests.

Local packaging arguments:

```
  --pack-id, --id TEXT          Model packaging ID  [required]
  -f, --manifest-file PATH      Path to a ODAHU-flow manifest file
  -d, --manifest-dir PATH       Path to a directory with ODAHU-flow manifest files
  --artifact-path PATH          Path to a training artifact
  -a, --artifact-name TEXT      Override artifact name from file
```

Paste the packaging integration manifest into the *mlflow/sklearn/wine/odahuflow/packager.yaml* file:

```yaml
kind: PackagingIntegration
id: docker-rest
spec:
  entrypoint: "/usr/local/bin/odahu-flow-pack-to-rest"
  defaultImage: "odahu/odahu-flow-packagers:1.1.0-rc11"
  privileged: true
  schema:
    targets:
      - name: docker-push
        connectionTypes: ["docker", "ecr"]
        required: true
      - name: docker-pull
        connectionTypes: ["docker", "ecr"]
        required: false
    arguments:
      properties:
        - name: dockerfileAddCondaInstallation
          parameters:
            - name: description
```

```yaml
            value: Add conda installation code to training.Dockerfile
          - name: type
            value: boolean
          - name: default
            value: true
      - name: dockerfileBaseImage
        parameters:
          - name: description
            value: Base image for training.Dockerfile.
          - name: type
            value: string
          - name: default
            value: 'odahu/odahu-flow-docker-packager-base:1.1.0-rc11'
      - name: dockerfileCondaEnvsLocation
        parameters:
          - name: description
            value: Conda env location in training.Dockerfile.
          - name: type
            value: string
          - name: default
            value: /opt/conda/envs/
      - name: host
        parameters:
          - name: description
            value: Host to bind.
          - name: type
            value: string
          - name: default
            value: 0.0.0.0
      - name: port
        parameters:
          - name: description
            value: Port to bind.
          - name: type
            value: integer
          - name: default
            value: 5000
      - name: timeout
        parameters:
          - name: description
            value: Serving timeout in seconds.
          - name: type
            value: integer
          - name: default
            value: 60
      - name: workers
        parameters:
          - name: description
            value: Count of serving workers.
          - name: type
            value: integer
          - name: default
            value: 1
      - name: threads
        parameters:
          - name: description
            value: Count of serving threads.
          - name: type
            value: integer
          - name: default
            value: 4
      - name: imageName
        parameters:
          - name: description
            value: |
              This option provides a way to specify the Docker image name. You can hardcode the full name or specify a template. Available template values:
                - Name (Model Name)
                - Version (Model Version)
                - RandomUUID
              The default value is '{{ Name }}/{{ Version }}:{{ RandomUUID }}'.
              Image name examples:
                - myservice:123
                - {{ Name }}:{{ Version }}
          - name: type
            value: string
          - name: default
            value: "{{ Name }}-{{ Version }}:{{ RandomUUID }}"
```


Choose the name of trained artifact and execute the following command:

```
$ odahuflowctl --verbose local pack run -d mlflow/sklearn/wine/odahuflow --id wine -a wine-1.0-wine-1.0-01-Mar-2020-18-33-35
```

The last lines of output must contains a name of model REST service.

At the last step, we run our REST service and make a predict.

```
$ docker run -it --rm -p 5000:5000 wine-1.0:cbf184d0-4b08-45c4-8efb-17e28a3b537e
```

```
$ odahuflowctl model invoke --url http://0:5000 --json-file mlflow/sklearn/wine/odahuflow/request.json
```

Invoke ODAHU models for prediction

You want to call the model that was deployed on ODAHU programmatically

You can call ODAHU models using *REST API* or using *Python SDK*

Python SDK

1. Install python SDK

```
pip install odahu-flow-sdk
```

2. Configure SDK

By default SDK config is located in ~/.odahuflow/config

But you can override it location using *ODAHUFLOW_CONFIG* environment variable

Configure next values in the config

```
[general]
api_url = https://replace.your.models.host
api_issuing_url = https://replace.your.oauth2.token.url
```

3. In python use ModelClient to invoke models

```
from odahuflow.sdk.clients.model import ModelClient, calculate_url
from odahuflow.sdk.clients.api import RemoteAPIClient
from odahuflow.sdk import config

# Change model deployment name to model name which you want to invoke
MODEL_DEPLOYMENT_NAME = "<model-deployment-name>"

# Get api token using client credentials flow via Remote client
remote_api = RemoteAPIClient(client_id='<your-client-id>', client_secret='<your-secret>')
remote_api.info()

# Build model client and invoke models
client = ModelClient(
    calculate_url(config.API_URL, model_deployment=MODEL_DEPLOYMENT_NAME),
    remote_api.authenticator.token
)

# Get swagger specification of model service
print(client.info())

# Invoke model
print(client.invoke(columns=['col1', 'col2'], data=[
    ['row1_at1', 'row1_at2'],
    ['row2_at1', 'row2_at2'],
]))
```

REST

If you use another language you can use pure REST to invoke models

You should get token by yourself using OpenID provider and OAuth2 Client Credentials Grant

Then call ODAHU next way

To get the swagger definition of model service

```
curl -X GET "https://replace.your.models.host/model/${MODEL_DEPLOYMENT_NAME}/api/model/info" \
            -H "accept: application/json" \
            -H "Authorization: Bearer <token>"
```

To invoke the model

```
curl -X POST "https://replace.your.models.host/model/${MODEL_DEPLOYMENT_NAME}/api/model/invoke" \
            -H "accept: application/json" \
            -H "Authorization: Bearer <token>" \
            -d @body.json
```

API

API service manages Odahu Platform entities.

- **Connections**
- **Trainings**
- **Packaging**
- **Deployments**

API service can provide the following data, when queried:

- Model Train and Deploy logs
- Model **Trainer Metrics**
- Model **Trainer Tags**

API-provided URLs

All information about URLs that **API service** provides can be viewed using the auto-generated, interactive Swagger page. It is located at `<api-address>/swagger/index.html`. You can read all of the up-to-date documentation and invoke all methods (allowed for your account) right from this web page.

Authentication and authorization

API service distributed in odahu-flow-core helm chart with enabled authorization and pre-defined OPA policies. If **Security Subsystem** is installed, then all requests to API service will be enforced using pre-defined OPA policies.

Implementation details

API service is a REST server, written in GoLang. For easy integration, it provides a Swagger endpoint with up-to-date protocol information.

Technologies used	GoLang
Distribution representation	Docker Image
Source code location	packages/operator
Can be used w/o Odahu Platform?	Yes
Does it connect to other services?	Yes (Kubernetes API)
Can it be deployed locally?	If a local Kubernetes cluster is present
Does it provide any interface?	Yes (HTTP REST API)

Feedback aggregator

Feedback aggregator is a service that provides a **Model Feedback API** and gathers input and output **prediction requests**

API-provided URLs

Model Feedback API provide just single endpoint that allow you send feedback on a **prediction request**:

POST /api/v1/feedback

Information about this URL can be viewed using the auto-generated, interactive Swagger page. It is located at `<api-address>/swagger/index.html`. You can read all of the up-to-date documentation and invoke this endpoint (allowed for your account) right from this web page.

Authentication and authorization

Feedback aggregator distributed in odahu-flow-core helm chart with enabled authorization and pre-defined OPA policies. If **Security Subsystem** is installed, then all requests to **Model Feedback API** service will be enforced using pre-defined OPA policies.

Implementation details

Feedback aggregator contains two major subcomponents

- REST Server provides **Model Feedback API** and sends them to configured fluentd server
- Envoy Proxy tap filter catches all requests and responses of deployed models and sends this info to configured fluentd server

Technologies used	GoLang, Envoy Proxy
Distribution representation	Docker Image
Source code location	packages/operator
Can be used w/o Odahu Platform?	No
Does it connect to other services?	Yes (Fluentd, Envoy Proxy)
Can it be deployed locally?	If a local Kubernetes cluster is present
Does it provide any interface?	Yes (HTTP REST API)

Operator

Operator monitors Odahu-provided Kubernetes (K8s) Custom Resources. This gives Operator the ability to manage Odahu entities using K8s infrastructure (Secrets, Pods, Services, etc). The K8s entities that belong to Odahu are referred to as **Odahu-flow's CRDs**.

Operator is a mandatory component in Odahu clusters.

Implementation details

Operator is a Kubernetes Operator, written using Kubernetes Go packages.

Technologies used	GoLang
Distribution representation	Docker Image
Source code location	packages/operator
Can be used w/o Odahu Platform?	Yes
Does it connect to another services?	Yes (Kubernetes API)
Can be deployed locally?	If local Kubernetes cluster is present
Does it provide any interface?	No

MLFlow Trainer

Odahu provides a **Trainer Extension** for the popular MLflow framework.

This allows model **Training** in Python, and provides support for MLflow APIs. Trained models are packaged using the **General Python Prediction Interface**.

Limitations

- Odahu supports Python (v. 3) libraries (e.g. Keras, Sklearn, TensorFlow, etc.)
- MLeap is not supported
- Required packages (system and python) must be declared in a conda environment file
- Train must save only one model, using one MLproject entry point method. Otherwise an exception will occur
- Input and output columns should be mapped to the specially-named `head_input.pkl` and `head_output.pkl` files to make it into the Packaged artifact
- Training code should avoid direct usage of MLflow client libraries

Implementation Details

Support	official
Language	Python 3.6+

Source code is available on GitHub.

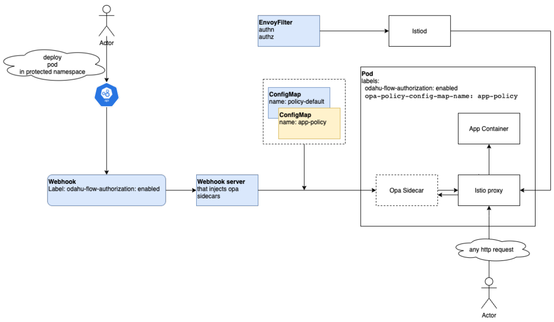
Low-level integration details are provided here.

Security subsystem

Security subsystem is distributed as a helm chart and relies on OpenPolicyAgent to make decisions about authorization and Istio to enforce authorization for requests to the protected services.

You can read about ODAHU security concepts in an appropriate docs section

Implementation details



Helm chart deploys

- Webhook server that injects OpenPolicyAgent sidecars into pods that labeled by `odahu-flow-authorization=enabled`
- EnvoyFilter that configures Istio-proxy sidecars located in pods labeled by `odahu-flow-authorization=enabled` to force authentication and authorization for all incoming http requests
- ConfigMap with common policy that included into all OpenPolicyAgent sidecars and implements masking of a sensitive data
- ConfigMap with default policy that included into OpenPolicyAgent sidecars when pod does not specify ConfigMap with its polices

When the webhook server deploys OpenPolicyAgent sidecar, it attaches policies from ConfigMap. ConfigMap can be found by the value of the pod label `opa-policy-config-map-name`. If this label is missed, then the default policy will be used. Default policy – reject all requests.

Different ODAHU components such as API and Feedback aggregator are distributed with a pre-defined set of OpenPolicyAgent policies. They create ConfigMap with their policies during the deployment process.

If you change ConfigMap with policies then the appropriate pod must be restarted to refresh its policies.

Technologies used	OpenPolicyAgent, Istio
Distribution representation	Helm chart
Source code location	packages/operator
Can be used w/o Odahu Platform?	No
Does it connect to other services?	Yes (Kubernetes, OpenPolicyAgent, Istio)
Can it be deployed locally?	If a local Kubernetes cluster is present

Does it provide any interface?	No

Metrics

Odahu is pluggable and can integrate with a variety of metrics monitoring tools, allowing monitoring for:

- Model training metrics
- Model performance metrics
- System metrics (e.g. operator counters)

Odahu’s installation Helm chart boostraps a Prometheus operator to persist metrics and Grafana dashboard to display them.

Alternative integrations can be similarly constructed that swap in other monitoring solutions.

Airflow

Odahu-flow provides a set of custom operators that allow you to interact with a Odahu cluster using Apache Airflow

Connections

The Airflow plugin should be authorized by Odahu. Authorization is implemented using regular Airflow Connections

All custom Odahu-flow operators accept *api_connection_id* as a parameter that refers to *Odahu-flow Connection*

Odahu-flow Connection

The Odahu connection provides access to a Odahu cluster for Odahu custom operators.

Configuring the Connection

Host (required)

The host to connect to. Usually available at: *odahu.<cluster-base-url>*

Type (required)

HTTP

Schema (optional)

https

Login (not required)

Leave this field empty

Password (required)

The client secret. The client MAY omit the parameter if the client secret is an empty string. See more

Extra (Required)

Specify the extra parameters (as json dictionary) that can be used in Odahu connection. Because Odahu uses OpenID authorization, additional OpenID/OAuth 2.0 parameters may be supplied here.

The following parameters are supported and must be defined:

- **auth_url**: url of authorization server
- **client_id**: The client identifier issued to the client during the registration process. See more
- **scope**: Access Token Scope

Example “extras” field:

```
{
    "auth_url": "https://keycloak.<my-app-domain>",
    "client_id": "my-app",
    "scope": "openid profile email offline_access groups",
}
```

Custom operators

This chapter describes the custom operators provided by Odahu.

Train, Pack, Deploy operators

class **TrainingOperator** (training=None, api_connection_id=None, *args, **kwargs)

The operator that runs **Train** phase

Use *args* and *kwargs* to override other operator parameters

Parameters:

- **training** (*odahuflow.sdk.models.ModelTraining*) – describes the **Train** phase
- **api_connection_id** (*str*) – conn_id of Odahu-flow Connection

class **TrainingSensor** (training_id=None, api_connection_id=None, *args, **kwargs)

The operator that waits for **Train** phase is finished

Use *args* and *kwargs* to override other operator parameters

Parameters:

- **training_id** (*str*) – *Train* id waits for
- **api_connection_id** (*str*) – conn_id of Odahu-flow Connection

class **PackagingOperator** (packaging=None, api_connection_id=None, trained_task_id: str = "", *args, **kwargs)

The operator that runs **Package** phase

Use *args* and *kwargs* to override other operator parameters

Parameters:

- **packaging** (*odahuflow.sdk.models.ModelPackaging*) – describes the **Package** phase
- **api_connection_id** (*str*) – conn_id of Odahu-flow Connection
- **trained_task_id** (*str*) – finished task id of TrainingSensor

class **PackagingSensor** (training_id=None, api_connection_id=None, *args, **kwargs)

The operator that waits for **Package** phase is finished

Use *args* and *kwargs* to override other operator parameters

Parameters:

- **packaging_id** (*str*) – *Package* id waits for
- **api_connection_id** (*str*) – conn_id of Odahu-flow Connection

class **DeploymentOperator** (deployment=None, api_connection_id=None, *args, **kwargs)

The operator that runs **Deploy** phase

Use *args* and *kwargs* to override other operator parameters

Parameters:
- **packaging** (*odahuflow.sdk.models.ModelDeployment*) – describes the **Deploy** phase
- **api_connection_id** (*str*) – conn_id of Odahu-flow Connection
- **packaging_task_id** (*str*) – finished task id of PackagingSensor

class **DeploymentSensor** (training_id=None, api_connection_id=None, *args, **kwargs)

The operator that waits for **Deploy** phase is finished

Use *args* and *kwargs* to override other operator parameters

Parameters:
- **deployment_id** (*str*) – *Deploy* id waits for
- **api_connection_id** (*str*) – conn_id of Odahu-flow Connection

Model usage operators

These operators are used to interact with deployed models.

class **ModelInfoRequestOperator** (self, model_deployment_name: str, api_connection_id: str, model_connection_id: str, md_role_name: str = "", *args, **kwargs)

The operator what extract metadata of deployed model.

Use *args* and *kwargs* to override other operator parameters

Parameters:
- **model_deployment_name** (*str*) – Model deployment name
- **api_connection_id** (*str*) – conn_id of Odahu-flow Connection
- **model_connection_id** (*str*) – id of Odahu **Connection** for deployed model access
- **md_role_name** (*str*) – Role name

class **ModelPredictRequestOperator** (self, model_deployment_name: str, api_connection_id: str, model_connection_id: str, request_body: typing.Any, md_role_name: str = "", *args, **kwargs)

The operator request prediction using deployed model.

Use *args* and *kwargs* to override other operator parameters

Parameters:
- **model_deployment_name** (*str*) – <paste>
- **api_connection_id** (*str*) – conn_id of Odahu-flow Connection
- **model_connection_id** (*str*) – id of Odahu **Connection** for deployed model access
- **request_body** (*dict*) – JSON Body with model parameters
- **md_role_name** (*str*) – Role name

Helper operators

These operators are helpers to simplify using Odahu-flow.

class **GcpConnectionToOdahuConnectionOperator** (self, api_connection_id: str, google_cloud_storage_conn_id: str, conn_template: typing.Any, *args, **kwargs)

Create Odahu-flow Connection using GCP Airflow Connection

Use *args* and *kwargs* to override other operator parameters

Parameters:
- **api_connection_id** (*str*) – conn_id of Odahu-flow Connection
- **google_cloud_storage_conn_id** (*str*) – conn_id to Gcp Connection
- **conn_template** (*odahuflow.sdk.models.connection.Connection*) – Odahu-flow Connection template

How to describe operators

When you initialize Odahu custom operators such as `TrainingOperator`, `PackagingOperator`, or `DeploymentOperator` you should pass odahu resource payload as a parameter.

