NYU FRE 7773 - Week 8

Machine Learning in Financial Engineering

Jacopo Tagliabue

How to Organize ML Projects

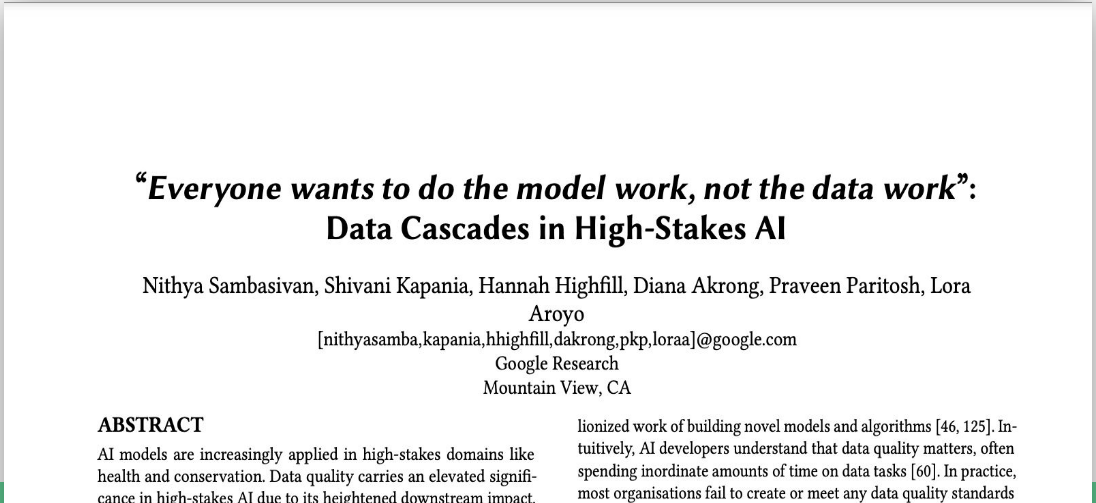
Machine Learning in Financial Engineering

Jacopo Tagliabue

MLSys: why?

MLSys: Use Cases

- Models are a tiny part of ML platforms, and often the least problematic (with some *caveat*);
- while everybody wants to do the model work, data work is often equally (or more) important in practice.



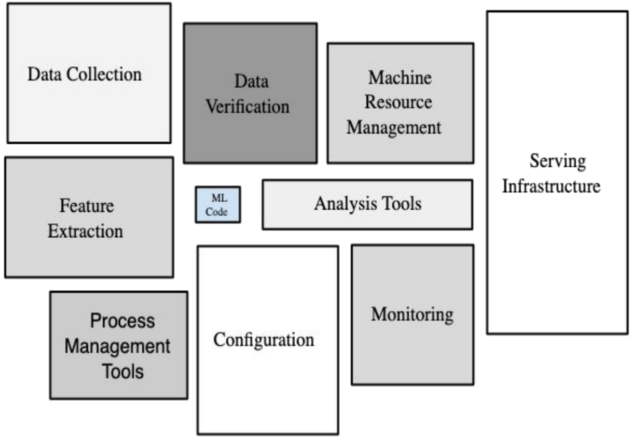
"*Everyone wants to do the model work, not the data work*": Data Cascades in High-Stakes AI

Nithya Sambasivan, Shivani Kapania, Hannah Highfill, Diana Akrong, Praveen Paritosh, Lora Aroyo
[nithyasamba,kapania,hhighfill,dakrong,pkp,loraa]@google.com
Google Research
Mountain View, CA

ABSTRACT

AI models are increasingly applied in high-stakes domains like health and conservation. Data quality carries an elevated significance in high-stakes AI due to its heightened downstream impact,

lionized work of building novel models and algorithms [46, 125]. Intuitively, AI developers understand that data quality matters, often spending inordinate amounts of time on data tasks [60]. In practice, most organisations fail to create or meet any data quality standards



Three major phases of ML projects

Data	*Training*	*Inference*
• Gathering • Cleaning • Testing • Encoding • ….	• Modelling • Hyper-param tuning • Testing • …	• Serving • Caching • Monitoring • ….

Three major phases of ML projects

Data	*Training*	*Inference*
• **Gathering** • **Cleaning** • Testing • Encoding • ….	• **Modelling** • **Hyper-param tuning** • **Testing** • …	• Serving • Caching • Monitoring • ….

