

Production-ready ML

(in Python)

Julian de Ruiter

GO 
DATA
DRIVEN

About me

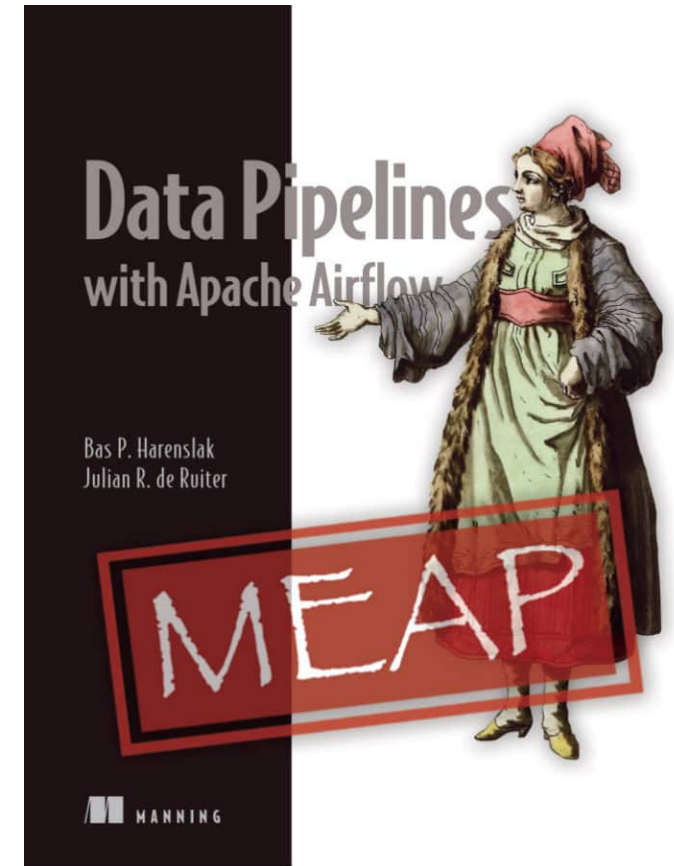
- Background in computer science and computational biology (TU Delft)
- Machine learning engineer at GDD
- Enthusiastic about using Python etc. to solve ML problems *in production*



Julian de Ruiter

About me

- Co-author of Manning's 'Data Pipelines with Apache Airflow'
- First 10 chapters available in preview, close to completing the book!
- <https://www.manning.com/books/data-pipelines-with-apache-airflow>



GO 
DATA
DRIVEN

About you

Program

- Part 1

- Introduction
- What is production-ready?
- Python packaging
- Object-oriented sklearn
- Code quality

- Part 2

- Testing
- CLI + logging
- API's with Flask
- What next?
- Wrap-up

Production-ready?

What does production-ready mean?

Some questions you can ask...

- Is the application/model robust?
- Is it maintainable? Is it secure/compliant?
- Does it add business value?
- What are the running/development costs?
- Can we easily add and test new features?
- How do we deploy and version the components?
- How is the deployed application monitored?
- ...

Why do we need to think about this?

Hidden Technical Debt in Machine Learning Systems

D. Sculley, Gary Holt, Daniel Golovin, Eugene Davydov, Todd Phillips
{dsculley, gholt, dgg, edavydov, toddphillips}@google.com
Google, Inc.

Dietmar Ebner, Vinay Chaudhary, Michael Young, Jean-François Crespo, Dan Dennison
{ebner, vchaudhary, mwyong, jfcrespo, dennison}@google.com
Google, Inc.

Abstract

Machine learning offers a fantastically powerful toolkit for building useful complex prediction systems quickly. This paper argues it is dangerous to think of these quick wins as coming for free. Using the software engineering framework of *technical debt*, we find it is common to incur massive ongoing maintenance costs in real-world ML systems. We explore several ML-specific risk factors to



ML applications are not just code



Data

Schema

Sampling over Time

Volume

+



Model

Algorithms

More Training

Experiments

+



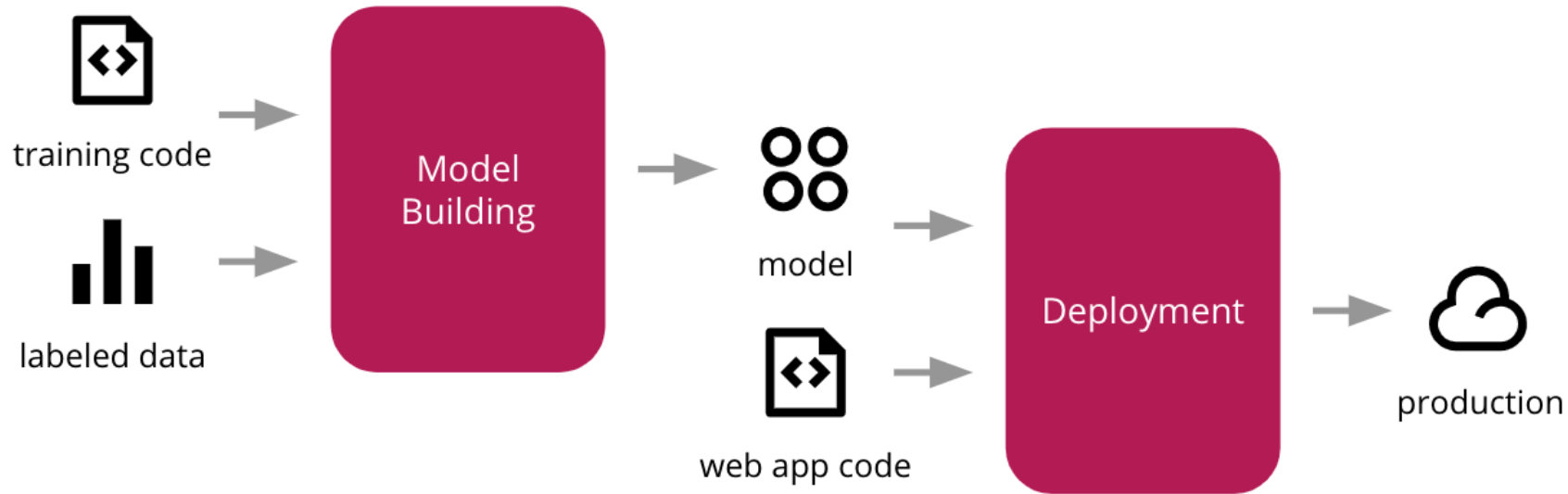
Code

Business Needs

Bug Fixes

Configuration

An example application



So, how do we know if this is production-ready?

Scoring production-readiness

The ML Test Score: A Rubric for ML Production Readiness and Technical Debt Reduction

Eric Breck, Shanqing Cai, Eric Nielsen, Michael Salib, D. Sculley
Google, Inc.

`ebreck, cais, nielsene, msalib, dsculley@google.com`

Abstract—Creating reliable, production-level machine learning systems brings on a host of concerns not found in small toy examples or even large offline research experiments. Testing and monitoring are key considerations for ensuring the production-readiness of an ML system, and for reducing technical debt of ML systems. But it can be difficult to formulate specific tests, given that the actual prediction behavior of any given model is difficult to specify *a priori*. In this paper, we present 28 specific tests and monitoring needs, drawn from experience with a wide range of production ML systems to help quantify these issues and present an easy to follow road-map to improve production readiness and pay down ML technical debt.

Keywords—Machine Learning, Testing, Monitoring, Reliability, Best Practices, Technical Debt

may find difficult. Note that this rubric focuses on issues specific to ML systems, and so does not include generic software engineering best practices such as ensuring good unit test coverage and a well-defined binary release process. Such strategies remain necessary as well. We do call out a few specific areas for unit or integration tests that have unique ML-related behavior.

How to read the tests: Each test is written as an assertion; our recommendation is to test that the assertion is true, the more frequently the better, and to fix the system if the assertion is not true.

