0.11.0

Getting Started

Introduction

Welcome to ZenML!

Are you an ML engineer or data scientist shipping models to production and jumbling a plethora of tools? Do you struggle with versioning data, code, and models in your projects? Have you had trouble replicating production pipelines and monitoring models in production?

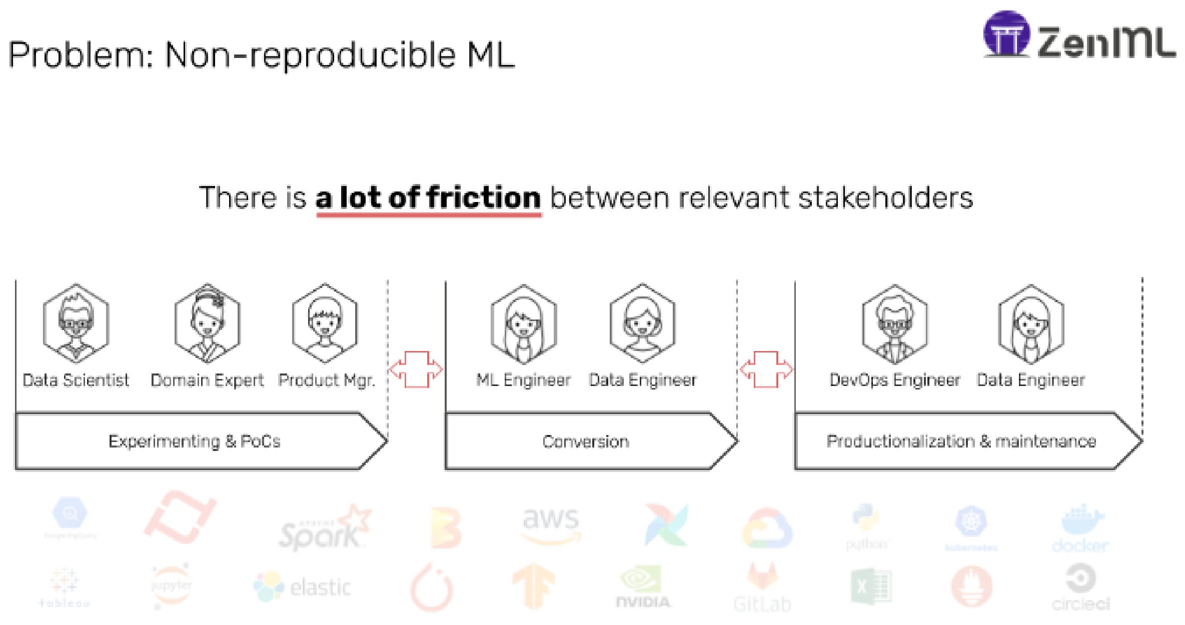
If you answered yes to any, **ZenML** is here to help with all that, and more.

Extensible open-source framework.

ZenML is an extensible, open-source MLOps framework for creating portable, production-ready **MLOps pipelines**. It's built for data scientists, ML Engineers, and MLOps Developers to collaborate as they develop to production. **ZenML** has simple, flexible syntax, is **cloud-** and **tool-agnostic**, and has interfaces/abstractions that are catered towards ML workflows. ZenML brings together all your favorite tools in one place so you can tailor your workflow to cater your needs.

Why ZenML

Everyone loves to train ML models, but few talks about shipping them into production, and even fewer can do it well. It's no wonder that 85% of ML models fail and 53% don't make it into production, according to a Gartner survey.

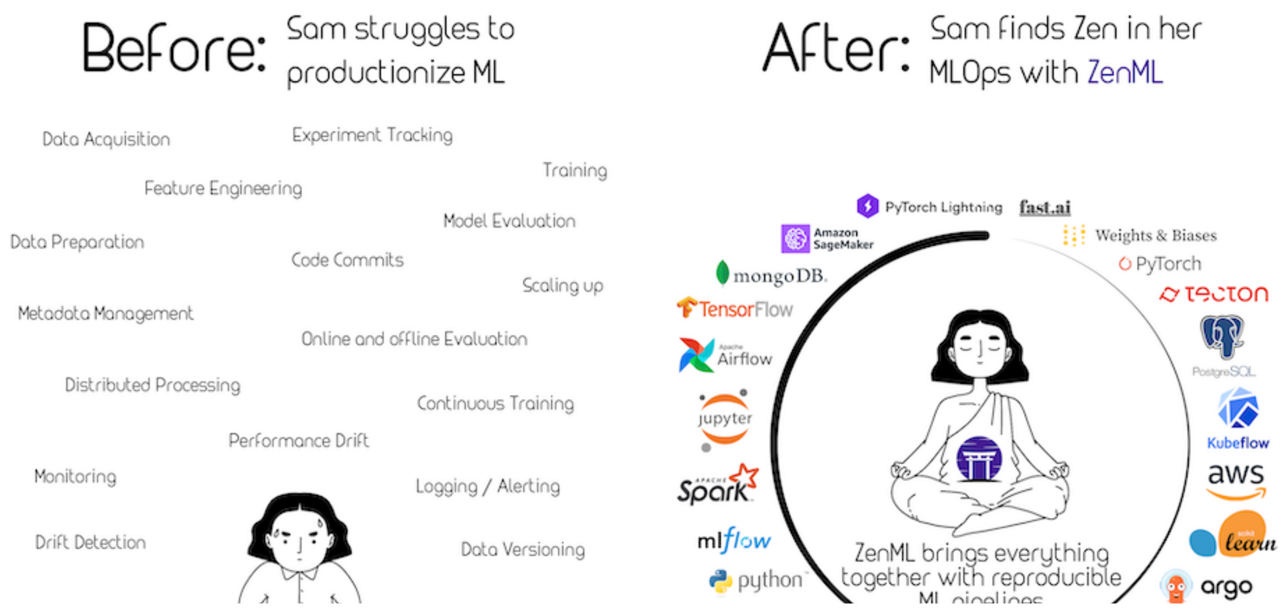


The long journey from experimentation to production.

At ZenML, we believe the journey from model development to production doesn't need to be long and meandering.

With ZenML, you as a data scientist can concentrate on what you do best - developing ML models, and not worry about infrastructure or deployment tools. If you come from unstructured notebooks or scripts with lots of manual processes, ZenML will make the path to production easier and faster for you and your team. Using ZenML allows data scientists like you to own the entire pipeline - from experimentation to production.

This is why we built ZenML.



ZenML unifies all your tools in one place.

Eager to get started? We recommend the following to start using **ZenML** right away!

Pipelining - Take Your First ZenML Steps

New to ZenML? Get acquainted with the core concepts first to understand what makes ZenML so special. Next, jump right into developer guide to go from zero to hero in no time.

Stacking - Connect Your Favorite Tools

Already ran your first pipeline and want to know about integrations and production use cases? Check out ZenML integrations. You'll find detailed descriptions of specific use cases and features. Learn how to take your MLOps stack from basic to fully-fledged.

Extending - Make It Your Own

ZenML doesn't support your favorite tool? Worry not! ZenML is built from the ground up with extensibility in mind. Find out how to integrate other tools or proprietary on-prem solutions for you and your team in Stack Components.

Collaborating - Work Together With Your Team

Share not only your code but also your ZenML stacks with your team. Find out how in collaboration.

Resources - Learn ZenML with Tutorials, Examples, and Guides

The ZenML team and community have put together resources to learn about the framework. Learn more in resources.

Community - Be Part of the ZenML Family

Can't really find what you want and need help? Join our Slack group. Ask questions about bugs or specific use cases and get a quick response from the ZenML core team.

Core Concepts

What are the core concepts in ZenML

ZenML consists of a few components. This guide walks through the various pieces you'll encounter, starting from the basics to things you'll only encounter when deploying to the cloud.

Here's a high-level overview of a typical ZenML workflow:

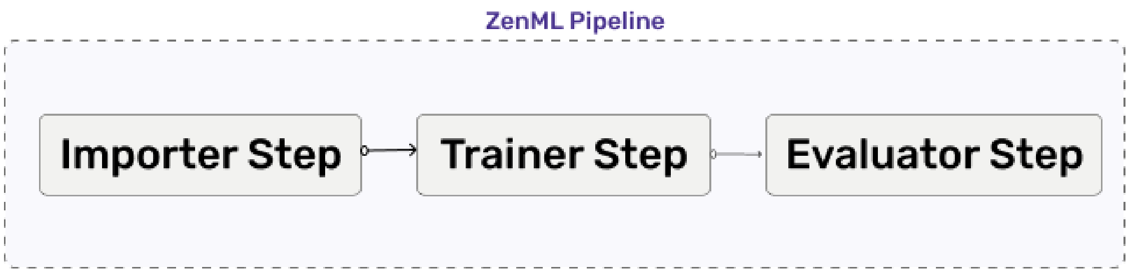
1. Writing a *pipeline* to define what happens in your machine learning workflow.
2. Configuring a ZenML *stack*.
3. Switching between *stacks* depending on needs.
4. Customizing your *stack* with different *components*.

So first, the basics.

Pipelines and Steps

At its core, ZenML follows a pipeline-based workflow for your data projects. A **pipeline** consist of a series of **steps**, organized in any order that makes sense for your use case.

Below, you can see three **steps** running one after another in a **pipeline**.



The most basic ZenML pipeline

The steps might have dependencies between them. For example, a step might use the outputs from a previous step and thus must wait until the previous step completes before starting. This is something you can keep in mind when organizing your steps.

You can have multiple pipelines for different purposes. For example, a training pipeline to train and evaluate models and an inference pipeline to serve the model.

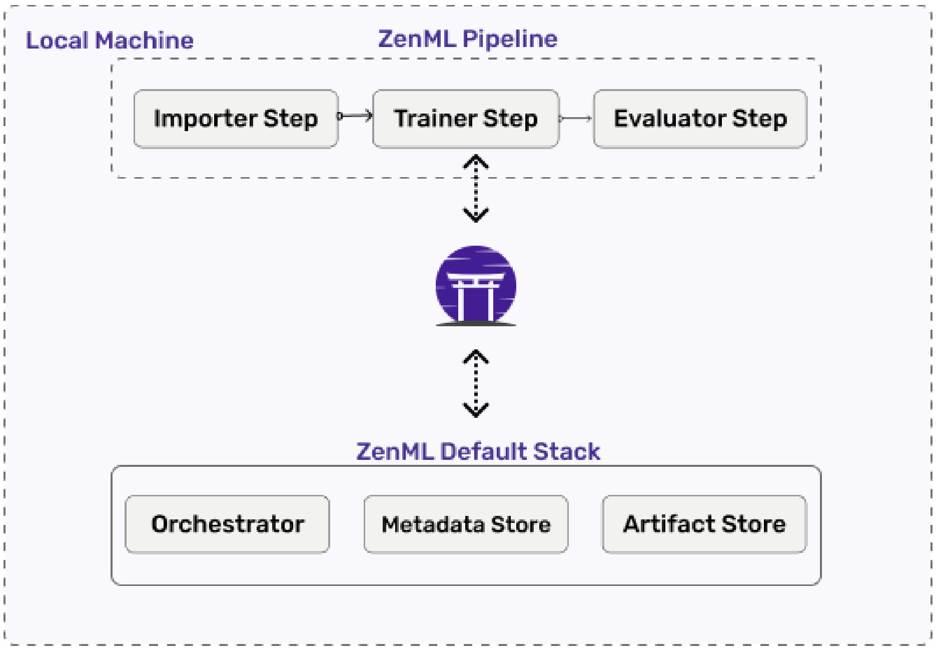
Pipelines and steps are defined in code using Python *decorators* or *classes*. This is where the core business logic and value of your work lives, and you will spend most of your time defining these two things. Your code lives inside a Repository, which is the main abstraction within which your project-specific pipelines should live.

When it comes to running your pipeline, ZenML offers an abstraction to handle how your pipeline gets run. This is where **Stacks** come into play.

Stacks, Components and Stores

A **Stack** is the configuration of the underlying infrastructure and choices around how your pipeline will be run. For example, you can choose to run your pipeline locally or on the cloud by changing the stack you use.

ZenML comes with a default stack that runs locally, as seen in the following diagram:



ZenML pipelines run on stacks

In any Stack, there **must** be at least three basic **Stack Components** -

1. Orchestrator.
2. Artifact Store.
3. Metadata Store.

Orchestrator

An **Orchestrator** is the workhorse that coordinates all the steps to run in a pipeline.

Since pipelines can be set up with complex combinations of steps with various asynchronous dependencies between them, the Orchestrator is the component that decides what steps to run, when, and how to pass data between the steps.

ZenML comes with a built-in *local orchestrator* designed to run on your local machine. This is useful especially during the exploration phase of your project. You don't have to rent a cloud instance just to try out basic things.

Once the pipeline is established you can switch to a full-fledged cloud stack that uses more sophisticated orchestrators like the Airflow or Kubeflow orchestrator. See the list of all orchestrators here.

Artifact Store

An **Artifact Store** is a component that houses all data that pass through the pipeline. Data in the artifact store are called *artifacts*.

These artifacts may have been produced by the pipeline steps, or they may be the data ingested into a pipeline via an importer step. The artifact store houses all intermediary pipeline step results, which in turn will be tracked in the metadata store.

The fact that all your data inputs and outputs are tracked and versioned in the artifact store allows for extremely useful features like data caching which speeds up your workflow.

Metadata Store

A **Metadata Store** keeps track of all the bits of extraneous data from a pipeline run. It allows you to fetch specific steps from your pipeline run and their output artifacts in a post-execution workflow.

With a metadata store, you are able to access all of your previous experiments with the associated details. This is extremely helpful in troubleshooting.

When you start working with ZenML, you'll likely spend most of your time here, working with the default stack on initialization.

Other Stack Components

We've covered the three basic stack components that you will encounter most frequently. They work well on a local machine, but is rarely enough in production.

At some point, you might want to scale up your stack to run elsewhere, for example on a cloud with powerful GPUs for training or CPU's for deployment.

ZenML provides many other stack components to suit these use cases. Having these components in your stack supercharges your pipeline for production.

For other stack components check out this page.

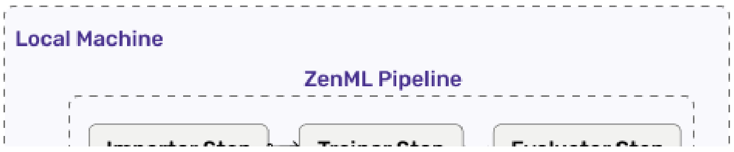
Switching Stacks to Scale Up

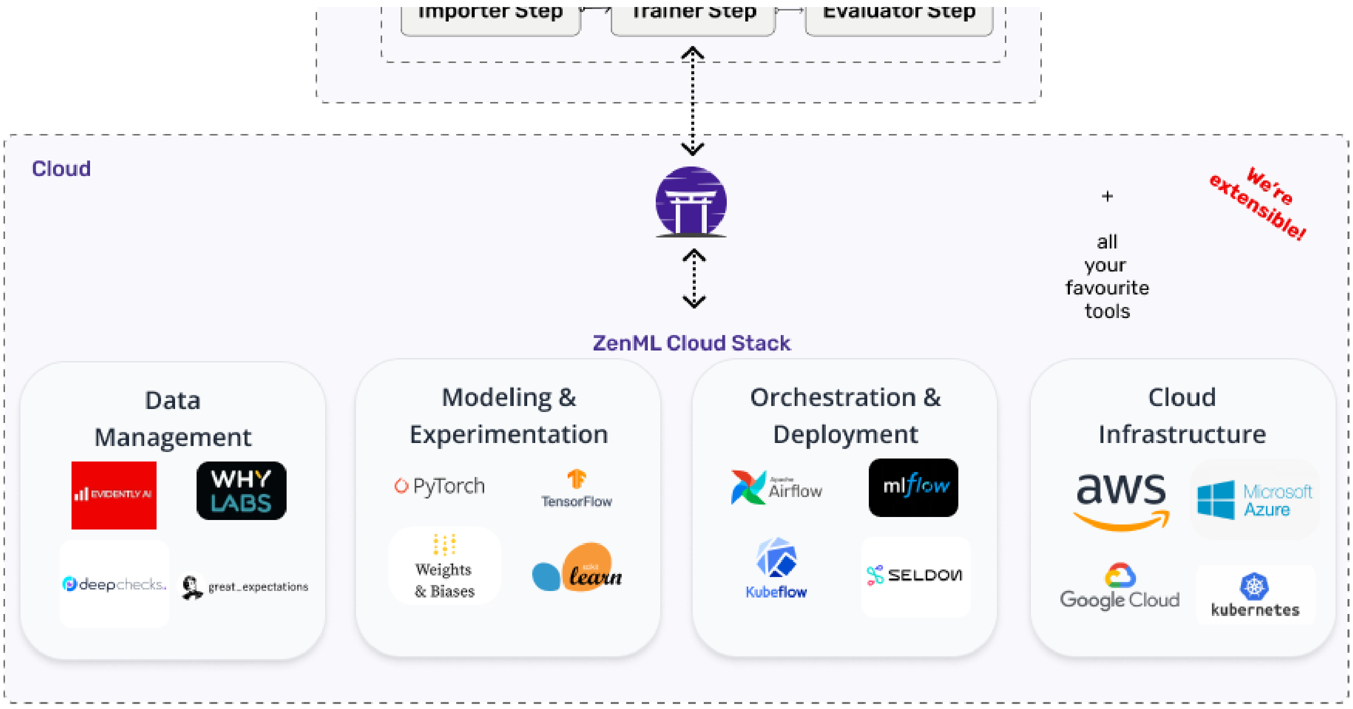
We've seen how to run a pipeline locally. But that is rarely enough in production machine learning which typically involves cloud infrastructure.

What's really cool with using ZenML is you can easily switch your stack from running on a local machine to running on the cloud with a single CLI command.

The rest of the code defining your steps and pipelines stays the same, whether it's running on a local machine or a cloud infrastructure of your choice. The only change is in the stack and its components.

Below is an illustration showing how the same pipeline on a local machine can be scaled up to run on a full-fledged cloud infrastructure by switching stacks. You get all the benefits of using cloud infrastructures with minimal changes in your code.





Running your pipeline in the cloud

Cloud Training, Deployment and Monitoring

Running workflows in the cloud often requires certain custom behaviors, so ZenML offers a number of extra Stack Components that handle these common use cases. For example, it's common to want to deploy models so we have a Model Deployer component. Similarly, you might want to use popular tools like Weights & Biases or MLflow to track your experiments, so we have an Experiment Tracker stack component. Any additional software needed for these components can be added and installed by using ZenML's Integration installer.

It is this modular and configurable nature of the ZenML stack that offers you ways to get productive quickly. If we don't support some specific tool you want to use, our stack components are extensible and well documented so this shouldn't be a barrier for you.

All the stack components configured as part of the stack carry their configuration parameters so whether it's an AWS SageMaker cluster you need to run your training on or an Google Cloud container registry you need to connect to, ZenML handles the connections between these various parts on your behalf.

Other Bits and Pieces

There are lots of different ways to use ZenML which will depend on your precise use case. The following concepts and stack components are things you'll possibly encounter further down the road while using ZenML.

- **Materializers** - ZenML stores the data inputs and outputs to your steps in the Artifact Store as we saw above. In order to store the data, it needs to serialize everything in a format that can fit into the Artifact

Store. ZenML handles serialization (and deserialization) of the most common artifacts, but if you try to do something we haven't already thought of you'll need to write your own custom materializer. This isn't hard, but you should be aware that it's something you might need do to. The ZenML CLI will let you know with a clear error message when you need to do this.

- **Profiles** - Profiles are groupings of stacks. You might want to keep all your AWS stacks separate from your GCP stacks, for example, or your work infrastructure use separate from that which you use for your personal projects. Profiles allow you to separate these out, and switching between them is effortless.
- **Service** - A service is a longer-lived entity that extends the capabilities of ZenML beyond the run of a pipeline. For example, a service could be a prediction service that loads models for inference in a production setting.
- **ZenServer** - ZenML is building out functionality to host a shared server that allows teams to collaborate and share stacks, data stores and more.

There's a lot more detail to digest when it comes to ZenML, but with the above you should be sufficiently armed to understand how the framework works and where you might want to extend it.

Installation

How to install ZenML

ZenML is a Python package that can be installed directly via `pip`:

```
pip install zenml
```

Please note that ZenML currently only supports Python 3.7, 3.8, and 3.9. Please adjust your Python environment accordingly.

Virtual Environments

We highly encourage you to install ZenML in a virtual environment. At ZenML, We like to use virtualenvwrapper or pyenv-virtualenv to manage our Python virtual environments.

As mentioned above, make sure that your virtual environment uses one of the supported Python versions.

Verifying Installations

Once the installation is completed, you can check whether the installation was successful through:

Bash

```
zenml version
```

Python

```python
import zenml
print(zenml.__version__)
```

If you would like to learn more about the current release, please visit our PyPi package page.

Known installation issues for M1 Mac Users

If you have a M1 Mac machine and you are encountering an error while trying to install ZenML, please try to setup `brew` and `pyenv` with Rosetta 2 and then install ZenML. The issue arises because some of the dependencies aren't fully compatible with the vanilla ARM64 Architecture. The following links may be helpful (Thank you Reid Falconer!):

- Pyenv with Apple Silicon
- Install Python Under Rosetta 2

Running with Docker

`zenml` is also available as a Docker image hosted publicly on DockerHub. Use the following command to get started in a bash environment with `zenml` available:

```
docker run -it zenmldocker/zenml /bin/bash
```

Installing Develop

If you want to use the bleeding edge of ZenML that has not even been released yet, you can install our `develop` branch directly.

Installing develop is mainly useful if there are key features or bug fixes that you urgently need so you can get those immediately and do not have to wait for the next release.

> As the name suggests, the new features in the `develop` branch are still under development and might not be as polished as the final released version.

Use at your own risk; no guarantees given!

```
pip install git+https://github.com/zenml-io/zenml.git@develop --upgrade
```

Using develop with Remote Orchestrators

Remote orchestrators like KubeFlow require Docker Images to set up the environmens of each step. By default, they use the official ZenML docker image that we provide with each release. However, if you install from develop, this image will be outdated, so you need to build a custom image instead, and specify it in the configuration of your orchestrator accordingly (see the MLOps Stacks Orchestrator page of your specific orchestrator flavor for more details on how this can be done).

Building Custom Docker Images

To build a custom image, you first need to install Docker, then run one of the following commands from the ZenML repo root depending on your operating system:

Linux, MacOS (Intel), Windows

```
docker build -t <IMAGE_NAME> -f docker/local-dev.Dockerfile .
```

MacOS (M1)

```
docker build --platform linux/amd64 -t <IMAGE_NAME> -f docker/local-dev.Dockerfile .
```

Examples & Use-Cases

Developer Guide

Steps & Pipelines

How to create ML pipelines in ZenML

ZenML helps you standardize your ML workflows as ML **Pipelines** consisting of decoupled, modular **Steps**. This enables you to write portable code that can be moved from experimentation to production in seconds.

If you are new to MLOps and would like to learn more about ML pipelines in general, checkout ZenBytes, our lesson series on practical MLOps, where we introduce ML pipelines in more detail in ZenBytes lesson 1.1.

Step

Steps are the atomic components of a ZenML pipeline. Each step is defined by its inputs, the logic it applies and its outputs. Here is a very basic example of such a step, which uses a utility function to load the Digits dataset:

```python
import numpy as np

from zenml.integrations.sklearn.helpers.digits import get_digits
from zenml.steps import Output, step


@step
def load_digits() -> Output(
    X_train=np.ndarray, X_test=np.ndarray, y_train=np.ndarray, y_test=np.ndarray
):
    """Loads the digits dataset as normal numpy arrays."""
    X_train, X_test, y_train, y_test = get_digits()
    return X_train, X_test, y_train, y_test
```

As this step has multiple outputs, we need to use the `zenml.steps.step_output.Output` class to indicate the names of each output. These names can be used to directly access the outputs of steps after running a pipeline, as we will see in a later chapter.

Let's come up with a second step that consumes the output of our first step and performs some sort of transformation on it. In this case, let's train a support vector machine classifier on the training data using sklearn:

```python
import numpy as np
from sklearn.base import ClassifierMixin
from sklearn.svm import SVC

from zenml.steps import step


@step

def svc_trainer(
    X_train: np.ndarray,
    y_train: np.ndarray,
) -> ClassifierMixin:
    """Train a sklearn SVC classifier."""
    model = SVC(gamma=0.001)
    model.fit(X_train, y_train)
    return model
```

Next, we will combine our two steps into our first ML pipeline.

In case you want to run the step function outside the context of a ZenML pipeline, all you need to do is call the `.entrypoint()` method with the same input signature. For example:

```
svc_trainer.entrypoint(X_train=..., y_train=...)
```

Pipeline

Let us now define our first ML pipeline. This is agnostic of the implementation and can be done by routing outputs through the steps within the pipeline. You can think of this as a recipe for how we want data to flow through our steps.

```python
from zenml.pipelines import pipeline


@pipeline
def first_pipeline(step_1, step_2):
    X_train, X_test, y_train, y_test = step_1()
    step_2(X_train, y_train)
```

Instantiate and run your Pipeline

With your pipeline recipe in hand you can now specify which concrete step implementations to use when instantiating the pipeline:

```python
first_pipeline_instance = first_pipeline(
    step_1=load_digits(),
    step_2=svc_trainer(),
)
```

Currently, you cannot use the same step twice in a pipeline because step names must be unique. If you would like to reuse a step, use the `clone_step()` utility function to create a copy of the step with a new name.

You can then execute your pipeline instance with the `.run()` method:

```python
first_pipeline_instance.run()
```

You should see the following output in your terminal:

```
Creating run for pipeline: `first_pipeline`
```

```
Cache disabled for pipeline `first_pipeline`
Using stack `default` to run pipeline `first_pipeline`
Step `load_digits` has started.
Step `load_digits` has finished in 0.049s.
Step `svc_trainer` has started.
Step `svc_trainer` has finished in 0.067s.
Pipeline run `first_pipeline-06_Jul_22-16_10_46_255748` has finished in 0.128s.
```

We will dive deeper into how to inspect the finished run within the chapter on Accessing Pipeline Runs.

Give each pipeline run a name

When running a pipeline by calling `my_pipeline.run()`, ZenML uses the current date and time as the name for the pipeline run. In order to change the name for a run, pass `run_name` as a parameter to the `run()` function:

```
first_pipeline_instance.run(run_name="custom_pipeline_run_name")
```

> Pipeline run names must be unique, so make sure to compute it dynamically if you plan to run your pipeline multiple times.

Code Summary

Code Example for this Section

```python
import numpy as np
from sklearn.base import ClassifierMixin
from sklearn.svm import SVC

from zenml.integrations.sklearn.helpers.digits import get_digits
from zenml.steps import Output, step
from zenml.pipelines import pipeline


@step
def load_digits() -> Output(
    X_train=np.ndarray, X_test=np.ndarray, y_train=np.ndarray, y_test=np.ndarray
):
    """Loads the digits dataset as normal numpy arrays."""
    X_train, X_test, y_train, y_test = get_digits()
    return X_train, X_test, y_train, y_test


@step
```

```python
def svc_trainer(
    X_train: np.ndarray,
    y_train: np.ndarray,
) -> ClassifierMixin:
    """Train a sklearn SVC classifier."""
    model = SVC(gamma=0.001)
    model.fit(X_train, y_train)
    return model


@pipeline
def first_pipeline(step_1, step_2):
    X_train, X_test, y_train, y_test = step_1()
    step_2(X_train, y_train)


first_pipeline_instance = first_pipeline(
    step_1=load_digits(),
    step_2=svc_trainer(),
)

first_pipeline_instance.run()
```

Configure Pipelines at Runtime

How to configure step and pipeline parameters for each run

A ZenML pipeline clearly separates business logic from parameter configuration. Business logic is what defines a step or a pipeline. Parameter configurations are used to dynamically set parameters of your steps and pipelines at runtime.

You can configure your pipelines at runtime in the following ways:

- Configuring from within code: Do this when you are quickly iterating on your code and don't want to change your actual step code. This is useful in the development phase.
- Configuring with YAML config files: Do this when you want to launch pipeline runs without modifying the code at all. This is the recommended way for production scenarios.

Configuring from within code

You can add a configuration to a step by creating your configuration as a subclass of the `BaseStepConfig`. When such a config object is passed to a step, it is not treated like other artifacts. Instead, it gets passed into the step when the pipeline is instantiated.

```python
import numpy as np
from sklearn.base import ClassifierMixin
from sklearn.svm import SVC

from zenml.steps import step, BaseStepConfig


class SVCTrainerStepConfig(BaseStepConfig):
    """Trainer params"""
    gamma: float = 0.001


@step
def svc_trainer(
    config: SVCTrainerStepConfig,
    X_train: np.ndarray,
    y_train: np.ndarray,
) -> ClassifierMixin:
    """Train a sklearn SVC classifier."""
    model = SVC(gamma=config.gamma)
    model.fit(X_train, y_train)
    return model
```

The default value for the `gamma` parameter is set to `0.001`. However, when the pipeline is instantiated you can override the default like this:

```python
first_pipeline_instance = first_pipeline(
    step_1=load_digits(),
    step_2=svc_trainer(SVCTrainerStepConfig(gamma=0.01)),
)

first_pipeline_instance.run()
```

Behind the scenes, `BaseStepConfig` is implemented as a Pydantic BaseModel. Therefore, any type that Pydantic supports is also supported as an attribute type in the `BaseStepConfig`.

Configuring with YAML config files

For production scenarios where you want to launch pipeline runs without modifying the code at all, you can also configure your pipeline runs using YAML config files.

There are two ways how YAML config files can be used:

- Defining step parameters in YAML and setting the path to the config with `pipeline.with_config()` before calling `pipeline.run()`,

- Configuring the entire pipeline at runtime in YAML and executing it with `zenml pipeline run`.

Defining step parameters in YAML

If you only want to configure step parameters as above, you can do so with a minimalistic configuration YAML file, which you use to configure a pipeline before running it using the `with_config()` method, e.g.:

```
first_pipeline_instance = first_pipeline(
    step_1=load_digits(),
    step_2=svc_trainer(),
)

first_pipeline_instance.with_config("path_to_config.yaml").run()
```

The `path_to_config.yaml` needs to have the following structure:

```
steps:
  <STEP_NAME_IN_PIPELINE>:
    parameters:
      <PARAMETER_NAME>: <PARAMETER_VALUE>
      ...
    ...
```

For our example from above, we could use the following configuration file:

```
steps:
  step_2:
    parameters:
      gamma: 0.01
```

Note that `svc_trainer()` still has to be defined to have a `config: SVCTrainerStepConfig` argument. The difference here is only that we provide `gamma` via a config file before running the pipeline, instead of explicitly passing a `SVCTrainerStepConfig` object during the step creation.

Configuring the entire pipeline at runtime in YAML

For production settings, you might want to use config files not only for your parameters, but even for choosing what code gets executed. This way, you can define entire pipeline runs without changing the code.

To run pipelines in this way, you can use the `zenml pipeline run` command with `-c` argument:

```
zenml pipeline run <PATH_TO_PIPELINE_PYTHON_FILE> -c <PATH_TO_CONFIG_YAML_FILE>
```

`<PATH_TO_PIPELINE_PYTHON_FILE>` should point to the Python file where your pipeline function or class is defined. Your steps can also be in that file, but they do not need to. If your steps are defined in separate code files, you can instead specify that in the YAML, as we will see below.

> Do **not** instantiate and run your pipeline within the python file that you want to run using the CLI, else your pipeline will be run twice, possibly with different configurations.

If you want to dynamically configure the entire pipeline, your config file will need a bit more information:

- The name of the function or class of your pipeline in `<PATH_TO_PIPELINE_PYTHON_FILE>`,
- The name of the function or class of each step, optionally with additional path to the the code file where it is defined (if it is not in `<PATH_TO_PIPELINE_PYTHON_FILE>`),
- Optionally, the name of each materializer (about which you will learn later in the section on Materializers).

Overall, the required structure of such a YAML should look like this:

```yaml
name: <PIPELINE_CLASS_OR_FUNCTION_NAME>
steps:
  <STEP_NAME_IN_PIPELINE>:
    source:
      name: <STEP_CLASS_OR_FUNCTION_NAME>
      file: <PATH_TO_STEP_PYTHON_FILE>  # (optional)
    parameters:  # (optional)
      <PARAMETER_NAME>: <PARAMETER_VALUE>
      <SOME_OTHER_PARAMETER>: ...
    materializers:  # (optional)
      <NAME_OF_OUTPUT_OF_STEP>:
        name: <MATERIALIZER_CLASS_NAME>
        file: <PATH_TO_MATERIALIZER_PYTHON_FILE>
      <SOME_OTHER_OUTPUT>:
        ...
  <SOME_OTHER_STEP>:
    ...
```

This might seem daunting at first, so let us go over it one by one:

Defining which pipeline to use

```yaml
name: <PIPELINE_CLASS_OR_FUNCTION_NAME>
```

The first line of the YAML defines which pipeline code to use. In case you defined your pipeline as Python function with `@pipeline` decorator, this name is the name of the decorated function. If you used the Class Based API (which you will learn about in the next section), it will be the name of the class.

For example, if you have defined a pipeline `my_pipeline_a` in `pipelines/my_pipelines.py`, then you would:

- Set `name: my_pipeline_a` in the YAML,
- Use `pipelines/my_pipelines.py` as `<PATH_TO_PIPELINE_PYTHON_FILE>`.

Defining which steps to use

For each step, you can define which source code to use via the `source` field:

```yaml
steps:
  <STEP_NAME_IN_PIPELINE>:
    source:
      name: <STEP_CLASS_OR_FUNCTION_NAME>
      file: <PATH_TO_STEP_PYTHON_FILE>  # (optional)
```

For example, if you have defined a step `my_step_1` in `steps/my_steps.py` that you want to use as `step_1` of your pipeline `my_pipeline_a`, then you would define that in your YAML like this:

```yaml
name: my_pipeline_a
steps:
  step_1:
    source:
      name: my_step_1
      file: steps/my_steps.py
```

If your step is defined in the same file as your pipeline, you can omit the last `file: ...` line.

Defining materializer source codes

The `materializers` field of a step can be used to specify custom materializers of your step outputs and inputs.

Materializers are responsible for saving and loading artifacts within each step. For more details on materializers and how to configure them in YAML config files, see the Materializers section in the list of advanced concepts.

Code Summary

Putting it all together, we can configure our entire example pipeline run like this in the CLI:

CLI Command

```
zenml pipeline run run.py -c config.yaml
```

run.py

```python
import numpy as np
from sklearn.base import ClassifierMixin
from sklearn.svm import SVC

from zenml.integrations.sklearn.helpers.digits import get_digits
from zenml.steps import BaseStepConfig, Output, step
from zenml.pipelines import pipeline


@step
def load_digits() -> Output(
    X_train=np.ndarray, X_test=np.ndarray, y_train=np.ndarray, y_test=np.ndarray
):
    """Loads the digits dataset as normal numpy arrays."""
    X_train, X_test, y_train, y_test = get_digits()
    return X_train, X_test, y_train, y_test


class SVCTrainerStepConfig(BaseStepConfig):
    """Trainer params"""
    gamma: float = 0.001


@step
def svc_trainer(
    config: SVCTrainerStepConfig,
    X_train: np.ndarray,
    y_train: np.ndarray,
) -> ClassifierMixin:
    """Train a sklearn SVC classifier."""
    model = SVC(gamma=config.gamma)
    model.fit(X_train, y_train)
    return model


@pipeline
def first_pipeline(step_1, step_2):
    X_train, X_test, y_train, y_test = step_1()
    step_2(X_train, y_train)
```

config.yaml

```yaml
name: first_pipeline
steps:
  step_1:
    source:
      name: load_digits
  step_2:
    source:
      name: svc_trainer
    parameters:
      gamma: 0.01
```

Equivalent run from python

This is what the same pipeline run would look like if triggered from within python.

```python
first_pipeline_instance = first_pipeline(
    step_1=load_digits(),
    step_2=svc_trainer(SVCTrainerStepConfig(gamma=0.01)),
)

first_pipeline_instance.run()
```

Pro-Tip: You can use this to configure and run your pipeline from within your github action (or comparable tools). This way you ensure each run is directly associated with a code version.

Choose Functional vs. Class-Based API

How to define pipelines and steps with functional and class-based APIs

In ZenML there are two different ways how you can define pipelines or steps. What you have seen in the previous sections is the **Functional API**, where steps and pipelines are defined as Python functions with a `@step` or `@pipeline` decorator respectively. This is the API that is used primarily throughout the ZenML docs and examples.

Alternatively, you can also define steps and pipelines using the **Class-Based API** by creating Python classes that subclass ZenML's abstract base classes `BaseStep` and `BasePipeline` directly. Internally, both APIs will result in similar definitions, so it is entirely up to you which API to use.

In the following, we will compare the two APIs using the code example from the previous section on Runtime Configuration:

Creating Steps

In order to create a step with the class-based API, you will need to create a subclass of `zenml.steps.BaseStep` and implement its `entrypoint()` method to perform the logic of the step.

Class-based API

```python
import numpy as np
from sklearn.base import ClassifierMixin
from sklearn.svm import SVC

from zenml.steps import BaseStep, BaseStepConfig


class SVCTrainerStepConfig(BaseStepConfig):
    """Trainer params"""
    gamma: float = 0.001


class SVCTrainerStep(BaseStep):
    def entrypoint(
        self,
        config: SVCTrainerStepConfig,
        X_train: np.ndarray,
        y_train: np.ndarray,
    ) -> ClassifierMixin:
        """Train a sklearn SVC classifier."""
        model = SVC(gamma=config.gamma)
        model.fit(X_train, y_train)
        return model
```

Functional API

```python
import numpy as np
from sklearn.base import ClassifierMixin
from sklearn.svm import SVC

from zenml.steps import step, BaseStepConfig


class SVCTrainerStepConfig(BaseStepConfig):
    """Trainer params"""
    gamma: float = 0.001


@step
def svc_trainer(
    config: SVCTrainerStepConfig,
    X_train: np.ndarray,
    y_train: np.ndarray,
) -> ClassifierMixin:
    """Train a sklearn SVC classifier."""
    model = SVC(gamma=config.gamma)
    model.fit(X_train, y_train)
    return model
```

Creating Pipelines

Similarly, you can define a pipeline with the class-based API by subclassing `zenml.pipelines.BasePipeline` and implementing its `connect()` method:

Class-based API

```python
from zenml.pipelines import BasePipeline


class FirstPipeline(BasePipeline):
    def connect(self, step_1, step_2):
        X_train, X_test, y_train, y_test = step_1()
        step_2(X_train, y_train)


first_pipeline_instance = FirstPipeline(
    step_1=load_digits(),
    step_2=SVCTrainerStep(SVCTrainerStepConfig(gamma=0.01)),
)

first_pipeline_instance.run()
```

Functional API

```python
from zenml.pipelines import pipeline


@pipeline
def first_pipeline(step_1, step_2):
    X_train, X_test, y_train, y_test = step_1()
    step_2(X_train, y_train)


first_pipeline_instance = first_pipeline(
    step_1=load_digits(),
    step_2=SVCTrainerStep(SVCTrainerStepConfig(gamma=0.01)),
)

first_pipeline_instance.run()
```

As you saw in the example above, you can even mix and match the two APIs. Choose whichever style feels most natural to you!

Advanced Usage: Using decorators to give your steps superpowers

As you will learn later, ZenML has many integrations that allow you to add functionality like automated experiment tracking to your steps. Some of those integrations need to be initialized before they can be used, which ZenML wraps using special `@enable_<INTEGRATION>` decorators. Similar to the functional API, you can do this in the class-based API by adding a decorator to your `BaseStep` subclass:

Class-based API

```python
from zenml.steps import BaseStep
from zenml.integrations.mlflow.mlflow_step_decorator import enable_mlflow


@enable_mlflow
class TFTrainer(BaseStep):
    def entrypoint(
        self,
        x_train: np.ndarray,
        y_train: np.ndarray,
    ) -> tf.keras.Model:
        mlflow.tensorflow.autolog()
        ...
```

Functional API

```python
from zenml.steps import step
from zenml.integrations.mlflow.mlflow_step_decorator import enable_mlflow


@enable_mlflow
@step
def tf_trainer(
    X_train: np.ndarray,
    y_train: np.ndarray,
) -> tf.keras.Model:
    mlflow.tensorflow.autolog()
    ...
```

Check out our MLflow, Wandb and whylogs examples for more information on how to use the specific decorators.

Inspect Pipeline Runs

How to inspect a finished pipeline run

Once a pipeline run has completed, we can access it using the ZenML Repository, about which you will find more details in a later section.

Each pipeline can have multiple runs associated with it, and for each run there might be several outputs for each step. Thus, to inspect a specific output, we first need to access the respective pipeline, then fetch the respective run, and then choose the step output of that specific run.

The overall hierarchy looks like this:

```
repository -> pipelines -> runs -> steps -> outputs

# where -> implies a 1-many relationship.
```

Let us investigate how to traverse this hierarchy level by level:

Repository

The highest level `Repository` object is where to start from.

```python
from zenml.repository import Repository


repo = Repository()
```

Pipelines

The repository contains a collection of all created pipelines with at least one run sorted by the time of their first run from oldest to newest.

You can either access this collection via the `get_pipelines()` method or query a specific pipeline by name using `get_pipeline(pipeline_name=...)`:

```python
# get all pipelines from all stacks
pipelines = repo.get_pipelines()

# now you can get pipelines by index
pipeline_with_latest_initial_run_time = pipelines[-1]

# or get one pipeline by name
pipeline_x = repo.get_pipeline(pipeline="example_pipeline")

```

```python
# or even use the pipeline class
pipeline_x = repo.get_pipeline(pipeline=example_pipeline)
```

Be careful when accessing pipelines by index. Even if you just ran a pipeline it might not be at index `-1`, due to the fact that the pipelines are sorted by time of *first* run. Instead, it is recommended to access the pipeline using the pipeline class, an instance of the class or even the name of the pipeline as a string: `get_pipeline(pipeline=...)`.

Runs

Each pipeline can be executed many times. You can get a list of all runs using the `runs` attribute of a pipeline. Or, you can query a specific run by run name using the `get_run(run_name=...)` method:

```python
# get all runs of a pipeline chronologically ordered
runs = pipeline_x.runs

# get the last run by index, runs are ordered by execution time in ascending order
last_run = runs[-1]

# or get a specific run by name
run = pipeline_x.get_run(run_name=...)
```

Calling `pipeline.runs` can currently be very slow when using remote metadata stores as all run data need to be transferred from the cloud to the local machine.

Alternatively, you can also access the runs from the pipeline class/instance itself.

```python
from zenml.pipelines import pipeline

@pipeline
def example_pipeline(...):
    ...

# get all runs of a pipeline chronologically ordered
runs = example_pipeline.get_runs()

# get the last run by index, runs are ordered by execution time in ascending order
last_run = runs[-1]

# or get a specific run by name
run = example_pipeline.get_run(run_name=...)
```

Steps

Within a given pipeline run you can now further zoom in on individual steps using the `steps` attribute or by querying a specific step using the `get_step(name=...)` method.

```python
# get all steps of a pipeline for a given run
steps = run.steps

# get the step that was executed first
first_step = steps[0]

# or get a specific step by name
step = run.get_step(step="first_step")

# or even use the step class
step = run.get_step(step=first_step)
```

The steps are ordered by time of execution. Depending on the orchestrator, steps can be run in parallel. Thus, accessing steps by index can be unreliable across different runs, and it is recommended to access steps by the step class, an instance of the class or even the name of the step as a string: `get_step(step=...)` instead.

Outputs

Finally, this is how you can inspect the output of a step:

- If there only is a single output, use the `output` attribute
- If there are multiple outputs, use the `outputs` attribute, which is a dictionary that can be indexed using the name of an output:

```python
# The outputs of a step
# if there are multiple outputs they are accessible by name
output = step.outputs["output_name"]

# if there is only one output, use the `.output` property instead
output = step.output

# read the value into memory
output.read()
```

The names of the outputs can be found in the `Output` typing of your steps:

```python
from zenml.steps import step, Output


@step
def some_step() -> Output(output_name=int):
    ...
```

Code Example

Putting it all together, this is how we can access the output of the last step of our example pipeline from the previous sections:

```python
from zenml.repository import Repository

repo = Repository()
pipeline = repo.get_pipeline(pipeline_name="first_pipeline")
last_run = pipeline.runs[-1]
last_step = last_run.steps[-1]
model = last_step.output.read()
```

Configure Automated Caching

How automated caching works in ZenML

Machine learning pipelines are rerun many times over throughout their development lifecycle. Prototyping is often a fast and iterative process that benefits a lot from caching. This makes caching a very powerful tool. Checkout this ZenML Blogpost on Caching for more context on the benefits of caching and ZenBytes lesson 1.2 for a detailed example on how to configure and visualize caching.

Caching in ZenML

ZenML comes with caching enabled by default. Since ZenML automatically tracks and versions all inputs, outputs, and parameters of steps and pipelines, ZenML will not re-execute steps within the same pipeline on subsequent pipeline runs as long as there is no change in these three.

> Currently, the caching does not automatically detect changes within the file system or on external APIs. Make sure to set caching to `False` on steps that depend on external inputs or if the step should run regardless of caching.

Configuring caching behavior of your pipelines

Although caching is desirable in many circumstances, one might want to disable it in certain instances. For example, if you are quickly prototyping with changing step definitions or you have an external API state change in your function that ZenML does not detect.

There are multiple ways to take control of when and where caching is used:

- Disabling caching for the entire pipeline: Do this if you want to turn off all caching (not recommended).

- Disabling caching for individual steps: This is required for certain steps that depend on external input.
- Dynamically disabling caching for a pipeline run: This is useful to force a complete rerun of a pipeline.

Disabling caching for the entire pipeline

On a pipeline level the caching policy can be set as a parameter within the decorator.

```
@pipeline(enable_cache=False)
def first_pipeline(....):
    """Pipeline with cache disabled"""
```

ⓘ If caching is explicitly turned off on a pipeline level, all steps are run without caching, even if caching is set to `True` for single steps.

Disabling caching for individual steps

Caching can also be explicitly turned off at a step level. You might want to turn off caching for steps that take external input (like fetching data from an API or File IO).

```
@step(enable_cache=False)
def import_data_from_api(...):
    """Import most up-to-date data from public api"""
    ...
```

ⓘ You can get a graphical visualization of which steps were cached using ZenML's Pipeline Run Visualization Tool.

Dynamically disabling caching for a pipeline run

Sometimes you want to have control over caching at runtime instead of defaulting to the backed in configurations of your pipeline and its steps. ZenML offers a way to override all caching settings of the pipeline at runtime.

```
first_pipeline(step_1=..., step_2=...).run(enable_cache=False)
```

Code Example

The following example shows caching in action with the code example from the previous section on Runtime Configuration.

For a more detailed example on how caching is used at ZenML and how it works under the hood, checkout ZenBytes lesson 1.2!

Code Example of this Section

```python
import numpy as np
from sklearn.base import ClassifierMixin
from sklearn.svm import SVC

from zenml.integrations.sklearn.helpers.digits import get_digits
from zenml.steps import BaseStepConfig, Output, step
from zenml.pipelines import pipeline


@step
def load_digits() -> Output(
    X_train=np.ndarray, X_test=np.ndarray, y_train=np.ndarray, y_test=np.ndarray
):
    """Loads the digits dataset as normal numpy arrays."""
    X_train, X_test, y_train, y_test = get_digits()
    return X_train, X_test, y_train, y_test


class SVCTrainerStepConfig(BaseStepConfig):
    """Trainer params"""
    gamma: float = 0.001


@step(enable_cache=False)  # never cache this step, always retrain
def svc_trainer(
    config: SVCTrainerStepConfig,
    X_train: np.ndarray,
    y_train: np.ndarray,
) -> ClassifierMixin:
    """Train a sklearn SVC classifier."""
    model = SVC(gamma=config.gamma)
    model.fit(X_train, y_train)
    return model


@pipeline
def first_pipeline(step_1, step_2):
    X_train, X_test, y_train, y_test = step_1()
    step_2(X_train, y_train)


first_pipeline_instance = first_pipeline(
    step_1=load_digits(),
    step_2=svc_trainer()
)

# The pipeline is executed for the first time, so all steps are run.
first_pipeline_instance.run()
```

```python
# Step one will use cache, step two will rerun due to the decorator config
first_pipeline_instance.run()

# The complete pipeline will be rerun
first_pipeline_instance.run(enable_cache=False)
```

Expected Output

Run 1:

```
Creating run for pipeline: first_pipeline
Cache enabled for pipeline first_pipeline
Using stack default to run pipeline first_pipeline...
Step load_digits has started.
Step load_digits has finished in 0.135s.
Step svc_trainer has started.
Step svc_trainer has finished in 0.109s.
Pipeline run first_pipeline-07_Jul_22-12_05_54_573248 has finished in 0.417s.
```

Run 2:

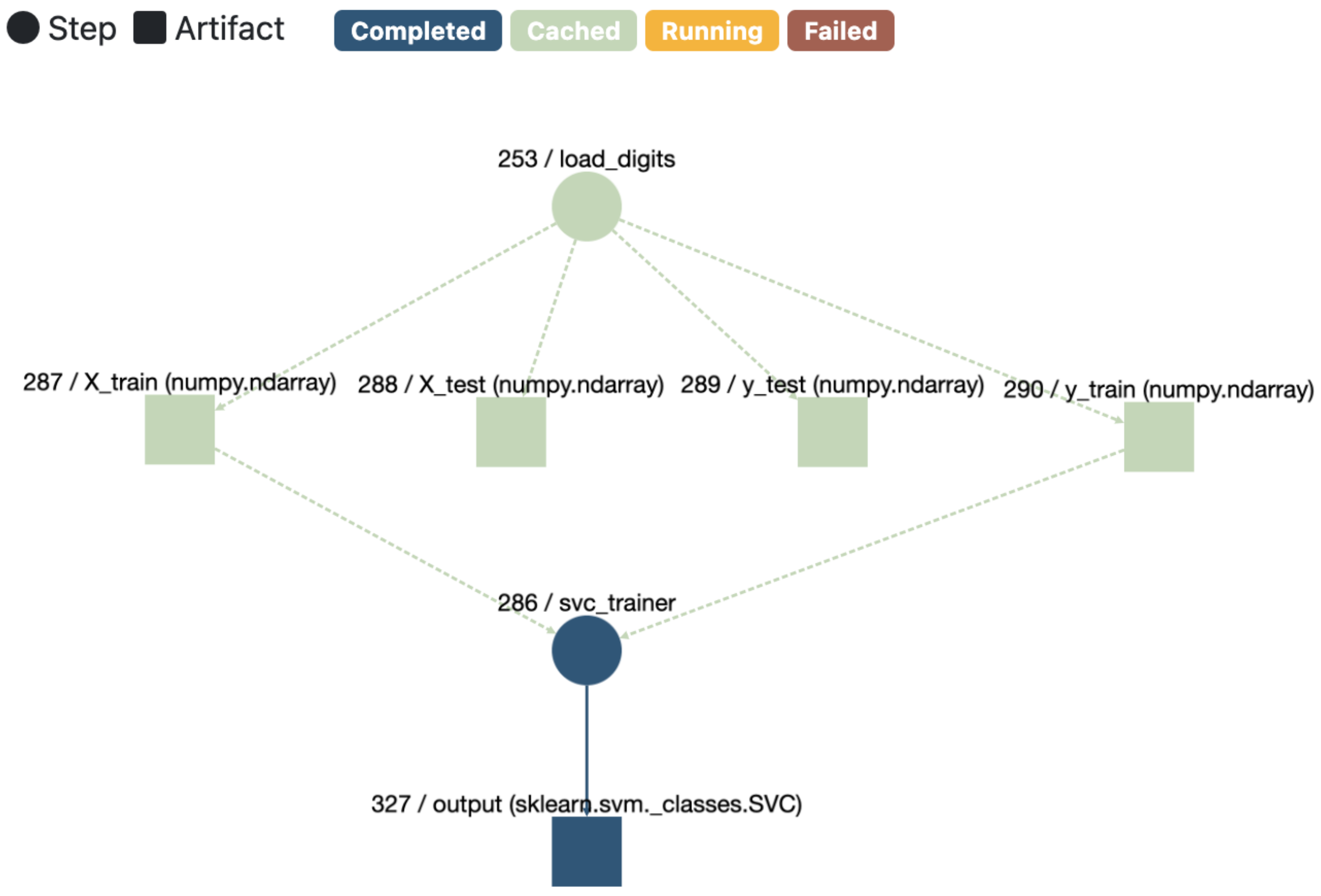
```
Creating run for pipeline: first_pipeline
Cache enabled for pipeline first_pipeline
Using stack default to run pipeline first_pipeline...
Step load_digits has started.
Using cached version of load_digits.
Step load_digits has finished in 0.014s.
Step svc_trainer has started.
Step svc_trainer has finished in 0.051s.
Pipeline run first_pipeline-07_Jul_22-12_05_55_813554 has finished in 0.161s.
```

Run 3:

```
Creating run for pipeline: first_pipeline
Cache enabled for pipeline first_pipeline
Using stack default to run pipeline first_pipeline...
Runtime configuration overwriting the pipeline cache settings to enable_cache=Fals
Step load_digits has started.
Step load_digits has finished in 0.078s.
Step svc_trainer has started.
Step svc_trainer has finished in 0.048s.
Pipeline run first_pipeline-07_Jul_22-12_05_56_718489 has finished in 0.219s.
```

Visualize Data Lineage

ZenML's **Dash** integration provides a `PipelineRunLineageVisualizer` that can be used to visualize pipeline runs in your local browser, as shown below:


● Step
■ Artifact
Completed
Cached
Running
Failed
253 / load_digits
287 / X_train (numpy.ndarray)
288 / X_test (numpy.ndarray)
289 / y_test (numpy.ndarray)
290 / y_train (numpy.ndarray)
286 / svc_trainer
327 / output (sklearn.svm._classes.SVC)

Pipeline Run Visualization Example

Requirements

Before you can use the Dash visualizer, you first need to install ZenML's Dash integration:

```
zenml integration install dash -y
```

See the Integrations page for more details on ZenML integrations and how to install and use them.

Visualizing Pipelines

After a pipeline run has been started, we can access it using the Repository, as you learned in the last section on Inspecting Finished Pipeline Runs.

We can then visualize a run using the `PipelineRunLineageVisualizer` class:

```python
from zenml.integrations.dash.visualizers.pipeline_run_lineage_visualizer import (
    PipelineRunLineageVisualizer,
)
from zenml.repository import Repository


repo = Repository()
latest_run = repo.get_pipeline(<PIPELINE_NAME>).runs[-1]
PipelineRunLineageVisualizer().visualize(latest_run)
```

This will open an interactive visualization in your local browser at `http://127.0.0.1:8050/`, where squares represent your artifacts and circles your pipeline steps.