Three major phases of ML projects at FRE 7773

Data

Training

Inference

Your Laptop*

Cloud

* Conditions apply. In particular, Metaflow sandboxes are *also cloud!*

MLSys: A Finance Example

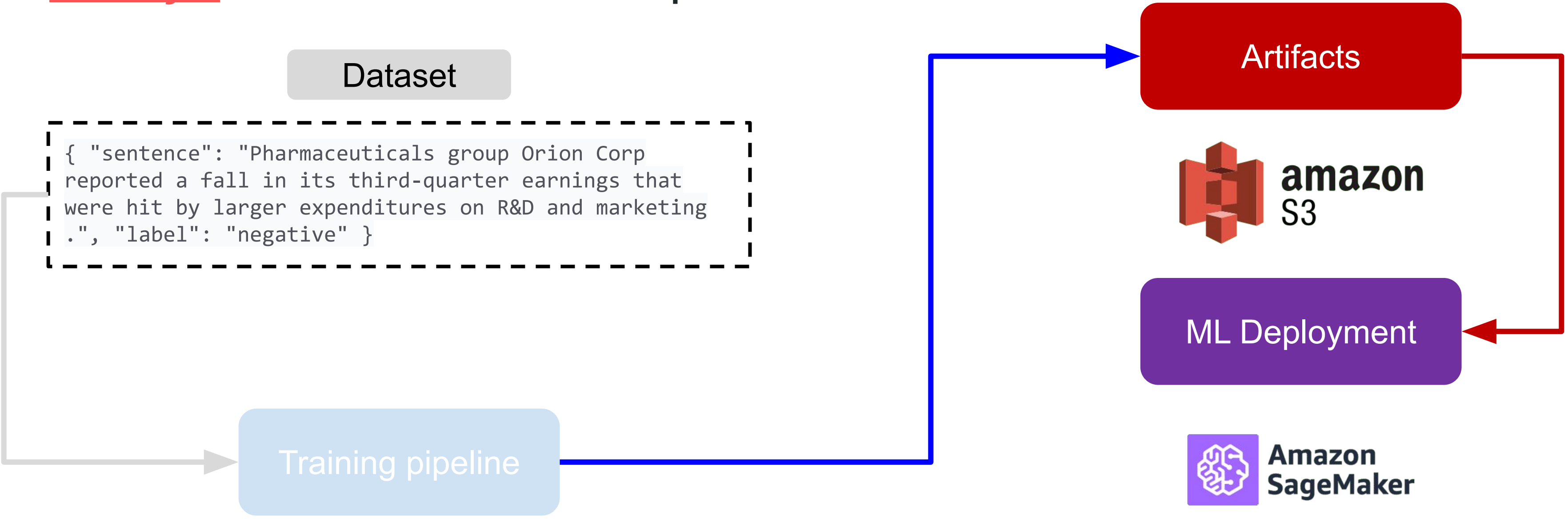
Dataset

```
{ "sentence": "Pharmaceuticals group Orion Corp
reported a fall in its third-quarter earnings that
were hit by larger expenditures on R&D and marketing
.", "label": "negative" }
```