Doesn't this all go without saying?: Before we enumerate our suggested tests, we should address one objection that you may have: why should we have unit tests for

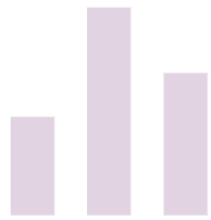
Scoring production-readiness

- Google defines several categories of checks
 - E.g. model evaluation, development practices, monitoring
- Scoring
 - Each check is scored:
 - 0 = not addressed,
 - 1 = partially addressed,
 - 2 = fully addressed + automated
 - Aggregated to calculate an overall score

Scoring production-readiness

- Interestingly, paper notes that 80%+ of Google solutions would not have a passing grade for these tests
- However, scoring can be useful to identify issues
 - Input for backlog/roadmap when moving to production
 - Flag key issues early in the development process

This training focusses on code



Data

Schema

Sampling over Time

Volume

+



Model

Algorithms

More Training

Experiments

+



Code

Business Needs

Bug Fixes

Configuration

Use case

- Our company is a ferry company looking to start things up again after being shut down during the Corona crisis
- To increase our revenue, management wants us to investigate how we can upsell more expensive tickets

Use case

- One of our data scientists was inspired after watching 'Titanic' on Netflix and created a model that predicts survival of Titanic passengers based on passenger class
- Management wants to move this model 'into production' to start upselling higher-class tickets
- Code: <https://github.com/jrderuiter/production-ready-ml>

Python packaging

Python packages

- Goal – package Python code into a redistributable package that can easily be installed in other environments
- Terminology
 - module – a something.py file
 - package – a directory with an `__init__.py`
 - subpackage – a package within a package

A basic example

my_package

└─ __init__.py

└─ my_module.py

└─ second_module.py

└─ subpackage

 └─ __init__.py

 └─ another_module.py

```
import my_package
from my_package import my_module
from my_package import second_module
from my_package import subpackage
from my_package.sub_package import another_module
```

Python path

- Importing only works if package is in your Python path

```
import sys  
print(sys.path)
```

- What's typically in your path?
 - Built-in standard library
 - Installed third-party packages
 - The current working directory (!)


Building installable packages

- Requires using a build tool like setuptools, flit, poetry, etc.
- These tools are used to build 'distributions' of your code
 - Two main formats: source (not-compiled) and wheels (compiled)
 - Can be installed using package managers like pip
- Configured using a pyproject.toml file (PEP517, PEP518)


A minimal pyproject.toml

```
[build-system]
requires = ["setuptools>=42", "wheel"]
build-backend = "setuptools.build_meta"
```

Specifies which
dependencies are needed
in the build environment



Specifies which
build tool to use
(in this case setuptools)



Additional options can be specified in other
sections (depending on the used build tool)

Setuptools

- Historically, probably the most frequently used tool for building distributions of Python packages
- Replaced distutils as de-facto tool around 2004
- Configured using a setup.py / setup.cfg file in the root of your Python package directory

setup.py

```
import setuptools

# Get the long description from the README file
with open('README.md', encoding='utf-8') as f:
    long_description = f.read()

setuptools.setup(
    name='sampleproject',
    version='1.3.1',
    description='A sample Python project',
    long_description=long_description,
    long_description_content_type='text/markdown',
    url='https://github.com/pypa/sampleproject',
    author='A. Random Developer',
    author_email='author@example.com',
    package_dir={'': 'src'},
    packages=setuptools.find_packages(where='src'),
    python_requires='>=3.5, <4',
    install_requires=['pytest>=0.23.0'],
    extras_require={
        'dev': ['pytest'],
    },
)
```



Name + version of your library



Meta data about your library



Where to find your Python code



Required
dependencies

setup.cfg

```
[metadata]
name = sampleproject
version = 1.3.1
author = A. Random Developer
description = A sample Python project.
license = unlicensed
long-description = file: README.md

[options]
zip_safe = false
include_package_data = true
python_requires = >= 3.5.0
install_requires =
    pandas >= 0.23.0
```

Alternative to setup.py,
which allows you to
specify these details in a
static, flat-text file.

To src or not to src?

- There is some discussion in the community about using a 'src' directory to separate your Python code from other files

Without src

```
|— LICENSE
|— README.md
|— my_package
|   |— __init__.py
|   └─ ...
└─ setup.py
```

With src

```
|— LICENSE
|— README.md
|— src
|   |— my_package
|       |— __init__.py
|       └─ ...
└─ setup.py
```

To src or not to src?

- Personally, I'm a fan of using a src directory
- Why? – Import parity
 - By default, Python adds the current directory to your import path
 - Means that you can accidentally import local modules
 - Src makes sure that you run + test your package as it would be when *installed* by the end user
- For more details see [here](#) and [here](#).

Additional files

- Besides code, Python packages also typically contain:
 - Documentation (Sphinx)
 - Unit/integration tests (Unittest library or pytest)
 - Additional readme/configuration files
- We will go into these later

Installing your library

- Your library can easily be installed using pip:
 - `pip install .`
- During development, you can use an editable install:
 - `pip install --editable .`
- This way, edits are directly reflected in your environment.
(Note: notebooks require the autoreload extension.)

Building your distribution

- Using pyproject.toml, you can build you package using:

```
pip install pep517
python -m pep517.build .
ls -l dist
```

- This will also build a wheel + source distribution, which can be distributed and installed elsewhere

Other build tools

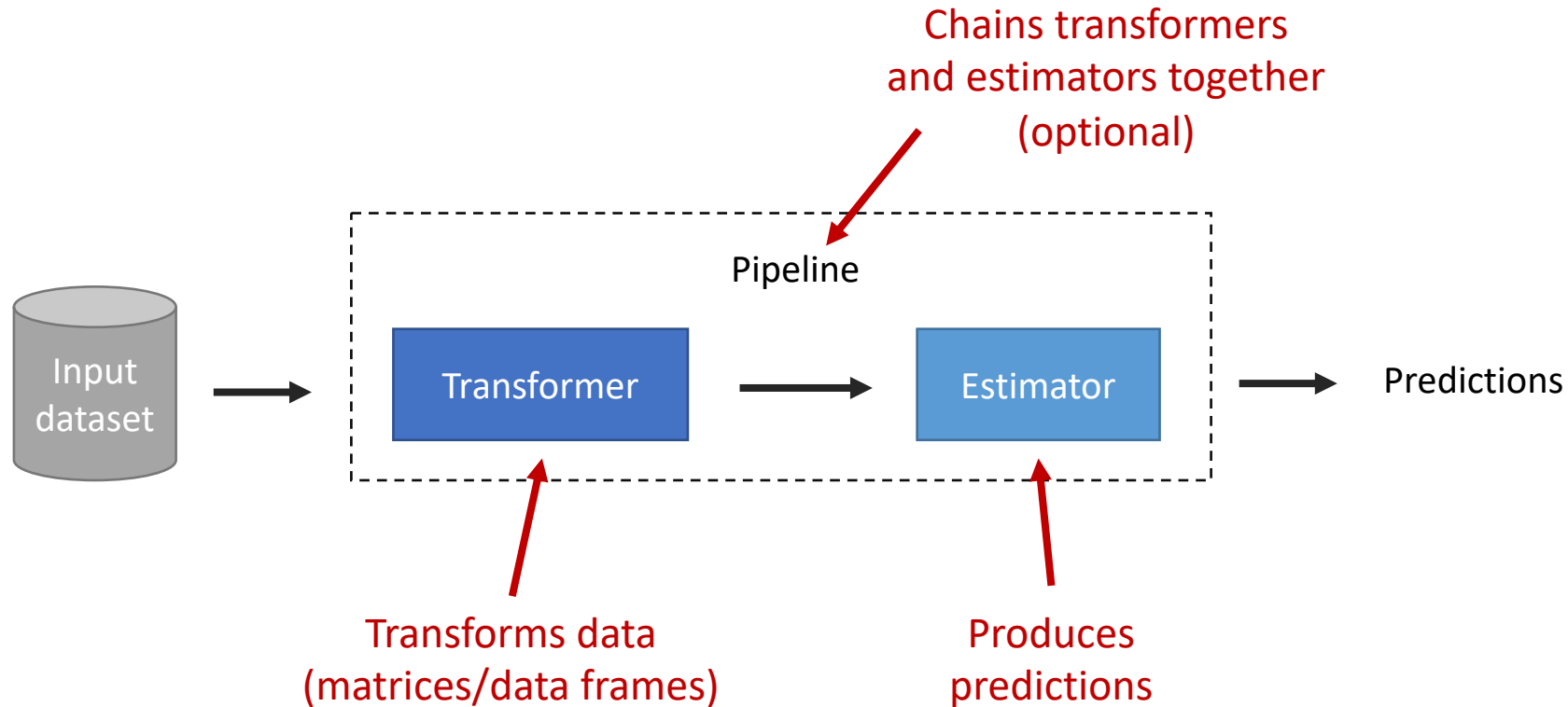
- PEPs 517 + 518 have resulted in several other build tools
 - Flit (<https://github.com/takluyver/flit>)
 - Poetry (<https://python-poetry.org/>)
- Check them out if you're interested in seeing how they differ from each other!