> The different nodes are color-coded in the visualization, so if your pipeline ever fails or runs for too long, you can find the responsible step at a glance, as it will be colored red or yellow respectively.

Visualizing Caching

In addition to `Completed`, `Running`, and `Failed`, there is also a separate `Cached` state. You already learned about caching in a previous section on Caching Pipeline Runs. Using the `PipelineRunLineageVisualizer`, you can see at a glance which steps were cached (green) and which were rerun (blue). See below for a detailed example.

Code Example

In the following example we use the `PipelineRunLineageVisualizer` to visualize the three pipeline runs from the Caching Pipeline Runs Example:

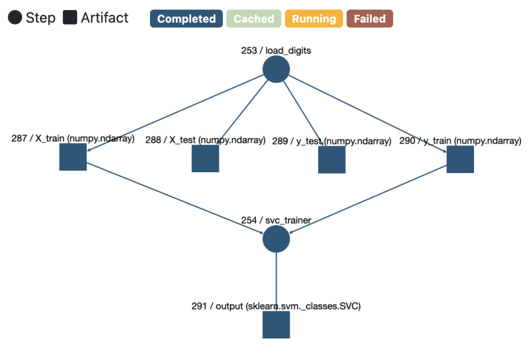
Code Example of this Section

```python
import numpy as np
from sklearn.base import ClassifierMixin
from sklearn.svm import SVC

from zenml.integrations.sklearn.helpers.digits import get_digits
from zenml.steps import BaseStepConfig, Output, step
from zenml.pipelines import pipeline

from zenml.integrations.dash.visualizers.pipeline_run_lineage_visualizer import (
    PipelineRunLineageVisualizer,
)
from zenml.repository import Repository


@step
```

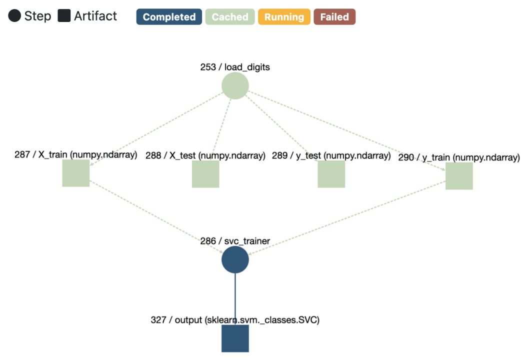
```python
def load_digits() -> Output(
    X_train=np.ndarray, X_test=np.ndarray, y_train=np.ndarray, y_test=np.ndarray
):
    """Loads the digits dataset as normal numpy arrays."""
    X_train, X_test, y_train, y_test = get_digits()
    return X_train, X_test, y_train, y_test


class SVCTrainerStepConfig(BaseStepConfig):
    """Trainer params"""
    gamma: float = 0.001


@step(enable_cache=False)  # never cache this step, always retrain
def svc_trainer(
    config: SVCTrainerStepConfig,
    X_train: np.ndarray,
    y_train: np.ndarray,
) -> ClassifierMixin:
    """Train a sklearn SVC classifier."""
    model = SVC(gamma=config.gamma)
    model.fit(X_train, y_train)
    return model


@pipeline
def first_pipeline(step_1, step_2):
    X_train, X_test, y_train, y_test = step_1()
    step_2(X_train, y_train)


first_pipeline_instance = first_pipeline(
    step_1=load_digits(),
    step_2=svc_trainer()
)


# The pipeline is executed for the first time, so all steps are run.
first_pipeline_instance.run()
latest_run= first_pipeline_instance.get_runs()[-1]
PipelineRunLineageVisualizer().visualize(latest_run)

# Step one will use cache, step two will rerun due to the decorator config
first_pipeline_instance.run()
latest_run = first_pipeline_instance.get_runs()[-1]
PipelineRunLineageVisualizer().visualize(latest_run)

# The complete pipeline will be rerun
first_pipeline_instance.run(enable_cache=False)
latest_run = first_pipeline_instance.get_runs()[-1]
PipelineRunLineageVisualizer().visualize(latest_run)
```

Expected Visualizations

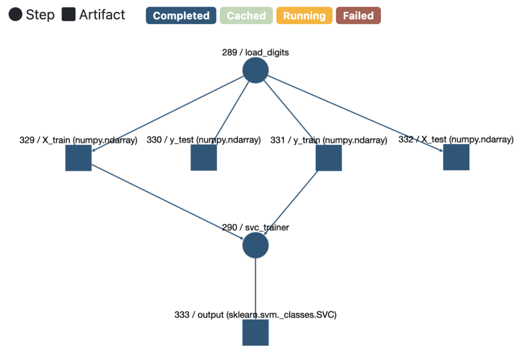
Run 1:


● Step ■ Artifact
Completed
Cached
Running
Failed
253 / load_digits
287 / X_train (numpy.ndarray)
288 / X_test (numpy.ndarray)
289 / y_test (numpy.ndarray)
290 / y_train (numpy.ndarray)
254 / svc_trainer
291 / output (sklearn.svm._classes.SVC)

Run 2:


● Step ■ Artifact
Completed
Cached
Running
Failed
253 / load_digits
287 / X_train (numpy.ndarray)
288 / X_test (numpy.ndarray)
289 / y_test (numpy.ndarray)
290 / y_train (numpy.ndarray)
286 / svc_trainer
327 / output (sklearn.svm._classes.SVC)

Run 3:


● Step ■ Artifact
Completed
Cached
Running
Failed
289 / load_digits
329 / X_train (numpy.ndarray)
330 / y_test (numpy.ndarray)
331 / y_train (numpy.ndarray)
332 / X_test (numpy.ndarray)
290 / svc_trainer
333 / output (sklearn.svm._classes.SVC)

Stacks, Profiles, Repositories

What are stacks, profiles, and repositories in ZenML

Machine learning in production is not just about designing and training models. It is a fractured space consisting of a wide variety of tasks ranging from experiment tracking to orchestration, from model deployment to monitoring, from drift detection to feature stores and much, much more than that. Even though there are already some seemingly well-established solutions for these tasks, it can become increasingly difficult to establish a running production system in a reliable and modular manner once all these solutions are brought together.

This is a problem which is especially critical when switching from a research setting to a production setting. Due to a lack of standards, the time and resources invested in proof of concepts frequently go completely to waste, because the initial system can not easily be transferred to a production-grade setting.

At **ZenML**, we believe that this is one of the most important and challenging problems in the field of MLOps, and it can be solved with a set of standards and well-structured abstractions. Owing to the nature of MLOps, it is essential that these abstractions not only cover concepts such as pipelines and steps but also the infrastructure elements on which the pipelines run.

Taking this into consideration, ZenML provides additional abstractions that help you simplify infrastructure configuration and management:

- Stacks represent different configurations of MLOps tools and infrastructure; Each stack consists of multiple **Stack Components** that each come in several **Flavors**,
- Profiles manage these stacks and enable having various different ZenML configurations on the same machine,
- Repositories link stacks to the pipeline and step code of your ML projects.

Stacks: Configure MLOps Tooling and Infrastructure

How to configure MLOps tooling and infrastructure with stacks

In ZenML, a **Stack** represents a set of configurations for your MLOps tools and infrastructure. For instance, you might want to:

- Orchestrate your ML workflows with Kubeflow,
- Save ML artifacts in an Amazon S3 bucket,
- Track ML metadata in a managed MySQL database,
- Track your experiments with Weights & Biases,
- Deploy models on Kubernetes with Seldon,

Any such combination of tools and infrastructure can be registered as a separate stack in ZenML. Since ZenML code is tooling-independent, you can switch between stacks with a single command and then automatically execute your ML workflows on the desired stack without having to modify your code.

Stack Components

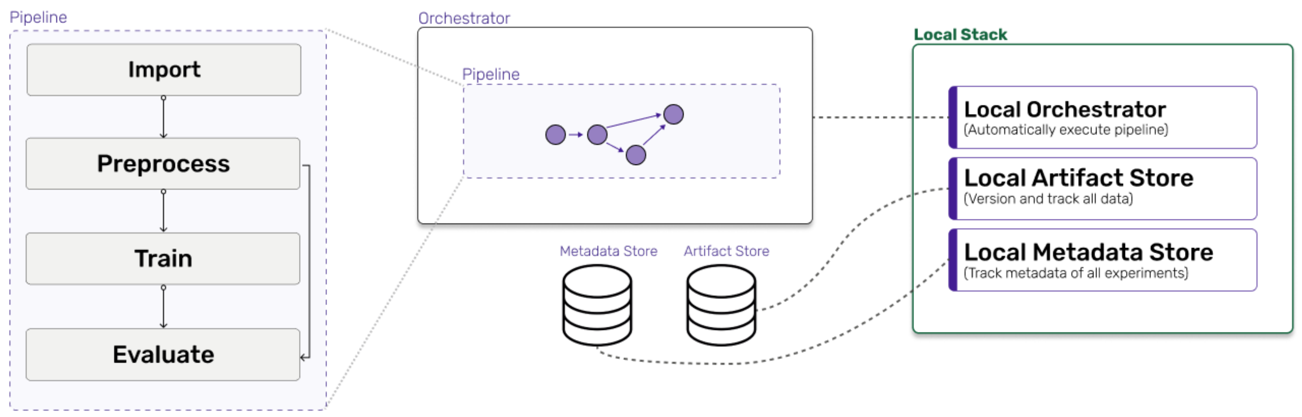
In ZenML, each MLOps tool is associated to a specific **Stack Component**, which is responsible for one specific task of your ML workflow.

For instance, each ZenML stack includes an *Orchestrator* which is responsible for the execution of the steps within your pipeline, an *Artifact Store* which is responsible for storing the artifacts generated by your pipelines, and a *Metadata Store* that tracks what artifact was produced or consumed by which pipeline steps.

ⓘ Check out the Categories of MLOps Tools page for a detailed overview of available stack components in ZenML.

Orchestrator, Artifact Store, and Metadata Store

As mentioned above, orchestrators, artifact stores, and metadata stores are the three components that need to be in every ZenML stack. The interaction of these three components enables a lot of the magic of ZenML, such as data and model versioning, automated artifact lineage tracking, automated caching, and more.



Orchestrators, Artifact Store, and Metadata Store

Orchestrator

The Orchestrator is the component that defines how and where each pipeline step is executed when calling `pipeline.run()`. By default, all runs are executed locally, but by configuring a different orchestrator you can, e.g., automatically execute your ML workflows on Kubeflow instead.

Artifact Stores

Under the hood, all the artifacts in our ML pipeline are automatically stored in an Artifact Store. By default, this is simply a place in your local file system, but we could also configure ZenML to store this data in a cloud bucket like Amazon S3 or any other place instead.

Metadata Stores

In addition to the artifact itself, ZenML automatically stores metadata about each pipeline run in a Metadata Store. By default, this uses an SQLite database on your local machine, but we could again switch it out for another storage type, such as a MySQL database deployed in the cloud.

Stack Component Flavors

The specific tool you are using is called a **Flavor** of the stack component. E.g., *Kubeflow* is a flavor of the *Orchestrator* stack component.

Out-of-the-box, ZenML already comes with a wide variety of flavors, which are either built-in or enabled through the installation of specific Integrations.

The default Stack

By default, ZenML itself and every Repository that you create already come with an initial active `default` stack, which features:

- A local orchestrator,
- A local artifact store,
- A local SQLite metadata store.

If you followed the code examples in the Steps and Pipelines section, then you have already used this stack implicitly to run all of your pipelines.

Listing Stacks, Stack Components, and Flavors

You can see a list of all your *registered* stacks with the following command:

```
zenml stack list
```

Similarly, you can see all *registered* stack components of a specific type using:

```
zenml <STACK_COMPONENT> list
```

In order to see all the *available* flavors for a specific stack component, use:

```
zenml <STACK_COMPONENT> flavor list
```

Our CLI features a wide variety of commands that let you manage and use your stacks. If you would like to learn more, please run: "`zenml stack --help`" or visit our CLI docs.

Registering New Stacks

You can combine various MLOps tools into a ZenML stack as follows:

1. Register a stack component for each tool using `zenml <STACK_COMPONENT> register`,
2. Register a stack to bring all tools together using `zenml stack register`,
3. Activate the stack using `zenml stack set`. Now all your code is automatically executed using the desired tools / infrastructure.

Registering Stack Components

First, you need to create a new instance of the respective stack component with the desired flavor using `zenml <STACK_COMPONENT> register <NAME> --flavor=<FLAVOR>`. Most flavors require further parameters that you can pass as additional arguments `--param=value`, similar to how we passed the flavor.

E.g., to register a *local* artifact store, we could use the following command:

```
zenml artifact-store register <ARTIFACT_STORE_NAME> \
    --flavor=local \
    --path=/path/to/your/store
```

In case you do not know all the available parameters, you can also use the interactive mode to register stack components. This will then walk you through each parameter (to skip just press ENTER):

```
zenml artifact-store register <ARTIFACT_STORE_NAME> \
    --flavor=local -i
```

Afterwards, you should be able to see the new artifact store in the list of registered artifact stores, which you can access using the following command:

```
zenml artifact-store list
```

Our CLI features a wide variety of commands that let you manage and use your stack components and flavors. If you would like to learn more, please run `zenml <STACK_COMPONENT> --help` or visit our CLI docs.

Registering a Stack

After registering each tool as the respective stack component, you can combine all of them into one stack using the `zenml stack register` command:

```
zenml stack register <STACK_NAME> \
    --orchestrator <ORCHESTRATOR_NAME> \
    --artifact-store <ARTIFACT_STORE_NAME> \
    --metadata-store <METADATA_STORE_NAME> \
    ...
```

You can use `zenml stack register --help` to see a list of all possible arguments to the `zenml stack register` command, including a list of which option to use for which stack component.

Activating a Stack

Finally, to start using the stack you just registered, set it as active:

```
zenml stack set <STACK_NAME>
```

Now all your code is automatically executed using this stack.

> Some advanced stack component flavors might require connecting to remote infrastructure components prior to running code on the stack. This can be done using `zenml stack up`. See the Managing Stack States section for more details.

Changing Stacks

If you have multiple stacks configured, you can switch between them using the `zenml stack set` command, similar to how you activate a stack.

Unregistering Stacks

To unregister (delete) a stack and all of its components, run

```
zenml stack delete <STACK_NAME>
```

to delete the stack itself, followed by

```
zenml <STACK_COMPONENT> delete <STACK_COMPONENT_NAME>
```

to delete each of the individual stack components.

> If you provisioned infrastructure related to the stack, make sure to deprovision it using `zenml stack down --force` before unregistering the stack. See the Managing Stack States section for more details.

Profiles: Manage Stack Configurations

How to manage stack configurations with profiles

ZenML implicitly stores all the information about the configured Stacks, Stack Components, and Stack Component Flavors in the Global Configuration on the filesystem of the machine where it is installed. The details of how ZenML stores this persistent data, where it is located, and how it is accessed can be configured via the ZenML **Profile**.

The default Profile

The first time you run any `zenml` CLI command on a machine, a `default` Profile is automatically created and set as active. Unless otherwise configured, this `default` profile will contain all stacks and stack components that you create on your machine.

```
$ zenml profile list
Creating default profile...
Initializing profile `default`...
Initializing store...
Registered stack component with type 'orchestrator' and name 'default'.
Registered stack component with type 'metadata_store' and name 'default'.
Registered stack component with type 'artifact_store' and name 'default'.
Registered stack with name 'default'.
Created and activated default profile.
Running without an active repository root.
Running with active profile: 'default' (global)
┏━━━━━━━━┯━━━━━━━━━━━━━━┯━━━━━━━━━━━━┯━━━━━━━━━━━━━━━━━━━━━━━━━━┯━━━━━━━━━━━━━━┓
┃ ACTIVE │ PROFILE NAME │ STORE TYPE │ URL                      │ ACTIVE STACK ┃
┠────────┼──────────────┼────────────┼──────────────────────────┼──────────────┨
┃        │ default      │ local      │ file:///home/stefan/.c…  │ default      ┃
┗━━━━━━━━┷━━━━━━━━━━━━━━┷━━━━━━━━━━━━┷━━━━━━━━━━━━━━━━━━━━━━━━━━┷━━━━━━━━━━━━━━┛
```

Creating and Changing Profiles

Multiple Profiles can be created on the same machine to simulate the experience of using several independent ZenML instances completely isolated from each other. The same or even different users can then manage their projects in the context of different Profiles on the same machine without having to worry about overwriting existing configurations.

To create a new profile, run

```
zenml profile create <NEW_PROFILE_NAME>
```

To set an existing profile as active, run

```
zenml profile set <PROFILE_NAME>
```

The active Profile determines the stacks and stack components that are available for use by ZenML pipelines. New stacks and stack components registered via the CLI are only added to the active profile and are available only as long as that profile is active.

If you want to reuse stacks from other profiles, you can use the `zenml stack copy` CLI command to copy stacks between profiles. For more information, run `zenml stack copy --help` or visit our CLI docs.

Detailed Example

Detailed usage example of multiple profiles

The following example creates a new profile named `zenml`, sets it as active, and then shows how the `default` profile is unaffected by the operations performed while the `zenml` profile is active:

```
$ zenml profile create zenml
Running without an active repository root.
Running with active profile: 'default' (global)
Initializing profile `zenml`...
Initializing store...
Registered stack component with type 'orchestrator' and name 'default'.
Registered stack component with type 'metadata_store' and name 'default'.
Registered stack component with type 'artifact_store' and name 'default'.
Registered stack with name 'default'.
Profile 'zenml' successfully created.

$ zenml profile list
Running without an active repository root.
Running with active profile: 'default' (global)
┏━━━━━━━━┯━━━━━━━━━━━━━━┯━━━━━━━━━━━━┯━━━━━━━━━━━━━━━━━━━━━━━━━━┯━━━━━━━━━━━━━━┓
┃ ACTIVE │ PROFILE NAME │ STORE TYPE │ URL                      │ ACTIVE STACK ┃
┠────────┼──────────────┼────────────┼──────────────────────────┼──────────────┨
┃       │ default      │ local      │ file:///home/stefan/.c… │ default      ┃
┃        │ zenml        │ local      │ file:///home/stefan/.c… │ default      ┃
┗━━━━━━━━┷━━━━━━━━━━━━━━┷━━━━━━━━━━━━┷━━━━━━━━━━━━━━━━━━━━━━━━━━┷━━━━━━━━━━━━━━┛

$ zenml profile set zenml
Running without an active repository root.
Running with active profile: 'default' (global)
Active profile changed to: 'zenml'

$ zenml profile list
Running without an active repository root.
Running with active profile: 'zenml' (global)
┏━━━━━━━━┯━━━━━━━━━━━━━━┯━━━━━━━━━━━━┯━━━━━━━━━━━━━━━━━━━━━━━━━━┯━━━━━━━━━━━━━━┓
┃ ACTIVE │ PROFILE NAME │ STORE TYPE │ URL                      │ ACTIVE STACK ┃
┠────────┼──────────────┼────────────┼──────────────────────────┼──────────────┨
┃        │ default      │ local      │ file:///home/stefan/.c… │ default      ┃
┃       │ zenml        │ local      │ file:///home/stefan/.c… │ default      ┃
┗━━━━━━━━┷━━━━━━━━━━━━━━┷━━━━━━━━━━━━┷━━━━━━━━━━━━━━━━━━━━━━━━━━┷━━━━━━━━━━━━━━┛

$ zenml stack register custom -m default -a default -o default
Running without an active repository root.
Running with active profile: 'zenml' (global)
Registered stack with name 'custom'.
Stack 'custom' successfully registered!

$ zenml stack set custom
Running without an active repository root.
Running with active profile: 'zenml' (global)
```

```
Active stack set to: 'custom'

$ zenml stack list
Running without an active repository root.
Running with active profile: 'zenml' (global)
┏━━━━━━━━┯━━━━━━━━━━━━┯━━━━━━━━━━━━━━━━┯━━━━━━━━━━━━━━━━┯━━━━━━━━━━━━━━┓
┃ ACTIVE │ STACK NAME │ ARTIFACT_STORE │ METADATA_STORE │ ORCHESTRATOR ┃
┠────────┼────────────┼────────────────┼────────────────┼──────────────┨
┃       │ custom     │ default        │ default        │ default     ┃
┃        │ default    │ default        │ default        │ default      ┃
┗━━━━━━━━┷━━━━━━━━━━━━┷━━━━━━━━━━━━━━━━┷━━━━━━━━━━━━━━━━┷━━━━━━━━━━━━━━┛

$ zenml profile list
Running without an active repository root.
Running with active profile: 'zenml' (global)
┏━━━━━━━━┯━━━━━━━━━━━━━━┯━━━━━━━━━━━━┯━━━━━━━━━━━━━━━━━━━━━━━━━┯━━━━━━━━━━━━━━┓
┃ ACTIVE │ PROFILE NAME │ STORE TYPE │ URL                     │ ACTIVE STACK ┃
┠────────┼──────────────┼────────────┼─────────────────────────┼──────────────┨
┃        │ default      │ local      │ file:///home/stefan/.c… │ default      ┃
┃       │ zenml        │ local      │ file:///home/stefan/.c… │ custom      ┃
┗━━━━━━━━┷━━━━━━━━━━━━━━┷━━━━━━━━━━━━┷━━━━━━━━━━━━━━━━━━━━━━━━━┷━━━━━━━━━━━━━━┛

$ zenml profile set default
Running without an active repository root.
Running with active profile: 'zenml' (global)
Active profile changed to: 'default'

$ zenml stack list
Running without an active repository root.
Running with active profile: 'default' (global)
┏━━━━━━━━┯━━━━━━━━━━━━┯━━━━━━━━━━━━━━━━┯━━━━━━━━━━━━━━━━┯━━━━━━━━━━━━━━┓
┃ ACTIVE │ STACK NAME │ ARTIFACT_STORE │ METADATA_STORE │ ORCHESTRATOR ┃
┠────────┼────────────┼────────────────┼────────────────┼──────────────┨
┃       │ default    │ default        │ default        │ default     ┃
┗━━━━━━━━┷━━━━━━━━━━━━┷━━━━━━━━━━━━━━━━┷━━━━━━━━━━━━━━━━┷━━━━━━━━━━━━━━┛
```

From the above example, you may have also noticed that the *Active Profile* and the *Active Stack* are global settings that affect all other user sessions open on the same machine. It is however possible to set the active profile and active stack individually for each user session or project. Keep reading to learn more.

Repositories: Link Stacks to Code

How to link stacks to code with repositories

ZenML has two main locations where it stores information on the local machine. These are the Global Configuration and the *Repository*. The latter is also referred to as the *.zen folder*.

The ZenML **Repository** related to a pipeline run is the folder that contains all the files needed to execute the run, such as the respective Python scripts and modules where the pipeline is defined, or other associated files. The repository plays a double role in ZenML:

- It is used by ZenML to identify which files must be copied into Docker images in order to execute pipeline steps remotely, e.g., when orchestrating pipelines with Kubeflow.
- It defines the local active Profile and active Stack that will be used when running pipelines from the repository root or one of its sub-folders, as shown below.

Registering a Repository

You can register your current working directory as a ZenML repository by running:

```
zenml init
```

This will create a `.zen` directory, which contains a single `config.yaml` file that stores the local settings:

```
active_profile_name: default
active_stack_name: default
```

It is recommended to use the `zenml init` command to initialize a ZenML *Repository* in the same location of your custom Python source tree where you would normally point `PYTHONPATH`, especially if your Python code relies on a hierarchy of modules spread out across multiple sub-folders.

ZenML CLI commands and ZenML code will display a warning if they are not running in the context of a ZenML repository, e.g.:

```
stefan@aspyre2:/tmp$ zenml stack list
Unable to find ZenML repository in your current working directory (/tmp) or any p
Running without an active repository root.
Running with active profile: 'default' (global)
┏━━━━━━━━┯━━━━━━━━━━━━┯━━━━━━━━━━━━━━━━┯━━━━━━━━━━━━━━━━┯━━━━━━━━━━━━━━┓
┃ ACTIVE │ STACK NAME │ ARTIFACT_STORE │ METADATA_STORE │ ORCHESTRATOR ┃
┠────────┼────────────┼────────────────┼────────────────┼──────────────┨
┃        │ default    │ default        │ default        │ default      ┃
┗━━━━━━━━┷━━━━━━━━━━━━┷━━━━━━━━━━━━━━━━┷━━━━━━━━━━━━━━━━┷━━━━━━━━━━━━━━┛
```

Setting Local Active Profile and Stack

One of the most useful features of repositories is that you can configure a different active profile and stack for each of your projects. This is great if you want to use ZenML for multiple projects on the same machine. Whenever you create a new ML project, we recommend you run `zenml init` to create a separate repository, then create and activate a new profile, and then use it to define your stacks:

```
zenml init
zenml profile create <NEW_PROFILE_NAME>
zenml profile set <NEW_PROFILE_NAME>
zenml stack register ...
zenml stack set ...
```

If you do this, the correct profile and stack will automatically get activated whenever you change directory from one project to another in your terminal.

> Note that the stacks and stack components are still stored globally, even when running from inside a ZenML repository. It is only the active profile and active stack settings that can be configured locally.

Detailed Example

Detailed usage example of local stacks and profiles

The following example shows how the active profile and active stack can be configured locally for a project without impacting the global settings:

```
/tmp/zenml$ zenml profile list
Running without an active repository root.
Running with active profile: 'default' (global)
┏━━━━━━━━┯━━━━━━━━━━━━━━┯━━━━━━━━━━━━┯━━━━━━━━━━━━━━━━━━━━━━━━━━┯━━━━━━━━━━━━━━┓
┃ ACTIVE │ PROFILE NAME │ STORE TYPE │ URL                      │ ACTIVE STACK ┃
┠────────┼──────────────┼────────────┼──────────────────────────┼──────────────┨
┃      │ default      │ local      │ file:///home/stefan/.c… │ default      ┃
┃        │ zenml        │ local      │ file:///home/stefan/.c… │ custom       ┃
┗━━━━━━━━┷━━━━━━━━━━━━━━┷━━━━━━━━━━━━┷━━━━━━━━━━━━━━━━━━━━━━━━━━┷━━━━━━━━━━━━━━┛

/tmp/zenml$ zenml init
ZenML repository initialized at /tmp/zenml.
The local active profile was initialized to 'default' and the local active stack
to 'default'. This local configuration will only take effect when you're running
ZenML from the initialized repository root, or from a subdirectory. For more
information on profile and stack configuration, please visit https://docs.zenml.io

/tmp/zenml$ zenml profile list
Running with active profile: 'default' (local)
┏━━━━━━━━┯━━━━━━━━━━━━━━┯━━━━━━━━━━━━┯━━━━━━━━━━━━━━━━━━━━━━━━━━┯━━━━━━━━━━━━━━┓
┃ ACTIVE │ PROFILE NAME │ STORE TYPE │ URL                      │ ACTIVE STACK ┃
┠────────┼──────────────┼────────────┼──────────────────────────┼──────────────┨
┃      │ default      │ local      │ file:///home/stefan/.c… │ default      ┃
┃        │ zenml        │ local      │ file:///home/stefan/.c… │ custom       ┃
┗━━━━━━━━┷━━━━━━━━━━━━━━┷━━━━━━━━━━━━┷━━━━━━━━━━━━━━━━━━━━━━━━━━┷━━━━━━━━━━━━━━┛

/tmp/zenml$ zenml profile set zenml
Running with active profile: 'default' (local)
```

```
Active profile changed to: 'zenml'
/tmp/zenml$ zenml stack list
Running with active profile: 'zenml' (local)
┏━━━━━━━━┯━━━━━━━━━━━━┯━━━━━━━━━━━━━━━━┯━━━━━━━━━━━━━━━━┯━━━━━━━━━━━━━━┓
┃ ACTIVE │ STACK NAME │ ARTIFACT_STORE │ METADATA_STORE │ ORCHESTRATOR ┃
┠────────┼────────────┼────────────────┼────────────────┼──────────────┨
┃        │ default    │ default        │ default        │ default      ┃
┃        │ custom     │ default        │ default        │ default      ┃
┗━━━━━━━━┷━━━━━━━━━━━━┷━━━━━━━━━━━━━━━━┷━━━━━━━━━━━━━━━━┷━━━━━━━━━━━━━━┛

/tmp/zenml$ zenml stack set default
Running with active profile: 'zenml' (local)
Active stack set to: 'default'

/tmp/zenml$ zenml stack list
Running with active profile: 'zenml' (local)
┏━━━━━━━━┯━━━━━━━━━━━━┯━━━━━━━━━━━━━━━━┯━━━━━━━━━━━━━━━━┯━━━━━━━━━━━━━━┓
┃ ACTIVE │ STACK NAME │ ARTIFACT_STORE │ METADATA_STORE │ ORCHESTRATOR ┃
┠────────┼────────────┼────────────────┼────────────────┼──────────────┨
┃        │ default    │ default        │ default        │ default      ┃
┃        │ custom     │ default        │ default        │ default      ┃
┗━━━━━━━━┷━━━━━━━━━━━━┷━━━━━━━━━━━━━━━━┷━━━━━━━━━━━━━━━━┷━━━━━━━━━━━━━━┛

/tmp/zenml$ cd ..
/tmp$ zenml profile list
Running without an active repository root.
Running with active profile: 'default' (global)
┏━━━━━━━━┯━━━━━━━━━━━━━━┯━━━━━━━━━━━━┯━━━━━━━━━━━━━━━━━━━━━━━━━━┯━━━━━━━━━━━━━━┓
┃ ACTIVE │ PROFILE NAME │ STORE TYPE │ URL                      │ ACTIVE STACK ┃
┠────────┼──────────────┼────────────┼──────────────────────────┼──────────────┨
┃        │ default      │ local      │ file:///home/stefan/.c… │ default      ┃
┃        │ zenml        │ local      │ file:///home/stefan/.c… │ custom       ┃
┗━━━━━━━━┷━━━━━━━━━━━━━━┷━━━━━━━━━━━━┷━━━━━━━━━━━━━━━━━━━━━━━━━━┷━━━━━━━━━━━━━━┛

/tmp$ cd zenml
/tmp/zenml$ zenml stack list
Running with active profile: 'zenml' (local)
┏━━━━━━━━┯━━━━━━━━━━━━┯━━━━━━━━━━━━━━━━┯━━━━━━━━━━━━━━━━┯━━━━━━━━━━━━━━┓
┃ ACTIVE │ STACK NAME │ ARTIFACT_STORE │ METADATA_STORE │ ORCHESTRATOR ┃
┠────────┼────────────┼────────────────┼────────────────┼──────────────┨
┃        │ default    │ default        │ default        │ default      ┃
┃        │ custom     │ default        │ default        │ default      ┃
┗━━━━━━━━┷━━━━━━━━━━━━┷━━━━━━━━━━━━━━━━┷━━━━━━━━━━━━━━━━┷━━━━━━━━━━━━━━┛
```

Using the Repository in Python

You can access your repository in Python using the `zenml.repository.Repository` class:

```python
from zenml.repository import Repository
```

```python
repo = Repository()
```

This allows you to perform various repository operations directly in Python, such as Inspecting Finished Pipeline Runs or accessing and managing stacks, as shown below.

> To explore all possible operations that can be performed via the `Repository`, please consult the API docs section on Repository.

Accessing the Active Stack

The following code snippet shows how you can retrieve or modify information of your active stack and stack components in Python:

```python
from zenml.repository import Repository


repo = Repository()
active_stack = repo.active_stack
print(active_stack.name)
print(active_stack.orchestrator.name)
print(active_stack.artifact_store.name)
print(active_stack.artifact_store.path)
print(active_stack.metadata_store.name)
print(active_stack.metadata_store.uri)
```

Registering and Changing Stacks

In the following we use the repository to register a new ZenML stack called `local` and set it as the active stack of the repository:

```python
from zenml.repository import Repository
from zenml.artifact_stores import LocalArtifactStore
from zenml.metadata_stores import SQLiteMetadataStore
from zenml.orchestrators import LocalOrchestrator
from zenml.stack import Stack


repo = Repository()

# Create a new orchestrator
orchestrator = LocalOrchestrator(name="local")

# Create a new metadata store
metadata_store = SQLiteMetadataStore(
    name="local",
    uri="/tmp/zenml/zenml.db",
)

# Create a new artifact store
artifact_store = LocalArtifactStore(
    name="local",
    path="/tmp/zenml/artifacts",
)

# Create a new stack with the new components
stack = Stack(
    name="local",
    orchestrator=orchestrator,
    metadata_store=metadata_store,
    artifact_store=artifact_store,
)

# Register the new stack in the currently active profile
repo.register_stack(stack)

# Set the stack as the active stack of the repository
repo.activate_stack(stack.name)
```

Unregistering a Repository

To unregister a repository, delete the `.zen` directory in the respective location, e.g., via

```
rm -rf .zen
```

Advanced Usage

An overview of advanced ZenML use cases

The previous sections on Steps and Pipelines and Stacks, Profiles, Repositories already cover most of the concepts you will need for developing ML workflows with ZenML.

However, there are a few additional use cases that you might or might not encounter throughout your journey, about which you can learn more here.

List of Advanced Use Cases

- Writing Custom Stack Component Flavors can be useful when trying to use ZenML with tooling or infrastructure for which no official integration exists yet.
- Managing Stack Component States is required for certain integrations with remote components and can also be used to configure custom setup behavior of custom stack component flavors.
- Passing Custom Data Types through Steps via **Materializers** is required if one of your steps outputs a custom class or other data types, for which materialization is not defined by ZenML itself.

- Accessing the Active Stack within Steps via **Step Fixtures** can, for instance, be used to load the best performing prior model to compare newly trained models against.
- Accessing Global Info within Steps via the **Environment** can be useful to get system information, the Python version, or the name of the current step, pipeline, and run.
- Managing External Services might be required for deploying custom models or for the UI's of some visualization tools like TensorBoard. These services are usually long-lived external processes that persist beyond the execution of your pipeline runs.
- Managing Docker Images is required for some remote orchestrators and step operators to run your pipeline code in an isolated and well-defined environment.
- Setting Stacks and Profiles with Environment Variables enables you to configure pipeline runs with environment variables instead of the repository.
- Migrating Legacy Stacks to ZenML Profiles contains additional information for returning users that want to port their ZenML stacks to the newest version.

Write Custom Stack Component Flavors

How to write a custom stack component flavor

When building sophisticated ML workflows, you will often need to come up with custom-tailed solutions. Sometimes, this might even require you to use custom components for your infrastructure or tooling.

That is exactly why the stack components in ZenML were designed to be modular and straightforward to extend. Using ZenML's base abstractions, you can create your own stack component flavors and use custom solutions for any stack component.

Base Abstractions

Before we get into how custom stack component flavors can be defined, let us briefly discuss how ZenML's abstraction for stack components are designed.

Abstract Stack Component Base Abstraction

All stack components in ZenML inherit from a common abstract base class `StackComponent`, parts of which you can see below:

```python
from abc import ABC
from pydantic import BaseModel, Field
from typing import ClassVar
from uuid import UUID, uuid4

from zenml.enums import StackComponentType

class StackComponent(BaseModel, ABC):
    """Abstract class for all components of a ZenML stack."""

    # Instance configuration
    name: str
    uuid: UUID = Field(default_factory=uuid4)

    # Class parameters
    TYPE: ClassVar[StackComponentType]
    FLAVOR: ClassVar[str]

    ...
```

There are a few things to unpack here. Let's talk about Pydantic first. Pydantic is a library for data validation and settings management. Using their `BaseModel` is helping us to configure and serialize these components while allowing us to add a validation layer to each stack component instance/implementation.

You can already see how that comes into play here within the base `StackComponent` implementation. As you can see, each instance of a `StackComponent` needs to include a `name` and an auto-generated `uuid`. These variables will be tracked when we serialize the stack component object. (You can exclude an instance configuration parameter from the serialization by giving it a name which starts with `_`.)

Moreover, you can use class variables by denoting them with the `ClassVar[..]`, which are also excluded from the serialization. Each `StackComponent` implementation features two important class variables called the `TYPE` and the `FLAVOR`. The `TYPE` is utilized when we set up the base implementation for a specific type of stack component whereas the `FLAVOR` parameter is used to denote different flavors (which we will cover in the next section).

Component-Specific Base Abstraction

For each stack component, there then exists another component-specific base abstraction which all flavors should inherit from. These component-specific are themselves subclasses of `StackComponent`.

As an example, let us take a look at the `BaseArtifactStore`:

```
from typing import ClassVar, Set

from zenml.enums import StackComponentType
from zenml.stack import StackComponent


class BaseArtifactStore(StackComponent):
    """Abstract class for all ZenML artifact stores."""

    # Instance configuration
    path: str

    # Class parameters
    TYPE: ClassVar[StackComponentType] = StackComponentType.ARTIFACT_STORE
    SUPPORTED_SCHEMES: ClassVar[Set[str]]


    ...
```

As you can see, the `BaseArtifactStore` sets the correct `TYPE`, while introducing a new instance variable called `path` and class variable called `SUPPORTED_SCHEMES`, which will be used by all the subclasses of this base implementation.

Building Custom Stack Component Flavors

In order to build a new flavor for a stack component, you need to create a custom class that inherits from the respective component-specific base abstraction, defines the `FLAVOR` class variable, and implements any abstract or flavor-specific methods and properties.

As an example, this is how you could define a custom artifact store:

```python
from typing import ClassVar, Set

from zenml.artifact_stores import BaseArtifactStore


class MyCustomArtifactStore(BaseArtifactStore):
    """Custom artifact store implementation."""

    # Class configuration
    FLAVOR: ClassVar[str] = "custom"  # the name you want your flavor to have
    SUPPORTED_SCHEMES: ClassVar[Set[str]] = {"custom://"}  # implement this

    ...  # custom functionality
```

As you can see from the example above, `MyCustomArtifactStore` inherits from the corresponding base abstraction `BaseArtifactStore` and implements a `custom` flavor.

You could now register this `custom` artifact store via `zenml artifact-store flavor register`:

```
zenml artifact-store flavor register <path.to.MyCustomArtifacStore>
```

Afterwards, you should see the new `custom` artifact store in the list of available artifact store flavors:

```
zenml artifact-store flavor list
```

And that's it, you now have defined a custom stack component flavor that you can use in any of your stacks just like any other flavor you used before, e.g.:

```
zenml artifact-store register <ARTIFACT_STORE_NAME> \
    --flavor=custom \
    ...

zenml stack register <STACK_NAME> \
    --artifact-store <ARTIFACT_STORE_NAME> \
    ...
```

If your custom stack component flavor requires special setup before it can be used, check out the Managing Stack Component States section for more details.

Extending Specific Stack Components

If you would like to learn more about how to build a custom stack component flavor for a specific stack component, please check the links below:

Type of Stack Component	Description
Orchestrator	Orchestrating the runs of your pipeline
Artifact Store	Storage for the artifacts created by your pipelines
Metadata Store	Tracking the execution of your pipelines/steps
Container Registry	Store for your containers
Secrets Manager	Centralized location for the storage of your secret
Step Operator	Execution of individual steps in specialized runtim environments
Model Deployer	Services/platforms responsible for online model serving
Feature Store	Management of your data/features
Experiment Tracker	Tracking your ML experiments
Alerter	Sending alerts through specified channels

Manage Stack Component States

How to start, stop, provision, and deprovision stacks and stack components

Some stack components come with built-in daemons for connecting to the underlying remote infrastructure. These stack components expose functionality for provisioning, deprovisioning, starting, or stopping the corresponding daemons.

See the advanced section on Services for more information on daemons.

For such components, you can manage the daemon state using the `zenml <STACK_COMPONENT> up` and `zenml <STACK_COMPONENT> down` commands. Alternatively, you can also use `zenml stack up` or `zenml stack down` to manage the state of your entire stack:

```
zenml stack up  # Provision and start all stack components
zenml metadata-store up  # Provision and start the metadata store only

zenml stack down  # Stop all stack components
zenml metadata-store down  # Stop the metadata store only

zenml stack down --force  # Stop and deprovision all stack components
zenml metadata-store down --force  # Stop and deprovision the metadata store only
```

Defining States of Custom Components

By default, each stack component is assumed to be in a provisioned and running state right after creation. However, if you want to write a custom component and have fine-grained control over its state, you can overwrite the following properties and methods of the `StackComponent` base interface to configure the component according to your needs:

```python
from abc import ABC
from pydantic import BaseModel


class StackComponent(BaseModel, ABC):
    """Abstract class for all components of a ZenML stack."""
    ...

    @property
    def is_provisioned(self) -> bool:
        """If the component provisioned resources to run."""
        return True

    @property
    def is_running(self) -> bool:
        """If the component is running."""
        return True

    def provision(self) -> None:
        """Provisions resources to run the component."""

    def deprovision(self) -> None:
        """Deprovisions all resources of the component."""

    def resume(self) -> None:
        """Resumes the provisioned resources of the component."""

    def suspend(self) -> None:
        """Suspends the provisioned resources of the component."""

    ...
```

Pass Custom Data Types through Steps

How to use materializers to pass custom data types through steps

A ZenML pipeline is built in a data-centric way. The outputs and inputs of steps define how steps are connected and the order in which they are executed. Each step should be considered as its very own process that reads and writes its inputs and outputs from and to the Artifact Store. This is where **Materializers** come into play.

A materializer dictates how a given artifact can be written to and retrieved from the artifact store and also contains all serialization and deserialization logic.

Whenever you pass artifacts as outputs from one pipeline step to other steps as inputs, the corresponding materializer for the respective data type defines how this artifact is first serialized and written to the artifact store, and then deserialized and read in the next step.

For most data types, ZenML already includes built-in materializers that automatically handle artifacts of those data types. For instance, all of the examples from the Steps and Pipelines section were using built-in materializers under the hood to store and load artifacts correctly.

However, if you want to pass custom objects between pipeline steps, such as a PyTorch model that does not inherit from `torch.nn.Module`, then you need to define a custom Materializer to tell ZenML how to handle this specific data type.

Building a Custom Materializer

Base Implementation

Before we dive into how custom materializers can be built, let us briefly discuss how materializers in general are implemented. In the following, you can see the implementation of the abstract base class `BaseMaterializer`, which defines the interface of all materializers:

```python
from typing import Type, Any
from zenml.materializers.base_materializer import BaseMaterializerMeta


class BaseMaterializer(metaclass=BaseMaterializerMeta):
    """Base Materializer to realize artifact data."""


    ASSOCIATED_ARTIFACT_TYPES = ()
    ASSOCIATED_TYPES = ()

    def __init__(self, artifact: "BaseArtifact"):
        """Initializes a materializer with the given artifact."""
        self.artifact = artifact

    def handle_input(self, data_type: Type[Any]) -> Any:
        """Write logic here to handle input of the step function.

        Args:
            data_type: What type the input should be materialized as.
        Returns:
            Any object that is to be passed into the relevant artifact in the
            step.
        """
        # read from self.artifact.uri
        ...

    def handle_return(self, data: Any) -> None:
        """Write logic here to handle return of the step function.

        Args:
            data: Any object that is specified as an input artifact of the step.
        """
        # write `data` to self.artifact.uri
        ...
```

Which Data Type to Handle?

Each materializer has an `ASSOCIATED_TYPES` attribute that contains a list of data types that this materializer can handle. ZenML uses this information to call the right materializer at the right time. I.e., if a ZenML step returns a `pd.DataFrame`, ZenML will try to find any materializer that has `pd.DataFrame` in its `ASSOCIATED_TYPES`. List the data type of your custom object here to link the materializer to that data type.

What Type of Artifact to Generate

Each materializer also has an `ASSOCIATED_ARTIFACT_TYPES` attribute, which defines what types of artifacts are being stored.

In most cases, you should choose either `DataArtifact` or `ModelArtifact` here. If you are unsure, just use `DataArtifact`. The exact choice is not too important, as the artifact type is only used as a tag in the visualization tools of some certain integrations like Facets.

> ⓘ You can find a full list of available artifact types in the API Docs.

Where to Store the Artifact

Each materializer has an `artifact` object. The most important property of an `artifact` object is the `uri`. The `uri` is automatically created by ZenML whenever you run a pipeline and points to the directory of a file system where the artifact is stored (location in the artifact store). This should not be modified.

How to Store and Retrieve the Artifact

The `handle_input()` and `handle_return()` methods define the serialization and deserialization of artifacts.

- `handle_input()` defines how data is read from the artifact store and deserialized,
- `handle_return()` defines how data is serialized and saved to the artifact store.

These methods you will need to overwrite according to how you plan to serialize your objects. E.g., if you have custom PyTorch classes as `ASSOCIATED_TYPES`, then you might want to use `torch.save()` and `torch.load()` here. For example, have a look at the materializer in the Neural Prophet integration.

Using a Custom Materializer

ZenML automatically scans your source code for definitions of materializers and registers them for the corresponding data type, so just having a custom materializer definition in your code is enough to enable the respective data type to be used in your pipelines.

Alternatively, you can also explicitly define which materializer to use for a specific step using the `with_return_materializers()` method of the step. E.g.:

```
first_pipeline(
    step_1=my_first_step().with_return_materializers(MyMaterializer),
    ...
).run()
```

When there are multiple outputs, a dictionary of type `{<OUTPUT_NAME>:<MATERIALIZER_CLASS>}` can be supplied to the `with_return_materializers()` method.

Note that `with_return_materializers` only needs to be called for the output of the first step that produced an artifact of a given data type, all downstream steps will use the same materializer by default.

Configuring Materializers at Runtime

As briefly outlined in the Runtime Configuration section, which materializer to use for the output of what step can also be configured within YAML config files.

For each output of your steps, you can define custom materializers to handle the loading and saving. You can configure them like this in the config:

```
...
steps:
  <STEP_NAME>:
    ...
    materializers:
      <OUTPUT_NAME>:
        name: <MaterializerName>
        file: <relative/filepath>
```

The name of the output can be found in the function declaration, e.g. `my_step() -> Output(a: int, b: float)` has `a` and `b` as available output names.

Similar to other configuration entries, the materializer `name` refers to the class name of your materializer, and the `file` should contain a path to the module where the materializer is defined.

Basic Example

Let's see how materialization works with a basic example. Let's say you have a custom class called `MyObject` that flows between two steps in a pipeline:

```python
import logging
from zenml.steps import step
from zenml.pipelines import pipeline


class MyObj:
    def __init__(self, name: str):
        self.name = name


@step
def my_first_step() -> MyObj:
    """Step that returns an object of type MyObj"""
    return MyObj("my_object")


@step
def my_second_step(my_obj: MyObj) -> None:
    """Step that logs the input object and returns nothing."""
    logging.info(
        f"The following object was passed to this step: `{my_obj.name}`"
    )


@pipeline
def first_pipeline(step_1, step_2):
    output_1 = step_1()
    step_2(output_1)


first_pipeline(
    step_1=my_first_step(),
    step_2=my_second_step()
).run()
```

Running the above without a custom materializer will result in the following error:

```
zenml.exceptions.StepInterfaceError: Unable to find materializer for output
'output' of type <class '__main__.MyObj'> in step 'step1'. Please make sure
```

```
to either explicitly set a materializer for step outputs using
step.with_return_materializers(...) or registering a default materializer
for specific types by subclassing BaseMaterializer and setting its
ASSOCIATED_TYPES class variable. For more information, visit
https://docs.zenml.io/guides/common-usecases/custom-materializer
```

The error message basically says that ZenML does not know how to persist the object of type `MyObj` (how could it? We just created this!). Therefore, we have to create our own materializer. To do this, you can extend the `BaseMaterializer` by sub-classing it, listing `MyObj` in `ASSOCIATED_TYPES`, and overwriting `handle_input()` and `handle_return()`:

```python
import os
from typing import Type

from zenml.artifacts import DataArtifact
from zenml.io import fileio
from zenml.materializers.base_materializer import BaseMaterializer


class MyMaterializer(BaseMaterializer):
    ASSOCIATED_TYPES = (MyObj,)
    ASSOCIATED_ARTIFACT_TYPES = (DataArtifact,)

    def handle_input(self, data_type: Type[MyObj]) -> MyObj:
        """Read from artifact store"""
        super().handle_input(data_type)
        with fileio.open(os.path.join(self.artifact.uri, 'data.txt'), 'r') as f:
            name = f.read()
        return MyObj(name=name)

    def handle_return(self, my_obj: MyObj) -> None:
        """Write to artifact store"""
        super().handle_return(my_obj)
        with fileio.open(os.path.join(self.artifact.uri, 'data.txt'), 'w') as f:
            f.write(my_obj.name)
```

Pro-tip: Use the ZenML `fileio` module to ensure your materialization logic works across artifact stores (local and remote like S3 buckets).

Now ZenML can use this materializer to handle outputs and inputs of your customs object. Edit the pipeline as follows to see this in action:

```python
first_pipeline(
    step_1=my_first_step().with_return_materializers(MyMaterializer),
    step_2=my_second_step()
).run()
```

Due to the typing of the inputs and outputs and the `ASSOCIATED_TYPES` attribute of the materializer, you won't necessarily have to add `.with_return_materializers(MyMaterializer)` to the step. It should automatically be detected. It doesn't hurt to be explicit though.

This will now work as expected and yield the following output:

```
Creating run for pipeline: `first_pipeline`
Cache enabled for pipeline `first_pipeline`
Using stack `default` to run pipeline `first_pipeline`...
Step `my_first_step` has started.
Step `my_first_step` has finished in 0.081s.
Step `my_second_step` has started.
The following object was passed to this step: `my_object`
Step `my_second_step` has finished in 0.048s.
Pipeline run `first_pipeline-22_Apr_22-10_58_51_135729` has finished in 0.153s.
```

Code Summary

Code Example for Materializing Custom Objects

```python
import logging
import os
from typing import Type

from zenml.steps import step
from zenml.pipelines import pipeline

from zenml.artifacts import DataArtifact
from zenml.io import fileio
from zenml.materializers.base_materializer import BaseMaterializer


class MyObj:
    def __init__(self, name: str):
        self.name = name


class MyMaterializer(BaseMaterializer):
    ASSOCIATED_TYPES = (MyObj,)
    ASSOCIATED_ARTIFACT_TYPES = (DataArtifact,)

    def handle_input(self, data_type: Type[MyObj]) -> MyObj:
        """Read from artifact store"""
        super().handle_input(data_type)
        with fileio.open(os.path.join(self.artifact.uri, 'data.txt'), 'r') as f:
            name = f.read()
        return MyObj(name=name)
    def handle_return(self, my_obj: MyObj) -> None:
        """Write to artifact store"""
        super().handle_return(my_obj)
        with fileio.open(os.path.join(self.artifact.uri, 'data.txt'), 'w') as f:
            f.write(my_obj.name)


@step
def my_first_step() -> MyObj:
    """Step that returns an object of type MyObj"""
    return MyObj("my_object")


@step
def my_second_step(my_obj: MyObj) -> None:
    """Step that log the input object and returns nothing."""
    logging.info(
        f"The following object was passed to this step: `{my_obj.name}`")


@pipeline
def first_pipeline(step_1, step_2):
    output_1 = step_1()
    step_2(output_1)


first_pipeline(
    step_1=my_first_step().with_return_materializers(MyMaterializer),
    step_2=my_second_step()
).run()
```

Skipping Materialization

Using artifacts directly might have unintended consequences for downstream tasks that rely on materialized artifacts. Only skip materialization if there is no other way to do what you want to do.

While materializers should in most cases be used to control how artifacts are returned and consumed from pipeline steps, you might sometimes need to have a completely non-materialized artifact in a step, e.g., if you need to know the exact path to where your artifact is stored.

A non-materialized artifact is a `BaseArtifact` (or any of its subclasses) and has a property `uri` that points to the unique path in the artifact store where the artifact is stored. One can use a non-materialized artifact by specifying it as the type in the step:

```python
from zenml.artifacts import DataArtifact
from zenml.steps import step
```

```python

@step
def my_step(my_artifact: DataArtifact)  # rather than pd.DataFrame
    pass
```

When using artifacts directly, one must be aware of which type they are by looking at the previous step's materializer: if the previous step produces a `ModelArtifact` then you should specify `ModelArtifact` in a non-materialized step.

> Materializers link pythonic types to artifact types implicitly. E.g., a `keras.model` or `torch.nn.Module` are pythonic types that are both linked to `ModelArtifact` implicitly via their materializers.
>
> You can find a full list of available artifact types in the API Docs.

Example

The following shows an example how non-materialized artifacts can be used in the steps of a pipeline. The pipeline we define will look like this:

```
s1 -> s3
s2 -> s4
```

`s1` and `s2` produce identical artifacts, however `s3` consumes materialized artifacts while `s4` consumes non-materialized artifacts. `s4` can now use the `dict_.uri` and `list_.uri` paths directly rather than their materialized counterparts.

```python
from typing import Dict, List

from zenml.artifacts import DataArtifact, ModelArtifact
from zenml.pipelines import pipeline
from zenml.steps import Output, step


@step
def step_1() -> Output(dict_=Dict, list_=List):
    return {"some": "data"}, []


@step
def step_2() -> Output(dict_=Dict, list_=List):
    return {"some": "data"}, []


@step
def step_3(dict_: Dict, list_: List) -> None:
    assert isinstance(dict_, dict)
    assert isinstance(list_, list)


@step
def step_4(dict_: DataArtifact, list_: ModelArtifact) -> None:
    assert hasattr(dict_, "uri")
    assert hasattr(list_, "uri")


@pipeline
def example_pipeline(step_1, step_2, step_3, step_4):
    step_3(*step_1())
    step_4(*step_2())


example_pipeline(step_1(), step_2(), step_3(), step_4()).run()
```

Access the Active Stack within Steps

How to use step fixtures to access the active ZenML stack from within a step

In general, when defining steps, you usually can only supply inputs that have been output by previous steps. However, there are two exceptions:

- An object which is a subclass of `BaseStepConfig`: This object is used to pass run-time parameters to a pipeline run. It can be used to send parameters to a step that are not artifacts. You learned about this one already in the section on Runtime Configuration.
- A Step Context object: This object gives access to the active stack, materializers, and special integration-specific libraries.

These two types of special parameters are comparable to Pytest fixtures, hence we call them **Step Fixtures** at ZenML.

To use step fixtures in your steps, just pass a parameter with the right type hint and ZenML will automatically recognize it.

```
from zenml.steps import step, BaseStepConfig, StepContext


class SubClassBaseStepConfig(BaseStepConfig):
    ...


@step
def my_step(
    config: SubClassBaseStepConfig,  # must be subclass of `BaseStepConfig`
    context: StepContext,  # must be of class `StepContext`
    artifact: str,  # other parameters are assumed to be outputs of other steps
):
    ...
```

The name of the argument can be anything, only the type hint is important. I.e., you don't necessarily need to call your fixtures `config` or `context`.

