

# Efficient Deep Learning Systems

Experiment management & ML code testing

Max Ryabinin

# Teaser

ML  
Code

=

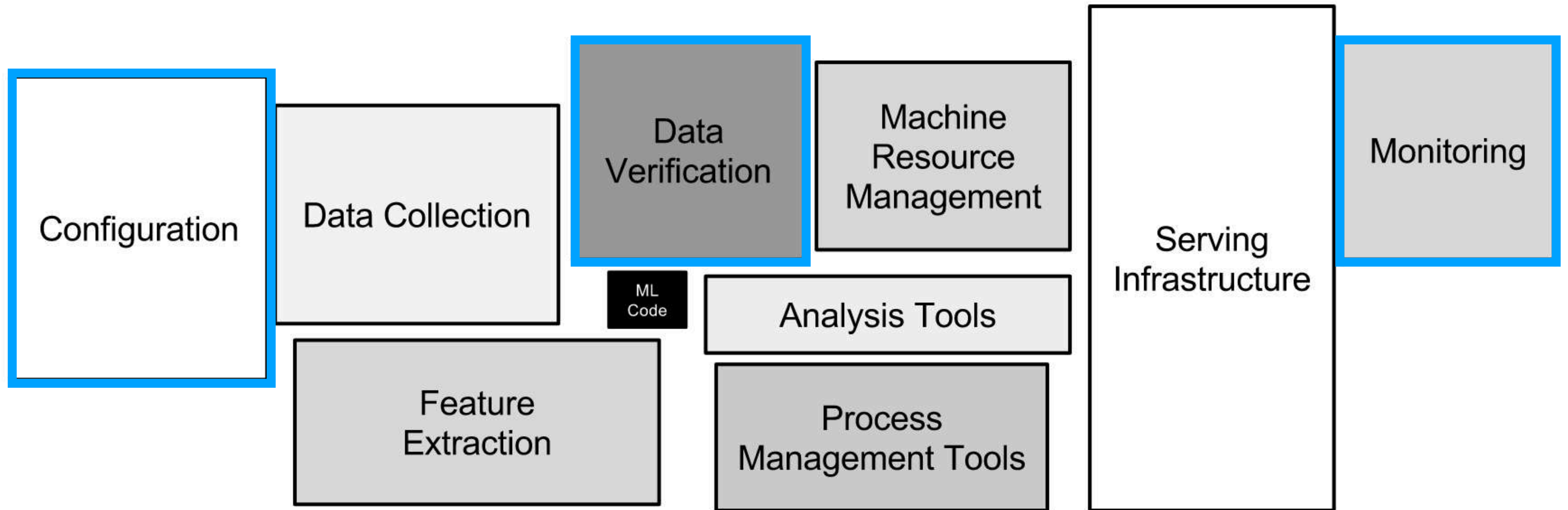
**Modeling, experiments**

**Offline quality evaluation**

**Data preprocessing**

**Code/model efficiency**

# Teaser



# Plan for today

- How (and why) to track your DL experiments
- Making your environment reproducible
- Versioning your data and models along with the code
- Flexible configuration of Python code
- Testing in general and for ML purposes

# Tracking experiments: motivation

- Usually, training a model once is not enough:
  - The data gets updated
  - Hyperparameters need tuning
  - We want to modify the training code for better quality
- For all these cases, we need a way to keep track of our experiments
- Even more important in a collaborative setup

# What to track

- Obviously, we want a table with run IDs and final metrics
- What else?
  - Plots with per-step/per-second metrics (convergence & performance)
  - Git commit hash for reproducibility (and diff for local changes!)
  - Visualizations of model inputs/outputs
  - Stdout/stderr of your training script (invest time in good logs)
  - In some cases, full info about the environment

# How to track

- There are many tools for that [1,2,3,4,5]
- Range from “just upload the logs” to full-fledged tracking of the entire environment
- Self-hosted versions are available

