# mltrace

*Release 0.16*

**Shreya Shankar**

**Nov 05, 2021**

# CONTENTS

mltrace is a lightweight, open-source Python tool to get "bolt-on" observability in ML pipelines. It offers the following:

- interface to define data and ML tests for components in pipelines

- coarse-grained lineage and tracing

- Python API to log versions of data and pipeline components

- database to store information about component runs

- UI and CLI to show the trace of steps in a pipeline taken to produce an output, flag outputs for review, and identify what steps of the pipeline to investigate first when debugging

mltrace is designed specifically for Agile or multidisciplinary teams collaborating on machine learning or complex data pipelines. A more detailed blog post on why the tool was developed can be found here.

# DESIGN PRINCIPLES

- Simplicity (users should know *exactly* what the tool does)
- **Rinse and repeat other successful designs**
    - Decorator design similar to Dagster solids
    - Logging design similar to MLFlow tracking
- API designed for both engineers and data scientists
- UI designed for people to help triage issues *even if they didn't build the ETL or models themselves*

# ROADMAP

We are actively working on the following:

- Component input and output monitoring

- Stateful testing (i.e., being able to use historical component inputs outputs in testing and monitoring)

- API to log from any type of file, not just a Python file

- Prometheus integrations to monitor component output distributions

- Support for finer-grained lineage (at the record level)

# GUIDES

## 3.1 Changelog

- #226: Adds functionality to run triggers before and after components are run. Thanks *@aditim1359* for taking this on!

- : Added ability to create tests and execute them before and after components are run. Also, the web app has a React Router refactor, thanks to *@Boyuan-Deng*.

> **Warning:** This change is requries a DB migration. You can follow the documentation to perform the migration if you are using a release prior to this one.

- #176: Adds functionality to log git tags an example of how to use DVC with mltrace. Thanks *@jeannefukumaru* for taking this on!

- #178: Adds the review feature to allow users to flag problematic outputs and determine common component runs used in producing the outputs. See details here: *Using the reviewer tool*

- : Added the review feature to aid in debugging erroneous outputs and functionality to log git tags to integrate with DVC.

> **Warning:** This change is requries a DB migration. You can follow the documentation to perform the migration if you are using a release prior to this one.

- #56: Adds CLI commands as an alternative to the UI. Thanks *@ariG23498* for taking this on! See documentation on how to use CLI here: *Using the CLI*

- #76: Add a staleness feature to component runs to hint whether the component needs to be rerun. See details here: *Staleness*

- : Added CLI (command line utilities) and component run staleness features.

> **Warning:** This change is requries a DB migration. You can follow the documentation to perform the migration if you are using a release prior to this one.

## 3.2 Quickstart

To use `mltrace`, you first need to set up a server to log to. You will need the following utilities:

- Python 3.7 or later

- Docker

- Postgres

- Yarn

### 3.2.1 Server

On the machine you would like to run the server (can be your local machine), clone the latest release of mltrace. In the root directory, start the server by running:

```
docker-compose build
docker-compose up [-d]
```

You can access the UI by navigating to `<SERVER'S IP ADDRESS>:8080` (or localhost:8080 if you are running locally) in your browser.

### 3.2.2 Client

To log to the server using the client library, install the latest version of mltrace on the machine executing your pipelines by running:

```
pip install mltrace
```

Next, you will need to set the database URI. It is recommended to use environment variables for this. To set the database address, set the `DB_SERVER` variable:

```
export DB_SERVER=<SERVER'S IP ADDRESS>
```

where `<SERVER'S IP ADDRESS>` is either the IP address of a remote machine or `localhost` if running locally. If, when you set up the server, you changed the URI in `docker-compose.yaml`, you can set the `DB_URI` variable (which represents the entire database URI) client-side instead of `DB_SERVER`.

## 3.3 Concepts

Machine learning pipelines, or even complex data pipelines, are made up of several *components.* For instance:

Keeping track of data flow in and out of these components can be tedious, especially if multiple people are collaborating on the same end-to-end pipeline.This is because in ML pipelines, *different* artifacts are produced (inputs and outputs) when the *same* component is run more than once.