Step Contexts

The `StepContext` provides additional context inside a step function. It can be used to access artifacts, materializers, and stack components directly from within the step.

Defining Steps with Step Contexts

Unlike `BaseStepConfig`, you do not need to create a `StepContext` object yourself and pass it when creating the step. As long as you specify a parameter of type `StepContext` in the signature of your step function or class, ZenML will automatically create the `StepContext` and take care of passing it to your step at runtime.

When using a `StepContext` inside a step, ZenML disables caching for this step by default as the context provides access to external resources which might influence the result of your step execution. To enable caching anyway, explicitly enable it in the `@step` decorator with `@step(enable_cache=True)` or when initializing your custom step class.

Using Step Contexts

Within a step, there are many things that you can use the `StepContext` object for. For example, to access materializers, artifact locations, etc:

```python
from zenml.steps import step, StepContext


@step
def my_step(context: StepContext):
    context.get_output_materializer()  # Get materializer for a given output.
    context.get_output_artifact_uri()  # Get URI for a given output.
    context.metadata_store  # Get access to the metadata store.
```

See the API Docs for more information on which attributes and methods the `StepContext` provides.

Fetching previous runs during pipeline execution

One of the most common usecases of the `StepContext` is to query the metadata store for a list of all previous pipeline runs, e.g., in order to compare a newly trained model to models trained in previous runs, or to fetch artifacts created in different pipelines.

As an example, see the following step that uses the `StepContext` to query the metadata store while running a step, which we use to find out whether the current run produced a better model than all previous runs:

```python
from zenml.steps import step, StepContext


@step
def find_best_model(context: StepContext, test_acc: float) -> bool:
    """Step that finds if this run produced the highest `test_acc` ever"""
    highest_acc = 0

    # Inspect all past runs of `best_model_pipeline`.
    try:
        metadata_store = context.metadata_store
        prior_runs = metadata_store.get_pipeline("best_model_pipeline").runs
    except KeyError:
        print("No prior runs found.")
        prior_runs = []

    # Find the highest accuracy produced in a prior run.
    for run in prior_runs:
        # get the output of the second step
        prior_test_acc = run.get_step("evaluator").output.read()
        if prior_test_acc > highest_acc:
            highest_acc = prior_test_acc

    # Check whether the current run produced the highest accuracy or not.
    if test_acc >= highest_acc:
        print(f"This run produced the highest test accuracy: {test_acc}.")
        return True
    print(
        f"This run produced test accuracy {test_acc}, which is not the highest. "
        f"The highest previous test accuracy was {highest_acc}."
    )
    return False
```

Full Code Example

Code Example for Fetching Historic Runs

```python
import numpy as np
from sklearn.base import ClassifierMixin
from sklearn.svm import SVC

from zenml.pipelines import pipeline
from zenml.integrations.sklearn.helpers.digits import get_digits
from zenml.steps import BaseStepConfig, Output, StepContext, step


@step
def load_digits() -> Output(
    X_train=np.ndarray, X_test=np.ndarray, y_train=np.ndarray, y_test=np.ndarray
):
    """Loads the digits dataset as normal numpy arrays."""
    X_train, X_test, y_train, y_test = get_digits()
    return X_train, X_test, y_train, y_test


class SVCTrainerStepConfig(BaseStepConfig):
    """Trainer params"""

    gamma: float = 0.001


@step()
def svc_trainer(
    config: SVCTrainerStepConfig,
    X_train: np.ndarray,
    y_train: np.ndarray,
) -> ClassifierMixin:
    """Train a sklearn SVC classifier."""
    model = SVC(gamma=config.gamma)
    model.fit(X_train, y_train)
    return model


@step
def evaluator(
    X_test: np.ndarray,
    y_test: np.ndarray,
    model: ClassifierMixin,
) -> float:
    """Calculate the accuracy on the test set"""
    test_acc = model.score(X_test, y_test)
    print(f"Test accuracy: {test_acc}")
    return test_acc


@step
def find_best_model(context: StepContext, test_acc: float) -> bool:
    """Step that finds if this run produced the highest `test_acc` ever"""
    highest_acc = 0

    # Inspect all past runs of `best_model_pipeline`.
    try:
        metadata_store = context.metadata_store
        prior_runs = metadata_store.get_pipeline("best_model_pipeline").runs
    except KeyError:
        print("No prior runs found.")
        prior_runs = []

    # Find the highest accuracy produced in a prior run.
    for run in prior_runs:
        # get the output of the second step
        prior_test_acc = run.get_step("evaluator").output.read()
        if prior_test_acc > highest_acc:
            highest_acc = prior_test_acc

    # Check whether the current run produced the highest accuracy or not.
    if test_acc >= highest_acc:
        print(f"This run produced the highest test accuracy: {test_acc}.")
        return True
    print(
        f"This run produced test accuracy {test_acc}, which is not the highest."
        f"\nThe highest previous test accuracy was {highest_acc}."
    )
    return False


@pipeline
def best_model_pipeline(importer, trainer, evaluator, find_best_model):
    X_train, X_test, y_train, y_test = importer()
    model = trainer(X_train, y_train)
    test_acc = evaluator(X_test, y_test, model)
    find_best_model(test_acc)


for gamma in (0.01, 0.001, 0.0001):

    best_model_pipeline(
        importer=load_digits(),
        trainer=svc_trainer(SVCTrainerStepConfig(gamma=gamma)),
        evaluator=evaluator(),
        find_best_model=find_best_model(),
    ).run()
```

Expected Output

```
...
No prior runs found.
This run produced the highest test accuracy: 0.6974416017797553.
...
This run produced the highest test accuracy: 0.9688542825361512.
...
This run produced test accuracy 0.9399332591768632, which is not the highest.
The highest previous test accuracy was 0.9688542825361512.
...
```

Access Global Info within Steps

How to access run names and other global data from within a step

In addition to Step Fixtures, ZenML provides another interface where ZenML data can be accessed from within a step, the `Environment`, which can be used to get further information about the environment where the step is executed, such as the system it is running on, the Python version, the name of the current step, pipeline, and run, and more.

As an example, this is how you could use the `Environment` to find out the name of the current step, pipeline, and run:

```
from zenml.environment import Environment


@step
def my_step(...)
    env = Environment().step_environment
    step_name = env.step_name
    pipeline_name = env.pipeline_name
    run_id = env.pipeline_run_id
```

To explore all possible operations that can be performed via the `Environment`, please consult the API docs section on Environment.

Manage External Services

How to manage external, longer-lived services

ZenML interacts with external systems (e.g. prediction services, monitoring systems, visualization services) via a so-called `Service` abstraction. The concrete implementation of this abstraction deals with functionality concerning the life-cycle management and tracking of an external service (e.g. process, container, Kubernetes deployment etc.).

Using Services in Steps

Services can be passed through steps like any other object, and used to interact with the external systems that they represent:

```
from zenml.steps import step


@step
```

```python
def my_step(my_service: MyService) -> ...:
    if not my_service.is_running:
        my_service.start()  # starts service
    my_service.stop()  # stops service
```

Examples

One concrete example of a `Service` is the built-in `LocalDaemonService`, a service represented by a local daemon process which extends the base `Service` class with functionality concerning the life-cycle management and tracking of local daemon processes. The `LocalDaemonService` is used by various integrations to connect your local machines to remote components such as a Metadata Store in KubeFlow.

Another example is the `TensorboardService`. It enables visualizing TensorBoard logs by managing a local TensorBoard server, which couples nicely with the `TensorboardVisualizer` to visualize Tensorboard logs:

```python
from zenml.integrations.tensorflow.services.tensorboard_service import (
    TensorboardService,
    TensorboardServiceConfig
)

service = TensorboardService(
    TensorboardServiceConfig(
        logdir=logdir,
    )
)

# start the service
service.start(timeout=20)

# stop the service
service.stop()
```

You can find full examples of using services here:

- Visualizing training with TensorBoard in the Kubeflow TensorBoard example.
- Interacting with the services of deployed models in the MLflow deployment example.
- Interacting with the services of deployed models in the Seldon deployment example.

Manage Docker Images

How ZenML uses Docker images to run your pipeline

When running locally, ZenML will execute the steps of your pipeline in the active Python environment. When using a remote orchestrators or step operators instead, ZenML builds Docker images to transport and run your pipeline code in an isolated and well-defined environment. For this purpose, a Dockerfile is dynamically generated and used to build the image using the local Docker client. This Dockerfile consists of the following steps:

- Starts from a base image which needs to have ZenML installed. By default, this will use the official ZenML image for the Python and ZenML version that you're using in the active Python environment. If you want to use a different image as the base for the following steps, check out this guide.
- **Installs additional pip dependencies**. ZenML will automatically detect which integrations are used in your stack and install the required dependencies. If your pipeline needs any additional requirements, check out our guide on including custom dependencies.
- **Copies your active stack configuration**. This is needed so that ZenML can execute your code on the stack that you specified.
- **Copies your source files**. These files need to be included in the Docker image so ZenML can execute your step code. Check out this section for more information on which files get included by default and how to exclude files.

Which files get included

ZenML will try to determine the root directory of your source files in the following order:

- If you've created a ZenML repository for your project, the repository directory will be used.
- Otherwise, the parent directory of the python file you're executing will be the source root. For example, running `python /path/to/file.py`, the source root would be `/path/to`.

By default, ZenML will copy all contents of this root directory into the Docker image. If you want to exclude files to keep the image smaller, you can do so using a .dockerignore file in either of the following two ways:

- Have a file called `.dockerignore` in your source root directory explained above.
- Explicitly specify a `.dockerignore` file that you want to use:

  ```
  @pipeline(dockerignore_file="/path/to/.dockerignore")
  def my_pipeline(...):
      ...
  ```

Customizing the build process

This process explained above is all done automatically by ZenML and covers most basic use cases. This section covers all the different ways in which you can hook into the Docker building process to customize the resulting image to your needs.

How to install additional pip dependencies

> You don't need to install the `zenml` pip package as well as any integration that is used for components of your active stack.

If you want ZenML to install additional pip dependencies on top of the base image, you can use any of the following three ways:

- Specify a list of ZenML integrations that you're using in your pipeline:

```python
from zenml.integrations.constants import PYTORCH, EVIDENTLY

@pipeline(required_integrations=[PYTORCH, EVIDENTLY])
def my_pipeline(...):
    ...
```

- Specify a list of pip requirements in code:

```python
@pipeline(requirements=["torch==1.12.0", "torchvision"])
def my_pipeline(...):
    ...
```

- Specify a pip requirements file:

```python
@pipeline(requirements="/path/to/requirements.txt")
def my_pipeline(...):
    ...
```

You can even combine these methods, but do make sure that your list of pip requirements doesn't overlap with the ones specified by your required integrations.

Using a custom base image

To have full control over the environment which is used to execute your pipelines, you can specify a custom base image which will be used as the starting point of the Docker image that ZenML will use to execute your code. For more information on how to specify a base image for the orchestrator or step operator you're using, visit the corresponding documentation page.

> If you're going to use a custom base image, you need to make sure that it has Python, pip and ZenML installed for it to work.

Set Stacks and Profiles with Environment Variables

How to set stacks and profiles with environment variables

Alternatively to using Repositories, the global active profile and global active stack can be overridden by using the environment variables `ZENML_ACTIVATED_PROFILE` and `ZENML_ACTIVATED_STACK`, as shown in the following example:

```
$ zenml profile list
Running without an active repository root.
Running with active profile: 'default' (global)
┏━━━━━━━━┯━━━━━━━━━━━━━━┯━━━━━━━━━━━━┯━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┯━━━━━━━━━━━━━━┓
┃ ACTIVE │ PROFILE NAME │ STORE TYPE │ URL                                   │ ACTIVE STACK ┃
```

```
┠────────┼─────────────┼────────────┼──────────────────────────┼──────────────┨
┃       │ default     │ local      │ file:///home/stefan/.c… │ default      ┃
┃        │ zenml       │ local      │ file:///home/stefan/.c… │ custom       ┃
┗━━━━━━━━┷━━━━━━━━━━━━━┷━━━━━━━━━━━━┷━━━━━━━━━━━━━━━━━━━━━━━━━━┷━━━━━━━━━━━━━━┛

$ export ZENML_ACTIVATED_PROFILE=zenml
$ export ZENML_ACTIVATED_STACK=default

$ zenml profile list
Running without an active repository root.
Running with active profile: 'zenml' (global)
┏━━━━━━━━┯━━━━━━━━━━━━━┯━━━━━━━━━━━━┯━━━━━━━━━━━━━━━━━━━━━━━━━━┯━━━━━━━━━━━━━━┓
┃ ACTIVE │ PROFILE NAME │ STORE TYPE │ URL                     │ ACTIVE STACK ┃
┠────────┼─────────────┼────────────┼──────────────────────────┼──────────────┨
┃        │ default     │ local      │ file:///home/stefan/.c… │ default      ┃
┃       │ zenml       │ local      │ file:///home/stefan/.c… │ default      ┃
┗━━━━━━━━┷━━━━━━━━━━━━━┷━━━━━━━━━━━━┷━━━━━━━━━━━━━━━━━━━━━━━━━━┷━━━━━━━━━━━━━━┛

$ zenml stack list
Running without an active repository root.
Running with active profile: 'zenml' (global)
┏━━━━━━━━┯━━━━━━━━━━━━┯━━━━━━━━━━━━━━━━┯━━━━━━━━━━━━━━━━┯━━━━━━━━━━━━━━┓
┃ ACTIVE │ STACK NAME │ ARTIFACT_STORE │ METADATA_STORE │ ORCHESTRATOR ┃
┠────────┼────────────┼────────────────┼────────────────┼──────────────┨
┃       │ default    │ default        │ default        │ default      ┃
┃        │ custom     │ default        │ default        │ default      ┃
┗━━━━━━━━┷━━━━━━━━━━━━┷━━━━━━━━━━━━━━━━┷━━━━━━━━━━━━━━━━┷━━━━━━━━━━━━━━┛
```

MLOps Stacks

Categories of MLOps Tools

Overview of categories of MLOps tools

If you are new to the world of MLOps, it is often daunting to be immediately faced with a sea of tools that seemingly all promise and do the same things. It is useful in this case to try to categorize tools in various groups in order to understand their value in your tool chain in a more precise manner.

ZenML tackles this problem by introducing Stack that are composed of **Stack Components**. These stack component represent categories, each of which has a particular function in your MLOps pipeline. ZenML realizes these stack components as base abstractions that standardize the entire workflow for your team. In order to then realize benefit, one can write a concrete implementation of the abstraction, or use one of the many built-in integrations that implement these abstractions for you.

This is a full list of all stack components currently supported in ZenML, with a description of that components role in the MLOps process:

Type of Stack Component	Description
Orchestrator	Orchestrating the runs of your pipeline
Artifact Store	Storage for the artifacts created by your pipelines
Metadata Store	Tracking the execution of your pipelines/steps
Container Registry	Store for your containers
Secrets Manager	Centralized location for the storage of your secret
Step Operator	Execution of individual steps in specialized runtim environments
Model Deployer	Services/platforms responsible for online model serving
Feature Store	Management of your data/features
Experiment Tracker	Tracking your ML experiments
Alerter	Sending alerts through specified channels
Annotator	Labeling and annotating data

Each pipeline run that you execute with ZenML will require a **stack** and each **stack** will be required to include at least an orchestrator, an artifact store, and a metadata store. Apart from these three, the other components are optional and to be added as your pipeline evolves in MLOps maturity.

In the upcoming sections, you will learn about each stack component, its role in further detail, and how to use them in your own ZenML pipelines.

Integration Overview

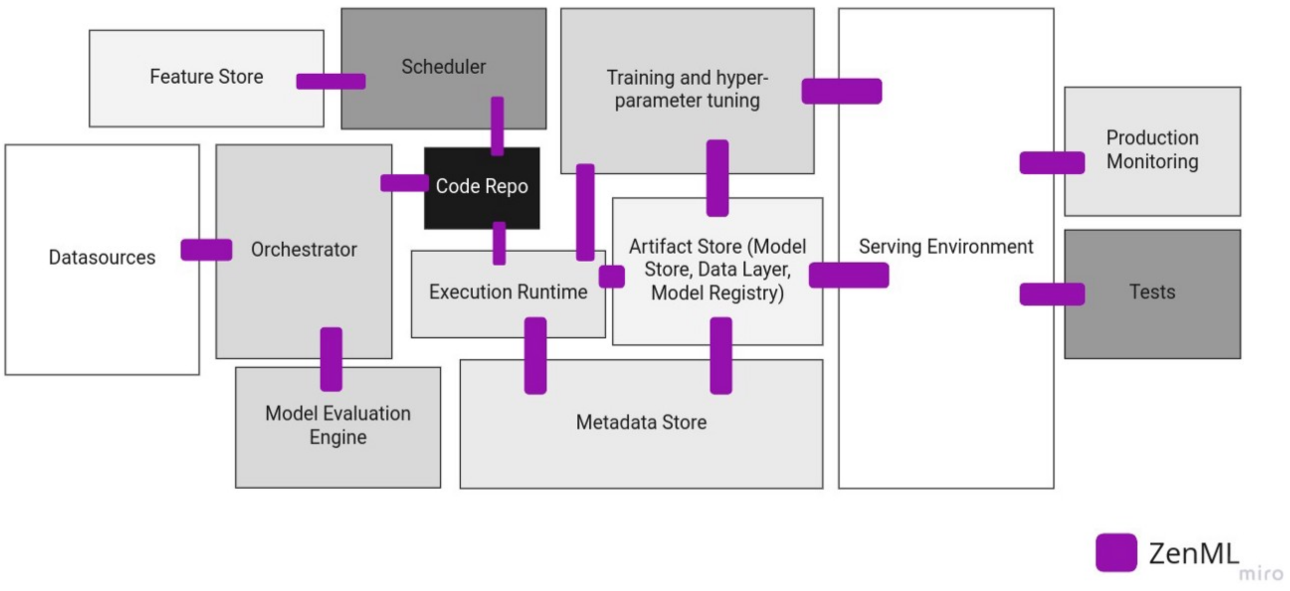
Overview of third-party ZenML integrations

Categorizing the MLOps stack is a good way to write abstractions for a MLOps pipeline and standardize your processes. But ZenML goes further and also provides concrete implementations of these categories by **integrating** with many different
tools for each category. Once code is organized into a ZenML pipeline, you can supercharge your ML workflows with the best-in-class solutions from various MLOps areas.

In short, an integration in ZenML utilizes a third-party tool to implement one or more Stack Component abstractions.

For example, you can orchestrate your ML pipeline workflows using Airflow or Kubeflow, track experiments using MLflow Tracking or Weights & Biases, and transition seamlessly from a local MLflow deployment to a deployed model on Kubernetes using Seldon Core.

There are lots of moving parts for all the MLOps tooling and infrastructure you require for ML in production and ZenML brings them all together and enables you to manage them in one place. This also allows you to delay the decision of which MLOps tool to use in your stack as you have no vendor lock-in with ZenML and can easily switch out tools as soon as your requirements change.



ZenML is the glue

Available Integrations

We have a dedicated webpage that indexes all supported ZenML integrations and their categories.

Another easy way of seeing a list of integrations is to see the list of directories in the integrations directory on our GitHub.

Installing ZenML Integrations

Before you can use integrations, you first need to install them using `zenml integration install`, e.g., you can install Kubeflow, MLflow Tracking, and Seldon Core, using:

```
zenml integration install kubeflow mlflow seldon -y
```

Under the hood, this simply installs the preferred versions of all integrations using pip, i.e., it executes in a sub-process call:

```
pip install kubeflow==<PREFERRED_VERSION> mlflow==<PREFERRED_VERSION> seldon==<PREFERRED_VI
```

The `-y` flag confirms all `pip install` commands without asking you for confirmation for every package first.

You can run `zenml integration --help` to see a full list of CLI commands that ZenML provides for interacting with integrations.

Note, that you can also install your dependencies directly, but please note that there is no guarantee that ZenML internals with work with any arbitrary version of any external library.

Help us with integrations!

There are countless tools in the ML / MLOps field. We have made an initial prioritization of which tools to support with integrations that is visible on our public roadmap.

We also welcome community contributions. Check our Contribution Guide and External Integration Guide for more details on how to best contribute new integrations.

Orchestrators

How to orchestrate ML pipelines

The orchestrator is an essential component in any MLOps stack as it is responsible for running your machine learning pipelines. To do so, the orchestrator provides an environment which is set up to execute the steps of your pipeline. It also makes sure that the steps of your pipeline only get executed once all their inputs (which are outputs of previous steps of your pipeline) are available.

Many of ZenML's remote orchestrators build Docker images in order to transport and execute your pipeline code. If you want to learn more about how Docker images are built by ZenML, check out this guide.

When to use it

The orchestrator is a mandatory component in the ZenML stack. It is used to store all artifacts produced by pipeline runs and you are required to configure it in all of your stacks.

Orchestrator Flavors

Out of the box, ZenML comes with a `local` orchestrator already part of the default stack that runs pipelines locally. Additional orchestrators are provided by integrations:

Orchestrator	Flavor	Integration	Notes

LocalOrchestrator	`local`	*built-in*	Runs your pipelines locally.
KubernetesOrchestrator	`kubernetes`	`kubernetes`	Runs your pipelines i Kubernetes clusters.
KubeflowOrchestrator	`kubeflow`	`kubeflow`	Runs your pipelines using Kubeflow.
VertexOrchestrator	`vertex`	`gcp`	Runs your pipelines i Vertex AI.
AirflowOrchestrator	`airflow`	`airflow`	Runs your pipelines locally using Airflow.
GitHubActionsOrchestrator	`github`	`github`	Runs your pipelines using GitHub Actions

If you would like to see the available flavors of orchestrators, you can use the command:

```
zenml orchestrator flavor list
```

How to use it

You don't need to directly interact with any ZenML orchestrator in your code. As long as the orchestrator that you want to use is part of your active ZenML stack, using the orchestrator is as simple as executing a python file which runs a ZenML pipeline:

```
python file_that_runs_a_zenml_pipeline.py
```

Local Orchestrator

How to orchestrate pipelines locally

The local orchestrator is an orchestrator flavor which comes built-in with ZenML and runs your pipelines locally.

When to use it

The local orchestrator is part of your default stack when you're first getting started with ZenML. Due to it running locally on your machine, it requires no additional setup and is easy to use and debug.

You should use the local orchestrator if

- you're just getting started with ZenML and want to run pipelines without setting up any cloud infrastructure.
-

- you're writing a new pipeline and want to experiment and debug quickly

How to deploy it

The local orchestrator comes with ZenML and works without any additional setup.

How to use it

To use the local orchestrator, we can register it and use it in our active stack:

```
zenml orchestrator register <NAME> --flavor=local

# Add the orchestrator to the active stack
zenml stack update -o <NAME>
```

You can now run any ZenML pipeline using the local orchestrator:

```
python file_that_runs_a_zenml_pipeline.py
```

For more information and a full list of configurable attributes of the local orchestrator, check out the API Docs.

Kubeflow Orchestrator

How to orchestrate pipelines with Kubeflow

The Kubeflow orchestrator is an orchestrator flavor provided with the ZenML `kubeflow` integration that uses Kubeflow Pipelines to run your pipelines.

When to use it

You should use the Kubeflow orchestrator if:

- you're looking for a proven production-grade orchestrator.
- you're looking for a UI in which you can track your pipeline runs.
- you're already using Kubernetes or are not afraid of setting up and maintaining a Kubernetes cluster.
- you're willing to deploy and maintain Kubeflow Pipelines on your cluster.

How to deploy it

The Kubeflow orchestrator supports two different modes: `Local` and `remote`. In case you want to run the orchestrator on a local Kubernetes cluster running on your machine, there is no additional infrastructure setup necessary.

If you want to run your pipelines on a remote cluster instead, you'll need to set up a Kubernetes cluster and deploy Kubeflow Pipelines:

AWS

- Have an existing AWS EKS cluster set up.
- Make sure you have the AWS CLI set up.
- Download and install `kubectl` and configure it to talk to your EKS cluster using the following command:

```
aws eks --region REGION update-kubeconfig --name CLUSTER_NAME
```

- Install Kubeflow Pipelines onto your cluster.

GCP

- Have an existing GCP GKE cluster set up.
- Make sure you have the Google Cloud CLI set up first.
- Download and install `kubectl` and configure it to talk to your GKE cluster using the following command:

```
gcloud container clusters get-credentials CLUSTER_NAME
```

- Install Kubeflow Pipelines onto your cluster.

Azure

- Have an existing AKS cluster set up.
- Make sure you have the `az` CLI set up first.
- Download and install `kubectl` and it to talk to your AKS cluster using the following command:

```
az aks get-credentials --resource-group RESOURCE_GROUP --name CLUSTER_NAME
```

- Install Kubeflow Pipelines onto your cluster.

> Since Kubernetes v1.19, AKS has shifted

to `containerd`

> . However, the workflow controller installed with the Kubeflow installation has `Docker` set as the default runtime. In order to make your pipelines work, you have to change the value to one of the options

listed here

> , preferably `k8sapi`.

This change has to be made by editing the `containerRuntimeExecutor` property of the `ConfigMap` corresponding to the workflow controller. Run the following commands to first know what config map to change and then to edit it to reflect your new value.

```
kubectl get configmap -n kubeflow
kubectl edit configmap CONFIGMAP_NAME -n kubeflow
# This opens up an editor that can be used to make the change.
```

If one or more of the deployments are not in the `Running` state, try increasing the number of nodes in your cluster.

If you're installing Kubeflow Pipelines manually, make sure the Kubernetes service is called exactly `ml-pipeline`. This is a requirement for ZenML to connect to your Kubeflow Pipelines deployment.

How to use it

To use the Kubeflow orchestrator, we need:

- The ZenML `kubeflow` integration installed. If you haven't done so, run

```
zenml integration install kubeflow
```

- Docker installed and running.
- kubectl installed.

Local

When using the Kubeflow orchestrator locally, you'll additionally need

- K3D installed to spin up a local Kubernetes cluster.
- A local container registry as part of your stack.

The local Kubeflow Pipelines deployment requires more than 2 GB of RAM, so if you're using Docker Desktop make sure to update the resource limits in the preferences.

We can then register the orchestrator and use it in our active stack:

```
zenml orchestrator register <NAME> \
    --flavor=kubeflow

# Add the orchestrator to the active stack
```

```
zenml stack update -o <NAME>
```

Remote

When using the Kubeflow orchestrator with a remote cluster, you'll additionally need

- Kubeflow pipelines deployed on a remote cluster. See the deployment section for more information.
- The name of your Kubernetes context which points to your remote cluster. Run `kubectl config get-contexts` to see a list of available contexts.
- A remote artifact store as part of your stack.
- A remote metadata store as part of your stack. Kubeflow Pipelines already comes with its own MySQL database that is deployed in your Kubernetes cluster. If you want to use this database as your metadata store to get started quickly, check out the corresponding documentation page. For a more production-ready setup we suggest using a MySQL metatadata store instead.
- A remote container registry as part of your stack.

We can then register the orchestrator and use it in our active stack:

```
zenml orchestrator register <NAME> \
    --flavor=kubeflow \
    --kubernetes_context=<KUBERNETES_CONTEXT>

# Add the orchestrator to the active stack
zenml stack update -o <NAME>
```

ZenML will build a Docker image called `zenml-kubeflow` which includes your code and use it to run your pipeline steps in Kubeflow. Check out this page if you want to learn more about how ZenML builds these images and how you can customize them.

If you decide you need the full flexibility of having a custom base image, you can update your existing orchestrator

```
zenml orchestrator update <NAME> \
--custom_docker_base_image_name=<IMAGE_NAME>
```

or set it when registering a new Kubeflow orchestrator:

```
zenml orchestrator register <NAME> \
--flavor=kubeflow \
--custom_docker_base_image_name=<IMAGE_NAME>
```

Once the orchestrator is part of the active stack, we need to run `zenml stack up` before running any pipelines. This command

- forwards a port so you can view the Kubeflow UI in your browser.
- (in the local case) uses K3D to provision a Kubernetes cluster on your machine and deploys Kubeflow Pipelines on it.

You can now run any ZenML pipeline using the Kubeflow orchestrator:

```
python file_that_runs_a_zenml_pipeline.py
```

A concrete example of using the Kubeflow orchestrator can be found here.

For more information and a full list of configurable attributes of the Kubeflow orchestrator, check out the API Docs.

Kubernetes Orchestrator

How to orchestrate pipelines with Kubernetes

The Kubernetes orchestrator is an orchestrator flavor provided with the ZenML `kubernetes` integration that runs your pipelines on a Kubernetes cluster.

When to use it

You should use the Kubernetes orchestrator if:

- you're looking lightweight way of running your pipelines on Kubernetes.
- you don't need a UI to list all your pipelines runs.
- you're not willing to maintain Kubeflow Pipelines on your Kubernetes cluster.
- you're not interested in paying for managed solutions like Vertex.

How to deploy it

The Kubernetes orchestrator requires a Kubernetes cluster in order to run. There are many ways to deploy a Kubernetes cluster using different cloud providers or on your custom infrastructure and we can't possibly cover all of them, but you can check out our cloud guide ZenML Cloud Guide for some complete stack deployments which use the Kubernetes orchestrator.

How to use it

To use the Kubernetes orchestrator, we need:

- The ZenML `kubernetes` integration installed. If you haven't done so, run
  ```
  zenml integration install kubernetes
  ```
- Docker installed and running.

- kubectl installed.
- A remote artifact store as part of your stack.
- A remote metadata store as part of your stack. If you want to use a MySQL database deployed in your Kubernetes cluster, you can use the Kubernetes metadata store. For a more production-ready setup we suggest using a MySQL metatadata store instead.
- A remote container registry as part of your stack.
- A Kubernetes cluster deployed and the name of your Kubernetes context which points to this cluster. Run `kubectl config get-contexts` to see a list of available contexts.

We can then register the orchestrator and use it in our active stack:

```
zenml orchestrator register <NAME> \
    --flavor=kubernetes \
    --kubernetes_context=<KUBERNETES_CONTEXT>

# Add the orchestrator to the active stack
zenml stack update -o <NAME>
```

> ZenML will build a Docker image called `zenml-kubernetes` which includes your code and use it to run your pipeline steps in Kubernetes. Check out this page if you want to learn more about how ZenML builds these images and how you can customize them.
>
> If you decide you need the full flexibility of having a custom base image, you can update your existing orchestrator
>
> ```
> zenml orchestrator update <NAME> \
> --custom_docker_base_image_name=<IMAGE_NAME>
> ```
>
> or set it when registering a new Kubernetes orchestrator:
>
> ```
> zenml orchestrator register <NAME> \
> --flavor=kubernetes \
> --custom_docker_base_image_name=<IMAGE_NAME>
> ```

You can now run any ZenML pipeline using the Kubernetes orchestrator:

```
python file_that_runs_a_zenml_pipeline.py
```

A concrete example of using the Kubernetes orchestrator can be found here.

For more information and a full list of configurable attributes of the Kubernetes orchestrator, check out the API Docs.

Google Cloud VertexAI Orchestrator

The Vertex orchestrator is an orchestrator flavor provided with the ZenML `gcp` integration that uses Vertex AI to run your pipelines.

When to use it

You should use the Vertex orchestrator if:

- you're already using GCP.
- you're looking for a proven production-grade orchestrator.
- you're looking for a UI in which you can track your pipeline runs.

- you're looking for a managed solution for running your pipelines.
- you're looking for a serverless solution for running your pipelines.

How to deploy it

Check out our cloud guide ZenML Cloud Guide for information on how to set up the Vertex orchestrator.

How to use it

To use the Vertex orchestrator, we need:

- The ZenML `gcp` integration installed. If you haven't done so, run
  ```
  zenml integration install gcp
  ```
- Docker installed and running.
- kubectl installed.
- A remote artifact store as part of your stack.
- A remote metadata store as part of your stack.
- A remote container registry as part of your stack.
- The GCP project ID and location in which you want to run your Vertex AI pipelines.

We can then register the orchestrator and use it in our active stack:

```
zenml orchestrator register <NAME> \
    --flavor=vertex \
    --project=<PROJECT_ID> \
    --location=<GCP_LOCATION>

# Add the orchestrator to the active stack
zenml stack update -o <NAME>
```

ZenML will build a Docker image called `zenml-vertex` which includes your code and use it to run your pipeline steps in Vertex AI. Check out this page if you want to learn more about how ZenML builds these images and how you can customize them.

If you decide you need the full flexibility of having a custom base image, you can update your existing orchestrator

```
zenml orchestrator update <NAME> \
--custom_docker_base_image_name=<IMAGE_NAME>
```

or set it when registering a new Vertex orchestrator:

```
zenml orchestrator register <NAME> \
--flavor=vertex \
--custom_docker_base_image_name=<IMAGE_NAME> \
...
```

You can now run any ZenML pipeline using the Vertex orchestrator:

```
python file_that_runs_a_zenml_pipeline.py
```

A concrete example of using the Vertex orchestrator can be found here.

For more information and a full list of configurable attributes of the Vertex orchestrator, check out the API Docs.

Github Actions Orchestrator

How to orchestrate pipelines with GitHub Actions

The GitHub Actions orchestrator is an orchestrator flavor provided with the ZenML `github` integration that uses GitHub Actions to run your pipelines.

When to use it

You should use the GitHub Actions orchestrator if:

- you're using GitHub for your projects.
- you're looking for a free, managed solution to run your pipelines.
- you're looking for a UI in which you can track your pipeline runs.
- your pipeline steps don't require much resources to run. The GitHub Actions orchestrator uses GitHub Actions runners to run your pipelines. These runner have access to limited hardware resources and are not able to run computationally intensive tasks.

How to deploy it

The GitHub Actions orchestrator runs on hardware provided by GitHub Actions runners and only requires you to have a GitHub account and repository.

How to use it

To use the GitHub Actions orchestrator, we need:

- The ZenML `github` integration installed. If you haven't done so, run
  ```
  zenml integration install github
  ```
- Docker installed and running.
- A remote artifact store as part of your stack.
- A remote metadata store as part of your stack.
- A GitHub container registry as part of your stack.

We can then register the orchestrator and use it in our active stack:

```
zenml orchestrator register <NAME> --flavor=github

# Add the orchestrator to the active stack
zenml stack update -o <NAME>
```

> ZenML will build a Docker image called `zenml-github-actions` which includes your code and use it to run your pipeline steps in GitHub. Check out this page if you want to learn more about how ZenML builds these images and how you can customize them.
>
> If you decide you need the full flexibility of having a custom base image, you can update your existing orchestrator
>
> ```
> zenml orchestrator update <NAME> \
> --custom_docker_base_image_name=<IMAGE_NAME>
> ```
>
> or set it when registering a new GitHub Actions orchestrator:
>
> ```
> zenml orchestrator register <NAME> \
> --flavor=kubernetes \
> --custom_docker_base_image_name=<IMAGE_NAME>
> ```

You can now run any ZenML pipeline using the GitHub Actions orchestrator:

```
python file_that_runs_a_zenml_pipeline.py
```

In contrast with our other orchestrators, this does not automatically run your pipeline. Your pipeline will only work once you push the workflow file that the orchestrator has written in the previous `python` call. If you want to automate this process and want the orchestrator to commit and run these files automatically, you can set the orchestrators `push` attribute to `True`. To do so, simply update your orchestrator:

```
zenml orchestrator update <NAME> --push=True
```

A concrete example of using the GitHub Actions orchestrator can be found here.

For more information and a full list of configurable attributes of the GitHub Actions orchestrator, check out the API Docs.

Airflow Orchestrator

How to orchestrate pipelines with Airflow

The Airflow orchestrator is an orchestrator flavor provided with the ZenML `airflow` integration that uses Airflow to run your pipelines.

When to use it

You should use the Airflow orchestrator if

- you're already using Airflow
- you want to run your pipelines locally with a more production-ready setup than the local orchestrator

If you're looking to run your pipelines in the cloud, take a look at other orchestrator flavors.

> We're currently reworking the Airflow orchestrator to make sure it works not only locally but also with Airflow instances deployed on cloud infrastructure.

How to deploy it

The Airflow orchestrator works without any additional infrastructure setup.

How to use it

To use the Airflow orchestrator, we need:

- The ZenML `airflow` integration installed. If you haven't done so, run
  ```
  zenml integration install airflow
  ```

We can then register the orchestrator and use it in our active stack:

```
zenml orchestrator register <NAME> \
    --flavor=airflow

# Add the orchestrator to the active stack
```

```
zenml stack update -o <NAME>
```

Once the orchestrator is part of the active stack, we can provision all required local resources by running:

```
zenml stack up
```

This command will start up an Airflow server on your local machine that's running in the same Python environment that you used to provision it. When it is finished, it will print a username and password which you can use to login to the Airflow UI here.

You can now run any ZenML pipeline using the Airflow orchestrator:

```
python file_that_runs_a_zenml_pipeline.py
```

A concrete example of using the Airflow orchestrator can be found here.

For more information and a full list of configurable attributes of the Airflow orchestrator, check out the API Docs.

Develop a Custom Orchestrator

How to develop a custom orchestrator

Base Implementation

ZenML aims to enable orchestration with any orchestration tool. This is where the `BaseOrchestrator` comes into play. It abstracts away many of the ZenML specific details from the actual implementation and exposes a simplified interface:

1. As it is the base class for a specific type of `StackComponent`, it inherits from the `StackComponent` class. This sets the `TYPE` variable to the specific `StackComponentType`.
2. The `FLAVOR` class variable needs to be set in the particular sub-class as it is meant to identify the implementation flavor of the particular orchestrator.
3. Lastly, the base class features one `abstractmethod`: `prepare_or_run_pipeline`. In the implementation of every `Orchestrator` flavor, it is required to define this method with respect to the flavor at hand.

Putting all these considerations together, we end up with the following (simplified) implementation:

```python
from abc import ABC, abstractmethod
from typing import ClassVar, List, Any

from tfx.proto.orchestration.pipeline_pb2 import Pipeline as Pb2Pipeline

from zenml.enums import StackComponentType
from zenml.stack import StackComponent
from zenml.steps import BaseStep


class BaseOrchestrator(StackComponent, ABC):
    """Base class for all ZenML orchestrators"""

    # --- Class variables ---
    TYPE: ClassVar[StackComponentType] = StackComponentType.ORCHESTRATOR

    @abstractmethod
    def prepare_or_run_pipeline(
            self,
            sorted_steps: List[BaseStep],
            pipeline: "BasePipeline",
            pb2_pipeline: Pb2Pipeline,
            stack: "Stack",
            runtime_configuration: "RuntimeConfiguration",
    ) -> Any:
        """Prepares and runs the pipeline outright or returns an intermediate
        pipeline representation that gets deployed.
        """
```

This is a slimmed-down version of the base implementation which aims to highlight the abstraction layer. In order to see the full implementation and get the complete docstrings, please check the source code on GitHub.

Build your own custom orchestrator

If you want to create your own custom flavor for an artifact store, you can follow the following steps:

1. Create a class which inherits from the `BaseOrchestrator`.
2. Define the `FLAVOR` class variable.
3. Implement the `prepare_or_run_pipeline()` based on your desired orchestrator.

Once you are done with the implementation, you can register it through the CLI as:

```
zenml orchestrator flavor register <THE-SOURCE-PATH-OF-YOUR-ORCHESTRATOR>
```

Some additional implementation details

Not all orchestrators are created equal. Here is a few basic categories that differentiate them.

Direct Orchestration

The implementation of a `local` orchestrator can be summarized in two lines of code:

```python
for step in sorted_steps:
    self.run_step(...)
```

The orchestrator basically iterates through each step and directly executes the step within the same Python process. Obviously all kind of additional configuration could be added around this.

Python Operator based Orchestration

The `airflow` orchestrator has a slightly more complex implementation of the `prepare_or_run_pipeline()` method. Instead of immediately executing a step, a `PythonOperator` is created which contains a `_step_callable`. This `_step_callable` will ultimately execute the `self.run_step(...)` method of the orchestrator. The PythonOperators are assembled into an AirflowDag which is returned. Through some Airflow magic, this DAG is loaded by the connected instance of Airflow and orchestration of this DAG is performed either directly or on a set schedule.

Container-based Orchestration

The `kubeflow` orchestrator is a great example of container-based orchestration. In an implementation-specific method called `prepare_pipeline_deployment()` a Docker image containing the complete project context is built.

Within `prepare_or_run_pipeline()` a yaml file is created as an intermediate representation of the pipeline and uploaded to the Kubeflow instance. To create this yaml file a callable is defined within which a `dsl.ContainerOp` is created for each step. This ContainerOp contains the container entrypoint command and arguments that will make the image run just the one step. The ContainerOps are assembled according to their interdependencies inside a `dsl.Pipeline` which can then be compiled into the yaml file.

Base Implementation of the Step Entrypoint Configuration

Within the base Docker images that are used for container-based orchestration the `src.zenml.entrypoints.step_entrypoint.py` is the default entrypoint to run a specific step. It does so by loading an orchestrator specific `StepEntrypointConfiguration` object. This object is then used to parse all entrypoint arguments (e.g. --step_source). Finally, the `StepEntrypointConfiguration.run()` method is used to execute the step. Under the hood this will eventually also call the orchestrators `run_step()` method.

The `StepEntrypointConfiguration` is the base class that already implements most of the required functionality. Let's dive right into it:

1. The `DEFAULT_SINGLE_STEP_CONTAINER_ENTRYPOINT_COMMAND` is the default entrypoint command for the Docker container.
2. Some arguments are mandatory for the step entrypoint. These are set as constants at the top of the file and used as the minimum required arguments.
3. The `run()` method uses the parsed arguments to set up all required prerequisites before ultimately

executing the step.

Here is a schematic view of what the `StepEntrypointConfiguration` looks like:

```python
from abc import ABC, abstractmethod
from typing import Any, List, Set

from zenml.steps import BaseStep

DEFAULT_SINGLE_STEP_CONTAINER_ENTRYPOINT_COMMAND = [
    "python",
    "-m",
    "zenml.entrypoints.step_entrypoint",
]
# Constants for all the ZenML default entrypoint options
ENTRYPOINT_CONFIG_SOURCE_OPTION = "entrypoint_config_source"
PIPELINE_JSON_OPTION = "pipeline_json"
MAIN_MODULE_SOURCE_OPTION = "main_module_source"
STEP_SOURCE_OPTION = "step_source"
INPUT_SPEC_OPTION = "input_spec"


class StepEntrypointConfiguration(ABC):

    # --- This has to be implemented by the subclass ---
    @abstractmethod
    def get_run_name(self, pipeline_name: str) -> str:
        """Returns the run name."""

    # --- These can be implemented by subclasses ---
    @classmethod
    def get_custom_entrypoint_options(cls) -> Set[str]:
        """Custom options for this entrypoint configuration"""
        return set()

    @classmethod
    def get_custom_entrypoint_arguments(
            cls, step: BaseStep, **kwargs: Any
    ) -> List[str]:
        """Custom arguments the entrypoint command should be called with."""
        return []

    # --- This will ultimately be called by the step entrypoint ---

    def run(self) -> None:
       """Prepares execution and runs the step that is specified by the
       passed arguments"""
       ...
```

This is a slimmed-down version of the base implementation which aims to highlight the abstraction layer. In order to see the full implementation and get the complete docstrings, please check the API docs.

Build your own Step Entrypoint Configuration

There is only one mandatory method `get_run_name(...)` that you need to implement in order to get a functioning entrypoint. Inside this method you need to return a string which **has** to be the same for all steps that are executed as part of the same pipeline run.

If you need to pass additional arguments to the entrypoint, there are two methods that you need to implement:

- `get_custom_entrypoint_options()`: This method should return all the additional options that you require in the entrypoint.
- `get_custom_entrypoint_arguments(...)`: This method should return a list of arguments that should be passed to the entrypoint. The arguments need to provide values for all options defined in the `custom_entrypoint_options()` method mentioned above.

Artifact Stores

How to set up the persistent storage for your artifacts

The Artifact Store is a central component in any MLOps stack. As the name suggests, it acts as a data persistence layer where artifacts (e.g. datasets, models) ingested or generated by the machine learning pipelines are stored.

ZenML automatically serializes and saves the data circulated through your pipelines in the Artifact Store: datasets, models, data profiles, data and model validation reports and generally any object that is returned by a pipeline step. This is coupled with tracking information saved in the Metadata Store to provide extremely useful features such as caching and provenance/lineage tracking and pipeline reproducibility.

Not all objects returned by pipeline steps are physically stored in the Artifact Store, nor do they have to be. How artifacts are serialized and deserialized and where their contents are stored is determined by the particular implementation of the Materializer associated with the artifact data type. The majority of Materializers shipped with ZenML use the Artifact Store that is part of the active Stack as the location where artifacts are kept.

If you need to store *a particular type of pipeline artifacts* in a different medium (e.g. use an external model registry to store model artifacts, or an external data lake or data warehouse to store dataset artifacts), you can write your own Materializer to implement the custom logic required for it. In contrast, if you need to use an entirely different storage backend to store artifacts, one that isn't already covered by one of the ZenML integrations, you can extend the Artifact Store abstraction to provide your own Artifact Store implementation.

In addition to pipeline artifacts, the Artifact Store may also be used as a storage backed by other specialized stack components that need to store their data in a form of persistent object storage. The Great Expectations Data Validator is such an example.

Related concepts:

- the Artifact Store is a type of Stack Component that needs to be registered as part of your ZenML Stack.
- the objects circulated through your pipelines are serialized and stored in the Artifact Store using Materializers. Materializers implement the logic required to serialize and deserialize the artifact contents and to store them and retrieve their contents to/from the Artifact Store.
- versioning and tracking for the pipeline artifacts stored in the Artifact Store is done through the Metadata Store.
- you can access the artifacts produced by your pipeline runs from the Artifact Store using the post-execution workflow API.

When to use it

The Artifact Store is a mandatory component in the ZenML stack. It is used to store all artifacts produced by pipeline runs and you are required to configure it in all of your stacks.

Artifact Store Flavors

Out of the box, ZenML comes with a `local` artifact store already part of the default stack that stores artifacts on your local filesystem. Additional Artifact Stores are provided by integrations:

Artifact Store	Flavor	Integration	URI Schema(s)	Notes
Local	`local`	*built-in*	None	This is the defau Artifact Store. It stores artifacts o your local filesystem. Should be used only for running ZenML locally.
Amazon S3	`s3`	`s3`	`s3://`	Uses AWS S3 a an object store backend
Google Cloud Storage	`gcp`	`gcp`	`gs://`	Uses Google Cloud Storage a an object store backend
Azure	`azure`	`azure`	`abfs://`, `az://`	Uses Azure Blo Storage as an

				object store
Custom Implementation	*custom*		*custom*	Extend the Artifact Store abstraction and provide your ow implementation

If you would like to see the available flavors of Artifact Stores, you can use the command:

```
zenml artifact-store flavor list
```

Every Artifact Store has a `path` attribute that must be configured when it is registered with ZenML. This is a URI pointing to the root path where all objects are stored in the Artifact Store. It must use a URI schema that is supported by the Artifact Store flavor. For example, the S3 Artifact Store will need a URI that contains the `s3://` schema:

```
zenml artifact-store register s3_store -f s3 --path s3://my_bucket
```

How to use it

The Artifact Store provides low-level object storage services for other ZenML mechanisms. When you develop ZenML pipelines, you normally don't even have to be aware of its existence or interact with it directly. ZenML provides higher-level APIs that can be used as an alternative to store and access artifacts:

- return one or more objects from your pipeline steps to have them automatically saved in the active Artifact Store as pipeline artifacts.
- use the post-execution workflow API to retrieve pipeline artifacts from the active Artifact Store after a pipeline run is complete.

You will probably need to interact with the low-level Artifact Store API directly:

- if you implement custom Materializers for your artifact data types
- if you want to store custom objects in the Artifact Store

The Artifact Store API

All ZenML Artifact Stores implement the same IO API that resembles a standard file system. This allows you to access and manipulate the objects stored in the Artifact Store in the same manner you would normally handle files on your computer and independently from the particular type of Artifact Store that is configured in your ZenML stack.

Accessing the low-level Artifact Store API can be done through the following Python modules:

- `zenml.io.fileio` provides low-level utilities for manipulating Artifact Store objects (e.g. `open`, `copy`, `rename`, `remove`, `mkdir`). These functions work seamlessly across Artifact Stores types. They have the same signature as the Artifact Store abstraction methods (in fact, they are one and the same under the hood).

- zenml.utils.io_utils includes some higher-level helper utilities that make it easier to find and transfer objects between the Artifact Store and the local filesystem or memory.

> When calling the Artifact Store API, you should always use URIs that are relative to the Artifact Store root path, otherwise you risk using an unsupported protocol or storing objects outside of the store. You can use the `Repository` singleton to retrieve the root path of the active Artifact Store and then use it as a base path for artifact URIs, e.g.:
>
> ```python
> import os
> from zenml.repository import Repository
> from zenml.io import fileio
>
> root_path = Repository().active_stack.artifact_store.path
>
> artifact_contents = "example artifact"
> artifact_path = os.path.join(root_path, "artifacts", "examples")
> artifact_uri = os.path.join(artifact_path, "test.txt")
> fileio.makedirs(artifact_path)
> with fileio.open(artifact_uri, "w") as f:
>     f.write(artifact_contents)
> ```
>
> When using the Artifact Store API to write custom Materializers, the base artifact URI path is already provided. See the documentation on Materializers for an example).

The following are some code examples showing how to use the Artifact Store API for various operations:

- creating folders, writing and reading data directly to/from an artifact store object

```python
import os
from zenml.utils import io_utils
from zenml.io import fileio

from zenml.repository import Repository

root_path = Repository().active_stack.artifact_store.path

artifact_contents = "example artifact"
artifact_path = os.path.join(root_path, "artifacts", "examples")
artifact_uri = os.path.join(artifact_path, "test.txt")
fileio.makedirs(artifact_path)
io_utils.write_file_contents_as_string(artifact_uri, artifact_contents)
```

```python
import os
from zenml.utils import io_utils
```

```python
from zenml.repository import Repository

root_path = Repository().active_stack.artifact_store.path

artifact_path = os.path.join(root_path, "artifacts", "examples")
artifact_uri = os.path.join(artifact_path, "test.txt")
artifact_contents = io_utils.read_file_contents_as_string(artifact_uri)
```

- using a temporary local file/folder to serialize and copy in-memory objects to/from the artifact store (heavily used in Materializers to transfer information between the Artifact Store and external libraries that don't support writing/reading directly to/from the artifact store backend):

```python
import os
import tempfile
import external_lib

root_path = Repository().active_stack.artifact_store.path

artifact_path = os.path.join(root_path, "artifacts", "examples")
artifact_uri = os.path.join(artifact_path, "test.json")
fileio.makedirs(artifact_path)

with tempfile.NamedTemporaryFile(
    mode="w", suffix=".json", delete=True
) as f:
    external_lib.external_object.save_to_file(f.name)
    # Copy it into artifact store
    fileio.copy(f.name, artifact_uri)
```

```python
import os
import tempfile
import external_lib

root_path = Repository().active_stack.artifact_store.path

artifact_path = os.path.join(root_path, "artifacts", "examples")
artifact_uri = os.path.join(artifact_path, "test.json")

with tempfile.NamedTemporaryFile(
    mode="w", suffix=".json", delete=True
) as f:
    # Copy the serialized object from the artifact store
    fileio.copy(artifact_uri, f.name)
    external_lib.external_object.load_from_file(f.name)
```

Local Artifact Store

How to store artifacts on your local filesystem

The local Artifact Store is a built-in ZenML Artifact Store flavor that uses a folder on your local filesystem to store artifacts.

When would you want to use it?

The local Artifact Store is a great way to get started with ZenML, as it doesn't require you to provision additional local resources, or to interact with managed object store services like Amazon S3 and Google Cloud Storage. All you need is the local filesystem. You should use the local Artifact Store if you're just evaluating or getting started with ZenML, or if you are still in the experimental phase and don't need to share your pipeline artifacts (dataset, models, etc.) with others.

> The local Artifact Store is not meant to be utilized in production. The local filesystem cannot be shared across your team and doesn't cover services like high-availability, scalability, backup and restore and other features that are expected from a production grade MLOps system.
>
> The fact that it stores artifacts on your local filesystem also means that not all stack components can be used in the same stack as a local Artifact Store:
>
> - only Orchestrators running on the local machine, such as the local Orchestrator, a local Kubeflow Orchestrator, or a local Kubernetes Orchestrator can be combined with a local Artifact Store
> - only Model Deployers that are running locally, such as the MLflow Model Deployer can only be used in combination with a local Artifact Store
> - Step Operators: none of the Step Operators can be used in the same stack as a local Artifact Store, given that their very purpose is to run ZenML steps in remote specialized environments
>
> As you transition to a team setting or a production setting, you can replace the local Artifact Store in your stack with one of the other flavors that are better suited for these purposes, with no changes required in your code.

How do you deploy it?

The `default` stack that comes pre-configured with ZenML already contains a local Artifact Store:

```
$ zenml stack list
Running without an active repository root.
Running with active profile: 'default' (global)

┏━━━━━━━━┯━━━━━━━━━━━━┯━━━━━━━━━━━━━━━━┯━━━━━━━━━━━━━━━━┯━━━━━━━━━━━━━━┓
┃ ACTIVE │ STACK NAME │ ARTIFACT_STORE │ METADATA_STORE │ ORCHESTRATOR ┃
┠────────┼────────────┼────────────────┼────────────────┼──────────────┨
┃        │ default    │ default        │ default        │ default      ┃
┗━━━━━━━━┷━━━━━━━━━━━━┷━━━━━━━━━━━━━━━━┷━━━━━━━━━━━━━━━━┷━━━━━━━━━━━━━━┛

$ zenml artifact-store describe
Running without an active repository root.
Running with active profile: 'default' (global)
Running with active stack: 'default'
No component name given; using `default` from active stack.
                           ARTIFACT_STORE Component Configuration (ACTIVE)
┏━━━━━━━━━━━━━━━━━━━━┯━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━
┃ COMPONENT_PROPERTY │ VALUE
┠────────────────────┼───────────────────────────────────────────────────────────────────────────────────────────
┃ TYPE               │ artifact_store
┠────────────────────┼───────────────────────────────────────────────────────────────────────────────────────────
┃ FLAVOR             │ local
┠────────────────────┼───────────────────────────────────────────────────────────────────────────────────────────
┃ NAME               │ default
┠────────────────────┼───────────────────────────────────────────────────────────────────────────────────────────
┃ UUID               │ 2b7773eb-d371-4f24-96f1-fad15e74fd6e
┠────────────────────┼───────────────────────────────────────────────────────────────────────────────────────────
┃ PATH               │ /home/stefan/.config/zenml/local_stores/2b7773eb-d371-4f24-96f1-fad
┗━━━━━━━━━━━━━━━━━━━━┷━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━
```

As shown by the `PATH` value in the `zenml artifact-store describe` output, the artifacts are stored inside a folder on your local filesystem.