Additional resources

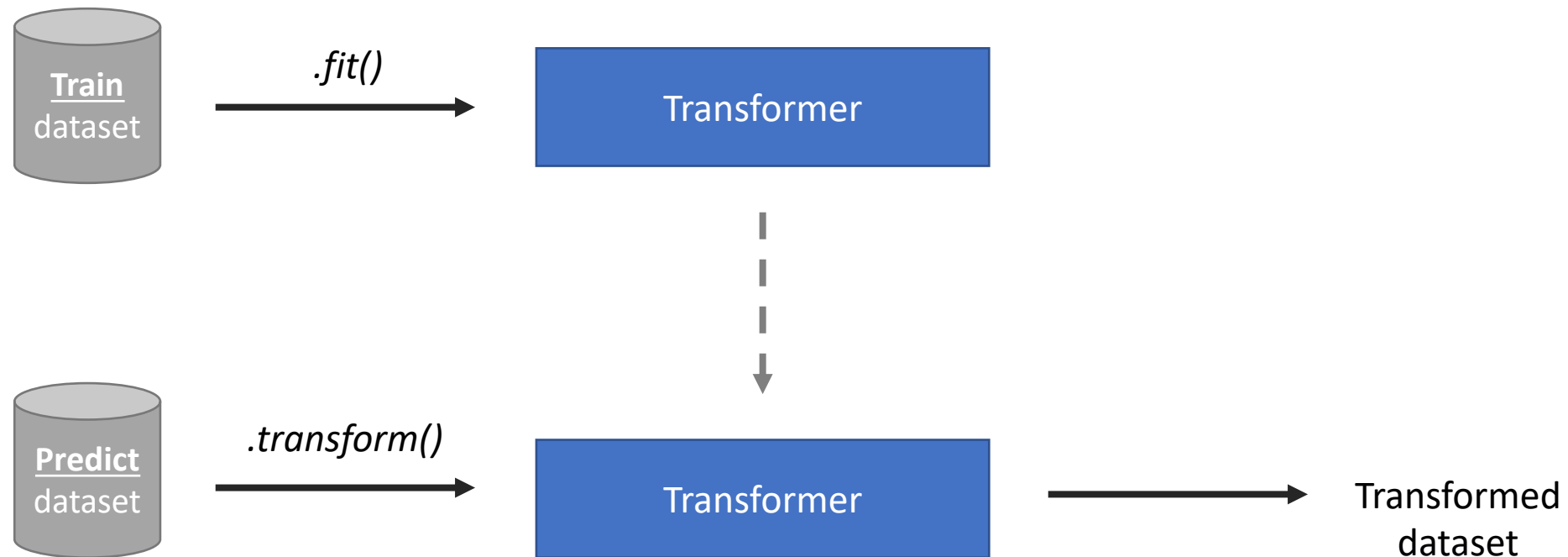
- [Python tutorial: packaging Python projects](#)
- [A tour on Python packaging](#)
- [Python's new package landscape](#)
- [Clarifying PEP 518](#)
- [Specifying dependencies \(setup.py vs requirements.txt\)](#)