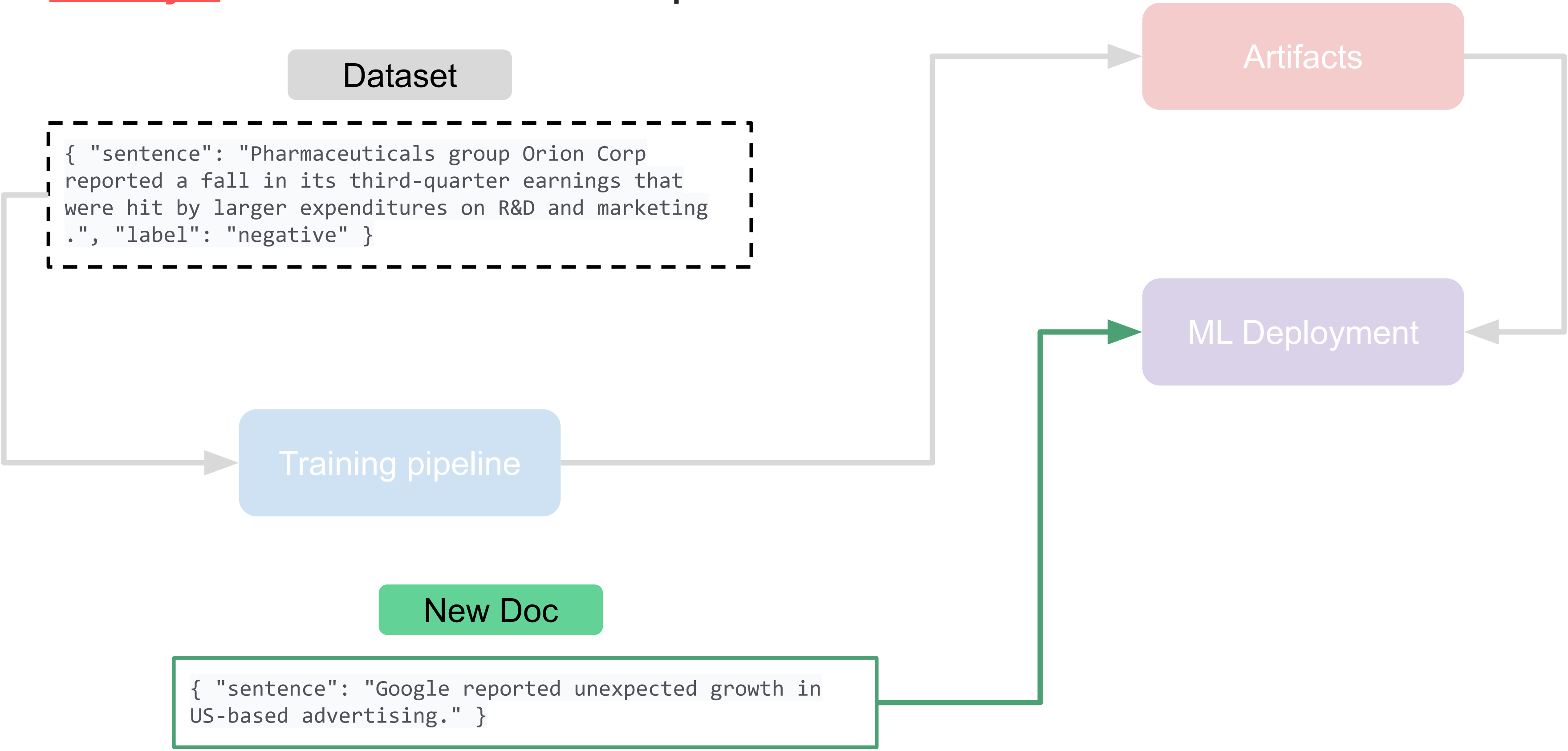
METAFLOW

Training pipeline

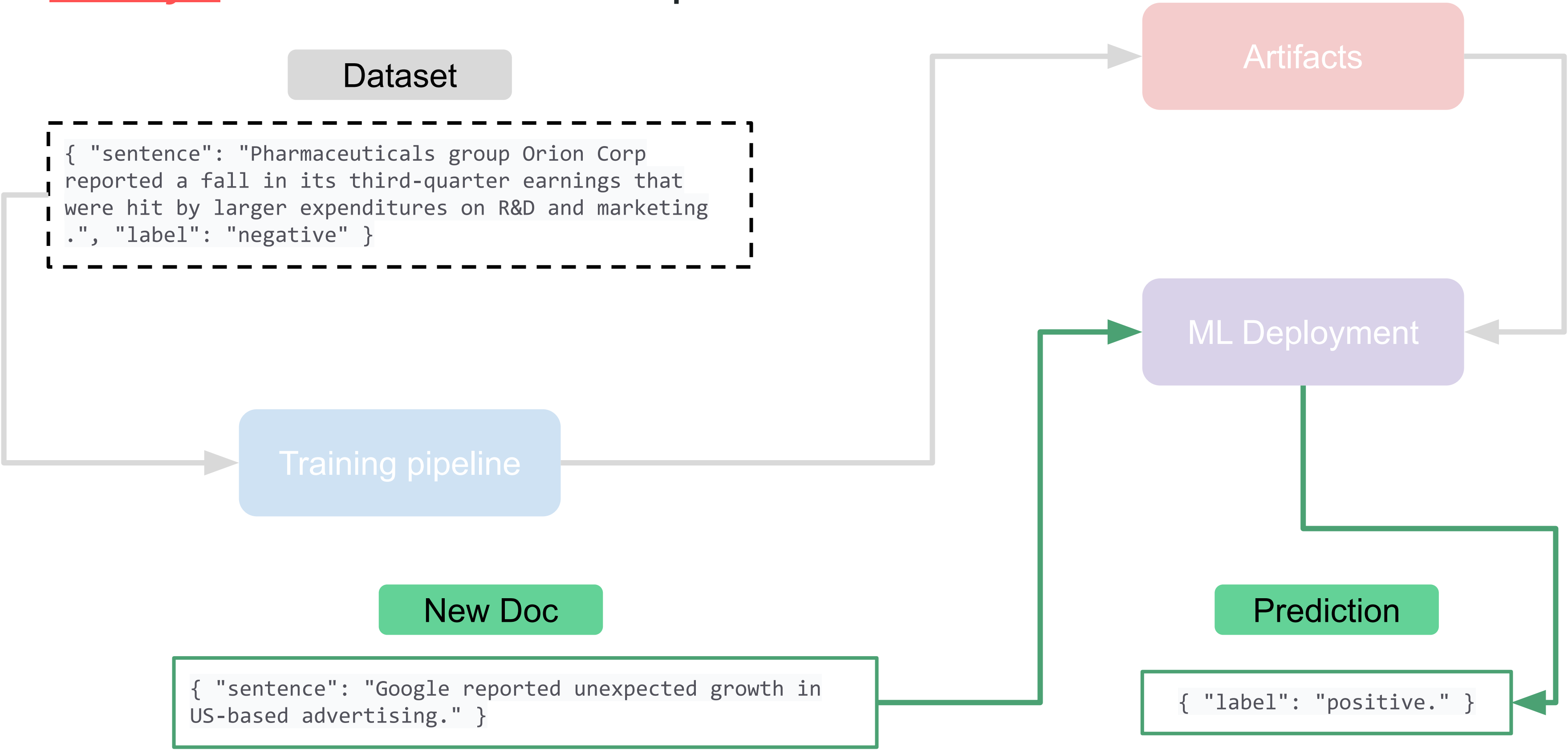
MLSys: A Finance Example



MLSys: A Finance Example



MLSys: A Finance Example



ML in the real-world

Do I really need ML?

While we will discuss ML projects from now on, in the real world you ALWAYS need to ask yourself a question first: is this project a good fit for machine learning?

Signs your project may not be a good fit for ML include:

1. Simpler solutions can do the trick.
2. There is no data (or no practical way to collect it).
3. One single prediction error can cause devastating consequences.
4. It is impossible to reliably measure the performance of the system.

Welcome to the jungle

If your work needs to have an impact, it needs to RUN OUTSIDE YOUR LAPTOP.

Welcome to the jungle

If your work needs to have an impact, it needs to RUN OUTSIDE YOUR LAPTOP:

1. Your code can be **inspected, modified, understood** by others, typically your technical colleagues: you need to write clean, modular, testable code and make your pipeline fully reproducible.

Welcome to the jungle

If your work needs to have an impact, it needs to RUN OUTSIDE YOUR LAPTOP:

1. Your code can be **inspected, modified, understood** by others, typically your technical colleagues: you need to write clean, modular, testable code and make your pipeline fully reproducible.
2. Your model can be **trusted** by others, typically, other stakeholders, who may or may not be technical folks: you need to “make sure” the model behaved as designed before pushing it in front of end-users.