You can create additional instances of local Artifact Stores and use them in your stacks as you see fit, e.g.:

```
# Register the local artifact store
zenml artifact-store register custom_local --flavor local

# Register and set a stack with the new artifact store
zenml stack register custom_stack -o default -m default -a custom_local --set
```

Same as all other Artifact Store flavors, the local Artifact Store does take in a `path` configuration parameter that can be set during registration to point to a custom path on your machine. However, it is highly recommended that you rely on the default `path` value, otherwise it may lead to unexpected results. Other local stack components depend on the convention used for the default path to be able to access the local Artifact Store.

For more, up-to-date information on the local Artifact Store implementation and its configuration, you can have a look at the API docs.

How do you use it?

Aside from the fact that the artifacts are stored locally, using the local Artifact Store is no different than using any other flavor of Artifact Store.

Amazon Simple Cloud Storage (S3)

How to store artifacts in an AWS S3 bucket

The S3 Artifact Store is an Artifact Store flavor provided with the S3 ZenML integration that uses the AWS

S3 managed object storage service or one of the self-hosted S3 alternatives, such as MinIO or Ceph RGW. to store artifacts in an S3 compatible object storage backend.

When would you want to use it?

Running ZenML pipelines with the local Artifact Store is usually sufficient if you just want to evaluate ZenML or get started quickly without incurring the trouble and the cost of employing cloud storage services in your stack. However, the local Artifact Store becomes insufficient or unsuitable if you have more elaborate needs for your project:

- if you want to share your pipeline run results with other team members or stakeholders inside or outside your organization
- if you have other components in your stack that are running remotely (e.g. a Kubeflow or Kubernetes Orchestrator running in public cloud).
- if you outgrow what your local machine can offer in terms of storage space and need to use some form of private or public storage service that is shared with others
- if you are running pipelines at scale and need an Artifact Store that can handle the demands of production grade MLOps

In all these cases, you need an Artifact Store that is backed by a form of public cloud or self-hosted shared object storage service.

You should use the S3 Artifact Store when you decide to keep your ZenML artifacts in a shared object storage and if you have access to the AWS S3 managed service or one of the S3 compatible alternatives (e.g. Minio, Ceph RGW). You should consider one of the other Artifact Store flavors if you don't have access to an S3 compatible service.

How do you deploy it?

The S3 Artifact Store flavor is provided by the S3 ZenML integration, you need to install it on your local machine to be able to register an S3 Artifact Store and add it to your stack:

```
zenml integration install s3 -y
```

The only configuration parameter mandatory for registering an S3 Artifact Store is the root path URI, which needs to point to an S3 bucket and takes the form `s3://bucket-name`. Please read the documentation relevant to the S3 service that you are using on how to create an S3 bucket. For example, the AWS S3 documentation is available here.

With the URI to your S3 bucket known, registering an S3 Artifact Store and using it in a stack can be done as follows:

```
# Register the S3 artifact-store
zenml artifact-store register s3_store -f s3 --path=s3://bucket-name

# Register and set a stack with the new artifact store
zenml stack register custom_stack -a s3_store ... --set
```

Depending on your use-case, however, you may also need to provide additional configuration parameters pertaining to authentication or pass advanced configuration parameters to match your S3 compatible service or deployment scenario.

> Configuring an S3 Artifact Store in can be a complex and error prone process, especially if you plan on using it alongside other stack components running in the AWS cloud. You might consider referring to the ZenML Cloud Guide for a more holistic approach to configuring full AWS-based stacks for ZenML.

Authentication Methods

Integrating and using an S3 compatible Artifact Store in your pipelines is not possible without employing some form of authentication. ZenML currently provides three options for configuring S3 credentials, the recommended one being to use a Secrets Manager in your stack to store the sensitive information in a secure location.

Implicit Authentication

This method uses the implicit AWS authentication available *in the environment where the ZenML code is running*. On your local machine, this is the quickest way to configure an S3 Artifact Store. You don't need to supply credentials explicitly when you register the S3 Artifact Store, as it leverages the local credentials and configuration that the AWS CLI stores on your local machine. However, you will need to install and set up the AWS CLI on your machine as a prerequisite, as covered in the AWS CLI documentation, before you register the S3 Artifact Store.

> The implicit authentication method needs to be coordinated with other stack components that are highly dependent on the Artifact Store and need to interact with it directly to function. If these components are not running on your machine, they do not have access to the local AWS CLI configuration and will encounter authentication failures while trying to access the S3 Artifact Store:
>
> - Orchestrators need to access the Artifact Store to manage pipeline artifacts
> - Step Operators need to access the Artifact Store to manage step level artifacts
> - Model Deployers need to access the Artifact Store to load served models
>
> These remote stack components can still use the implicit authentication method: if they are also running on Amazon EC2 or EKS nodes, ZenML will try to load credentials from the instance metadata service. In order to take advantage of this feature, you must have specified an IAM role to use when you launched your EC2 instance or EKS cluster. This mechanism allows AWS workloads like EC2 instances and EKS pods to access other AWS services without requiring explicit credentials. For more information on how to configure IAM roles:
>
> - on EC2 instances, see the IAM Roles for Amazon EC2 guide
> - on EKS clusters, see the Amazon EKS cluster IAM role guide

If you have remote stack components that are not running in AWS Cloud, or if you are unsure how to configure them to use IAM roles, you should use one of the other authentication methods.

Explicit Credentials

Use this method to store the S3 credentials in the Artifact Store configuration. When you register the S3 Artifact Store, you may include the credentials directly in the Artifact Store configuration. This has the advantage that you don't need to install and configure the AWS CLI or set up a Secrets Manager in your stack, but the credentials won't be stored securely and will be visible in the stack configuration:

```
# Register the S3 artifact-store
zenml artifact-store register s3_store -f s3 \
    --path='s3://your-bucket' \
    --key='your-S3-access-key-ID' \
    --secret='your-S3-secret-key' \
    --token='your-S3-token'

# Register and set a stack with the new artifact store
zenml stack register custom_stack -a s3_store ... --set
```

Secrets Manager (Recommended)

This method requires using a Secrets Manager in your stack to store the sensitive S3 authentication information in a secure location.

The S3 credentials are configured as a ZenML secret that is referenced in the Artifact Store configuration, e.g.:

```
# Register the S3 artifact store
zenml artifact-store register s3_store -f s3 \
    --path='s3://your-bucket' \
    --authentication_secret=s3_secret

# Register a secrets manager
zenml secrets-manager register secrets_manager \
    --flavor=<FLAVOR_OF_YOUR_CHOICE> ...

# Register and set a stack with the new artifact store and secrets manager
zenml stack register custom_stack -a s3_store -x secrets_manager ... --set

# Create the secret referenced in the artifact store
zenml secret register s3_secret -s aws \
    --aws_access_key_id='your-S3-access-key-ID' \
    --aws_secret_access_key='your-S3-secret-key' \
    --aws_session_token='your-S3-token'
```

Advanced Configuration

The S3 Artifact Store accepts a range of advanced configuration options that can be used to further customize how ZenML connects to the S3 storage service that you are using. These are accessible via the `client_kwargs`, `config_kwargs` and `s3_additional_kwargs` configuration attributes and are passed transparently to the underlying S3Fs library:

- `client_kwargs`: arguments that will be transparently passed to the botocore client. You can use it to configure parameters like `endpoint_url` and `region_name` when connecting to an S3 compatible endpoint (e.g. Minio).
- `config_kwargs`: advanced parameters passed to botocore.client.Config.
- `s3_additional_kwargs`: advanced parameters that are used when calling S3 API, typically used for things like `ServerSideEncryption` and `ACL`.

To include these advanced parameters in your Artifact Store configuration, pass them using JSON format during registration, e.g.:

```
zenml artifact-store register minio_store -f s3 \
    --path='s3://minio_bucket' \
    --authentication_secret=s3_secret
    --client_kwargs='{"endpoint_url:"http://minio.cluster.local:9000", "region_name":"us-ea
```

For more, up-to-date information on the S3 Artifact Store implementation and its configuration, you can have a look at the API docs.

How do you use it?

Aside from the fact that the artifacts are stored in an S3 compatible backend, using the S3 Artifact Store is no different than using any other flavor of Artifact Store.

Google Cloud Storage (GCS)

How to store artifacts using GCP Cloud Storage

The GCS Artifact Store is an Artifact Store flavor provided with the GCP ZenML integration that uses the Google Cloud Storage managed object storage service to store ZenML artifacts in a GCP Cloud Storage bucket.

When would you want to use it?

Running ZenML pipelines with the local Artifact Store is usually sufficient if you just want to evaluate ZenML or get started quickly without incurring the trouble and the cost of employing cloud storage services in your stack. However, the local Artifact Store becomes insufficient or unsuitable if you have more elaborate needs for your project:

- if you want to share your pipeline run results with other team members or stakeholders inside or outside your organization
- if you have other components in your stack that are running remotely (e.g. a Kubeflow or Kubernetes Orchestrator running in public cloud).
- if you outgrow what your local machine can offer in terms of storage space and need to use some form of private or public storage service that is shared with others
- if you are running pipelines at scale and need an Artifact Store that can handle the demands of production grade MLOps

In all these cases, you need an Artifact Store that is backed by a form of public cloud or self-hosted shared object storage service.

You should use the GCS Artifact Store when you decide to keep your ZenML artifacts in a shared object storage and if you have access to the Google Cloud Storage managed service. You should consider one of the other Artifact Store flavors if you don't have access to the GCP Cloud Storage service.

How do you deploy it?

The GCS Artifact Store flavor is provided by the GCP ZenML integration, you need to install it on your local machine to be able to register a GCS Artifact Store and add it to your stack:

```
zenml integration install gcp -y
```

The only configuration parameter mandatory for registering a GCS Artifact Store is the root path URI, which needs to point to a GCS bucket and takes the form `gs://bucket-name`. Please read the Google Cloud Storage documentation on how to configure a GCS bucket.

With the URI to your GCS bucket known, registering an GCS Artifact Store can be done as follows:

```
# Register the GCS artifact store
zenml artifact-store register gs_store -f gcp --path=gs://bucket-name

# Register and set a stack with the new artifact store
zenml stack register custom_stack -a gs_store ... --set
```

Depending on your use-case, however, you may also need to provide additional configuration parameters pertaining to authentication to match your deployment scenario.

> Configuring a GCS Artifact Store in can be a complex and error prone process, especially if you plan on using it alongside other stack components running in the Google cloud. You might consider referring to the ZenML Cloud Guide for a more holistic approach to configuring full GCP-based stacks for ZenML.

Authentication Methods

Integrating and using a GCS Artifact Store in your pipelines is not possible without employing some form of authentication. ZenML currently provides two options for configuring GCP credentials, the recommended one being to use a Secrets Manager in your stack to store the sensitive information in a secure location.

Implicit Authentication

This method uses the implicit GCP authentication available *in the environment where the ZenML code is running*. On your local machine, this is the quickest way to configure a GCS Artifact Store. You don't need to supply credentials explicitly when you register the GCS Artifact Store, as it leverages the local credentials and configuration that the Google Cloud CLI stores on your local machine. However, you will need to install and set up the Google Cloud CLI on your machine as a prerequisite, as covered in the Google Cloud documentation, before you register the GCS Artifact Store.

> The implicit authentication method needs to be coordinated with other stack components that are highly dependent on the Artifact Store and need to interact with it directly to function. If these components are not running on your machine, they do not have access to the local Google Cloud CLI configuration and will encounter authentication failures while trying to access the GCS Artifact Store:
>
> - Orchestrators need to access the Artifact Store to manage pipeline artifacts
> - Step Operators need to access the Artifact Store to manage step level artifacts
> - Model Deployers need to access the Artifact Store to load served models
>
> These remote stack components can still use the implicit authentication method: if they are also running within Google Kubernetes Engine, ZenML will try to load credentials from the Google compute metadata service. In order to take advantage of this feature, you must have configured a Service Account with the proper permissions or enabled Workload Identity when you launched your GKE cluster. These mechanisms allows Google workloads like GKE pods to access other Google services without requiring explicit credentials.
>
> If you have remote stack components that are not running in GKE, or if you are unsure how to configure them to use Service Accounts or Workload Identity, you should use one of the other authentication methods.

Secrets Manager (Recommended)

This method requires using a Secrets Manager in your stack to store the sensitive GCP authentication information in a secure location.

A Google access key needs to be generated using the gcloud utility or Google Cloud console, as covered here. This comes in the form of a file containing JSON credentials.

The GCP credentials are configured as a ZenML secret that is referenced in the Artifact Store configuration, e.g.:

```
# Register the GCS artifact store
zenml artifact-store register gcs_store -f gcp \
    --path='gs://your-bucket' \
    --authentication_secret=gcp_secret

# Register a secrets manager
zenml secrets-manager register secrets_manager \
    --flavor=<FLAVOR_OF_YOUR_CHOICE> ...

# Register and set a stack with the new artifact store and secrets manager
zenml stack register custom_stack -a gs_store -x secrets_manager ... --set

# Create the secret referenced in the artifact store
zenml secret register gcp_secret -s gcp \
    --token=@path/to/token/file.json
```

For more, up-to-date information on the GCS Artifact Store implementation and its configuration, you can have a look at the API docs.

How do you use it?

Aside from the fact that the artifacts are stored in GCP Cloud Storage, using the GCS Artifact Store is no different than using any other flavor of Artifact Store.

Azure Blob Storage

How to store artifacts using Azure Blob Storage

The Azure Artifact Store is an Artifact Store flavor provided with the Azure ZenML integration that uses the Azure Blob Storage managed object storage service to store ZenML artifacts in a Azure Blob Storage container.

When would you want to use it?

Running ZenML pipelines with the local Artifact Store is usually sufficient if you just want to evaluate ZenML or get started quickly without incurring the trouble and the cost of employing cloud storage services in your stack. However, the local Artifact Store becomes insufficient or unsuitable if you have more elaborate needs for your project:

- if you want to share your pipeline run results with other team members or stakeholders inside or outside your organization
- if you have other components in your stack that are running remotely (e.g. a Kubeflow or Kubernetes Orchestrator running in public cloud).
- if you outgrow what your local machine can offer in terms of storage space and need to use some form of

private or public storage service that is shared with others
- if you are running pipelines at scale and need an Artifact Store that can handle the demands of production grade MLOps

In all these cases, you need an Artifact Store that is backed by a form of public cloud or self-hosted shared object storage service.

You should use the Azure Artifact Store when you decide to keep your ZenML artifacts in a shared object storage and if you have access to the Azure Blob Storage managed service. You should consider one of the other Artifact Store flavors if you don't have access to the Azure Blob Storage service.

How do you deploy it?

The Azure Artifact Store flavor is provided by the Azure ZenML integration, you need to install it on your local machine to be able to register an Azure Artifact Store and add it to your stack:

```
zenml integration install azure -y
```

The only configuration parameter mandatory for registering an Azure Artifact Store is the root path URI, which needs to point to an Azure Blog Storage container and takes the form `az://container-name` or `abfs://container-name`. Please read the Azure Blob Storage documentation on how to configure an Azure Blob Storage container.

With the URI to your Azure Blob Storage container known, registering an Azure Artifact Store can be done as follows:

```
# Register the Azure artifact store
zenml artifact-store register az_store -f azure --path=az://container-name

# Register and set a stack with the new artifact store
zenml stack register custom_stack -a az_store ... --set
```

Depending on your use-case, however, you may also need to provide additional configuration parameters pertaining to authentication to match your deployment scenario.

> Configuring an Azure Artifact Store in can be a complex and error prone process, especially if you plan on using it alongside other stack components running in the Azure cloud. You might consider referring to the ZenML Cloud Guide for a more holistic approach to configuring full Azure-based stacks for ZenML.

Authentication Methods

Integrating and using an Azure Artifact Store in your pipelines is not possible without employing some form of authentication. ZenML currently provides two options for configuring Azure credentials, the recommended one being to use a Secrets Manager in your stack to store the sensitive information in a secure location.

You will need the following information to configure Azure credentials for ZenML, depending on which type of Azure credentials you want to use:

- an Azure connection string
- an Azure account key
- the client ID, client secret and tenant ID of the Azure service principle

For more information on how to retrieve information about your Azure Storage Account and Access Key or connection string, please refer to this Azure guide.

For information on how to configure an Azure service principle, please consult the Azure documentation.

Implicit Authentication

This method uses the implicit Azure authentication available *in the environment where the ZenML code is running*. On your local machine, this is the quickest way to configure an Azure Artifact Store. You don't need to supply credentials explicitly when you register the Azure Artifact Store, instead you have to set one of the following sets of environment variables:

- to use an Azure connection string, set `AZURE_STORAGE_CONNECTION_STRING` to your Azure Storage Key connection string
- to use an Azure account key, set `AZURE_STORAGE_ACCOUNT_NAME` to your account name and one of `AZURE_STORAGE_ACCOUNT_KEY` or `AZURE_STORAGE_SAS_TOKEN` to the Azure key value.
- to use Azure ServicePrincipal credentials, set `AZURE_STORAGE_ACCOUNT_NAME` to your account name and `AZURE_STORAGE_CLIENT_ID`, `AZURE_STORAGE_CLIENT_SECRET` and `AZURE_STORAGE_TENANT_ID` to the

The implicit authentication method needs to be coordinated with other stack components that are highly dependent on the Artifact Store and need to interact with it directly to function. If these components are not running on your machine, they do not have access to the local environment variables and will encounter authentication failures while trying to access the Azure Artifact Store:

- Orchestrators need to access the Artifact Store to manage pipeline artifacts
- Step Operators need to access the Artifact Store to manage step level artifacts
- Model Deployers need to access the Artifact Store to load served models

These remote stack components can still use the implicit authentication method: if they are also running within the Azure Kubernetes Service, you can configure your cluster to use Azure Managed Identities. This mechanism allows Azure workloads like AKS pods to access other Azure services without requiring explicit credentials.

If you have remote stack components that are not running in AKS, or if you are unsure how to configure them to use Managed Identities, you should use one of the other authentication methods.

Secrets Manager (Recommended)

This method requires using a Secrets Manager in your stack to store the sensitive Azure authentication information in a secure location and configuring the Azure credentials using a ZenML secret.

The ZenML Azure secret schema supports a variety of credentials:

- use an Azure account name and access key or SAS token (the `account_name`, `account_key` and `sas_token` attributes need to be configured in the ZenML secret)
- use an Azure connection string (configure the `connection_string` attribute in the ZenML secret)
- use Azure ServicePrincipal credentials (which requires `account_name`, `tenant_id`, `client_id` and `client_secret` to be configured in the ZenML secret)

The Azure credentials are configured as a ZenML secret that is referenced in the Artifact Store configuration, e.g.:

```
# Register the Azure artifact store
zenml artifact-store register az_store -f azure \
    --path='az://your-container' \
    --authentication_secret=az_secret

# Register a secrets manager
zenml secrets-manager register secrets_manager \
    --flavor=<FLAVOR_OF_YOUR_CHOICE> ...

# Register and set a stack with the new artifact store and secrets manager
zenml stack register custom_stack -a az_store -x secrets_manager ... --set

# Create the secret referenced in the artifact store
zenml secret register az_secret -s azure \
    --connection_sting='your-Azure-connection-string'
```

For more, up-to-date information on the Azure Artifact Store implementation and its configuration, you can have a look at the API docs.

How do you use it?

Aside from the fact that the artifacts are stored in Azure Blob Storage, using the Azure Artifact Store is no different than using any other flavor of Artifact Store.

Develop a Custom Artifact Store

How to develop a custom artifact store

ZenML comes equipped with Artifact Store implementations that you can use to store artifacts on a local filesystem or in the managed AWS, GCP or Azure cloud object storage services. However, if you need to use a different type of object storage service as a backend for your ZenML Artifact Store, you can extend ZenML to provide your own custom Artifact Store implementation.

Base Abstraction

The Artifact Store establishes one of the main components in every ZenML stack. Now, let us take a deeper dive into the fundamentals behind its abstraction, namely the `BaseArtifactStore` class:

1. As it is the base class for a specific type of StackComponent, it inherits from the StackComponent class. This sets the `TYPE` variable to a StackComponentType. The `FLAVOR` class variable needs to be set in the specific subclass.
2. As ZenML only supports filesystem-based artifact stores, it features an instance configuration parameter called `path`, which will indicate the root path of the artifact store. When creating an instance of any flavor of an `ArtifactStore`, users will have to define this parameter.
3. Moreover, there is an empty class variable called `SUPPORTED_SCHEMES` that needs to be defined in every flavor implementation. It indicates the supported filepath schemes for the corresponding implementation. For instance, for the Azure artifact store, this set will be defined as `{"abfs://", "az://"}`.
4. Lastly, the base class features a set of `abstractmethod`s: `open`, `copyfile`, `exists`, `glob`, `isdir`, `listdir`, `makedirs`, `mkdir`, `remove`, `rename`, `rmtree`, `stat`, `walk`. In the implementation of every `ArtifactStore` flavor, it is required to define these methods with respect to the flavor at hand.

Putting all these considerations together, we end up with the following implementation:

```python
from zenml.enums import StackComponentType
from zenml.stack import StackComponent

PathType = Union[bytes, str]

class BaseArtifactStore(StackComponent):
    """Base class for all ZenML artifact stores."""

    # --- Instance configuration ---
    path: str  # The root path of the artifact store.

    # --- Class variables ---
    TYPE: ClassVar[StackComponentType] = StackComponentType.ARTIFACT_STORE
    SUPPORTED_SCHEMES: ClassVar[Set[str]]

    # --- User interface ---
    @abstractmethod
    def open(self, name: PathType, mode: str = "r") -> Any:
        """Open a file at the given path."""

    @abstractmethod
    def copyfile(
        self, src: PathType, dst: PathType, overwrite: bool = False
    ) -> None:
        """Copy a file from the source to the destination."""

    @abstractmethod
    def exists(self, path: PathType) -> bool:
        """Returns `True` if the given path exists."""

    @abstractmethod
    def glob(self, pattern: PathType) -> List[PathType]:
        """Return the paths that match a glob pattern."""

    @abstractmethod
    def isdir(self, path: PathType) -> bool:
        """Returns whether the given path points to a directory."""

    @abstractmethod
    def listdir(self, path: PathType) -> List[PathType]:
        """Returns a list of files under a given directory in the filesystem."""

    @abstractmethod
    def makedirs(self, path: PathType) -> None:
        """Make a directory at the given path, recursively creating parents."""

    @abstractmethod
    def mkdir(self, path: PathType) -> None:
        """Make a directory at the given path; parent directory must exist."""

    @abstractmethod
    def remove(self, path: PathType) -> None:
        """Remove the file at the given path. Dangerous operation."""

    @abstractmethod
    def rename(
        self, src: PathType, dst: PathType, overwrite: bool = False
    ) -> None:
        """Rename source file to destination file."""

    @abstractmethod
    def rmtree(self, path: PathType) -> None:
        """Deletes dir recursively. Dangerous operation."""

    @abstractmethod
    def stat(self, path: PathType) -> Any:
        """Return the stat descriptor for a given file path."""

    @abstractmethod
    def walk(
        self,
        top: PathType,
        topdown: bool = True,
        onerror: Optional[Callable[..., None]] = None,
    ) -> Iterable[Tuple[PathType, List[PathType], List[PathType]]]:
        """Return an iterator that walks the contents of the given directory."""
```

This is a slimmed-down version of the base implementation which aims to highlight the abstraction layer. In order to see the full implementation and get the complete docstrings, please check the API docs.

The effect on the `zenml.io.fileio`

If you created an instance of an artifact store, added it to your stack and activated the stack, it will create a filesystem each time you run a ZenML pipeline and make it available to the `zenml.io.fileio` module.

This means that when you utilize a method such as `fileio.open(...)` with a filepath which starts with one of the `SUPPORTED_SCHEMES` within your steps or materializers, it will be able to use the `open(...)` method that you defined within your artifact store.

Build your own custom artifact store

If you want to implement your own custom Artifact Store, you can follow the following steps:

1. Create a class which inherits from the `BaseArtifactStore` base class.
2. Define the `FLAVOR` and `SUPPORTED_SCHEMES` class variables.
3. Implement the `abstractmethod`s based on your desired filesystem.

Once you are done with the implementation, you can register it through the CLI as:

```
zenml artifact-store flavor register <THE-SOURCE-PATH-OF-YOUR-ARTIFACT-STORE>
```

ZenML includes a range of Artifact Store implementations, some built-in and other provided by specific integration modules. You can use them as examples of how you can extend the base Artifact Store abstraction to implement your own custom Artifact Store:

Text	Flavor	Integration
LocalArtifactStore	local	`built-in`
S3ArtifactStore	s3	s3
GCPArtifactStore	gcp	gcp
AzureArtifactStore	azure	azure

Metadata Stores

Keeping a historical record of your pipeline runs is a core MLOps practice. This makes it possible to trace back the lineage or provenance of the data that was used to train a model, for example, or to go back and compare the performance of a particular model at different points in time. Features like these are becoming increasingly important in production ML to help you make informed decision about the project and to provide better visibility when something goes wrong. They are especially useful in cases where legal compliance and liability are a factor.

The Metadata Store is a central component in the MLOps stack where the pipeline runtime information is versioned and stored. The configuration of each pipeline, step and produced artifacts are all tracked within the Metadata Store.

ZenML puts a lot of emphasis on guaranteed tracking of inputs across pipeline steps. Information about every pipeline run is collected and automatically recorded in the Metadata Store: the pipeline configuration, the pipeline steps and their configuration, as well as the types of artifacts produced by pipeline step runs and the location in the Artifact Store where they are kept. This is coupled with saving the artifact contents themselves in the Artifact Store to provide extremely useful features such as caching, provenance/lineage tracking and pipeline reproducibility.

Related concepts:

- the Metadata Store is a type of Stack Component that needs to be registered as part of your ZenML Stack.
- the Metadata Store stores information about where the artifacts produced by your pipelines are kept in the Artifact Store.
- the Orchestrators are the stack components responsible for collecting the pipeline runtime information and storing it in the Metadata Store.
- you can access the information recorded about your pipeline runs in the Metadata Store using the post-execution workflow API.

When to use it

The Metadata Store is a mandatory component in the ZenML stack. It is used to keep a log of detailed information about every pipeline run and you are required to configure it in all of your stacks.

Metadata Store Flavors

Out of the box, ZenML comes with a `sqlite` Metadata Store already part of the default stack that stores metadata in a SQLite database file on your local filesystem and a `mysql` Metadata Store flavor that you can connect to a MySQL compatible database. Additional Metadata Store flavors are provided by integrations. These flavors are to be used in different contexts, but in general, we suggest to use the `mysql` flavor for most use cases:

Metadata Store	Flavor	Integration	Notes
			This is the default Metadata Store. It

SQLite	`sqlite`	*built-in*	stores metadata information in a SQLi file on your local filesystem. Should be used only for running ZenML locally.
MySQL	`mysql`	*built-in*	Connects to a MySQL compatible database service to store metadata information Suitable and recommended for mo production settings.
Kubeflow	`kubeflow`	`kubeflow`	Kubeflow deploymen include an internal metadata store, which ZenML can leverage. This flavor of Metadat Store can only to be used in combination with the Kubeflow Orchestrator.
Kubernetes	`kubernetes`	`kubernetes`	Use this Metadata Store flavor to automatically deploy MySQL database as a Kubernetes workload and use it as a Metadata Store backend. Not recommended for production settings.
Custom Implementation	*custom*		*custom*

If you would like to see the available flavors of Metadata Stores, you can use the command:

```
zenml metadata-store flavor list
```

How to use it

The Metadata Store provides low-level metadata storage services for other ZenML mechanisms. When you develop ZenML pipelines, you don't even have to be aware of its existence or interact with it directly. ZenML provides higher-level APIs that can be used as an alternative to record and access information about

- information about your pipeline step executions is automatically recorded in the Metadata Store.
- use the post-execution workflow API to retrieve information about your pipeline runs from the active Metadata Store after a pipeline run is complete.

SQLite Metadata Store

How to store ML metadata in a SQLite database on your local filesystem

The SQLite Metadata Store is a built-in ZenML Metadata Store flavor that uses a SQLite database file on your local filesystem to store ML metadata information.

When would you want to use it?

The SQLite Metadata Store is a great way to get started with ZenML, as it doesn't require you to provision additional local resources, or to deploy and manage external database services, like a Kubeflow metadata service or a MySQL database. All you need is the local filesystem. You should use the local Metadata Store if you're just evaluating or getting started with ZenML, or if you are still in the experimental phase and don't need to share your pipeline run results with others.

> The local Metadata Store is not meant to be utilized in production. The local filesystem cannot be shared across your team and doesn't cover services like high-availability, scalability, backup and restore and other features that are expected from a production grade MLOps system.
>
> The fact that it stores information on your local filesystem also means that not all Orchestrators can be used in the same stack as a local Metadata Store. Only Orchestrators running on the local machine, such as the local Orchestrator, a local Kubeflow Orchestrator, or a local Kubernetes Orchestrator can be combined with a local Metadata Store
>
> As you transition to a team setting or a production setting, you can replace the local Metadata Store in your stack with one of the other flavors that are better suited for these purposes, with no changes required in your code.

How do you deploy it?

The `default` stack that comes pre-configured with ZenML already contains a SQLite Metadata Store:

```
$ zenml stack list
Running without an active repository root.
Running with active profile: 'default' (global)
┏━━━━━━━━┯━━━━━━━━━━━━┯━━━━━━━━━━━━━━━━┯━━━━━━━━━━━━━━━━┯━━━━━━━━━━━━━━┓
┃ ACTIVE │ STACK NAME │ ARTIFACT_STORE │ METADATA_STORE │ ORCHESTRATOR ┃
┠────────┼────────────┼────────────────┼────────────────┼──────────────┨
┃        │ default    │ default        │ default        │ default      ┃
┗━━━━━━━━┷━━━━━━━━━━━━┷━━━━━━━━━━━━━━━━┷━━━━━━━━━━━━━━━━┷━━━━━━━━━━━━━━┛
```

```

$ zenml metadata-store describe
Running without an active repository root.
Running with active profile: 'default' (global)
Running with active stack: 'default'
No component name given; using `default` from active stack.
                                    METADATA_STORE Component Configuration (ACTIVE)
┏━━━━━━━━━━━━━━━━━━━━━━━━━━━┯━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━
┃ COMPONENT_PROPERTY        │ VALUE
┠───────────────────────────┼──────────────────────────────────────────────────────────────
┃ TYPE                      │ metadata_store
┠───────────────────────────┼──────────────────────────────────────────────────────────────
┃ FLAVOR                    │ sqlite
┠───────────────────────────┼──────────────────────────────────────────────────────────────
┃ NAME                      │ default
┠───────────────────────────┼──────────────────────────────────────────────────────────────
┃ UUID                      │ 71bcbd88-6862-4e9a-8667-92539109a234
┠───────────────────────────┼──────────────────────────────────────────────────────────────
┃ UPGRADE_MIGRATION_ENABLED │ True
┠───────────────────────────┼──────────────────────────────────────────────────────────────
┃ URI                       │ /home/stefan/.config/zenml/local_stores/e76755ca-663d-4919-9
┗━━━━━━━━━━━━━━━━━━━━━━━━━━━┷━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━
```

As shown by the `URI` value in the `zenml metadata-store describe` output, the Metadata Store uses a database file on your local filesystem.

You can create additional instances of SQLite Metadata Stores and use them in your stacks as you see fit, e.g.:

```
# Register the sqlite metadata store
zenml metadata-store register custom_sqlite --flavor sqlite

# Register and set a stack with the new metadata store
zenml stack register custom_stack -o default -a default -m custom_sqlite --set
```

> ⓘ The SQLite Metadata Store takes in a `uri` configuration parameter that can be set during registration to point to a custom path on your machine. However, it is highly recommended that you rely on the default `uri` value, otherwise it may lead to unexpected results. Other local stack components depend on the convention used for the default path to be able to access the local Metadata Store.

For more, up-to-date information on the SQLite Metadata Store implementation and its configuration, you can have a look at the API docs.

How do you use it?

Aside from the fact that the metadata information is stored locally, using the SQLite Metadata Store is no different than using any other flavor of Metadata Store.

MySQL Metadata Store

How to store ML metadata in a MySQL database

The MySQL Metadata Store is a built-in ZenML Metadata Store flavor that connects to a MySQL compatible service to store metadata information.

When would you want to use it?

Running ZenML pipelines with the default SQLite Metadata Store is usually sufficient if you just want to evaluate ZenML or get started quickly without incurring the trouble and the cost of managing additional services like a self-hosted MySQL database or one of the managed cloud SQL database services. However, the local SQLite Metadata Store becomes insufficient or unsuitable if you have more elaborate needs for your project:

- if you want to share your pipeline run results with other team members or stakeholders inside or outside your organization
- if you have other components in your stack that are running remotely (e.g. a Kubeflow or Kubernetes Orchestrator running in public cloud).
- if you are running pipelines at scale and need a Metadata Store that can handle the demands of production grade MLOps

In all these cases, you need a Metadata Store that is backed by a form of public cloud or self-hosted MySQL database service.

You should use the MySQL Metadata Store when you need a shared, scalable and performant Metadata Store and if you have access to a MySQL database service. The database should ideally be accessible both from your local machine and the Orchestrator that you use in your ZenML stack. You should consider one of the other Metadata Store flavors if you don't have access to a MySQL compatible database service.

How do you deploy it?

Using the MySQL Metadata Store in your stack assumes that you already have access to a MySQL database service. This can be an on-premise MySQL database that you deployed explicitly for ZenML, or that you share with other services in your team or organization. It can also be a managed MySQL compatible database service deployed in the cloud.

> Configuring and deploying a managed MySQL database cloud service to use with ZenML can be a complex and error prone process, especially if you plan on using it alongside other stack components running in the cloud. You might consider referring to the ZenML Cloud Guide for a more holistic approach to configuring full cloud stacks for ZenML.

In order to connect to your MySQL instance, there are a few parameters that you will need to configure and register a MySQL Metadata Store stack component. In all cases you will need to set the following fields:

- `host` - The hostname or public IP address of your MySQL instance. This needs to be reachable from your Orchestrator and, ideally, also from your local machine.
- `port` - The port at which to reach the MySQL instance (default is 3306)
- `database` - The name of the database that will be used by ZenML to store metadata information

Additional authentication related parameters need to be configured differently depending on the chosen authentication method.

Authentication Methods

Basic Authentication

This option configures the username and password directly as stack component attributes.

> This is not recommended for production settings as the password is clearly accessible on your machine in clear text and communication with the database is unencrypted.

```
# Register the mysql metadata store
zenml metadata-store register mysql_metadata_store --flavor=mysql \
    --host=<database-host> --port=<port> --database=<database-name> \
    --username=<database-user> --password=<database-password>

# Register and set a stack with the new metadata store
zenml stack register custom_stack -m mysql_metadata_store ... --set
```

Secrets Manager (Recommended)

This method requires you to include a Secrets Manager in your stack and configure a ZenML secret to store the username and password credentials securely.

It is strongly recommended to also enable the use of SSL connections in your database service and configure SSL related parameters:

- `ssl_ca` - The SSL CA server certificate associated with your MySQL server.
- `ssl_cert` and `ssl_key` - SSL client certificate and private key. Only required if you set up client certificates for the MySQL service.

The MySQL credentials are configured as a ZenML secret that is referenced in the Metadata Store configuration, e.g.:

```
# Register the MySQL metadata store
zenml metadata-store register mysql_metadata_store --flavor=mysql \
    --host=<database-host> --port=<port> --database=<database-name> \
    --secret=mysql_secret
```

```

# Register a secrets manager
zenml secrets-manager register secrets_manager \
    --flavor=<FLAVOR_OF_YOUR_CHOICE> ...

# Register and set a stack with the new metadata store and secret manager
zenml stack register custom_stack -m mysql_metadata_store ... \
    -x secrets_manager --set

# Create the secret referenced in the metadata store
zenml secret register mysql_secret --schema=mysql \
    --user=<database-user> --password=<database-password> \
    --ssl_ca=@path/to/server/ca/certificate \
    --ssl_cert=@path/to/client/certificate \
    --ssl_key=@path/to/client/certificate/key
```

For more, up-to-date information on the MySQL Metadata Store implementation and its configuration, you can have a look at the API docs.

How do you use it?

Aside from the fact that the metadata information is stored in a MySQL database, using the MySQL Metadata Store is no different than using any other flavor of Metadata Store.

Kubeflow Metadata Store

How to store ML metadata using the Kubeflow ML metadata service

Kubeflow deployments include a ML metadata service that is compatible with ZenML. The Kubeflow Metadata Store is a Metadata Store flavor provided with the Kubeflow ZenML integration that uses the Kubeflow ML metadata service to store metadata information.

When would you want to use it?

The Kubeflow Metadata Store can only be used in conjunction with the Kubeflow Orchestrator.

You should use the Kubeflow Metadata Store if you already use a Kubeflow Orchestrator in your stack and if you wish to reuse the Kubeflow ML metadata service as a backend for your Metadata Store. This is a convenient way to avoid having to deploy and manage additional services such as a MySQL database for your Metadata Store. However, this will put additional strain on your Kubeflow ML metadata service and you may have to resize it accordingly.

You should consider one of the other Metadata Store flavors if you don't use a Kubeflow Orchestrator in your stack, or if you wish to use an alternative more suited for production.

How do you deploy it?

Using the Kubeflow Metadata Store in your stack assumes that you already have a Kubeflow Orchestrator included in the same stack. With that prerequisite in check, configuring the Kubeflow Metadata Store is pretty straightforward, e.g.:

```
# Register the Kubeflow orchestrator
zenml orchestrator register kubeflow_orchestrator --flavor=kubeflow \
    --kubernetes_context=<kubernetes_context>

# Register the Kubeflow metadata store
zenml metadata-store register kubeflow_metadata_store \
    --flavor=kubeflow

# Register and set a stack with the new metadata store and orchestrator
zenml stack register kubeflow_stack -m kubeflow_metadata_store \
    -o kubeflow_metadata_store ... --set

# Run `zenml stack up` to forward the metadata store port locally
zenml stack up
```

The `zenml stack up` command must be run locally to forward the metadata store port locally.

For more, up-to-date information on the Kubeflow Metadata Store implementation and its configuration, you can have a look at the API docs.

How do you use it?

Aside from the fact that the metadata information is stored using a Kubeflow ML metadata service, using the Kubeflow Metadata Store is no different than using any other flavor of Metadata Store.

You can also check out our examples pages for working examples that use the Kubeflow Metadata Store in their stacks:

- Pipeline Orchestration with Kubeflow

Kubernetes Metadata Store

How to store ML metadata in MySQL database deployed on Kubernetes

The Kubernetes Metadata Store is a spin off the MySQL Metadata Store provided with the Kubernetes ZenML integration. In addition to functioning like a regular MySQL Metadata Store, it is also able to automatically provision a MySQL database service on top of a Kubernetes cluster and connect to it.

When would you want to use it?

You should use the Kubernetes Metadata Store if you have access to a Kubernetes cluster and are looking for a quick and convenient way to provision and configure a Metadata Store that can be used to share your pipeline information with other members of your team or organization. A quick way to get you started using

ZenML in a collaborative setting, if you have access to a Kubernetes cluster, is to use the Kubernetes

The Kubernetes Metadata Store is not suited for production use because it lacks the scalability, performance and maintainability required from a production MLOps stack.

You should consider one of the other Metadata Store flavors if you don't have access to a Kubernetes cluster or if you wish to use an alternative that is better suited for production.

How do you deploy it?

The Kubernetes Metadata Store flavor is provided by the Kubernetes ZenML integration, you need to install it on your local machine to be able to register a Kubernetes Metadata Store and add it to your stack:

```
zenml integration install kubernetes -y
```

This guide also assumes you have already installed the `kubectl` utility on your machine and have configured access to a Kubernetes cluster using a kubectl config context. With the kubectl configuration context name on hand, registering a Kubernetes Metadata Store and using it in a stack can be done as follows:

```
# Register the Kubernetes metadata store
zenml metadata-store register k8s_metadata_store --flavor=kubernetes \
    --deployment_name=mysql \
    --kubernetes_context==<kubernetes-context-name> \
    --kubernetes_namespace=zenml

# Register and set a stack with the new metadata store
zenml stack register k8s_stack -m k8s_metadata_store  ... --set

# Run `zenml stack up` to deploy the MySQL database service and forward the
# MySQL port locally
zenml stack up
```

The `zenml stack up` command must be run locally to deploy the MySQL database service in the remote Kubernetes cluster and to forward the MySQL service port locally.

For more, up-to-date information on the Kubernetes Metadata Store implementation and its configuration, you can have a look at the API docs.

How do you use it?

Aside from the fact that the metadata information is stored using a MySQL database deployed in a Kubernetes cluster, using the Kubernetes Metadata Store is no different than using any other flavor of Metadata Store.

You can also check out our examples pages for working examples that use the Kubernetes Metadata Store in their stacks:

- Pipeline Orchestration on Kubernetes

Develop a Custom Metadata Store

How to develop a custom metadata store

Base abstraction in progress!

We are actively working on the base abstraction for the Metadata Stores, which will be available soon. As a result, their extension is not recommended at the moment. When you are selecting a metadata store for your stack, you can use one of the existing flavors.

If you need to implement your own Metadata Store flavor, you can still do so, but keep in mind that you may have to refactor it when the base abstraction is released.

ZenML comes equipped with Metadata Store implementations that you can use to store artifacts on a local filesystem, in a MySQL database or in the metadata service installed with Kubeflow. However, if you need to use a different type of service as a backend for your ZenML Metadata Store, you can extend ZenML to provide your own custom Metadata Store implementation.

Build your own custom metadata store

If you want to implement your own custom Metadata Store, you can follow the following steps:

1. Create a class which inherits from the `BaseMetadataStore` class.
2. Define the `FLAVOR` class variable.
3. Implement the `abstractmethod`s based on your desired metadata store connection.

Once you are done with the implementation, you can register it through the CLI as:

```
zenml metadata-store flavor register <THE-SOURCE-PATH-OF-YOUR-METADATA-STORE>
```

ZenML includes a range of Metadata Store implementations, some built-in and other provided by specific integration modules. You can use them as examples of how you can extend the base Metadata Store class to implement your own custom Metadata Store:

Text	Flavor	Integration
SQLiteMetadataStore	sqlite	`built-in`
MySQLMetadataStore	mysql	`built-in`
KubeflowMetadataStore	kubeflow	kubeflow
KubernetesMetadataStore	kubernetes	kubernetes

Container Registries

How to set up the storage for Docker images

The container registry is an essential part in most remote MLOps stacks. It is used to store container images that are built to run machine learning pipelines in remote environments. Containerization of the pipeline code creates a portable environment which allows code to run in an isolated manner.

When to use it

The container registry is needed whenever other components of your stack need to push or pull container images. Currently, this is the case for most of ZenML's remote orchestrators, step operators and some model deployers. These containerize your pipeline code and therefore require a container registry to store the resulting Docker images. Take a look at the documentation page of the component you want to use in your stack to see if it requires a container registry or even a specific container registry flavor.

Container Registry Flavors

ZenML comes with a few container registry flavors that you can use:

- Default flavor: Allows any URI without validation. Use this if you want to use a local container registry or when using a remote container registry that is not covered by other flavors.
- Specific flavors: Validates your container registry URI and performs additional checks to ensure you're able to push to the registry.

> We highly suggest using the specific container registry flavors in favor of the `default` one to make use of the additional URI validations.

Container Registry	Flavor	Integration	URI example
DefaultContainerRegistry	`default`	*built-in*	-
DockerHubContainerRegistry	`dockerhub`	*built-in*	docker.io/zenml
GCPContainerRegistry	`gcp`	*built-in*	gcr.io/zenml
AzureContainerRegistry	`azure`	*built-in*	zenml.azurecr.io
GitHubContainerRegistry	`github`	*built-in*	ghcr.io/zenml

AWSContainerRegistry	`aws`	`aws`	123456789.dkr.ecr.us east- 1.amazonaws.com

If you would like to see the available flavors of container registries, you can use the command:

```
zenml container-registry flavor list
```

Default Container Registry

How to store container images

The Default container registry is a container registry flavor which comes built-in with ZenML and allows container registry URIs of any format.

When to use it

You should use the Default container registry if you want to use a **local** container registry or when using a remote container registry that is not covered by other container registry flavors.

Local registry URI format

To specify a URI for a local container registry, use the following format:

```
localhost:<PORT>

# Examples:
localhost:5000
localhost:8000
localhost:9999
```

How to use it

To use the Default container registry, we need:

- Docker installed and running.
- The registry URI. If you're using a local container registry, check out the previous section on the URI format.

We can then register the container registry and use it in our active stack:

```
zenml container-registry register <NAME> \
    --flavor=default \
    --uri=<REGISTRY_URI>

```

```
# Add the container registry to the active stack
zenml stack update -c <NAME>
```

For more information and a full list of configurable attributes of the Default container registry, check out the API Docs.

DockerHub

How to store container images in DockerHub

The DockerHub container registry is a container registry flavor which comes built-in with ZenML and uses DockerHub to store container images.

When to use it

You should use the DockerHub container registry if:

- one or more components of your stack need to pull or push container images.
- you have a DockerHub account. If you're not using DockerHub, take a look at the other container registry flavors.

How to deploy it

To use the DockerHub container registry, all you need to do is create a DockerHub account.

When this container registry is used in a ZenML stack, the Docker images that are built will be published in a **public** repository and everyone will be able to pull your images. If you want to use a **private** repository instead, you'll have to create a private repository on the website before running the pipeline. The repository name depends on the remote orchestrator or step operator that you're using in your stack.

How to find the registry URI

The DockerHub container registry URI should have one of the two following formats:

```
<ACCOUNT_NAME>
# or
docker.io/<ACCOUNT_NAME>

# Examples:
zenml
my-username
docker.io/zenml
docker.io/my-username
```

To figure our the URI for your registry:

-

- Find out the account name of your DockerHub account.
- Use the account name to fill the template `docker.io/<ACCOUNT_NAME>` and get your URI.

How to use it

To use the Azure container registry, we need:

- Docker installed and running.
- The registry URI. Check out the previous section on the URI format and how to get the URI for your registry.

We can then register the container registry and use it in our active stack:

```
zenml container-registry register <NAME> \
    --flavor=dockerhub \
    --uri=<REGISTRY_URI>

# Add the container registry to the active stack
zenml stack update -c <NAME>
```

Additionally, we'll need to login to the container registry so Docker can pull and push images. This will require your DockerHub account name and either your password or preferably a personal access token.

```
docker login
```

For more information and a full list of configurable attributes of the Azure container registry, check out the API Docs.

Amazon Elastic Container Registry (ECR)

How to store container images in Amazon ECR

The AWS container registry is a container registry flavor provided with the ZenML `aws` integration and uses Amazon ECR to store container images.

When to use it

You should use the AWS container registry if:

- one or more components of your stack need to pull or push container images.
- you have access to AWS ECR. If you're not using AWS, take a look at the other container registry flavors.

How to deploy it

The ECR registry is automatically activated once you create an AWS account. However, you'll need to create a `Repository` in order to push container images to it:

- Go to the ECR website.
- Make sure the correct region is selected on the top right.
- Click on `Create repository`.
- Create a private repository. The name of the repository depends on the [orchestrator](../orchestrators/orchestrators.md or step operator you're using in your stack.

URI format

The AWS container registry URI should have the following format:

```
<ACCOUNT_ID>.dkr.ecr.<REGION>.amazonaws.com
# Examples:
123456789.dkr.ecr.eu-west-2.amazonaws.com
987654321.dkr.ecr.ap-south-1.amazonaws.com
135792468.dkr.ecr.af-south-1.amazonaws.com
```

To figure our the URI for your registry:

- Go to the AWS console and click on your user account in the top right to see the `Account ID`.
- Go here and choose the region in which you would like to store your container images. Make sure to choose a nearby region for faster access.
- Once you have both these values, fill in the values in this template `<ACCOUNT_ID>.dkr.ecr.<REGION>.amazonaws.com` to get your container registry URI.

How to use it

To use the AWS container registry, we need:

- The ZenML `aws` integration installed. If you haven't done so, run

  ```
  zenml integration install aws
  ```
- Docker installed and running.
- The AWS CLI installed and authenticated.
- The registry URI. Check out the previous section on the URI format and how to get the URI for your registry.

We can then register the container registry and use it in our active stack:

```
zenml container-registry register <NAME> \
    --flavor=aws \
    --uri=<REGISTRY_URI>

```

```
# Add the container registry to the active stack
zenml stack update -c <NAME>
```

Additionally, we'll need to login to the container registry so Docker can pull and push images:

```
# Fill your REGISTRY_URI and REGION in the placeholders in the following command.
# You can find the REGION as part of your REGISTRY_URI: `<ACCOUNT_ID>.dkr.ecr.<REGION>.ama
aws ecr get-login-password --region <REGION> | docker login --username AWS --password-stdi
```

For more information and a full list of configurable attributes of the AWS container registry, check out the API Docs.

Google Cloud Container Registry

How to store container images in GCP

The GCP container registry is a container registry flavor which comes built-in with ZenML and uses the Google Artifact Registry or the Google Container Registry to store container images.

When to use it

You should use the GCP container registry if:

- one or more components of your stack need to pull or push container images.
- you have access to GCP. If you're not using GCP, take a look at the other container registry flavors.

How to deploy it

Google Container Registry

When using the Google Container Registry, all you need to do is enabling it here.

Google Artifact Registry

When using the Google Artifact Registry, you need to:

- enable it here
- go here and create a `Docker` repository.

How to find the registry URI

Google Container Registry

When using the Google Container Registry, the GCP container registry URI should have one of the following formats:

```
gcr.io/<PROJECT_ID>
# or
us.gcr.io/<PROJECT_ID>
# or
eu.gcr.io/<PROJECT_ID>
# or
asia.gcr.io/<PROJECT_ID>

# Examples:
gcr.io/zenml
us.gcr.io/my-project
asia.gcr.io/another-project
```

To figure our the URI for your registry:

- Go to the GCP console.
- Click on the dropdown menu in the top left to get a list of available projects with their names and IDs.
- Use the ID of the project you want to use fill the template `gcr.io/<PROJECT_ID>` and get your URI (You can also use the other prefixes `<us/eu/asia>.gcr.io` as explained above if you want your images stored in a different region).

Google Artifact Registry

When using the Google Artifact Registry, the GCP container registry URI should have the following format:

```
<REGION>-docker.pkg.dev/<PROJECT_ID>/<REPOSITORY_NAME>

# Examples:
europe-west1-docker.pkg.dev/zenml/my-repo
southamerica-east1-docker.pkg.dev/zenml/zenml-test
asia-docker.pkg.dev/my-project/another-repo
```

To figure our the URI for your registry:

- Go here and select the repository that you want to uses to store Docker images. If you don't have a repository yet, take a look at the deployment section.
- On the top, click the copy button to copy the full repository URL.

How to use it

To use the Azure container registry, we need:

- Docker installed and running.
- The GCP CLI installed and authenticated.
- The registry URI. Check out the previous section on the URI format and how to get the URI for your registry.

We can then register the container registry and use it in our active stack:

```
zenml container-registry register <NAME> \
    --flavor=gcp \
    --uri=<REGISTRY_URI>

# Add the container registry to the active stack
zenml stack update -c <NAME>
```

Additionally, we'll need to configure Docker so it can pull and push images:

Google Container Registry

```
gcloud auth configure-docker
```

Google Artifact Registry

```
gcloud auth configure-docker <REGION>-docker.pkg.dev
```

For more information and a full list of configurable attributes of the GCP container registry, check out the API Docs.

Azure Container Registry

How to store container images in Azure

The Azure container registry is a container registry flavor which comes built-in with ZenML and uses the Azure Container Registry to store container images.

When to use it

You should use the Azure container registry if:

- one or more components of your stack need to pull or push container images.
- you have access to Azure. If you're not using Azure, take a look at the other container registry flavors.

How to deploy it

Go here and choose a subscription, resource group, location and registry name. Then click on `Review + Create` and to create your container registry.

How to find the registry URI

The Azure container registry URI should have the following format:

```
<REGISTRY_NAME>.azurecr.io
# Examples:
zenmlregistry.azurecr.io
myregistry.azurecr.io
```

To figure our the URI for your registry:

- Go to the Azure portal.
- In the search bar, enter `container registries` and select the container registry you want to use. If you don't have any container registries yet, check out the deployment section on how to create one.
- Use the name of your registry to fill the template `<REGISTRY_NAME>.azurecr.io` and get your URI.

How to use it

To use the Azure container registry, we need:

- Docker installed and running.
- The Azure CLI installed and authenticated.
- The registry URI. Check out the previous section on the URI format and how to get the URI for your registry.

We can then register the container registry and use it in our active stack:

```
zenml container-registry register <NAME> \
    --flavor=azure \
    --uri=<REGISTRY_URI>

# Add the container registry to the active stack
zenml stack update -c <NAME>
```

Additionally, we'll need to login to the container registry so Docker can pull and push images:

```
# Fill your REGISTRY_NAME in the placeholder in the following command.
# You can find the REGISTRY_NAME as part of your registry URI: `<REGISTRY_NAME>.azurecr.io
az acr login --name=<REGISTRY_NAME>
```

For more information and a full list of configurable attributes of the Azure container registry, check out the API Docs.

GitHub Container Registry

How to store container images in GitHub

The GitHub container registry is a container registry flavor which comes built-in with ZenML and uses the GitHub Container Registry to store container images.

When to use it

You should use the GitHub container registry if:

- one or more components of your stack need to pull or push container images.
- you're using GitHub for your projects. If you're not using GitHub, take a look at the other container registry flavors.

How to deploy it

The GitHub container registry is enabled by default when you create a GitHub account.

How to find the registry URI

The GitHub container registry URI should have the following format:

```
ghcr.io/<USER_OR_ORGANIZATION_NAME>

# Examples:
ghcr.io/zenml
ghcr.io/my-username
ghcr.io/my-organization
```

To figure our the URI for your registry:

- Use the GitHub user or organization name to fill the template `ghcr.io/<USER_OR_ORGANIZATION_NAME>` and get your URI.