(Object-oriented) Scikit-learn

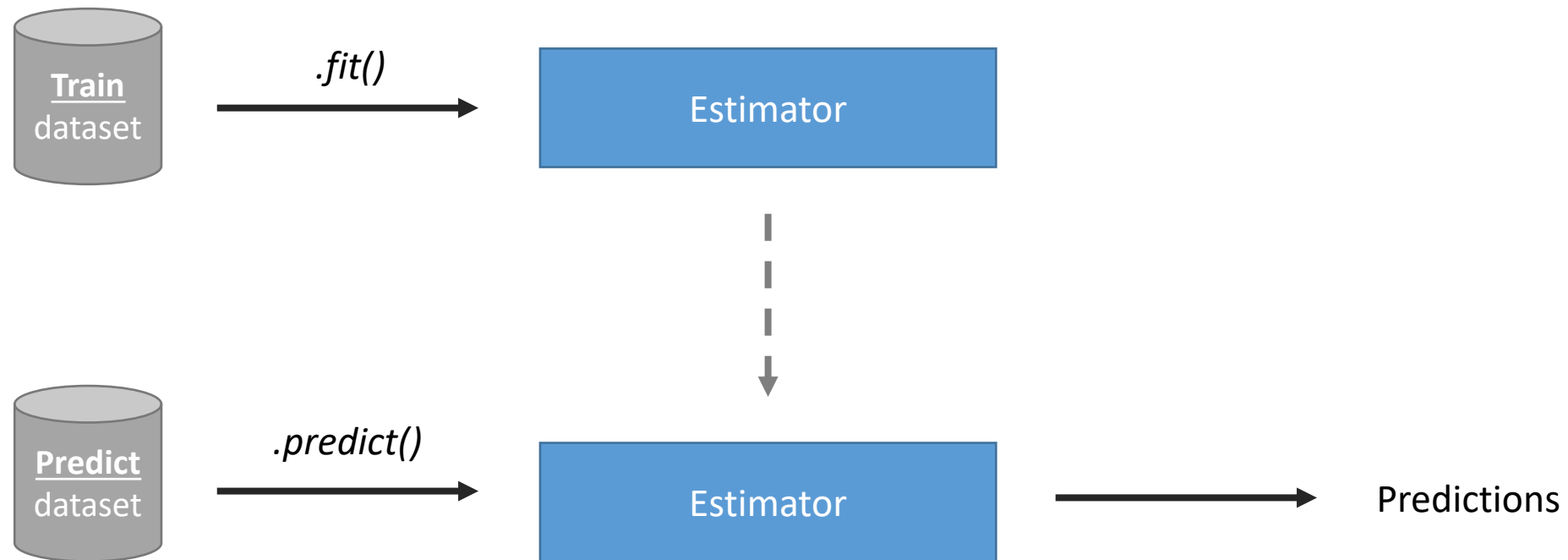
Transformers & Estimators



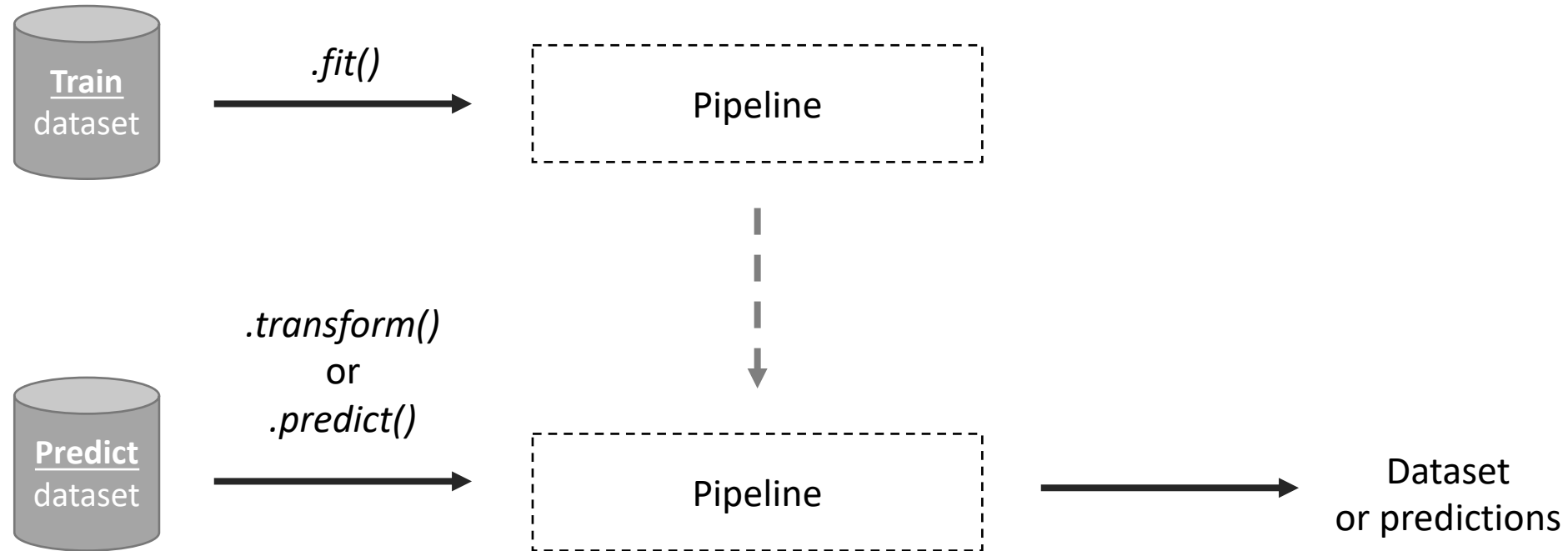
Fitting transformers & estimators



Fitting transformers & estimators



Fitting transformers & estimators



A pipeline is just another transformer/estimator!

Why use this approach?

Why use this approach?

- Provides a well-defined interface
 - Use any kind of transformation/model with minimal changes
 - Allows us to create clear, reusable building blocks
 - Improves testability of these blocks
- Class-based approach allows keeping state
 - Important for storing fitted parameters

Building your own Transformer

```
from sklearn.base import BaseEstimator, TransformerMixin
```

```
class MyTransformer(BaseEstimator, TransformerMixin):
```

```
    def __init__(self, my_arg):  
        super().__init__()  
        self.my_arg = my_arg
```

```
    def fit(self, X, y=None):  
        self.my_param_ = ...  
        return self
```

```
    def transform(self, X):  
        X_transformed = ...  
        return X_transformed
```

Building your own Estimator

```
from sklearn.base import BaseEstimator

class MyEstimator(BaseEstimator):

    def __init__(self, my_arg):
        super().__init__()
        self.my_arg

    def fit(self, X, y=None, **fit_params):
        estimator = ...
        self.estimator_ = estimator.fit(X, y=y)
        return self

    def predict(self, X, **predict_params):
        check_is_fitted(self, ["estimator_"])
        return self.estimator_.predict(X)
```

Caveats with parameters

- Scikit-learn expects you to follow these conventions:
 - Set any arguments to init on self with the same name
 - Suffix fitted parameters with a `_` (e.g. `self.coefficients_`)
- Not following conventions will probably break the standard `get_params/set_params` functionality
- Alternatively, implement your own `get/set_params`

*Always good
to test!*

Code quality

Code quality

Code is a means to communicate not only with machines but also with other developers. High quality code is good communication.

High quality code is:

- Correct
- Human readable
- Consistent
- Modular
- Reusable

Programming principles

- Do one thing and do it well
 - Have small, focused functions/classes that only do one thing
 - Functions should be logical units



```
def detect_machines(data, start, end):  
    # Filter data  
    dates = pd.date_range(start, end)  
    start = start - 1  
    end = end + 1  
    filtered = data[start:end]  
    # Find events  
    ...  
    # Count machines  
    ...
```



```
def detect_machines(data, start, end):  
    filtered = filter_data(data[start:end])  
    machines = find_machines(filtered)  
    n_machines = count_machines(machines)  
    return n_machines  
  
def filter_data(start, end):  
    ...  
  
def detect_events(filtered):  
    ...  
  
def count_machines(machines):  
    ...
```

Programming principles

Goal – write code other people (and future you) can understand

Don'ts

- Long functions doing multiple things
- Copy/paste code
- Re-use variable names in function
- One-char variables, abbreviations
- Variables that differ by one character
- Long, complicated variable names
- Many temp vars (or using your own defaults)
 - `bassie, buh, zip`

Do's

- Small functions doing one thing
 - `check_boiler()`
 - `load_rankings()`
- Build libraries, functions, classes
- Follow existing design patterns
- Descriptive and concise variables:
 - `male_user, is_fridge`
- Common temp vars
 - `temp, df`

Code style

- Every language has it's own accepted style guide(s)
 - Consistent reading experience
 - Easy to recognize what code does
 - Don't invent your own style
- Examples
 - Python – PEP8
 - R – Google Style or Advanced R

Code style – PEP8

- Examples

- Functions/variables `lower_case_variable`
- Classes `UpperCaseClass`
- Whitespace `do_this(whitespace, next, to, commas,
and, proper, indentation)`

- Many useful tools

- Style checkers – Flake8, Pylint
- Automated formatters – YAPF, Black

Using formatting / linting tools

- Many tools such as Pylint/Black are installable using pip:

```
$ python -m pip install black pylint
$ python -m black --check .
$ python -m pylint .
```

- Note that this does require installing them into your environment (typically as a development dependency)

Documentation

- One way to improve the readability of your code is to add (proper) documentation
- Different documentation types
 - Inline comments - explain what specific pieces of code do
 - Docstrings - document Python functions/classes/modules
 - Actual documentation – how to install, usage guide, etc.

Comments

- Avoid adding comments explaining the obvious
- Think about the choices/assumptions you make in your code, which are not directly clear from the code itself



```
# import packages  
import pandas as pd  
  
# load some data  
df = pd.read_csv('data.csv', skiprows=2)
```



```
# Data contains two lines of description  
# text, skip to avoid errors.  
df = pd.read_csv('data.csv', skiprows=2)
```

Docstrings

- Docstrings document how to use specific functionality

```
def rescale_between(array1d, lower=0.0, upper=5.0):  
    """Rescales array values between given upper/lower bounds.  
  
    :param np.ndarray array1d: Values to be rescaled.  
    :param float lower: Lower bound of the rescaled values.  
    :param float upper: Upper bound of the rescaled values.  
  
    :returns: Array containing rescaled values.  
    :rtype: np.ndarray  
    """  
    ...
```

- Can be accessed using ``help(...)`` (or `?` in IPython/Jupyter)

Docstrings – Other styles

- Other docstring styles can also be used, such as the Numpy style:

```
def rescale_between(array1d, lower=0.0, upper=5.0):  
    """  
    Rescales array values between given upper/lower bounds.  
  
    Parameters  
    -----  
    array1d : np.ndarray  
        Values to be rescaled.  
    upper : float  
        Lower bound of the rescaled values.  
    lower : float  
        Upper bound of the rescaled values.  
  
    Returns  
    -----  
    Array containing rescaled values.  
    """  
    ...
```

Sphinx



- Sphinx is the de-facto tool to use in Python for writing and generating docs
- Docs are written in [reStructuredText](#)
- Many features
 - Hierarchical structure, table of contents, etc.
 - Generating docs from docstrings
 - Different themes, output types (html)

Sphinx – getting started



- [Getting started](#)
 - Install sphinx using ``pip install sphinx``
 - Generate initial template using ``sphinx-quickstart``
 - Start editing your docs
 - Build your docs using ``make html`` (in the docs folder)
- Some templates (cookiecutter-pypackage) include an initial structure that you can use

Automating commands

- For complicated commands, you may want to provide a default setup to make them easy to run by everyone
- One approach is to use [Makefiles](#) to define simple commands (e.g. *make lint*) for people to run tools
- Another approach is to use tools like [pre-commit](#) for running commands automatically with every commit

Additional resources

- [PEP8 – Python Style guide](#)
- [Code formatting using Black](#)
- [Checking code quality with Pylint](#)
- [Makefiles in Python projects](#)
- [Automating checks with pre-commit](#)