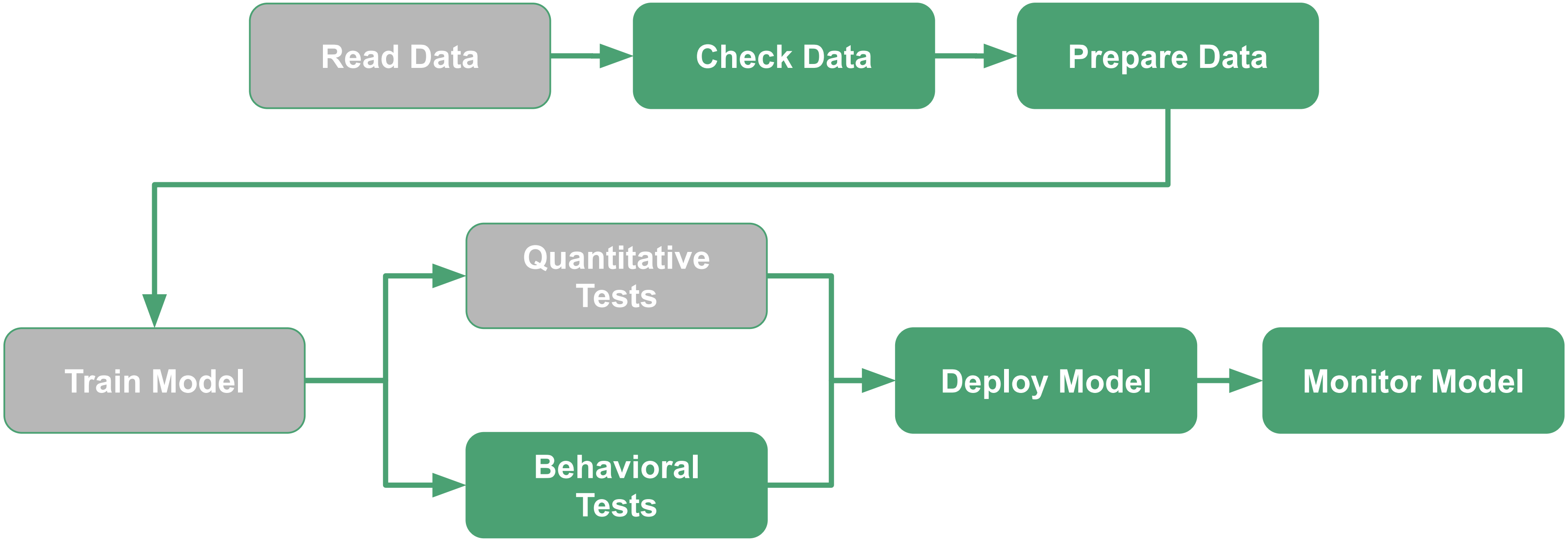
Welcome to the jungle

If your work needs to have an impact, it needs to RUN OUTSIDE YOUR LAPTOP:

1. Your code can be **inspected, modified, understood** by others, typically your technical colleagues: you need to write clean, modular, testable code and make your pipeline fully reproducible.
2. Your model can be **trusted** by others, typically, other stakeholders, who may or may not be technical folks: you need to “make sure” the model behaved as designed before pushing it in front of end-users.
3. Predictions can be **consumed** by others, typically anybody with an internet connection: you need to expose your model as an endpoint which returns predictions when supplied with the appropriate parameters.

School vs Real World

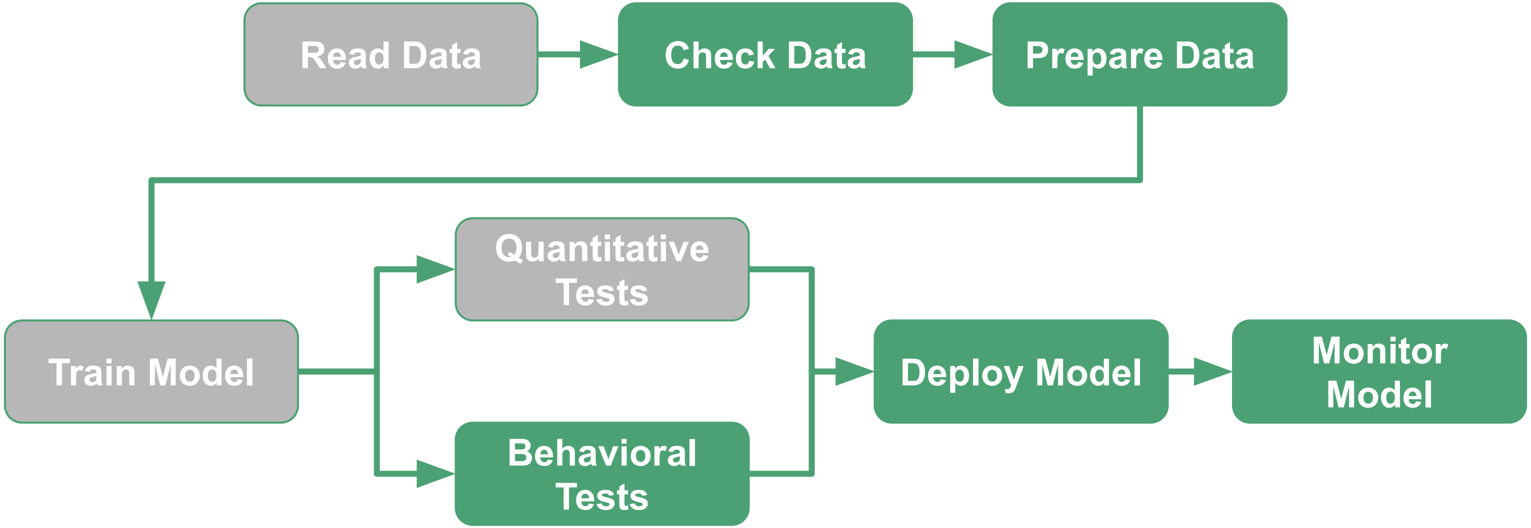
School vs **Real World**



Structuring your project

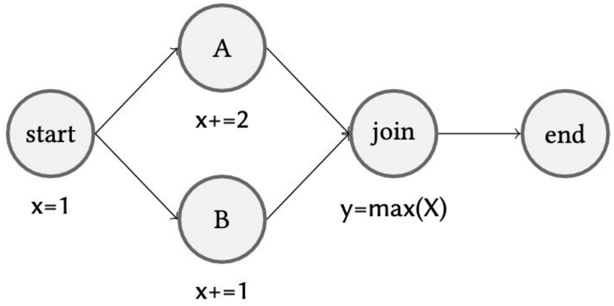
Everything is a **DAG** (Directed Acyclic Graph)

- A ML project is “just” a sequence of steps:
  - You should **not** execute a step **before all its parent steps are done**;
  - NOTE: some steps can “branch out” in parallel (Q: can you think of something that can be easily parallelized in ML?)