How to use it

To use the GitHub container registry, we need:

- Docker installed and running.
- The registry URI. Check out the previous section on the URI format and how to get the URI for your registry.
- Our Docker client configured so it can pull and push images. Follow this guide to create a personal access token and login to the container registry.

We can then register the container registry and use it in our active stack:

```
zenml container-registry register <NAME> \
    --flavor=github \
    --uri=<REGISTRY_URI>

# Add the container registry to the active stack
zenml stack update -c <NAME>
```

For more information and a full list of configurable attributes of the GitHub container registry, check out the API Docs.

Develop a Custom Container Registry

How to develop a custom container registry

Base Abstraction

In the current version of ZenML, container registries have a rather basic base abstraction. In essence, each container registry features a `uri` as an instance configuration and a non-abstract `prepare_image_push` method for validation.

```python
from typing import ClassVar

from zenml.enums import StackComponentType
from zenml.stack import StackComponent
from zenml.utils import docker_utils


class BaseContainerRegistry(StackComponent):
    """Base class for all ZenML container registries."""

    # Instance configuration
    uri: str

    # Class variables
    TYPE: ClassVar[StackComponentType] = StackComponentType.CONTAINER_REGISTRY
    def prepare_image_push(self, image_name: str) -> None:
        """Conduct necessary checks/preparations before an image gets pushed."""

    def push_image(self, image_name: str) -> None:
        """Pushes a docker image."""
        if not image_name.startswith(self.uri):
            raise ValueError(
                f"Docker image `{image_name}` does not belong to container "
                f"registry `{self.uri}`."
            )

        self.prepare_image_push(image_name)
        docker_utils.push_docker_image(image_name)
```

This is a slimmed-down version of the base implementation which aims to highlight the abstraction layer. In order to see the full implementation and get the complete docstrings, please check the API docs.

Building your own container registry

If you want to create your own custom flavor for a container registry, you can follow the following steps:

1. Create a class which inherits from the `BaseContainerRegistry`.
2. Define the `FLAVOR` class variable.
3. If you need to execute any checks/validation before the image gets pushed, you can define these operations in the `prepare_image_push` method. As an example, you can check the `AWSContainerRegistry`.
4. Once the `prepare_image_push` gets completed, the `push_image` method will come into play and utilize a `DockerClient` object to push the image.

Once you are done with the implementation, you can register it through the CLI as:

```
zenml container-registry flavor register <THE-SOURCE-PATH-OF-YOUR-REGISTRY>
```

Secrets Managers

How to set up storage for secrets

Secrets managers provide a secure way of storing confidential information that is needed to run your ML pipelines. Most production pipelines will run on cloud infrastructure and therefore need credentials to authenticate with those services. Instead of storing these credentials in code or files, ZenML secrets managers can be used to store and retrieve these values in a secure manner.

When to use it

You should include a secrets manager in your ZenML stack if any other component of your stack requires confidential information (such as authentication credentials) or you want to access secret values inside your pipeline steps.

Secrets Manager Flavors

Out of the box, ZenML comes with a `local` secrets manager that stores secrets in local files. Additional cloud secrets managers are provided by integrations:

Secrets Manager	Flavor	Integration	Notes
Local	`local`	*built-in*	Uses local files to sto secrets
AWS	`aws`	`aws`	Uses AWS to store secrets
GCP	`gcp_secrets_manager`	`gcp`	Uses GCP to store secretes
Azure	`azure_key_vault`	`azure`	Uses Azure Key Vaul to store secrets
HashiCorp Vault	`vault`	`vault`	Uses HashiCorp Vau to store secrets
Custom Implementation	*custom*		Extend the secrets manager abstraction and provide your owr implementation

If you would like to see the available flavors of secrets managers, you can use the command:

```
zenml secrets-manager flavor list
```

How to use it

In the CLI

A full guide on using the CLI interface to register, access, update and delete secrets is available here.

A ZenML secret is a grouping of key-value pairs which are defined by a schema. An AWS SecretSchema, for example, has key-value pairs for `AWS_ACCESS_KEY_ID` and `AWS_SECRET_ACCESS_KEY` as well as an optional `AWS_SESSION_TOKEN`. If you don't specify a schema when registering a secret, ZenML will use the `ArbitrarySecretSchema`, a schema where arbitrary keys are allowed.

Note that there are two ways you can register or update your secrets. If you wish to do so interactively, passing the secret name in as an argument (as in the following example) will initiate an interactive process:

```
zenml secret register SECRET_NAME -i
```

If you wish to specify key-value pairs using command line arguments, you can do so instead:

```
zenml secret register SECRET_NAME --key1=value1 --key2=value2
```

For secret values that are too big to pass as a command line argument, or have special characters, you can also use the special `@` syntax to indicate to ZenML that the value needs to be read from a file:

```
zenml secret register SECRET_NAME --attr_from_literal=value \
   --attr_from_file=@path/to/file.txt ...
```

In a ZenML Step

You can access the secrets manager directly from within your steps through the `StepContext`. This allows you to use your secrets for querying APIs from within your step without hard-coding your access keys. Don't forget to make the appropriate decision regarding caching as it will be disabled by default when the `StepContext` is passed into the step.

```python
from zenml.steps import step, StepContext


@step(enable_cache=True)
def secret_loader(
    context: StepContext,
) -> None:
    """Load the example secret from the secret manager."""
    # Load Secret from active secret manager. This will fail if no secret
    # manager is active or if that secret does not exist.
    retrieved_secret = context.stack.secrets_manager.get_secret(<SECRET_NAME>)

    # retrieved_secret.content will contain a dictionary with all Key-Value
    # pairs within your secret.
    return
```

This will only work if the environment that your ochestrator uses to execute steps has access to the secrets manager. For example a local secrets manager will not work in combination with a remote orchestrator.

Secret Schemas

The concept of secret schemas exists to support strongly typed secrets that validate which keys can be configured for a given secret and which values are allowed for those keys.

Secret schemas are available as builtin schemas, or loaded when an integration is installed. Custom schemas can also be defined by sub-classing the `zenml.secret.BaseSecretSchema` class. For example, the following is the builtin schema defined for the MySQL Metadata Store secrets:

```python
from typing import ClassVar, Optional

from zenml.secret.base_secret import BaseSecretSchema

MYSQL_METADATA_STORE_SCHEMA_TYPE = "mysql"

class MYSQLSecretSchema(BaseSecretSchema):
    TYPE: ClassVar[str] = MYSQL_METADATA_STORE_SCHEMA_TYPE

    user: Optional[str]
    password: Optional[str]
    ssl_ca: Optional[str]
    ssl_cert: Optional[str]
    ssl_key: Optional[str]
    ssl_verify_server_cert: Optional[bool] = False
```

To register a secret regulated by a schema, the `--schema` argument must be passed to the `zenml secret register` command:

```
zenml secret register mysql_secret --schema=mysql --user=user --password=password
--ssl_ca=@./ca.pem --ssl_verify_server_cert=true
```

The keys and values passed to the CLI are validated using regular Pydantic rules:

- optional attributes don't need to be passed to the CLI and will be set to their default value if omitted
- required attributes must be passed to the CLI or an error will be raised
- all values must be a valid string representation of the data type indicated in the schema (i.e. that can be converted to the type indicated) or an error will be raised

Local Secrets Manager

How to store secrets locally

The local secrets manager is a secrets manager flavor which comes built-in with ZenML and uses the local filesystem to store secrets.

The local secrets manager is built for early local development and should not be used it a production setting. It stores your secrets without encryption and only works in combination with the local orchestrator.

How to deploy it

The local secrets manager comes with ZenML and works without any additional setup.

How to use it

To use the local secrets manager, we can register it and use it in our active stack:

```
zenml secrets-manager register <NAME> --flavor=local

# Add the secrets manager to the active stack
zenml stack update -x <NAME>
```

You can now register, update or delete secrets using the CLI or fetch secret values inside your steps.

For more information and a full list of configurable attributes of the local secrets manager, check out the API Docs.

AWS Secrets Manager

How to store secrets in AWS

The AWS secrets manager is a secrets manager flavor provided with the ZenML `aws` integration that uses AWS to store secrets.

When to use it

You should use the AWS secrets manager if:

- a component of your stack requires a secret for authentication or you want to use secrets inside your steps.
- you're already using AWS, especially if your orchestrator is running in AWS. If you're using a different cloud provider, take a look at the other secrets manager flavors.

How to deploy it

The AWS secrets manager is automatically activated once you create an AWS account.

How to use it

To use the AWS secrets manager, we need:

-

The ZenML `aws` integration installed. If you haven't done so, run

```
zenml integration install aws
```

- The AWS CLI installed and authenticated.
- A region in which you want to store your secrets. Choose one from the list here.

We can then register the secrets manager and use it in our active stack:

```
zenml secrets-manager register <NAME> \
    --flavor=aws \
    --region_name=<REGION>

# Add the secrets manager to the active stack
zenml stack update -x <NAME>
```

You can now register, update or delete secrets using the CLI or fetch secret values inside your steps.

A concrete example of using the AWS secrets manager can be found here.

For more information and a full list of configurable attributes of the AWS secrets manager, check out the API Docs.

Google Cloud Secrets Manager

How to store secrets in GCP

The GCP secrets manager is a secrets manager flavor provided with the ZenML `gcp` integration that uses GCP to store secrets.

When to use it

You should use the GCP secrets manager if:

- a component of your stack requires a secret for authentication or you want to use secrets inside your steps.
- you're already using GCP, especially if your orchestrator is running in GCP. If you're using a different cloud provider, take a look at the other secrets manager flavors.

How to deploy it

In order to use the GCP secrets manager, you need to enable it here.

How to use it

To use the GCP secrets manager, we need:

- The ZenML `gcp` integration installed. If you haven't done so, run

  ```
  zenml integration install gcp
  ```
- The GCP CLI installed and authenticated.
- The ID of the project in which you want to store secrets. Follow this guide to find your project ID.

We can then register the secrets manager and use it in our active stack:

```
zenml secrets-manager register <NAME> \
    --flavor=gcp_secrets_manager \
    --project_id=<PROJECT_ID>

# Add the secrets manager to the active stack
zenml stack update -x <NAME>
```

You can now register, update or delete secrets using the CLI or fetch secret values inside your steps.

A concrete example of using the GCP secrets manager can be found here.

For more information and a full list of configurable attributes of the GCP secrets manager, check out the API Docs.

Azure Secrets Manager

How to store secrets in Azure

The Azure secrets manager is a secrets manager flavor provided with the ZenML `azure` integration that uses Azure Key Vault to store secrets.

When to use it

You should use the Azure secrets manager if:

- a component of your stack requires a secret for authentication or you want to use secrets inside your steps.
- you're already using Azure, especially if your orchestrator is running in Azure. If you're using a different cloud provider, take a look at the other secrets manager flavors.

How to deploy it

- Go to the Azure portal.
- In the search bar, enter `key vaults` and open up the corresponding service.
- Click on `+ Create` in the top left.
- Fill in all values and create the key vault.

How to use it

To use the Azure secrets manager, we need:

- The ZenML `azure` integration installed. If you haven't done so, run
  ```
  zenml integration install azure
  ```
- The Azure CLI installed and authenticated.
- The name of the key vault to use. You can find a list of your key vaults by going to the Azure portal and searching for `key vaults`. If you don't have any key vault yet, follow the deployment guide to create one.

We can then register the secrets manager and use it in our active stack:

```
zenml secrets-manager register <NAME> \
    --flavor=azure_key_vault \
    --key_vault_name=<KEY_VAULT_NAME>

# Add the secrets manager to the active stack
zenml stack update -x <NAME>
```

You can now register, update or delete secrets using the CLI or fetch secret values inside your steps.

A concrete example of using the Azure secrets manager can be found here.

For more information and a full list of configurable attributes of the Azure secrets manager, check out the API Docs.

Github Secrets Manager

How to store secrets in GitHub

The GitHub secrets manager is a secrets manager flavor provided with the ZenML `github` integration that uses GitHub secrets to store secrets.

When to use it

> The GitHub secrets manager does not allow reading secret values unless it's running inside a GitHub Actions workflow. For this reason, this secrets manager **only** works in combination with a GitHub Actions orchestrator.

How to deploy it

GitHub secrets are automatically enabled when creating a GitHub repository.

How to use it

To use the GitHub secrets manager, we need:

- The ZenML `github` integration installed. If you haven't done so, run
  ```
  zenml integration install github
  ```
- A personal access token to authenticate with the GitHub API. Follow this guide to create one and make sure to give it the `repo` scope.
- Our GitHub username and the personal access token set as environment variables:
  ```
  export GITHUB_USERNAME=<GITHUB_USERNAME>
  export GITHUB_AUTHENTICATION_TOKEN=<PERSONAL_ACCESS_TOKEN>
  ```
- The owner and name of the repository that we want to add secrets to.

We can then register the secrets manager and use it in our active stack:

```
zenml secrets-manager register <NAME> \
    --flavor=github \
    --owner=<OWNER> \
    --repository=<REPOSITORY>

# Add the secrets manager to the active stack
zenml stack update -x <NAME>
```

You can now register, update or delete secrets using the CLI or fetch secret values inside your steps.

A concrete example of using the GitHub secrets manager can be found here.

For more information and a full list of configurable attributes of the GitHub secrets manager, check out the API Docs.

HashiCorp Vault Secrets Manager

How to store secrets in HashiCorp Vault

The HashiCorp Vault secrets manager is a secrets manager flavor provided with the ZenML `vault` integration that uses HashiCorp Vault to store secrets.

When to use it

You should use the HashiCorp Vault secrets manager if:

- a component of your stack requires a secret for authentication or you want to use secrets inside your steps.
- you're already using HashiCorp Vault to store your secrets or want a self-hosted secrets solution.

How to deploy it

To get started with this secrets manager, you need to either:

- self-host a Vault server
- register for the managed HashiCorp Cloud Platform Vault

Once you decided and finished setting up one of the two solutions, you need to enable the KV Secrets Engine - Version 2.

How to use it

To use the Vault secrets manager, we need:

- The ZenML `vault` integration installed. If you haven't done so, run
  ```
  zenml integration install vault
  ```
- The Vault server URL and KV Secrets Engine v2 endpoint.
- A client token to authenticate with the Vault server. Follow this tutorial to generate one.

We can then register the secrets manager and use it in our active stack:

```
zenml secrets-manager register <NAME> \
    --flavor=vault \
    --url=<VAULT_SERVER_URL> \
    --token=<VAULT_TOKEN> \
    --mount_point=<PATH_TO_KV_V2_ENGINE>

# Add the secrets manager to the active stack
zenml stack update -x <NAME>
```

You can now register, update or delete secrets using the CLI or fetch secret values inside your steps.

A concrete example of using the HashiCorp Vault secrets manager can be found here.

For more information and a full list of configurable attributes of the HashiCorp Vault secrets manager, check out the API Docs.

Develop a Custom Secrets Manager

How to develop a custom secrets manager

Base Abstraction

The secret manager acts as the one-stop shop for all the secrets to which your pipeline or stack components might need access.

1. As it is the base class for a specific type of `StackComponent`, it inherits from the StackComponent class. This sets the `TYPE` variable to the secrets manager . The `FLAVOR` class variable needs to be set in the specific subclass.
2. The `BaseSecretsManager` implements the interface for a set of CRUD operations as `abstractmethod`s: `register_secret`, `get_secret`, `get_all_secret_keys`, `update_secret`, `delete_secret`, `delete_all_secrets`. In the implementation of every `SecretsManager` flavor, it is required to define these methods with respect to the flavor at hand.

```python
from abc import ABC, abstractmethod
from typing import ClassVar, List

from zenml.enums import StackComponentType
from zenml.secret.base_secret import BaseSecretSchema
from zenml.stack import StackComponent

class BaseSecretsManager(StackComponent, ABC):
    """Base class for all ZenML secrets managers."""

    # Class configuration
    TYPE: ClassVar[StackComponentType] = StackComponentType.SECRETS_MANAGER
    FLAVOR: ClassVar[str]

    @abstractmethod
    def register_secret(self, secret: BaseSecretSchema) -> None:
        """Registers a new secret."""

    @abstractmethod
    def get_secret(self, secret_name: str) -> BaseSecretSchema:
        """Gets the value of a secret."""

    @abstractmethod
    def get_all_secret_keys(self) -> List[str]:
        """Get all secret keys."""

    @abstractmethod
    def update_secret(self, secret: BaseSecretSchema) -> None:
        """Update an existing secret."""

    @abstractmethod
    def delete_secret(self, secret_name: str) -> None:
        """Delete an existing secret."""

    @abstractmethod
    def delete_all_secrets(self) -> None:
        """Delete all existing secrets."""
```

ⓘ This is a slimmed-down version of the base implementation which aims to highlight the

> abstraction layer. In order to see the full implementation and get the complete docstrings, please check the API docs.

Build your own custom secrets manager

If you want to create your own custom flavor for a secrets manager, you can follow the following steps:

1. Create a class which inherits from the `BaseSecretsManager`.
2. Define the `FLAVOR` class variable.
3. Implement the `abstactmethod`s based on the API of your desired secrets manager

Once you are done with the implementation, you can register it through the CLI as:

```
zenml secrets-manager flavor register <THE-SOURCE-PATH-OF-YOUR-SECRETS-MANAGER>
```

Some additional implementation details

Different providers in the space of secrets manager have different definitions of what constitutes a secret. While some providers consider a single key-value pair a secret: (`'secret_name': 'secret_value'`), other providers have a slightly different definition. For them, a secret is a collection of key-value pairs: `{'some_username': 'user_name_1', 'some_pwd': '1234'}`.

ZenML falls into the second category. The implementation of the different methods should reflect this convention. In case the specific implementation interfaces with a secrets manager that uses the other definition of a secret, working with tags can be helpful. See the `GCPSecretsManager` for inspiration.

SecretSchemas

One way that ZenML expands on the notion of secrets as dictionaries is the secret schema. A secret schema allows the user to create and use a specific template. A schema could, for example, require the combination of a username, password and token. All schemas must sub-class from the `BaseSecretSchema`.

1. All Secret Schemas will need to have a defined `TYPE`.

2. The required and optional keys of the secret need to be defined as class variables.

```python
class BaseSecretSchema(BaseModel, ABC):
    name: str
    TYPE: ClassVar[str]

    @property
    def content(self) -> Dict[str, Any]:
        ...
```

One such schema could look like this.

```python
from typing import ClassVar, Optional

from zenml.secret import register_secret_schema_class
from zenml.secret.base_secret import BaseSecretSchema

EXAMPLE_SECRET_SCHEMA_TYPE = "example"

@register_secret_schema_class
class ExampleSecretSchema(BaseSecretSchema):

    TYPE: ClassVar[str] = EXAMPLE_SECRET_SCHEMA_TYPE

    username: str
    password: str
    token: Optional[str]
```

Experiment Trackers

How to log and visualize ML experiments

Experiment trackers let you track your ML experiments by logging extended information about your models, datasets, metrics and other parameters and allowing you to browse them, visualize them and compare them between runs. In the ZenML world, every pipeline run is considered an experiment, and ZenML facilitates the storage of experiment results through Experiment Tracker stack components. This establishes a clear link between pipeline runs and experiments.

Related concepts:

- the Experiment Tracker is an optional type of Stack Component that needs to be registered as part of your ZenML Stack.
- ZenML already includes versioning and tracking for the pipeline artifacts by storing artifacts in the Artifact Store and maintaining a record of pipeline executions through the Metadata Store.

When to use it

ZenML already records information about the artifacts circulated through your pipelines by means of the mandatory Artifact Store and Metadata Store stack components.

However, these ZenML mechanisms are meant to be used programmatically and can be more difficult to work with without a visual interface.

Experiment Trackers on the other hand are tools designed with usability in mind. They include extensive UI's providing users with an interactive and intuitive interface that allows them to browse and visualize the information logged during the ML pipeline runs.

You should add an Experiment Tracker to your ZenML stack and use it when you want to augment ZenML

Experiment Tracker Flavors

Experiment Trackers are optional stack components provided by integrations:

Experiment Tracker	Flavor	Integration	Notes
MLflow	`mlflow`	`mlflow`	Add MLflow experiment tracking and visualization capabilities to your ZenML pipelines
Wights & Biases	`wandb`	`wandb`	Add Weights & Biase experiment tracking and visualization capabilities to your ZenML pipelines
Custom Implementation	*custom*		*custom*

If you would like to see the available flavors of Experiment Tracker, you can use the command:

```
zenml experiment-tracker flavor list
```

How to use it

Every Experiment Tracker has different capabilities and use a different way of logging information from your pipeline steps, but it generally works as follows:

- first, you have to configure and add an Experiment Tracker to your ZenML stack
- next, you have to explicitly enable the Experiment Tracker for individual steps in your pipeline by decorating them with the included decorator
- in your steps, you have to explicitly log information (e.g. models, metrics, data) to the Experiment Tracker same as you would if you were using the tool independently of ZenML
- finally, you can access the Experiment Tracker UI to browse and visualize the information logged during your pipeline runs

Consult the documentation for the particular Experiment Tracker flavor that you plan on using or are using in your stack for detailed information about how to use it in your ZenML pipelines.

MLflow

The MLflow Experiment Tracker is an Experiment Tracker flavor provided with the MLflow ZenML integration that uses the MLflow tracking service to log and visualize information from your pipeline steps (e.g. models, parameters, metrics).

When would you want to use it?

MLflow Tracking is a very popular tool that you would normally use in the iterative ML experimentation phase to track and visualize experiment results. That doesn't mean that it cannot be repurposed to track and visualize the results produced by your automated pipeline runs, as you make the transition towards a more production oriented workflow.

You should use the MLflow Experiment Tracker:

- if you have already been using MLflow to track experiment results for your project and would like to continue doing so as you are incorporating MLOps workflows and best practices in your project through ZenML.
- if you are looking for a more visually interactive way of navigating the results produced from your ZenML pipeline runs (e.g. models, metrics, datasets)
- if you or your team already have a shared MLflow Tracking service deployed somewhere on-premise or in the cloud, and you would like to connect ZenML to it to share the artifacts and metrics logged by your pipelines

You should consider one of the other Experiment Tracker flavors if you have never worked with MLflow before and would rather use another experiment tracking tool that you are more familiar with.

How do you deploy it?

The MLflow Experiment Tracker flavor is provided by the MLflow ZenML integration, you need to install it on your local machine to be able to register an MLflow Experiment Tracker and add it to your stack:

```
zenml integration install mlflow -y
```

The MLflow Experiment Tracker can be configured to accommodate the following MLflow deployment scenarios:

- Scenario 1: This scenario requires that you use a local Artifact Store alongside the MLflow Experiment Tracker in your ZenML stack. The local Artifact Store comes with limitations regarding what other types of components you can use in the same stack. This scenario should only be used to run ZenML locally and is not suitable for collaborative and production settings. No parameters need to be supplied when configuring the MLflow Experiment Tracker, e.g:

```
# Register the MLflow experiment tracker
zenml experiment-tracker register mlflow_experiment_tracker --flavor=mlflow

# Register and set a stack with the new experiment tracker
```

```
zenml stack register custom_stack -e mlflow_experiment_tracker ... --set
```

- Scenario 5: This scenario assumes that you have already deployed an MLflow Tracking Server enabled with proxied artifact storage access. There is no restriction regarding what other types of components it can be combined with. This option requires authentication related parameters to be configured for the MLflow Experiment Tracker.

Authentication Methods

You need to configure the following credentials for authentication to a remote MLflow tracking server:

- `tracking_uri`: The URL pointing to the MLflow tracking server.
- `tracking_username`: Username for authenticating with the MLflow tracking server.
- `tracking_password`: Password for authenticating with the MLflow tracking server.
- `tracking_token`: Token for authenticating with the MLflow tracking server.
- `tracking_insecure_tls`: Set to skip verifying the MLflow tracking server SSL certificate.

Either `tracking_token` or `tracking_username` and `tracking_password` must be specified.

Basic Authentication

This option configures the credentials for the MLflow tracking service directly as stack component attributes.

> This is not recommended for production settings as the credentials won't be stored securely and will be clearly visible in the stack configuration.

```
# Register the MLflow experiment tracker
zenml experiment-tracker register mlflow_experiment_tracker --flavor=mlflow \
    --tracking_uri=<URI> --tracking_token=<token>

# Register and set a stack with the new experiment tracker
zenml stack register custom_stack -e mlflow_experiment_tracker ... --set
```

Secrets Manager (Recommended)

This method requires you to include a Secrets Manager in your stack and configure a ZenML secret to store the MLflow tracking service credentials securely.

> **This method is not yet supported!**
>
> We are actively working on adding Secrets Manager support to the MLflow Experiment Tracker.

For more, up-to-date information on the MLflow Experiment Tracker implementation and its configuration, you can have a look at the API docs.

How do you use it?

To be able to log information from a ZenML pipeline step using the MLflow Experiment Tracker component in the active stack, you need to use the `enable_mlflow` step decorator on all pipeline steps where you plan on doing that. Then use MLflow's logging or auto-logging capabilities as you would normally do, e.g.:

```python
from zenml.integrations.mlflow.mlflow_step_decorator import enable_mlflow
import mlflow

# Define the step and enable mlflow - order of decorators is important here
@enable_mlflow
@step
def tf_trainer(
    x_train: np.ndarray,
    y_train: np.ndarray,
) -> tf.keras.Model:
    """Train a neural net from scratch to recognize MNIST digits return our
    model or the learner"""

    # compile model

    mlflow.tensorflow.autolog()

    # train model


    # log additional information to MLflow explicitly if needed

    mlflow.log_param(...)
    mlflow.log_metric(...)
    mlflow.log_artifact(...)

    return model
```

The `enable_mlflow` decorator accepts additional parameters. An especially useful argument is `nested=True`. When supplied, it will log the parameters, metrics and artifacts of each step into nested runs, e.g.:

```python
from zenml.integrations.mlflow.mlflow_step_decorator import enable_mlflow
import mlflow

@enable_mlflow(nested=True)
@step
def step_one(
    data: np.ndarray,
) -> np.ndarray:
    ...

@enable_mlflow(nested=True)
@step
def step_two(
    data: np.ndarray,
) -> np.ndarray:
    ...
```

You can also check out our examples pages for working examples that use the MLflow Experiment Tracker in their stacks:

- Track Experiments with MLflow

Weights & Biases

How to log and visualize experiments with Weights & Biases

The Weights & Biases Experiment Tracker is an Experiment Tracker flavor provided with the Weights & Biases ZenML integration that uses the Weights & Biases experiment tracking platform to log and visualize information from your pipeline steps (e.g. models, parameters, metrics).

When would you want to use it?

Weights & Biases is a very popular platform that you would normally use in the iterative ML experimentation phase to track and visualize experiment results. That doesn't mean that it cannot be repurposed to track and visualize the results produced by your automated pipeline runs, as you make the transition towards a more production oriented workflow.

You should use the Weights & Biases Experiment Tracker:

- if you have already been using Weights & Biases to track experiment results for your project and would like to continue doing so as you are incorporating MLOps workflows and best practices in your project through ZenML.
- if you are looking for a more visually interactive way of navigating the results produced from your ZenML pipeline runs (e.g. models, metrics, datasets)
- if you would like to connect ZenML to Weights & Biases to share the artifacts and metrics logged by your pipelines with your team, organization or external stakeholders

You should consider one of the other Experiment Tracker flavors if you have never worked with Weights & Biases before and would rather use another experiment tracking tool that you are more familiar with.

How do you deploy it?

The Weights & Biases Experiment Tracker flavor is provided by the MLflow ZenML integration, you need to install it on your local machine to be able to register a Weights & Biases Experiment Tracker and add it to your stack:

```
zenml integration install wandb -y
```

The Weights & Biases Experiment Tracker needs to be configured with the credentials required to connect to the Weights & Biases platform using one of the available authentication methods.

Authentication Methods

You need to configure the following credentials for authentication to the Weights & Biases platform:

- `api_key`: Mandatory API key token of your Weights & Biases account.
- `project_name`: The name of the project where you're sending the new run. If the project is not specified, the run is put in an "Uncategorized" project.
- `entity`: An entity is a username or team name where you're sending runs. This entity must exist before you can send runs there, so make sure to create your account or team in the UI before starting to log runs. If you don't specify an entity, the run will be sent to your default entity, which is usually your username.

Basic Authentication

This option configures the credentials for the Weights & Biases platform directly as stack component attributes.

> This is not recommended for production settings as the credentials won't be stored securely and will be clearly visible in the stack configuration.

```
# Register the Weights & Biases experiment tracker
zenml experiment-tracker register wandb_experiment_tracker --flavor=wandb \
    --entity=<entity> --project_name=<project_name> --api_key=<key>

# Register and set a stack with the new experiment tracker
zenml stack register custom_stack -e wandb_experiment_tracker ... --set
```

Secrets Manager (Recommended)

This method requires you to include a Secrets Manager in your stack and configure a ZenML secret to store the Weights & Biases credentials securely.

> **This method is not yet supported!**
>
> We are actively working on adding Secrets Manager support to the Weights & Biases

Experiment Tracker.

For more, up-to-date information on the Weights & Biases Experiment Tracker implementation and its configuration, you can have a look at the API docs.

How do you use it?

To be able to log information from a ZenML pipeline step using the Weights & Biases Experiment Tracker component in the active stack, you need to use the `enable_wandb` step decorator on all pipeline steps where you plan on doing that. Then use the Weights & Biases logging or auto-logging capabilities as you would normally do, e.g.:

```python
import wandb
from wandb.integration.keras import WandbCallback
from zenml.integrations.wandb.wandb_step_decorator import enable_wandb

@enable_wandb
@step
def tf_trainer(
    config: TrainerConfig,
    x_train: np.ndarray,
    y_train: np.ndarray,
    x_val: np.ndarray,
    y_val: np.ndarray,
) -> tf.keras.Model:

    ...

    model.fit(
        x_train,
        y_train,
        epochs=config.epochs,
        validation_data=(x_val, y_val),
        callbacks=[
            WandbCallback(
                log_evaluation=True,
                validation_steps=16,
                validation_data=(x_val, y_val),
            )
        ],
    )

    ...
```

ZenML allows you to override the wandb.Settings class in the `enable_wandb` decorator to allow for even further control of the Weights & Biases integration. One feature that is super useful is to enable `magic=True`, like so:

```python
import wandb

@enable_wandb(wandb.Settings(magic=True))
@step
def my_step(
        x_test: np.ndarray,
        y_test: np.ndarray,
        model: tf.keras.Model,
) -> float:
    """Everything in this step is autologged"""
    ...
```

Doing the above auto-magically logs all the data, metrics, and results within the step, no further action required!

You can also check out our examples pages for working examples that use the Weights & Biases Experiment Tracker in their stacks:

- Track Experiments with Weights & Biases

Develop a Custom Experiment Tracker

How to develop a custom experiment tracker

Base abstraction in progress!

We are actively working on the base abstraction for the Experiment Tracker, which will be available soon. As a result, their extension is not recommended at the moment. When you are selecting an Experiment Tracker for your stack, you can use one of the existing flavors.

If you need to implement your own Experiment Tracker flavor, you can still do so, but keep in mind that you may have to refactor it when the base abstraction is released.

Build your own custom experiment tracker

If you want to implement your own custom Experiment Tracker, you can follow the following steps:

1. Create a class which inherits from the `BaseExperimentTracker` class.
2. Define the `FLAVOR` class variable.

Once you are done with the implementation, you can register it through the CLI as:

```
zenml experiment-tracker flavor register <THE-SOURCE-PATH-OF-YOUR-EXPERIMENT-TRACKER>
```

ZenML includes a range of Experiment Tracker implementations provided by specific integration modules. You can use them as examples of how you can extend the base Experiment Tracker class to implement your own custom Experiment Tracker:

Text	Flavor	Integration
MLFlowExperimentTracker	mlflow	mlflow
WandbExperimentTracker	wandb	wandb

Model Deployers

How to deploy your models and serve real-time predictions

Model Deployment is the process of making a machine learning model available to make predictions and decisions on real world data. Getting predictions from trained models can be done in different ways depending on the use-case, a batch prediction is used to generate prediction for a large amount of data at once, while a real-time prediction is used to generate predictions for a single data point at a time.

Model deployers are stack components responsible for serving models on a real-time or batch basis.

Online serving is the process of hosting and loading machine-learning models as part of a managed web service and providing access to the models through an API endpoint like HTTP or GRPC. Once deployed, model inference can be triggered at any time and you can send inference requests to the model through the web service's API and receive fast, low-latency responses.

Batch inference or offline inference is the process of making a machine learning model make predictions on a batch observations. This is useful for generating predictions for a large amount of data at once. The predictions are usually stored as files or in a database for end users or business applications.

Custom pre-processing and post-processing

- Pre-processing is the process of transforming the data before it is passed to the machine learning model.
- Post-processing is the process of transforming the data after it is returned from the machine learning model and before it is returned to the user.

Both pre- and post-processing are very essential to the model deployment process, since majority of the models require specific input format which requires transforming the data before it is passed to the model and after it is returned from the model. ZenML is allowing you to define your own pre- and post-processing within a pipeline level by defining a custom steps before and after the predict step.

> The support for custom pre- and post-processing at the model deployment level is not yet available for use. This is a work in progress and will be available soon. You can find more information about the custom deployment here

When to use it?

The model deployers are optional components in the ZenML stack. They are used to deploy machine learning models to a target environment either a development (local) or a production (Kubernetes), the model deployers are mainly used to deploy models for real time inference use cases. With the model deployers and other stack components, you can build pipelines that are continuously trained and deployed to a production.

Model Deployers Flavors

ZenML comes with a `local` MLflow model deployer which is a simple model deployer that deploys models to a local MLflow server. Additional model deployers that can be used to deploy models on production environments are provided by integrations:

Model Deployer	Flavor	Integration	Notes
MLflow	`mlflow`	`mlflow`	Deploys ML Model locally
Seldon Core	`seldon`	`seldon Core`	Built on top of Kubernetes to deploy models for productior grade environment
KServe	`kserve`	`kserve`	Kubernetes based model deployment framework
Custom Implementation	*custom*		Extend the Artifact Store abstraction and provide your own implementation

Every model deployer may have different attributes that must be configured in order to interact with the model serving tool, framework or platform (e.g. hostnames, URLs, references to credentials, other client related configuration parameters). The following example shows the configuration of the MLflow and Seldon Core model deployers:

```
# Configure MLflow model deployer
zenml model-deployer register mlflow --flavor=mlflow

# Configure Seldon Core model deployer
zenml model-deployer register seldon --flavor=seldon \
--kubernetes_context=zenml-eks --kubernetes_namespace=zenml-workloads \
--base_url=http://abb84c444c7804aa98fc8c097896479d-377673393.us-east-1.elb.amazor
...
```

The role Model Deployer plays in a ZenML Stack

1

1. Holds all the stack related configuration attributes required to interact with the remote model serving tool, service or platform (e.g. hostnames, URLs, references to credentials, other client related configuration parameters). The following are examples of configuring the MLflow and Seldon Core

```
zenml integration install mlflow
zenml model-deployer register mlflow --flavor=mlflow
zenml stack register local_with_mlflow -m default -a default -o default -d mlflow --set
```

```
zenml integration install seldon
zenml model-deployer register seldon --flavor=seldon \
--kubernetes_context=zenml-eks --kubernetes_namespace=zenml-workloads \
--base_url=http://abb84c444c7804aa98fc8c097896479d-377673393.us-east-1.elb.amazonaws.co
...

zenml stack register seldon_stack -m default -a aws -o default -d seldon
```

2. Implements the continuous deployment logic necessary to deploy models in a way that updates an existing model server that is already serving a previous version of the same model instead of creating a new model server for every new model version. Every model server that the Model Deployer provisions externally to deploy a model is represented internally as a `Service` object that may be accessed for visibility and control over a single model deployment. This functionality can be consumed directly from ZenML pipeline steps, but it can also be used outside of the pipeline to deploy ad-hoc models. The following code is an example of using the Seldon Core Model Deployer to deploy a model inside a ZenML pipeline step:

```
from zenml.artifacts import ModelArtifact
from zenml.environment import Environment
from zenml.integrations.seldon.model_deployers import SeldonModelDeployer
from zenml.integrations.seldon.services.seldon_deployment import (
  SeldonDeploymentConfig,
  SeldonDeploymentService,
)
from zenml.steps import (
  STEP_ENVIRONMENT_NAME,
  StepContext,
  step,
)

@step(enable_cache=True)
def seldon_model_deployer_step(
  context: StepContext,
  model: ModelArtifact,
) -> SeldonDeploymentService:
  model_deployer = SeldonModelDeployer.get_active_model_deployer()

  # get pipeline name, step name and run id
  step_env = Environment()[STEP_ENVIRONMENT_NAME]

  service_config=SeldonDeploymentConfig(
      model_uri=model.uri,
      model_name="my-model",
      replicas=1,
      implementation="TENSORFLOW_SERVER",
      secret_name="seldon-secret",
      pipeline_name = step_env.pipeline_name,
      pipeline_run_id = step_env.pipeline_run_id,
      pipeline_step_name = step_env.step_name,
  )

  service = model_deployer.deploy_model(
      service_config, replace=True, timeout=300
  )

  print(
      f"Seldon deployment service started and reachable at:\n"
      f"    {service.prediction_url}\n"
  )

  return service
```

3. Acts as a registry for all Services that represent remote model servers. External model deployment servers can be listed and filtered using a variety of criteria, such as the name of the model or the names of the pipeline and step that was used to deploy the model. The Service objects returned by the Model Deployer can be used to interact with the remote model server, e.g. to get the operational status of a model server, the prediction URI that it exposes, or to stop or delete a model server:

```
from zenml.integrations.seldon.model_deployers import SeldonModelDeployer

model_deployer = SeldonModelDeployer.get_active_model_deployer()
services = model_deployer.find_model_server(
    pipeline_name="continuous-deployment-pipeline",
    pipeline_step_name="seldon_model_deployer_step",
    model_name="my-model",
)
if services:
    if services[0].is_running:
        print(
            f"Seldon deployment service started and reachable at:\n"
            f"    {services[0].prediction_url}\n"
        )
    elif services[0].is_failed:
        print(
            f"Seldon deployment service is in a failure state. "
            f"The last error message was: {services[0].status.last_error}"
        )
    else:
        print(f"Seldon deployment service is not running")

        # start the service
        services[0].start(timeout=100)

    # delete the service
    model_deployer.delete_service(services[0].uuid, timeout=100, force=False)
```

How to Interact with model deployer after deployment?

When a Model Deployer is part of the active ZenML Stack, it is also possible to interact with it from the CLI to list, start, stop or delete the model servers that is manages:

```
$ zenml served-models list
┏━━━━━━━━┯━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┯━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┯━━━━━━━━━
┃ STATUS │ UUID                                 │ PIPELINE_NAME                  │ PIPELINI
┠────────┼──────────────────────────────────────┼────────────────────────────────┼─────────
┃   ✅   │ 8cbe671b-9fce-4394-a051-68e001f92765 │ continuous_deployment_pipeline │ seldon_m
┗━━━━━━━━┷━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┷━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┷━━━━━━━━━

$ zenml served-models describe 8cbe671b-9fce-4394-a051-68e001f92765
                        Properties of Served Model 8cbe671b-9fce-4394-a051-68e001f92765

┏━━━━━━━━━━━━━━━━━━━━━━━━┯━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━
┃ MODEL SERVICE PROPERTY │ VALUE
┠────────────────────────┼──────────────────────────────────────────────────────────────────────
┃ MODEL_NAME             │ mnist
┠────────────────────────┼──────────────────────────────────────────────────────────────────────
┃ MODEL_URI              │ s3://zenfiles/seldon_model_deployer_step/output/884/seldon
┠────────────────────────┼──────────────────────────────────────────────────────────────────────
┃ PIPELINE_NAME          │ continuous_deployment_pipeline
┠────────────────────────┼──────────────────────────────────────────────────────────────────────
┃ PIPELINE_RUN_ID        │ continuous_deployment_pipeline-11_Apr_22-09_39_27_648527
┠────────────────────────┼──────────────────────────────────────────────────────────────────────
┃ PIPELINE_STEP_NAME     │ seldon_model_deployer_step
┠────────────────────────┼──────────────────────────────────────────────────────────────────────
┃ PREDICTION_URL         │ http://abb84c444c7804aa98fc8c097896479d-377673393.us-east-1.elb
┠────────────────────────┼──────────────────────────────────────────────────────────────────────
┃ SELDON_DEPLOYMENT      │ zenml-8cbe671b-9fce-4394-a051-68e001f92765
┠────────────────────────┼──────────────────────────────────────────────────────────────────────
┃ STATUS                 │ ✅
┠────────────────────────┼──────────────────────────────────────────────────────────────────────
┃ STATUS_MESSAGE         │ Seldon Core deployment 'zenml-8cbe671b-9fce-4394-a051-68e001f92
┠────────────────────────┼──────────────────────────────────────────────────────────────────────
┃ UUID                   │ 8cbe671b-9fce-4394-a051-68e001f92765
┗━━━━━━━━━━━━━━━━━━━━━━━━┷━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━

$ zenml served-models get-url 8cbe671b-9fce-4394-a051-68e001f92765
  Prediction URL of Served Model 8cbe671b-9fce-4394-a051-68e001f92765 is:
  http://abb84c444c7804aa98fc8c097896479d-377673393.us-east-1.elb.amazonaws.com/seldon/zenm
1b-9fce-4394-a051-68e001f92765/api/v0.1/predictions

$ zenml served-models delete 8cbe671b-9fce-4394-a051-68e001f92765
```

Services can be passed through steps like any other object, and used to interact with the external systems that they represent:

```python
from zenml.steps import step

@step
```

```python
def my_step(my_service: MyService) -> ...:
    if not my_service.is_running:
        my_service.start()  # starts service
    my_service.stop()  # stops service
```

The ZenML integrations that provide Model Deployer stack components also include standard pipeline steps that can directly be inserted into any pipeline to achieve a continuous model deployment workflow. These steps take care of all the aspects of continuously deploying models to an external server and saving the Service configuration into the Artifact Store, where they can be loaded at a later time and re-create the initial conditions used to serve a particular model.

MLflow

How to deploy your models locally with MLflow

The MLflow Model Deployer is one of the available flavors of the Model Deployer stack component. Provided with the MLflow integration it can be used to deploy and manage MLflow models on a local running MLflow server.

> The MLflow Model Deployer is not yet available for use in production. This is a work in progress and will be available soon. At the moment it is only available for use in a local development environment.

When to use it?

MLflow is a popular open source platform for machine learning. It's a great tool for managing the entire lifecycle of your machine learning. One of the most important features of MLflow is the ability to package your model and its dependencies into a single artifact that can be deployed to a variety of deployment targets.

You should use the MLflow Model Deployer:

- if you want to have an easy way to deploy your models locally and perform real-time predictions using the running MLflow prediction server.
- if you are looking to deploy your models in a simple way without the need for a dedicated deployment environment like Kubernetes or advanced infrastructure configuration.

If you are looking to deploy your models in a more complex way, you should use one of the other Model Deployer Flavors available in ZenML (e.g. Seldon Core, KServe, etc.)

How do you deploy it?

The MLflow Model Deployer flavor is provided by the MLflow ZenML integration, you need to install it on your local machine to be able to deploy your models. You can do this by running the following command:

```
zenml integration install mlflow -y
```

To register the MLflow model deployer with ZenML you need to run the following command:

```
zenml model-deployer register mlflow_deployer --flavor=mlflow
```

The ZenML integration will provision a local MLflow deployment server as a daemon process that will continue to run in the background to serve the latest MLflow model.

How do you use it?

The first step to be able to deploy and use your MLflow model is to create Service deployment from code, this is done by setting the different parameters that the MLflow deployment step requires.

```python
from zenml.steps import BaseStepConfig
from zenml.integrations.mlflow.steps import mlflow_deployer_step
from zenml.integrations.mlflow.steps import MLFlowDeployerConfig

...

class MLFlowDeploymentLoaderStepConfig(BaseStepConfig):
    """MLflow deployment getter configuration

    Attributes:
        pipeline_name: name of the pipeline that deployed the MLflow prediction
            server
        step_name: the name of the step that deployed the MLflow prediction
            server
        running: when this flag is set, the step only returns a running service
    """

    pipeline_name: str
    step_name: str
    running: bool = True

model_deployer = mlflow_deployer_step(name="model_deployer")

...

# Initialize a continuous deployment pipeline run
deployment = continuous_deployment_pipeline(
    ...,
    # as a last step to our pipeline the model deployer step is run with it config in plac
    model_deployer=model_deployer(config=MLFlowDeployerConfig(workers=3)),
)
```

You can run predictions on the deployed model with something like:

```python
from zenml.integrations.mlflow.services import MLFlowDeploymentService
from zenml.steps import BaseStepConfig, Output, StepContext, step
from zenml.services import load_last_service_from_step

...

class MLFlowDeploymentLoaderStepConfig(BaseStepConfig):
    # see implementation above

    ...

# Step to retrieve the service associated with the last pipeline run
@step(enable_cache=False)
def prediction_service_loader(
    config: MLFlowDeploymentLoaderStepConfig, context: StepContext
) -> MLFlowDeploymentService:
    """Get the prediction service started by the deployment pipeline"""

    service = load_last_service_from_step(
        pipeline_name=config.pipeline_name,
        step_name=config.step_name,
        step_context=context,
        running=config.running,
    )
    if not service:
        raise RuntimeError(
            f"No MLflow prediction service deployed by the "
            f"{config.step_name} step in the {config.pipeline_name} pipeline "
            f"is currently running."
        )

    return service

# Use the service for inference
@step
def predictor(
    service: MLFlowDeploymentService,
    data: np.ndarray,
) -> Output(predictions=np.ndarray):
    """Run a inference request against a prediction service"""

    service.start(timeout=10)  # should be a NOP if already started
    prediction = service.predict(data)
    prediction = prediction.argmax(axis=-1)

    return prediction

# Initialize an inference pipeline run
inference = inference_pipeline(
    ...,
    prediction_service_loader=prediction_service_loader(
        MLFlowDeploymentLoaderStepConfig(
            pipeline_name="continuous_deployment_pipeline",
            step_name="model_deployer",
        )
    ),
    predictor=predictor(),
)
```

You can check the MLflow deployment example for more details.

- Model Deployer with MLflow

For more information and a full list of configurable attributes of the MLflow Model Deployer, check out the API Docs.

Seldon

How to deploy models to Kubernetes with Seldon Core

The Seldon Core Model Deployer is one of the available flavors of the Model Deployer stack component. Provided with the MLflow integration it can be used to deploy and manage models on a inference server running on top of a Kubernetes cluster.

When to use it?

Seldon Core is a production grade open source model serving platform. It packs a wide range of features built around deploying models to REST/GRPC microservices that include monitoring and logging, model explainers, outlier detectors and various continuous deployment strategies such as A/B testing, canary deployments and more.

Seldon Core also comes equipped with a set of built-in model server implementations designed to work with standard formats for packaging ML models that greatly simplify the process of serving models for real-time inference.

You should use the Seldon Core Model Deployer:

- If you are looking to deploy your model on a more advanced infrastructure like Kubernetes.
- If you want to handle the lifecycle of the deployed model with no downtime, including updating the runtime graph, scaling, monitoring, and security.
- Looking for more advanced API endpoints to interact with the deployed model, including REST and GRPC endpoints.
- If you want more advanced deployment strategies like A/B testing, canary deployments, and more.
- if you have a need for a more complex deployment process which can be customized by the advanced inference graph that includes custom TRANSFORMER and ROUTER

If you are looking for a more easy way to deploy your models locally, you can use the MLflow Model Deployer flavor.

How to deploy it?

ZenML provides a Seldon Core flavor build on top of the Seldon Core Integration to allow you to deploy and use your models in a production-grade environment. In order to use the integration you need to install it on your local machine to be able to register an Seldon Core Model deployer with ZenML and add it to your stack:

```
zenml integration install seldon-core -y
```

To deploy and make use of the Seldon Core integration we need to have the following prerequisites:

1. access to a Kubernetes cluster. The example accepts a `--kubernetes-context` command line argument. This Kubernetes context needs to point to the Kubernetes cluster where Seldon Core model servers will be deployed. If the context is not explicitly supplied to the example, it defaults to using the locally active context. You can find more information about setup and usage of the Kubernetes cluster in the ZenML Cloud Guide
2. Seldon Core needs to be preinstalled and running in the target Kubernetes cluster. Check out the official Seldon Core installation instructions).
3. models deployed with Seldon Core need to be stored in some form of persistent shared storage that is accessible from the Kubernetes cluster where Seldon Core is installed (e.g. AWS S3, GCS, Azure Blob Storage, etc.). You can use one of the supported remote storage flavors to store your models as part of your stack

Since the Seldon Model Deployer is interacting with the Seldon Core model server deployed on a Kubernetes cluster, you need to provide a set of configuration parameters. These parameters are:

- kubernetes_context: the Kubernetes context to use to contact the remote Seldon Core installation. If not specified, the current configuration is used. Depending on where the Seldon model deployer is being used
- kubernetes_namespace: the Kubernetes namespace where the Seldon Core deployment servers are provisioned and managed by ZenML. If not specified, the namespace set in the current configuration is used.
- base_url: the base URL of the Kubernetes ingress used to expose the Seldon Core deployment servers.
- secret: the name of a ZenML secret containing the credentials used by Seldon Core storage initializers to authenticate to the Artifact Store

> Configuring an Seldon Core in a Kubernetes cluster can be a complex and error prone process, so we have provided a set of of Terraform-based recipes to quickly provision popular combinations of MLOps tools. More information about these recipes can be found in the Open Source MLOps Stack Recipes

Managing Seldon Core Credentials

The Seldon Core model servers need to access the Artifact Store in the ZenML stack to retrieve the model artifacts. This usually involve passing some credentials to the Seldon Core model servers required to authenticate with the Artifact Store. In ZenML, this is done by creating a ZenML secret with the proper credentials and configuring the Seldon Core Model Deployer stack component to use it, by passing the `--secret` argument to the CLI command used to register the model deployer. We've already done the latter, now all that is left to do is to configure the `s3-store` ZenML secret specified before as a Seldon Model

There are built-in secret schemas that the Seldon Core integration provides which can be used to configure credentials for the 3 main types of Artifact Stores supported by ZenML: S3, GCS and Azure.

you can use `seldon_s3` for AWS S3 or `seldon_gs` for GCS and `seldon_az` for Azure. To read more about secrets, secret schemas and how they are used in ZenML, please refer to the Secrets Manager.

The following is an example of registering an S3 secret with the Seldon Core model deployer:

```
$ zenml secret register -s seldon_s3 s3-store \
    --rclone_config_s3_env_auth=False \
    --rclone_config_s3_access_key_id='ASAK2NSJVO4HDQC7Z25F' \ --rclone_config_s3_secret_ac
    --rclone_config_s3_session_token=@./aws_session_token.txt \
    --rclone_config_s3_region=us-east-1
Expanding argument value rclone_config_s3_session_token to contents of file ./aws_session_
The following secret will be registered.
┏━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┯━━━━━━━━━━━━━━┓
┃             SECRET_KEY             │ SECRET_VALUE ┃
┠────────────────────────────────────┼──────────────┨
┃       rclone_config_s3_type        │ ***          ┃
┃     rclone_config_s3_provider      │ ***          ┃
┃     rclone_config_s3_env_auth      │ ***          ┃
┃   rclone_config_s3_access_key_id   │ ***          ┃
┃ rclone_config_s3_secret_access_key │ ***          ┃
┃   rclone_config_s3_session_token   │ ***          ┃
┃      rclone_config_s3_region       │ ***          ┃
┗━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┷━━━━━━━━━━━━━━┛
INFO:botocore.credentials:Found credentials in shared credentials file: ~/.aws/credentials

$ zenml secret get s3-store
INFO:botocore.credentials:Found credentials in shared credentials file: ~/.aws/credentials
┏━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┯━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┓
┃             SECRET_KEY             │ SECRET_VALUE                                         ┃
┠────────────────────────────────────┼──────────────────────────────────────────────────────┨
┃       rclone_config_s3_type        │ s3                                                   ┃
┃     rclone_config_s3_provider      │ aws                                                  ┃
┃     rclone_config_s3_env_auth      │ False                                                ┃
┃   rclone_config_s3_access_key_id   │ ASAK2NSJVO4HDQC7Z25F                                 ┃
┃ rclone_config_s3_secret_access_key │ AhkFSfhjj23fSDFfjklsdfj34hkls32SDfscs…               ┃
┃   rclone_config_s3_session_token   │ FwoGZXIvYXdzEG4aDHogqi7YRrJyVJUVfSKpA…               ┃
┃                                    │                                                      ┃
┃      rclone_config_s3_region       │ us-east-1                                            ┃
┗━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┷━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┛
```

How do you use it?

We can register the model deployer and use it in our active stack:

```
zenml model-deployer register seldon_deployer --flavor=seldon \
  --kubernetes_context=zenml-eks \
  --kubernetes_namespace=zenml-workloads \
  --base_url=http://$INGRESS_HOST \
  --secret=s3-store-credentials

# Now we can use the model deployer in our stack
zenml stack update seldon_stack --model-deployer=seldon_deployer
```

The following code snippet shows how to use the Seldon Core Model Deployer to deploy a model inside a ZenML pipeline step:

```python
from zenml.artifacts import ModelArtifact
from zenml.environment import Environment
from zenml.integrations.seldon.model_deployers import SeldonModelDeployer
from zenml.integrations.seldon.services.seldon_deployment import (
  SeldonDeploymentConfig,
  SeldonDeploymentService,
)
from zenml.steps import (
  STEP_ENVIRONMENT_NAME,
  StepContext,
  step,
)

@step(enable_cache=True)
def seldon_model_deployer_step(
  context: StepContext,
  model: ModelArtifact,
) -> SeldonDeploymentService:
  model_deployer = SeldonModelDeployer.get_active_model_deployer()

  # get pipeline name, step name and run id
  step_env = Environment()[STEP_ENVIRONMENT_NAME]

  service_config=SeldonDeploymentConfig(
      model_uri=model.uri,
      model_name="my-model",
      replicas=1,
      implementation="TENSORFLOW_SERVER",
      secret_name="seldon-secret",
      pipeline_name = step_env.pipeline_name,
      pipeline_run_id = step_env.pipeline_run_id,
      pipeline_step_name = step_env.step_name,
  )

  service = model_deployer.deploy_model(
      service_config, replace=True, timeout=300
  )

  print(
      f"Seldon deployment service started and reachable at:\n"
      f"    {service.prediction_url}\n"
  )

  return service
```

A concrete example of using the Seldon Core Model Deployer can be found here.

For more information and a full list of configurable attributes of the Seldon Core Model Deployer, check out the API Docs.