Writing tests

Testing

- Why test?
 - Confirm that your code does what you expect
 - Prevent regressions (code changes that change behavior)
- Two (main) types of tests
 - Unit tests – tests a single function/method
 - Integration tests – tests behavior of combined functions

Testing frameworks

- Python has multiple testing frameworks
 - unittest – builtin framework, inspired by JUnit
 - Nose/Pytest – popular third party libraries
- We will focus on [Pytest](#)
 - Easy to use, with little boilerplate code
 - Can be a bit 'magic' in the beginning

Test structure

```
|— my_package
|   |— __init__.py
|   |— helpers.py
|   |— utils.py
|— tests
|   |— conftest.py
|   |— my_package
|       |— test_helpers.py
|— setup.py
|— ...
```

Package code

Tests mirror
package structure

A simple example

```
# test_helpers.py

from my_package.helpers import add_two

class TestAddTwo:
    """Tests for the add_two helper function."""

    def test_positive(self):
        """Tests addition with a positive number."""
        assert add_two(1) == 3

    def test_negative(self):
        """Tests addition with a negative number."""
        assert add_two(-3) == -1
```

Fixtures

- Fixtures allow you to define functions that setup elements required by (multiple) tests

```
import pytest
import smtplib

@pytest.fixture(scope="module")
def smtp_connection():
    return smtplib.SMTP("smtp.gmail.com", 587, timeout=5)

def test_smtp(smtp_connection):
    ...
```


Running pytest

```
$ pytest
```

```
===== test session starts =====  
platform linux -- Python 3.x.y, pytest-3.x.y, py-1.x.y, pluggy-0.x.y  
rootdir: $REGENDOC_TMPDIR, inifile:  
collected 1 item
```

```
test_sample.py F [100%]
```

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

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

```
test_sample.py:6: AssertionError  
===== 1 failed in 0.12 seconds =====
```

When to stop testing?

- So when do we have enough tests?
 - Ideally – when our code is bug-free
 - In practice – when we have 'enough' confidence in our code
- A popular metric is code coverage
 - Percentage of code covered by tests
 - Note: code with 100% coverage is not bug-free
- Can be generated in pytest using `pytest-cov` plugin

Command line scripts

Python scripts

- You can easily convert a Python module into a script:

```
#!/usr/bin/env python
```



Tells the loader to use Python

```
def main():  
    ...
```



Our function that will do the work.

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



Tells Python to execute the main function when file is executed as script.

Argument parsing - Argparse

- For many applications, you will need to parse some kind of arguments passed by the user
- Python provides a built-in library for this called argparse
 - Allows you to define a 'parser' with some specific arguments
 - Parses passed user arguments and prints help messages

Argument parsing - Argparse

```
import argparse

def parse_args():
    parser = argparse.ArgumentParser()
    parser.add_argument("input_file")
    parser.add_argument("output_file")
    parser.add_argument(
        "--my_option", default=2,
        help="Some optional value", type=int
    )
    return parser.parse_args()

def main():
    args = parse_args()
    ...
```

Argument parsing - Click

```
import click

@click.command()
@click.argument("input_file")
@click.argument("output_file")
@click.option("--my_option", help="Some optional value", default=2)
def main(input_file, output_file, my_option):
    """My CLI application."""
    ...

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

Usage

```
$ python my_script.py --help
Usage: my_script.py [OPTIONS] INPUT_FILE OUTPUT_FILE
```

My CLI application.

Options:

```
--my_option INTEGER  Some optional value
--help               Show this message and exit.
```


Entrypoints

- You can also automatically create commands for your CLI when your package is installed using entrypoints

```
import setuptools

setuptools.setup(
    ...
    entry_points={
        "console_scripts": [
            "titanic=titanic.cli:cli"
        ]
    },
    ...
)
```

Executes CLI function in the titanic.cli module for your command.

Run with: `$ titanic --help`

Logging

Why logging?

- See what's happening when running your code
 - Trace your execution to identify potential issues
 - Monitor a model performance in production
- Better than *print*
 - Easy to see where messages come from
 - You can log to different outputs (stdout, files, etc.)
 - Allows filtering based on severity (info, debug, etc.)

An example

```
import logging

logging.basicConfig(
    filename='output.log',
    level=logging.DEBUG
)

logger = logging.getLogger(__name__)
logger.info('This an info statement')
logger.warning('This a warning statement.')
logger.debug('This is a debug statement.')
```

Formatters & handlers

```
import logging
import sys

logger = logging.getLogger("mypackage")

# Set logger level.
logger.setLevel(logging.INFO)

# Configure a handler that writes to stderr.
handler = logging.StreamHandler(sys.stderr)

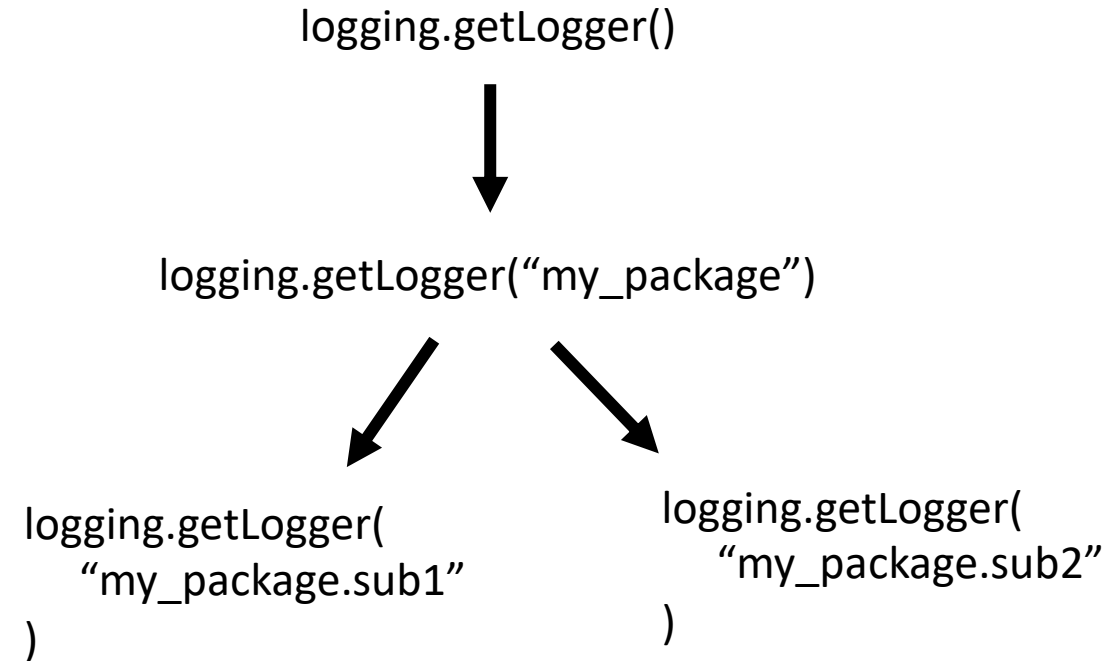
# Create formatter and add it to the handler
formatter = logging.Formatter(
    "[%asctime)-15s] %(name)s - %(levelname)s - %(message)s"
)
handler.setFormatter(formatter)

logger.handlers = [handler]
```

Logging hierarchy

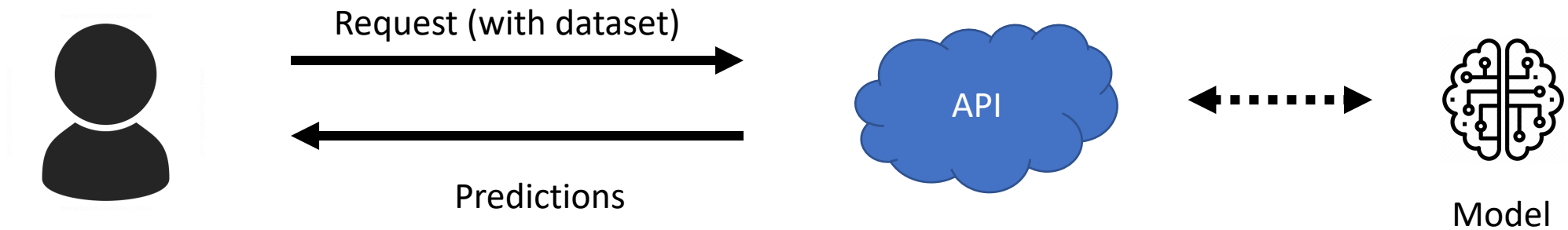
- Loggers are organized hierarchically
 - Any settings also apply for nested loggers
 - Can be used to configure different logging per package/subpackage
- This is why people often use `logging.getLogger(__name__)`

References the
current module name



Building API's with Flask

Building a (prediction) API



Flask

"Flask is a microframework for Python based on Werkzeug, Jinja 2 and good intentions."



- A minimal [example](#):