Knowing data flow is a precursor to debugging issues in data pipelines. `mltrace` also determines whether components of pipelines are stale.

### 3.3.1 Data model

The two prominent client-facing abstractions are the *Component* and *ComponentRun* abstractions.

#### Test

The `Test` abstraction represents some reusable computation to perform on component inputs and outputs. Defining a `Test` is similar to writing a unit test:

```python
from mltrace import Test

class OutliersTest(Test):
    def __init__(self):
        super().__init__(name='outliers')

    def testSomething(self; df: pd.DataFrame):
        ....

    def testSomethingElse(self; df: pd.DataFrame):
        ....
```

Tests can be defined and passed to components as arguments, as described in the section below.

#### mltrace.Component

The `Component` abstraction represents a stage in a pipeline and its static metadata, such as:

- name
- description
- owner
- tags (optional list of string values to reference the component by)
- tests

Tags are generally useful when you have multiple components in a higher-level stage. For example, ETL computation could consist of different components such as "cleaning" or "feature generation." You could create the "cleaning" and "feature generation" components with the tag `etl` and then easily query component runs with the `etl` tag in the UI.

Components have a life-cycle:

- `c = Component(...)`: construction of the component object
- `c.beforeTests`: a list of `Tests` to run before the component is run
- `c.run`: a decorator for a user-defined function that represents the component's computation
- `c.afterTests`: a list of `Tests` to run after the component is run

Putting it all together, we can define our own component:

```python
from mltrace import Component

class Featuregen(Component):
    def __init__(self, beforeTests=[], afterTests=[OutliersTest]):
```

```
    super().__init__(
        name="featuregen",
        owner="spark-gymnast",
        description="Generates features for high tip prediction problem",
        tags=["nyc-taxicab"],
        beforeTests=beforeTests,
        afterTests=afterTests,
    )
```

And in our main application code, we can decorate any feature generation function:

```
@Featuregen().run
def generateFeatures(df: pd.DataFrame):
    # Generate features
    df = ...
    return df
```

See the next page for a more in-depth tutorial on instrumenting a pipeline.

### mltrace.ComponentRun

The `ComponentRun` abstraction represents an instance of a `Component` being run. Think of a `ComponentRun` instance as an object storing *dynamic* metadata for a `Component`, such as:

- start timestamp

- end timestamp

- inputs

- outputs

- git hash

- source code

- dependencies (you do not need to manually declare)

If you dig into the codebase, you will find another abstraction, the `IOPointer`. Inputs and outputs to a `ComponentRun` are stored as `IOPointer` objects. You do not need to explicitly create an `IOPointer` – the abstraction exists so that `mltrace` can easily find and store dependencies between `ComponentRun` objects.

You will not need to explicitly define all of these variables, nor do you have to create instances of a `ComponentRun` yourself. See the next section for logging functions and an example.

## 3.3.2 Staleness

We define a component run as "stale" if it may need to be rerun. Currently, `mltrace` detects two types of staleness in component runs:

1. A significant number of days (default 30) have passed between when a component run's inputs were generated and the component is run

2. At the time a component is run, its dependencies have fresher runs that began before the component run started

We are working on "data drift" as another measure of staleness.

---

### 3.3.3 _Reviewing erroneous outputs

Oftentimes there is a bug or error in some output of a pipeline that surfaces after the output has been produced. ML and data bugs are extra elusive because it can take a nontrivial number of mispredicted or buggy outputs to indicate that there is actually an issue with the pipeline. Given a set of erroneous outputs, it can be challenging to know where to begin debugging! Fortunately, `mltrace` can help with this.

The idea here is to identify the common ``ComponentRun``s used in producing the erroneous outputs, as these might provide a good suggestion for what component to debug first or artifacts (inputs and outputs) to dive into. See steps on how to use the reviewer tool in the :ref:querying section.