A gentle introduction to Metaflow...

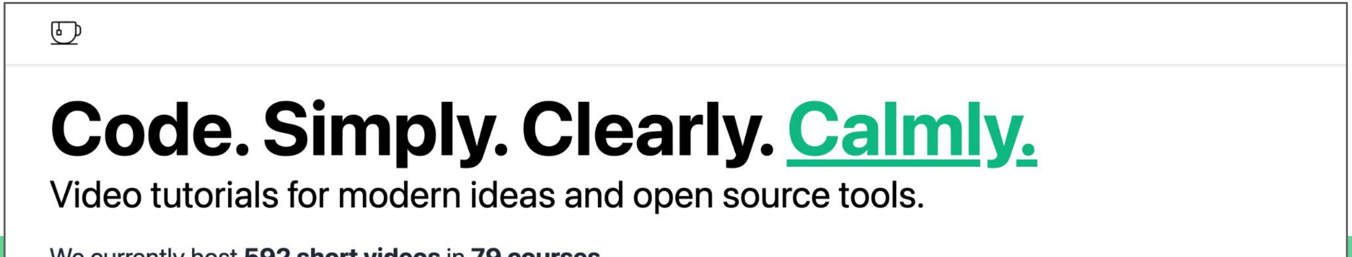
- From DAG to code and vice versa...




```
class ExampleGraph(FlowSpec):
    @step
    def start:
        self.x = 1
        self.next(self.A, self.B)

    @step
    def A(self):
        self.x += 2
        self.next(self.join)
    @step
    def B(self):
        self.x += 1
        self.next(self.join)
    @step
    def join(self, inputs):
        self.y = max(i.x for i in inputs)
        self.next(self.end)
    @step
    def end(self):
        print("y", self.y)
```

Part 0: virtualenv (one more time!)

- ML is done mainly in **Python** today: the web is full of excellent tutorials / courses / books on how to learn Python or be better at it. We focus here only on one core concept: virtual environments.
- Since different projects have different dependencies, we may want to *isolate the environments*: ideally, we should run project A *only with the packages needed by A*, B only with those needed by B etc.
- Practically this is accomplished by using virtual envs, cleanly separated environments to execute specific projects: for an introduction see the calmcode page.



Part 1: Structuring the code

```
def monolith():
    # read the data in and split it
    Xs = []
    Ys = []
    with open('regression_dataset.txt') as f:
        lines = f.readlines()
        for line in lines:
            x, y = line.split('\t')
            Xs.append([float(x)])
            Ys.append(float(y))
    X_train, X_test, y_train, y_test = train_test_split(Xs, Ys, test_size=0.20, random_state=42)
    print(len(X_train), len(X_test))
    # train a regression model
    reg = linear_model.LinearRegression()
    reg.fit(X_train, y_train)
    print("Coefficient {}, intercept {}".format(reg.coef_, reg.intercept_))
    # predict unseeen values and evaluate the model
    y_predicted = reg.predict(X_test)
    fig, ax = plt.subplots()
    ax.scatter(y_predicted, y_test, edgecolors=(0, 0, 1))
    ax.plot([min(y_test), max(y_test)], [min(y_test), max(y_test)], 'r--', lw=3)
    ax.set_xlabel('Predicted')
    ax.set_ylabel('Actual')
    plt.savefig('monolith_regression_analysis.png', bbox_inches='tight')
    mse = metrics.mean_squared_error(y_test, y_predicted)
    r2 = metrics.r2_score(y_test, y_predicted)
    print('MSE is {}, R2 score is {}'.format(mse, r2))

    # all done
    print("See you, space cowboys!")

    return
```

Iteration #1: the monolith (check the repo!)

- All the code is in one main script

PROs

- Fast to write

CONs

- Hard to understand (no logical separation between steps)
- Nothing can be re-used
- Hard to test

Part 1: Structuring the code

```python
def composable_script(file_name: str, test_size: float=0.20):
    # all done
    print("Starting up at {}".format(datetime.utcnow()))
    # read the data into a tuple
    dataset = load_data(file_name)
    # check data quality
    is_data_valid = check_dataset(dataset)
    # split the data
    splits = prepare_train_and_test_dataset(dataset, test_size=test_size)
    # train the model
    regression = train_model(splits, is_debug=True)
    # evaluate model
    model_metrics = evaluate_model(regression.model, splits, with_plot=True)
    # all done
    print("All done at {}!\n See you, space cowboys!".format(datetime.utcnow()))

    return


if __name__ == "__main__":
    # TODO: we can move this to read from a command line option, for example
    FILE_NAME = 'regression_dataset.txt'
    TEST_SIZE = 0.20
    composable_script(FILE_NAME, TEST_SIZE)
```

Iteration #2: breaking down the monolith (check the repo!)

- Tasks are now in separate functions

PROs

- More readable
- Easy to change, test, re-use

CONs

- No versioning
- No replayability
- Hard to scale task selectively

Part 1: Structuring the code

```python
class SampleRegressionFlow(FlowSpec):
    """
    SampleRegressionFlow is a minimal DAG showcasing reading data from a file
    and training a model successfully.
    """

    # if a static file is part of the flow, it can be called in any downstream process, gets versioned etc.
    # https://docs.metaflow.org/metaflow/data#data-in-local-files
    DATA_FILE = IncludeFile(
        'dataset',
        help='Text file with the dataset',
        is_text=True,
        default='regression_dataset.txt')

    TEST_SPLIT = Parameter(
        name='test_split',
        help='Determining the split of the dataset for testing',
        default=0.20
    )

    @step
    def start(self):
        """
        Start up and print out some info to make sure everything is ok metaflow-side
        """
        print("Starting up at {}".format(datetime.utcnow()))
        # debug printing - this is from https://docs.metaflow.org/metaflow/tagging
        # to show how information about the current run can be accessed programmatically
        print("flow name: %s" % current.flow_name)
        print("run id: %s" % current.run_id)
        print("username: %s" % current.username)
        self.next(self.load_data)
```

Iteration #3: Metaflow (check the repo!)