Actually, this is a payload that describes a resource that will be created at Odahu-flow cluster. You should describe such payloads using odahuflow.sdk models

Creating training payload

```
training = ModelTraining(
    id=training_id,
    spec=ModelTrainingSpec(
        model=ModelIdentity(
            name="wine",
            version="1.0"
        ),
        toolchain="mlflow",
        entrypoint="main",
        work_dir="mlflow/sklearn/wine",
        hyper_parameters={
            "alpha": "1.0"
        },
        data=[
            DataBindingDir(
```

```
                conn_name='wine',
                local_path='mlflow/sklearn/wine/wine-quality.csv'
            ),
        ],
        resources=ResourceRequirements(
            requests=ResourceList(
                cpu="2024m",
                memory="2024Mi"
            ),
            limits=ResourceList(
                cpu="2024m",
                memory="2024Mi"
            )
        ),
        vcs_name="odahu-flow-examples"
    ),
)
```

But if you did some RnD work with Odahu-flow previously, it's likely that you already have yaml/json files that describe the same payloads. You can reuse them to create odahuflow.sdk models automatically

Using plain yaml/json text

```
from odahuflow.airflow.resources import resource

packaging_id, packaging = resource("""
id: airlfow-wine
kind: ModelPackaging
spec:
  artifactName: "<fill-in>"
  targets:
    - connectionName: docker-ci
      name: docker-push
  integrationName: docker-rest
""")
```

Or refer to yaml/json files that must be located at Airflow DAGs folder or Airflow Home folder (these folders are configured at airflow.cfg file)

Creating training payload

```
from odahuflow.airflow.resources import resource
training_id, training = resource('training.odahuflow.yaml')
```

In this file, we refer to file *training.odahuflow.yaml* that is located at airflow dag's folder

For example, if you use Google Cloud Composer then you can locate your yamls inside DAGs bucket and refer to them by relative path:

```
gsutil cp ~/.training.odahuflow.yaml gs://<your-composer-dags-bucket>/
```

DAG example

The example of the DAG that uses custom Odahu-flow operators is shown below. Four DAGs are described.

dag.py

```
from datetime import datetime
from airflow import DAG
from airflow.contrib.operators.gcs_to_gcs import GoogleCloudStorageToGoogleCloudStorageOperator
from airflow.models import Variable
from airflow.operators.bash_operator import BashOperator
from odahuflow.sdk.models import ModelTraining, ModelTrainingSpec, ModelIdentity, ResourceRequirements, ResourceList, \
    ModelPackaging, ModelPackagingSpec, Target, ModelDeployment, ModelDeploymentSpec, Connection, ConnectionSpec, \
    DataBindingDir

from odahuflow.airflow.connection import GcpConnectionToOdahuConnectionOperator
from odahuflow.airflow.deployment import DeploymentOperator, DeploymentSensor
from odahuflow.airflow.model import ModelPredictRequestOperator, ModelInfoRequestOperator
from odahuflow.airflow.packaging import PackagingOperator, PackagingSensor
from odahuflow.airflow.training import TrainingOperator, TrainingSensor

default_args = {
    'owner': 'airflow',
    'depends_on_past': False,
    'start_date': datetime(2019, 9, 3),
    'email_on_failure': False,
    'email_on_retry': False,
    'end_date': datetime(2099, 12, 31)
}

api_connection_id = "odahuflow_api"
model_connection_id = "odahuflow_model"

gcp_project = Variable.get("GCP_PROJECT")
wine_bucket = Variable.get("WINE_BUCKET")

wine_conn_id = "wine"
wine = Connection(
    id=wine_conn_id,
    spec=ConnectionSpec(
        type="gcs",
        uri=f'gs://{wine_bucket}/data/wine-quality.csv',
        region=gcp_project,
    )
)

training_id = "airlfow-wine"
training = ModelTraining(
    id=training_id,
    spec=ModelTrainingSpec(
        model=ModelIdentity(
            name="wine",
            version="1.0"
        ),
        toolchain="mlflow",
        entrypoint="main",
        work_dir="mlflow/sklearn/wine",
        hyper_parameters={
            "alpha": "1.0"
        },
        data=[
            DataBindingDir(
                conn_name='wine',
                local_path='mlflow/sklearn/wine/wine-quality.csv'
            ),
        ],
        resources=ResourceRequirements(
            requests=ResourceList(
                cpu="2024m",
                memory="2024Mi"
            ),
            limits=ResourceList(
                cpu="2024m",
                memory="2024Mi"
            )
        ),
        vcs_name="odahu-flow-examples"
```

```
    ),
)

packaging_id = "airlfow-wine"
packaging = ModelPackaging(
    id=packaging_id,
    spec=ModelPackagingSpec(
        targets=[Target(name="docker-push", connection_name="docker-ci")],
        integration_name="docker-rest"
```

```
    ),
)

deployment_id = "airlfow-wine"
deployment = ModelDeployment(
    id=deployment_id,
    spec=ModelDeploymentSpec(
        min_replicas=1,
    ),
)

model_example_request = {
    "columns": ["alcohol", "chlorides", "citric acid", "density", "fixed acidity", "free sulfur dioxide", "pH",
                "residual sugar", "sulphates", "total sulfur dioxide", "volatile acidity"],
    "data": [[12.8, 0.029, 0.48, 0.98, 6.2, 29, 3.33, 1.2, 0.39, 75, 0.66],
             [12.8, 0.029, 0.48, 0.98, 6.2, 29, 3.33, 1.2, 0.39, 75, 0.66]]
}

dag = DAG(
    'wine_model',
    default_args=default_args,
    schedule_interval=None
)

with dag:
    data_extraction = GoogleCloudStorageToGoogleCloudStorageOperator(
        task_id='data_extraction',
        google_cloud_storage_conn_id='wine_input',
        source_bucket=wine_bucket,
        destination_bucket=wine_bucket,
        source_object='input/*.csv',
        destination_object='data/',
        project_id=gcp_project,
        default_args=default_args
    )
    data_transformation = BashOperator(
        task_id='data_transformation',
        bash_command='echo "imagine that we transform a data"',
        default_args=default_args
    )
    odahuflow_conn = GcpConnectionToOdahuConnectionOperator(
        task_id='odahuflow_connection_creation',
        google_cloud_storage_conn_id='wine_input',
        api_connection_id=api_connection_id,
        conn_template=wine,
        default_args=default_args
    )

    train = TrainingOperator(
        task_id="training",
        api_connection_id=api_connection_id,
        training=training,
        default_args=default_args
    )

    wait_for_train = TrainingSensor(
        task_id='wait_for_training',
        training_id=training_id,
        api_connection_id=api_connection_id,
        default_args=default_args
    )

    pack = PackagingOperator(
        task_id="packaging",
        api_connection_id=api_connection_id,
        packaging=packaging,
        trained_task_id="wait_for_training",
        default_args=default_args
    )

    wait_for_pack = PackagingSensor(
        task_id='wait_for_packaging',
        packaging_id=packaging_id,
```

```
        api_connection_id=api_connection_id,
        default_args=default_args
    )

    dep = DeploymentOperator(
        task_id="deployment",
        api_connection_id=api_connection_id,
        deployment=deployment,
        packaging_task_id="wait_for_packaging",
```

```
163         default_args=default_args
164     )
165
166     wait_for_dep = DeploymentSensor(
167         task_id='wait_for_deployment',
168         deployment_id=deployment_id,
169         api_connection_id=api_connection_id,
170         default_args=default_args
171     )
172
173     model_predict_request = ModelPredictRequestOperator(
174         task_id="model_predict_request",
175         model_deployment_name=deployment_id,
176         api_connection_id=api_connection_id,
177         model_connection_id=model_connection_id,
178         request_body=model_example_request,
179         default_args=default_args
180     )
181
182     model_info_request = ModelInfoRequestOperator(
183         task_id='model_info_request',
184         model_deployment_name=deployment_id,
185         api_connection_id=api_connection_id,
186         model_connection_id=model_connection_id,
187         default_args=default_args
188     )
189
190     data_extraction >> data_transformation >> odahuflow_conn >> train
191     train >> wait_for_train >> pack >> wait_for_pack >> dep >> wait_for_dep
192     wait_for_dep >> model_info_request
193     wait_for_dep >> model_predict_request
```

In this file, we create four dags:

- DAG on line 190 extract and transform data, create Odahu-flow connection and run **Train**
- DAG on line 191 sequentially run phases **Train**, **Package**, **Deploy**
- DAG on line 192 wait for model deploy and then extract schema of model predict API
- DAG on line 193 wait for model deploy and then invoke model prediction API

JupyterLab extension

Odahu-flow provides the JupyterLab extension that allows you to interact with an Odahu cluster from JupyterLab web-based IDEs.

Installation

Prerequisites:

- Python 3.6 or higher
- Jupyterlab GUI
- Preferable to use Google Chrome or Mozilla Firefox browsers

To install the extension, perform the following steps:

```
pip install odahu-flow-jupyterlab-plugin
jupyter serverextension enable --sys-prefix --py odahuflow.jupyterlab
jupyter labextension install odahu-flow-jupyterlab-plugin
```

Another option is prebuilt Jupyterlab Docker Image with the extension.

Configuration

The extension can be configured though the environment variables.

Environment name	Default	Value example	Description
DEFAULT_API_ENDPOINT		https://odahu.company.com/	Default URL to the Odahu-flow API server
API_AUTH_ENABLED	true	true	Change the value to false if authorization is disabled on the Odahu-flow API server
ODAHUFLOWCTL_OAUTH_AUTH_URL		https://keycloak.company.org/auth/realms/master/protocol/openid-connect/auth	Keycloak authorization endpoint
JUPYTER_REDIRECT_URL		http://localhost:8888	JupyterLab external URL

ODAHUFLOWCTL_OAUTH_CLIENT_ID			Oauth client ID
ODAHUFLOWCTL_OAUTH_CLIENT_SECRET			Oauth2 client secret

To enable SSO, you should provide the following options:

- *ODAHUFLOWCTL_OAUTH_AUTH_URL*
- *JUPYTER_REDIRECT_URL*
- *ODAHUFLOWCTL_OAUTH_CLIENT_SECRET*
- *ODAHUFLOWCTL_OAUTH_CLIENT_ID*

Login

To authorize on an Odahu-flow API service in the Jupyterlab extension, you should perform the following steps:

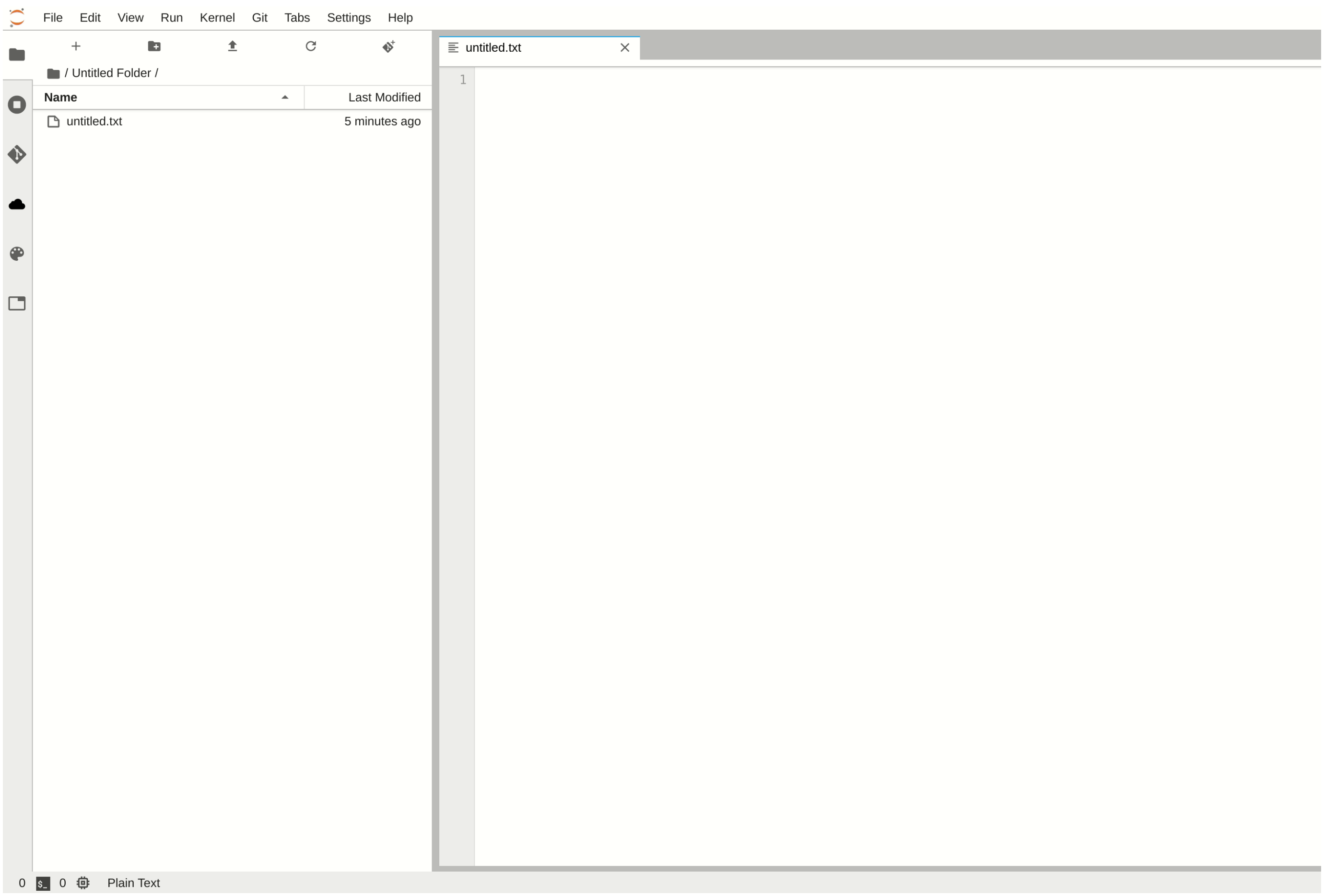
- Copy and paste the Odahu-flow API service URL.
- Open an API server URL in a browser to get the token. Copy and paste this token in the login form.

Usage

Below we consider all views of the JupyterLab extension.

Templates

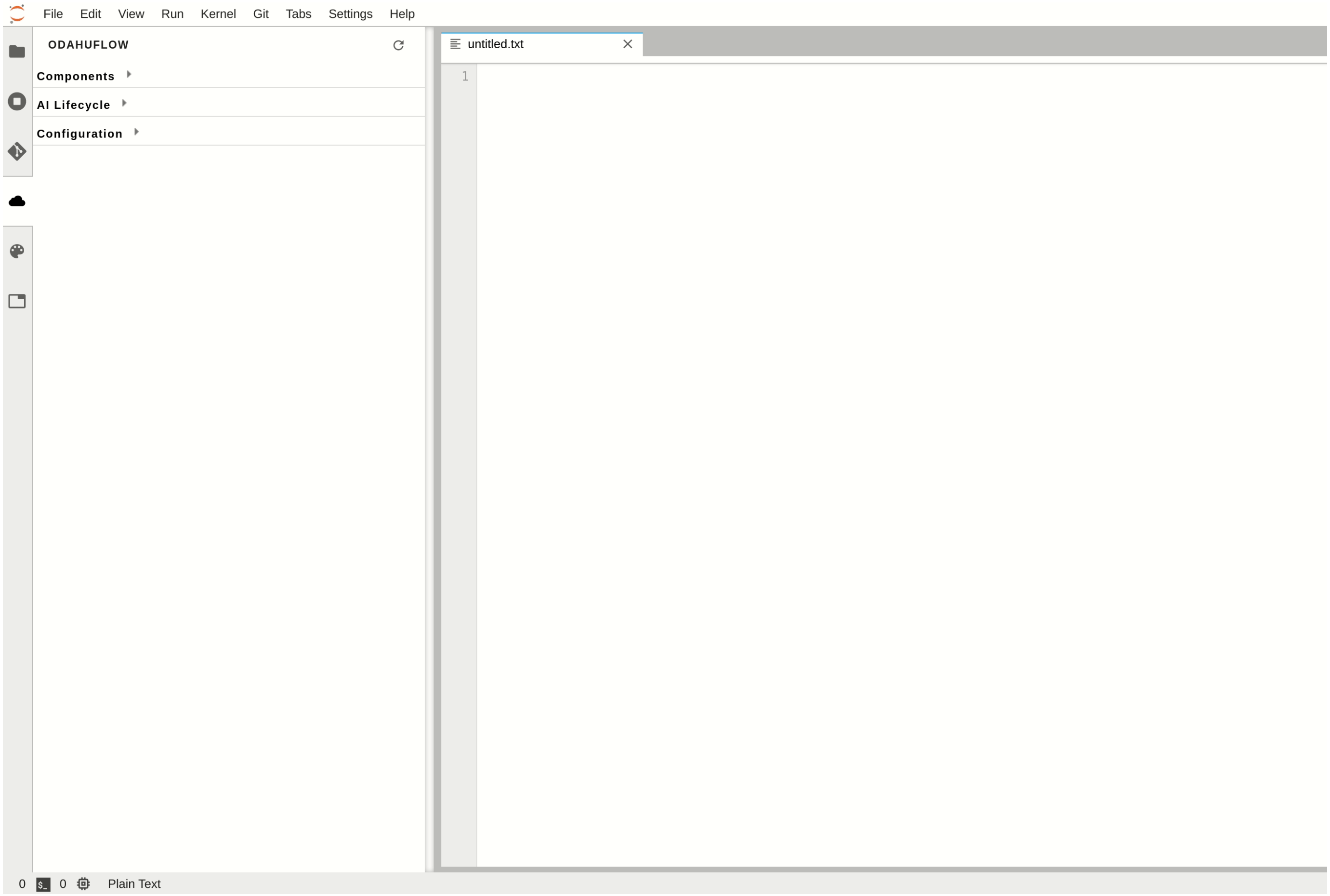
The extension provides predefined list of API file templates. You can create a file from a template.