## 3.4 Logging

`mltrace` functions can be added to existing Python files to log component and run information to the server. Logging can be done via a decorator or explicit Python API. All logging functions are defined in the *mltrace* module, which you can install via pip:

```
pip install mltrace
```

For this example, we will add logging functions to a hypothetical `cleaning.py` that loads raw data and cleans it. In your Python file, before you call any logging functions, you will need to make sure you are connected to your server. You can easily do so by setting the environment variable `DB_SERVER` to your server's IP address:

```
export DB_SERVER=SERVER_IP_ADDRESS
```

where `SERVER_IP_ADDRESS` is your server's IP address or "localhost" if you are running locally. You can also call `mltrace.set_address(SERVER_IP_ADDRESS)` in your Python script instead if you do not want to set the environment variable.

If you plan to use the auto logging functionalities for component run inputs and outputs (turned off by default), you will need to set the environment variable `SAVE_DIR` to the directory you want to save versions of your inputs and outputs to. The default is `.mltrace` in the user directory.

### 3.4.1 Component creation

For runs of components to be logged, you must first create the components themselves using *mltrace.Component*. You can subclass the main Component class if you want to make a custom Component, for example:

```python
from mltrace import Component

class Cleaning(Component):
    def __init__(self, name, owner, tags=[], beforeTests=[], afterTests=[]):

        super().__init__(
            name="cleaning_" + name,
            owner=owner,
            description="Basic component to clean raw data",
            tags=tags,
            beforeTests=beforeTests,
            afterTests=afterTests,
        )
```

Components are intended to be defined once and reused throughout your application. You can define them in a separate file or folder and import them into your main Python application. If you do not want a custom component, you can also just use the default Component class, as shown below.

## 3.4.2 Logging runs

### Decorator approach

Suppose we have a function `clean` in our `cleaning.py` file:

```python
import pandas as pd


def clean_data(df: pd.DataFrame) -> str:
    # Do some cleaning
    clean_df = ...
    return clean_df
```

We can include the *run()* decorator such that every time this function is run, dynamic information is logged:

```python
from mltrace import Component
import pandas as pd

c = Component(
    name="cleaning",
    owner="plumber",
    description="Cleans raw NYC taxicab data",
)

@c.run(auto_log=True)
def clean_data(df: pd.DataFrame) -> str:
    # Do some cleaning
    clean_df = ...
    return clean_df
```

We will refer to `clean_data` as the clean_data as the decorated component run function. The `auto_log` parameter is set to False by default, but you can set it to True to automatically log inputs and outputs. If `auto_log` is True, `mltrace` will save and log paths to any dataframes, variables with "data" or "model" in their names, and any other variables greater than 1MB. `mltrace` will save to the directory defined by the environment variable `SAVE_DIR`. If `MLTRACE_DIR` is not set, `mltrace` will save to a `.mltrace` folder in the user directory.

If you do not set `auto_log` to True, then you will need to manually define your input and output variables in the *run()* function. Note that `input_vars` and `output_vars` correspond to variables in the function. Their values at the time of return are logged. The start and end times, git hash, and source code snapshots are automatically captured. The dependencies are also automatically captured based on the values of the input variables.

**Python approach**

You can also create an instance of a *ComponentRun* and log it using *mltrace.log_component_run()* yourself for greater flexibility. An example of this is as follows:

```python
from datetime import datetime
from mltrace import ComponentRun
from mltrace import get_git_hash, log_component_run
import pandas as pd

def clean_data(filename: str) -> str:
    # Create ComponentRun object
    cr = ComponentRun("cleaning")
    cr.set_start_timestamp()
    cr.add_input(filename)
    cr.git_hash = get_git_hash() # Sets git hash, not source code snapshot!

    df = pd.read_csv(filename)
    # Do some cleaning
    ...
    # Save cleaned dataframe
    clean_version = filename[:-4] + '_clean_{datetime.utcnow().strftime("%m%d%Y%H%M%S")}.
→csv'
    df.to_csv(clean_version)

    # Finish logging
    cr.set_end_timestamp()
    cr.add_output(clean_version)
    log_component_run(cr)

    return clean_version
```

Note that in *log_component_run()*, `set_dependencies_from_inputs` is set to `True` by default. You can set it to `False` if you want to manually specify the names of the components that this component run depends on. To manually specify a dependency, you can call *set_upstream()* with the dependent component name or list of component names before you call *log_component_run()*.