- Tasks are now in a DAG

PROs

- Fully modular
- Scale selectively per task
- All versioned and replayable

CONs

- Additional complexity

Metaflow as a shared lexicon

1. **Flow:** the DAG describing the pipeline itself.
2. **Run:** each time a DAG is executed, it is a new *run*. Runs are isolated and namespaced, e.g. runs tagged as **user:jacopo** vs **user:ethan** may be the same flow, but executed by different people.
3. **Step:** a node of the DAG.
4. **Task**: an execution of a step, isolated and self-contained.
5. **Artifact:** any data / model / state produced by a run, and versioned in the metadata store (e.g. myFlow/12/training/dataset).
6. **Client API:** Python based interactive mode, in which you can inspect metadata and artifacts of all runs for debugging and visualization purposes.

Metaflow projects as (special) Python classes - I

```python
class SampleRegressionFlow(FlowSpec):
    """
    SampleRegressionFlow is a minimal DAG showcasing reading data from a file
    and training a model successfully.
    """

    # if a static file is part of the flow,
    # it can be called in any downstream process,
    # gets versioned etc.
    # https://docs.metaflow.org/metaflow/data#data-in-local-files
    DATA_FILE = IncludeFile(
        'dataset',
        help='Text file with the dataset',
        is_text=True,
        default='regression_dataset.txt')

    TEST_SPLIT = Parameter(
        name='test_split',
        help='Determining the split of the dataset for testing',
        default=0.20
    )
```

A project class inheriting from FlowSpec

OPTIONAL: Parameters to configure the flow, Files as input

Metaflow projects as (special) Python classes - II

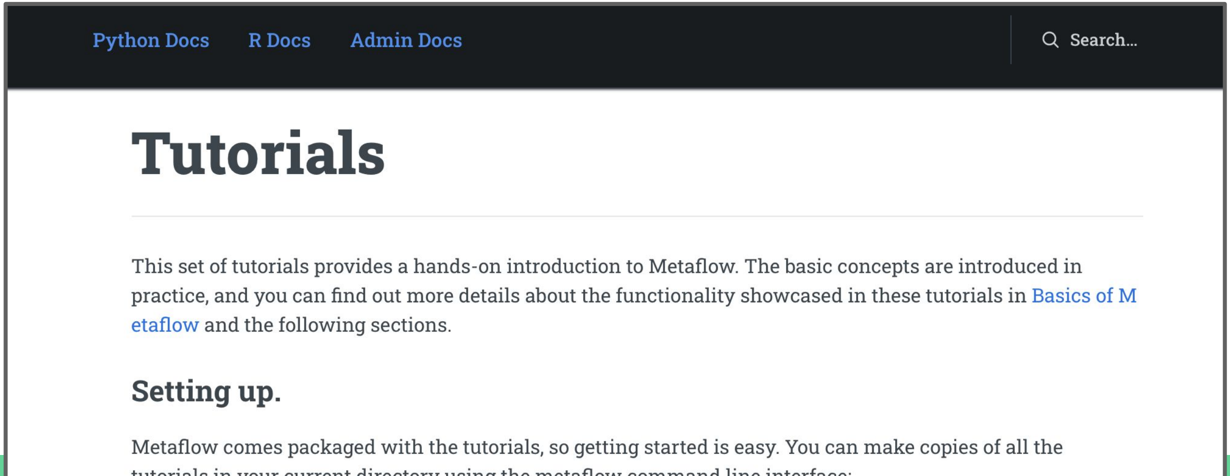
```python
    )

    @step
    def start(self):
        """
        Start up and print out some info to make sure everything is ok metaflow-side
        """
        print("Starting up at {}".format(datetime.utcnow()))
        # debug printing - this is from https://docs.metaflow.org/metaflow/tagging
        # to show how information about the current run can be accessed programmatically
        print("flow name: %s" % current.flow_name)
        print("run id: %s" % current.run_id)
        print("username: %s" % current.username)
        self.next(self.load_data)

    @step
    def load_data(self):
        """
        Read the data in from the static file
        """
        from io import StringIO

        raw_data = StringIO(self.DATA_FILE).readlines()
        print("Total of {} rows in the dataset!".format(len(raw_data)))
        self.dataset = [[float(_) for _ in d.strip().split('\t')] for d in raw_data]
        print("Raw data: {}, cleaned data: {}".format(raw_data[0].strip(), self.dataset[0]))
        self.Xs = [[_[0]] for _ in self.dataset]
        self.Ys = [_[1] for _ in self.dataset]
        # go to the next step
        self.next(self.check_dataset)
```

Functions decorated with @steps: each function is a node in the DAG

Each function lists its descendant(s) through the next command.