Develop a Custom Model Deployer

How to develop a custom model deployer

To deploy and manage your trained machine learning models, ZenML provides a stack component called `Model Deployer`. This component is responsible for interacting with the deployment tool, framework or platform.

When present in a stack, the model deployer can also acts as a registry for models that are served with ZenML. You can use the model deployer to list all models that are currently deployed for online inference or filtered according to a particular pipeline run or step, or to suspend, resume or delete an external model server managed through ZenML.

Base Abstraction

In ZenML, the base abstraction of the model deployer is built on top of three major criteria:

1. It needs to contain all the stack-related configuration attributes required to interact with the remote model serving tool, service or platform (e.g. hostnames, URLs, references to credentials, other client-related configuration parameters).
2. It needs to implement the continuous deployment logic necessary to deploy models in a way that updates an existing model server that is already serving a previous version of the same model instead of creating a new model server for every new model version (see the `deploy_model` abstract method). This functionality can be consumed directly from ZenML pipeline steps, but it can also be used outside the pipeline to deploy ad-hoc models. It is also usually coupled with a standard model deployer step, implemented by each integration, that hides the details of the deployment process from the user.
3. It needs to act as a ZenML BaseService registry, where every BaseService instance is used as an internal representation of a remote model server (see the `find_model_server` abstract method). To achieve this, it must be able to re-create the configuration of a BaseService from information that is persisted externally, alongside or even as part of the remote model server configuration itself. For example, for model servers that are implemented as Kubernetes resources, the BaseService instances can be serialized and saved as Kubernetes resource annotations. This allows the model deployer to keep track of all externally running model servers and to re-create their corresponding BaseService instance representations at any given time. The model deployer also defines methods that implement basic life-cycle management on remote model servers outside the coverage of a pipeline (see `stop_model_server`, `start_model_server` and `delete_model_server`).

Putting all these considerations together, we end up with the following interface:

```python
from abc import ABC, abstractmethod
from typing import ClassVar, Dict, Generator, List, Optional
from uuid import UUID

from zenml.enums import StackComponentType
from zenml.services import BaseService, ServiceConfig
from zenml.stack import StackComponent

DEFAULT_DEPLOYMENT_START_STOP_TIMEOUT = 300


class BaseModelDeployer(StackComponent, ABC):
    """Base class for all ZenML model deployers."""

    # Class variables
    TYPE: ClassVar[StackComponentType] = StackComponentType.MODEL_DEPLOYER

    @abstractmethod
    def deploy_model(
        self,
        config: ServiceConfig,
        replace: bool = False,
        timeout: int = DEFAULT_DEPLOYMENT_START_STOP_TIMEOUT,
    ) -> BaseService:
        """Abstract method to deploy a model."""

    @staticmethod
    @abstractmethod
    def get_model_server_info(
        service: BaseService,
    ) -> Dict[str, Optional[str]]:
        """Give implementation-specific way to extract relevant model server
        properties for the user."""

    @abstractmethod
    def find_model_server(
        self,
        running: bool = False,
        service_uuid: Optional[UUID] = None,
        pipeline_name: Optional[str] = None,
        pipeline_run_id: Optional[str] = None,
        pipeline_step_name: Optional[str] = None,
        model_name: Optional[str] = None,
        model_uri: Optional[str] = None,
        model_type: Optional[str] = None,
    ) -> List[BaseService]:
        """Abstract method to find one or more model servers that match the
        given criteria."""

    @abstractmethod
    def stop_model_server(
        self,
        uuid: UUID,
        timeout: int = DEFAULT_DEPLOYMENT_START_STOP_TIMEOUT,
        force: bool = False,
    ) -> None:
        """Abstract method to stop a model server."""

    @abstractmethod
    def start_model_server(
        self,
        uuid: UUID,
        timeout: int = DEFAULT_DEPLOYMENT_START_STOP_TIMEOUT,
    ) -> None:
        """Abstract method to start a model server."""

    @abstractmethod
    def delete_model_server(
        self,
        uuid: UUID,
        timeout: int = DEFAULT_DEPLOYMENT_START_STOP_TIMEOUT,
        force: bool = False,
    ) -> None:
        """Abstract method to delete a model server."""
```

This is a slimmed-down version of the base implementation which aims to highlight the abstraction layer. In order to see the full implementation and get the complete docstrings, please check the API docs.

Building your own model deployers

If you want to create your own custom flavor for a model deployer, you can follow the following steps:

1. Create a class which inherits from the `BaseModelDeployer`.
2. Define the `FLAVOR` class variable.
3. Implement the `abstactmethod`s based on the API of your desired model deployer.

Once you are done with the implementation, you can register it through the CLI as:

```
zenml model-deployer flavor register <THE-SOURCE-PATH-OF-YOUR-MODEL_DEPLOYER>
```

Step Operators

How to execute individual steps in specialized environments

The step operator enables the execution of individual pipeline steps in specialized runtime environments that are optimized for certain workloads. These specialized environments can give your steps access to resources like GPUs or distributed processing frameworks like Spark.

> **Comparison to orchestrators:** The orchestrator is a mandatory stack component that is responsible for executing all steps of a pipeline in the correct order and provide additional features such as scheduling pipeline runs. The step operator on the other hand is used to only execute individual steps of the pipeline in a separate environment in case the environment provided by the orchestrator is not feasible.

When to use it

A step operator should be used if one or more steps of a pipeline require resources that are not available in the runtime environments provided by the orchestrator. An example would be a step that trains a computer vision model and requires a GPU to run in reasonable time, combined with a Kubeflow orchestrator running on a kubernetes cluster which does not contain any GPU nodes. In that case it makes sense to include a step operator like SageMaker, Vertex or AzureML to execute the training step with a GPU.

Step Operator Flavors

Step operators to execute steps on one of the big cloud providers are provided by the following ZenML integrations:

Step Operator	Flavor	Integration	Notes
SageMaker	`sagemaker`	`aws`	Uses SageMaker to execute steps
Vertex	`vertex`	`gcp`	Uses Vertex AI to execute steps
AzureML	`azureml`	`azure`	Uses AzureML to execute steps
Custom Implementation	*custom*		Extend the step operator abstraction and provide your owr implementation

If you would like to see the available flavors of step operators, you can use the command:

```
zenml step-operator flavor list
```

How to use it

You don't need to directly interact with any ZenML step operator in your code. As long as the step operator that you want to use is part of your active ZenML stack, you can simply specify it in the `@step` decorator of your step:

```python
from zenml.steps import step

@step(custom_step_operator=<STEP_OPERATOR_NAME>)
def my_step(...) -> ...:
    ...
```

Amazon SageMaker

How to execute individual steps in SageMaker

The SageMaker step operator is a step operator flavor provided with the ZenML `aws` integration that uses SageMaker to execute individual steps of ZenML pipelines.

When to use it

You should use the SageMaker step operator if:

- one or more steps of your pipeline require computing resources (CPU, GPU, memory) that are not provided by your orchestrator.
- you have access to SageMaker. If you're using a different cloud provider, take a look at the Vertex or AzureML step operators.

How to deploy it

- Create a role in the IAM console that you want the jobs running in SageMaker to assume. This role should at least have the `AmazonS3FullAccess` and `AmazonSageMakerFullAccess` policies applied. Check here for a guide on how to set up this role.

How to use it

To use the SageMaker step operator, we need:

- The ZenML `aws` integration installed. If you haven't done so, run
  ```
  zenml integration install aws
  ```
- Docker installed and running.
- An IAM role with the correct permissions. See the deployment section for detailed instructions.
- An AWS container registry as part of our stack. Take a look here for a guide on how to set that up.
- The `aws` cli set up and authenticated. Make sure you have the permissions to create and manage SageMaker runs.
-

- An instance type that we want to execute our steps on. See here for a list of available instance types.
- (Optional) An experiment which is used to group SageMaker runs. Check this guide to see how to create an experiment.

We can then register the step operator and use it in our active stack:

```
zenml step-operator register <NAME> \
    --flavor=sagemaker \
    --role=<SAGEMAKER_ROLE> \
    --instance_type=<INSTANCE_TYPE> \
#   --experiment_name=<SEXPERIMENT_NAME> # optionally specify an experiment to assign this

# Add the step operator to the active stack
zenml stack update -s <NAME>
```

Once you added the step operator to your active stack, you can use it to execute individual steps of your pipeline by specifying it in the `@step` decorator as follows:

```python
from zenml.steps import step

@step(custom_step_operator=<NAME>)
def trainer(...) -> ...:
    """Train a model."""
    # This step will be executed in SageMaker.
```

ZenML will build a Docker image called `zenml-sagemaker` which includes your code and use it to run your steps in SageMaker. Check out this page if you want to learn more about how ZenML builds these images and how you can customize them.

If you decide you need the full flexibility of having a custom base image, you can update your existing step operator

```
zenml step-operator update <NAME> \
--base_image=<IMAGE_NAME>
```

or set it when registering a new SageMaker step operator:

```
zenml step-operator register <NAME> \
--flavor=sagemaker \
--base_image=<IMAGE_NAME>
...
```

A concrete example of using the SageMaker step operator can be found here.

For more information and a full list of configurable attributes of the SageMaker step operator, check out the API Docs.

Google Cloud VertexAI

How to execute individual steps in Vertex AI

The Vertex step operator is a step operator flavor provided with the ZenML `gcp` integration that uses Vertex AI to execute individual steps of ZenML pipelines.

When to use it

You should use the Vertex step operator if:

- one or more steps of your pipeline require computing resources (CPU, GPU, memory) that are not provided by your orchestrator.
- you have access to Vertex AI. If you're using a different cloud provider, take a look at the SageMaker or AzureML step operators.

How to deploy it

- Enable Vertex AI here.
- Create a service account with the right permissions to create Vertex AI jobs (`roles/aiplatform.admin`) and push to the container registry (`roles/storage.admin`).

How to use it

To use the Vertex step operator, we need:

- The ZenML `gcp` integration installed. If you haven't done so, run
  ```
  zenml integration install gcp
  ```
- Docker installed and running.
- Vertex AI enabled and a service account file. See the deployment section for detailed instructions.
- A GCR container registry as part of our stack.
- (Optional) A machine type that we want to execute our steps on (this defaults to `n1-standard-4`). See here for a list of available machine types.

We can then register the step operator and use it in our active stack:

```
zenml step-operator register <NAME> \
    --flavor=vertex \
    --project=<GCP_PROJECT> \
    --region=<REGION> \
    --service_account_path=<SERVICE_ACCOUNT_PATH> \
#   --machine_type=<MACHINE_TYPE> # optionally specify the type of machine to run on

# Add the step operator to the active stack
zenml stack update -s <NAME>
```

Once you added the step operator to your active stack, you can use it to execute individual steps of your pipeline by specifying it in the `@step` decorator as follows:

```python
from zenml.steps import step

@step(custom_step_operator=<NAME>)
def trainer(...) -> ...:
    """Train a model."""
    # This step will be executed in Vertex.
```

ZenML will build a Docker image called `zenml-vertex` which includes your code and use it to run your steps in Vertex. Check out this page if you want to learn more about how ZenML builds these images and how you can customize them.

If you decide you need the full flexibility of having a custom base image, you can update your existing step operator

```shell
zenml step-operator update <NAME> \
--base_image=<IMAGE_NAME>
```

or set it when registering a new Vertex step operator:

```shell
zenml step-operator register <NAME> \
--flavor=vertex \
--base_image=<IMAGE_NAME>
...
```

A concrete example of using the Vertex step operator can be found here.

For more information and a full list of configurable attributes of the Vertex step operator, check out the API Docs.

AzureML

How to execute individual steps in AzureML

The AzureML step operator is a step operator flavor provided with the ZenML `azure` integration that uses AzureML to execute individual steps of ZenML pipelines.

When to use it

You should use the AzureML step operator if:

- one or more steps of your pipeline require computing resources (CPU, GPU, memory) that are not provided by your orchestrator.
- you have access to AzureML. If you're using a different cloud provider, take a look at the SageMaker or

Vertex step operators.

How to deploy it

- Create a `Machine learning` resource on Azure.
- Once your resource is created, you can head over to the `Azure Machine Learning Studio` and create a compute cluster to run your pipelines.
- Create an `environment` for your pipelines. Follow this guide to set one up.
- (Optional) Create a Service Principal for authentication. This is required if you intend to run your pipelines with a remote orchestrator.

How to use it

To use the AzureML step operator, we need:

- The ZenML `azure` integration installed. If you haven't done so, run

  ```
  zenml integration install azure
  ```
- An AzureML compute cluster and environment. See the deployment section for detailed instructions.

We can then register the step operator and use it in our active stack:

```
zenml step-operator register <NAME> \
    --flavor=azureml \
    --subscription_id=<AZURE_SUBSCRIPTION_ID> \
    --resource_group=<AZURE_RESOURCE_GROUP> \
    --workspace_name=<AZURE_WORKSPACE_NAME> \
    --compute_target_name=<AZURE_COMPUTE_TARGET_NAME> \
    --environment_name=<AZURE_ENVIRONMENT_NAME> \
# only pass these if using Service Principal Authentication

#   --tenant_id=<TENANT_ID> \
#   --service_principal_id=<SERVICE_PRINCIPAL_ID> \
#   --service_principal_password=<SERVICE_PRINCIPAL_PASSWORD> \

# Add the step operator to the active stack
zenml stack update -s <NAME>
```

Once you added the step operator to your active stack, you can use it to execute individual steps of your pipeline by specifying it in the `@step` decorator as follows:

```python
from zenml.steps import step

@step(custom_step_operator=<NAME>)
def trainer(...) -> ...:
    """Train a model."""
    # This step will be executed in AzureML.
```

ZenML will build Docker images which include your code and use these to run your steps in AzureML. Check out this page if you want to learn more about how ZenML builds these images and how you can customize them.

If you decide you need the full flexibility of having a custom base image, you can update your existing step operator

```
zenml step-operator update <NAME> \
--docker_base_image=<IMAGE_NAME>
```

or set it when registering a new AzureML step operator:

```
zenml step-operator register <NAME> \
--flavor=sagemaker \
--docker_base_image=<IMAGE_NAME>
...
```

A concrete example of using the AzureML step operator can be found here.

For more information and a full list of configurable attributes of the AzureML step operator, check out the API Docs.

Develop a Custom Step Operator

How to develop a custom step operator

Base Abstraction

The `BaseStepOperator` is the abstract base class that needs to be subclassed in order to run specific steps of your pipeline in a separate environment. As step operators can come in many shapes and forms, the base class exposes a deliberately basic and generic interface:

1. As it is the base class for a specific type of `StackComponent`, it inherits from the `StackComponent` class. This sets the `TYPE` variable to the specific `StackComponentType`.
2. The `FLAVOR` class variable needs to be set in subclasses as it is meant to identify the implementation flavor of the particular step operator.
3. Lastly, the base class features one `abstractmethod` called `launch()`. In the implementation of every step operator flavor, the `launch()` method is responsible for preparing the environment and executing the step.

```python
from abc import ABC, abstractmethod
from typing import ClassVar, List

from zenml.enums import StackComponentType
from zenml.stack import StackComponent
```

```python

class BaseStepOperator(StackComponent, ABC):
    """Base class for all ZenML step operators."""

    # Class Configuration
    TYPE: ClassVar[StackComponentType] = StackComponentType.STEP_OPERATOR

    @abstractmethod
    def launch(
        self,
        pipeline_name: str,
        run_name: str,
        requirements: List[str],
        entrypoint_command: List[str],
    ) -> None:
        """Abstract method to execute a step.

        Concrete step operator subclasses must implement the following
        functionality in this method:
        - Prepare the execution environment and install all the necessary
          `requirements`
        - Launch a **synchronous** job that executes the `entrypoint_command`
        """
```

This is a slimmed-down version of the base implementation which aims to highlight the abstraction layer. In order to see the full implementation and get the complete docstrings, please check the API docs.

Building your own custom step operator

If you want to create a custom step operator, you can follow these steps:

1. Create a class which inherits from the `BaseStepOperator`.
2. Define the `FLAVOR` class variable.
3. Implement the abstract `launch()` method. This method has two main responsibilities:
   - Preparing a suitable execution environment (e.g. a Docker image): The general environment is highly dependent on the concrete step operator implementation, but for ZenML to be able to run the step it requires you to install some `pip` dependencies. The list of requirements needed to successfully execute the step is passed in via the `requirements` argument of the `launch()` method. Additionally, you'll have to make sure that all the source code of your ZenML step and pipeline are available within this execution environment.
   - Running the entrypoint command: Actually running a single step of a pipeline requires knowledge of many ZenML internals and is implemented in the `zenml.step_operators.entrypoint` module. As long as your environment was set up correctly (see the previous bullet point), you can run the step using the command provided via the `entrypoint_command` argument of the `launch()` method.

Once you are done with the implementation, you can register it through the CLI as:

```
zenml step-operator flavor register <SOURCE-PATH-OF-YOUR-STEP-OPERATOR-CLASS>
```

Alerters

How to send automated alerts to chat services

Alerters allow you to send messages to chat services (like Slack, Discord, Mattermost, etc.) from within your pipelines. This is useful to immediately get notified when failures happen, for general monitoring/reporting, and also for building human-in-the-loop ML.

Alerter Flavors

Currently, the SlackAlerter is the only available alerter integration. However, it is straightforward to extend ZenML and build an alerter for other chat services.

Alerter	Flavor	Integration	Notes
Slack	`slack`	`slack`	Interacts with a Slack channel
Custom Implementation	*custom*		Extend the alerter abstraction and provide your own implementation

If you would like to see the available flavors of alerters in your terminal, you can use the following command:

```
zenml alerter flavor list
```

How to use Alerters with ZenML

Each alerter integration comes with specific standard steps that you can use out-of-the-box.

However, you first need to register an alerter component in your terminal:

```
zenml alerter register <ALERTER_NAME> ...
```

Then you can add it to your stack using

```
zenml stack register ... -al <ALERTER_NAME>
```

Afterwards, you can import the alerter standard steps provided by the respective integration and directly use them in your pipelines.

Slack Alerter

How to send automated alerts to a Slack channel

The `SlackAlerter` enables you to send messages to a dedicated Slack channel directly from within your pipelines.

The `slack` integration also contains the following two standard steps:

- slack_alerter_post_step takes a string, posts it to Slack, and returns `True` if the operation succeeded, else `False`.
- slack_alerter_ask_step does the same as `slack_alerter_post_step`, but after sending the message, it waits until someone approves or rejects the operation from within Slack (e.g., by sending "approve" / "reject" to the bot in response). `slack_alerter_ask_step` then only returns `True` if the operation succeeded and was approved, else `False`.

Interacting with Slack from within your pipelines can be very useful in practice:

- The `slack_alerter_post_step` allows you to get notified immediately when failures happen (e.g., model performance degradation, data drift, ...),
- The `slack_alerter_ask_step` allows you to integrate a human-in-the-loop into your pipelines before executing critical steps, such as deploying new models.

How to use it

Requirements

Before you can use the `SlackAlerter`, you first need to install ZenML's `slack` integration:

```
zenml integration install slack -y
```

> See the Integrations page for more details on ZenML integrations and how to install and use them.

Setting Up a Slack Bot

In order to use the `SlackAlerter`, you first need to have a Slack workspace set up with a channel that you want your pipelines to post to.

Then, you need to create a Slack App with a bot in your workspace.

> ⓘ Make sure to give your Slack bot `chat:write` and `chat:write.public` permissions in the `OAuth & Permissions` tab under `Scopes`.

Registering a Slack Alerter in ZenML

Next, you need to register a `slack` alerter in ZenML and link it to the bot you just created. You can do this with the following command:

```
zenml alerter register slack_alerter \
    --flavor=slack \
    --slack_token=<SLACK_TOKEN> \
    --default_slack_channel_id=<SLACK_CHANNEL_ID>
```

Here is where you can find the required parameters:

- `<SLACK_CHANNEL_ID>`: Open your desired Slack channel in a browser, and copy out the last part of the URL starting with `C....`.
- `<SLACK_TOKEN>`: This is the Slack token of your bot. You can find it in the Slack app settings under `OAuth & Permissions`.

After you have registered the `slack_alerter`, you can add it to your stack like this:

```
zenml stack register ... -al slack_alerter
```

How to Use the Slack Alerter

After you have a `SlackAlerter` configured in your stack, you can directly import the slack_alerter_post_step and slack_alerter_ask_step steps and use them in your pipelines.

Since these steps expect a string message as input (which needs to be the output of another step), you typically also need to define a dedicated formatter step that takes whatever data you want to communicate and generates the string message that the alerter should post.

As an example, adding `slack_alerter_ask_step()` into your pipeline could look like this:

```
from zenml.integrations.slack.steps.slack_alerter_ask_step import slack_alerter_ask_step
from zenml.steps import step
from zenml.pipelines import pipeline


@step
def my_formatter_step(artifact_to_be_communicated) -> str:
    return f"Here is my artifact {artifact_to_be_communicated}!"


@pipeline
def my_pipeline(..., formatter, alerter):
    ...
    artifact_to_be_communicated = ...
    message = formatter(artifact_to_be_communicated)
    approved = alerter(message)
    ... # Potentially have different behavior in subsequent steps if `approved`


my_pipeline(
    ...
    formatter=my_formatter_step(),
    alerter=slack_alerter_ask_step(),
).run()
```

For complete code examples of both Slack alerter steps, see the slack alerter example, where we first send the test accuracy of a model to Slack and then wait with model deployment until a user approves it in Slack.

For more information and a full list of configurable attributes of the Slack alerter, check out the API Docs.

Develop a Custom Alerter

How to develop a custom alerter

Base Abstraction

The base abstraction for alerters is very basic, as it only defines two abstract methods that subclasses should implement:

- `post()` takes a string, posts it to the desired chat service, and returns `True` if the operation succeeded, else `False`.
- `ask()` does the same as `post()`, but after sending the message, it waits until someone approves or rejects the operation from within the chat service (e.g., by sending "approve" / "reject" to the bot as response). `ask()` then only returns `True` if the operation succeeded and was approved, else `False`.

Then base abstraction looks something like this:

```
class BaseAlerter(StackComponent, ABC):
```

```python
    """Base class for all ZenML alerters."""

    def post(
        self, message: str, config: Optional[BaseAlerterStepConfig]
    ) -> bool:
        """Post a message to a chat service."""
        return True

    def ask(
        self, question: str, config: Optional[BaseAlerterStepConfig]
    ) -> bool:
        """Post a message to a chat service and wait for approval."""
        return True
```

This is a slimmed-down version of the base implementation. To see the full docstrings and imports, please check the source code on GitHub.

Building your own custom alerter

Creating your own custom alerter can be done in three steps:

1. Create a class that inherits from the `BaseAlerter`.
2. Define the `FLAVOR` class variable in your class (e.g., `FLAVOR="discord"`)
3. Implement the `post()` and `ask()` methods for your alerter.

Once you are done with the implementation, you can register it through the CLI as:

```
zenml alerter flavor register <THE-SOURCE-PATH-OF-YOUR-ALERTER>
```

Feature Stores

How to manage data in feature stores

Feature stores allow data teams to serve data via an offline store and an online low-latency store where data is kept in sync between the two. It also offers a centralized registry where features (and feature schemas) are stored for use within a team or wider organization.

As a data scientist working on training your model, your requirements for how you access your batch / 'offline' data will almost certainly be different from how you access that data as part of a real-time or online inference setting. Feast solves the problem of developing train-serve skew where those two sources of data diverge from each other.

Feature stores are a relatively recent addition to commonly-used machine learning stacks.

When to use it

The feature store is an optional stack component in the ZenML Stack. The feature store as a technology should be used to store the features and inject them into the process in the server-side. This includes

- Productionalize new features
- Reuse existing features across multiple pipelines and models
- Achieve consistency between training and serving data (Training Serving Skew)
- Provide a central registry of features and feature schemas

List of available feature stores

For production use cases, some more flavors can be found in specific `integrations` modules. In terms of features stores, ZenML features an integration of `feast`.

Feature Store	Flavor	Integration	Notes
FeastFeatureStore	`feast`	`feast`	Connect ZenML with already existing Feas
Custom Implementation	*custom*		Extend the feature store abstraction and provide your own implementation

If you would like to see the available flavors for feature stores, you can use the command:

```
zenml feature-store flavor list
```

How to use it

The available implementation of the feature store is built on top of the feast integration, which means that using a feature store is no different than what's described in the feast page: How to use it?.

Feast

How to manage data in Feast feature stores

Feast (Feature Store) is an operational data system for managing and serving machine learning features to models in production. Feast is able to serve feature data to models from a low-latency online store (for real-time prediction) or from an offline store (for scale-out batch scoring or model training).

When would you want to use it?

There are two core functions that feature stores enable:

- access to data from an offline / batch store for training.
- access to online data at inference time.

Feast integration currently supports your choice of offline data sources, and a Redis backend for your online feature serving. We encourage users to check out Feast's documentation and guides on how to set up your offline and online data sources via the configuration `yaml` file.

> COMING SOON: While the ZenML integration has an interface to access online feature store data, it currently is not usable in production settings with deployed models. We will update the docs when we enable this functionality.

How to deploy it?

The Feast Feature Store flavor is provided by the Feast ZenML integration, you need to install it, to be able to register it as a Feature Store and add it to your stack:

```
zenml integration install feast
```

Since this example is built around a Redis use case, a Python package to interact with Redis will get installed alongside Feast, but you will still first need to install Redis yourself. See this page for some instructions on how to do that on your operating system.

You will then need to run a Redis server in the background in order for this example to work. You can either use the redis-server command in your terminal (which will run a continuous process until you CTRL-C out of it), or you can run the daemonized version:

```
redis-server --daemonize yes

# verify it is running (Unix machines)
ps aux | grep redis-server
```

How do you use it?

ZenML assumes that users already have a feature store that they just need to connect with. The ZenML Online data retrieval is currently possible in a local setting, but we don't currently support using the online data serving in the context of a deployed model or as part of model deployment. We will update this documentation as we develop out this feature.

ZenML supports access to your feature store via a stack component that you can configure via the CLI tool. (See here for details on how to do that.)

Getting features from a registered and active feature store is possible by creating your own step that interfaces into the feature store:

```
from datetime import datetime
```

```python
from typing import Any, Dict, List, Union
import pandas as pd

from zenml.steps import BaseStepConfig, step, StepContext
entity_dict = {…}  # defined in earlier code
features = […]  # defined in earlier code

class FeastHistoricalFeaturesConfig(BaseStepConfig):
    """Feast Feature Store historical data step configuration."""

    entity_dict: Union[Dict[str, Any], str]
    features: List[str]
    full_feature_names: bool = False

    class Config:
        arbitrary_types_allowed = True


@step
def get_historical_features(
        config: FeastHistoricalFeaturesConfig,
        context: StepContext,
) -> pd.DataFrame:
    """Feast Feature Store historical data step


    Args:
        config: The step configuration.
        context: The step context.

    Returns:
        The historical features as a DataFrame.
    """
    if not context.stack:
        raise DoesNotExistException(
            "No active stack is available. Please make sure that you have registered and s
        )
    elif not context.stack.feature_store:
        raise DoesNotExistException(
            "The Feast feature store component is not available. "
            "Please make sure that the Feast stack component is registered as part of your
        )

    feature_store_component = context.stack.feature_store
    config.entity_dict["event_timestamp"] = [
        datetime.fromisoformat(val)
        for val in config.entity_dict["event_timestamp"]
    ]
    entity_df = pd.DataFrame.from_dict(config.entity_dict)

    return feature_store_component.get_historical_features(
        entity_df=entity_df,
        features=config.features,
    )   full_feature_names=config.full_feature_names,


historical_features = get_historical_features(
    config=FeastHistoricalFeaturesConfig(
        entity_dict=historical_entity_dict, features=features
    ),
)
```

Note that ZenML's use of Pydantic to serialize and deserialize inputs stored in the ZenML metadata means that we are limited to basic data types. Pydantic cannot handle Pandas `DataFrame` s, for example, or `datetime` values, so in the above code you can see that we have to convert them at various points.

A concrete example of using the Feast feature store can be found here.

For more information and a full list of configurable attributes of the Kubeflow orchestrator, check out the API Docs.

Develop a Custom Feature Store

How to develop a custom feature store

Feature stores allow data teams to serve data via an offline store, and an online low-latency store where data is kept in sync between the two. It also offers a centralized registry where features (and feature schemas) are stored for use within a team or wider organization.

Base abstraction in progress!

We are actively working on the base abstraction for the feature stores, which will be available soon. As a result, their extension is not possible at the moment. If you would like to use a feature store in your stack, please check the list of already available feature stores down below.

Annotators

How to annotate your data

Annotators are a stack component that enables the use of data annotation as part of your ZenML stack and pipelines. You can use the associated CLI command to launch annotation, configure your datasets and get stats on how many labeled tasks you have ready for use.

Data annotation/labeling is a core part of MLOps that is frequently left out of the conversation. ZenML will incrementally start to build in features that support an iterative annotation workflow that sees the people doing labeling (and their workflows/behaviors) as integrated parts of their ML process(es).

Could not load image

When and where to annotate.

There are a number of different places in the ML lifecycle where this can happen:

- **At the start**: You might be starting out without any data, or with a ton of data but no clear sense of which parts of it are useful to your particular problem. It's not uncommon to have a lot of data, but to be lacking accurate labels for that data. So you can start and get great value from bootstrapping your model: label some data, train your model, use your model to suggest labels allowing you to speed up your labeling, iterating on and on in this way. Labeling data early on in the process also helps clarify and condense down your specific rules and standards. For example, you might realize that you need to have specific definitions for certain concepts so that your labeling efforts are consistent across your team.
- **As new data comes in**: New data will likely continue to come in, and you might want to check in with the labeling process at regular intervals to expose yourself to this new data. (You'll probably also want to have some kind of automation around detecting data or concept drift, but for certain kinds of unstructured data you probably can never completely abandon the instant feedback of actual contact with the raw data.)
- **Samples generated for inference**: Your model will be making predictions on real-world data being passed in. If you store and label this data, you'll gain a valuable set of data which you can use to compare your labels with what the model was predicting, another possible way to flag drifts of various kinds. This data can then (subject to privacy / user consent) be used in retraining or fine-tuning your model.
- **Other ad hoc interventions**: You will probably have some kind of process to identify bad labels, or to find the kinds of examples that your model finds really difficult to make correct predictions. For these, and for areas where you have clear class imbalances, you might want to do ad hoc annotation to supplement the raw materials your model has to learn from.

ZenML currently offers standard steps that help you tackle the above use cases, but the stack component and abstraction will continue to be developed to make it easier to use.

When to use it

The annotator is an optional stack component in the ZenML Stack. We designed our abstraction to fit into the larger ML use cases, particularly the training and deployment parts of the lifecycle.

The core parts of the annotation workflow include:

- using labels or annotations in your training steps in a seamless way
- handling the versioning of annotation data
- allow for the conversion of annotation data to and from custom formats
- handle annotator-specific tasks, for example the generation of UI config files that Label Studio requires for the web annotation interface

List of available annotators

For production use cases, some more flavors can be found in specific `integrations` modules. In terms of annotators, ZenML features an integration with `label_studio`.

Annotator	Flavor	Integration	Notes
LabelStudioAnnotator	`label_studio`	`label_studio`	Connect ZenML with Label Studio
Custom Implementation	*custom*		Extend the annotator abstraction and provide your own implementation

If you would like to see the available flavors for annotators, you can use the command:

```
zenml annotator flavor list
```

How to use it

The available implementation of the annotator is built on top of the Label Studio integration, which means that using an annotator currently is no different than what's described in the Label Studio page: How to use it?.

A note on names

The various annotation tools have mostly standardized around the naming of key concepts as part of how they build their tools. Unfortunately, this hasn't been completely unified so ZenML takes an opinion on which names we use for our stack component and integrations. Key differences to note:

- Label Studio refers to the grouping of a set of annotations / tasks as a 'Project', whereas most other tools use the term 'Dataset', so ZenML also calls this grouping a 'Dataset'.
-

- The individual metaunit for 'an annotation + the source data' is referred to in different ways, but at ZenML (and with Label Studio) we refer to them as 'tasks'.

The remaining core concepts ('annotation' and 'prediction', in particular) are broadly used among annotation tools.

Label Studio

How to annotate data using Label Studio with ZenML

Label Studio is one of the leading open-source annotation platforms available to data scientists and ML practitioners. It is used to create or edit datasets that you can then use as part of training or validation workflows. It supports a broad range of annotation types, including:

- Computer Vision (image classification, object detection, semantic segmentation)
- Audio & Speech (classification, speaker diarization, emotion recognition, audio transcription)
- Text / NLP (classification, NER, question answering, sentiment analysis)
- Time Series (classification, segmentation, event recognition)
- Multi Modal / Domain (dialogue processing, OCR, time series with reference)

When would you want to use it?

If you need to label data as part of your ML workflow, that is the point at which you could consider adding in the optional annotator stack component as part of your ZenML stack.

We currently support the use of annotation at the various stages described in the main annotators docs page, and also offer custom utility functions to generate Label Studio label config files for image classification and object detection. (More will follow in due course.)

The Label Studio integration currently is built to support workflows using the following three cloud artifact stores: AWS S3, GCP/GCS and Azure Blob Storage. Purely local stacks will currently *not* work if you want to do add the annotation stack component as part of your stack.

> COMING SOON: The Label Studio Integration supports the use of annotations in an ML workflow, but we do not currently handle the universal conversion between data formats as part of the training workflow. Our initial use case was built to support image classification and object detection, but we will add helper steps and functions for other use cases in due course. We will update the docs when we enable this functionality.

How to deploy it?

The Label Studio Annotator flavor is provided by the Label Studio ZenML integration, you need to install it, to be able to register it as an Annotator and add it to your stack:

```
zenml integration install label_studio
```

Before registering a `label_studio` flavor stack component as part of your stack, you'll need to have registered a cloud artifact store and a secrets manager to handle authentication with Label Studio as well as any secrets required for the Artifact Store. (See the docs on how to register and setup a cloud artifact store as well as a secrets manager.)

Be sure to register an secret schema for whichever artifact store you choose, and then you should make sure to pass the name of that secret into the artifact store as the `--authentication_secret` as described in this guide, for example in the case of AWS.

You will next need to obtain your Label Studio API key. This will give you access to the web annotation interface.

```
# choose a username and password for your label-studio account
label-studio reset_password --username <USERNAME> --password <PASSWORD>

# start a temporary / one-off label-studio instance to get your API key
label-studio start -p 8094
```

Then visit http://localhost:8094/ to log in, and then visit http://localhost:8094/user/account and get your Label Studio API key (from the upper right hand corner). You will need it for the next step. `Ctrl-c` out of the Label Studio server that is running on the terminal.

At this point you should register the API key with your secrets manager under a custom secret name, making sure to replace the two parts in `<>` with whatever you choose:

```
zenml secret register <LABEL_STUDIO_SECRET_NAME> --api_key="<your_label_studio_api_key>"
```

Then register your annotator with ZenML:

```
zenml annotator register label_studio --flavor label_studio --authentication_secret="<LABEL
```

Finally, add all these components to a stack and set it as your active stack. For example:

```
zenml stack copy annotation
zenml stack update annotation -x <YOUR_SECRETS_MANAGER> -a <YOUR_CLOUD_ARTIFACT_STORE>
# this must be done separately so that the other required stack components are first regis
zenml stack update annotation -an <YOUR_LABEL_STUDIO_ANNOTATOR>
zenml stack set annotation
# optionally also
zenml stack describe
```

Now if you run a simple CLI command like `zenml annotator dataset list` this should work without any errors. You're ready to use your annotator in your ML workflow!

How do you use it?

ZenML assumes that users have registered a cloud artifact store, a secrets manager and an annotator as described above. ZenML currently only supports this setup, but we will add in the fully local stack option in the future.

ZenML supports access to your data and annotations via the `zenml annotator...` CLI command.

You can access information about the datasets you're using with the `zenml annotator dataset list`. To work on annotation for a particular dataset, you can run `zenml annotator dataset annotate <dataset_name>`.

Our full continuous annotation / training example is the best place to see how all the pieces of making this integration work fit together. What follows is an overview of some of the key components to the Label Studio integration and how it can be used.

Label Studio Annotator Stack Component

Our Label Studio annotator component inherits from the `BaseAnnotator` class. There are some methods that are core methods that must be defined, like being able to register or get a dataset. Most annotators handle things like the storage of state and have their own custom features, so there are quite a few extra methods specific to Label Studio.

The core Label Studio functionality that's currently enabled includes a way to register your datasets, export any annotations for use in separate steps as well as to start the annotator daemon process. (Label Studio requires a server to be running in order to use the web interface, and ZenML handles the provisioning of this server locally using the details you passed in when registering the component.)

Standard Steps

ZenML offers some standard steps (and their associated config objects) which will get you up and running with the Label Studio integration quickly. These include:

- `LabelStudioDatasetRegistrationConfig` - a step config object to be used when registering a dataset with Label studio using the `get_or_create_dataset` step
- `LabelStudioDatasetSyncConfig` - a step config object to be used when registering a dataset with Label studio using the `sync_new_data_to_label_studio` step. Note that this requires a secret schema to have been pre-registered with your artifact store as being the one that holds authentication secrets specific to your particular cloud provider. (Label Studio provides some documentation on what permissions these secrets require here.)
- `get_or_create_dataset` step - This takes a `LabelStudioDatasetRegistrationConfig` config object which includes the name of the dataset. If it exists, this step will return the name, but if it doesn't exist then ZenML will register the dataset along with the appropriate label config with Label Studio.
- `get_labeled_data` step - This step will get all labeled data available for a particular dataset. Note that these are output in a Label Studio annotation format, which will subsequently converted into a format appropriate for your specific use case.
- `sync_new_data_to_label_studio` step - This step is for ensuring that ZenML is handling the annotations and the files being used are stored and synced with the ZenML cloud artifact store. This is

an important step as part of a continuous annotation workflow since you want all the subsequent steps of your workflow to remain in sync with whatever new annotations are being made or have been created.

Helper Functions

Label Studio requires the use of what it calls 'label config' when you are creating/registering your dataset. These are strings containing HTML-like syntax that allow you to define a custom interface for your annotation. ZenML provides two helper functions that will construct these label config strings in the case of object detection and image classification. See the `integrations.label_studio.label_config_generators` module for those two functions.

A concrete example of using the Label Studio annotator can be found here.

Develop a Custom Annotator

How to develop a custom annotator

Annotators are a stack component that enables the use of data annotation as part of your ZenML stack and pipelines. You can use the associated CLI command to launch annotation, configure your datasets and get stats on how many labeled tasks you have ready for use.

Base abstraction in progress!

We are actively working on the base abstraction for the annotators, which will be available soon. As a result, their extension is not possible at the moment. If you would like to use an annotator in your stack, please check the list of already available feature stores down below.

Cloud Guide

Overview: Options for Deploying Infrastructure

An overview of cloud infrastructure for your ML workflows

There can be many motivations behind taking your ML application setup to a cloud environment, from needing specialized compute for training jobs to having a 24x7 load-balanced deployment of your trained model, serving user requests . Whatever your reasons may be, we believe that the process of infrastructure setup should be as simple as possible to allow you to focus more on building your ML solution and less on battling irrelevant cloud woes.

This cloud guide has a collection of an increasing number of tutorials that you can follow to set up some common stacks to run your ZenML pipelines on. It has three sections as of now, one for each of the major cloud providers.

Before you dive into the tutorials however, save yourself some time and frustration by reading the following section to know what deployment option would be best-suited for you!

Options for setting up your infrastructure

This section aims to list out different scenarios that you may find yourself in (subject to your background) and helps you to navigate between the available choices.

□ I'm experienced in operations and prefer having the knowledge of every cloud resource that is being set up, intimately.

We have a step-by-step guide for AWS and GCP that allows you to follow along and create the necessary infrastructure for your ZenML stack.

The guides have detailed information on every stack component along with other essential things like roles, permissions, connection settings and more. Head over to the page corresponding to your cloud provider to get started!

□ I have never worked on the infrastructure side of things before and I don't want to invest my time in it. Is there another way?

There certainly is! You can check out our open-source repository mlops-stacks that has a number of different "recipes", each of which can set up a specialized stack with the execution of just two commands.

The `README` file in the project lays out exactly what you need to know and what values you can customize for each recipe while creating your setup. It's also the fastest way to get ready for running your ZenML pipelines □

□ Not really sure that I want to install a new tool in my workflow. Is there a way to achieve a similar ease of setup without the use of Terraform?

We've got you covered, yet again! Within each cloud guide you can find a set of provider-native CLI commands that you can leverage for the same outcome.

You can just copy and paste each command with minimal modifications for customizing your setup and you're set! □

□ I can't really decide between these options. What is the recommended approach?

ZenML recommends the use of the stack recipes inside the `mlops-stacks` repository as the fastest and most reliable way to set up your stack. It has the following advantages:

- Easy and fast deployment with just two commands □.
- Simple and efficient to modify any of the components: You can change properties in a config file and Terraform identifies and applies only the relevant changes □.
- Comprehensive clean-up: `terraform destroy` completely deletes any resources that the module had created anywhere in your cloud. Thus, you can be sure that there are no extra bills that you will tragically discover going forward □.

More recipes are always coming up! If you feel there's one missing, raise an issue and feel free to create a PR too; we are here for the support.

AWS

How to set up stacks on Amazon Web Services (AWS)

AWS is one of the most popular cloud providers and offers a range of services that can be used while building your MLOps stacks. You can learn more about machine learning at AWS on their website.

Available Stack Components

This is a list of all supported AWS services that you can use as ZenML stack components.

Elastic Kubernetes Service (EKS)

Amazon Elastic Kubernetes Service (Amazon EKS) is a managed container service to run and scale Kubernetes applications in the cloud or on-premises. Learn more here.

- An EKS cluster can be used to run multiple **orchestrators**.
  - A Kubernetes-native orchestrator.
  - A Kubeflow orchestrator.
- You can host **model deployers** on the cluster.
  - A Seldon model deployer.
  - An MLflow model deployer.
- Experiment trackers can also be hosted on the cluster.
  - An MLflow experiment tracker

Simple Storage Service (S3)

Amazon Simple Storage Service (Amazon S3) is an object storage service that offers scalability, data availability, security, and performance. Learn more here.

- You can use an S3 bucket as an artifact store to hold files from our pipeline runs like models, data and more.

Elastic Container Registry (ECR)

Amazon Elastic Container Registry (Amazon ECR) is an AWS managed container image registry service that is secure, scalable, and reliable. Learn more here.

- An ECS registry can be used as a container registry stack component to host images of your pipelines.

SageMaker

Amazon SageMaker is a fully managed machine learning service. With SageMaker, data scientists and developers can quickly build and train machine learning models, and then directly deploy them into a production-ready hosted environment. Learn more here.

- You can use SageMaker as a step operator to run specific steps from your pipeline using it as the backend.

Relational Database Service (RDS)

Amazon Relational Database Service (Amazon RDS) is a web service that makes it easier to set up, operate, and scale a relational database in the AWS Cloud. Learn more here.

- You can use Amazon RDS as a metadata store to track metadata from your pipeline runs.

Secrets Manager

Secrets Manager enables you to replace hardcoded credentials in your code, including passwords, with an API call to Secrets Manager to retrieve the secret programmatically. Learn more here.

- You can store your secrets to be used inside a pipeline by registering the AWS Secrets Manager as a ZenML secret manager stack component.

In the following pages, you will find step-by-step guides for setting up some common stacks using the AWS console and the CLI. More combinations and components are progressively updated in the form of new pages.

Set Up a Minimal MLOps Stack on AWS

How to set up a minimal stack on Amazon Web Services (AWS)

To get started using ZenML on the cloud, you need some basic infrastructure up and running which ZenML can use to run your pipelines. This step-by-step guide explains how to set up a basic cloud stack on AWS.

> This guide represents **one** of many ways to create a cloud stack on AWS. You can customize this by adding additional components of replacing one of the components described in this guide.

Prerequisites

- Docker installed and running.

- kubectl installed.
- The AWS CLI installed and authenticated.
- ZenML and the integrations for this tutorial stack installed:

```
pip install zenml
zenml integration install aws s3 kubernetes
```

Setting up the AWS resources

All the AWS setup steps can either be done using the AWS UI or CLI. Simply select the tab for your preferred option and let's get started. First open up a terminal which we'll use to store some values along the way which we'll need to configure our ZenML stack later.

Artifact Store (S3 bucket)

AWS UI

- Go to the S3 website.
- Click on `Create bucket`.
- Select a descriptive name and a region. Let's also store these values in our terminal:

```
REGION=<REGION> # for example us-west-1
S3_BUCKET_NAME=<S3_BUCKET_NAME>
```

AWS CLI

```
# Set a name for your bucket and the AWS region for your resources
# Select one of the region codes for <REGION>: https://docs.aws.amazon.com/gener
REGION=<REGION>
S3_BUCKET_NAME=zenml-artifact-store

aws s3api create-bucket --bucket=$S3_BUCKET_NAME \
    --region=$REGION \
    --create-bucket-configuration=LocationConstraint=$REGION
```

Metadata Store (RDS MySQL database)

AWS UI

- Go to the RDS website.
- Make sure the correct region is selected on the top right (this region must be the same for all following steps).
- Click on `Create database`.
- Select `Easy Create`, `MySQL`, `Free tier` and enter values for your database name, username and password.
- Note down the username and password:

```
RDS_MYSQL_USERNAME=<RDS_MYSQL_USERNAME>
RDS_MYSQL_PASSWORD=<RDS_MYSQL_PASSWORD>
```

- Wait until the deployment is finished.
- Select your new database and note down its endpoint:

```
RDS_MYSQL_ENDPOINT=<RDS_MYSQL_ENDPOINT>
```

- Click on the active VPC security group, select `Inbound rules` and click on `Edit inbound rules`
- Add a new rule with type `MYSQL/Aurora` and source `Anywhere-IPv4`. (**Note**: You can also restrict this to more limited IP address ranges or security groups if you want to limit access to your database.)
- Go back to your database page and click on `Modify` in the top right.
- In the `Connectivity` section, open the `Advanced configuration` and enable public access.

AWS CLI

```
# Set values for the database identifier and username/password to access it
MYSQL_DATABASE_ID=zenml-metadata-store
RDS_MYSQL_USERNAME=admin
RDS_MYSQL_PASSWORD=<RDS_MYSQL_PASSWORD>

aws rds create-db-instance --engine=mysql \
    --db-instance-class=db.t3.micro \
    --allocated-storage=20 \
    --publicly-accessible \
    --db-instance-identifier=$MYSQL_DATABASE_ID \
    --region=$REGION \
    --master-username=$RDS_MYSQL_USERNAME \
    --master-user-password=$RDS_MYSQL_PASSWORD

# Wait until the database is created
aws rds wait db-instance-available --db-instance-identifier=$MYSQL_DATABASE_ID \
    --region=$REGION

```

```
# Fetch the endpoint
RDS_MYSQL_ENDPOINT=$(aws rds describe-db-instances --query='DBInstances[0].Endpo
    --output=text \
    --db-instance-identifier=$MYSQL_DATABASE_ID \
    --region=$REGION)

# Fetch the security group id
SECURITY_GROUP_ID=$(aws rds describe-db-instances --query='DBInstances[0].VpcSec
    --output=text
    --db-instance-identifier=$MYSQL_DATABASE_ID \
    --region=$REGION)

aws ec2 authorize-security-group-ingress \
    --protocol=tcp \
    --port=3306 \
    --cidr=0.0.0.0/0 \
    --group-id=$SECURITY_GROUP_ID \
    --region=$REGION
```

Container Registry (ECR)

AWS UI

- Go to the ECR website.
- Make sure the correct region is selected on the top right.
- Click on `Create repository`.
- Create a private repository called `zenml-kubernetes` with default settings.
- Note down the URI of your registry:

```
# This should be the prefix of your just created repository URI,
# e.g. 714803424590.dkr.ecr.eu-west-1.amazonaws.com
ECR_URI=<ECR_URI>
```

AWS CLI

```
aws ecr create-repository --repository-name=zenml-kubernetes --region=$REGION

REGISTRY_ID=$(aws ecr describe-registry --region=$REGION --query=registryId --ou
ECR_URI="$REGISTRY_ID.dkr.ecr.$REGION.amazonaws.com"
```

Orchestrator (EKS)

AWS UI

- Follow this guide to create an Amazon EKS cluster role. We'll refer to this role as the **cluster role** in following steps.
- Follow this guide to create an Amazon EC2 node role. We'll refer to this role as the **node role** in following steps.
- Go to the IAM website, select `Roles` and edit the **node role** role.
- Click on `Add permissions` and select `Attach policies`.
- Attach the `SecretsManagerReadWrite`, and `AmazonS3FullAccess` policies to the role. The node role should now have the following attached policies: `AmazonEKSWorkerNodePolicy`, `AmazonEC2ContainerRegistryReadOnly`, `AmazonEKS_CNI_Policy`, `SecretsManagerReadWrite` and `AmazonS3FullAccess`.
- Go to the EKS website.
- Make sure the correct region is selected on the top right.
- Click on `Add cluster` and select `Create`.
- Enter a name and select the **cluster role** for `Cluster service role`.
- Keep the default values for the networking and logging steps and create the cluster.
- Note down the cluster name:

```
EKS_CLUSTER_NAME=<EKS_CLUSTER_NAME>
```

- After the cluster is created, select it and click on `Add node group` in the `Compute` tab.
- Enter a name and select the **node role**.
- Keep all other default values and create the node group.

AWS CLI

```
# Choose names for your EKS cluster, node group and their respective roles
EKS_CLUSTER_NAME=zenml-eks-cluster
NODEGROUP_NAME=zenml-eks-cluster-nodes
EKS_ROLE_NAME=ZenMLEKSRole
EC2_ROLE_NAME=ZenMLEKSNodeRole

EKS_POLICY_JSON='{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "Service": "eks.amazonaws.com"
      },
      "Action": "sts:AssumeRole"
    }
  ]
}'
aws iam create-role \
    --role-name=$EKS_ROLE_NAME \
    --assume-role-policy-document="$EKS_POLICY_JSON"
aws iam attach-role-policy \
    --policy-arn='arn:aws:iam::aws:policy/AmazonEKSClusterPolicy' \
    --role-name=$EKS_ROLE_NAME


EC2_POLICY_JSON='{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "Service": "ec2.amazonaws.com"
      },
      "Action": "sts:AssumeRole"
    }
  ]
}'
aws iam create-role \
    --role-name=$EC2_ROLE_NAME \
    --assume-role-policy-document="$EC2_POLICY_JSON"
aws iam attach-role-policy \
    --policy-arn='arn:aws:iam::aws:policy/AmazonEKSWorkerNodePolicy' \
    --role-name=$EC2_ROLE_NAME
aws iam attach-role-policy \
    --policy-arn='arn:aws:iam::aws:policy/AmazonEC2ContainerRegistryReadOnly' \
    --role-name=$EC2_ROLE_NAME
aws iam attach-role-policy \
    --policy-arn='arn:aws:iam::aws:policy/AmazonEKS_CNI_Policy' \
    --role-name=$EC2_ROLE_NAME
aws iam attach-role-policy \
    --policy-arn='arn:aws:iam::aws:policy/SecretsManagerReadWrite' \
    --role-name=$EC2_ROLE_NAME
aws iam attach-role-policy \
    --policy-arn='arn:aws:iam::aws:policy/AmazonS3FullAccess' \
    --role-name=$EC2_ROLE_NAME


# Get the role ARN's
EKS_ROLE_ARN=$(aws iam get-role --role-name=$EKS_ROLE_NAME --query='Role.Arn' --
EC2_ROLE_ARN=$(aws iam get-role --role-name=$EC2_ROLE_NAME --query='Role.Arn' --


# Get default VPC ID
VPC_ID=$(aws ec2 describe-vpcs --filters='Name=is-default,Values=true' \
    --query='Vpcs[0].VpcId' \
    --output=text \
    --region=$REGION)

# Get subnet IDs
SUBNET_IDS=$(aws ec2 describe-subnets --region=$REGION \
    --filters="Name=vpc-id,Values=$VPC_ID" \
    --query='Subnets[*].SubnetId' \
    --output=json)

aws eks create-cluster --region=$REGION \
    --name=$EKS_CLUSTER_NAME \
    --role-arn=$EKS_ROLE_ARN \
    --resources-vpc-config="{\"subnetIds\": $SUBNET_IDS}"

# Wait until the cluster is active
aws eks wait cluster-active --name=$EKS_CLUSTER_NAME \
    --region=$REGION

aws eks create-nodegroup --region=$REGION \
    --cluster-name=$EKS_CLUSTER_NAME \
    --nodegroup-name=$NODEGROUP_NAME \
    --node-role=$EC2_ROLE_ARN \
    --subnets="$SUBNET_IDS"

# Wait until the node group is active
aws eks wait nodegroup-active --cluster-name=$EKS_CLUSTER_NAME \
    --nodegroup-name=$NODEGROUP_NAME \
    --region=$REGION
```

Register the ZenML stack

- Register the artifact store:

```
zenml artifact-store register s3_store \
    --flavor=s3 \
    --path=s3://$S3_BUCKET_NAME
```

- Register the container registry and authenticate your local docker client

```
zenml container-registry register ecr_registry \
    --flavor=aws \
    --uri=$ECR_URI

aws ecr get-login-password --region $REGION | docker login --username AWS --password-st
```

- Register the metadata store:

```
zenml metadata-store register rds_mysql \
    --flavor=mysql \
    --database=zenml \
    --secret=rds_authentication \
    --host=$RDS_MYSQL_ENDPOINT
```

- Register the secrets manager:

```
zenml secrets-manager register aws_secrets_manager \
    --flavor=aws \
    --region_name=$REGION
```

- Configure your `kubectl` client and register the orchestrator:

```
aws eks --region=$REGION update-kubeconfig --name=$EKS_CLUSTER_NAME
kubectl create namespace zenml

zenml orchestrator register eks_kubernetes_orchestrator \
    --flavor=kubernetes \
    --kubernetes_context=$(kubectl config current-context)
```

- Register the ZenML stack and activate it:

```
zenml stack register kubernetes_stack \
    -o eks_kubernetes_orchestrator \
    -a s3_store \
    -m rds_mysql \
    -c ecr_registry \
    -x aws_secrets_manager \
    --set
```

- Register the secret for authenticating with your MySQL database:

```
zenml secret register rds_authentication \
    --schema=mysql \
    --user=$RDS_MYSQL_USERNAME \
    --password=$RDS_MYSQL_PASSWORD
```

After all of this setup, you're now ready to run any ZenML pipeline on AWS!