Main view

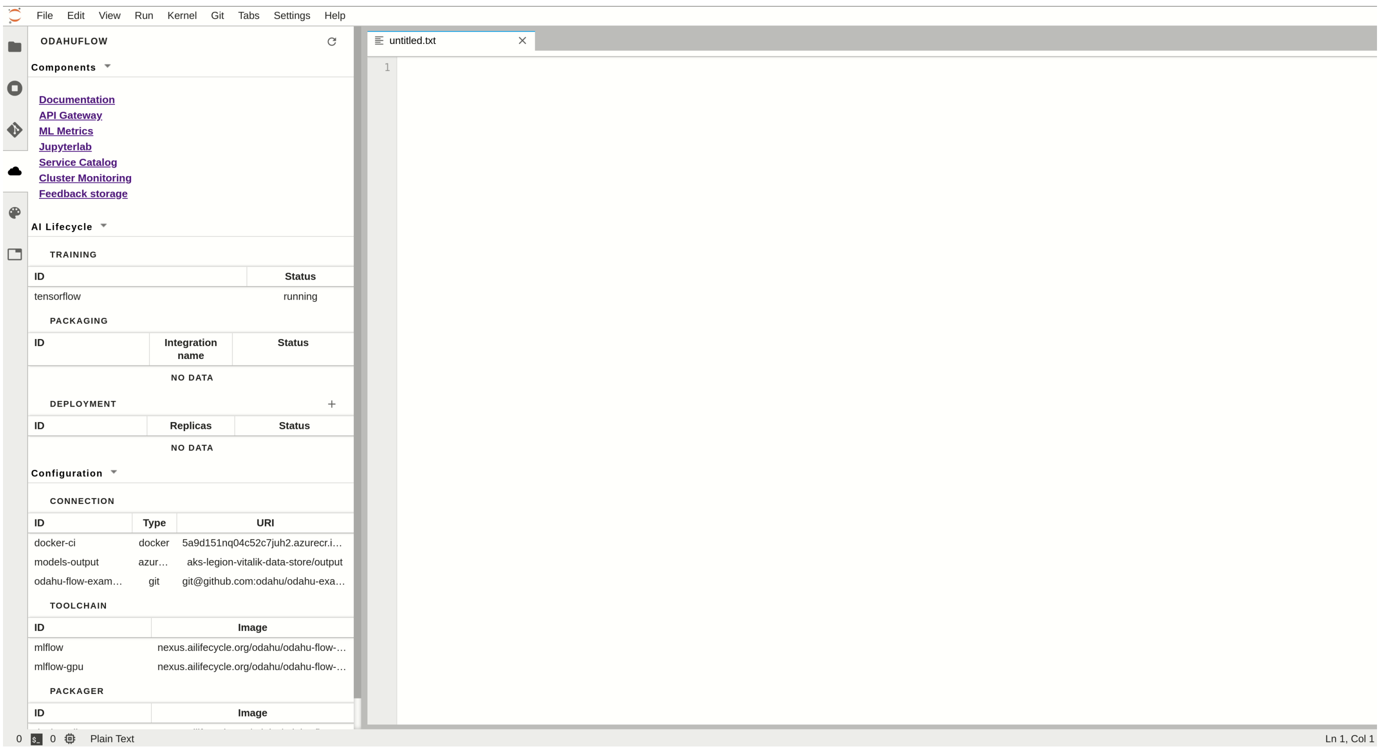
The main view contains all Odahu-flow entities. You can view or delete them.

Login



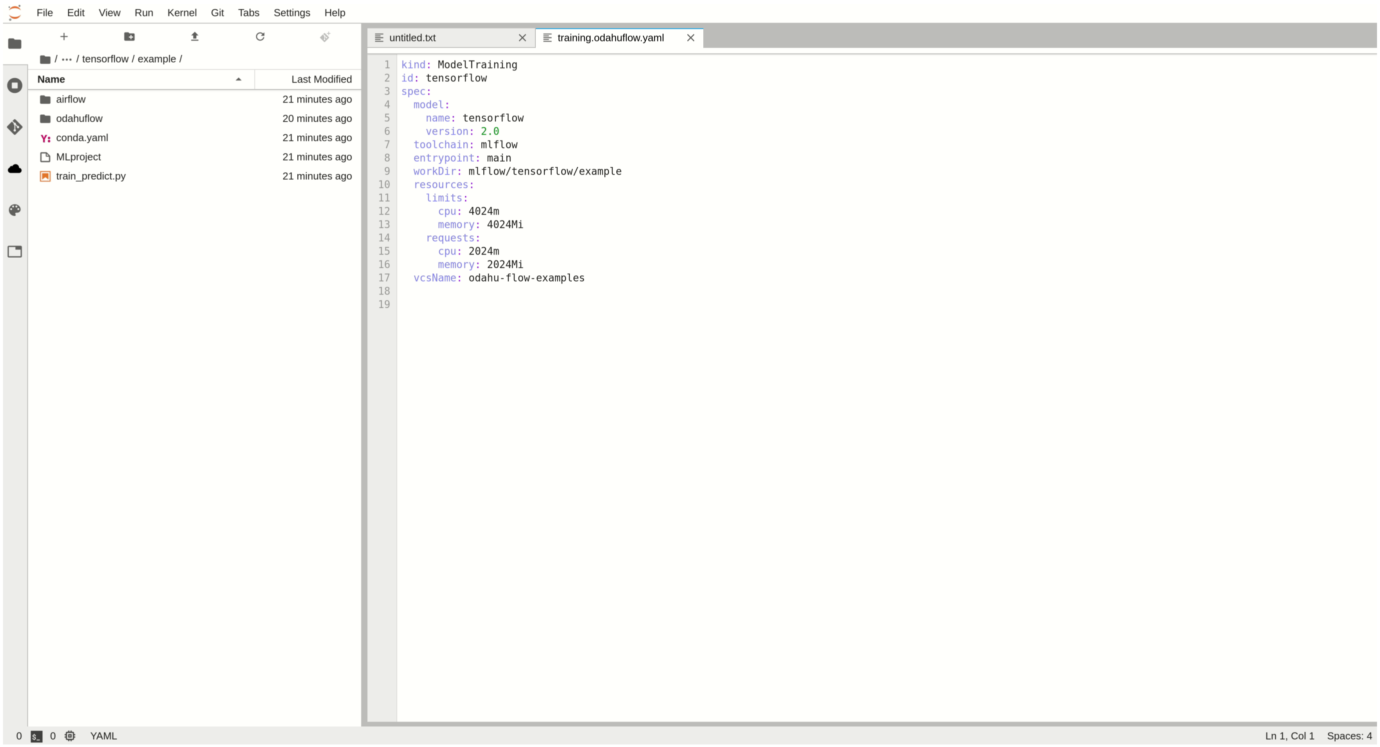
Log viewer

For troubleshooting, you can get access to the training, packaging or deployment logs. If the job is running then logs will be updated in runtime.



Submit resources

You can create any Odahu-flow entities from the extension. The button `Submit` only appears in the context menu when file ends with `.yaml` or `json`.



Odahuflowctl

Odahuflowctl (`odahuflowctl`) is a command-line interface for interacting with Odahu-flow API service.

Prerequisites:

- Python 3.6 or higher

Installation

Odahu-flow CLI is available in PyPi repository. You should execute the following command to install `odahuflowctl`:

```
pip install odahu-flow-cli
odahuflowctl --version
```

Help

To read odahuflowctl help, you should use the following command:

```
odahuflowctl --help
```

for a specific command, for example, get list of model deployments:

```
odahuflowctl deployment get --help
```

Login

There are two authentication types for Odahu CLI.

Specifying of a token explicitly

You should open an API server URL in a browser to get the login command. The command already contains your token. Copy and paste provided command into your shell.

Example of command:

```
odahuflowctl login --url <api-url> --token <your-token>
```

Sign in interactively

This method will use a web browser to sign in.

Run the login command:

```
odahuflowctl login --url <api-url>
```

Odahu CLI will open an IAM server in your default browser. Sign in with your account credentials.

Completion

`odahuflowctl` cli supports completion for following shells: bash, zsh, fish, PowerShell.

To activate it, evaluate the output of `odahuflowctl completion <YOUR_SHELL>`.<YOURSHELL> is the optional, it can be automatically identified.

Bash example:

```
source <(odahuflowctl completion bash)
```

PowerShell example:

```
odahuflowctl completion > $HOME\.odahuflow\odahu_completion.ps1;
. $HOME\.odahuflow\odahu_completion.ps1;
Remove-Item $HOME\.odahuflow\odahu_completion.ps1
```

To activate completion automatically in any new shell, you can save the completion code to a file and add it to your shell profile.

Bash example:

```
odahuflowctl completion bash > ${HOME}/.odahuflow/odahuflowctl_completion.sh
(echo ""; echo "source ${HOME}/.odahuflow/odahuflowctl_completion.sh"; echo "") >> ${HOME}/.bashrc
```

PowerShell example:

```
write "`n# odahuflowctl completion" (odahuflowctl completion) >> $PROFILE.CurrentUserAllHosts
```

Model Format

The **Odahu Model Artifact Format** (OMAF) describes a format to package, store, and transport ML models.

Models can be built in different languages and use different platform libraries. For example: {Python, Scala, R, …} using {scikit-learn, tensorflow, keras, …}.

An OMAF **Artifact** is stored as a file-system folder packed into a ZIP file using the Deflate ZIP compression algorithm.

The Artifact contains:

- `odahuflow.model.yaml` a YAML file in the root folder. This file contains meta-information about the type of binary model and other model related information (e.g. language, import endpoints, dependencies).
- Additional folders and files, depending upon meta-information declared in `odahuflow.model.yaml`.

odahuflow.model.yaml

File structure:

- `binaries` - Language and dependencies that should be used to load model binaries
- `binaries.type` - Required Odahu Model Environments. See section Odahu Model Environments.
- `binaries.dependencies` - Dependency management system, compatible with the selected Odahu Model Environment
- `binaries.<additional>` - Model Environment and dependency management system values, for example ‘a path to the requirements file’
- `model` - Location of the model artifact Model artifact format depends on Odahu Model Environment.
- `model.name` - name of the model, `[a-Z0-9-]+`
- `model.version` - version of model. Format is `<Apache Version>-<Additional suffix>`, where `Additional suffix` is a `[a-Z0-9-.]+` string.
- `model.workDir` - working directory to start model from.
- `model.entrypoint` - name of model artifact (e.g. Python module or Java JAR file).
- `odahuflowVersion` - OMAF version
- `toolchain` - toolchain used for training and preparing the Artifact
- `toolchain.name` - name of the toolchain
- `toolchain.version` - version of used toolchain.
- `toolchain.<additional>` - additional fields, related to used toolchain (e.g. used submodule of toolchain).

Examples:

Example with GPPI using conda for dependency management, mlflow toolchain.

```
binaries:
  type: python
  dependencies: conda
  conda_path: mlflow/model/mlflow_env.yml
model:
  name: wine-quality
  version: 1.0.0-12333122
  workDir: mlflow/model
  entrypoint: entrypoint
odahuflowVersion: '1.0'
toolchain:
  name: mlflow
  version: 1.0.0
```

Odahu Model Environments

Odahu supports these model environments:

- General Python Prediction Interface (GPPI). Can import a trained model as a python module and use a predefined function for prediction. Value for `binaries.type` should be `python`.
- General Java Prediction Interface (GJPI). Can import a trained model as a Java Library and use a predefined interfaces for prediction. Value for `binaries.type` should be `java`.

Odahu's General Python Prediction Interface (GPPI)

General Information

Description

This interface is an importable Python module with a declared interface (functions with arguments and return types). Toolchains that save models in this format must provide an `entrypoint` with this interface or they may provide a wrapper around their interface for this interface.

Required Environment variables

- MODEL_LOCATION – path to model's file, relative to working directory.

Interface declaration

Interface functions:

Connections

Odahu needs to know how to connect to a bucket, git repository, and so on. This kind of information is handled by Connection API.

General connection structure

All types of connections have the same general structure. But different connections require a different set of fields. You can find the examples of specific type of connection in the id of the Connection types section. Below you can find the description of all fields:

Connection API

```yaml
kind: Connection
# Unique value among all connections
# Id must:
#  * contain at most 63 characters
#  * contain only lowercase alphanumeric characters or '-'
#  * start with an alphanumeric character
#  * end with an alphanumeric character
id: "id-12345"
spec:
    # Optionally description of a connection
    description: "Some description"
    # Optionally link to the web resource. For example, git repo or a gcp bucket
    webUILink: https://test.org/123
    # URI. It is a required value.
    uri: s3://some-bucket/path/file
    # Type of a connection. Available values: s3, gcs, azureblob, git, docker, ecr.
    type: s3
    # Username
    username: admin
    # Password, must be base64-encoded
    password: admin
    # Service account role
    role: some-role
    # AWS region or GCP project
    region: some region
    # VCS reference
    reference: develop
    # Key ID, must be base64-encoded
    keyID: "1234567890"
    # SSH or service account secret, must be base64-encoded
    keySecret: b2RhaHUK
    # SSH public key, must be base64-encoded
    publicKey: b2RhaHUK
    # Defines if connection is vital. Vital connections cannot be deleted
    vital: false
```

Connection management

Connections can be managed using the following ways.

Swagger UI

Swagger UI is available at http://api-service/swagger/index.html URL.

Odahu-flow CLI

Odahuflowctl supports the connection API. You must be login if you want to get access to the API.

- Getting all connections in json format:

```
odahuflowctl conn get --format json
```

- Getting the reference of the connection:

```
odahuflowctl conn get --id odahu-flow-examples -o 'jsonpath=[*].spec.reference'
```

- Creating of a connection from *conn.yaml* file:

```
odahuflowctl conn create -f conn.yaml
```

- All connection commands and documentation:

```
odahuflowctl conn --help
```

JupyterLab

Odahu-flow provides the JupyterLab extension for interacting with Connection API.

Connection types

For now, Odahu-flow supports the following connections types:

- S3
- Google Cloud Storage
- Azure Blob storage
- GIT
- Docker
- Amazon Elastic Container Registry

S3

An S3 connection allows interactions with s3 API. This type of connection is used as storage of:

- model trained artifacts.
- input data for ML models.

> ***Note***
>
> You can use any S3 compatible API, for example minio or Ceph.

Before usage, make sure that:

- You have created an AWS S3 bucket. Examples of Creating a Bucket.
- You have created an IAM user that has access to the AWS S3 bucket. Creating an IAM User in Your AWS Account.
- You have created the IAM keys for the user. Managing Access Keys for IAM Users.

> ***Note***
>
> At that moment, Odahu-flow only supports authorization though IAM User. We will support AWS service role and authorization using temporary credentials in the near future.

The following fields of connection API are required:

- `spec.type` - It must be equal **s3**.
- `spec.keyID` - **base64-encoded** access key ID (for example, `AKIAIOSFODNN7EXAMPLE`).
- `spec.keySecret` - **base64-encoded** secret access key (for example, `wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY`).
- `spec.uri` - S3 compatible URI, for example s3://<bucket-name>/dir1/dir2/
- `spec.region` - AWS Region, where a bucket was created.

Example of Connection S3:

```yaml
kind: Connection
id: "training-data"
spec:
    type: s3
    uri: s3://raw-data/model/input
    # keyID before base64-encoding: AKIAIOSFODNN7EXAMPLE
    keyID: "QUtJQUlPU0ZPRE5ON0VYQU1QTEU="
    # keySecret before base64 encoding: wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY
    keySecret: "d0phbHJYVXRuRkVNSS9LN01ERU5HL2JQeFJmaUNZRVhBTVBMRUtFWQ=="
    description: "Training data for a model"
    region: eu-central-1
```

Google Cloud Storage

[Google Cloud Storage](#) allows storing and accessing data on Google Cloud Platform infrastructure. This type of connection is used as storage of:

- model trained artifacts.
- input data for ML models.

Before usage, make sure that:

- You have created an GCS bucket. Creating storage buckets.
- You have created an service account. Creating and managing service accounts.
- You have assigned `roles/storage.objectAdmin` role on the service account for the GCS bucket. Using Cloud IAM permissions.
- You have created the IAM keys for the service account. Creating and managing service account keys.

> ***Note***
>
> Workload Identity is the recommended way to access Google Cloud services from within GKE due to its improved security properties and manageability. We will support the Workload Identity in the near future.

The following fields of connection API are required:

- `spec.type` - It must be equal **gcs**.
- `spec.keySecret` - **base64-encoded** service account key in json format.
- `spec.uri` - GCS compatible URI, for example gcs://<bucket-name>/dir1/dir2/
- `spec.region` - GCP Region, where a bucket was created.

Example of Connection GCS:

```yaml
kind: Connection
id: "training-data"
spec:
    type: gcs
    uri: gsc://raw-data/model/input
    keySecret: [illegible]
    description: "Training data for a model"
    region: us-central2
```

Original service account JSON key, that is used in the example above, before base64-encoding:

```json
{
    "type": "service_account",
    "project_id": "project_id",
    "private_key_id": "private_key_id",
    "private_key": "-----BEGIN PRIVATE KEY-----\nprivate_key\n-----END PRIVATE KEY-----\n",
    "client_email": "test@project_id.iam.gserviceaccount.com",
    "client_id": "123455678",
    "auth_uri": "https://accounts.google.com/o/oauth2/auth",
    "token_uri": "https://oauth2.googleapis.com/token",
    "auth_provider_x509_cert_url": "https://www.googleapis.com/oauth2/v1/certs",
    "client_x509_cert_url": "https://www.googleapis.com/robot/v1/metadata/x509/test@project_id.iam.gserviceaccount.com"
}
```

Azure Blob storage

Odahu-flow uses the Blob storage in Azure to store:

- model trained artifacts.
- input data for ML models.