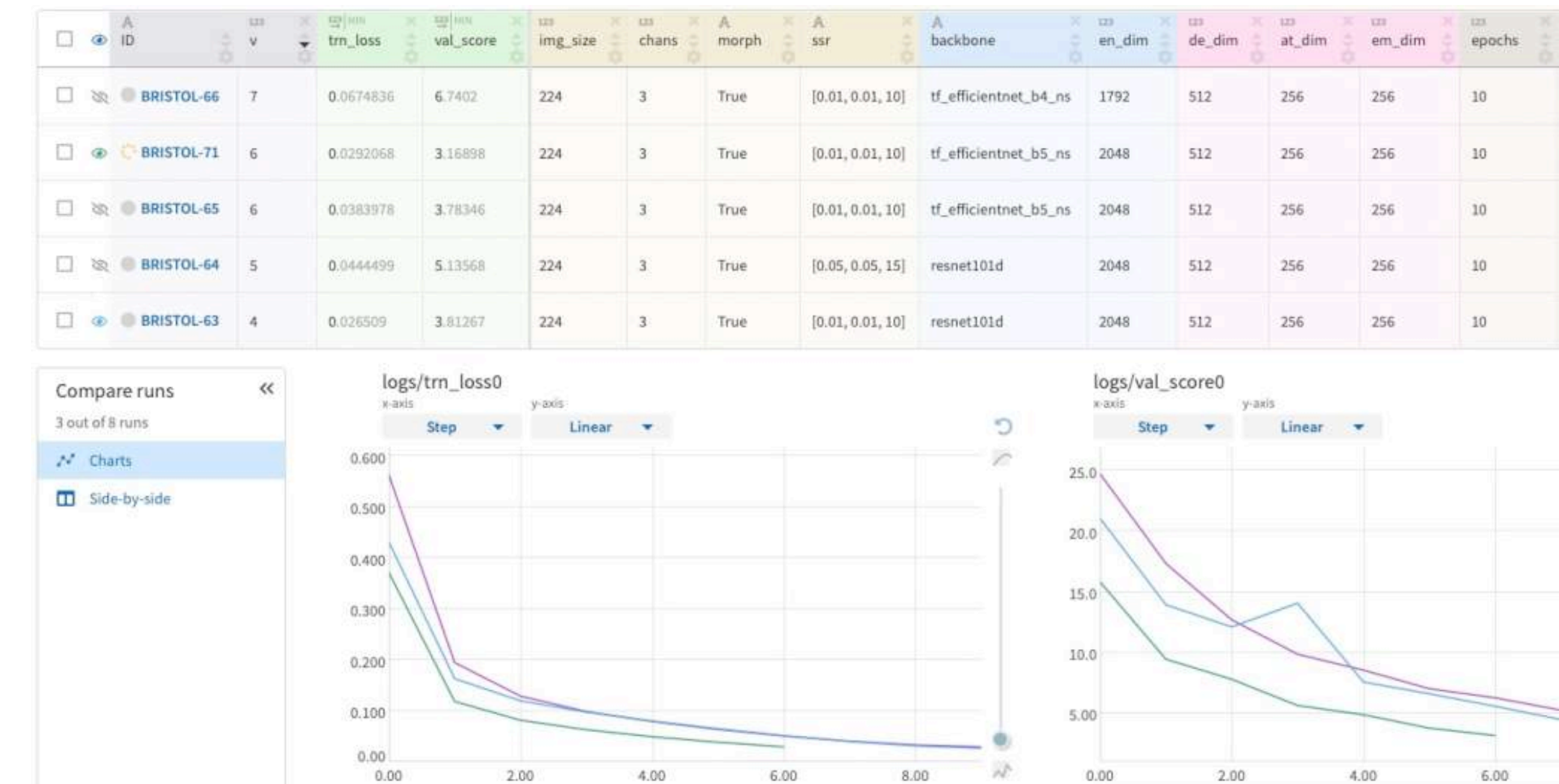


TABLE VIEW		PARALLEL COORDINATES VIEW		SCATTER PLOT MATRIX VIEW	
Trial ID	Show Metrics	model	accuracy_test	lr	train_loss
1b2758da0f912...	<input type="checkbox"/>	ViT-L_32	0.82009	1.9467e-10	0.42093
606b401e6f84...	<input type="checkbox"/>	ViT-B_16	0.84609	1.9467e-10	1.0754
60c54527f120...	<input type="checkbox"/>	ViT-L_16	0.85066	1.9467e-10	0.65612
6c5951c20e95...	<input type="checkbox"/>	ViT-B_32	0.81788	1.9467e-10	1.3841
85c586a058ca...	<input type="checkbox"/>	ViT-B_16	0.84621	1.9467e-10	0.45114
a2316e5f8b991...	<input type="checkbox"/>	ViT-L_16	0.85050	1.9467e-10	0.40394
c32186203a97...	<input type="checkbox"/>	ViT-B_32	0.81790	1.9467e-10	1.4536
f853f3481c8db...	<input type="checkbox"/>	ViT-L_32	0.81780	1.9467e-10	0.81554

[1] <https://www.wandb.com/>

[2] <https://www.comet.ml/>

[3] <https://neptune.ai/>

[4] <https://tensorboard.dev/>

[5] <https://clear.ml/>



# Dependency versioning

- Large Python projects involve numerous dependencies
- Over time, this presents multiple issues:

Summary: State-of-the-art Machine Learning for JAX, PyTorch and TensorFlow

Latest version: 4.48.1

Required dependencies: [filelock](#) | [huggingface-hub](#) | [numpy](#) | [packaging](#) | [pyyaml](#) | [regex](#) | [requests](#) | [safetensors](#) | [tokenizers](#) | [tqdm](#)

Optional dependencies: [accelerate](#) | [av](#) | [beautifulsoup4](#) | [blobfile](#) | [codecarbon](#) | [cookiecutter](#) | [datasets](#) | [deepspeed](#) | [diffusers](#) | [dill](#) | [evaluate](#) | [faiss-cpu](#) | [fastapi](#) | [filelock](#) | [flax](#) | [ftfy](#) | [fugashi](#) | [gitpython](#) | [huggingface-hub](#) | [importlib-metadata](#) | [ipadic](#) | [isort](#) | [jax](#) | [jaxlib](#) | [kenlm](#) | [keras](#) | [keras-nlp](#) | [libbst](#) | [librosa](#) | [natten](#) | [nltk](#) | [numpy](#) | [onnxconverter-common](#) | [onnxruntime](#) | [onnxruntime-tools](#) | [opencv-python](#) | [optax](#) | [optimum-benchmark](#) | [optuna](#) | [packaging](#) | [parameterized](#) | [phonemizer](#) | [pillow](#) | [protobuf](#) | [psutil](#) | [pyctcdecode](#) | [pydantic](#) | [pytest](#) | [pytest-asyncio](#) | [pytest-rich](#) | [pytest-timeout](#) | [pytest-xdist](#) | [ray](#) | [regex](#) | [requests](#) | [rhoknp](#) | [rich](#) | [rjieba](#) | [rouge-score](#) | [ruff](#) | [sacrebleu](#) | [sacremoses](#) | [sagemaker](#) | [scikit-learn](#) | [scipy](#) | [sentencepiece](#) | [sigopt](#) | [starlette](#) | [sudachidict-core](#) | [sudachipy](#) | [tensorboard](#) | [tensorflow](#) | [tensorflow-cpu](#) | [tensorflow-probability](#) | [tensorflow-text](#) | [tf2onnx](#) | [tiktoken](#) | [timeout-decorator](#) | [timm](#) | [tokenizers](#) | [torch](#) | [torchaudio](#) | [torchvision](#) | [tqdm](#) | [unicid](#) | [unicid-lite](#) | [urllib3](#) | [uvicorn](#)



# Dependency versioning

- Large Python projects involve numerous dependencies
- Over time, this presents multiple issues:
  - Version upgrades lead to breaking changes
  - Different libraries might have different transitive dependencies
  - Crucial dependencies are simply not specified

Closed Breaking change in v4.48.0 and Python 3.9 #35639  
#35666

## Reproduction

v4.48.0 introduced a breaking change that happens on import of TextIteratorStreamer