## 3.4.3 Testing

You can define Tests, or reusable blocks of computation, to run before and after components are run. To define a test, you need to subclass the *Test* class. Defining a test is similar to defining a Python unittest, for example:

```python
from mltrace import Test

class OutliersTest(Test):
    def __init__(self):
        super().__init__(name='outliers')

    def testComputeStats(self; df: pd.DataFrame):
        # Get numerical columns
        num_df = df.select_dtypes(include=["number"])

        # Compute stats
```

(continues on next page)

```python
        stats = num_df.describe()
        print("Dataframe statistics:")
        print(stats)

    def testZScore(
        self,
        df: pd.DataFrame,
        stdev_cutoff: float = 5.0,
        threshold: float = 0.05,
    ):
        """
        Checks to make sure there are no outliers using z score cutoff.
        """
        # Get numerical columns
        num_df = df.select_dtypes(include=["number"])

        z_scores = (
            (num_df - num_df.mean(axis=0, skipna=True))
            / num_df.std(axis=0, skipna=True)
        ).abs()

        if (z_scores > stdev_cutoff).to_numpy().sum() > threshold * len(df):
            print(
                f"Number of outliers: {(z_scores > stdev_cutoff).to_numpy().sum()}"
            )
            print(f"Outlier threshold: {threshold * len(df)}")
            raise Exception("There are outlier values!")
```

Any function you expect to execute as a test must be prefixed with the name `test` in lowercase, like `testSomething`. Arguments to test functions must be defined in the decorated component run function signature if the tests will be run before the component run function; otherwise the arguments to test functions must be defined as variables somewhere in the decorated component run function. You can integrate the tests into components in the constructor:

```python
from mltrace import Component
import pandas as pd

c = Component(
    name="cleaning",
    owner="plumber",
    description="Cleans raw NYC taxicab data",
    beforeTests=[OutliersTest],
)

@c.run(auto_log=True)
def clean_data(df: pd.DataFrame) -> str:
    # Do some cleaning
    clean_df = ...
    return clean_df
```

At runtime, the `OutliersTest` test functions will run before the `clean_data` function. Note that all arguments to the test functions executed in `beforeTests` must be arguments to `clean_data`. All arguments to the test functions executed in `afterTests` must be variables defined somewhere in `clean_data`.

### 3.4.4 End-to-end example

To see an example of `mltrace` integrated into a Python pipeline, check out this tutorial. The full pipeline with `mltrace` integrations is defined in `solutions/main.py`.

## 3.5 Querying

The simplest way to query the logged runs is to use the `mltrace` UI. There are also some functions defined in the `mltrace` module for querying.