Before usage, make sure that:

- You have created a storage account . Create a storage account.
- You have created a storage container in the storage account . Create a container.
- You have created a SAS token. Create an account SAS.

The following fields of connection API are required:

- `spec.type` - It must be equal **azureblob**.
- `spec.keySecret` - Odahu-flow uses the shared access signatures to authorize in Azure. The key has the following format: “<primary_blob_endpoint>/<sas_token>” and must be **base64-encoded**.
- `spec.uri` - Azure storage compatible URI, for example <bucket-name>/dir1/dir2/

Example of Connection Blob Storage:

```yaml
kind: Connection
id: "training-data"
spec:
    type: azureblob
    uri: raw-data/model/input
    # keySecret before base64-encoding: https://myaccount.blob.core.windows.net/?restype=service&comp=properties&sv=2019-02-02&ss=bf&srt=s&st=2019-08-01T22%3A18%3A26Z&se=2019-08-10T02%3A23%3A26Z&sr=b&sp=rw&sip=168.1.5.60-168.1.5.70&spr=https&sig=F%6GRVAZ5Cdj2Pw4tgU7IlSTkWgn7bUkkAg8P6HESXwmf%4B
    keySecret: aHR0cHM6Ly9teWFjY291bnQuYmxvYi5jb3JlLndpbmRvd3MubmV0Lz9yZXN0eXBlPXNlcnZpY2UmY29tcD1wcm9wZXJ0aWVzJnN2PTIwMTktMDItMDImc3M9YmYmc3J0PXMmc3Q9MjAxOS0wOC0wMVQyMiUzQTE4JTNBMjZaJnNlPTIwMTktMDgtMTBUMDIlM0EyMyUzQTI2WiZzcj1iJnNwPXJ3JnNpcD0xNjguMS41LjYwLTE2OC4xLjUuNzAmc3ByPWh0dHBzJnNpZz1GJTZHUlZBWjVDZGoyUHc0dGdVN0lsU1RrV2duN2JVa2tBZzhQNkhFU1h3bWYlNEI=
    description: "Training data for a model"
```

GIT

Odahu-flow uses the GIT type connection to download a ML source code from a git repository.

The following fields of connection API are required:

- `spec.type` - It must be equal **git**.
- `spec.keySecret` - a base64 encoded SSH private key.
- `spec.uri` - GIT SSH URL, for example git@github.com:odahu/odahu-examples.git

`spec.reference` must be provided either in a connection OR in a model training object (General training structure).

Example of command to encode ssh key:

```
cat ~/.ssh/id_rsa | base64 -w0
```

> ***Note***
>
> Odahu-flow only supports authorization through SSH.

> ***Warning***
>
> We recommend using the read-only deploy keys: Github docs or Gitlab docs.

Example of GIT Connection:

```yaml
kind: Connection
id: "ml-repository"
spec:
    type: git
    uri: git@github.com:odahu/odahu-examples.git
    keySecret: ClNVUEVSIFNFQ1JFVAoK
    reference: master
    description: "Git repository with the Odahu-Flow examples"
    webUILink: https://github.com/odahu/odahu-examples
```

Docker

This type of connection is used for pulling and pushing of the Odahu packager result Docker images to a Docker registry. We have been testing the following Docker repositories:

- Docker Hub
- Nexus
- Google Container Registry
- Azure Container Registry

> ***Warning***
>
> Every docker registry has its authorization specificity. But you must be able to authorize by a username and password. Read the documentation.

Before usage, make sure that:

- You have a username and password.

The following fields of connection API are required:

- `spec.type` - It must be equal **docker**.
- `spec.username` - docker registry username.
- `spec.password` - **base64-encoded** docker registry password.
- `spec.uri` - docker registry host.

> ***Warning***
>
> Connection URI must not contain a URI schema.

Example of GCR Docker connection

```yaml
kind: Connection
id: "docker-registry"
spec:
    type: docker
    uri: gcr.io/project/odahuflow
    username: "_json"
    password: [illegible]
```

Original service account JSON key, that is used in the example above, before base64-encoding:

```json
{
    "type": "service_account",
    "project_id": "project_id",
    "private_key_id": "private_key_id",
    "private_key": "-----BEGIN PRIVATE KEY-----\nprivate_key\n-----END PRIVATE KEY-----\n",
    "client_email": "test@project_id.iam.gserviceaccount.com",
    "client_id": "123455678",
    "auth_uri": "https://accounts.google.com/o/oauth2/auth",
    "token_uri": "https://oauth2.googleapis.com/token",
    "auth_provider_x509_cert_url": "https://www.googleapis.com/oauth2/v1/certs",
    "client_x509_cert_url": "https://www.googleapis.com/robot/v1/metadata/x509/test@project_id.iam.gserviceaccount.com"
}
```

Example of Docker Hub

```yaml
kind: Connection
id: "docker-registry"
spec:
    type: docker
    uri: docker.io/odahu/
    username: username
    # password before encoding: mypassword
    password: bXlwYXNzd29yZA===
```

Amazon Elastic Container Registry

Amazon Elastic Container Registry is a managed AWS Docker registry. This type of connection is used for pulling and pushing of the Odahu packager result Docker images.

> **Note**
>
> The Amazon Docker registry does not support a long-lived credential and requires explicitly to create a repository for every image. These are the reasons why we create a dedicated type of connection for the ECR.

Before usage, make sure that:

- You have created an ECR repository. Creating an ECR Repository.
- You have created an IAM user that has access to the ECR repository. Creating an IAM User in Your AWS Account.
- You have created the IAM keys for the user. Managing Access Keys for IAM Users.

The following fields of connection API are required:

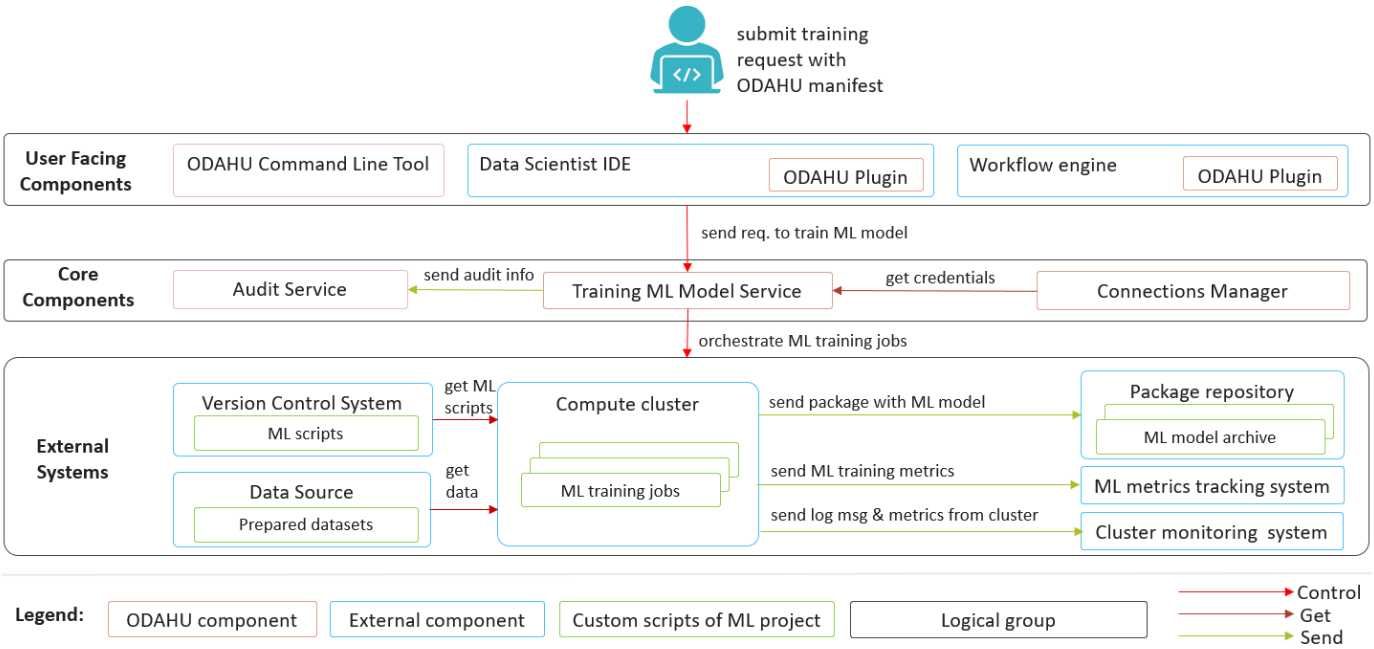
- `spec.type` - It must be equal **ecr**.

- `spec.keyID` - **base64-encoded** access key ID (for example, `AKIAIOSFODNN7EXAMPLE`).
- `spec.keySecret` - **base64-encoded** secret access key (for example, `wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY`).
- `spec.uri` - The url must have the following format, *aws_account_id*.dkr.ecr.`region`.amazonaws.com/*some-prefix*.
- `spec.region` - AWS Region, where a docker registry was created.

Example of Connection ECR:

```yaml
kind: Connection
id: "docker-registry"
spec:
    type: ecr
    uri: 5555555555.dkr.ecr.eu-central-1.amazonaws.com/odahuflow
    # keyID before base64-encoding: "AKIAIOSFODNN7EXAMPLE"
    keyID: QUtJQUlPU0ZPRE5ON0VYQU1QTEU=
    # keySecret before base64-encoding: "wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY"
    keySecret: d0phbHJYVXRuRkVNSS9LN01ERU5HL2JQeFJmaUNZRVhBTVBMRUtFWQ==
    description: "Packager registry"
    region: eu-central-1
```

Model Trainings



ODAHU model training component helps to automate ML model training jobs execution in K8S. The primary goal of model training component is to create a **Trained Model Binary** for a **Packager**. The API is pluggable and can be extended for different ML frameworks.

You can find the list of out-of-the-box trainers below:

- MLFlow toolchain
- MLFlow Project toolchain

General training structure

Training API

```yaml
kind: ModelTraining
# Some unique value among all trainings. if not, the training with the same name will be overwritten.
# Id must:
#  * contain at most 63 characters
#  * contain only lowercase alphanumeric characters or ‘-’
#  * start with an alphanumeric character
#  * end with an alphanumeric character
id: wine-12345
spec:
  model:
    # Human-readable model name
    name: wine
    # Human-readable model version
    version: 3.0
    # Optionally, you can specify template for output artifact
    # The default value is {{ .Name }}-{{ .Version }}-{{ .RandomUUID }}.zip
    # where:
    #   Name - spec.model.name
    #   Version - spec.model.version
    #   RandomUUID - a random UUID v4, for example be17d12d-df43-4588-99e7-56a0db3cad77
    artifactNameTemplate: {{ .Name }}-{{ .Version }}-{{ .RandomUUID }}.zip
  # The toolchain parameter is a point of extension for different ML frameworks.
```

```yaml
  # For now, we only support the Mlfow toolchain
  toolchain: mlflow
  # Mlflow MLProject file contains the list of entrypoints. You must choose one of those.
  entrypoint: main
  # Working directory inside a training (docker) container, which GIT repository copied in.
  # Optional. The default directory is "./".
  workDir: work/dir
  # The training data for a ML script. You can find full description there: https://docs.odahu.epam.com/ref_trainings.html#training-data
  data:
      # You can specify a connection name
    - connection: wine
      # Path to a file or a dir where data will copy from a bucket; relative to your Git repository root derictory.
      localPath: mlflow/wine-quality/
      # Path to the dir or file in a bucket
      # Optional. If it is missing then the path from connection will be used.
      remotePath: training-data/
  # You can specify the map of hyperparameters
  hyperParameters:
    key: value
    var2: test
  # Compute resources for the training job.
  resources:
    limits:
      cpu: 1
      memory: 1Gi
    requests:
      cpu: 1
      memory: 1Gi
  # Custom environment variables that should be set before entrypoint invocation.
  envs:
      # The name of variable
    - name: TEST_ENV_KEY
      # The value of variable
      value: TEST_ENV_VALUE
  # A Docker image where the training will be launched.
  # By default, the image from a toolchain is used.
  image: python:3.8
  # A section defining training source code
  algorithmSource:
    # Use vcs if source code located in a repository and objectStorage if in a storage. Should not use both
    vcs:
      # A connection which describes credentials to a GIT repository or to a bucket if using objectStorage
      connection: <git-connection>
      # Git reference (branch or tag)
      # This must be specified here OR in Git connection itself
      # In case of using objectStorage, specify path: <remote path> instead of reference
      reference: master
  # You can set a connection that points where the Trained Model Binary will be stored.
  # Optional. Default value is taken from the ODAHU cluster configuration.
  outputConnection: custom-connection
  # Node selector that exactly matches a node pool from ODAHU config
  # This is optional; when omitted, ODAHU uses any of available training node pools
  # Read more about node selector: https://kubernetes.io/docs/concepts/scheduling-eviction/assign-pod-node/
  nodeSelector:
    label: value
status:
  # One of the following states: scheduling, running, succeeded, failed, unknown
  state: running
  # List of training results
  artifacts:
      # Mlflow run ID
    - runId: 12345678
      # Trained artifact name
      artifactName: wine-10.zip
      # VCS commit ID
      commitID: d3d6e8ed776ed37fd2efd7a1b8d5fabdd7e3eea5
```

Training data

Odahu-flow allows downloading data from various sources to the local file system of a training job. Data source supports the following types of Odahu-flow connections:

- S3
- Google Cloud Storage
- Azure Blob storage

Let's consider the following example of downloading training data from Google Cloud Storage.

Prerequisites:

- The training data set is located in the *wine-training-data* bucket by *wine/11-11-2011/* directory.
- The ML script expects that the data will be located in the training (docker) container by *data/* directory relative to the root git directory.

First of all, we should create an Odahu-flow GCS connection.

Training Data GCS:

```
kind: ModelTraining
id: "wine-training-data-conn"
spec:
    type: gcs
    uri: gsc://wine-training-data/
    keySecret: '{"type": "service_account", "project_id": "project_id", "private_key_id": "private_key_id", "private_key": "-----BEGIN PRIVATE KEY-----\nprivate_key\n-----END PRIVATE KEY-----\n", "client_email": "test@project_id.iam.gserviceaccount.com", "client_id": "123455678", "auth_uri": "https://accounts.google.com/o/oauth2/auth", "token_uri": "https://oauth2.googleapis.com/token", "auth_provider_x509_cert_url": "https://www.googleapis.com/oauth2/v1/certs", "client_x509_cert_url": "https://www.googleapis.com/robot/v1/metadata/x509/test@project_id.iam.gserviceaccount.com"}'
    description: "Training data for a model"
    region: us-central2
```

Finally, we provide a data section of Model Training.

Example of Connection GCS:

```
spec:
  data:
    - connection: wine-training-data-conn
      localPath: data/
      remotePath: wine/11-11-2011/
```

GPU

Odahu-flow supports model training on GPU nodes.

You can find more about GPU deployment configuration in the installation guide.

In order to provision a training container in the GPU node pool, you must specify the GPU resource in the model training manifest.

Training on GPU

```
kind: ModelTraining
id: gpu-model
spec:
  resources:
    limits:
      cpu: 1
      memory: 1Gi
      gpu: 1
    requests:
      cpu: 1
      memory: 1Gi
```

NVIDIA libraries will be mount by ODAHU to the training container. But if you want to use a CUDA library, you should install it manually.

For example, you can add the following dependencies to a conda file: cudatoolkit-dev and cudatoolkit.

Model Dependencies Cache

ODAHU Flow downloads your dependencies on every model training launch. You can experience the following troubles with this approach:

- downloading and installation of some dependencies can take a long time
- network errors during downloading dependencies

To overcome these and other problems, ODAHU Flow provides a way to specify a prebuilt training Docker image with your dependencies.

> **Note**
>
> If you have different versions of a library in your model conda file and cache container, then the model dependency has a priority. It will be downloaded during model training.

First of all, you have to describe the Dockerfile:

- Inherit from a release version of odahu-flow-mlflow-toolchain
- Optionally, add install dependencies
- Add a model conda file
- Update the `odahu_model` conda environment.

Example of Dockerfile:

```
FROM odahu/odahu-flow-mlflow-toolchain:1.1.0-rc11

# Optionally
# apt-get install -y wget

ADD conda.yaml ./
RUN conda env update -n ${ODAHU_CONDA_ENV_NAME} -f conda.yaml
```

Build the docker image:

```
docker build -t training-model-cache:1.0.0 .
```

Push the docker image to a registry:

```
docker push training-model-cache:1.0.0
```

Specify the image in a model training:

Training example

```
kind: ModelTraining
id: model-12345
spec:
  image: training-model-cache:1.0.0
  ...
```

Trainings management

Trainings can be managed using the following ways.

Swagger UI

ModelTraining and ToolchainIntegration are available on the Swagger UI at http://api-service/swagger/index.html URL.

Odahu-flow CLI

Odahuflowctl supports the Training API. You must be logged in if you want to get access to the API.

Getting all trainings in json format:

```
odahuflowctl train get --format json
```

Getting the model name of the trainings:

```
odahuflowctl train get --id tensorflow-cli -o 'jsonpath=[*].spec.model.name'
```

- Creating a training from *train.yaml* file:

```
odahuflowctl train create -f train.yaml
```

- Reruning a training from *train.yaml* file:

```
odahuflowctl train edit -f train.yaml
```

- All training commands and documentation:

```
odahuflowctl train --help
```

We also have local training:

```
odahuflowctl local train --help
```

and can run trainings locally:

```
odahuflowctl local train run --id [Model training ID] -d [Path to Odahu manifest files]
```

more information you can find at **Local Quickstart**

JupyterLab

Odahu-flow provides the JupyterLab extension for interacting with Training API.