Quick setup

If you're looking for a way to get started quickly, we've combined all the commands so you can copy-paste them and execute them in a single go. You'll only need to set values for the `<REGION>` and `<RDS_MYSQL_PASSWORD>` right at the beginning before executing the rest.

Quick setup commands

```
# Select one of the region codes for <REGION>: https://docs.aws.amazon.com/general
REGION=<REGION>
# Choose a secure password for your database admin account. Make sure it includes:
```

```
# - at least 8 printable ASCII characters
# - no slash, single or double quotes or @ signs
RDS_MYSQL_PASSWORD=<RDS_MYSQL_PASSWORD>

# Other parameters (we've set some defaults for these but feel free to change them
S3_BUCKET_NAME=zenml-artifact-store
MYSQL_DATABASE_ID=zenml-metadata-store
RDS_MYSQL_USERNAME=admin
EKS_CLUSTER_NAME=zenml-eks-cluster
NODEGROUP_NAME=zenml-eks-cluster-nodes
EKS_ROLE_NAME=ZenMLEKSRole
EC2_ROLE_NAME=ZenMLEKSNodeRole


aws s3api create-bucket --bucket=$S3_BUCKET_NAME \
    --region=$REGION \
    --create-bucket-configuration=LocationConstraint=$REGION

aws rds create-db-instance --engine=mysql \
    --db-instance-class=db.t3.micro \
    --allocated-storage=20 \
    --publicly-accessible \
    --db-instance-identifier=$MYSQL_DATABASE_ID \
    --region=$REGION \
    --master-username=$RDS_MYSQL_USERNAME \
    --master-user-password=$RDS_MYSQL_PASSWORD

# Wait until the database is created
aws rds wait db-instance-available --db-instance-identifier=$MYSQL_DATABASE_ID \
    --region=$REGION

# Fetch the endpoint
RDS_MYSQL_ENDPOINT=$(aws rds describe-db-instances --query='DBInstances[0].Endpoir
    --output=text \
    --db-instance-identifier=$MYSQL_DATABASE_ID \
    --region=$REGION)

# Fetch the security group id
SECURITY_GROUP_ID=$(aws rds describe-db-instances --query='DBInstances[0].VpcSecur
    --output=text
    --db-instance-identifier=$MYSQL_DATABASE_ID \
    --region=$REGION)

aws ec2 authorize-security-group-ingress \
    --protocol=tcp \
    --port=3306 \
    --cidr=0.0.0.0/0 \
    --group-id=$SECURITY_GROUP_ID \
    --region=$REGION

aws ecr create-repository --repository-name=zenml-kubernetes --region=$REGION

REGISTRY_ID=$(aws ecr describe-registry --region=$REGION --query=registryId --outp
ECR_URI="$REGISTRY_ID.dkr.ecr.$REGION.amazonaws.com"
EKS_POLICY_JSON='{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "Service": "eks.amazonaws.com"
      },
      "Action": "sts:AssumeRole"
    }
  ]
}'
aws iam create-role \
    --role-name=$EKS_ROLE_NAME \
    --assume-role-policy-document="$EKS_POLICY_JSON"
aws iam attach-role-policy \
    --policy-arn='arn:aws:iam::aws:policy/AmazonEKSClusterPolicy' \
    --role-name=$EKS_ROLE_NAME


EC2_POLICY_JSON='{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "Service": "ec2.amazonaws.com"
      },
      "Action": "sts:AssumeRole"
    }
  ]
}'
aws iam create-role \
    --role-name=$EC2_ROLE_NAME \
    --assume-role-policy-document="$EC2_POLICY_JSON"
aws iam attach-role-policy \
    --policy-arn='arn:aws:iam::aws:policy/AmazonEKSWorkerNodePolicy' \
    --role-name=$EC2_ROLE_NAME
aws iam attach-role-policy \
    --policy-arn='arn:aws:iam::aws:policy/AmazonEC2ContainerRegistryReadOnly' \
    --role-name=$EC2_ROLE_NAME
aws iam attach-role-policy \
    --policy-arn='arn:aws:iam::aws:policy/AmazonEKS_CNI_Policy' \
    --role-name=$EC2_ROLE_NAME
aws iam attach-role-policy \
    --policy-arn='arn:aws:iam::aws:policy/SecretsManagerReadWrite' \
    --role-name=$EC2_ROLE_NAME
aws iam attach-role-policy \
    --policy-arn='arn:aws:iam::aws:policy/AmazonS3FullAccess' \
    --role-name=$EC2_ROLE_NAME

```

```
# Get the role ARN's
EKS_ROLE_ARN=$(aws iam get-role --role-name=$EKS_ROLE_NAME --query='Role.Arn' --ou
EC2_ROLE_ARN=$(aws iam get-role --role-name=$EC2_ROLE_NAME --query='Role.Arn' --ou


# Get default VPC ID
VPC_ID=$(aws ec2 describe-vpcs --filters='Name=is-default,Values=true' \
    --query='Vpcs[0].VpcId' \
    --output=text \
    --region=$REGION)

# Get subnet IDs
SUBNET_IDS=$(aws ec2 describe-subnets --region=$REGION \
    --filters="Name=vpc-id,Values=$VPC_ID" \
    --query='Subnets[*].SubnetId' \
    --output=json)

aws eks create-cluster --region=$REGION \
    --name=$EKS_CLUSTER_NAME \
    --role-arn=$EKS_ROLE_ARN \
    --resources-vpc-config="{\"subnetIds\": $SUBNET_IDS}"

# Wait until the cluster is active
aws eks wait cluster-active --name=$EKS_CLUSTER_NAME \
    --region=$REGION

aws eks create-nodegroup --region=$REGION \
    --cluster-name=$EKS_CLUSTER_NAME \
    --nodegroup-name=$NODEGROUP_NAME \
    --node-role=$EC2_ROLE_ARN \
    --subnets="$SUBNET_IDS"

# Wait until the node group is active
aws eks wait nodegroup-active --cluster-name=$EKS_CLUSTER_NAME \
    --nodegroup-name=$NODEGROUP_NAME \
    --region=$REGION

# ZenML stack setup
zenml artifact-store register s3_store \
    --flavor=s3 \
    --path=s3://$S3_BUCKET_NAME

zenml container-registry register ecr_registry \
    --flavor=aws \
    --uri=$ECR_URI

aws ecr get-login-password --region $REGION | docker login --username AWS --passwo

zenml metadata-store register rds_mysql \
    --flavor=mysql \
    --database=zenml \
    --secret=rds_authentication \
    --host=$RDS_MYSQL_ENDPOINT
zenml secrets-manager register aws_secrets_manager \
    --flavor=aws \
    --region_name=$REGION

aws eks --region=$REGION update-kubeconfig --name=$EKS_CLUSTER_NAME
kubectl create namespace zenml

zenml orchestrator register eks_kubernetes_orchestrator \
    --flavor=kubernetes \
    --kubernetes_context=$(kubectl config current-context)

zenml stack register kubernetes_stack \
        -o eks_kubernetes_orchestrator \
        -a s3_store \
        -m rds_mysql \
        -c ecr_registry \
        -x aws_secrets_manager \
        --set

zenml secret register rds_authentication \
        --schema=mysql \
        --user=$RDS_MYSQL_USERNAME \
        --password=$RDS_MYSQL_PASSWORD
```

GCP

How to set up stacks on Google Cloud Platform (GCP)

GCP is one of the most popular cloud providers and offers a range of services that can be used while building your MLOps stacks. You can learn more about machine learning at GCP on their website.

Available Stack Components

This is a list of all supported GCP services that you can use as ZenML stack components.

Google Kubernetes Engine (GKE)

Google Kubernetes Engine (GKE) provides a managed environment for deploying, managing, and scaling your containerized applications using Google infrastructure. The GKE environment consists of multiple machines (specifically, Compute Engine instances) grouped together to form a cluster. Learn more here.

- An GKE cluster can be used to run multiple **orchestrators**.
    - A Kubernetes-native orchestrator.

  - A Kubeflow orchestrator.
- You can host **model deployers** on the cluster.
  - A Seldon model deployer.
  - An MLflow model deployer.
- Experiment trackers can also be hosted on the cluster.
  - An MLflow experiment tracker

Cloud Storage Bucket (GCS)

Cloud Storage is a service for storing your objects in Google Cloud. An object is an immutable piece of data consisting of a file of any format. You store objects in containers called buckets. Learn more here.

- You can use a GCS bucket as an artifact store to hold files from our pipeline runs like models, data and more.

Google Container Registry (GCR)

Container Registry is a service for storing private container images. It is being deprecated in favor of Artifact Registry, support for which will be coming soon to ZenML!

- A GCR registry can be used as a container registry stack component to host images of your pipelines.

Vertex AI

Vertex AI brings together the Google Cloud services for building ML under one, unified UI and API. In Vertex AI, you can now train and compare models using AutoML or custom code training and all your models are stored in one central model repository. Learn more here.

- You can use Vertex AI as a step operator to run specific steps from your pipeline using it as the backend.
- Vertex AI can also be used as an orchestrator for your pipelines.

CloudSQL

Cloud SQL is a fully-managed database service that helps you set up, maintain, manage, and administer your relational databases on Google Cloud Platform. You can use Cloud SQL with a MySQL server in ZenML. Learn more here.

- You can use a CloudSQL MySQL instance as a metadata store to track metadata from your pipeline runs.

Secret Manager

Secret Manager is a secure and convenient storage system for API keys, passwords, certificates, and other sensitive data. Secret Manager provides a central place and single source of truth to manage, access, and audit secrets across Google Cloud. Learn more here.

- You can store your secrets to be used inside a pipeline by registering the Google Secret Manager as a ZenML secret manager stack component.

In the following pages, you will find step-by-step guides for setting up some common stacks using the GCP console and the CLI. More combinations and components are progressively updated in the form of new pages.

Set Up a Minimal MLOps Stack on GCP

How to set up a minimal stack on Google Cloud Platform (GCP)

To get started using ZenML on the cloud, you need some basic infrastructure up and running that you can then make more complicated depending on your use-case. This guide sets up the easiest MLOps stack that we can run on GCP with ZenML.

> This guide represents **one** of many ways to create a cloud stack on GCP. Every component could be replaced by a different implementation. Feel free to take this as your starting point.

Prerequisites

For this to work you need to have ZenML installed locally with all GCP requirements.

```
pip install zenml
zenml integration install gcp
```

Additionally, you will need Docker installed on your system.

The cloud stack

A full cloud stack will necessarily contain these five stack components:

- An **artifact store** to save all step output artifacts, in this guide we will use a GCP bucket for this purpose
- A **metadata store** that keeps track of the relationships between artifacts, runs and parameters. In our case we will opt for a MySQL database on GCP Cloud SQL.
- The **orchestrator** to run the pipelines. Here we will opt for a Vertex AI pipelines orchestrator. This is a serverless GCP specific offering with minimal hassle.
- A **container registry** for pushing and pulling the pipeline image.
-

Finally, the **secrets Manager** to store passwords and SSL certificates.

Set Up `gcloud` CLI

Install the `gcloud` CLI on your machine. Here is a guide on how to install it.

```
gcloud auth
```

Set up a GCP project (Optional)

GCP UI

As a first step it might make sense to create a separate GCP project for your ZenML resources. However, this step is completely optional, and you can also move forward within an existing project. If some resources already exist, feel free to skip their creation step and just note down the relevant information.

For simplicity, just open up a terminal on the side and save relevant values as we go along. You will use these when we set up the ZenML stack. ZenML will use your project number at a later stage to connect to some resources, so let's it. You'll most probably find it right here.

```
PROJECT_NUMBER=<PROJECT_NUMBER> # for example '492014921912'
GCP_LOCATION=<GCP_LOCATION> # for example 'europe-west3'
```

gcloud CLI

```
PARENT_ORG_ID=<PARENT_ORG_ID> # for example 3928562984638
PROJECT_NAME=<PROJECT_NAME> # for example zenml-vertex-prj
GCP_LOCATION=<GCP_LOCATION> # for example 'europe-west3'

gcloud projects create $PROJECT_NAME --organization=$PARENT_ORG_ID
gcloud config set project $PROJECT_NAME
PROJECT_NUMBER=$(gcloud projects describe $PROJECT_NAME --format="value(projectN
```

Enable billing

Before moving on, you'll have to make sure you attach a billing account to your project. In case you do not have the permissions to do so, you'll have to ask an organization administrator.

GCP UI

Here is a relevant page.

gcloud CLI

In case you don't have permissions on your companies billing account you might need to ask your admin to do this for you.

```
BILLING_ACC=<BILLING_ACC>
```

```
gcloud beta billing projects link $PROJECT_NAME --billing-account $BILLING_ACC
```

Enable Vertex AI

Vertex AI pipelines is at the heart of our GCP stack. As the orchestrator Vertex AI will run your pipelines and use all the other stack components.

GCP UI

All you'll need to do at this stage is enable Vertex AI here.

gcloud CLI

```
gcloud services enable aiplatform.googleapis.com
```

Enable Secrets Manager

The Secrets Manager will be needed so that the orchestrator will have secure access to the other resources.

GCP UI

Here is where you'll be able to enable the secrets manager.

gcloud CLI

```
gcloud services enable secretmanager.googleapis.com
```

Enable Container Registry

The Vertex AI orchestrator uses Docker Images containing your pipeline code for pipeline orchestration.

GCP UI

For this to work you'll need to enable the GCP Docker registry here.

In order to use the container registry at a later point you will need to set the container registry URI. This is how it is usually constructed: `gcr.io/<PROJECT_ID>`.

> The container registry has four options: `gcr.io` , `us.gcr.io`, `eu.gcr.io` , or `asia.gcr.io`. Choose the one appropriate for you.

```
CONTAINER_REGISTRY_URI=<CONTAINER_REGISTRY_URI> # for example 'eu.gcr.io/zenml-p
```

gcloud CLI

```
CONTAINER_REGISTRY_REGION=<CONTAINER_REGISTRY_REGION> # can be 'eu', 'us', 'asia
gcloud services enable containerregistry.googleapis.com
CONTAINER_REGISTRY_URI=$CONTAINER_REGISTRY_REGION".gcr.io/"$PROJECT_NAME
```

Set up Cloud Storage as Artifact Store

Storing of step artifacts is an important part of reproducible MLOps.

GCP UI

Create a bucket here.

Within the configuration of the newly created bucket you can find the gsutil URI which you will need at a later point. It's usually going to look like this: `gs://<bucket-name>`

```
GSUTIL_URI=<GSUTIL_URI> # for example 'gs://zenml_vertex_storage'
```

gcloud CLI

```
GSUTIL_URI=gs://<BUCKET-NAME>
gsutil mb -p $PROJECT_NAME $GSUTIL_URI
```

Set up a Cloud SQL instance as Metadata Store

One of the most complex resources that you'll need to manage is the MySQL database.

GCP UI

To start, we create a MySQL database. Once created, it will take some time for the database to be set up.

Once it is set up you can find the IP-address. The password you set during creation of the instance is the root password. The default port for MySQL is 3306.

```
DB_HOST=<DB_HOST> # for example '35.137.24.15'
DB_PWD=<DB_PWD> # for example 'secure_root_pwd'
```

Time to set up the connections to our database. To do this you'll need to go into the `Connections` menu. Under the `Networking` tab you'll need to add **0.0.0.0/0** to the authorized networks, thereby allowing all incoming traffic from everywhere. (Feel free to restrict this to your outgoing IP address)

For security reasons, it is also recommended to configure your database to only accept SSL connections. You'll find the relevant setting in the **Security** tab. Select **SSL Connections only** in order to encrypt all traffic with your database.

Now **Create Client Certificate** and download all three files. Save the paths to these three files as follows.

```
SSL_CA=<SSL_CA> # for example /home/zen/Downloads/server-ca.pem
SSL_CERT=<SSL_CERT> # for example /home/zen/Downloads/client-cert.pem
SSL_KEY=<SSL_KEY> # for example /home/zen/Downloads/client-key.pem
```

Note the @ sign in front of these three variables. The @ sign tells the secret manager that these are file paths to be loaded from.

Finally, head on over to the `Databases` submenu and create your database and save its name.

```
DB_NAME=<DB_NAME> # for example zenml_db
```

gcloud CLI

We have set some sensible defaults here, feel free to replace these with names of your own.

```
DB_INSTANCE=zenml-inst
DB_NAME=zenml_metadata_store_db # make sure this contains no '-'
CERT_NAME=zenml-cert
CLIENT_KEY_PATH=$PROJECT_NAME"client-key.pem"
CLIENT_CERT_PATH=$PROJECT_NAME"client-cert.pem"
SERVER_CERT_PATH=$PROJECT_NAME"server-ca.pem"
```

```
# Enable the sql api for database creation
gcloud services enable sqladmin.googleapis.com
# Create the db instance
gcloud sql instances create $DB_INSTANCE --tier=db-f1-micro \
    --region=$GCP_LOCATION --authorized-networks 0.0.0.0/0
```

Make sure the instance is fully set up before continuing.

```
DB_HOST=$(gcloud sql instances describe $DB_INSTANCE --format='get(ipAddresses[0
gcloud sql users set-password root --host=% --instance $DB_INSTANCE --password $

# Create Client certificate and download all three
gcloud sql instances patch $DB_INSTANCE --require-ssl
```

This might take some time to finish again.

```
gcloud sql ssl client-certs create $CERT_NAME $CLIENT_KEY_PATH \
    --instance $DB_INSTANCE
gcloud sql ssl client-certs describe $CERT_NAME --instance=$DB_INSTANCE \
    --format="value(cert)" > $CLIENT_CERT_PATH
gcloud sql instances describe $DB_INSTANCE \
    --format="value(serverCaCert.cert)" > $SERVER_CERT_PATH

gcloud sql databases create $DB_NAME --instance=$DB_INSTANCE \
    --collation=utf8_general_ci --charset=utf8
```

Create a Service Account

All the resources are created. Now we need to make sure the instance performing the compute engine (Vertex AI) needs to have the relevant permissions to access the other resources. For this you'll need to go

GCP UI

here to create a service account. Give it a relevant name and allow access to the following roles:

- Vertex AI Custom Code Service Agent
- Vertex AI Service Agent
- Container Registry Service Agent
- Secret Manager Admin

Also give your user access to the service account. This is the service account that will be used by the Vertex AI compute engine.

```
SERVICE_ACCOUNT=<SERVICE_ACCOUNT> # for example zenml-vertex-sa@zenml-project.ia
```

gcloud CLI

```
SERVICE_ACCOUNT_ID=zenml-vertex-sa
USER_EMAIL=<USER_EMAIL> # for example user@zenml.io
SERVICE_ACCOUNT=${SERVICE_ACCOUNT_ID}"@"${PROJECT_NAME}".iam.gserviceaccount.com
```

```
gcloud iam service-accounts create $SERVICE_ACCOUNT_ID \
    --display-name="zenml-vertex-sa" \
    --description="Service account for running Vertex Ai workflows from ZenML."
gcloud projects add-iam-policy-binding ${PROJECT_NAME} \
    --role="roles/aiplatform.customCodeServiceAgent" \
    --member="serviceAccount:"${SERVICE_ACCOUNT}
gcloud projects add-iam-policy-binding ${PROJECT_NAME} \
    --role="roles/aiplatform.serviceAgent" \
    --member="serviceAccount:"${SERVICE_ACCOUNT}
gcloud projects add-iam-policy-binding ${PROJECT_NAME} \
    --role="roles/containerregistry.ServiceAgent" \
    --member="serviceAccount:"${SERVICE_ACCOUNT}
gcloud projects add-iam-policy-binding ${PROJECT_NAME} \
    --role="roles/secretmanager.admin" \
    --member="serviceAccount:"${SERVICE_ACCOUNT}
gcloud iam service-accounts add-iam-policy-binding $SERVICE_ACCOUNT \
    --member="user:"${USER_EMAIL} --role="roles/iam.serviceAccountUser"
```

ZenML Stack

Everything on the GCP side is set up, you're ready to set up the ZenML stack components now.

Copy-paste this into your terminal and press enter.

```
zenml orchestrator register vertex_orchestrator --flavor=vertex \
      --project=$PROJECT_NUMBER --location=$GCP_LOCATION \
      --workload_service_account=$SERVICE_ACCOUNT
zenml secrets-manager register gcp_secrets_manager \
      --flavor=gcp_secrets_manager --project_id=$PROJECT_NUMBER
zenml container-registry register gcp_registry --flavor=gcp \
      --uri=$CONTAINER_REGISTRY_URI
zenml artifact-store register gcp_artifact_store --flavor=gcp \
      --path=$GSUTIL_URI
zenml metadata-store register gcp_metadata_store --flavor=mysql \
      --host=$DB_HOST --port=3306 --database=$DB_NAME \
      --secret=mysql_secret
zenml stack register gcp_vertex_stack -m gcp_metadata_store \
      -a gcp_artifact_store -o vertex_orchestrator -c gcp_registry \
      -x gcp_secrets_manager --set
zenml secret register mysql_secret --schema=mysql \
      --user=root --password=$DB_PASSWORD \
      --ssl_ca="@"$SERVER_CERT_PATH --ssl_cert="@"$CLIENT_CERT_PATH \
      --ssl_key="@"$CLIENT_KEY_PATH
```

This is where your ZenML stack is created and connected to the GCP cloud resources. If you now run `zenml stack describe` you should see this:

```
            Stack Configuration
┏━━━━━━━━━━━━━━━━━━━━┯━━━━━━━━━━━━━━━━━━━━━┓
┃ COMPONENT_TYPE     │ COMPONENT_NAME      ┃
┠────────────────────┼─────────────────────┨
┃ ARTIFACT_STORE     │ gcp_artifact_store  ┃
┠────────────────────┼─────────────────────┨
┃ CONTAINER_REGISTRY │ gcp_registry        ┃
┠────────────────────┼─────────────────────┨
┃ METADATA_STORE     │ gcp_metadata_store  ┃
┠────────────────────┼─────────────────────┨
┃ ORCHESTRATOR       │ vertex_orchestrator ┃
┠────────────────────┼─────────────────────┨
┃ SECRETS_MANAGER    │ gcp_secrets_manager ┃
┗━━━━━━━━━━━━━━━━━━━━┷━━━━━━━━━━━━━━━━━━━━━┛
     'gcp_vertex_stack' stack (ACTIVE)
```

Run your pipeline in the cloud

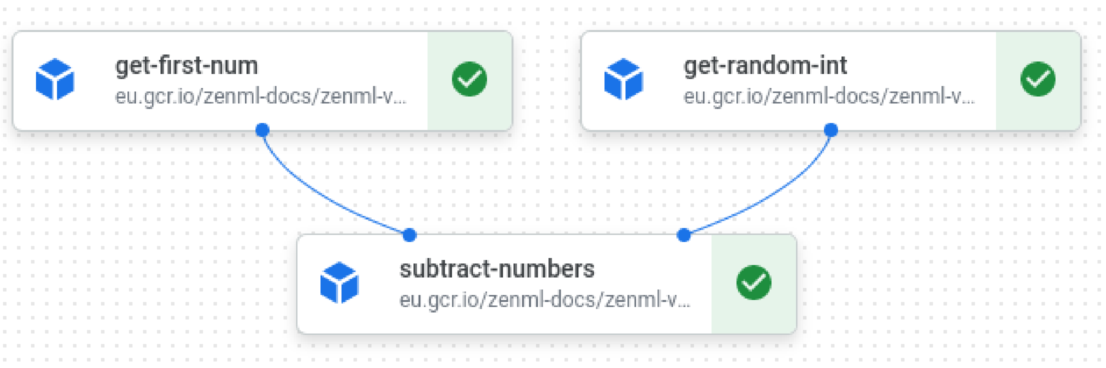
With your ZenML stack set up and active, you are now ready to run your ZenML pipeline on Vertex AI.

For example, you could pull the ZenML Vertex AI example and run it.

```
zenml example pull vertex_ai_orchestration
cd zenml_examples/vertex_ai_orchestration/
python run.py
```

Your first run might fail as one of the service accounts is only created once vertex is run for the first time. This service account will need to be given appropriate rights after the first run fails.

At the end of the logs you should be seeing a link to the Vertex AI dashboard. It should look something like this:


get-first-num
eu.gcr.io/zenml-docs/zenml-v...
get-random-int
eu.gcr.io/zenml-docs/zenml-v...
subtract-numbers
eu.gcr.io/zenml-docs/zenml-v...

Finished Run

In case you get an error message like this:

```
Failed to create pipeline job. Error: Vertex AI Service Agent
'service-...@gcp-sa-aiplatform-cc.iam.gserviceaccount.com' should be
granted access to the image eu.gcr.io/...
```

You will need to follow these instructions:

GCP UI

On the IAM page you will need to give permissions to the service account of the custom code workers.

For this, head over to your IAM configurations, click on **Include Google-provided role grants** on the top right and find the **<project_number>@gcp-sa-aiplatform-cc.iam.gserviceaccount.com** service account.

Now give this one the **Container Registry Service Agent** role on top of its existing role.

gcloud CLI

```
gcloud projects add-iam-policy-binding ${PROJECT_NAME} --role="roles/aiplatform.
    --member="serviceAccount:service-"${PROJECT_NUMBER}"@gcp-sa-aiplatform-cc.ia
gcloud projects add-iam-policy-binding ${PROJECT_NAME} --role="roles/containerre
    --member="serviceAccount:service-"${PROJECT_NUMBER}"@gcp-sa-aiplatform-cc.ia
```

Now rerun your pipeline, it should work now.

Conclusion

Within this guide you have set up and used a stack on GCP using the Vertex AI orchestrator. For more guides on different cloud set-ups, check out the Kubeflow and Kubernetes orchestrators respectively and find out if these are a better fit for you.

One Shot Setup

Quick setup commands

Set these parameters:

```
USER_EMAIL=<USER_EMAIL>  # for example user@zenml.io
PARENT_ORG_ID=<PARENT_ORG_ID> # for example 294710374920
BILLING_ACC=<BILLING_ACC> # for example 20CIW7-183916-8GA18Z
PROJECT_NAME=<PROJECT_NAME>
CONTAINER_REGISTRY_REGION=eu # can be 'eu', 'us', 'asia'
GCP_LOCATION=<GCP_LOCATION> # for example europe-west3
DB_PASSWORD=<DB_PASSWORD> # for example auk(/194
```

And run this (make sure all the commands worked):

```
# Create a project and attach it to the specified billing account
gcloud projects create $PROJECT_NAME --organization=$PARENT_ORG_ID
gcloud config set project $PROJECT_NAME
gcloud beta billing projects link $PROJECT_NAME --billing-account $BILLING_ACC
PROJECT_NUMBER=$(gcloud projects describe $PROJECT_NAME --format="value(projectNum

# Enable the three APIs for vertex ai, the container registry and the
```

```
gcloud services enable aiplatform.googleapis.com
gcloud services enable secretmanager.googleapis.com
gcloud services enable containerregistry.googleapis.com

CONTAINER_REGISTRY_URI=$CONTAINER_REGISTRY_REGION".gcr.io/"$PROJECT_NAME

# Create a storage bucket
GSUTIL_URI=gs://${PROJECT_NAME}-bucket
gsutil mb -p $PROJECT_NAME $GSUTIL_URI

# Enable the sql api for database creation
gcloud services enable sqladmin.googleapis.com

# Create the db instance
DB_INSTANCE=zenml-inst
gcloud sql instances create $DB_INSTANCE --tier=db-f1-micro --region=$GCP_LOCATION

DB_HOST=$(gcloud sql instances describe $DB_INSTANCE --format='get(ipAddresses[0].
gcloud sql users set-password root --host=% --instance $DB_INSTANCE --password $DB

# Create Client certificate and download all three
CERT_NAME=zenml-cert
CLIENT_KEY_PATH=$PROJECT_NAME"client-key.pem"
CLIENT_CERT_PATH=$PROJECT_NAME"client-cert.pem"
SERVER_CERT_PATH=$PROJECT_NAME"server-ca.pem"
gcloud sql instances patch $DB_INSTANCE --require-ssl
gcloud sql ssl client-certs create $CERT_NAME $CLIENT_KEY_PATH --instance $DB_INST
gcloud sql ssl client-certs describe $CERT_NAME --instance=$DB_INSTANCE --format="
gcloud sql instances describe $DB_INSTANCE --format="value(serverCaCert.cert)" > $

DB_NAME=zenml_metadata_store_db # make sure this contains no '-' as this will fail
gcloud sql databases create $DB_NAME --instance=$DB_INSTANCE --collation=utf8_gene

# Configure the service accounts
SERVICE_ACCOUNT_ID=zenml-vertex-sa
SERVICE_ACCOUNT=${SERVICE_ACCOUNT_ID}"@"${PROJECT_NAME}".iam.gserviceaccount.com"

gcloud iam service-accounts create $SERVICE_ACCOUNT_ID --display-name="zenml-verte
    --description="Service account for running Vertex Ai workflows from ZenML."
gcloud projects add-iam-policy-binding ${PROJECT_NAME} --role="roles/aiplatform.cu
    --member="serviceAccount:"${SERVICE_ACCOUNT}
gcloud projects add-iam-policy-binding ${PROJECT_NAME} --role="roles/aiplatform.se
    --member="serviceAccount:"${SERVICE_ACCOUNT}
gcloud projects add-iam-policy-binding ${PROJECT_NAME} --role="roles/containerregi
    --member="serviceAccount:"${SERVICE_ACCOUNT}
gcloud projects add-iam-policy-binding ${PROJECT_NAME} --role="roles/secretmanager
    --member="serviceAccount:"${SERVICE_ACCOUNT}
gcloud iam service-accounts add-iam-policy-binding $SERVICE_ACCOUNT \
    --member="user:"${USER_EMAIL} --role="roles/iam.serviceAccountUser"

ORCHESTRATOR_NAME=$PROJECT_NAME"-gcp_vo"
ARTIFACT_STORE_NAME=$PROJECT_NAME"-gcp_as"
METADATA_STORE_NAME=$PROJECT_NAME"-gcp_ms"
```

```
CONTAINER_REGISTRY_NAME=$PROJECT_NAME"-gcp_cr"
SECRET_MANAGER_NAME=$PROJECT_NAME"-gcp_sm"
STACK_NAME=$PROJECT_NAME"-gcp_stack"

zenml orchestrator delete $ORCHESTRATOR_NAME
zenml artifact-store delete $ARTIFACT_STORE_NAME
zenml metadata-store delete $METADATA_STORE_NAME
zenml container-registry delete $CONTAINER_REGISTRY_NAME
zenml secrets-manager delete $SECRET_MANAGER_NAME

zenml orchestrator register $ORCHESTRATOR_NAME --flavor=vertex \
      --project=$PROJECT_NUMBER --location=$GCP_LOCATION \
      --workload_service_account=$SERVICE_ACCOUNT
zenml container-registry register $CONTAINER_REGISTRY_NAME --flavor=gcp \
      --uri=$CONTAINER_REGISTRY_URI
zenml secrets-manager register $SECRET_MANAGER_NAME \
      --flavor=gcp_secrets_manager --project_id=$PROJECT_NUMBER
zenml artifact-store register $ARTIFACT_STORE_NAME --flavor=gcp \
      --path=$GSUTIL_URI
zenml metadata-store register $METADATA_STORE_NAME --flavor=mysql \
      --host=$DB_HOST --port=3306 --database=$DB_NAME \
      --secret=mysql_secret

zenml stack register $STACK_NAME -o $ORCHESTRATOR_NAME \
      -c $CONTAINER_REGISTRY_NAME -x $SECRET_MANAGER_NAME \
      -a $ARTIFACT_STORE_NAME -m $METADATA_STORE_NAME --set

zenml secret register mysql_secret --schema=mysql \
      --user=root --password=$DB_PASSWORD \
      --ssl_ca="@"$SERVER_CERT_PATH --ssl_cert="@"$CLIENT_CERT_PATH --ssl_key="@"$
```

If the first pipeline run fails:

```
gcloud projects add-iam-policy-binding ${PROJECT_NAME} --role="roles/aiplatform.cu
    --member="serviceAccount:service-"${PROJECT_NUMBER}"@gcp-sa-aiplatform-cc.iam.
gcloud projects add-iam-policy-binding ${PROJECT_NAME} --role="roles/containerregi
    --member="serviceAccount:service-"${PROJECT_NUMBER}"@gcp-sa-aiplatform-cc.iam.
```

Azure

How to set up stacks on Microsoft Azure

Azure is one of the most popular cloud providers and offers a range of services that can be used while building your MLOps stacks. You can learn more about machine learning at Azure on their website.

Available Stack Components

This is a list of all supported Azure services that you can use as ZenML stack components.

Azure Kubernetes Service (AKS)

Azure Kubernetes Service (AKS) is a managed Kubernetes service with hardened security and fast delivery.It allows you to quickly deploy a production ready Kubernetes cluster in Azure. Learn more here.

- An AKS cluster can be used to run multiple **orchestrators**.
  - A Kubernetes-native orchestrator.
  - A Kubeflow orchestrator.
- You can host **model deployers** on the cluster.
  - A Seldon model deployer.
  - An MLflow model deployer.
- Experiment trackers can also be hosted on the cluster.
  - An MLflow experiment tracker

Azure Blob Storage

Azure Blob storage is Microsoft's object storage solution for the cloud. Blob storage is optimized for storing massive amounts of unstructured data. Blob storage offers three types of resources: the storage account, a container in the storage account and a blob in a container. Learn more here.

- You can use an Azure Blob Storage Container as an artifact store to hold files from our pipeline runs like models, data and more.

Azure Container Registry

Azure Container Registry is a managed registry service based on the open-source Docker Registry 2.0. Create and maintain Azure container registries to store and manage your container images and related artifacts. Learn more here.

- An Azure container registry can be used as a ZenML container registry stack component to host images of your pipelines.

AzureML

Azure Machine Learning is a cloud service for accelerating and managing the machine learning project lifecycle. Machine learning professionals, data scientists, and engineers can use it in their day-to-day workflows to train and deploy models, and manage MLOps. Learn more here.

- You can use AzureML compute as a step operator to run specific steps from your pipeline using it as the backend.

Azure SQL server

The Azure SQL metadata service is not yet supported because ZenML doesn't yet support a matching Azure secrets manager.

Key Vault

ZenML doesn't support the key vault secret manager by Azure yet. The integration is coming soon and contributions are welcome!

In the following pages, you will find step-by-step guides for setting up some common stacks using the Azure console and the CLI. More combinations and components are progressively updated in the form of new pages.

Set Up a Minimal MLOps Stack on Azure

How to set up a minimal stack on Microsoft Azure

To get started using ZenML on the cloud, you need some basic infrastructure up and running which ZenML can use to run your pipelines. This step-by-step guide explains how to set up a basic cloud stack on Azure.

> This guide represents **one** of many ways to create a cloud stack on Azure. You can customize this by adding additional components of replacing one of the components described in this guide.

What will the stack look like?

Prerequisites

- Docker installed and running.
- kubectl installed.
- The az CLI installed and authenticated.
- ZenML and the integrations for this tutorial stack installed:

```
pip install zenml
zenml integration install azure kubernetes
```

Setting up the Azure resources

All the Azure setup steps can either be done using the Azure UI or CLI. Simply select the tab for your preferred option and let's get started. First open up a terminal which we'll use to store some values along the way which we'll need to configure our ZenML stack later.

Create an account

If you don't have an Azure account yet, go to https://azure.microsoft.com/en-gb/free/ and create one.

Create a resource group

Resource groups are a concept in Azure that allows us to bundle different resources that share a similar lifecycle. We'll create a new resource group for this tutorial so we'll be able to differentiate them from other resources in our account and easily delete them at the end.

> When creating a resource group, you need to provide a location for that resource group. You may be wondering, "Why does a resource group need a location? And, if the resources can have different locations than the resource group, why does the resource group location matter at all?" The resource group stores metadata about the resources. Therefore, when you specify a location for the resource group, you are specifying where that metadata is stored. For compliance reasons, you may need to ensure that your data is stored in a particular region. https://docs.microsoft.com/en-us/azure/azure-resource-manager/resource-group-overview

Azure UI

- Go to the Azure portal, click the hamburger button in the top left to open up the portal menu. Then, hover over the `Resource groups` section until a popup appears and click on the `+ Create` button.
- Select a region and enter a name for your resource group before clicking on `Review + create`.
- Verify that all the information is correct and click on `Create`.
- Set the following variables in your shell.

```
RESOURCE_GROUP=<RESOURCE_GROUP_NAME>
RG_LOCATION=<RG_LOCATION>
```

Azure CLI

```
RESOURCE_GROUP=<RESOURCE_GROUP_NAME>
RG_LOCATION=<RG_LOCATION>
az group create --name $RESOURCE_GROUP --location $RG_LOCATION
```

Create a storage account

An Azure storage account is a grouping of Azure data storage objects which also provides a namespace and authentication options to access them. We'll need a storage account to hold the blob storage container we'll create in the next step.

Azure UI

- Open up the portal menu again, but this time hover over the `Storage accounts` section and click on the `+ Create` button in the popup once it appears:
- Select your previously created **resource group**, a **region** and a **globally unique name** and then click on `Review + create`:
- Make sure that all the values are correct and click on `Create`:
- Wait until the deployment is finished and click on `Go to resource` to open up your newly created storage account:
- In the left menu, select `Access keys`:
- Click on `Show keys`, and once the keys are visible, note down the **storage account name** and the value of the **Key** field of either key1 or key2. We're going to use them for the `<STORAGE_ACCOUNT_NAME>` and `<STORAGE_ACCOUNT_KEY>` placeholders later.
- Set the following variables in your shell.

```
REGION=<REGION>
STORAGE_ACCOUNT_NAME=<STORAGE_ACCOUNT_NAME>
STORAGE_ACCOUNT_KEY=<STORAGE_ACCOUNT_KEY>
```

Azure CLI

```
# Set a name for your bucket and the Azure region for your resources
REGION=<REGION>
STORAGE_ACCOUNT_NAME=<STORAGE_ACCOUNT_NAME>

az storage account create -n $STORAGE_ACCOUNT_NAME -g $RESOURCE_GROUP -l $REGION
STORAGE_ACCOUNT_KEY=$(az storage account keys list -g $RESOURCE_GROUP -n $STORAG
```

Create an Azure Blob Storage Container

Next, we're going to create an Azure Blob Storage Container. It will be used by ZenML to store the output artifacts of all our pipeline steps.

- To do so, select `Containers` in the Data storage section of the storage account:
- Then click the `+ Container` button on the top to create a new container:
- Choose a name for the container and note it down. We're going to use it later for the `<BLOB_STORAGE_CONTAINER_NAME>` placeholder. Then create the container by clicking the `Create` button.
- Set the following variables in your shell.

```
BLOB_STORAGE_CONTAINER_NAME=<BLOB_STORAGE_CONTAINER_NAME>
```

Azure CLI

```
BLOB_STORAGE_CONTAINER_NAME=<BLOB_STORAGE_CONTAINER_NAME>
az storage container create --$BLOB_STORAGE_CONTAINER_NAME
```

Set up a MySQL database

Now let's set up a managed MySQL database. This will act as ZenML's metadata store and store metadata regarding our pipeline runs which will enable features like caching and establish a consistent lineage between our pipeline steps.

Azure UI

- Open up the portal menu and click on `+ Create a resource`.
- Search for `Azure Database for MySQL` and once found click on `Create`.Make sure you select `Flexible server` and then continue by clicking the `Create` button.
- Select a **resource group** and **region** and fill in values for the **server name** as well as **admin username** and **password**. Note down the username and password you chose as we're going to need them later for the `<MYSQL_USERNAME>` and `<MYSQL_PASSWORD>` placeholders. Then click on `Next: Networking`.
- Now click on `Add 0.0.0.0 - 255.255.255.255` to allow access from all public IPs. This is necessary so the machines running our GitHub Actions can access this database. It will still require username, password as well as a SSL certificate to authenticate.
- In the opened up popup, click on `Continue` and click on `Review + create`.
- Verify the configuration and click the `Create` button. Now we'll have to wait until the deployment is finished (this might take ~15 minutes).

Note: If the deployment fails for some reason, delete the resource and restart from the beginning of this section.

-

- Once the deployment is finished, click on `Go to resource`.
- On the overview page of your MySQL server resource, note down the server name in the top right. We'll use it later for the `<MYSQL_SERVER_NAME>` placeholder.
- Then click on `Networking` in the left menu. Click on `Download SSL Certificate` on the top. Make sure to note down the path to the certificate file which we'll use for the `<PATH_TO_SSL_CERTIFICATE>` placeholder in a later step.
- Set the following values in your shell.

```
MYSQL_SERVER_NAME=<MYSQL_SERVER_NAME>
PATH_TO_SSL_CERTIFICATE=<PATH_TO_SSL_CERTIFICATE>
MYSQL_USERNAME=<MYSQL_USERNAME>
MYSQL_PASSWORD=<MYSQL_PASSWORD>
```

Azure CLI

```
MYSQL_SERVER_INSTANCE_NAME=<MYSQL_SERVER_INSTANCE_NAME>
MYSQL_USERNAME=<MYSQL_USERNAME>
MYSQL_PASSWORD=<MYSQL_PASSWORD>

az mysql flexible-server create -l $REGION -g $RESOURCE_GROUP \
 -n $MYSQL_SERVER_INSTANCE_NAME -u $MYSQL_USERNAME -p $MYSQL_PASSWORD \
 --public-access 0.0.0.0-255.255.255.255

MYSQL_SERVER_NAME="$MYSQL_SERVER_INSTANCE_NAME.mysql.database.azure.com"

PATH_TO_SSL_CERTIFICATE=<PATH_TO_SSL_CERTIFICATE>  # should end with a filename
wget --no-check-certificate https://dl.cacerts.digicert.com/DigiCertGlobalRootCA
```

Container Registry

Azure UI

- Set up an Azure Container Registry (ACR).
- Set the regsitry name in your shell.

```
REGISTRY_NAME=<REGISTRY_NAME>  # should be <some-name>.azurecr.io
```

Azure CLI

```
REGISTRY_NAME=<REGISTRY_NAME>  # should be <some-name>.azurecr.io
```

```
az acr create -n $REGISTRY_NAME -g $RESOURCE_GROUP
```

Orchestrator (Azure Kubernetes Service)

Azure UI

- On the Azure portal menu or from the Home page, select Create a resource. Select `Containers` > `Kubernetes Service`.
- On the Basics page, configure the following options. Under Project details,
  - Select your Azure Subscription.
  - Select your Resource group.
- Under Cluster details,
  - Ensure the the Preset configuration is Standard ($$). For more details on preset configurations, see Cluster configuration presets in the Azure portal.
  - Enter a Kubernetes cluster name. We will use it later as <AKS_CLUSTER_NAME>
- Enter your `Region` for the AKS cluster, and leave the default value selected for Kubernetes version.
- Keep the default values for all other sections and click on `Review + Create`.
- Set the following variable in your shell.

  ```
  AKS_CLUSTER_NAME=<AKS_CLUSTER_NAME>
  ```

Azure CLI

```
AKS_CLUSTER_NAME=<AKS_CLUSTER_NAME>

az aks create -g $RESOURCE_GROUP -n $AKS_CLUSTER_NAME --node-count 1 \
 --enable-cluster-autoscaler
```

Register the ZenML stack

- Register the artifact store:

```
zenml artifact-store register azure_store \
    --flavor=azure \
```

```
    --path=az://$BLOB_STORAGE_CONTAINER_NAME
```

- Register the container registry and authenticate your local docker client

```
zenml container-registry register acr_registry \
    --flavor=azure \
    --uri=$REGISTRY_NAME
az acr login --name $REGISTRY_NAME
```

- Register the metadata store:

```
zenml metadata-store register azure_mysql \
    --flavor=mysql \
    --database=zenml \
    --secret=azure_authentication \
    --host=$MYSQL_SERVER_NAME
```

- Register the secrets manager:

```
zenml secrets-manager register azure_secrets_manager \
    --flavor=azure \
    --region_name=$REGION
```

- Configure your `kubectl` client and register the orchestrator:

```
az aks get-credentials --resource-group $RESOURCE_GROUP --name $AKS_CLUSTER_NAME
kubectl create namespace zenml
zenml orchestrator register aks_kubernetes_orchestrator \
    --flavor=kubernetes \
    --kubernetes_context=$(kubectl config current-context)
```

- Register the ZenML stack and activate it:

```
zenml stack register kubernetes_stack \
    -o aks_kubernetes_orchestrator \
    -a azure_store \
    -m azure_mysql \
    -c acr_registry \
    -x azure_secrets_manager \
    --set
```

- Register the secret for authenticating with your MySQL database:

```
zenml secret register azure_authentication \
    --schema=mysql \
    --user=$MYSQL_USERNAME \
    --password=$MYSQL_PASSWORD
```

After all of this setup, you're now ready to run any ZenML pipeline on Azure!

Quick setup

If you're looking for a way to get started quickly, we've combined all the commands so you can copy-paste them and execute them in a single go. You'll only need to set values for the `<REGION>`, `<RESOURCE_GROUP>` and `<MYSQL_PASSWORD>` right at the beginning before executing the rest. If

Quick setup commands

```
# Select one of the region codes for <REGION>: https://docs.microsoft.com/en-us/az
REGION=<REGION>
RESOURCE_GROUP=<RESOURCE_GROUP>
# Choose a secure password for your database admin account. Make sure it includes:
# - at least 8 printable ASCII characters
# - no slash, single or double quotes or @ signs
MYSQL_PASSWORD=<MYSQL_PASSWORD>

# Other parameters (we've set some defaults for these but feel free to change them
RG_LOCATION=westus
STORAGE_ACCOUNT_NAME=zenml-storage-account
BLOB_STORAGE_CONTAINER_NAME=zenml-artifact-store
MYSQL_SERVER_INSTANCE_NAME=zenml-metadata-store
MYSQL_USERNAME=admin
REGISTRY_NAME=zenml
PATH_TO_SSL_CERTIFICATE="./certificate.pem"  # WINDOWS USERS, use the backslash
AKS_CLUSTER_NAME=zenml-aks-cluster

az group create --name $RESOURCE_GROUP --location $RG_LOCATION

# storage account
az storage account create -n $STORAGE_ACCOUNT_NAME -g $RESOURCE_GROUP -l $REGION
STORAGE_ACCOUNT_KEY=$(az storage account keys list -g $RESOURCE_GROUP -n $STORAGE_

az storage container create --$BLOB_STORAGE_CONTAINER_NAME

# MySQL database
az mysql flexible-server create -l $REGION -g $RESOURCE_GROUP \
 -n $MYSQL_SERVER_INSTANCE_NAME -u $MYSQL_USERNAME -p $MYSQL_PASSWORD \
 --public-access 0.0.0.0-255.255.255.255

MYSQL_SERVER_NAME="$MYSQL_SERVER_INSTANCE_NAME.mysql.database.azure.com"

wget --no-check-certificate https://dl.cacerts.digicert.com/DigiCertGlobalRootCA.c

# container registry
az acr create -n $REGISTRY_NAME -g $RESOURCE_GROUP

# orchestrator
az aks create -g $RESOURCE_GROUP -n $AKS_CLUSTER_NAME --node-count 1 \
 --enable-cluster-autoscaler

# register with zenml

zenml artifact-store register azure_store \
    --flavor=azure \
    --path=az://$BLOB_STORAGE_CONTAINER_NAME

zenml container-registry register acr_registry \
    --flavor=azure \
    --uri=$REGISTRY_NAME
az acr login --name $REGISTRY_NAME

zenml metadata-store register azure_mysql \
    --flavor=mysql \
    --database=zenml \
    --secret=azure_authentication \
    --host=$MYSQL_SERVER_NAME

zenml secrets-manager register azure_secrets_manager \
    --flavor=azure \
    --region_name=$REGION

az aks get-credentials --resource-group $RESOURCE_GROUP --name $AKS_CLUSTER_NAME
kubectl create namespace zenml
zenml orchestrator register aks_kubernetes_orchestrator \
    --flavor=kubernetes \
    --kubernetes_context=$(kubectl config current-context)

zenml stack register kubernetes_stack \
    -o aks_kubernetes_orchestrator \
    -a azure_store \
    -m azure_mysql \
    -c acr_registry \
    -x azure_secrets_manager \
    --set

zenml secret register azure_authentication \
    --schema=mysql \
    --user=$MYSQL_USERNAME \
    --password=$MYSQL_PASSWORD
```

Collaborate

Collaborate with ZenML

How to collaborate with your team in ZenML

Using ZenML to develop and execute pipelines from the comfortable confines of your workstation or laptop is a great way to incorporate MLOps best practices and production grade quality into your project from day one. However, as machine learning projects grow, managing and running ZenML in a single-user and single-machine setting can become a strenuous task. More demanding projects involve creating and managing an increasing number of pipelines and complex Stacks built on a wider and continuously evolving set of technologies. This can easily exceed the capabilities of a single machine and person or role.

The same principles observed in the ZenML single-user experience are applicable as a framework of collaboration between the various specialized roles in the AI/ML team. We purposefully maintain a clean separation between the core ZenML concepts, with the intention that they may be managed as individual responsibilities assigned to different members of a team or department without incurring the overhead and friction usually associated with larger organizations.

One such example is the decoupling of ZenML Stacks from pipelines and their associated code, a principle that provides for a smooth transition from experimenting with and running pipelines locally to deploying them to production environments. It is this same decoupling that also allows ZenML to be used in a setting where part of a team is iterating on writing ML code and implementing pipelines, while the other part is defining and actively maintaining the infrastructure and the Stacks that will be used to execute pipelines in production. Everyone can remain focused on their responsibilities with ZenML acting as the central piece that connects and coordinates everything together.

The ability to configure modular Stacks in which every component has a specialized function is another way in which ZenML maintains a clean separation of concerns. This allows for a MLOps Stack design that is natively flexible and extensible that can evolve organically to match the size and structure of your AI/ML team and organization as you iterate on your project and invest more resources into its development.

This documentation section is dedicated to describing several ways in which you can deploy ZenML as a collaboration framework and enable your entire AI/ML team to enjoy its advantages.

Export and Import ZenML Stacks

If you need to quickly share your Stack configuration with someone else, there is nothing easier than using the ZenML CLI to export a Stack in the form of a YAML file and import it somewhere else.

Organize and Share with Profiles

With ZenML Profiles, you can unlock a range of strategies for organizing and managing ZenML configurations that are available across your entire team. Stacks, Stack Components and other classes of ZenML objects can be stored in a central location and shared across multiple users, teams and automated systems such as CI/CD processes.

Centralized ZenML Management with ZenServer

With the *ZenServer*, you can deploy ZenML as a centralized service and connect entire teams and organizations to an easy to manage collaboration platform that provides a unified view on the MLOps processes, tools and technologies that support your entire AI/ML project lifecycle.

Export/Import Stacks

How to export and import stacks to YAML files

If you wish to transfer one of your stacks to another profile or even another machine, you can do so by exporting the stack configuration and then importing it again.

To export a stack to YAML, run the following command:

```
zenml stack export STACK_NAME FILENAME.yaml
```

This will create a FILENAME.yaml containing the config of your stack and all of its components, which you can then import again like this:

```
zenml stack import STACK_NAME FILENAME.yaml
```

Known Limitations

The exported Stack is only a configuration. It may have local dependencies that are not exported and thus will not be available when importing the Stack on another machine:

- the secrets stored in the local Secrets Managers
- any references to local files and local services not accessible from outside the machine where the Stack is exported, such as the local Artifact Store and Metadata Store

Share Stacks and Profiles via ZenStores

How to manage stacks in a centralized location with ZenStores

The ZenML **Store** (or **ZenStore**) is a low-level concept used to represent the particular driver used to store and retrieve the data that is managed through a ZenML Profile. The ZenStore concept is not directly represented in the CLI commands, but it is reflected in the Profile configuration and can be manipulated by passing advanced parameters to the `zenml profile create` CLI command. The particular ZenStore driver type and configuration used for a Profile can be viewed by bringing up the detailed Profile description (note the store type and URL):

```
$ zenml profile describe
Running without an active repository root.
Running with active profile: 'default' (global)
          'default' Profile Configuration (ACTIVE)
┏━━━━━━━━━━━━━━━┯━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┓
┃ PROPERTY      │ VALUE                                                     ┃
┃ NAME         │ default                                  ┃
┠──────────────┼──────────────────────────────────────────┨
┃ STORE_URL    │ file:///home/zenml/.config/profiles/default ┃
┠──────────────┼──────────────────────────────────────────┨
┃ STORE_TYPE   │ local                                    ┃
┠──────────────┼──────────────────────────────────────────┨
┃ ACTIVE_STACK │ default                                  ┃
┠──────────────┼──────────────────────────────────────────┨
┃ ACTIVE_USER  │ default                                  ┃
┗━━━━━━━━━━━━━━┷━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┛
```

Different ZenStore types can be used to implement different deployment use-cases with regards to where the ZenML managed data is stored and how and where it can be accessed:

- the `local` ZenStore stores Profile data in YAML files on your local filesystem. This is the default ZenStore type and is suitable for local development and testing. Local Profiles can also be shared between multiple users even hosts by using some form of source version control or shared filesystem.
- the `sql` ZenStore driver is based on SQLAlchemy and can interface with any SQL database service, local or remote, to store the Profile data in a SQL database. The SQL driver is an easy way to extend ZenML to accommodate multiple users working from multiple hosts.
- the `rest` ZenStore is a special type of store that connects via a REST API to a remote ZenServer instance. This use-case is applicable to larger teams and organizations that need to deploy ZenML as a dedicated service providing centralized management of Stacks, Pipelines and other ZenML concepts. Please consult the ZenServer documentation dedicated to this deployment model.

Local ZenML Store

By default, newly created Profiles use the `local` ZenStore driver that stores the Profile data on the local filesystem, in the global configuration directory, as a collection of YAML files.

The YAML representation makes it suitable to commit Stack configurations and all other information stored in the Profile into a version control system such as Git, where they can be versioned and shared with other users.