### 3.5.1 Using the UI

As mentioned in the *Quickstart*, you should set up the database, server, and UI using `docker-compose`. The UI starts up showing the results of the `recent` command, or the most recent component runs logged.
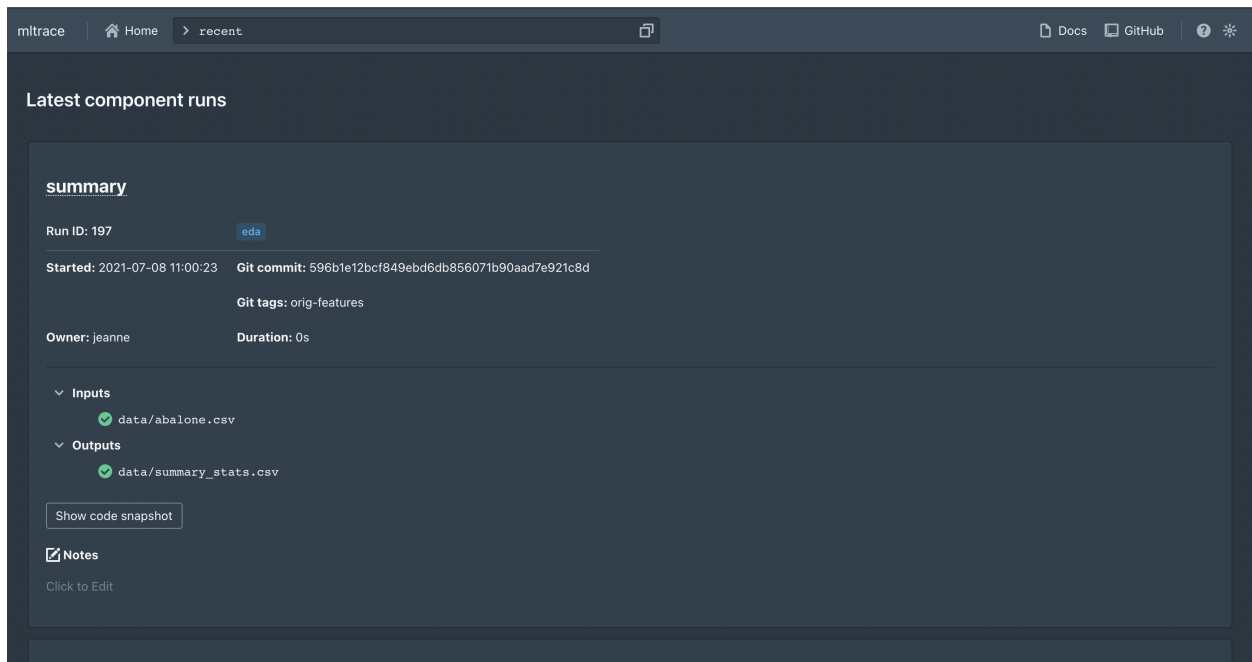


You can toggle between light and dark mode using the moon or sun button at the top right. You can also view a list of supported commands by clicking the help or question mark button at the top right. The commands currently supported are below:

| Command | Description | Usage |
|---|---|---|
| `recent` | Displays the most recent runs across all components. Also serves as the default or "home" page. | `recent` |
| `history` | Displays most recent runs for a given component name. Shows latest 10 runs by default, but you can specify the number of runs you want to see by appending a positive integer to the command. | `history COMPONENT_NAME 15` |
| `inspect` | Displays information such as inputs/outputs, code, git snapshot, owner, and more for a given component run ID. | `inspect COMPONENT_RUN_ID` |
| `trace` | Displays a trace of versioned steps that produced a given output. | `trace OUTPUT_NAME` |
| `tag` | Displays all components with the given tag name. | `tag TAG_NAME` |

`flag` | Flags an output ID for further review. Necessary to see any results from the `review` command. | `flag OUTPUT_ID` |

`unflag` | Unflags an output ID. Removes this output ID from any results from the `review` command. | `unflag OUTPUT_ID` |

`review` | Shows a list of output IDs flagged for review and the common component runs involved in producing the output IDs. The component runs are sorted from most frequently occurring to least frequently occurring. | `review` |

### 3.5.2 Using the CLI

The following commands are supported via CLI:

- `history()`
- `recent()`
- `trace()`
- `flag()`
- `unflag()`
- `review()`

You can execute `mltrace --help` in your shell for usage instructions, or you can execute `mltrace command --help` for usage instructions for a specific command.

### 3.5.3 Using the reviewer tool

To use the reviewer tool, you first need to "flag" some output IDs. One way to do this is to toggle the status indicator on the output ID when viewing the `ComponentRun`'s info card in the UI:

Another way to do this is to execute the `flag` commmand in either the UI or CLI. To flag an output, simply execute:

```
mltrace flag OUTPUT_ID
```

in the CLI or `flag OUTPUT_ID` in the UI command bar. You can flag as many output IDs as you would like. Once you have flagged some outputs, you can execute:

```
mltrace review
```

in the CLI or `review` in the UI command bar to see a list of all the output IDs you have flagged and the ``ComponentRun``s used to produce those outputs. The list of ``ComponentRun``s is sorted by highest to lowest coverage, where coverage for *ComponentRun* X is defined by the fraction of the erroneous outputs that X was involved in producing. Here's an example of how the UI might look:
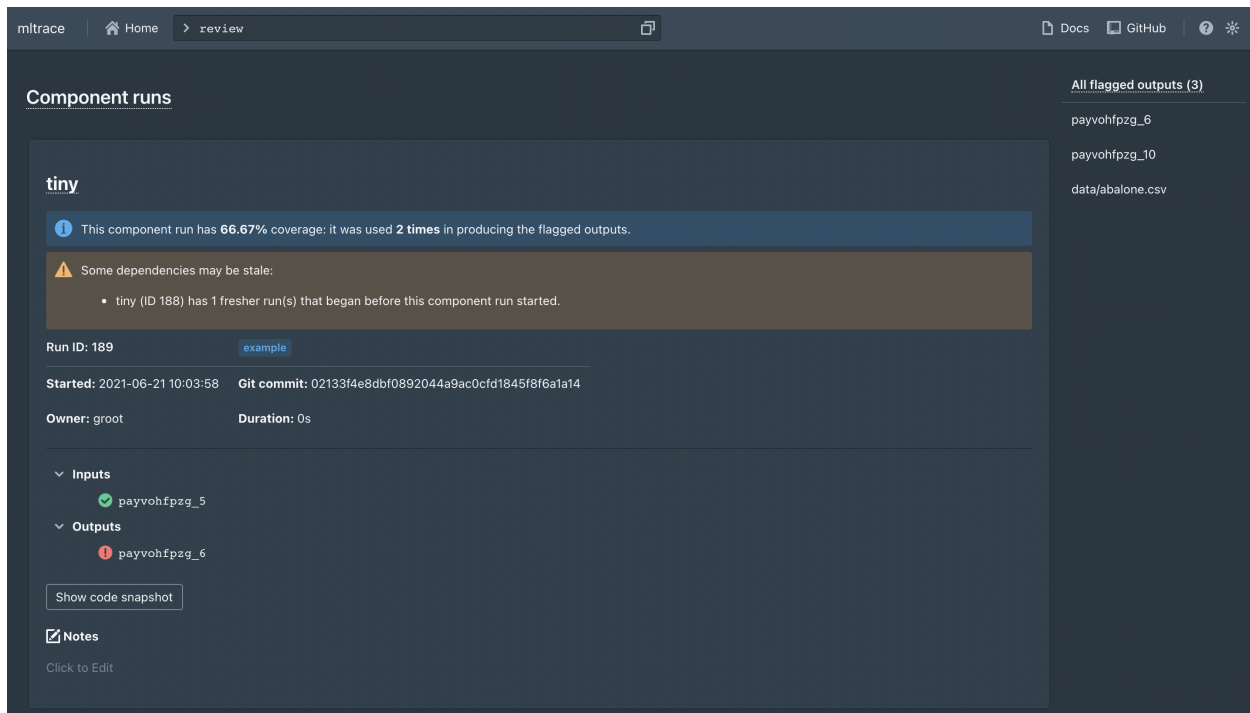


To begin debugging, we recommend looking at the code, inputs, and outputs for the ``ComponentRun``s with highest coverage as a first step to see if there are any logical errors or data issues.

You can unflag output IDs by using the `unflag` command, which has usage patterns similar to `flag`.

### 3.5.4 `mltrace` module functions

- *backtrace()*
- *get_component_information()*
- *get_component_run_information()*
- get_components_with_owner()
- get_components_with_tag()
- *get_history()*
- *get_recent_run_ids()*
- *review_flagged_outputs()*

## 3.6 mltrace package

### 3.6.1 Module contents

**class** `mltrace.`**`Component`**(*name: str = '', owner: str = '', description: str = '', beforeTests: list = [], afterTests: list = [], tags: List[str] = []*)

    Bases: `mltrace.entities.base.Base`

    **`afterRun`**(*\*\*local_vars*)

        Computation to execute after running a component. Will run all test objects listed in afterTests.