MLFlow toolchain

MLflow is library-agnostic. You can use it with any machine learning library, and in any programming language, since all functions are accessible through a REST API and CLI.

Installation

The most straightforward way to install the MLFlow trainer on an Odahu Cluster is to deploy the *odahu-flow-mlflow* helm chart. The helm chart registers the trainer in the API Odahu and deploys an MLflow Tracking Server. By default, the deployed MLflow Tracking Server is available at *https://cluster-url/mlflow* address.

```
# Add the odahu-flow helm repository
helm repo add odahu-flow 'https://raw.githubusercontent.com/odahu/odahu-helm/master/'
helm repo update
# Fill in the values for the chart or leave the default values
helm inspect values odahu-flow/odahu-flow-mlflow --version 1.0.0 > values.yaml
vim values.yaml
# Deploy the helm chart
helm install odahu-flow/odahu-flow-mlflow --name odahu-flow-mlflow --namespace odahu-flow --debug -f values.yaml --atomic --wait --timeout 120
```

> **Warning**
>
> Odahu-flow must be deployed before the mlflow trainer.

MLProject file

Let's look at how the MLProject file is related to Model Training API.

```
name: My Project

entry_points:
  main:
    parameters:
      data_file: path
      regularization: {type: float, default: 0.1}
    command: "python train.py -r {regularization} {data_file}"
  test:
    parameters:
      data_file: path
    command: "python validate.py {data_file}"
```

Model Training API can contain only one entry point. You have to add all hyperparameters, which do not have a default value, to a Model Training. Next, you can find the Model Trainings for the MLProject file.

```yaml
spec:
  entrypoint: main
  hyperParameters:
    data_file: test/123.txt
    regularization: 0.2
```

```yaml
spec:
  entrypoint: main
  hyperParameters:
    data_file: test/456.txt
```

```yaml
spec:
  entrypoint: test
  hyperParameters:
    data_file: test/123.txt
```

MLFlow protocol

Odahu-flow requires that a model is logged through mlflow API.

Example of sklearn model logging:

```python
mlflow.sklearn.log_model(lr, "model")
```

Optionally, you can provide input and output samples for Odahu-flow. It allows determining input and output types for Odahu-flow packagers. These names must be *head_input.pkl* and *head_output.pkl*, respectively.

Example of input and output samples logging:

```python
train_x.head().to_pickle('head_input.pkl')
mlflow.log_artifact('head_input.pkl', 'model')
train_y.head().to_pickle('head_output.pkl')
mlflow.log_artifact('head_output.pkl', 'model')
```

MLFlow Project toolchain

MLFlow Project toolchain is a lightweight version of **MLFlow toolchain**.

The main difference is that MLFlow Project toolchain does not require user to store models using MLFlow Tracking API and therefore does not require models stored in MLFlow format as a resulted artifact.

Instead, MLFlow Project toolchain relies only on MLFlow Project functionality to run training script and manage dependencies. User can store result artifacts in any format as they wish.

Installation

Installation of MLFlow Project toolchain is identical to MLFlow installation

MLProject file

MLFlow Project toolchain runs training script using MLProject specification. Please refer to previous section or official MLFlow documentation to learn more about MLProject file.

Storing training artifacts

You can store any artifacts during script execution in a special directory. To get a path to output directory read value of `$ODAHUFLOW_OUTPUT_DIR` environment variable.

Example

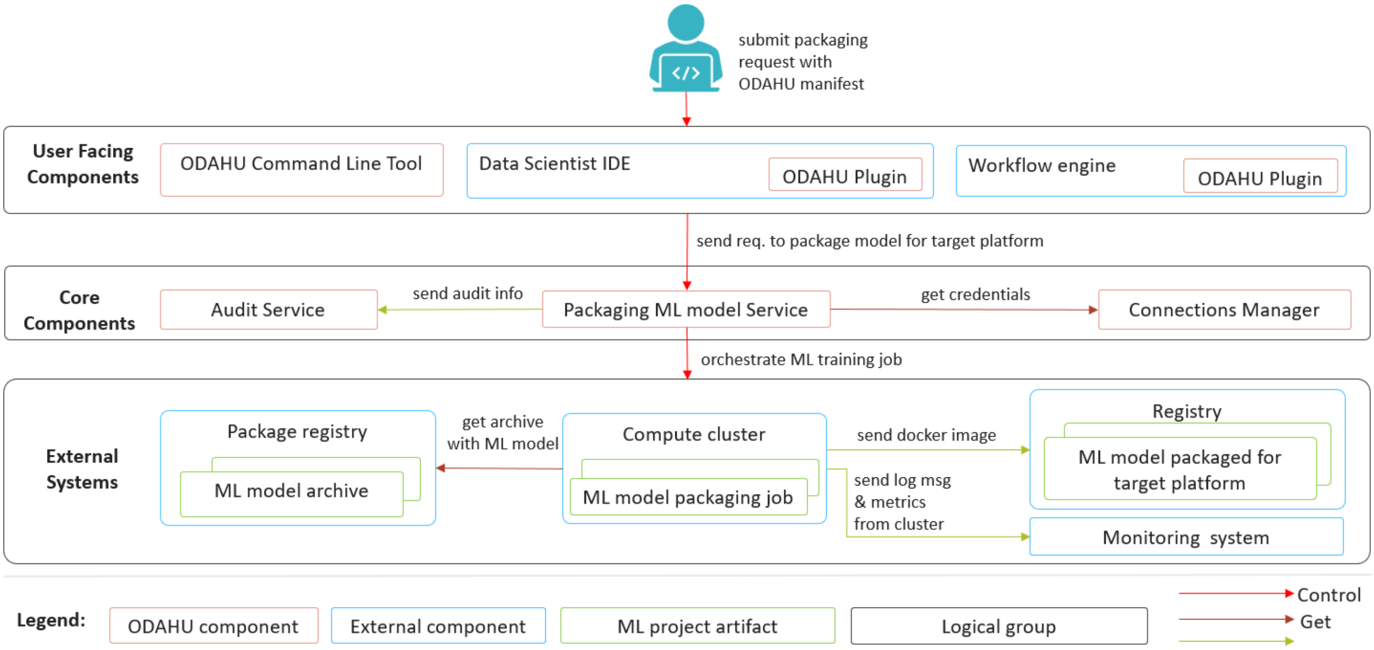
```
output_dir = os.environ.get("ODAHUFLOW_OUTPUT_DIR")

train_x.head().to_pickle(os.path.join(output_dir, 'head_input.pkl'))
```

Additionally, if `$STATIC_ARTIFACTS_DIR` variable is specified with a path to directory, all the contents is copied to final artifact. Path must be relative to working directory.

You can use this feature if you have some file(s) that are required by further steps and can be defined statically before script execution. For example, some python wrapper scripts to deploy a model into a specific ML Server in the future.

Model Packagers



ODAHU packaging component helps to wrap a **Trained Model Binary** artifact into a inference service, batch job or command line tool. You can find the list of out-of-the-box packagers below:

- Docker REST
- Docker CLI

Installation

A packager installation is the creation of a new PackagingIntegration entity in the **API service**. The most straightforward way is to deploy the *odahu-flow-packagers* helm chart.

```
# Add the odahu-flow helm repository
helm repo add odahu-flow 'https://raw.githubusercontent.com/odahu/odahu-helm/master/'
helm repo update
# Fill in the values for the chart or leave the default values
helm inspect values odahu-flow/odahu-flow-packagers --version 1.0.0-rc35 > values.yaml
vim values.yaml
# Deploy the helm chart
helm install odahu-flow/odahu-flow-packagers --name odahu-flow-packagers --namespace odahu-flow --debug -f values.yaml --atomic --wait --timeout 120
```

> ***Warning***
>
> Odahu-flow must be deployed before the packagers installation.

General packager structure

All packagers have the same structure. But different packagers provide a different set of arguments and targets. You can find the description of all fields below:

Packager API

```yaml
kind: ModelPackaging
# Unique value among all packagers
# Id must:
#  * contain at most 63 characters
#  * contain only lowercase alphanumeric characters or ‘-’
#  * start with an alphanumeric character
#  * end with an alphanumeric character
id: "id-12345"
spec:
  # Type of a packager. Available values: docker-rest, docker-cli.
  integrationName: docker-rest
  # Training output artifact name
  artifactName: wine-model-123456789.zip
  # Compute resources
  resources:
    limits:
      cpu: 1
      memory: 1Gi
    requests:
      cpu: 1
      memory: 1Gi
  # List of arguments. Depends on a Model Packaging integration.
  # You can find specific values in the sections below.
  # This parameter is used for customizing a packaging process.
  arguments: {}
  # List of targets. Depends on a Model Packaging integration.
  # You can find specific values in the sections below.
  # A packager can interact with a Docker registry, PyPi repository, and so on.
  # You should provide a list of connections for a packager to get access to them.
  targets: []
  # You can set connection which points to some bucket where the Trained Model Binary is stored
  # then packager will extract your binary from this connection.
  # Optional. Default value is taken from the ODAHU cluster configuration.
  outputConnection: custom-connection
  # Node selector that exactly matches a node pool from ODAHU config
  # This is optional; when omitted, ODAHU uses any of available packaging node pools
  # Read more about node selector: https://kubernetes.io/docs/concepts/scheduling-eviction/assign-pod-node/
  nodeSelector:
    label: value
# Every packager saves its results into status field.
# Example of fields: docker image or python packager name.
status:
  results:
    - name: some_param
      value: some_value
```

> ***Note***
>
> You can find an artifactName in the *status.artifactName* field of a model training entity.

Packagers management

Packagers can be managed using the following ways.

Swagger UI

ModelPackaging and PackagingIntegration are available on the Swagger UI at http://api-service/swagger/index.html URL.

ODAHU CLI

Odahuflowctl supports the Packagers API. You must be logged in if you want to get access to the API.

Getting all packaging in json format:

```
odahuflowctl pack get --format json
```

Getting the arguments of the packagers:

```
odahuflowctl pack get --id tensorflow-cli -o 'jsonpath=[*].spec.arguments'
```

- Creating of a packager from *pack.yaml* file:

```
odahuflowctl pack create -f pack.yaml
```

- All commands and documentation for packager at Odahu cluster:

```
odahuflowctl pack --help
```

We also have local packager:

```
odahuflowctl local pack --help
```

and can run packaging locally:

```
odahuflowctl local pack run --id [Model packaging ID] -d [Path to an Odahu manifest file]
```

more information you can find at **Local Quickstart**

JupyterLab

Odahu-flow provides the JupyterLab extension for interacting with Packagers API.

Model Docker Dependencies Cache

ODAHU Flow downloads your dependencies on every docker model packaging launch. You can experience the following troubles with this approach:

- downloading and installation of some dependencies can take a long time
- network errors during downloading dependencies

To overcome these and other problems, ODAHU Flow provides a way to specify a prebuilt packaging Docker image with your dependencies.

> **Note**
>
> If you have different versions of a library in your model conda file and cache container, then the model dependency has a priority. It will be downloaded during model packaging.

First of all, you have to describe the Dockerfile:

- Inherit from a release version of odahu-flow-docker-packager-base
- Optionally, add install dependencies
- Add a model conda file
- Update the `odahu_model` conda environment.

Example of Dockerfile:

```
FROM odahu/odahu-flow-docker-packager-base:1.1.0-rc11

# Optionally
# RUN pip install gunicorn[gevent]

ADD conda.yaml ./
RUN conda env update -n ${ODAHU_CONDA_ENV_NAME} -f conda.yaml
```

Build the docker image:

```
docker build -t packaging-model-cache:1.0.0 .
```

Push the docker image to a registry:

```
docker push packaging-model-cache:1.0.0
```

Specify the image in a model packaging:

Packaging example

```
kind: ModelPackaging
id: model-12345
spec:
  arguments:
    dockerfileBaseImage: packaging-model-cache:1.0.0
  ...
```

Docker REST

The Docker REST packager wraps an ML model into the REST service inside a Docker image. The resulting service can be used for point prediction through HTTP.

The packager provides the following list of targets:

Target Name	Connection Types	Required	Description
docker-push	docker, ecr	True	The packager will use the connection for pushing a Docker image result
docker-pull	docker, ecr	False	The packager will use the connection for pulling a custom base Docker image

The packager provides the following list of arguments:

Argument Name	Type	Default	Required	Description
imageName	string	{{ Name }}-{{ Version }}:{{ RandomUUID }}	False	This option provides a way to specify the Docker image name. You can hardcode the full name or specify a template. Available template values: Name (Model Name), Version (Model Version), RandomUUID. Examples: myservice:123, {{ Name }}:{{ Version }}
port	integer	5000	False	Port to bind
timeout	integer	60	False	Serving timeout in seconds.
workers	integer	1	False	Count of serving workers
threads	integer	4	False	Count of serving threads
host	string	0.0.0.0	False	Host to bind
dockerfileBaseImage	string	python:3.6	False	Base image for Dockerfile

The packager provides the following list of result fields:

Name	Type	Description
image	string	The full name of a built Docker image

Let’s build a couple of examples of Docker REST packager. The packager requires docker or ecr connection types. The following example assumes that you have created a connection with *test-docker-registry* id and *gcr.io/project/odahuflow* URI.

Minimal Example of Docker REST packager

```yaml
kind: ModelPackaging
id: "docker-rest-packager-example"
spec:
    integrationName: docker-rest
    artifactName: wine-model-123456789.zip
    targets:
        - connectionName: test-docker-registry
          name: docker-push
```

Then a result of the packager will be something like this: “gcr.io/project/odahuflow/wine-0-1:ec1bf1cd-216d-4f0a-a62f-bf084c79c58c”.

Now, let’s try to change the docker image name and number of workers.

Docker REST packager with custom arguments

```yaml
kind: ModelPackaging
id: "docker-rest-packager-example"
spec:
    integrationName: docker-rest
    artifactName: wine-model-123456789.zip
    targets:
        - connectionName: test-docker-registry
          name: docker-push
    arguments:
        imageName: "wine-test:prefix-{{ RandomUUID }}"
        workers: 4
```

```
odahuflowctl pack get --id "docker-rest-packager-example" -o 'jsonpath=$[0].status.results[0].value'
```

Then a result of the packager will be something like this: “gcr.io/project/odahuflow/wine-test:prefix-ec1bf1cd-216d-4f0a-a62f-bf084c79c58c”.

You can run the image locally using the following command:

```
docker run -it --rm --net host gcr.io/project/odahuflow/wine-test:prefix-ec1bf1cd-216d-4f0a-a62f-bf084c79c58c
```

The model server provides two urls:

- GET */api/model/info* - provides a swagger documentation for a model
- POST */api/model/invoke* - executes a prediction

```
curl http://localhost:5000/api/model/info
curl -X POST -d '{"columns": ["features","features","features"], "data": [[1, 2, 3], [4, 5, 6]]}' -H "Content-Type: application/json" http://localhost:5000/api/model/invoke
```

Docker REST predict API

```json
{
  "columns": [
    "features",
    "features",
    "features"
  ],
  "data": [
    [
      1,
      2,
      3,
    ],
```

```
      [
        4,
        5,
        6,
      ]
    ]
}
```

Docker REST prediction result

```
{
  "prediction": [
    [
      0.09405578672885895
    ],
    [
      0.01238546592343845
    ]
  ],
  "columns": [
    "predictions"
  ]
}
```

Docker CLI

The Docker CLI packager wraps an ML model into the CLI inside a Docker image. The resulting service can be used for batch prediction.

The packager provides the following list of targets:

Target Name	Connection Types	Required	Description
docker-push	docker, ecr	True	The packager will use the connection for pushing a Docker image result
docker-pull	docker, ecr	False	The packager will use the connection for pulling a custom base Docker image

The packager provides the following list of arguments:

Argument Name	Type	Default	Required	Description
imageName	string	{{ Name }}-{{ Version }}:{{ RandomUUID }}	False	This option provides a way to specify the Docker image name. You can hardcode the full name or specify a template. Available template values: Name (Model Name), Version (Model Version), RandomUUID. Examples: myservice:123, {{ Name }}:{{ Version }}

dockerfileBaseImage	string	python:3.6	False	Base image for Dockerfile

The packager provides the following list of result fields:

Name	Type	Description
image	string	The full name of a built Docker image

Let's build a couple of examples of Docker CLI packager. The packager requires docker or ecr connection types. The following example assumes that you have created a connection with *test-docker-registry* id and *gcr.io/project/odahuflow* URI.

Minimal Example of Docker CLI packager

```yaml
kind: ModelPackaging
id: "docker-cli-packager-example"
spec:
    integrationName: docker-cli
    artifactName: wine-model-123456789.zip
    targets:
        - connectionName: test-docker-registry
          name: docker-push
```

Then a result of the packager will be something like this: "gcr.io/project/odahuflow/wine-0-1:ec1bf1cd-216d-4f0a-a62f-bf084c79c58c".

Now, let's try to change the docker image name and the base image.

Docker CLI packager with custom arguments

```yaml
kind: ModelPackaging
id: "docker-cli-packager-example"
spec:
    integrationName: docker-cli
    artifactName: wine-model-123456789.zip
    targets:
        - connectionName: test-docker-registry
          name: docker-push
    arguments:
        imageName: "wine-test:prefix-{{ RandomUUID }}"
        dockerfileBaseImage: "python:3.7"
```

```
odahuflowctl pack get --id "docker-cli-packager-example" -o 'jsonpath=$[0].status.results[0].value'
```

Then a result of the packager will be something like this: "gcr.io/project/odahuflow/wine-test:prefix-ec1bf1cd-216d-4f0a-a62f-bf084c79c58c".