Metaflow components

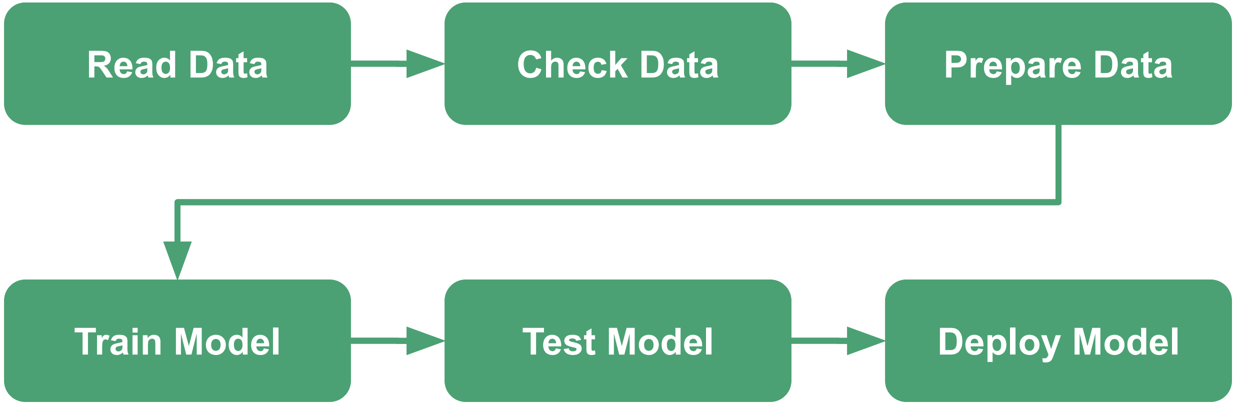
1. **Dag definition:** what are we doing? Steps, dependencies, parallelization etc.
2. **Metastore:** where do we store stuff? Variables, states, meta-data etc.
3. **Computational layer:** what is executing the computation? Resources, cloud tools etc.



Metaflow in 4 principles

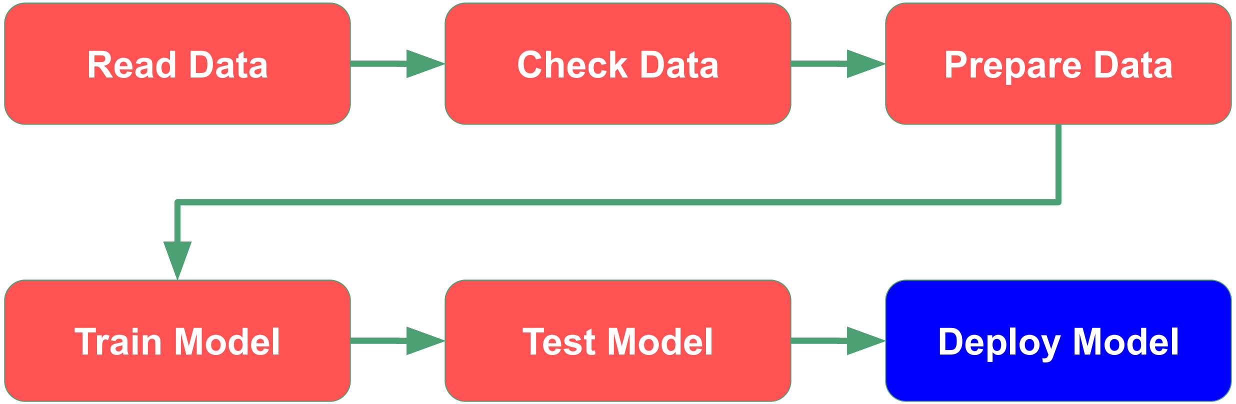
#1: ML projects are a DAG

Tasks depends only on a subset of other tasks: parallelization is possible, and retry can be smart in case of failure!



FRE 7773 Bonus Point

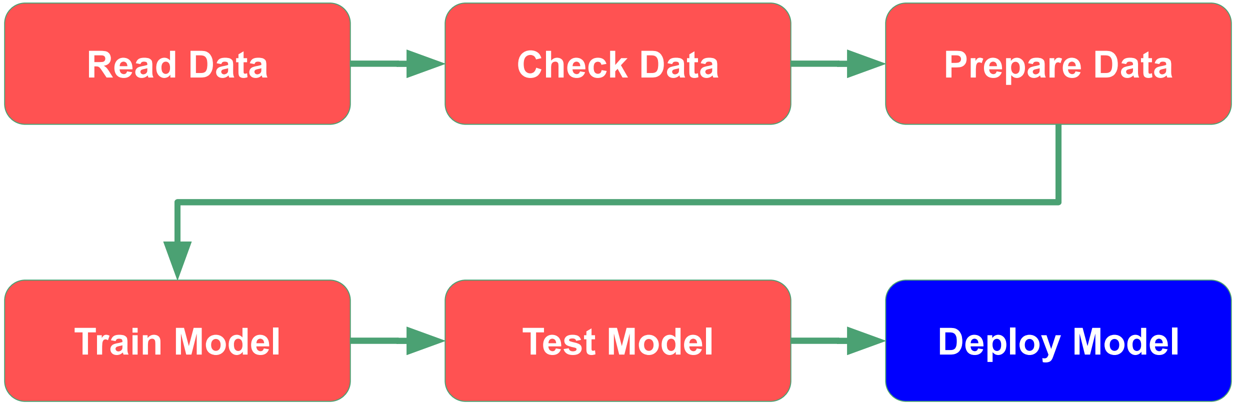
We distinguish between two phases of our ML project: a training phase (load data, data checks, training and testing model...) and a serving phase (expose the model prediction to other users).



FRE 7773 Bonus Point

In this class (and also when developing new projects in the industry), we have:

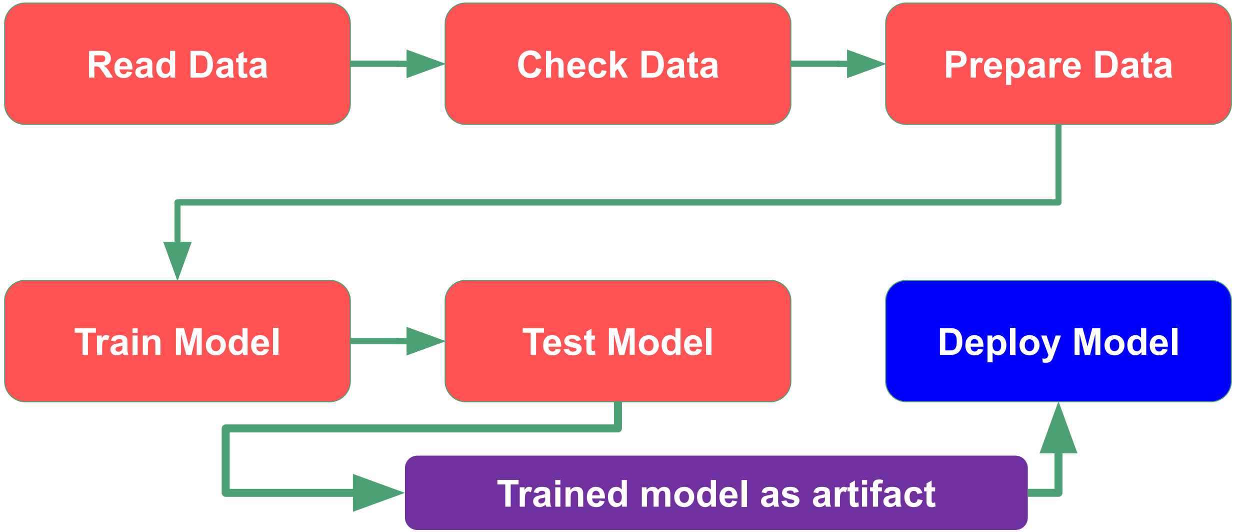
- training phase: done locally (in Metaflow)
- serving phase: done in the cloud (in AWS)



FRE 7773 Bonus Point

In this class (and also when developing new projects in the industry), we have:

- training phase: done locally (in Metaflow) and **produces a model artifact**
- serving phase: done in the cloud (in AWS)



Metaflow in 4 principles

#2: Data and states are part of ML pipelines (versioning, replayability)

```
@step
def load_data(self):
    """
    Read the data in from the static file
    """
    from io import StringIO

    raw_data = StringIO(self.DATA_FILE).readlines()
    print("Total of {} rows in the dataset!".format(len(raw_data)))
    self.dataset = [[float(_) for _ in d.strip().split('\t')] for d in raw_data]
    print("Raw data: {}, cleaned data: {}".format(raw_data[0].strip(), self.dataset[0]))
    self.Xs = [[_[0]] for _ in self.dataset]
    self.Ys =  [_[1] for _ in self.dataset]
    # go to the next step
    self.next(self.check_dataset)
```

The raw dataset is saved!

The X,Y dataset is saved!

Metaflow in 4 principles

#2: Data and states can always be inspected (check the notebook)

Get artifacts from latest successful run

In [4]:
```
def get_latest_successful_run(flow_name: str):
    "Gets the latest successfull run."
    for r in Flow(flow_name).runs():
        if r.successful:
            return r
```

In [5]:
```
latest_run = get_latest_successful_run(FLOW_NAME)
latest_model = latest_run.data.model
latest_dataset = latest_run.data.dataset
```

Verify we can inspect the dataset...

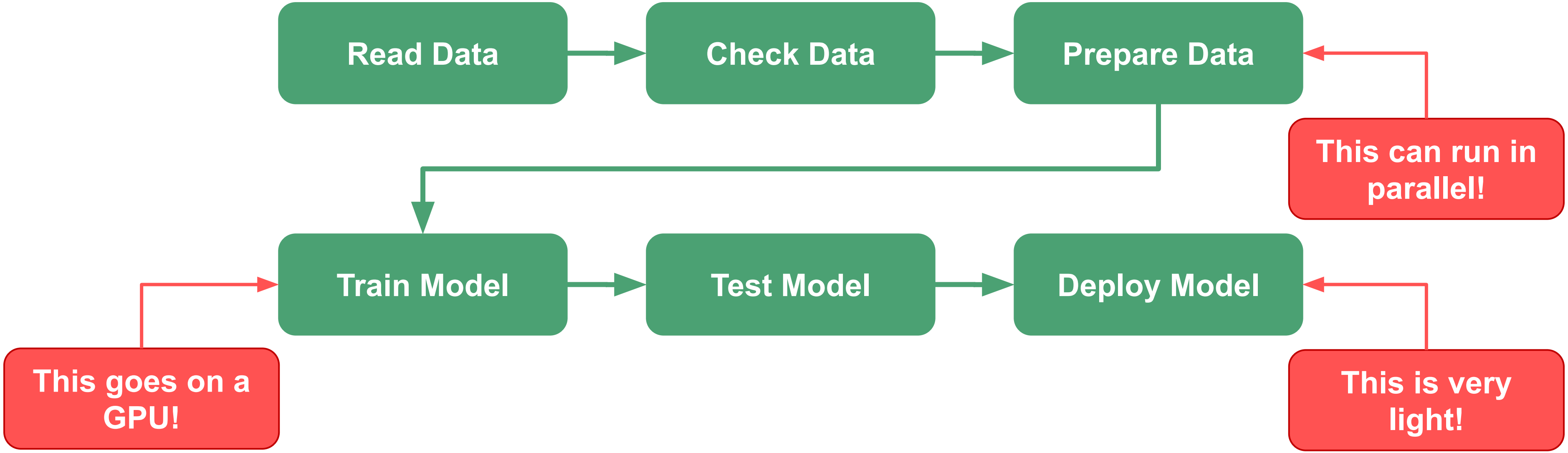
In [6]:
```
latest_dataset[:10]
```

Out[6]:
```
[[-1.7587394864231143, -32.770386047959725],
 [1.0318445394686349, 3.5045910648442344],
 [-0.48760622407249354, -17.930307666159294],
 [0.18645431476942764, -3.990201236512462],
 [0.725766623898692, 13.105264342363048],
 [0.9725544496267299, 33.7844061138283],
 [0.6453759495851475, -6.568374494070948],
```

Metaflow in 4 principles

#3: One computing size does not fit all

You can define computing resources (and packages) *per task*, switching between local and cloud computing only when necessary.



Metaflow in 4 principles

#4: Everything is cool when you're part of a team

Multiple users can run the same flow together, and then the team can analyze the artifacts produced independently by all runs.