Steps to reproduce

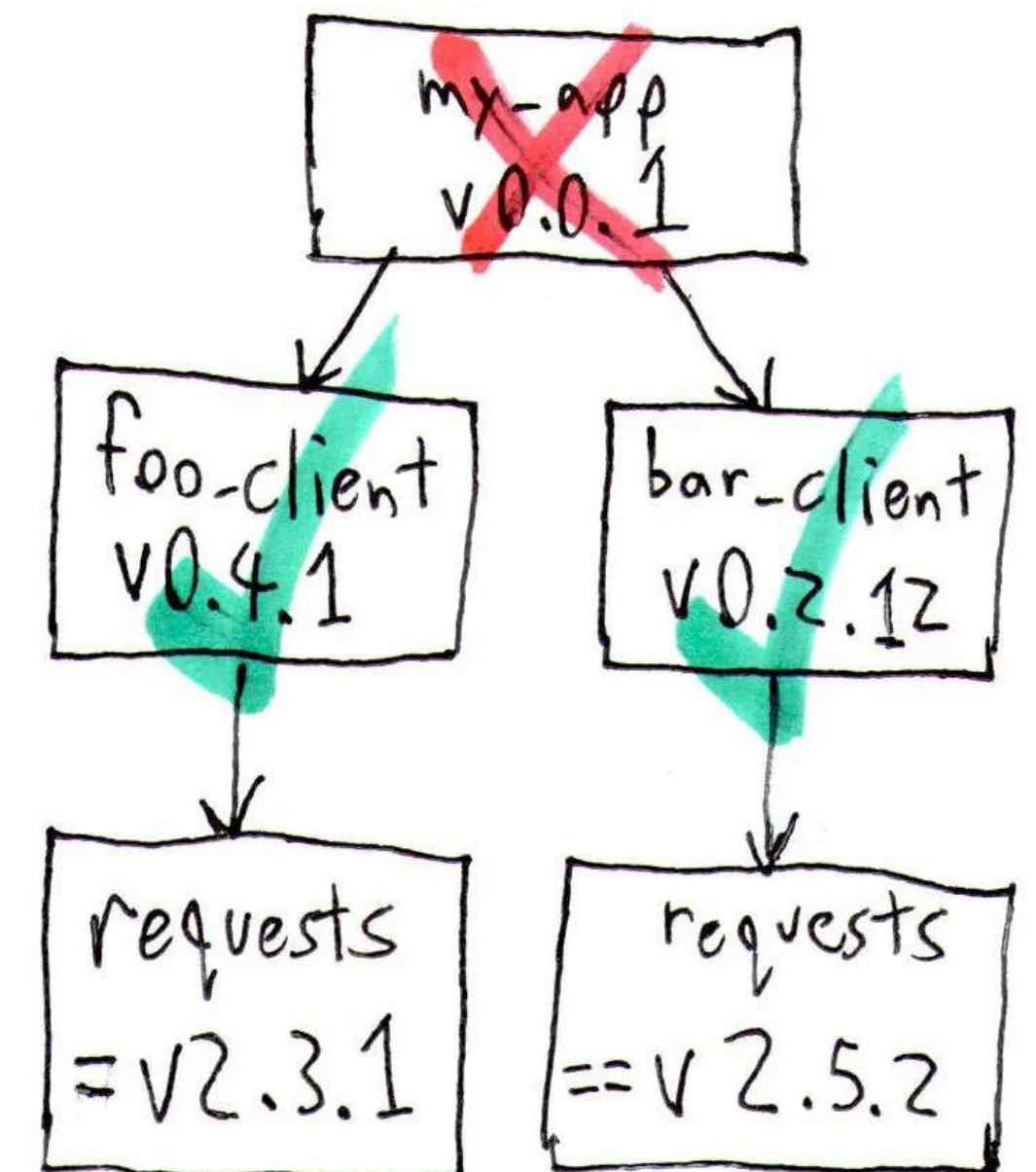
1. Install Python 3.9
2. Install Transformers >= v4.48.0

Run following:

```
from transformers import TextIteratorStreamer
```

Receive the following stack

```
Traceback (most recent call first):
  File "/tmp/test/lib/python3.9/site-packages/transformers/streamers/text_iterator_streamer.py", line 10, in <module>
    from transformers import TextIteratorStreamer
```



# Dependency versioning tools

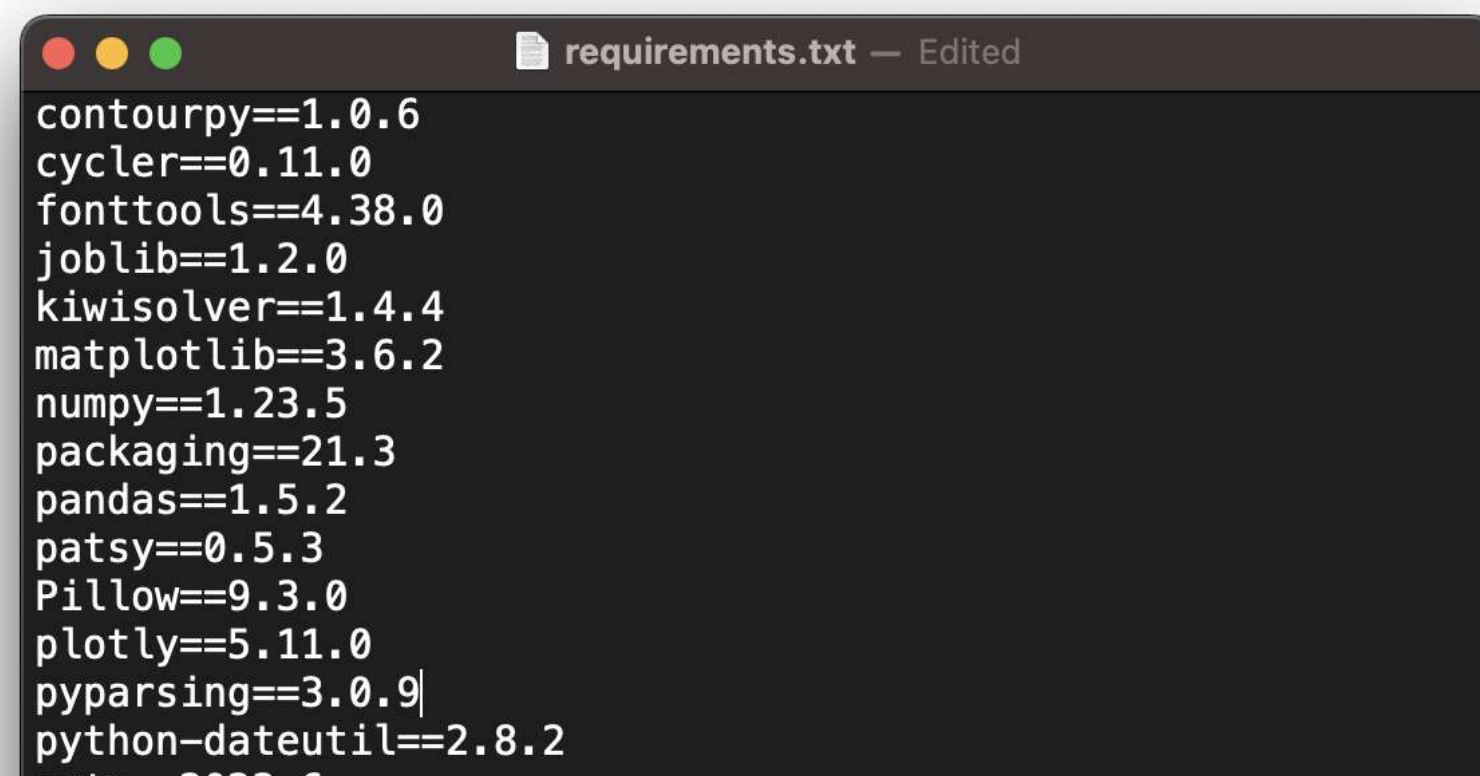
Several solutions for this exist:

- The simplest one: `pip freeze>requirements.txt`

Works, but hard to upgrade + your project will be tough to depend on

- Better: PEP 508 version constraints

Upgrades are still hard, and transitive dependencies are not tracked

A screenshot of a code editor window titled 'requirements.txt — Edited'. The window contains a list of package names followed by their installed versions, separated by double equals signs. The packages listed are: contourpy==1.0.6, cycler==0.11.0, fonttools==4.38.0, joblib==1.2.0, kiwisolver==1.4.4, matplotlib==3.6.2, numpy==1.23.5, packaging==21.3, pandas==1.5.2, patsy==0.5.3, Pillow==9.3.0, plotly==5.11.0, pyparsing==3.0.9, python-dateutil==2.8.2, and python==3.9.6.

```
contourpy==1.0.6
cycler==0.11.0
fonttools==4.38.0
joblib==1.2.0
kiwisolver==1.4.4
matplotlib==3.6.2
numpy==1.23.5
packaging==21.3
pandas==1.5.2
patsy==0.5.3
Pillow==9.3.0
plotly==5.11.0
pyparsing==3.0.9
python-dateutil==2.8.2
python==3.9.6
```

A list of examples for PEP 508 version constraints. The examples show various ways to specify version ranges, minimum/maximum versions, and platform-specific constraints. The constraints are color-coded: blue for version numbers, green for platform names, and red for comparison operators.

```
SomeProject
SomeProject == 1.3
SomeProject >= 1.2, < 2.0
SomeProject[foo, bar]
SomeProject ~= 1.4.2
SomeProject == 5.4 ; python_version < '3.8'
SomeProject ; sys_platform == 'win32'
requests [security] >= 2.8.1, == 2.8.* ; python_version < "2.7"
```



# Dedicated versioning tools

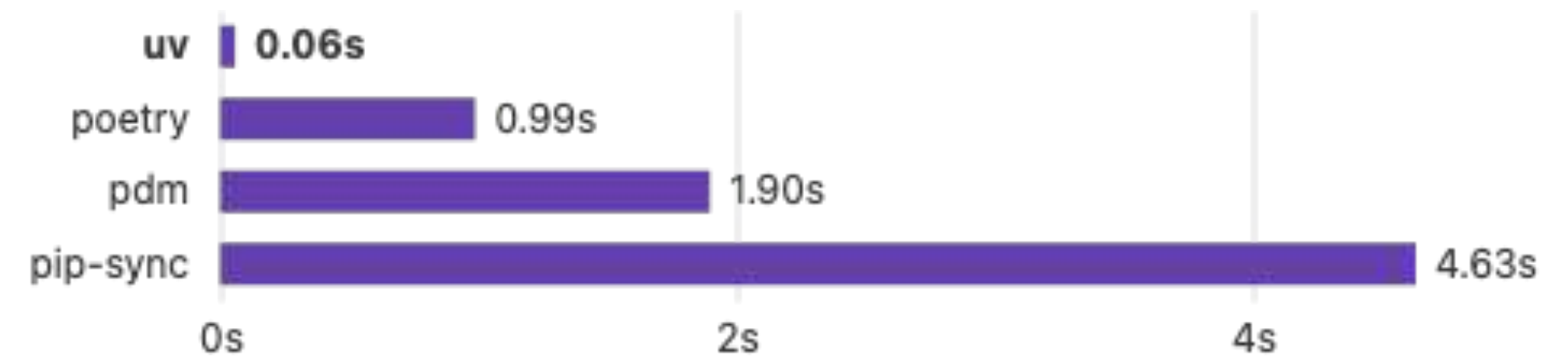
- Poetry introduces a *lockfile* which specifies all permissible dependencies with their versions
- Manages Python versions as well!

[python-poetry.org](https://python-poetry.org)

```
822
823 [metadata.files]
824 astroid = [
825     {file = "astroid-2.9.3-py3-none-an
826     {file = "astroid-2.9.3.tar.gz", ha
827 ]
828 atomicwrites = [
829     {file = "atomicwrites-1.4.0-py2.py
830     {file = "atomicwrites-1.4.0.tar.gz
831 ]
832 attrs = [
833     {file = "attrs-21.4.0-py2.py3-none
834     {file = "attrs-21.4.0.tar.gz", has
835 ]
836 basedmypy = [
837     {file = "basedmypy-1.2.2-py3-none-
838     {file = "basedmypy-1.2.2.tar.gz",
839 ]
840 basedtyping = [] You, 3 days ago
841 beautifulsoup4 = [
842     {file = "beautifulsoup4-4.10.0-py3
843     {file = "beautifulsoup4-4.10.0.tar
844 ]
845 black = [
846     {file = "black-21.12b0-py3-none-an
847     {file = "black-21.12b0.tar.gz", ha
```

# Dedicated versioning tools

- Poetry introduces a *lockfile* which specifies all permissible dependencies with their versions
- Manages Python versions as well!
- uv offers similar functionality while being a much faster replacement of pip



Creating virtual environments (replacing `venv` and `virtualenv`):

- `uv venv` : Create a new virtual environment.

See the documentation on [using environments](#) for details.

Managing packages in an environment (replacing `pip` and `pipdeptree`):

- `uv pip install` : Install packages into the current environment.
- `uv pip show` : Show details about an installed package.
- `uv pip freeze` : List installed packages and their versions.
- `uv pip check` : Check that the current environment has compatible packages.
- `uv pip list` : List installed packages.
- `uv pip uninstall` : Uninstall packages.
- `uv pip tree` : View the dependency tree for the environment.