You can run the image locally using the following command:

```
docker run -it --rm --net host gcr.io/project/odahuflow/wine-test:prefix-ec1bf1cd-216d-4f0a-a62f-bf084c79c58c --help
```

The model CLI provides two commands:

- *predict* - Make predictions using GPPI model
- *info* - Show model input/output data schema

Docker CLI info command

```
docker run -it --rm --net host gcr.io/project/odahuflow/wine-test:prefix-ec1bf1cd-216d-4f0a-a62f-bf084c79c58c info
```

Docker CLI info command output

```
Input schema:
{
    "columns": {
        "example": [
            "features",
            "features",
            "features",
        ],
        "items": {
            "type": "string"
        },
        "type": "array"
    },
    "data": {
        "items": {
            "items": {
                "type": "number"
            },
            "type": "array"
        },
        "type": "array",
        "example": [
            [
                0,
                0,
                0,
            ]
        ]
    }
}
Output schema:
{
    "prediction": {
        "example": [
            [
                0
            ]
        ],
        "items": {
            "type": "number"
        },
        "type": "array"
    },
    "columns": {
        "example": [
            "predictions"
        ],
        "items": {
            "type": "string"
        },
        "type": "array"
    }
}
```

Let’s make a batch prediction.

Create a predict file

```
mkdir volume
cat > volume/predicts.json <<EOL
{
  "columns": [
```

```
    "features",
    "features",
    "features",
  ],
  "data": [
    [
      1,
      2,
      3
    ],
    [
      4,
      5,
      6
    ]
  ]
}
EOL
docker run -it --rm --net -v volume:/volume host gcr.io/project/odahuflow/wine-test:prefix-ec1bf1cd-216d-4f0a-a62f-bf084c79c58c predict /volume/predicts.json /volume
```

Result of prediction

```
cat volumes/result.json
{
  "prediction": [
    [
      0.09405578672885895
    ],
    [
      0.01238546592343845
    ]
  ],
  "columns": [
    "predictions"
  ]
}
```

Nvidia Triton Packager

Triton Packager wraps model with Triton Inference Server. The server supports multiple ML frameworks. Depending on the framework the packager expects different input.

Required files:

- model file/directory with fixed naming. Refer to Triton Backend Docs to find more specific information on particular Triton backend.
    - TensorRT: `model.plan`
    - TensorFlow SavedModel: `model.savedmodel/...`
    - TensorFlow Grafdef: `model.graphdef`
    - ONNX: `model.onnx` file or directory
    - TorchScript: `model.pt`
    - Caffe 2 Netdef: `model.netdef` + `init_model.netdef`
- `config.pbtxt`, Triton config file (Triton Model Configuration Docs). Optional for the following backends:
    - TensorRT

- TF SavedModel
- ONNX

Optional files:

- `odahuflow.model.yaml` in the following format. When omitted defaults to model `model` of version `1`;

```yaml
name: model
version: 1
```

- `conda.yaml` for Python backend. If conda-file detected new conda env is created and used for run model.
- Any other arbitrary files will be copied and put next to model file.

Targets, Arguments and Results

Triton Packager Targets:

Target Name	Connection Types	Required	Description
docker-push	docker, ecr	True	The packager will use the connection for pushing a Docker image result

Triton Packager Arguments:

Argument Name	Type	Default	Required	Description
imageName	string	{{ Name }}-{{ Version }}:{{ RandomUUID }}	False	This option provides a way to specify the Docker image name. You can hardcode the full name or specify a template. Available template values: Name (Model Name), Version (Model Version), RandomUUID. Examples: myservice:123, {{ Name }}:{{ Version }}
triton_base_image_tag	string	20.11-py3	False	Triton Base image tag for Dockerfile

Triton Packager Results:

Name	Type	Description
image	string	The full name of a built Docker image

Example

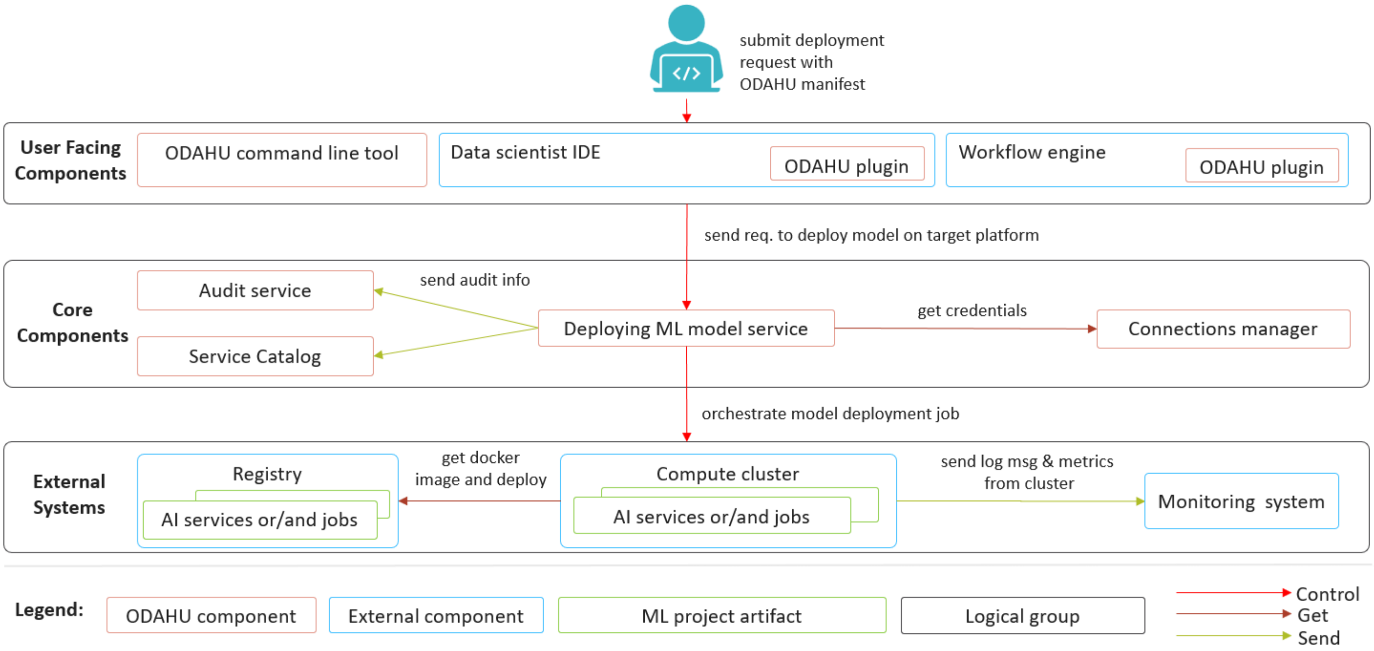
Example input file structure for Python Backend:

- `model.py` - the Python module that implements interface expected by Triton;
- `odahuflow.model.yaml` - simple manifest with model name and version
- `conda.yaml` - describes Conda environment for model
- `config.pbtxt` - Triton Model config file (specification)
- `data.json`... - arbitrary file(s) that will be put next to model file

Triton packaging with custom arguments

```yaml
id: "triton-packager-example"
spec:
    integrationName: docker-triton
    artifactName: model-123456789.tar
    targets:
        - connectionName: test-docker-registry
          name: docker-push
    arguments:
        imageName: "triton-model:prefix-{{ RandomUUID }}"
```

Model Deployments



ODAHU model deployment component allows to deploy ML models as an inference online services or batch jobs in a Kubernetes cluster.

Features:

- Automatic scaling of deployed inference service instances.
- Monitoring of deployed inference services.
- Various API traffic routing polices (A/B, Canary).
- Inference request and response logging in a structured form with unique id to be used in feedback loop.
- Dynamic OpenAPI/Swagger for deployed inference service APIs.
- Inference service API secured with JWT and access control polices.

Inference Servers

A model can be deployed in ODAHU if only it is packed with a supported Inference Server. Inference Server is typically a web service that “wraps” an ML model and lets remote clients to invoke the model via HTTP (or any other protocol).

An Inference Servers that wraps the model has to be indicated in `predictor` field of a Model Deployment.

ODAHU currently supports several Inference Servers:

- ODAHU Inference Server: `predictor: odahu-ml-server`
- NVIDIA Triton Inference Server: `predictor: triton`

ODAHU Inference Server

Value for "predictor" field of Model Deployment: `predictor: odahu-ml-server`

ODAHU Inference Server is an inference server that builds a simple HTTP layer on top of any MLFlow model with an HTTP layer.

To pack a model into ODAHU Inference Server Docker REST packager has to be used.

NVIDIA Triton Inference Server

Value for "predictor" field of Model Deployment: `predictor: triton`

Triton Server is a feature-rich inference server. To pack a model into a Triton Server, Triton Packager has to be used.

Triton Server uses KFServing Inference Protocol.

General Deployment Manifest Structure

Deployment API

```yaml
kind: ModelDeployment
# Some unique value among all deployments
# Id must:
#  * contain at most 63 characters
#  * contain only lowercase alphanumeric characters or '-'
#  * start with an alphabetic character
#  * end with an alphanumeric character
id: wine-12345
spec:
    # Predictor is an inference backend name; required field
    # Possible values are: odahu-ml-server, triton
    predictor: odahu-ml-server

    # Model image is required value. Change it
    image: gcr.io/project/test-e2e-wine-1.0:b591c752-43d4-43e0-8392-9a5715b67573
    # If the Docker image is pulled from a private Docker repository then
    # you have to create a Odahu-flow connecton and specify its id here.
    # imagePullConnID: test

    # Compute resources for the deployment job.
    resources:
      limits:
        cpu: 1
        memory: 1Gi
      requests:
        cpu: 1
        memory: 1Gi

    # Minimum number of replicas
    minReplicas: 0
    # Maximum number of replicas
    maxReplicas: 1
```

Model Deployment management

Model Deployments can be managed using the following ways.

ModelDeployments are available in the Swagger UI at http://api-service/swagger/index.html URL.

Odahuflowctl supports the Model Deployment API. You must be logged in if you want to access the API.

Getting all model deployments in json format:

```
odahuflowctl deployment get --format json
```

Getting the model name of the model deployments:

```
odahuflowctl deployment get --id tensorflow-cli -o 'jsonpath=[*].spec.model.name'
```

- Creating of a deployment from *deploy.yaml* file:

```
odahuflowctl deployment create -f deploy.yaml
```

- All model deployments commands and documentation:

```
odahuflowctl deployment --help
```

- All model deployments commands and documentation:

```
odahuflowctl deployment --help
```

- Getting a model deployment information:

```
odahuflowctl model info --md wine
```

- Making a prediction:

```
odahuflowctl model invoke --md wine --file request.json
```

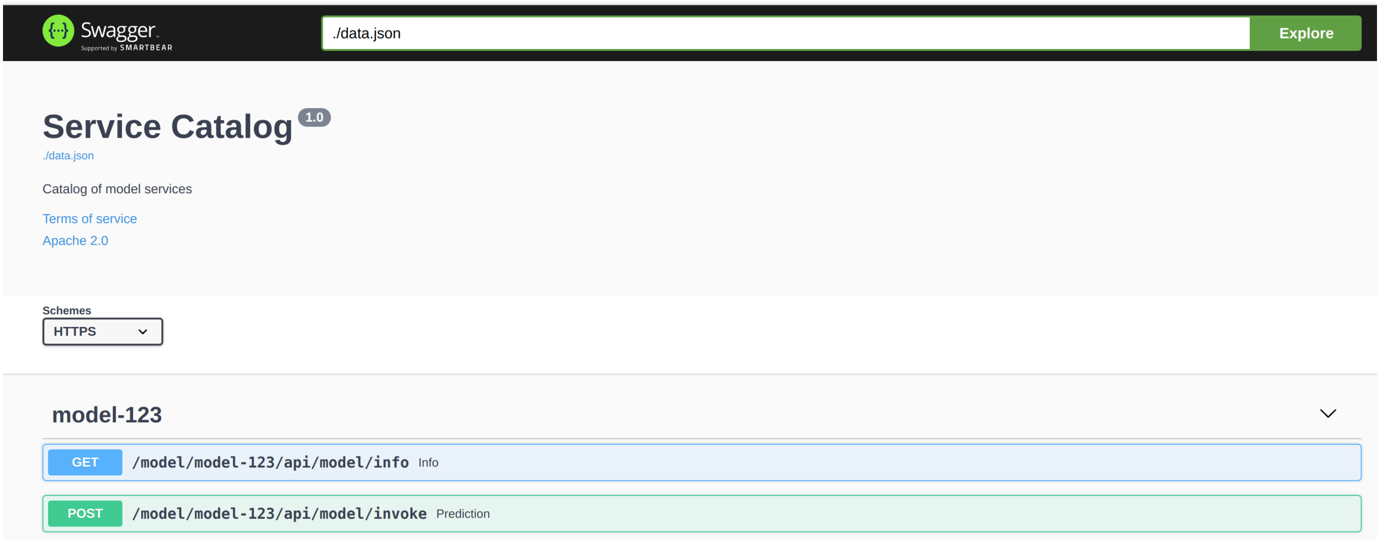
Odahu-flow provides the JupyterLab extension for interacting with Model Deployments API.

Service Catalog

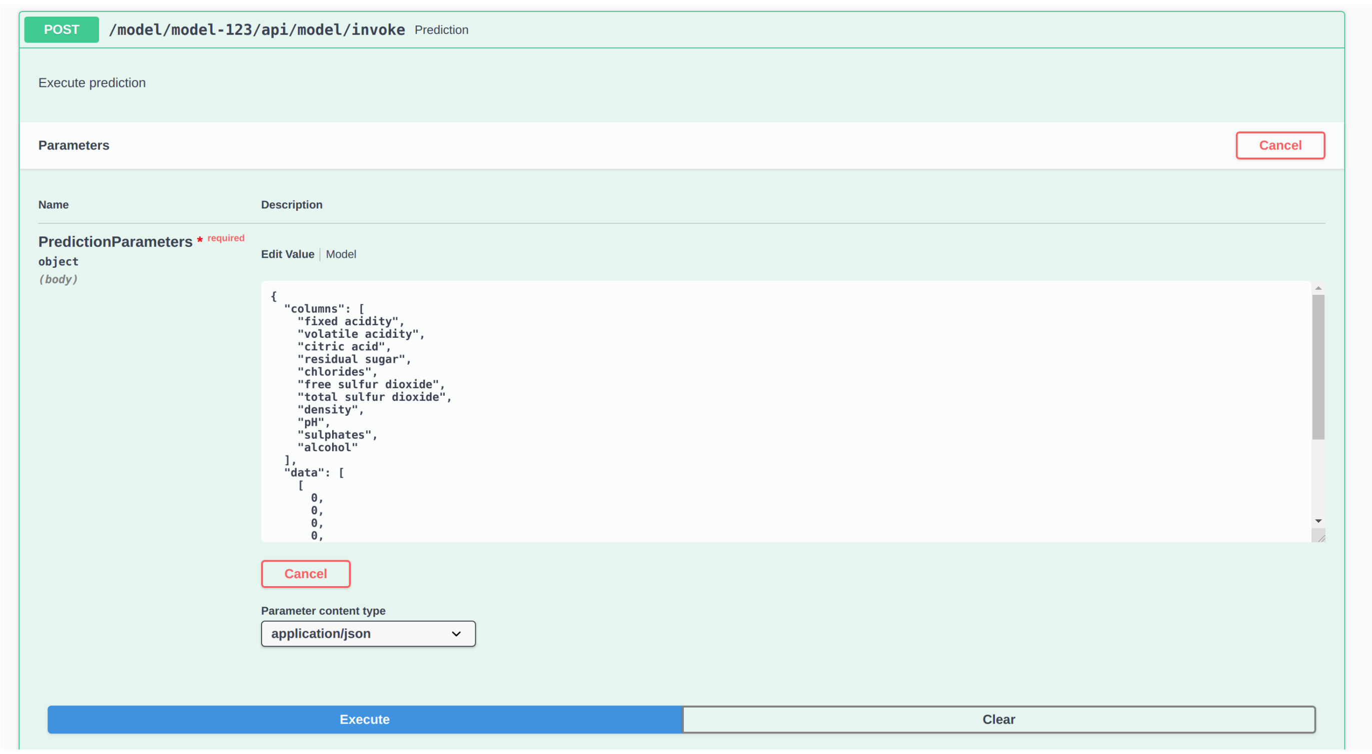
Service catalog provides a Swagger UI for Model Deployments.