    **property afterTests:  list**

    **`beforeRun`**(*\*\*kwargs*)

        Computation to execute before running a component. Will run each test object listed in beforeTests.

    **property beforeTests:  list**

    **property description:  str**

    **property name:  str**

    **property owner:  str**

    **`run`**(*inputs: List[str] = [], outputs: List[str] = [], input_vars: List[str] = [], output_vars: List[str] = [], input_kwargs: Dict[str, str] = {}, output_kwargs: Dict[str, str] = {}, endpoint: bool = False, staleness_threshold: int = 2592000, auto_log: bool = False, \*user_args, \*\*user_kwargs*)

        Decorator around the function executed: c = Component() @c.run def my_function(arg1, arg2):

            do_something()

        arg1 and arg2 are the arguments passed to the beforeRun and afterRun methods. We first execute the beforeRun method, then the function itself, then the afterRun method with the values of the args at the end of the function.

        ADD DESCRIPTION HERE ABOUT INPUT VARIABLEs and what they are

    **property tags:  List[str]**

**class** mltrace.**ComponentRun**(*component_name: str*, *notes: str = ''*, *start_timestamp:*
*Optional[datetime.datetime] = None*, *end_timestamp:*
*Optional[datetime.datetime] = None*, *inputs:*
*List[mltrace.entities.io_pointer.IOPointer] = []*, *outputs:*
*List[mltrace.entities.io_pointer.IOPointer] = []*, *git_hash: Optional[str] = None*,
*git_tags: Optional[List[str]] = None*, *code_snapshot: Optional[str] = None*, *id:*
*Optional[str] = None*, *stale: List[str] = []*, *dependencies: List[str] = []*)

Bases: mltrace.entities.base.Base

Component Run abstraction.

**add_input**(*inp: Union[str, mltrace.entities.io_pointer.IOPointer]*, *pointer_type:*
*Optional[mltrace.db.models.PointerTypeEnum] = None*)
Add a single input (instance of IOPointer).

**add_inputs**(*inputs: List[Union[str, mltrace.entities.io_pointer.IOPointer]]*)
Add a list of inputs (each element should be an instance of IOPointer).

**add_output**(*out: Union[str, mltrace.entities.io_pointer.IOPointer]*, *pointer_type:*
*Optional[mltrace.db.models.PointerTypeEnum] = None*)
"Add a single output (instance of IOPointer).

**add_outputs**(*outputs: List[Union[str, mltrace.entities.io_pointer.IOPointer]]*)
Add a list of outputs (each element should be an instance of IOPointer).

**property code_snapshot: str**

**property component_name: str**

**property dependencies: List[str]**

**property end_timestamp: datetime.datetime**

**property git_hash: str**

**property git_tags: List[str]**

**property id: str**

**property inputs: List[mltrace.entities.io_pointer.IOPointer]**

**property notes: str**

**property outputs: List[mltrace.entities.io_pointer.IOPointer]**

**set_end_timestamp**(*ts: Optional[datetime.datetime] = None*)

**set_start_timestamp**(*ts: Optional[datetime.datetime] = None*)

**set_upstream**(*dependencies: Union[str, List[str]]*)
Set dependencies for this ComponentRun. API similar to Airflow set_upstream. It will grab the most recent run for the dependency name.

**property stale: List[str]**

**property start_timestamp: datetime.datetime**

**class** mltrace.**IOPointer**(*name: str*, *value: Any = ''*, *pointer_type: mltrace.db.models.PointerTypeEnum =*
*PointerTypeEnum.UNKNOWN*, *flag: bool = False*)

Bases: mltrace.entities.base.Base

**property flag: bool**

**property name: str**

**property pointer_type: mltrace.db.models.PointerTypeEnum**

property value:  Any

**class** mltrace.**Test**(*name: str = ''*)

> Bases: object

> **getTestMethods**()
>> Gets all methods in this class that start with "test"