See the documentation on [managing packages](#) for details.

[python-poetry.org](https://python-poetry.org)

[docs.astral.sh/uv](https://docs.astral.sh/uv)

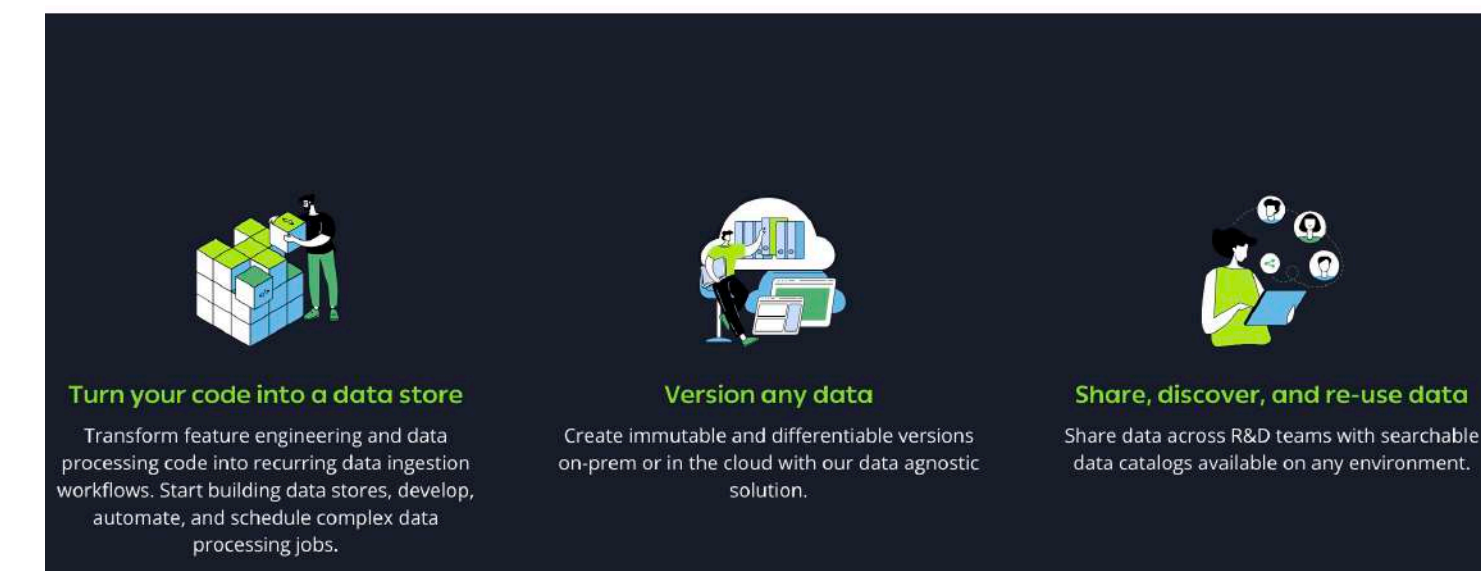
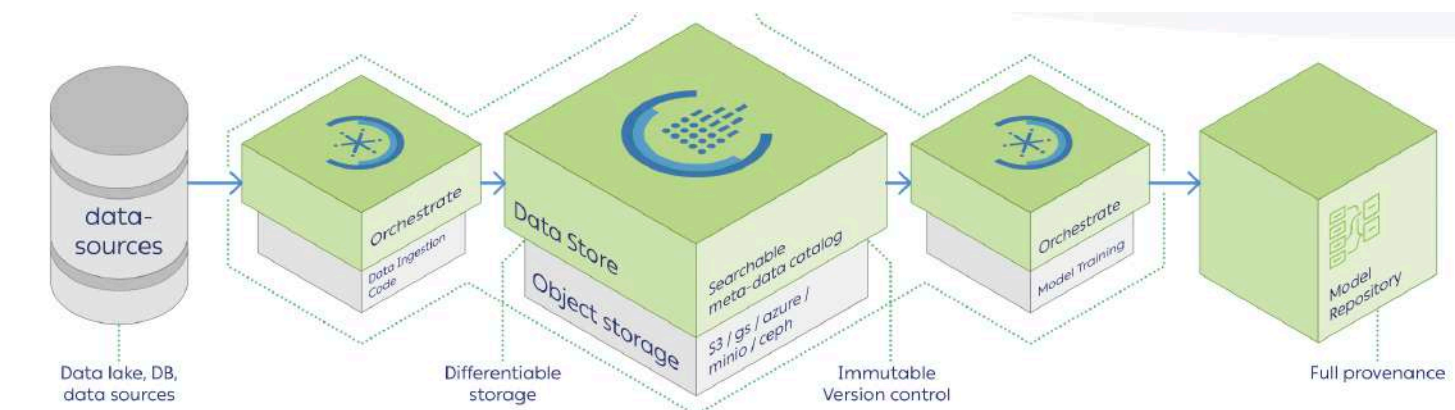
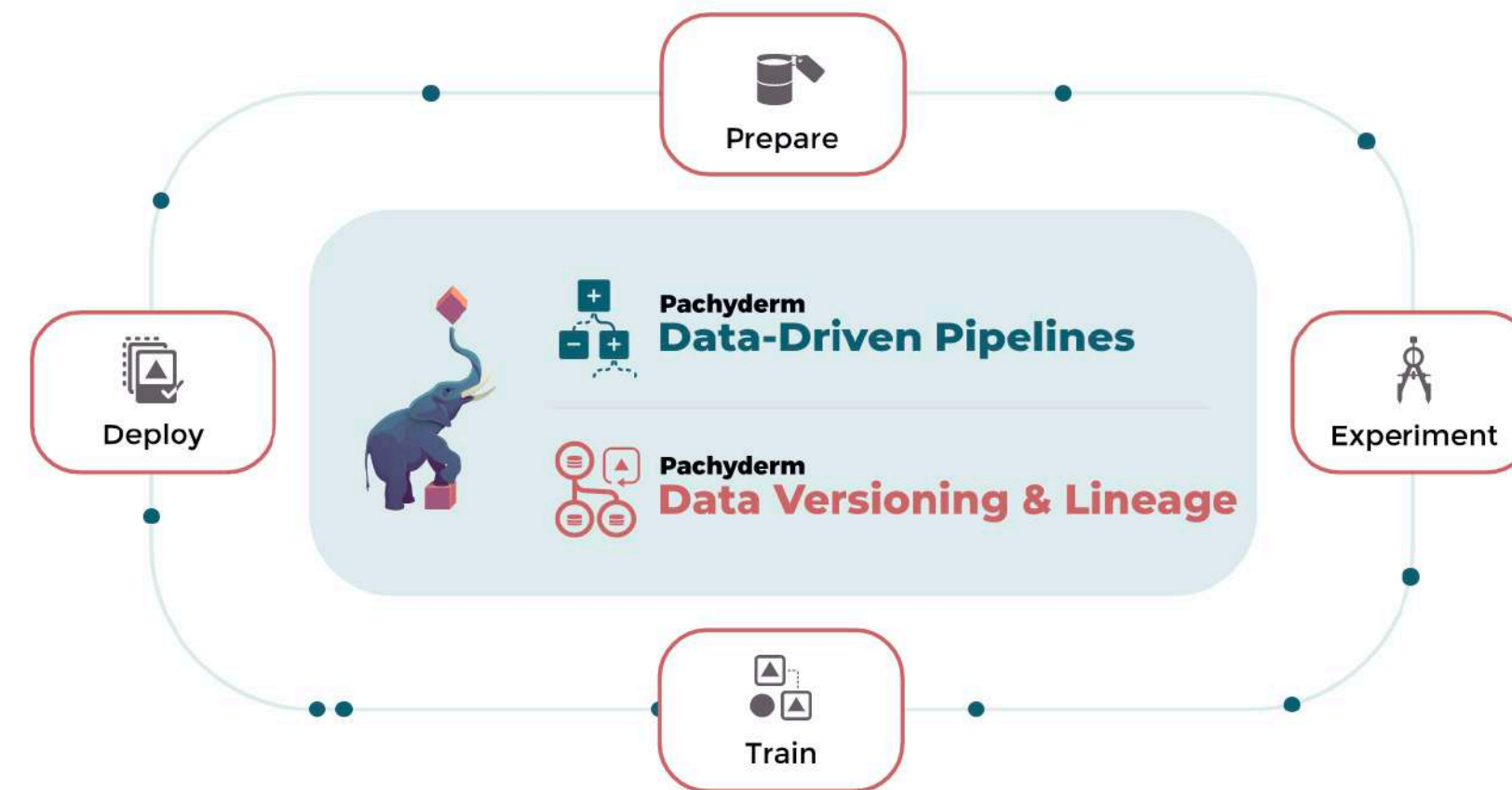
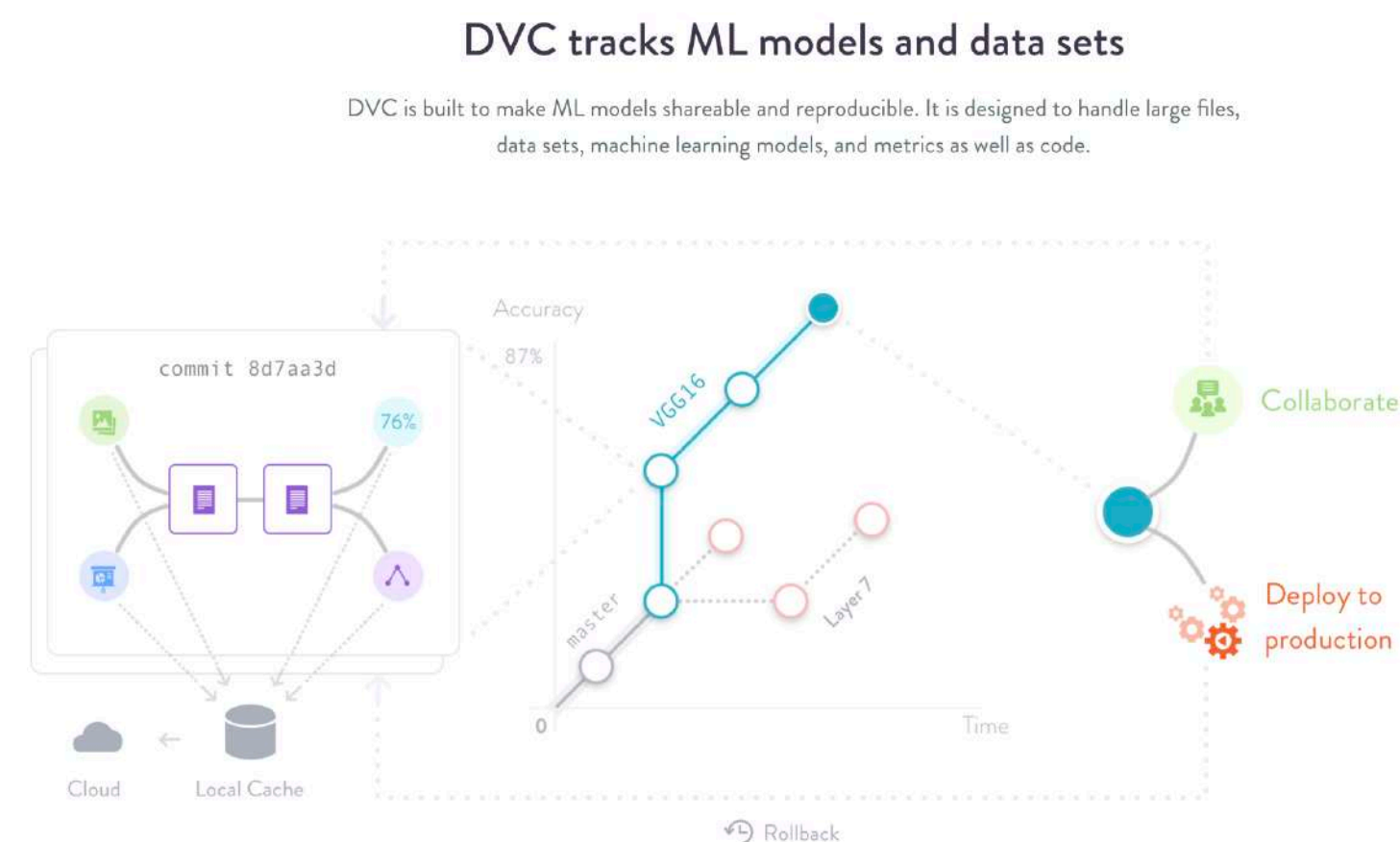
# Data versioning