```
from flask import Flask
app = Flask(__name__)

@app.route('/')
def hello_world():
    return 'Hello, World!', 200
```

```
$ export FLASK_APP=hello.py
$ python -m flask run
```

Flask – class-based approach

```
from flask import Flask

class App(Flask):
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.add_url_rule(
            "/",
            view_func=self.hello_world,
            methods=["GET"])

    def hello_world(self):
        return "Hello world!", 200
```

Flask is single-threaded

- Flask starts a Python process which is a single thread. This means it can only handle one request at a time.
- You must wrap the application in a WSGI (Web Server Gateway Interface) to serve multiple clients simultaneously.
- For more details, take a look at [deploying Flask in Production](#).

Making requests

"Requests is the only Non-GMO HTTP library for Python, safe for human consumption."



```
>>> r = requests.get('https://api.github.com/user')
>>> r.status_code
200
>>> r.headers['content-type']
'application/json; charset=utf8'
>>> r.text
'{"type":"User"... '
>>> r.json()
{'private_gists': 419, 'total_private_repos': 77, ...}
```

GO 
DATA
DRIVEN

Web theory – methods

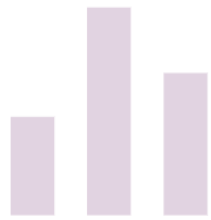
- There are different types of request methods:
 - GET - Retrieve the resource from the server
 - POST - Create a resource on the server
 - PUT - Update the resource on the server
 - DELETE - Delete the resource from the server