> ***Note***
>
> The model must provide input and output samples to appear in the Service Catalog

Service catalog Swagger UI:



Example of a prediction request:

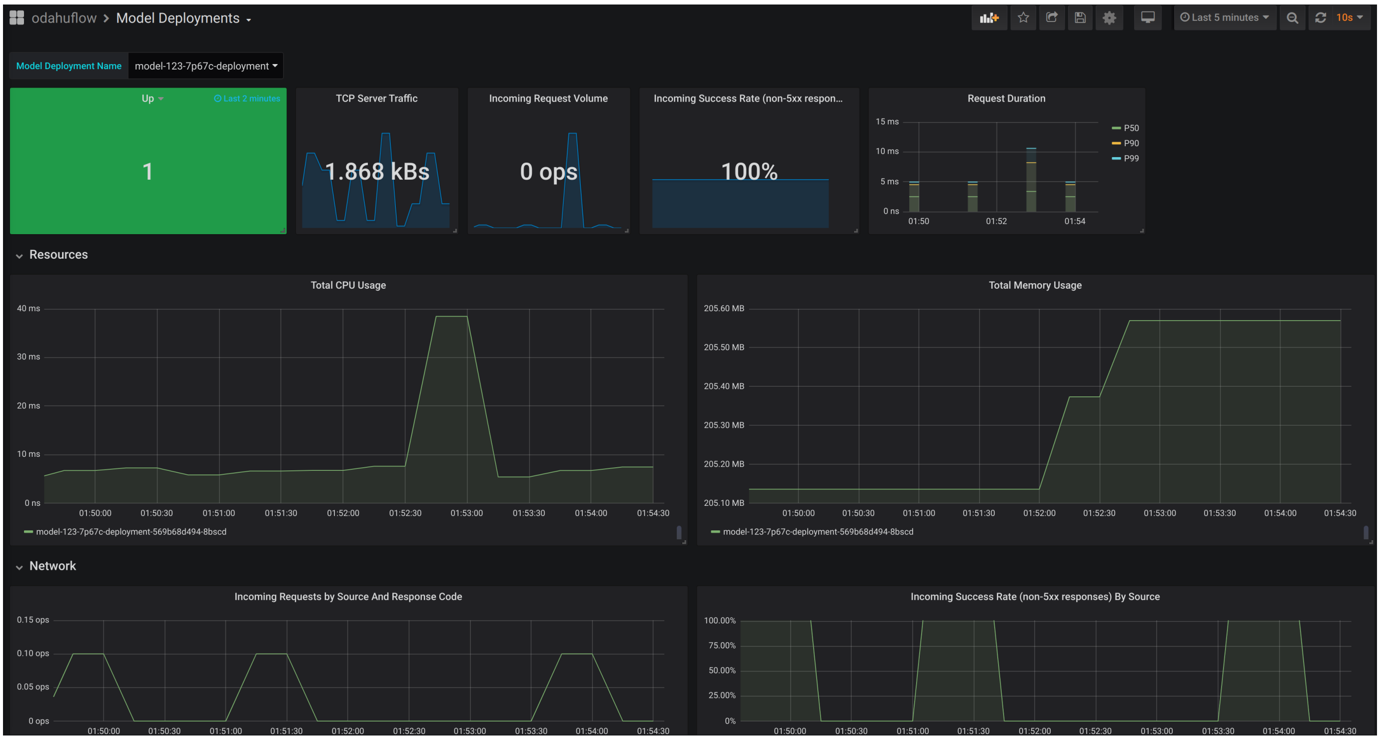


Grafana Dashboard

Out of the box, Odahu-flow provides the Grafana Model Deployment dashboard. It contains the charts with following system metrics:

- availability
- replicas
- CPU
- memory
- number of failed HTTP requests
- latency
- ...

Example of the dashboard:



Feedback

Model Feedback provides a view of performance over all stages of model lifecycle.

The mechanism is simple:

1. Ask Deploy for prediction (with or without `Request-Id` provided)
2. Send prediction feedback to Odahu-flow (with `Request-Id` returned from previous step)
3. Odahu-flow stores the prediction and feedback to a configurable location

> ***Important***
>
> This flow requires `feedback` to be enabled in `values.yaml` during Helm chart installation

1. If prediction is requested without Request-ID: `Request-ID` header with random ID is added to the request. Otherwise, Request-ID is not generated.
2. Request and response are stored on configured external storage (eg. S3, GCS)

3. User sends Model Feedback as an argument to the feedback endpoint. (Feedback can be arbitrary JSON.) 5. All Feedback is persisted on external storage and can be used by models during subsequent Trains.

Making a prediction request:

```
curl -X POST -vv "https://``cluster-url``/model/``model-deployment-id``/api/model/invoke" \
-H "Authorization: Bearer ``JWT`` \
-H "accept: application/json" \
-H "Content-Type: application/json" \
-d "{ \"columns\": [ \"fixed acidity\", \"volatile acidity\", \"citric acid\", \"residual sugar\", \"chlorides\", \"free sulfur dioxide\", \"total sulfur dioxide\", \"density\", \"pH\", \"sulphates\", \"alcohol\" ], \"data\": [ [ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 ] ]}"
```

The response contains a generated `Request-Id` header.

```
HTTP/2 200
server: nginx/1.13.12
date: Tue, 17 Dec 2019 10:58:49 GMT
content-type: application/json
content-length: 45
model-name: test-e2e-wine
model-version: 1.0
request-id: 6fa1f636-fb80-9979-b8c6-d78f5e90f0c1
x-envoy-upstream-service-time: 43
strict-transport-security: max-age=15724800; includeSubDomains

{"prediction": [6.0], "columns": ["quality"]}
```

Requests and responses are persisted in a bucket. (File name ~= `/request_response/income/1.1/year=2019/month=07/day=24/2019072414_4.json`)

The first file contains meta information about request and response:


```
{
  "request_id": "6fa1f636-fb80-9979-b8c6-d78f5e90f0c1",
  "request_content": "{\"columns\": [\"alcohol\", \"chlorides\", \"citric acid\", \"density\", \"fixed acidity\", \"free sulfur dioxide\", \"pH\", \"residual sugar\", \"sulphates\", \"total sulfur dioxide\", \"volatile acidity\"], \"data\": [[12.8, 0.029, 0.48, 0.98, 6.2, 29, 3.33, 1.2, 0.39, 75, 0.66]]}",
  "request_uri": "/model/test-e2e-wine/api/model/invoke",
  "response_http_headers": {
    ":status": "200",
    "date": "Tue, 17 Dec 2019 08:46:40 GMT",
    "model-name": "test-e2e-wine",
    "model-version": "1.0",
    "server": "istio-envoy",
    "connection": "close",
    "content-length": "45",
    "content-type": "application/json",
    "request-id": "12dcddd5-771d-9fc0-b326-816211cf8172",
    "x-envoy-upstream-service-time": "6"
  },
  "model_version": "1.0",
  "request_http_headers": {
    ":method": "POST",
    "x-scheme": "https",
    "accept": "*/*",
    "x-istio-attributes": "ClEKF2Rlc3RpbmF0aW9uLnNlcnZpY2UudWlkEjYSNGlzdGlvOi8vb2RhaHUtZmxvdy1kZXBsb3ltZW50L3NlcnZpY2VzL3Rlc3QtZTJlLXdpbmUKUwoYZGVzdGluYXRpb24uc2VydmljZS5ob3N0EjcSNXRlc3QtZTJlLXdpbmUub2RhaHUtZmxvdy1kZXBsb3ltZW50LnN2Yy5jbHVzdGVyLmxvY2FsCisKGGRlc3RpbmF0aW9uLnNlcnZpY2UubmFtZRIPEg10ZXN0LWUyZS13aW5lCjgKHWRlc3RpbmF0aW9uLnNlcnZpY2UubmFtZXNwYWNlEhcSFW9kYW[illegible]lcnZpY2UubmFtZXNwYWNlEhcSFW9kYWh1LWZsb3ctZGVwbG95bWVudApPCgpzb3VyY2UudWlkEkESP2t1YmVybmV0ZXM6Ly9pc3Rpby1pbmdyZXNzZ2F0ZXdheS04NjlkYjdkOWJiLWpsemtyLmlzdGlvLXN5c3RlbQ==",
    "knative-serving-namespace": "odahu-flow-deployment",
    "x-envoy-original-path": "/model/test-e2e-wine/api/model/invoke",
    ":scheme": "http",
    "x-forwarded-host": "odahu.example.com",
    "x-original-uri": "/model/test-e2e-wine/api/model/invoke",
    "user-agent": "python-requests/2.22.0",
    "x-forwarded-port": "443",
    "content-type": "application/json",
    "x-b3-sampled": "1",
    "content-length": "257",
    "x-real-ip": "10.4.0.11",
    "x-forwarded-for": "10.4.0.11,10.44.0.10",
    "x-envoy-external-address": "10.44.0.10",
    "x-request-id": "12dcddd5-771d-9fc0-b326-816211cf8172",
    "x-forwarded-proto": "http",
    ":authority": "odahu.example.com",
    "x-b3-traceid": "488abe197a652c2ce80e6b848a4c56e6",
    ":path": "/api/model/invoke",
    "accept-encoding": "gzip, deflate",
    "x-envoy-decorator-operation": "test-e2e-wine.odahu-flow-deployment.svc.cluster.local:80/model/test-e2e-wine/api*",
    "x-b3-spanid": "e80e6b848a4c56e6",
    "knative-serving-revision": "test-e2e-wine-l4xt7"
  },
  "response_status": "200",
  "request_host": "odahu.example.com",
  "model_name": "test-e2e-wine",
  "request_http_method": "POST",
  "time": "2019-12-17 08:46:40 +0000"
}
```


The second file contains the response body with the same `Request-Id` (File name ~= `/response_body/income/1.1/year=2019/month=07/day=24/2019072414_1.json`)


```
{
  "request_id": "6fa1f636-fb80-9979-b8c6-d78f5e90f0c1",
  "model_version": "1.0",
  "model_name": "test-e2e-wine",
  "response_content": "{\"prediction\": [6.0], \"columns\": [\"quality\"]}",
  "time": "2019-12-17 08:46:40 +0000"
}
```


Send Model Feedback request:

```
curl -X POST -vv "${BASE_URL}/feedback/model/" \
-H "Authorization: Bearer ${JWT}" \
-H "x-model-name: income" \
```

```
-H "x-model-version: 1.1" \
-H "Request-ID: previous-prediction-id" \
-H 'Content-Type: application/json' \
-d '{"truthful": 1}'
```

Note that the `-d` argument can pass arbitrary JSON.

A successful feedback request will have the following properties:

- HTTP response: 200
- Response field `error` is `false`.
- Response field `registered` is `true`.
- Response field `message` is what was sent to storage.

Example response

```
{
  "message": {
    "RequestID": "previous-prediction-id",
    "ModelVersion": "1.0",
    "ModelName": "test-e2e-wine",
    "Payload": {
      "json": {
        "truthful": 1
      }
    }
  }
}
```

File name ~= `/feedback/test-e2e-wine/1.0/year=2019/month=11/day=23/2019072311_2.json` will have a format like this, with feedback stored in the `payload` field:

```
{
  "request_id": "previous-prediction-id",
  "model_version": "1.0",
  "model_name": "test-e2e-wine",
  "payload": {
    "json": {
      "truthful": 1.0
    }
  },
  "time": "2019-12-17 20:08:05 +0000"
}
```

Batch Inference

This section describes API and protocols related to Batch inference using ODAHU.

ODAHU Batch Inference feature allows user to get inferences using ML model for large datasets that are delivered asyncronously, not via HTTP API, but through other mechanisms.

Currently Batch Inference supports the following ways to delivery data for forecasting:

- Object storage
  - GCS
  - S3
  - Azureblob

In future we consider to add ability to process data directly from Kafka topic and other async data sources.

Please also take a look at example.

API Reference

InferenceService

InferenceService represents the following required entities:

- Predictor docker image that contains predictor code
- Model files location on object storage (directory or .zip / .tar.gz archive)
- Command and Arguments that describe how to execute image

When a user trains a model then they should build an image with code that follows Predictor code protocol and register this image as well as appropriate model files using `InferenceService` entity in ODAHU Platform.

User describes how inference should be triggered using different options in `[].spec.triggers`.

InferenceJob

`InferenceJob` describes forecast process that was triggered by one of the triggers in `InferenceService`. If `[].spec.triggers.webhook` is enabled then its possible to run `InferenceJob` by making POST request as described below. By default webhook trigger is enabled. Note, that currently its the only one way to trigger jobs.

Predictor code protocol

ODAHU Platform launches docker image provided by user as `[].spec.image` (InferenceService) and guarantees the next conventions about input/model location inside container as well as format of input and output data.

Env variables

Title

Env variable	Description
$ODAHU_MODEL	Path in local filesystem that contains all model files from `[].spec.modelSource`
$ODAHU_MODEL_INPUT	Path in local filesystem that contains all input files from `[].spec.dataSource`
$ODAHU_MODEL_OUTPUT	Path in local filesystem that will be uploaded to `[].spec.outputDestination`

Input and output formats

Predictor code must expect input as set of JSON files with extensions `.json` located in folder that can be found in `$ODAHU_MODEL_INPUT` environment variable. These JSON files have structure of Kubeflow inference request objects.

Predictor code must save results as set of JSON files with extension `.json` in the folder that can be found in `$ODAHU_MODEL_INPUT` environment variable. These JSON files must have structure of Kubeflow inference response objects.

Implementation details

This section helps with deeper understanding of underlying mechanisms.

`InferenceJob` is implemented as TektonCD TaskRun with 9 steps

1. Configure rclone using ODAHU connections described in `BatchInferenceService`
2. Sync data input from object storage to local fs using rclone
3. Sync model from object storage to local fs using rclone
4. Validate input to Predict Protocol - Version 2
5. Log Model Input to feedback storage
6. Run user container with setting `$ODAHU_MODEL`, `$ODAHU_MODEL_INPUT`, `$ODAHU_MODEL_OUTPUT`

7. Validate output to Predict Protocol - Version 2
8. Log Model Output to feedback storage
9. Upload data from `$ODAHU_MODEL_OUTPUT` to `[].spec.outputDestination.path`

Glossary

VCS
: Version control system. A service that stores model source code for development and deployment procedures (e.g. a GitHub Repository).

Trained Model Binary
: An archive containing a trained ML/AI model (inference code, model weights, etc). Odahu defines a format for these binaries. See <ref_model_format.html>

Trainer
: Application that uses model source code, **Data Bindings**, **Connections** and **Training Hyperparameters** to produce a **Trained Model Binary**.

Data Binding
: Reference to remote data (e.g. files from S3) should be placed for a **Train** process.

Connection
: Credentials for an external system. For example: Docker Registry, cloud storage location, etc.

Training Hyperparameters
: Parameter for Training process. For example, count of epochs in evolution algorithms.

Train
: A containerized process that converts model source code, **Data Bindings**, **Connections** and **Training Hyperparameters** to **Trained Model Binary** using a **Trainer** defined in a **Trainer Extension**

Trainer Extension
: A pluggable **Train** implementation.

Packager
: Containerized application that uses a **Trained Model Binary** and **Connections** and converts them into a target Archive. Typically this is a Docker image with REST API.

Package
: Containerized process which turns a **Trained Model Binary** into a Docker image with REST API using a **Packager Extension**.

Packager Extension
: A pluggable **Package** implementation.

Deployer
: Containerized application that uses the results of a **Package** process and **Connections** to deploy a packaged model on a Kubernetes cluster.

Deploy
: Containerized process that deploys results of a **Package** operation to Kubernetes cluster with a REST web service.

Trainer Metrics

Metrics set by *Trainer* code during *Train* (e.g. accuracy of model). These metrics can be used for querying and comparing *Train* events.

Trainer Tags

Key/value value pairs that are set by *Trainer* code (e.g. type of algorithm). Can be used for querying and comparing *Train* runs.

General Python Prediction Interface

Format of storing models, written in a Python language

MLflow Trainer

Integration of MLflow library for training models, written in a Python. Details - MLFlow Trainer

REST API Packager

Integration for packing trained models into Docker Image with served via REST API

API service

API for managing Odahu Platform resources for cloud deployed Platform

Operator

A Kubernetes Operator that manages Kubernetes resources (Pods, Services and etc.) for Odahu **Train**, **Package**, and **Deploy** instances.

Prediction

A deployed model output, given input parameters.

Model prediction API

API provided by deployed models to allow users to request predictions through a web service.

Prediction Feedback

Feedback versus the previous **prediction**, e.g. prediction correctness.

Model Feedback API

An API for gathering **Prediction Feedback**

Feedback aggregator

A service that provides a **Model Feedback API** and gathers input and output **prediction requests**

Odahu-flow SDK

An extensible Python client library for **API service**, written in Python language. Can be installed from PyPi.

Odahu-flow CLI

Command Line Interface for **API service**, written in Python. Can be installed from PyPi. It uses the **Odahu-flow SDK**.

Plugin for JupyterLab

A odahu-specific plugin that provides Odahu Platform management controls in JupyterLab.

Plugin for Jenkins

A library for managing Odahu Platform resources from Jenkins Pipelines.

Plugin for Airflow

A library that provides Hooks and Operators for managing Odahu Platform resources from Airflow.

Model Deployment Access Role Name

Name of scope or role for accessing model deployments.

JWT Token

A JSON Web Token that allows users to query deployed models and to provide feedback. This token contains an encoded **role name**.

A/B testing

Process of splitting predictions between multiple **Model Deployments** in order to compare prediction metrics and **Model Feedback** for models, which can vary by **source code**, **dataset** and/or **training hyperparameters**

Odahu distribution

A collection of Docker Images, Python packages, or NPM packages, which are publicly available for installation as a composable Odahu Platform.

Odahu Helm Chart

A YAML definition for Helm that defines a Odahu Platform deployed on a Kubernetes cluster.

Odahu-flow's CRDs

Objects that **API service** creates for actions that require computing resources to be stored. For example: **connections**, **Trains**, etc.

These objects are Kubernetes Custom Resources and are managed by **operator**.

Identity Provider (idP)

A component that provides information about an entity (user or service).

Policy Enforcement Point (PEP)

A component that enforces security policies against each request to API or other protected resources.

Policy Decision Point (PDP)

A component that decides whether the request (action in the system) should be permitted or not.

Changelog

Odahu 1.6.0, 3 September 2021

Features:

- Core:
  - MLFlow artifacts storage is now correctly works with cloud storage for Google Cloud and Amazon.

Bug Fixes:

- Core:
  - Model feedback & Triton model logs are now stored with model name/version (#607,).

Odahu 1.5.0, 1 August 2021

Features:

- Core:
  - New Batch Inference API (#500, #537).
  - Object storage added as an option of ML project source code repository (#360).
- Python SDK:
  - Add clients to work with User and Feedback entities (#295).