- Code is not the only component in your system
- Data is a **crucial** dependency, especially in complicated pipelines
- Tracking changes in it is equally important
- Pinning each experiment to its data enhances reproducibility



# Solutions

- Several existing projects allow to integrate artifact versioning into pipelines
- Support external storage, matching with commits, metric comparison
- Possible to rerun specific parts of the pipeline on data/config change



<https://dvc.org/>

<https://www.pachyderm.com/>

<https://clear.ml/>



# Configuration

- As your project grows, the number of “moving parts” increases
  - Infrastructure: API endpoints, data URLs, etc.
  - Model hyperparameters and components
- Changing them manually across the entire repo is not sustainable
- `argparse/click`-based solutions are hard to write and properly version
- Hardcoding values in dedicated Python files is not flexible enough

# Hydra

- One of the most popular solutions for handling configuration
- Uses YAML configs, allows overriding values from the command line
- Simple type checking via Structured Configs
- Grouped configs offer easy switching between groups of presets



## Basic example

Config:

conf/config.yaml

```
db:
  driver: mysql
  user: omry
  pass: secret
```

Application:

my\_app.py

```
import hydra
from omegaconf import DictConfig, OmegaConf

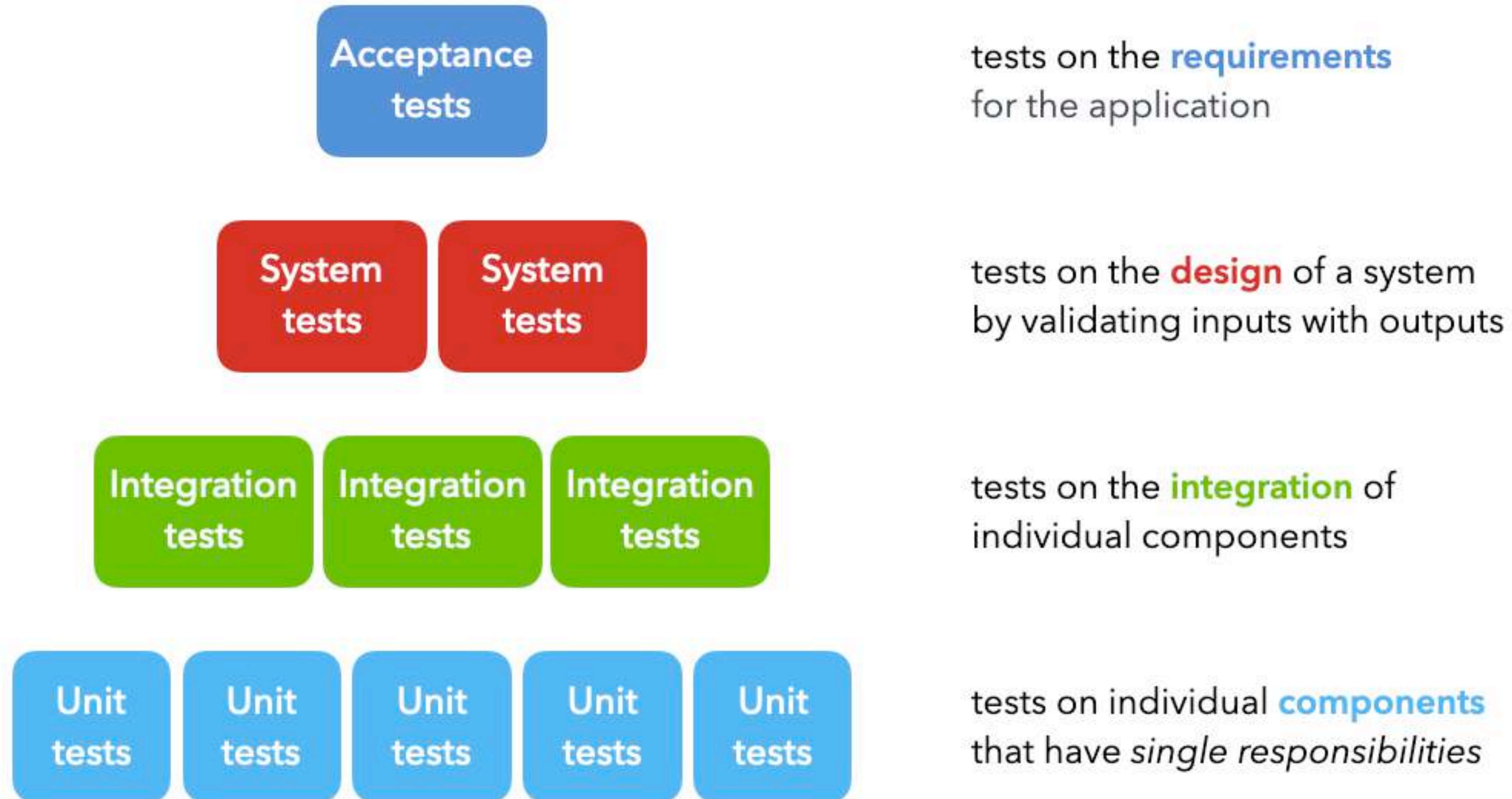
@hydra.main(config_path="conf", config_name="config")
def my_app(cfg : DictConfig) -> None:
    print(OmegaConf.to_yaml(cfg))

if __name__ == "__main__":
    my_app()
```

# Testing

- In general, testing refers to verifying the intended code properties:
  - Not only correctness, but also performance, handling inputs, etc.
- Why should we test our code?
  - It helps avoid the bugs (both now and when refactoring)
  - But it **does not** prevent them! Treat tests like classifiers applied to your code
  - It improves the overall code quality by decoupling
  - Essentially, you get self-documented code for free

# Types of software tests



# Types of software tests

- There are many kinds and typologies, e.g.:
  1. **Unit tests** verify the correctness of a single component
  2. **Integration tests** ensure that modules work together
  3. **End-to-end tests** verify that the entire application is correct
  4. **Stress/load/performance** tests check the speed of code under load
- We'll focus on 1 and 2: they are the easiest to write and cover most cases

<https://www.atlassian.com/continuous-delivery/software-testing/types-of-software-testing>