- In general; you should keep GET requests limited to requests that do not change the state of the server.

Web theory – response types

- The status of a HTTP request is indicated using a code:
 - 1xx - continue
 - 2xx - you got a response
 - 4xx - server thinks a client made an error
 - 3xx - redirect
 - 5xx - server thinks that it made an error
- We're omitting a lot of details now and a full summary can be found [here](#).

What next?

Up to now, we've focused on code



Data

Schema

Sampling over Time

Volume

+



Model

Algorithms

More Training

Experiments

+



Code

Business Needs

Bug Fixes

Configuration

GO 
DATA
DRIVEN

What about everything else?



Data

Schema

Sampling over Time

Volume

+



Model

Algorithms

More Training

Experiments

+



Code

Business Needs

Bug Fixes

Configuration

GO 
DATA
DRIVEN

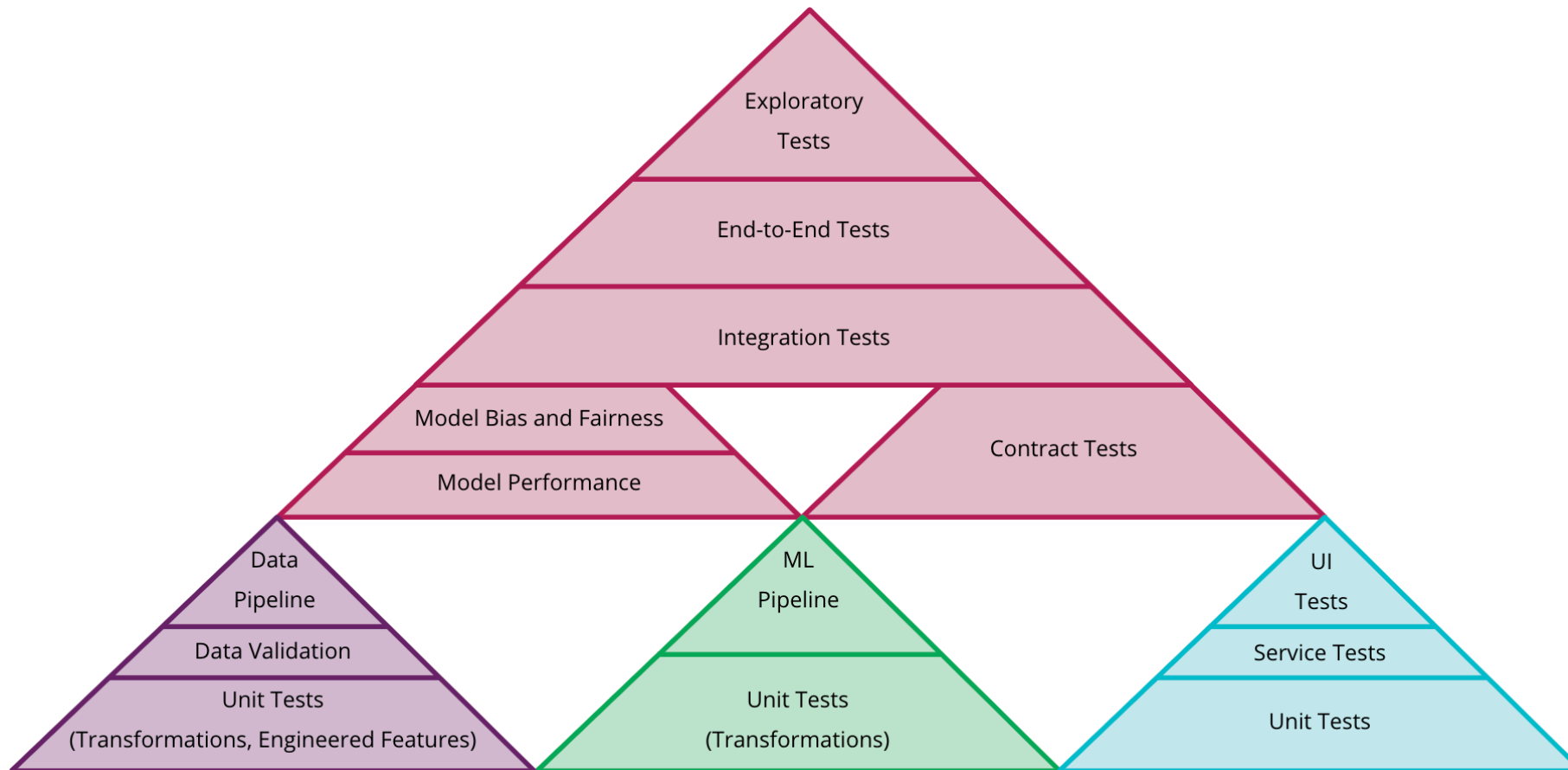
Managing data

- How is your data organized?
 - Easily discoverable/accessible?
 - Clean and well-defined?
 - How are delta's handled?
 - Are datasets versioned?
- Important to treat data pipelines etc. as artifacts, similar to the model and other components

Managing models

- How do you organize the modeling process?
 - Do you track the results of training runs?
 - Are training runs reproducible?
 - Strategy for adding changes? (Branching, etc.)
 - Approach for comparing performance?
- Should be an integral part of the process

Testing and quality



<https://martinfowler.com/articles/cd4ml.html>