To use a custom location for a Profile, point the ZenStore URL to a directory on your local filesystem. For example:

```
$ zenml profile create git_store --url /tmp/zenml/.zenprofile
Running with active profile: 'default' (local)
Initializing profile git...
Registering default stack...
Registered stack component with type 'orchestrator' and name 'default'.
Registered stack component with type 'metadata_store' and name 'default'.
Registered stack component with type 'artifact_store' and name 'default'.
Registered stack with name 'default'.
Profile 'git_store' successfully created.

$ zenml profile set git_store
```

```
Running with active profile: 'default' (local)
Active profile changed to: 'git_store'

$ zenml profile describe
Running with active profile: 'git_store' (local)
  'git_store' Profile Configuration (ACTIVE)
┏━━━━━━━━━━━━━━┯━━━━━━━━━━━━━━━━━━━━━━━━┓
┃ PROPERTY     │ VALUE                  ┃
┠──────────────┼────────────────────────┨
┃ NAME         │ git_store              ┃
┠──────────────┼────────────────────────┨
┃ STORE_URL    │ /tmp/zenml/.zenprofile ┃
┠──────────────┼────────────────────────┨
┃ STORE_TYPE   │ local                  ┃
┠──────────────┼────────────────────────┨
┃ ACTIVE_STACK │ default                ┃
┠──────────────┼────────────────────────┨
┃ ACTIVE_USER  │ default                ┃
┗━━━━━━━━━━━━━━┷━━━━━━━━━━━━━━━━━━━━━━━━┛
```

Assuming the `/tmp/zenml` location used above is part of a local git clone that is regularly synchronized with a remote server, replicating the same Profile on another machine is straightforward: if the URL points to a location where a Profile already exists, the Profile information is loaded from the existing YAML files:

```
user@another_machine:/tmp$ git clone <git-repo-location> zenml
user@another_machine:/tmp$ cd zenml
user@another_machine:/tmp/zenml$

user@another_machine:/tmp$ zenml profile create git-clone --url ./.zenprofile
Running with active profile: 'git' (local)
Initializing profile git-clone...
Profile 'git-clone' successfully created.
```

As alternatives to version control, a Profile could be shared by using a distributed filesystem (e.g. NFS) or by regularly syncing the folder with a remote central repository using some other means. However, a better solution of sharing Profiles across multiple machines is be to use the SQL ZenStore driver to store the Profile data in a SQL database, or to manage ZenML data through a centralized ZenServer instance.

SQL ZenML Store

The SQL ZenStore type uses SQLAlchemy to store Profile data in a local SQLite database file, on a remote MySQL server or any SQL compatible database system for that matter. The URL value passed during the Profile creation controls the type, location and other parameters for the SQL database connection. To explore the full range of configuration options, consult the SQLAlchemy documentation.

Local SQLite Profile

The simplest form of SQL-based Profile uses a SQLite file located in the global configuration directory:

```
$ zenml profile create sqlite_profile -t sql
Running without an active repository root.
Running with active profile: 'default' (global)
Initializing profile sqlite_profile...
Registering default stack...
Registered stack with name 'default'.
Profile 'sqlite_profile' successfully created.

$ zenml profile set sqlite_profile
Running without an active repository root.
Running with active profile: 'zenml' (global)
Active profile changed to: 'sqlite_profile'

$ zenml profile describe
Running without an active repository root.
Running with active profile: 'sqlite_profile' (global)
                    'sqlite_profile' Profile Configuration (ACTIVE)
┏━━━━━━━━━━━━━━┯━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┓
┃ PROPERTY     │ VALUE                                                                  ┃
┠──────────────┼────────────────────────────────────────────────────────────────────────┨
┃ NAME         │ sqlite_profile                                                         ┃
┠──────────────┼────────────────────────────────────────────────────────────────────────┨
┃ STORE_URL    │ sqlite:////home/stefan/.config/zenml/profiles/sqlite_profile/zenml.db ┃
┠──────────────┼────────────────────────────────────────────────────────────────────────┨
┃ STORE_TYPE   │ sql                                                                    ┃
┠──────────────┼────────────────────────────────────────────────────────────────────────┨
┃ ACTIVE_STACK │ default                                                                ┃
┠──────────────┼────────────────────────────────────────────────────────────────────────┨
┃ ACTIVE_USER  │ default                                                                ┃
┗━━━━━━━━━━━━━━┷━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┛
```

The location of the SQLite database can be customized during profile creation:

```
$ zenml profile create custom_sqlite -t sql --url=sqlite:////tmp/zenml/zenml_profile.db
Running without an active repository root.
Running with active profile: 'sqlite_profile' (global)
Initializing profile custom_sqlite...
Registering default stack...
Registered stack with name 'default'.
Profile 'custom_sqlite' successfully created.

$ zenml profile set custom_sqlite
Running without an active repository root.
Running with active profile: 'sqlite_profile' (global)
Active profile changed to: 'custom_sqlite'

$ zenml profile describe
Running without an active repository root.
Running with active profile: 'custom_sqlite' (global)
     'custom_sqlite' Profile Configuration (ACTIVE)
┏━━━━━━━━━━━━━━┯━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┓
┃ PROPERTY     │ VALUE                                 ┃
```

```
┠──────────────┼──────────────────────────────────────┨
┃ NAME         │ custom_sqlite                        ┃
┠──────────────┼──────────────────────────────────────┨
┃ STORE_URL    │ sqlite:////tmp/zenml/zenml_profile.db ┃
┠──────────────┼──────────────────────────────────────┨
┃ STORE_TYPE   │ sql                                  ┃
┠──────────────┼──────────────────────────────────────┨
┃ ACTIVE_STACK │ default                              ┃
┠──────────────┼──────────────────────────────────────┨
┃ ACTIVE_USER  │ default                              ┃
┗━━━━━━━━━━━━━━┷━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┛
```

MySQL Profile

To connect the ZenML Profile to an existing MySQL database, some additional configuration is required on the MySQL server to create a user and database:

```
mysql -u root
GRANT ALL PRIVILEGES ON *.* TO 'zenml'@'%' IDENTIFIED BY 'password';

mysql -u zenml -p
CREATE DATABASE zenml;
```

Then, on the client machine, some additional packages need to be installed. Check the SQLAlchemy documentation for the various MySQL drivers that are supported and how to install and use them. The following is an example of using the mysqlclient driver for SQLAlchemy on an Ubuntu OS. Depending on your choice of driver and host OS, your experience may vary:

```
sudo apt install libmysqlclient-dev
pip install mysqlclient
```

Finally, the command to create a new Profile would look like this:

```
$ zenml profile create --store-type sql --url "mysql://zenml:password@10.11.12.13/zenml" my
```

Organization-Wide Collaboration with ZenServer

How to collaboratively use ZenML in larger organizations with the ZenServer

Sometimes, you may need to exchange or collaborate on Stack configurations with other developers or even just have your Stacks available on multiple machines. While you can always zip up your Profile files or check the local ZenStore files into version control, a more elegant solution is to have some kind of service accessible from anywhere in your network that provides a REST API that can be used to store and access your Stacks over the network.

The ZenServer, short for ZenML Server, is a distributed client-server ZenML deployment scenario in which multiple ZenML clients can connect to a remote service that provides persistent storage and acts as a central management hub for all ZenML operations involving Stack configurations, Stack Components and other ZenML objects.

Working with a ZenServer involves two main aspects: deploying the ZenServer somewhere and connecting to it from your ZenML client. Keep reading to learn more about how to configure your ZenML client to connect to a remote ZenServer instance, or jump straight to the available ZenServer deployment options by visiting the relevant sections:

- run a ZenServer locally
- deploy ZenServer with Docker
- (more to come ...)

The ZenServer is still undergoing heavy development. Some features are in Alpha state and may change in future releases. We are also working on providing more deployment and lifecycle management options for the ZenServer that will expand the currently supported deployment use-cases with improved scalability, security and robustness.

Interacting with a Remote ZenServer

The ZenML Profile and ZenStore are the main configuration concepts behind the ZenServer architecture. To connect your ZenML client to a remote ZenServer instance, you need the URL of the ZenServer REST API and a user that has been configured on the server.

Connecting to the ZenServer is just a matter of creating a ZenML Profile backed by a ZenStore of type `rest` and pointing it to the ZenServer REST API URL, e.g.:

```
$ zenml profile create zenml-remote -t rest --url http://192.168.178.40:8080 --user defaul
Running without an active repository root.
Running with active profile: 'default' (global)
Initializing profile zenml-remote...
Profile 'zenml-remote' successfully created.

$ zenml profile set zenml-remote
Running without an active repository root.
Running with active profile: 'devel' (global)
Active profile changed to: 'zenml-remote'
```

All ZenML pipelines executed while the ZenServer backed Profile is active will use the Stack and Stack Component definitions stored on the server.

Running the ZenServer Locally

The ZenServer can be deployed locally on any machine, as a means of assessing its functionality without going through the pains of provisioning a complicated infrastructure setup. Before we get started, we need to make sure that all the necessary dependencies for the ZenServer are installed. To do that, we install ZenML with the server extras like this:

```
pip install zenml[server]
```

Starting a local ZenServer instance can be done by typing `zenml server up`. This will start the server as a background daemon process accessible on your local machine, by default on port 8000:

```
$ zenml server up
Running without an active repository root.
Starting a new ZenServer local instance.
ZenServer running at 'http://127.0.0.1:8000/'.
```

The local ZenServer exposes a local REST API through which clients can access the same data available in your active Profile. You can also specify a different profile by passing the `--profile=$PROFILE_NAME` command-line argument, or change the HTTP port and address on which the ZenServer is accepting requests using `--port=$PORT` and `--ip-address=$IP_ADDRESS`.The following example creates a new SQL profile named `zen-server` and starts a ZenServer with that profile that listens to port 8080 on all interfaces. This is a typical use case where you might want to expose the ZenServer to other machines in your network:

```
$ zenml server down
Shutting down the local ZenService instance.

$ zenml profile create -t sql zen-server
Running without an active repository root.
Running with active profile: 'devel' (global)
Initializing profile zen-server...
Registering default stack...
Registered stack with name 'default'.
Profile 'zen-server' successfully created.

$ zenml server up --port 8080 --ip-address 0.0.0.0 --profile zen-server
Running without an active repository root.
Starting a new ZenServer local instance.
ZenServer running at 'http://0.0.0.0:8080/'.
```

The status of the local ZenServer instance can be checked at any time using:

```
$ zenml server status
The ZenServer status is active and running at http://0.0.0.0:8080/..
```

and shut down using:

```
$ zenml server down
Shutting down the local ZenService instance.
```

To launch the ZenServer manually, without using the ZenML CLI, you can use the `uvicorn` command line launch utility. For example, the equivalent of the previous `zenml server up` example would be:

```
ZENML_PROFILE_NAME=zen-server uvicorn zenml.zen_server.zen_server_api:app \
    --port 8080 --host 0.0.0.0
```

Deploy the ZenServer with Docker

The ZenServer can be deployed as a Docker container. To persist the ZenStore information between container restarts, the Docker container should be configured to mount a host volume where its global configuration is kept. The following is an example of starting a Docker container with a ZenServer instance exposed on the local port 8080 and using a local `zenserver` folder to persist the ZenML data between container restarts:

```
mkdir zenserver
docker run -it -d -p 8080:8000 \
    -v $PWD/zenserver:/zenserver \
    -e ZENML_CONFIG_PATH=/zenserver \
    zenmldocker/zenml-server \
    uvicorn zenml.zen_server.zen_server_api:app --host 0.0.0.0
```

ZenServer User Management

All clients connecting to a ZenServer instance must use a user account. This is currently available only as a rudimentary user management system until the ZenServer is fully developed.

A `default` user is automatically created when the ZenServer is started. The ZenML CLI can be used to register additional users, either directly into the Profile that the ZenServer is running on, or remotely, from any ZenML client:

```
$ zenml profile describe
Running without an active repository root.
Running with active profile: 'zenml-remote' (global)
'zenml-remote' Profile Configuration (ACTIVE)
┏━━━━━━━━━━━━━━┯━━━━━━━━━━━━━━━━━━━━━━━━━━━━┓
┃ PROPERTY     │ VALUE                      ┃
┠──────────────┼────────────────────────────┨
┃ NAME         │ zenml-remote               ┃
┠──────────────┼────────────────────────────┨
┃ STORE_URL    │ http://192.168.178.40:8080 ┃
```

```
┠──────────────┼─────────────────────────────┨
┃ STORE_TYPE   │ rest                        ┃
┠──────────────┼─────────────────────────────┨
┃ ACTIVE_STACK │ default                     ┃
┠──────────────┼─────────────────────────────┨
┃ ACTIVE_USER  │ default                     ┃
┗━━━━━━━━━━━━━━┷━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┛

$ zenml user list
Running without an active repository root.
Running with active profile: 'zenml-remote' (global)
┏━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┯━━━━━━━━━━━━━━━━━━━━━━━━━━━━┯━━━━━━━━━┓
┃                  ID                  │ CREATION_DATE              │ NAME    ┃
┠──────────────────────────────────────┼────────────────────────────┼─────────┨
┃ 085cb51e-37cf-4661-a8cb-ec885b218387 │ 2022-05-16 18:12:16.355773 │ default ┃
┗━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┷━━━━━━━━━━━━━━━━━━━━━━━━━━━━┷━━━━━━━━━┛

$ zenml user create aria
Running without an active repository root.
Running with active profile: 'zenml-remote' (global)

$ zenml user list
Running without an active repository root.
Running with active profile: 'zenml-remote' (global)
┏━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┯━━━━━━━━━━━━━━━━━━━━━━━━━━━━┯━━━━━━━━━┓
┃                  ID                  │ CREATION_DATE              │ NAME    ┃
┠──────────────────────────────────────┼────────────────────────────┼─────────┨
┃ 085cb51e-37cf-4661-a8cb-ec885b218387 │ 2022-05-16 18:12:16.355773 │ default ┃
┠──────────────────────────────────────┼────────────────────────────┼─────────┨
┃ 16dfc5da-9462-4a36-8502-15f0204501cf │ 2022-05-16 18:39:09.461282 │ aria    ┃
┗━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┷━━━━━━━━━━━━━━━━━━━━━━━━━━━━┷━━━━━━━━━┛
```

ZenServer API Reference

For more details on the ZenServer REST API, spin up the service and visit `http://$SERVICE_URL/docs` to see the OpenAPI specification.

Resources

Contribution Guide

Contribute to ZenML!

We would love to develop ZenML together with our community! The best way to get started is to select any issue from the `good-first-issue` label . If you would like to contribute, please review our Contributing

Guide for all relevant details.

▯ Where to get help

The first point of call should be our Slack group. Ask your questions about bugs or specific use cases and someone from the core team will respond.

▯ License

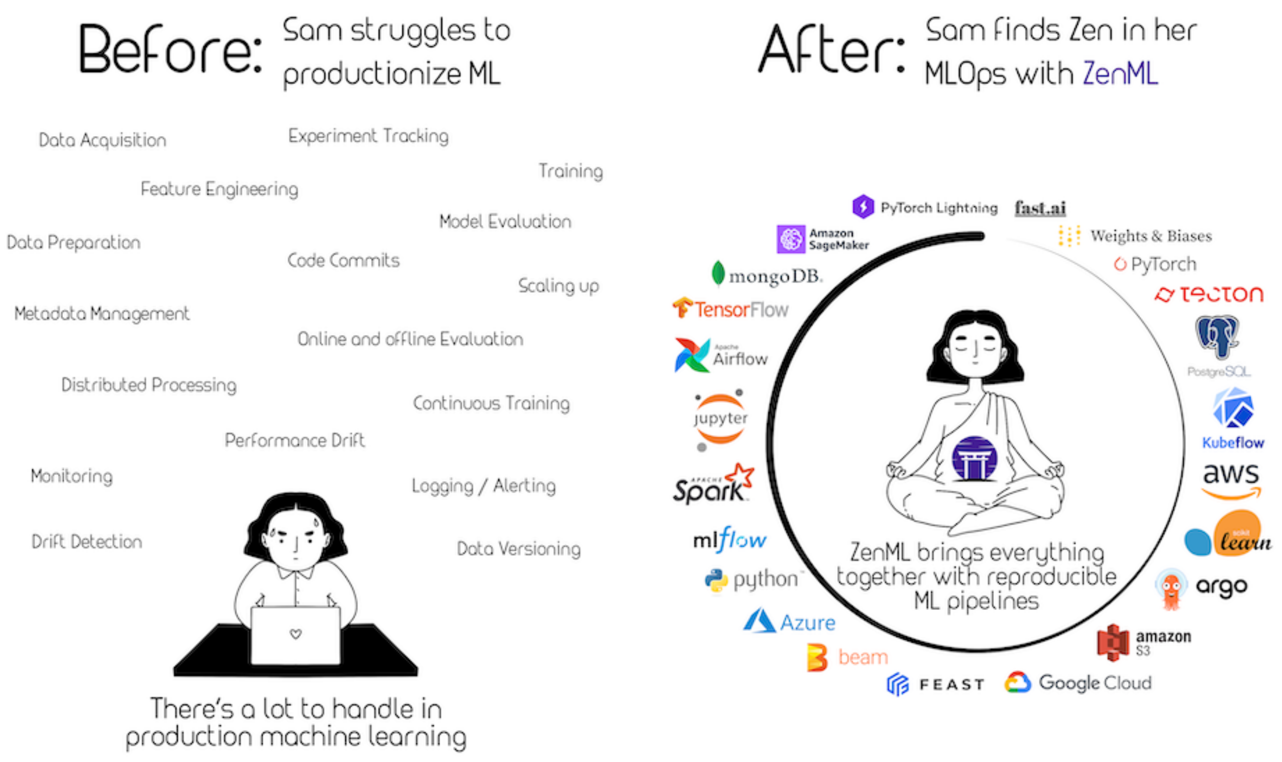
ZenML is distributed under the terms of the Apache License Version 2.0. A complete version of the license is available in the LICENSE.md in this repository. Any contribution made to this project will be licensed under the Apache License Version 2.0.

▯ Email

Email us any time at support@zenml.io for additional queries.

External Integration Guide

How to integrate with ZenML



ZenML integrates with a bunch of tools from the MLOps landscape

One of the main goals of ZenML is to find some semblance of order in the ever-growing MLOps landscape. ZenML already provides numerous integrations into many popular tools, and allows you to extend ZenML in order to fill in any gaps that are remaining.

However, what if you want to make your extension of ZenML part of the main codebase, to share it with others? If you are such a person, e.g., a tooling provider in the ML/MLOps space, or just want to contribute a tooling integration to ZenML, this guide is intended for you.

Step 1: Categorize your integration

In Extending ZenML, we already looked at the categories and abstractions that core ZenML defines. In order to create a new integration into ZenML, you would need to first find the categories that your integration belongs to. The list of categories can be found on this page.

Note that one integration may belong to different categories: For example, the cloud integrations (AWS/GCP/Azure) contain container registries, artifact stores, metadata stores, etc.

Step 2: Create individual stack components

Each category selected above would correspond to a stack component. You can now start developing these individual stack components by following the detailed instructions on each stack component page.

Before you package your new components into an integration, you may want to first register them with the `zenml <STACK_COMPONENT> flavor register` command and use/test them as a regular custom flavor. E.g., when developing an orchestrator you can use:

```
zenml orchestrator flavor register <THE-SOURCE-PATH-OF-YOUR-ORCHESTRATOR>
```

See the docs on extensibility of the different components here or get inspired by the many integrations that are already implemented, for example the mlflow experiment tracker.

Step 3: Integrate into the ZenML repo

You can now start the process of including your integration into the base ZenML package. Follow this checklist to prepare everything:

Clone Repo

Once your stack components work as a custom flavor, you can now clone the main zenml repository and follow the contributing guide to set up your local environment for develop.

Create the integration directory

All integrations live within `src/zenml/integrations/` in their own subfolder. You should create a new folder in this directory with the name of your integration.

Define the name of your integration in constants

In `zenml/integrations/constants.py`, add:

```
EXAMPLE_INTEGRATION = "<name-of-integration>"
```

This will be the name of the integration when you run:

```
 zenml integration install <name-of-integration>
```

Or when you specify pipeline requirements:

```
from zenml.pipelines import pipeline
from zenml.integrations.constants import <EXAMPLE_INTEGRATION>

@pipeline(required_integrations=[<EXAMPLE_INTEGRATION>])
def custom_pipeline():
    ...
```

Create the integration class __init__.py

In `src/zenml/integrations/<YOUR_INTEGRATION>/__init__.py` you must now create an new class, which is a subclass of the `Integration` class, set some important attributes (`NAME` and `REQUIREMENTS`), and overwrite the `flavors` class method.

```
from typing import List

from zenml.enums import StackComponentType
from zenml.integrations.constants import <EXAMPLE_INTEGRATION>
from zenml.integrations.integration import Integration
from zenml.zen_stores.models import FlavorWrapper

# This is the flavor that will be used when registering this stack component
#  `zenml <type-of-stack-component> register ... -f example-orchestrator-flavor`

EXAMPLE_ORCHESTRATOR_FLAVOR = <"example-orchestrator-flavor">

# Create a Subclass of the Integration Class
class ExampleIntegration(Integration):
    """Definition of Example Integration for ZenML."""

    NAME = <EXAMPLE_INTEGRATION>
    REQUIREMENTS = ["<INSERT PYTHON REQUIREMENTS HERE>"]

    @classmethod
    def flavors(cls) -> List[FlavorWrapper]:
        """Declare the stack component flavors for the <EXAMPLE> integration."""
        return [
            # Create a FlavorWrapper for each Stack Component this Integration implements
            FlavorWrapper(
                name=EXAMPLE_ORCHESTRATOR_FLAVOR,
                source="<path.to.the.implementation.of.the.component",      # Give the sou
                type=StackComponentType.<TYPE-OF-STACK-COMPONENT>,      # Define which comp
                integration=cls.NAME,
            )
        ]

ExampleIntegration.check_installation() # this checks if the requirements are installed
```

Have a look at the MLflow Integration as an example for how it is done.

Copy your implementation(s)

As said above, each Integration can have implementations for multiple ZenML stack components. Generally the outer repository structure `src/zenml/<stack-component>/<base-component-impl.py` is reflected inside the integration folder: `integrations/<name-of-integration>/<stack-component>/<custom-component-impl.py`

Here, you can now copy the code you created in Step 2 above.

Import in all the right places

The Integration itself must be imported within `src/zenml/integrations/__init__.py` .

The Implementation of the individual stack components also needs to be imported within the sub-modules: `src/zenml/integrations/<name-of-integration>/<specifc-component>/__init__.py.` For example, in the mlflow integration, the experiment tracker and deployer are imported in the sub-module `__init__.py` files.

Step 4: Create a PR and celebrate

You can now create a PR to ZenML and wait for the core maintainers to take a look. Thank you so much for your contribution to the code-base, rock on!

Best Practices

Best practices, recommendations, and tips from the ZenML team

Recommended Repository Structure

```
├── notebooks                   <- All notebooks in one place
│   ├── *.ipynb
├── pipelines                   <- All pipelines in one place
│   ├── training_pipeline
│   │   ├── .dockerignore
│   │   ├── config.yaml
│   │   ├── Dockerfile
│   │   ├── training_pipeline.py
│   │   ├── requirements.txt
│   ├── deployment_pipeline
│   │   ├── ...
├── steps                       <- All steps in one place
│   ├── loader_step
│   │   ├── loader_step.py
│   ├── training_step
│   │   ├── ...
├── .dockerignore
├── .gitignore
├── config.yaml
├── Dockerfile
├── README.md
├── requirements.txt
├── run.py
└── setup.sh
```

Best Practices and Tips

Pass integration requirements to your pipelines through the decorator:

```python
from zenml.integrations.constants import TENSORFLOW
from zenml.pipelines import pipeline


@pipeline(required_integrations=[TENSORFLOW])
def training_pipeline():
    ...
```

Writing your pipeline like this makes sure you can change out the orchestrator at any point without running into dependency issues.

Do not overlap `required_integrations` and `requirements`

Setting requirements twice can lead to unexpected behavior as you will end up with *only* one of the two defined package versions, which might cause problems.

Nest `pipeline_instance.run()` in `if __name__ == "__main__"`

```python
pipeline_instance = training_pipeline(...)

if __name__ == "__main__":
    pipeline_instance.run()
```

This ensures that loading the pipeline from elsewhere does not also run it.

Never call the pipeline instance `pipeline` or a step instance `step`

Doing this will overwrite the imported `pipeline` and `step` decorators and lead to failures at later stages if more steps and pipelines are decorated there.

```python
from zenml.pipelines import pipeline


@pipeline
def first_pipeline():
    ...


pipeline = first_pipeline(...)


@pipeline  # The code will fail here
def second_pipeline():
    ...
```

Explicitly set `enable_cache` at the `@pipeline` level

Caching is enabled by default for ZenML Pipelines. It is good to be explicit, though, so that it is clear when looking at the code if caching is enabled or disabled for any given pipeline.

Explicitly disable caching when loading data from filesystem or external APIs

ZenML inherently uses caching. However, this caching relies on changes of input artifacts to invalidate the cache. In case a step has external data sources like external APIs or filesystems, caching should be disabled explicitly for the step.

Enable cache explicitly for steps that have a `context` argument, if they don't invalidate the caching behavior

Cache is implicitly disabled for steps that have a context argument, because it is assumed that you might use the step context to retrieve artifacts from the artifact store that are unrelated to the current step. However, if that is not the case, and your step logic doesn't invalidate the caching behavior, it would be better to explicitly enable the cache for your step.

Don't use the same metadata stores across multiple artifact stores

You might run into issues as the metadata store will point to artifacts in inactive artifact stores.

Use Profiles to manage Stacks

Using Profiles allows you to separate your ZenML stacks and work locally within independent ZenML instances. See our docs on profiles to learn more.

Use unique pipeline names across projects, especially if used with the same metadata store

Pipeline names are their unique identifiers, so using the same name for different pipelines will create a mixed history of runs between the two pipelines.

Check which integrations are required for registering a stack component

You can do so by running `zenml flavor list` and installing the missing integration(s) with `zenml integration install`.

Initialize the ZenML repository in the root of the source code tree of a project, even if it's optional

This will set the ZenML project root for the project and create a local profile. The advantage is that you create and maintain your active stack on a project level.

Include a `.dockerignore` in the ZenML repository to exclude files and folders from the container images built by ZenML for containerized environments

Containerized Orchestrators and Step Operators load your complete project files into a Docker image for execution. To speed up the process and reduce Docker image sizes, exclude all unnecessary files (like data, virtual environments, git repos, etc.) within the `.dockerignore`.

Use `get_pipeline(pipeline=...)` instead of indexing (`[-1]`) to retrieve previous pipelines

When inspecting pipeline runs it is tempting to access the pipeline views directly by their index, but the pipelines within your `Repository` are sorted by time of first run, so the pipeline at `[-1]` might not be the one you are expecting.

```python
from zenml.repository import Repository

first_pipeline.run()
second_pipeline.run()
first_pipeline.run()

repo = Repository()
repo.get_pipelines()
>>> [PipelineView('first_pipeline'), PipelineView('second_pipeline')]

# This is the recommended explicit way to retrieve your specific pipeline
# using the pipeline class if you have it at hand
repo.get_pipeline(pipeline=first_pipeline)
```

```
# Alternatively you can also use the name of the pipeline
repo.get_pipeline(pipeline="first_pipeline")
```

Have your imports relative to your `.zen` directory OR have your imports relative to the root of your repository in cases when you don't have a `.zen` directory (=> which means to have the runner at the root of your repository)

ZenML uses the `.zen` repository root to resolve the class path of your functions and classes in a way that is portable across different types of environments such as containers. If a repository is not present, the location of the main Python module is used as an implicit repository root.

Put your runners in the root of the repository

Putting your pipeline runners in the root of the repository ensures that all imports that are defined relative to the project root resolve for the pipeline runner.

Tips

- Use `zenml GROUP explain` to explain what everything is
- Run `zenml stack up` after switching stacks (but this is also enforced by validations that check if the stack is up)

For a practical example on all of the above, please check out ZenFiles which are practical end-to-end projects built using ZenML.

Global Configuration

What is the global ZenML config

ZenML has two main locations where it stores information on the local machine. These are the *Global Config* and the *Repository* (see here for more information on the repository).

Most of the information stored by ZenML on a machine, such as the global settings, the configured Profiles, and even the configured Stacks and Stack Components, is kept in a folder commonly referred to as the *ZenML Global Config Directory* or the *ZenML Config Path*. The location of this folder depends on the operating system type and the current system user, but is usually located in the following locations:

- Linux: `~/.config/zenml`
- Mac: `~/Library/Application Support/ZenML`
- Windows: `C:\Users\%USERNAME%\AppData\Local\ZenML`

The default location may be overridden by setting the `ZENML_CONFIG_PATH` environment variable to a custom value. The current location of the *Global Config Directory* used on a system can be retrieved by running the following command:

```
python -c 'from zenml.utils.io_utils import get_global_config_directory; print(get_global_
```

Manually altering or deleting the files and folders stored under the *ZenML Global Config Directory* is not recommended, as this can break the internal consistency of the ZenML configuration. As an alternative, ZenML provides CLI commands that can be used to manage the information stored there:

- `zenml analytics` - manage the analytics settings
- `zenml profile` - manage configuration Profiles
- `zenml stack` - manage Stacks
- `zenml <stack-component>` - manage Stack Components
- `zenml clean` - to be used only in case of emergency, to bring the ZenML configuration back to its default factory state

The first time that ZenML is run on a machine, it creates the *Global Config Directory* and initializes the default configuration in it, along with a default Profile and Stack:

```
$ zenml stack list
Unable to find ZenML repository in your current working directory (/home/stefan)
or any parent directories. If you want to use an existing repository which is in
a different location, set the environment variable 'ZENML_REPOSITORY_PATH'. If
you want to create a new repository, run zenml init.
Initializing the ZenML global configuration version to 0.7.3
Creating default profile...
Initializing profile default...
Registering default stack...
Registered stack component with type 'orchestrator' and name 'default'.
Registered stack component with type 'metadata_store' and name 'default'.
Registered stack component with type 'artifact_store' and name 'default'.
Registered stack with name 'default'.

Created and activated default profile.
Running without an active repository root.
Running with active profile: 'default' (global)
┏━━━━━━━━┯━━━━━━━━━━━━┯━━━━━━━━━━━━━━━━┯━━━━━━━━━━━━━━━━┯━━━━━━━━━━━━━━┓
┃ ACTIVE │ STACK NAME │ ARTIFACT_STORE │ METADATA_STORE │ ORCHESTRATOR ┃
┠────────┼────────────┼────────────────┼────────────────┼──────────────┨
┃        │ default    │ default        │ default        │ default      ┃
┗━━━━━━━━┷━━━━━━━━━━━━┷━━━━━━━━━━━━━━━━┷━━━━━━━━━━━━━━━━┷━━━━━━━━━━━━━━┛
```

The following is an example of the layout of the *Global Config Directory* immediately after initialization:

```
/home/stefan/.config/zenml   <- Global Config Directory
├── config.yaml              <- Global Configuration Settings
├── local_stores             <- Every Stack component that stores information
|   |                        locally will have its own subdirectory here.
```

```
│   ├── 09fcdb1c-4079-4d20-afdb-957965405863   <- Local Store path for the `default`
|                                              local Artifact Store
|
└── profiles                  <- root path where Profiles data (stacks, components,
    |                         etc) are stored by default. Every Profile will have
    |                         its own subdirectory here, unless the Profile is
    |                         configured with a custom configuration path.
    |
    └── default               <- configuration folder for the `default` Profile.
        ├── artifact_stores
        │   └── default.yaml
        ├── metadata_stores
        │   └── default.yaml
        ├── orchestrators
        │   └── default.yaml
        └── stacks.yaml
```

As shown above, the *Global Config Directory* stores the following information:

1. The `global.yaml` file stores the global configuration settings: the unique ZenML user ID, the active Profile, the analytics related options and a list of all configured Profiles, along with their configuration attributes, such as the active Stack set for each Profile. This is an example of the `global.yaml` file contents immediately after initialization:

```yaml
activated_profile: default
analytics_opt_in: true
profiles:
default:
    active_stack: default
    active_user: default
    name: default
    store_type: local
    store_url: file:///home/stefan/.config/zenml/profiles/default
user_id: 4b773740-fddc-46ee-938e-1c78a075cfc7
user_metadata: null
version: 0.7.3
```

2. The `local_stores` directory is where some "local" flavors of Stack Components, such as the `local` Artifact Store, the `sqlite` Metadata Store or the `local` Secrets Manager persist data locally. Every local Stack Component will have its own subdirectory here named after the Stack Component's unique UUID. One notable example is the `local` Artifact Store flavor that, when part of the active Stack, stores all the artifacts generated by Pipeline runs in the designated local directory.

3. The `profiles` directory is used as a default root path location where ZenML stores information about the Stacks, Stack Components, custom Stack Component flavors etc. that are configured under each Profile. Every Profile will have its own subdirectory here, unless the Profile is explicitly created with a custom configuration path. (See the `zenml profile` command and the section on ZenML Profiles for more information about Profiles.)

In addition to the above, you may also find the following files and folders under the *Global Config Directory*, depending on what you do with ZenML:

- `zenml_examples` - used as a local cache by the `zenml example` command, where the pulled ZenML examples are stored.
- `kubeflow` - this is where the Kubeflow orchestrators that are part of a Stack store some of their configuration and logs.

Accessing the global configuration in Python

You can access the global ZenML configuration from within Python using the `zenml.config.global_config.GlobalConfiguration` class:

```python
from zenml.config.global_config import GlobalConfiguration
config = GlobalConfiguration()
```

This can be used to manage your profiles and other global settings from within Python. For instance, we can use it to create and activate a new profile:

```python
from zenml.repository import Repository
from zenml.config.global_config import GlobalConfiguration
from zenml.config.profile_config import ProfileConfiguration

repo = Repository()
config = GlobalConfiguration()

# Create a new profile called "local"
profile = ProfileConfiguration(name="local")
config.add_or_update_profile(profile)

# Set the profile as active profile of the repository
repo.activate_profile("local")
```

To explore all possible operations that can be performed via the `GlobalConfiguration`, please consult the API docs sections on GlobalConfiguration.

System Environmental Variables

How to control ZenML behavior with environmental variables

There are a few pre-defined environmental variables that can be used to control some of the behavior of ZenML. See the list below with default values and options:

Choose from `INFO`, `WARN`, `ERROR`, `CRITICAL`, `DEBUG`:

```
ZENML_LOGGING_VERBOSITY=INFO
```

Explicit path to the ZenML repository:

```
ZENML_REPOSITORY_PATH
```

Name of the active profile, see Setting Stacks and Profiles with Environment Variables:

```
ZENML_ACTIVATED_PROFILE
```

Name of the active stack, see Setting Stacks and Profiles with Environment Variables:

```
ZENML_ACTIVATED_STACK
```

Setting to `false` disables analytics:

```
ZENML_ANALYTICS_OPT_IN=true
```

Setting to `true` switches to developer mode:

```
ZENML_DEBUG=false
```

When `true`, this prevents a pipeline from executing:

```
ZENML_PREVENT_PIPELINE_EXECUTION=false
```

Set to `false` to disable the `rich` traceback:

```
ZENML_ENABLE_RICH_TRACEBACK=true
```

Path to global ZenML config:

```
ZENML_CONFIG_PATH
```

Setting to `false` disables integrations logs suppression:

```
ZENML_SUPPRESS_LOGS=false
```

Usage Analytics

What are the usage statistics ZenML collects

In order to help us better understand how the community uses **ZenML**, the pip package reports *anonymized* usage statistics. You can always opt-out by using the CLI command:

```
zenml config analytics opt-out
```

Currently, opting in and out of analytics is a global setting applicable to all ZenML repositories within your system.

Why ZenML collects analytics

In addition to the community at large, **ZenML** is created and maintained by a startup based in Munich, Germany called ZenML GmbH. We're a team of techies that love MLOps and want to build tools that fellow developers would love to use in their daily work. This is us, if you want to put faces to the names!

However, in order to improve **ZenML** and understand how it is being used, we need to use analytics to have an overview of how it is used 'in the wild'. This not only helps us find bugs but also helps us prioritize features and commands that might be useful in future releases. If we did not have this information, all we really get is pip download statistics and chatting with people directly, which while being valuable, is not enough to seriously better the tool as a whole.

How ZenML collects these statistics

ZenML uses `Segment` as the data aggregation library for all our analytics. The entire code is entirely visible and can be seen at `zenml_analytics.py`. The main function is the `track` function that triggers a Segment Analytics Track event, which runs on a separate background thread from the main thread.

None of the data sent can identify you individually but allows us to understand how **ZenML** is being used holistically.

What does ZenML collect?

ZenML triggers an asynchronous Segment Track Event on the following events, which is also viewable in the `zenml_analytics.py` file in the GitHub repository.

```
# Pipelines
    RUN_PIPELINE = "Pipeline run"
    GET_PIPELINES = "Pipelines fetched"
    GET_PIPELINE = "Pipeline fetched"

```

```python
# Repo
INITIALIZE_REPO = "ZenML initialized"

# Profile
INITIALIZED_PROFILE = "Profile initialized"

# Components
REGISTERED_STACK_COMPONENT = "Stack component registered"
UPDATED_STACK_COMPONENT = "Stack component updated"

# Stack
REGISTERED_STACK = "Stack registered"
SET_STACK = "Stack set"
UPDATED_STACK = "Stack updated"

# Analytics opt in and out
OPT_IN_ANALYTICS = "Analytics opt-in"
OPT_OUT_ANALYTICS = "Analytics opt-out"

# Examples
RUN_EXAMPLE = "Example run"
PULL_EXAMPLE = "Example pull"

# Integrations
INSTALL_INTEGRATION = "Integration installed"
```

In addition, each Segment Track event collects the following metadata:

- A unique UUID that is anonymous.
- The **ZenML** version.
- Operating system information, e.g. Ubuntu Linux 16.04

FAQ

Find answers to the most frequently asked questions about ZenML

Is ZenML just another orchestrator like Airflow, Kubeflow, Flyte, etc?

Not really! An orchestrator in MLOps is the system component that is responsible for executing and managing the execution of a ML pipeline. ZenML is a framework that allows you to run your pipelines on whatever orchestrator you like, and we coordinate with all the other parts of an ML system in production. There are standard orchestrators that ZenML supports out-of-the-box, but you are encouraged to write your own orchestrator in order to gain more control as to exactly how your pipelines are executed!

Can I use tool X? How does tool Y integrate with ZenML?

Take a look at our examples directory, which showcases detailed examples for each integration that ZenML supports out of the box

The ZenML team and community is constantly working to include more tools and integrations to the above list (check out the roadmap for more details). You can upvote features you'd like and add your ideas to the roadmap.

Most importantly, ZenML is extensible and we encourage you to use it with whatever other tools you require as part of your ML process and system(s). Check out our documentation on how to get started with extending ZenML to learn more!

How do I install on an M1 Mac

If you have an M1 Mac machine and you are encountering an error while trying to install ZenML, please try to setup brew and pyenv with Rosetta 2 and then install ZenML. The issue arises because some of the dependencies aren't fully compatible with the vanilla ARM64 Architecture. The following links may be helpful.

Pyenv with Apple Silicon Install Python Under Rosetta 2

How can I make ZenML work with my custom tool? How can I extend or build on ZenML?

This depends on the tool and its respective MLOps category. We have a full guide on this over here!

Why did you build ZenML?

We built it because we scratched our own itch while deploying multiple machine learning models in production over the past three years. Our team struggled to find a simple yet production-ready solution whilst developing large-scale ML pipelines. We built a solution for it that we are now proud to share with all of you! Read more about this backstory on our blog here.

How can I contribute?

We would love to develop ZenML together with our community! The best way to get started is to select any issue from the `good-first-issue` label. If you would like to contribute, please review our Contributing Guide for all relevant details.

How can I learn more about MLOps?

Check out our ZenBytes repository and course, where you learn MLOps concepts in a practical manner with the ZenML framework. Other great resources are:

- MadeWithML
- Full Stack Deep Learning
- CS 329S: Machine Learning Systems Design

Why should I use ZenML?

ZenML pipelines are designed to be written early on the development lifecycle. Data scientists can explore their pipelines as they develop towards production, switching stacks from local to cloud deployments with ease. You can read more about why we started building ZenML on our blog. By using ZenML in the early stages of your project, you get the following benefits:

- Extensible so you can build out the framework to suit your specific needs
- Reproducibility of training and inference workflows
- A simple and clear way to represent the steps of your pipeline in code
- Batteries-included integrations: bring all your favorite tools together
- Easy switch between local and cloud stacks
- Painless deployment and configuration of infrastructure

How can I be sure you'll stick around as a tool?

The team behind ZenML have a shared vision of making MLOps simple and accessible to accelerate problem solving in the world. We recently raised our seed round to fulfill this vision, and you can be sure we're here to stay!

Plus, ZenML is and always will be an open-source effort, which lowers the risk of it just going away any time soon.

How can I speak with the community?

The first point of call should be our Slack group. Ask your questions about bugs or specific use cases and someone from the core team will respond.

Which license does ZenML use?

ZenML is distributed under the terms of the Apache License Version 2.0. A complete version of the license is available in the LICENSE.md in this repository. Any contribution made to this project will be licensed under the Apache License Version 2.0.

Reference

Glossary

Glossary of terminology used in ZenML

Annotator

Annotators are a stack component that enables the use of data annotation as part of your ZenML stack and pipelines. You can use the associated CLI command to launch annotation, configure your datasets and get stats on how many labeled tasks you have ready for use.

Artifact

Artifacts are the data that power your experimentation and model training. It is actually steps that produce artifacts, which are then stored in the artifact store.

Artifacts can be serialized and deserialized (i.e. written and read from the Artifact Store) in different ways like `TFRecord`s or saved model pickles, depending on what the step produces.The serialization and deserialization logic of artifacts is defined by Materializers.

Artifact Store

An artifact store is a place where artifacts are stored. These artifacts may have been produced by the pipeline steps, or they may be the data first ingested into a pipeline via an ingestion step.

CLI

Our command-line tool is your entry point into ZenML. You install this tool and use it to set up and configure your repository to work with ZenML. A single `init` command serves to get you started, and then you can provision the infrastructure that you wish to work with using the `stack register` command with the relevant arguments passed in.

Container Registry

A container registry is a store for (Docker) containers. A ZenML workflow involving a container registry would see you spinning up a Kubernetes cluster and then deploying a pipeline to be run on Kubeflow Pipelines. As part of the deployment to the cluster, the ZenML base image would be downloaded (from a cloud container registry) and used as the basis for the deployed 'run'. When you are running a local Kubeflow stack, you would therefore have a local container registry which stores the container images you create that bundle up your pipeline code. These images would in turn be built on top of a base image or custom image of your choice.

DAG

Pipelines are traditionally represented as DAGs. DAG is an acronym for Directed Acyclic Graph.

- Directed, because the nodes of the graph (i.e. the steps of a pipeline), have a sequence. Nodes do not

exist as free-standing entities in this way.
- Acyclic, because there must be one (or more) straight paths through the graph from the beginning to the end. It is acyclic because the graph doesn't loop back on itself at any point.
- Graph, because the steps of the pipeline are represented as nodes in a graph.

ZenML follows this paradigm and it is a useful mental model to have in your head when thinking about how the pieces of your pipeline get executed and how dependencies between the different stages are managed.

Integrations

An integration is a third-party tool or platform that implements a ZenML abstraction. A tool can implement many abstractions and therefore an integration can have different entrypoints for the user. We have a consistently updated integrations page which shows all current integrations supported by the ZenML core team here. However, as ZenML is a framework users are encouraged to use these as a guideline and implement their own integrations by extending the various ZenML abstractions.

Materializers

A materializer defines how and where Artifacts live in between steps. It is used to convert a ZenML artifact into a specific format. They are most often used to handle the input or output of ZenML steps, and can be extended by building on the `BaseMaterializer` class. We care about this because steps are not just isolated pieces of work; they are linked together and the outputs of one step might well be the inputs of the next.

We have some built-in ways to serialize and deserialize the data flowing between steps. Of course, if you are using some library or tool which doesn't work with our built-in options, you can write your own custom materializer to ensure that your data can be passed from step to step in this way. We use our `fileio` utilities to do the disk operations without needing to be concerned with whether we're operating on a local or cloud machine.

Metadata

Metadata are the pieces of information tracked about the pipelines, experiments and configurations that you are running with ZenML. Metadata are stored inside the metadata store.

Metadata Store

The configuration of each pipeline, step and produced artifacts are all tracked within the metadata store. The metadata store is an SQL database, and can be `sqlite` or `mysql`.

Orchestrator

An orchestrator manages the running of each step of the pipeline, administering the actual pipeline runs. You can think of it as the 'root' of any pipeline job that you run during your experimentation.

Parameter

When we think about steps as functions, we know they receive input in the form of artifacts. We also know that they produce output (also in the form of artifacts, stored in the artifact store). But steps also take parameters. The parameters that you pass into the steps are also (helpfully!) stored in the metadata store. This helps freeze the iterations of your experimentation workflow in time, so you can return to them exactly as you ran them.

Pipeline

Pipelines are designed as basic Python functions. They are created by using decorators appropriate to the specific use case you have. The moment it is `run`, a pipeline is compiled and passed directly to the orchestrator, to be run in the orchestrator environment.

Within your repository, you will have one or more pipelines as part of your experimentation workflow. A ZenML pipeline is a sequence of tasks that execute in a specific order and yield artifacts. The artifacts are stored within the artifact store and indexed via the metadata store. Each individual task within a pipeline is known as a step.

Repository

Every ZenML project starts inside a ZenML repository and, it is at the core of all ZenML activity. Every action that can be executed within ZenML must take place within such a repository. ZenML repositories are denoted by a local `.zen` folder in your project root where various information about your local configuration lives, e.g., the active Stack that you are using to run pipelines, is stored.

Runner Scripts

A runner script is a Python file, usually called `run.py` and located at the root of a ZenML repository, which has the code to actually create a pipeline run. The code usually looks like this:

```python
from pipelines.my_pipeline import my_pipeline
from steps.step_1 import step_1

if __name__ == "__main__":
    p = my_pipeline(
        step_1=step_1(),
    )
    p.run()
```

Secret

A ZenML Secret is a grouping of key-value pairs. These are accessed and administered via the ZenML Secret Manager (a stack component).

Secrets are distinguished by having different schemas. An AWS SecretSchema, for example, has key-value pairs for `AWS_ACCESS_KEY_ID` and `AWS_SECRET_ACCESS_KEY` as well as an optional `AWS_SESSION_TOKEN`. If you don't specify a schema at the point of registration, ZenML will set the schema as `ArbitrarySecretSchema`, a kind of default schema where things that aren't attached to a grouping can be stored.

Secrets Manager

Most projects involving either cloud infrastructure or of a certain complexity will involve secrets of some kind. You use secrets, for example, when connecting to AWS, which requires an `access_key_id` and a `secret_access_key` which it (usually) stores in your `~/.aws/credentials` file.

You might find you need to access those secrets from within your Kubernetes cluster as it runs individual steps, or you might just want a centralized location for the storage of secrets across your project. ZenML offers a local secrets manager and an integration with the managed AWS Secrets Manager.

Stack

A stack is made up of the following three core components:

- An Artifact Store
- A Metadata Store
- An Orchestrator

A ZenML stack also happens to be a Pydantic `BaseSettings` class, which means that there are multiple ways to use it.

```
zenml stack register STACK_NAME \
    -m METADATA_STORE_NAME \
    -a ARTIFACT_STORE_NAME \
    -o ORCHESTRATOR_NAME
```

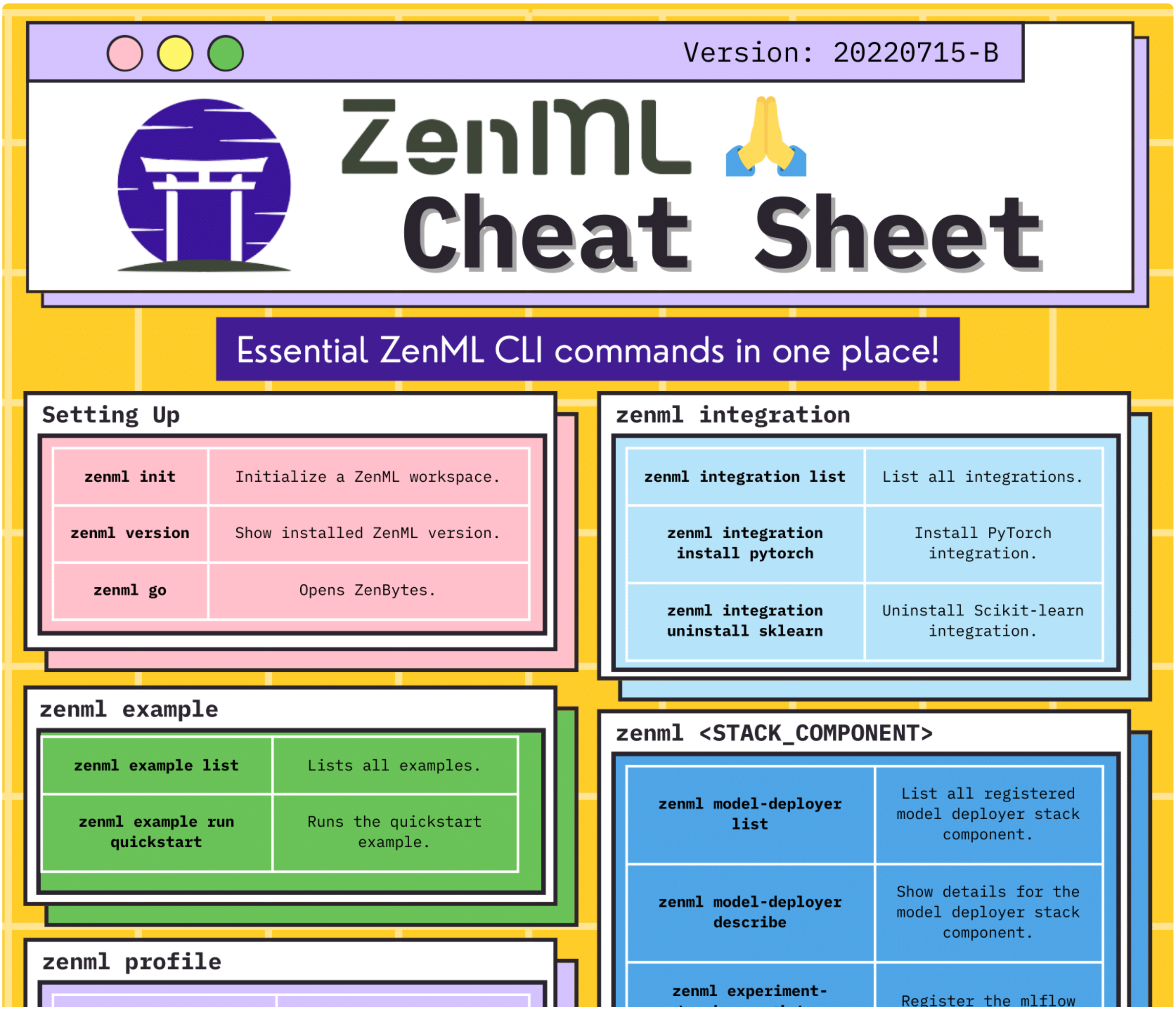
Step

A step is a single piece or stage of a ZenML pipeline. Think of each step as being one of the nodes of the DAG. Steps are responsible for one aspect of processing or interacting with the data / artifacts in the pipeline.

CLI Cheat Sheet

Cheat sheet for ZenML CLI commands.

Download as PDF



Version: 20220715-B

ZenML 🙏 Cheat Sheet

Essential ZenML CLI commands in one place!

Setting Up

zenml init	Initialize a ZenML workspace.
zenml version	Show installed ZenML version.
zenml go	Opens ZenBytes.

zenml integration

zenml integration list	List all integrations.
zenml integration install pytorch	Install PyTorch integration.
zenml integration uninstall sklearn	Uninstall Scikit-learn integration.

zenml example

zenml example list	Lists all examples.
zenml example run quickstart	Runs the quickstart example.

zenml <STACK_COMPONENT>

zenml model-deployer list	List all registered model deployer stack component.
zenml model-deployer describe	Show details for the model deployer stack component.
zenml experiment-	Register the mlflow

zenml profile

zenml profile create myzenprofile	Create a profile with the name myzenprofile.
zenml profile set myzenprofile	Set the active profile to myzenprofile.

tracker register mlflow_tracker --flavor=mlflow	experiment tracker as a stack component.
zenml experiment-tracker delete myexptracker	Delete registered experiment tracker named myexptracker.

zenml stack

zenml stack set mystackname	Set the mystackname as the active stack.
zenml stack list	List all stacks.
zenml stack describe	Describe the current active stack.

zenml stack get	Get the active stack.
zenml stack up	Provision resources for the active stack.
zenml stack down	Suspend resources for the active stack.

zenml stack copy mycurrentstack mynewstack	Copy mycurrentstack stack to mynewstack.
zenml stack rename oldstackname newstackname	Rename a stack from oldstackname to newstackname.
zenml stack delete myoldstack	Delete myoldstack.

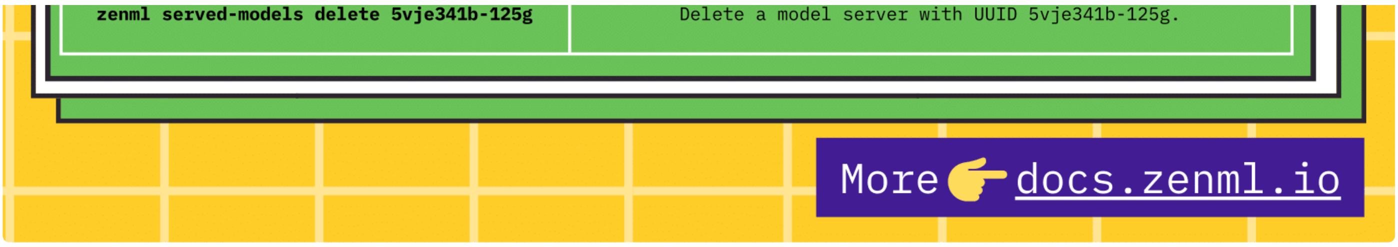
zenml stack export mystack.yaml	Export a stack to a YAML file.
zenml stack import mystack.yaml	Import a stack from a YAML file.
zenml stack recipe deploy myrecipename	Deploys a recipe called myrecipename.

zenml secret

zenml secret register example_secret --example_secret_key=example_secret_value	Create a secret called example_secret which contains a single key-value pair: {example_secret_key: example_secret_value}.
zenml secret delete example_secret	Delete the secret called example secret.
zenml secret get example_secret	Get a secret called example_secret.
zenml secret list	List all secrets.

zenml served-models

zenml served-models list	Get a list of all served models within the model-deployer stack component.
zenml served-models start 5vje341b-125g	Start a model server with UUID 5vje341b-125g.
zenml served-models describe 5vje341b-125g	Describe a served model with the UUID 5vje341b-125g.
zenml served-models get-url 5vje341b-125g	Get the prediction URL of a model server with UUID 5vje341b-125g.
zenml served-models stop 5vje341b-125g	Stop a model server with UUID 5vje341b-125g.



zenml served-models delete 5vje341b-125g	Delete a model server with UUID 5vje341b-125g.

More 👉 docs.zenml.io

ZenML CLI Cheat Sheet