> **property name**

> **runTests**(*\*\*kwargs*)
>> Runs all tests in this class.

mltrace.**add_notes_to_component_run**(*component_run_id: str*, *notes: str*) → str

> Adds notes to component run.

mltrace.**backtrace**(*output_pointer: str*)

> Prints trace for an output id. Returns list of tuples (level, ComponentRun) where level is how many hops away the node is from the node that produced the output_id.

mltrace.**clean_db**()

> Deletes database and reinitializes tables.

mltrace.**create_component**(*name: str*, *description: str*, *owner: str*, *tags: List[str] = []*)

> Creates a component entity in the database.

mltrace.**create_random_ids**(*num_outputs=1*) → List[str]

> Returns a list of num_outputs ids that a client can use to tag outputs.

mltrace.**flag_output_id**(*output_id: str*) → bool

> Sets the flag property of an IOPointer to true.

mltrace.**get_all_tags**() → List[str]

mltrace.**get_component_information**(*component_name: str*) → mltrace.entities.base_component.Component

> Returns a Component with the name, info, owner, and tags.

mltrace.**get_component_run_information**(*component_run_id: str*) →
>> mltrace.entities.component_run.ComponentRun

> Returns a ComponentRun object.

mltrace.**get_components**(*tag=''*, *owner=''*) → List[mltrace.entities.base_component.Component]

> Returns all components with the specified owner and/or tag. Else, returns all components.

mltrace.**get_db_uri**() → str

mltrace.**get_git_hash**() → str

> Gets hash of the parent git repo.

mltrace.**get_git_tags**() → List[str]

> Gets tags associated with commit of parent git repo, if exists ref:https://stackoverflow.com/questions/34932306/get-tags-of-a-commit

mltrace.**get_history**(*component_name: str*, *limit: int = 10*, *date_lower: Union[datetime.datetime, str] = datetime.datetime(1, 1, 1, 0, 0)*, *date_upper: Union[datetime.datetime, str] = datetime.datetime(9999, 12, 31, 23, 59, 59, 999999)*) → List[mltrace.entities.component_run.ComponentRun]

> Returns a list of ComponentRuns that are part of the component's history.

mltrace.**get_io_pointer**(*io_pointer_id: str*, *io_pointer_val: Optional[Any] = None*, *create=True*)

> Returns IO Pointer metadata.

mltrace.**get_recent_run_ids**(*limit: int = 5*, *last_run_id=None*)

> Returns most recent component run ids.

mltrace.**load**(*pathname: str*)
> Loads joblib file at pathname.

mltrace.**log_component_run**(*component_run: mltrace.entities.component_run.ComponentRun*,
*set_dependencies_from_inputs=True*, *staleness_threshold: int = 2592000*)
> Takes client-facing ComponentRun object and logs it to the DB.

mltrace.**register**(*component_name: str*, *inputs: List[str] = []*, *outputs: List[str] = []*, *input_vars: List[str] = []*,
*output_vars: List[str] = []*, *input_kwargs: Dict[str, str] = {}*, *output_kwargs: Dict[str, str] =
{}*, *endpoint: bool = False*, *staleness_threshold: int = 2592000*, *auto_log: bool = False*)

mltrace.**review_flagged_outputs**()
> Finds common ComponentRuns for a group of flagged outputs. Returns a list of ComponentRuns and occurrence counts in the group of flagged outputs, sorted by descending count and then alphabetically.

mltrace.**save**(*obj*, *pathname: Optional[str] = None*) → str
> Saves joblib object to pathname.

mltrace.**set_address**(*address: str*)

mltrace.**set_db_uri**(*uri: str*)

mltrace.**tag_component**(*component_name: str*, *tags: List[str]*)
> Adds tags to existing component.

mltrace.**unflag_all**()

mltrace.**unflag_output_id**(*output_id: str*) → bool
> Sets the flag property of an IOPointer to false.

mltrace.**web_trace**(*output_id: str*)

# PYTHON MODULE INDEX

## m
mltrace, 18