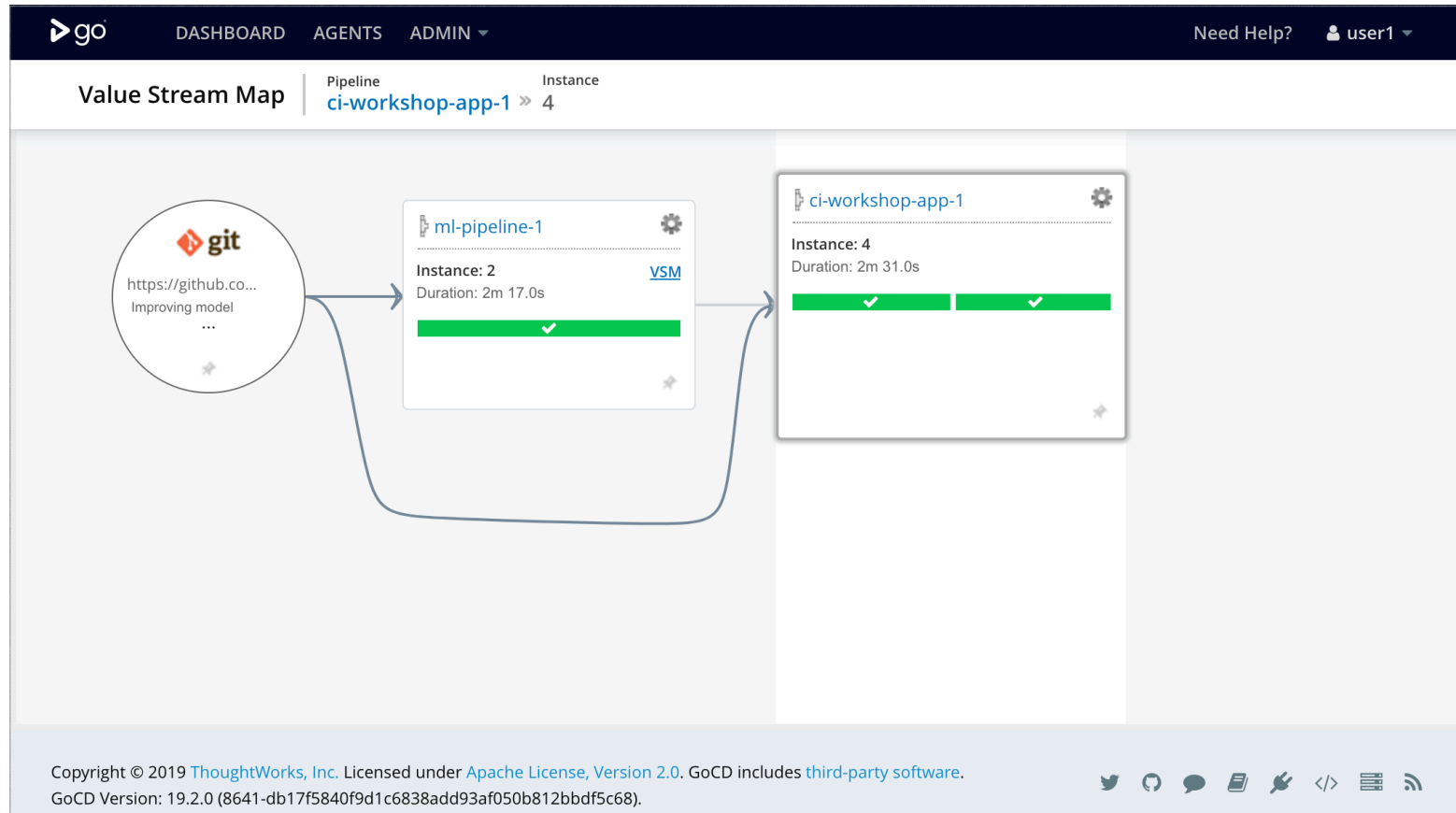
Deployment

- How will our models be deployed?
 - One or multiple models?
 - Shadow models
 - Competing models
 - Online learning
- What about other components? (E.g. pipelines, web app, ...)
- Should be managed and automated

Continuous delivery

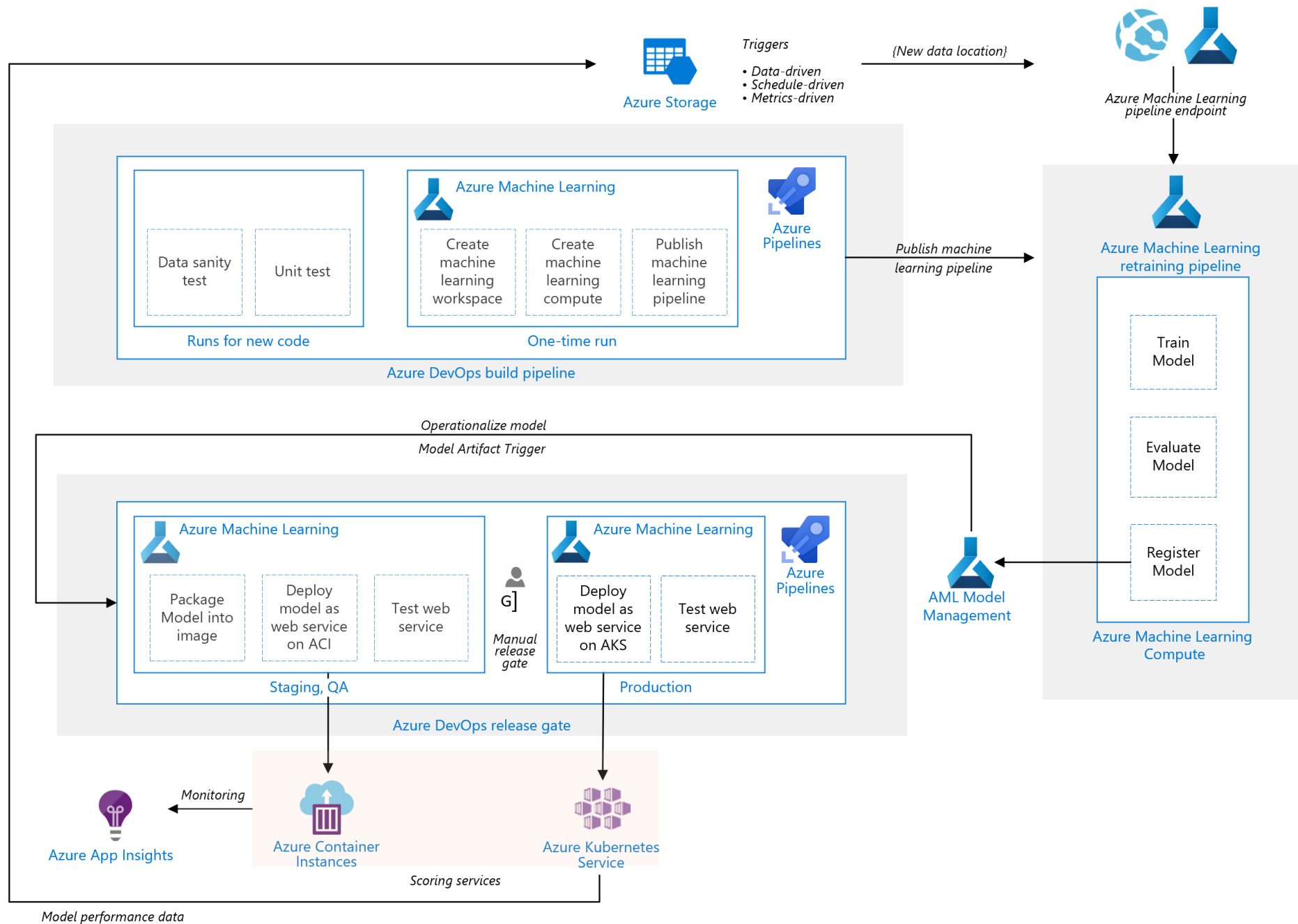
- Machine learning pipeline
 - Performs model training and evaluation
 - Can push new model versions, if sufficient
- Application deployment pipeline
 - Builds and tests application code, together with model
 - Packages and deploys new artifact to platform

Continuous delivery



<https://martinfowler.com/articles/cd4ml.html>

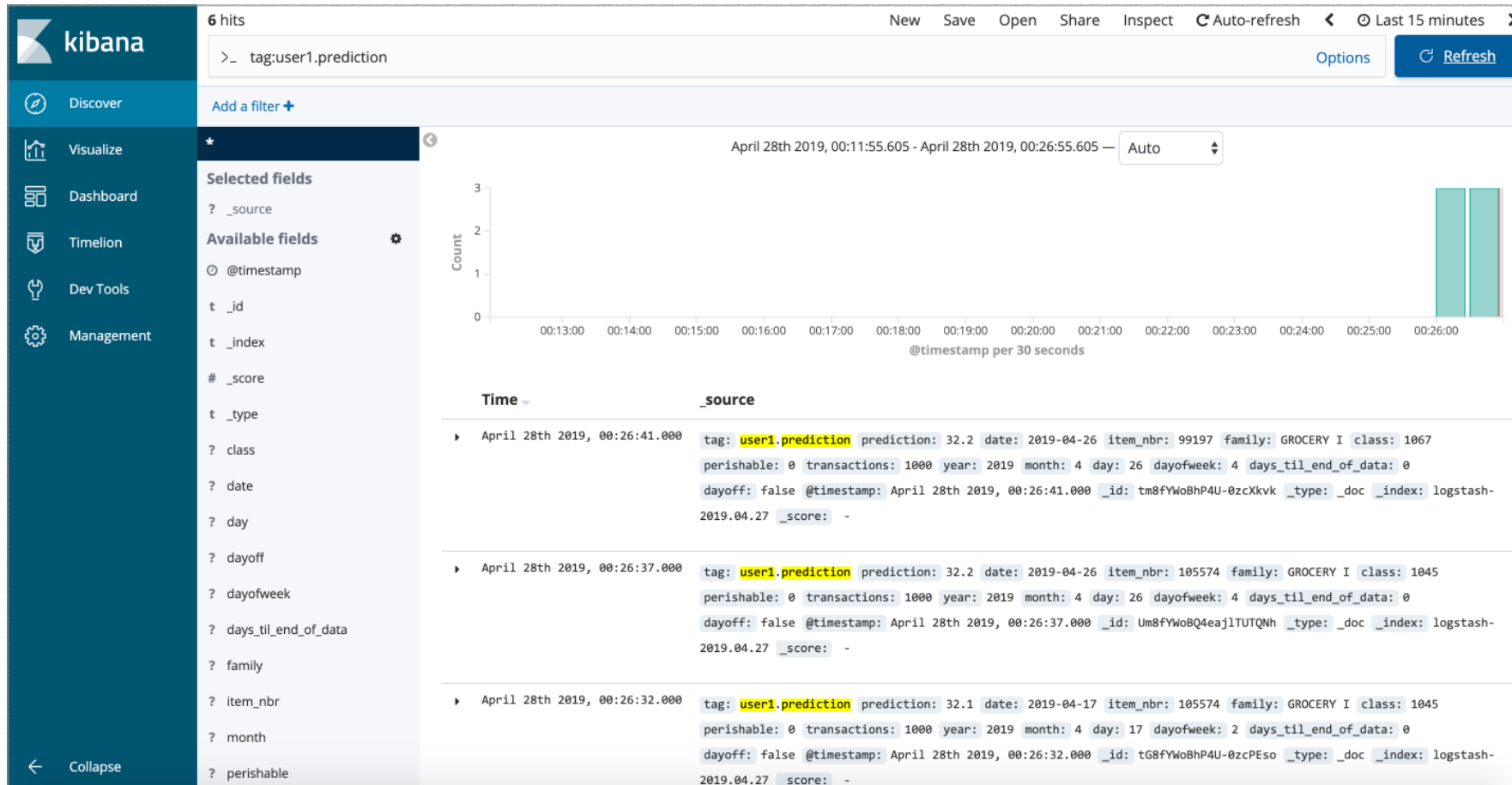
GO
DATA
DRIVEN



Monitoring

- Input datasets (data skew)
- Outputs and decisions
- Interpretability outputs (coefficients, LIME, etc.)
- User action and rewards
- Fairness (bias towards certain groups)
- Performance (latency, hardware usage, etc.)

Monitoring

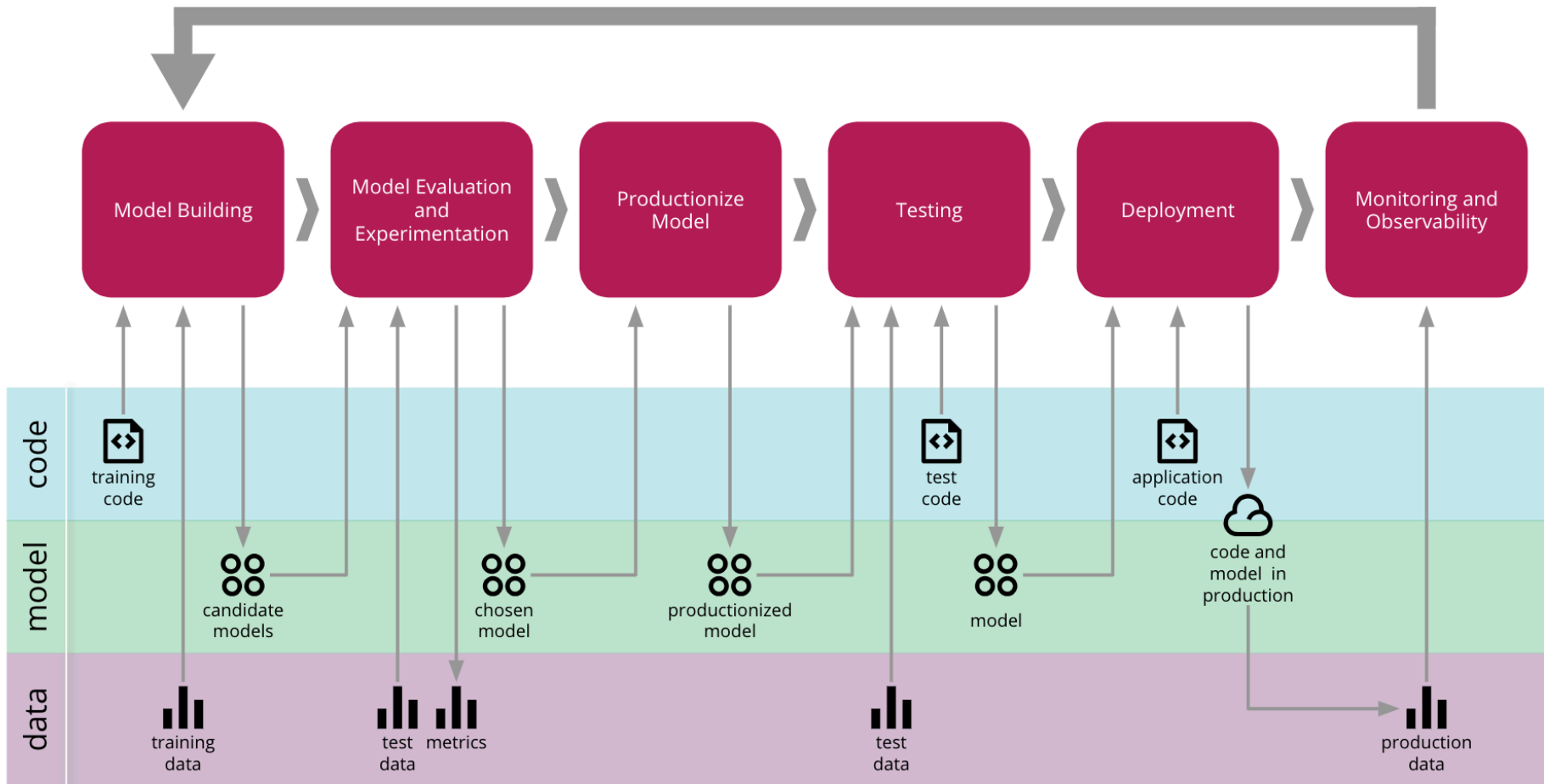


<https://martinfowler.com/articles/cd4ml.html>

EFK stack

GO
DATA
DRIVEN

End-to-end process



<https://martinfowler.com/articles/cd4ml.html>

Exercise (homework)

- Read through the excellent article on CD4ML by Martin Fowler (<https://martinfowler.com/articles/cd4ml.html>).
- Think about how the different aspects of the article are handled in your platform + solution.
 - What is going well?
 - What still needs addressing?
 - What ideas do you have for addressing these issues?

Questions?