Updates

- Core:
  - Set *model-name/model-version* headers on service mesh level (#496). That looses the requirements to inference servers. Previously any inference server (typically a model is packed into one on Packaging

stage) was obligated to include the headers into response for feedback loop to work properly. That rule restricts from using any third-party inference servers (such as NVIDIA Triton), because we cannot control the response headers.

- Removed deprecated fields *updateAt*/*createdAt* from core API entities (#394).
- Move to recommended and more high-level way of using Knative which under-the-hood is responsible for a big part of *ModelDeployment* functionality (#347).

- CLI:
  - Model *info* and *invoke* parameter *JWT* renamed to *token* (#577).
  - Usage descriptions updated (#577).
  - Auth tokens are automatically refreshing (#509).
- Aiflow plugin:
  - Airflow plugin operators expect a service account's `client_secret` in a `password` field of *Airflow Connection* now. previously it expects `client_secret` in `extra` field. (#29).

    `Breaking change!`: You should recreate all Airflow connections for ODAHU server by moving the `client_secret` from the `extra` field into the `password` field.

    Please do not forget to remove your `client_secret` from the `extra` field **for security reasons**.

Bug Fixes:

- Core:
  - Fix & add missing updatedAT/createdAT (#583, #600, #601, #602).
  - Training result doesn't contain commit ID when using object storage as algorythm source (#584).
  - RunID is now present for model training with mlflow toolchain (#581).
  - InferenceJob objects can now be deleted correctly (#555).
  - Deployment *roleName* changes now applies correctly (#533).
  - X-REQUEST-ID header are now correctly handled on service mesh layer to support third-party inference servers (#525).
  - Fix packaging deletion via *bulk delete* command (#416).

Odahu 1.4.0, 27 February 2021

Features:

- Core:
    - Triton Packaging Integration (Nvidia Triton Packager) added as a part of Triton pipeline (#437).
    - Local training & packaging now covered with tests (#157).
    - MLflow toolchain with custom format for model training artifact (#31).
- UI:
    - New *Play* tab on Deployment page provides a way to get deployed model metadata and make inference requests from the UI (#61).
    - New *Logs* tab on Deployment page provides a way to browse logs of deployed model (#45).
    - User now can create packaging and deployments based on finished trainings and packagings (#38).

Updates:

- Core:
    - Service catalog is rewritten (#457).
    - Deployed ML models performance optimized (#357).
    - OpenPolicyAgent-based RBAC for deployed models are implemented (#238).
- CLI:
    - Option `--disable-target` for `odahuflowctl local pack run` command added. targets which will be passed to packager process. You can use multiple options a `odahuflowctl local pack run ... --disable-target=docker-pull --disab`
    - Options `--disable-package-targets/--no-disable-package-targets` for `odahuflowctl local pack run` command are deprecated.
    - `odahuflowctl local pack run` behavior that implicitly disables all targets by default is deprecated.

Bug Fixes:

- Core:
    - Knative doesn't create multiple releases anymore when using multiple node pools (#434).
    - Liveness & readiness probes lowest values are now 0 instead of 1 (#442).

    - Correct error code now returned on failed deployment validation (#441).
    - Empty *uri* param is not longer validated for *ecr* connection type (#440).
    - Return correct error when missed *uri* param passed for *git* connection type (#436).
    - Return correct error when user has insufficient privileges (#444).
    - Default branch is now taken for VCS connection if it's not provided by user (#148).
- UI:
    - Auto-generated predictor value doesn't show warning on deploy creation (#80).
    - Default deploy liveness & readiness delays are unified with server values (#74).
    - Deployment doesn't raise error when valid predictor value passed (#46).
    - Sorting for some columns fixed (#48).
    - Secrets are now masked on review stage of connection creation (#42).
    - Interface is now works as expected with long fields on edit connection page (#65)

Odahu 1.3.0, 7 October 2020

Features:

- Core:
    - Persistence Agent added to synchronize k8s CRDS into main storage (#268).
    - All secrets passed to ODAHU API now should be base64 encoded. Decrypted secrets retrieved from ODAHU API via */connection/:id/decrypted* are now also base64 encoded. (#181, #308).
    - Positive and negative (for 404 & 409 status codes) API tests via odahuflow SDK added (#247).

Updates:

- Core:
    - Robot tests will now output pods state after each API call to simplify debugging.

Bug Fixes:

- Core:
    - Refactoring: some abstractions & components were renamed and moved to separate packages to facilitate future development.
    - For connection create/update operations ODAHU API will mask secrets in response body.
    - Rclone output will not reveal secrets on unit test setup stage anymore.
    - *Output-dir* option path is now absolute (#208).
    - Respect *artifactNameTemplate* for local training result directory name (#193).
    - Allow to pass Azure BLOB URI without schema on connection creation (#345)
    - Validate model deployment ID to ensure it starts with alphabetic character (#294)
- UI:
    - State of resources now updates correctly after changing in UI (#11).
    - User aren't able to submit training when resource request is bigger than limit '(#355).
    - Mask secrets on review page during conenction creation process (#42)
    - UI now responds correct in case of concurrent deletion of entities (#44).
    - Additional validation added to prevent creation of resources with unsupported names (#342, #34).
    - Sorting added for training & packaging views (#13, #48).
    - *reference* field become optional for VCS connection (#50).
    - Git connection hint fixed (#7).
- CLI:
    - Configuration secrets is now masked in config output (#307).
    - Local model output path will now display correctly (#371).
    - Local training output will now print only local training results (#370).
    - Help message fixed for *odahuflowctl gppi* command (#375).
- SDK:
    - All API connection errors now should be correctly handled and retried.

Odahu 1.2.0, 21 August 2020

Features:

- Core:
  - PostgreSQL became main database backend as part of increasing project maturity (#175). You can find additional documentation in instructions.
- ODAHU CLI:
  - Option *–ignore-if-exist* added for entities creation (#199).
  - Descriptions updated for commands & options (#160, #197, #209).
- ODAHU UI:
  - ODAHU UI turned into open-source software and now available on github under Apache License Version 2.0. UDAHU UI is an WEB-interface for ODAHU based on React and TypeScript. It provides ODAHU workflows overview and controls, log browsing and entity management.

Updates:

- Knative updated to version 0.15.0. That makes it possible to deploy model services to different node pools (#123).
- Go dependencies was globally updated to migrate from GOPATH to go modules (#32).

Bug Fixes:

- Core:
  - Training now will fail if wrong data path or unexisted storage bucket name is provided (#229).
  - Training log streaming is now working on log view when using native log viewer (#234).
  - ODAHU pods now redeploying during helm chart upgrade (#111).
  - ODAHU docker connection now can be created with blank username & password to install from docker public repo (#184).
- ODAHU CLI:
  - Return training artifacts list sorted by name (#165).
  - Don’t output logs for bulk command (#200).
  - Fix *local pack cleanup-containers* command (#204).
  - Return correct message if entity not found (#210).
  - Return correct message if no options provided (#211).

- ODAHU UI:
    - Fix description of replicas of Model Deployment.
    - Trim spaces for input values.
    - Fix incorrect selection of VCS connection.
    - Close 'ODAHU components' menu after opening link in it.

Odahu 1.1.0, 16 March 2020

New Features:

- Jupyterhub:

  Supported the JupyterHub in our deployment scripts. JupyterHub allows spawning multiple instances of the JupyterLab server. By default, we provide the prebuilt ODAHU JupyterLab plugin in the following Docker images: base-notebook, datascience-notebook, and tensorflow-notebook. To build a custom image, you can use our Docker image template or follow the instructions.

- GPU:

  Added the ability to deploy a model training on GPU nodes. You can find an example of training here. This is one of the official MLFlow examples that classifies flower species from photos.

- Secuirty:

  We integrated our WEB API services with Open Policy Agent that flexibly allows managing ODAHU RBAC. Using Istio, we forbid non-authorize access to our services. You can find the ODAHU security documentation here.

- Vault:

  ODAHU-Flow has the Connection API that allows managing credentials from Git repositories, cloud storage, docker registries, and so on. The default backend for Connection API is Kubernetes. We integrated the Vault as a storage backend for the backend for Connection API to manage your credentials securely.

- Helm 3:

  We migrated our Helm charts to the Helm 3 version. The main goals were to simplify a deployment process to an Openshift and to get rid of the tiller.

- ODAHU UI:

  ODAHU UI provides a user interface for the ODAHU components in a browser. It allows you to manage and view ODAHU Connections, Trainings, Deployments, and so on.

- Local training and packaging:

You can train and package an ML model with the *odahuflowctl* utility using the same ODAHU manifests, as you use for the cluster training and packaging. The whole process is described here.

- Cache for training and packaging:

  ODAHU Flow downloads your dependencies on every model training and packaging launch. To avoid this, you can provide a prebuilt Docker image with dependencies. Read more for model training and packagings.

- Performance improvement training and packaging:

  We fixed multiple performance issues to speed up the training and packaging processes. For our model examples, the duration of training and packaging was reduced by 30%.

- Documentation improvement:

  We conducted a hard work to improve the documentation. For example, the following new sections were added: Security, Installation, Training, Packager, and Model Deployment.

- Odahu-infra:

  We created the new odahu-infra Git repository, where we placed the following infra custom helm charts: Fluentd, Knative, monitoring, Open Policy Agent, Tekton.

- Preemptible nodes:

  Preemptible nodes are priced lower than standard virtual machines of the same types. But they provide no availability guarantees. We added new deployment options to allow training and packaging pods to be deployed on preemptible nodes.

- Third-parties updates:

  - Istio
  - Grafana
  - Prometheus
  - MLFlow
  - Terraform
  - Buildah
  - Kubernetes

Misc/Internal

- Google Cloud Registry:

  We have experienced multiple problems while using Nexus as a main dev Docker registry. This migration also brings us additional advantages, such as in-depth vulnerability scanning.

- Terragrunt:

We switched to using Terragrunt for our deployment scripts. That allows reducing the complexity of our terraform modules and deployment scripts.

Development

Pre requirements

To participate in developing of Odahu project you have to meet these requirements:

- Linux/macOS operating systems (due to tools used for development)
- Python:
  - Python v3.6
  - Pipenv
- Golang:
  - Golang v1.14
  - Dep
  - golangci-lint
  - Kubebuilder
  - swag
  - gotestsum
- JupyterLab plugin:
  - Typescript
  - Yarn
- Infra:
  - HELM v3.2.4
  - Kubectl v1.16.10
  - Docker v17+
  - Swagger codegen 2.4.7

Useful links

- Python:
  - Mlflow
  - Robot Framework Guide
  - Odahu Airflow Plugins Development
- Golang:
  - Kubebuilder
- JupyterLab plugin:
  - Typescript handbook

    - React documentation
    - JupyterLab plugin Extension Developer Guide
- Infra:
    - Helm

Repositories

A repository directory structure

- `containers` - docker files
- `helms` - core helm chart
- `packages` - source code of packages and applications.
- `scripts` - utility scripts for CI/CD process.

odahu/odahu-flow

Core services of Odahu-flow project.

- `odahu-flow-cli` python package
- `odahu-flow-sdk` python package
- E2E Odahu tests
- Training, packaging and deployment operator
- API server
- Feedback services

odahu/odahu-trainer

Collection of training extensions:

- mlflow

odahu/odahu-packager

Collection of model packagers:

- docker-rest
- docker-cli

odahu/odahu-flow-jupyterlab-plugin

The jupyterlab-plugin that provides UI for Odahu-flow API service.

odahu/odahu-airflow-plugin

An apache airflow plugin for the Odahu Platform.

odahu/odahu-docs

The repository contains Odahu documentation, which is available here.

odahu/odahu-examples

Examples of ML models.

odahu/odahu-infra

Docker images and deployments script for third-party services.

Development hints

Set up a development environment

Odahu product contains 5 main development parts:

- Python packages
  - Executes the `make install-all` command to downloads all dependencies and install Odahu python packages.
  - Verifies that the command finished successfully, for example: `odahuflowctl --version`
  - Main entrypoints:
    - Odahu-flow SDK - `packages/sdk`
    - Odahu-flow CLI - `packages/cli`
- Odahu-flow JupyterLab plugin
  - Workdir is `odahu/jupyterlab-plugin`
  - Executes the `yarn install` command to download all JavaScript dependencies.
  - Executes the `npm run build && jupyter labextension install` command to build the JupyterLab plugin.
  - Starts the JyputerLab server using `jupyter lab` command.
- Golang services:
  - Executes the `dep ensure` command in the `packages/operator` directory to downloads all dependencies.
  - Executes the `make build-all` command in the `packages/operator` to build all Golang services.
  - Main entrypoints:
    - API Gateway service - `packages/operator/cmd/edi/main.go`
    - Kubernetes operator - `packages/operator/cmd/operator/main.go`
    - AI Trainer - `packages/operator/cmd/trainer/main.go`
    - AI Packager - `packages/operator/cmd/packager/main.go`
    - Service catalog - `packages/operator/cmd/service_catalog/main.go`
- Odahu-flow Mlflow integration
  - Executes the `pip install -e .` command in the `odahu-flow-mlflow` repository.
- Odahu-flow Airflow plugin
  - Executes the `pip install -e .` command in the `odahu-flow-airflow-plugins` repository.

Update dependencies

- `Python`. Update dependencies in a `Pipfile`. Execute `make update-python-deps` command.
- `Golang`. Update dependencies in a `go.mod`. Execute `go build ./...` command in `packages/operator` directory.
- `Typescript`. Odahu-flow uses the `yarn` to manipulate the typescript dependencies.

Make changes in API entities

All API entities are located in `packages/operator/pkg/api` directory.

To generate swagger documentation execute `make generate-all` in `packages/operator` directory. Important for Mac users: Makefile uses GNU `sed` tool, but MacOS uses BSD `sed` by default. They are not fully compatible. So you need install and use GNU `sed` on your Mac for using Makefile.

After previous action you can update python and typescript clients using the following command: `make generate-clients`.

Actions before a pull request

Make sure you have done the following actions before a pull request:

- for python packages:
  - `make unittest` - Run the python unit tests.
  - `make lint` - Run the python linters.
- for golang services in the `packages/operator` directory:
  - `make test` - Run the golang unit tests.
  - `make lint` - Run the golang linters.
  - `make build-all` - Compile all golang Odahu-flow services
- for typescript code in the `packages/jupyterlab-plugin` directory:
  - `yarn lint` - Run the typescript linter.
  - `jlpm run build` - Compile the jupyterlab plugin.

Local Helm deploy

During development, you often have to change the helm chart, to test the changes you can use the following command quickly: `make helm-install`.

Optionally, you can create the variables helm file and specify it using the `HELM_ADDITIONAL_PARAMS` Makefile option. You always can download real variables file from a Terraform state.

Integration Testing

This page provides information about testing of ODAHU. ODAHU uses Robot Framework for an integration, system and end-to-end testings.

All tests are located in the following directories of the ODAHU project:

- `packages/robot/` - a python package with additional Robot libraries. For example: kubernetes, auth_client, feedback, and so on.
- `packages/tests/stuff/` - setup, cleanup scripts and artifacts for integration testing. For example: pre-trained ML artifacts, test toolchain integrations, and so on.
- `packages/tests/e2e/` - directory with the Robot Framework tests.

Preparing for testing

It's expected that you are using a Unix-like operating system and have installed conda (4.10+), preferably miniconda.

Clone ODAHU project from git repository and proceed to main dir – `odahu-flow`.

Create Conda virtual environment with python version 3.6+.

Update and/or install **pip** and **setuptools**:

```
$ pip install -U pip setuptools
```

Proceed to the `odahu-flow` main directory where the `Makefile` is located and run **make** command:

```
/odahu-flow$ make install-all
```

Check that odahuflowctl works:

```
/odahu-flow$ odahuflowctl
```

Also, you should have installed jq and rclone packages.

Running tests

We set up robot tests for `gke-odahu-flow-test` cluster in the example below.

NB. Do not forget change **your cluster url** and **odahu-flow version**.

By default put `cluster_profile.json` file in `odahu-flow/.secrets/` folder (by default) or you can specify another default name of file or directory in *'Makefile'* in parameters: `SECRET_DIR` and `CLUSTER_PROFILE`.

You can optionally override the following parameters in `.env` file (which by default are taken from `Makefile`).

- `CLUSTER_NAME`
- `ROBOT_OPTIONS`
- `ROBOT_FILES`
- `HIERA_KEYS_DIR`
- `SECRET_DIR`
- `CLOUD_PROVIDER`
- `DOCKER_REGISTRY`
- `EXPORT_HIERA_DOCKER_IMAGE`
- `ODAHUFLOW_PROFILES_DIR`
- `ODAHUFLOW_VERSION`, etc.

For that, you should create `.env` file in the main dir of the project (`odahu-flow`).

In our example, we will override the parameters of `Makefile` in `.env` file:

```
# Cluster name
CLUSTER_NAME=gke-odahu-flow-test
# Optionally, you can provide RobotFramework settings below.
# Additional robot parameters. For example, you can specify tags or variables.
ROBOT_OPTIONS=-e disable
# Robot files
ROBOT_FILES=**/*.robot
# Cloud which will be used
CLOUD_PROVIDER=gcp
# Docker registry
DOCKER_REGISTRY=gcr.io/or2-msq-<myprojectid>-t1iylu/odahu
# Version of odahu-flow
ODAHUFLOW_VERSION=1.1.0-rc8
```

Afterwards, you should prepare an Odahu cluster for Robot Framework tests by using the command:

```
/odahu-flow$ make setup-e2e-robot
```

NB. You should execute the `setup` command only once for a new cluster.

The next step is to run the Robot Framework tests:

```
/odahu-flow$ make e2e-robot
```

Finally, cleanup the cluster after testing:

```
/odahu-flow$ make cleanup-e2e-robot
```

NB. You should run the `cleanup` command only once, after all testing has been completed.

Indices and tables

Index