<https://hackr.io/blog/types-of-software-testing>



# How to test Python code

- Python built-in: unittest
- Quite simple, ready to use
- Cons: has its own syntax, not that flexible

```
import unittest

class TestStringMethods(unittest.TestCase):

    def test_upper(self):
        self.assertEqual('foo'.upper(), 'FOO')

    def test_isupper(self):
        self.assertTrue('FOO'.isupper())
        self.assertFalse('Foo'.isupper())

    def test_split(self):
        s = 'hello world'
        self.assertEqual(s.split(), ['hello', 'world'])
        # check that s.split fails when the separator is not a string
        with self.assertRaises(TypeError):
            s.split(2)

if __name__ == '__main__':
    unittest.main()
```

```
...
-----
Ran 3 tests in 0.000s

OK
```



# How to test Python code

- Python built-in: unittest
  - Quite simple, ready to use
  - Cons: has its own syntax, not that flexible
- Better: pytest
  - Flexible, works with assert statements, has plenty of integrations via plugins



```
...
-----
Ran 3 tests in 0.000s

OK
```

```
# content of test_sample.py
def func(x):
    return x + 1

def test_answer():
    assert func(3) == 5
```

```
$ pytest
===== test session starts =====
platform linux -- Python 3.x.y, pytest-6.x.y, py-1.x.y, pluggy-1.x.y
cachedir: $PYTHON_PREFIX/.pytest_cache
rootdir: $REGENDOC_TMPDIR
collected 1 item

test_sample.py F [100%]

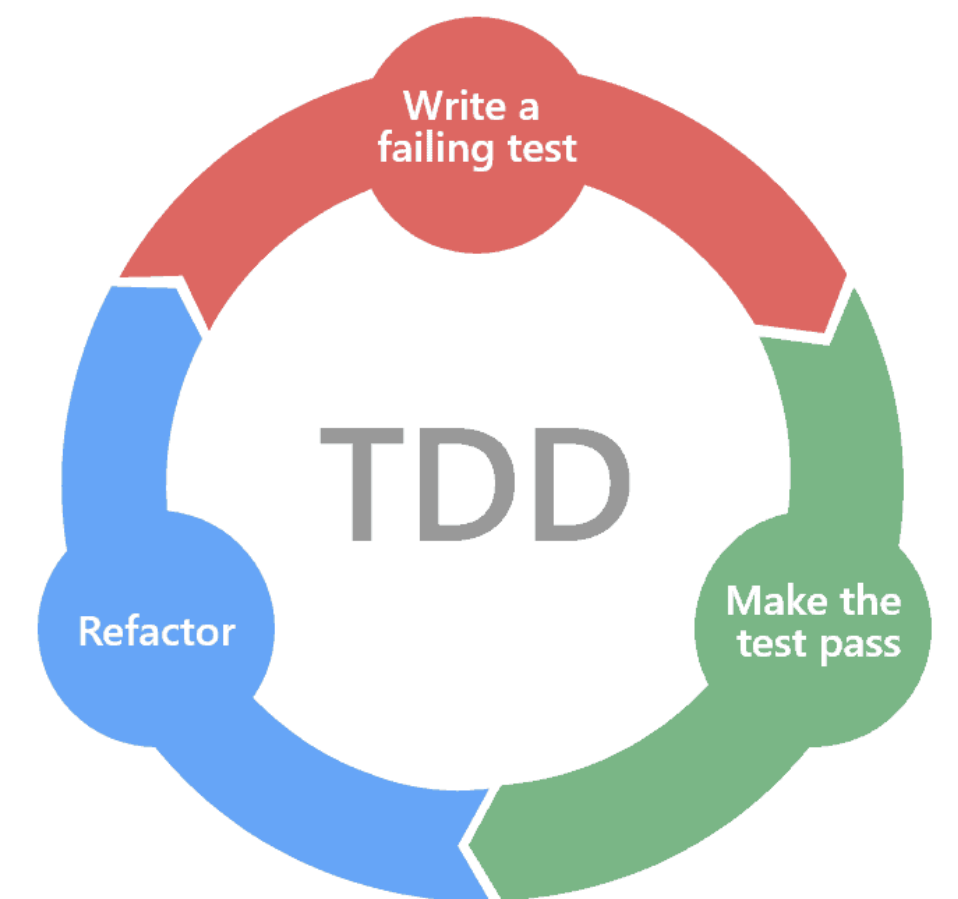
===== FAILURES =====
_____ test_answer _____

    def test_answer():
>         assert func(3) == 5
E         assert 4 == 5
E         + where 4 = func(3)

test_sample.py:6: AssertionError
===== short test summary info =====
FAILED test_sample.py::test_answer - assert 4 == 5
===== 1 failed in 0.12s =====
```

# Test-driven development in ML context

- We can start from end-goal (aka "business") requirements
- **Keep tests a natural part of your workflow!**  
This means getting a convenient setup both locally and in CI
- Leverage TDD for your ML code as well
- Test the expected changes in behavior of your model



# Property-based testing

- How do we generate test cases?
  - Coming up with our own inputs is not exhaustive
  - Basically, we only test that the code works for given inputs
  - Furthermore, our requirements become unclear
- Property-based testing aims to solve this problem
  - Instead of specifying exact inputs, we tell what they should be
  - The framework tests the code on many inputs and tries to simplify failing cases



# Hypothesis

- A Python framework for property-based-testing
- Integrates with pytest
- Has strategies for generating NumPy arrays (which generalizes to PyTorch tensors)

```
from hypothesis import given, strategies as st

@given(st.integers(), st.integers())
def test_ints_are_commutative(x, y):
    assert x + y == y + x

@given(x=st.integers(), y=st.integers())
def test_ints_cancel(x, y):
    assert (x + y) - y == x

@given(st.lists(st.integers()))
def test_reversing_twice_gives_same_list(xs):
    # This will generate lists of arbitrary length (usually between 0 and
    # 100 elements) whose elements are integers.
    ys = list(xs)
    ys.reverse()
    ys.reverse()
    assert xs == ys

@given(st.tuples(st.booleans(), st.text()))
def test_look_tuples_work_too(t):
    # A tuple is generated as the one you provided, with the corresponding
    # types in those positions.
    assert len(t) == 2
    assert isinstance(t[0], bool)
    assert isinstance(t[1], str)
```

```
>>> import numpy as np
>>> from hypothesis.strategies import floats
>>> arrays(np.float, 3, elements=floats(0, 1)).example()
array([ 0.88974794,  0.77387938,  0.1977879 ])
```

# Takeaways

- Invest time in good and convenient logging
- Make sure to keep track of your data and models
- Flexible configuration makes it easy to alter your experiment setups
- Good tests simplify debugging and maintenance