CI/CD/CT

Robert Clements

MSDS Program

University of San Francisco

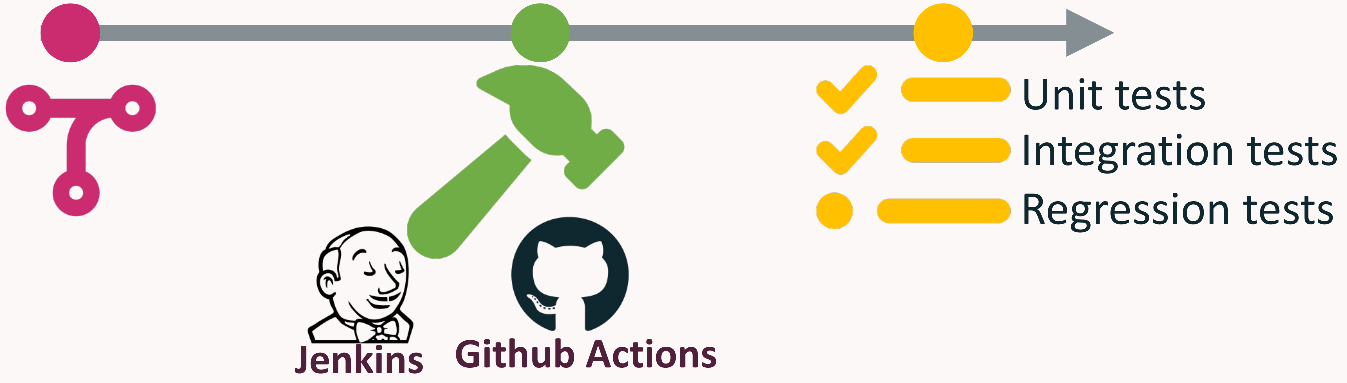
What to Expect

- Goal: to understand different best practices and the CI/CD/CT process as applied to ML pipelines only

- How: we will discuss best practices in terms of testing, using Git, and also CI/CD/CT using Github Actions

CI/CD for ML Revisited

- DevOps – for speeding up deployment of software applications using automation
- Build, test, and deploy
- Integration:
  - Merge branches
  - Kick off build
  - Kick off tests
- Delivery:
  - Deployment



CI/CD + CT (Continuous Training) for ML Revisited

Experimentation

- Ingest data
- Train and validate model

Experimentation: data scientist develops a model, and then creates all of the components needed for training and scoring the model.

- ingest/validate data
- create/validate features
- train/validate model
- model scoring

CI/CD + CT (Continuous Training) for ML Revisited

Experimentation

- Ingest data
- Train and validate model

CI/CD →

ML Pipeline/Scoring Pipeline

Continuous Integration and Delivery:
Data scientist wraps up everything into ML pipelines that can train, validate, deploy, and serve their model.

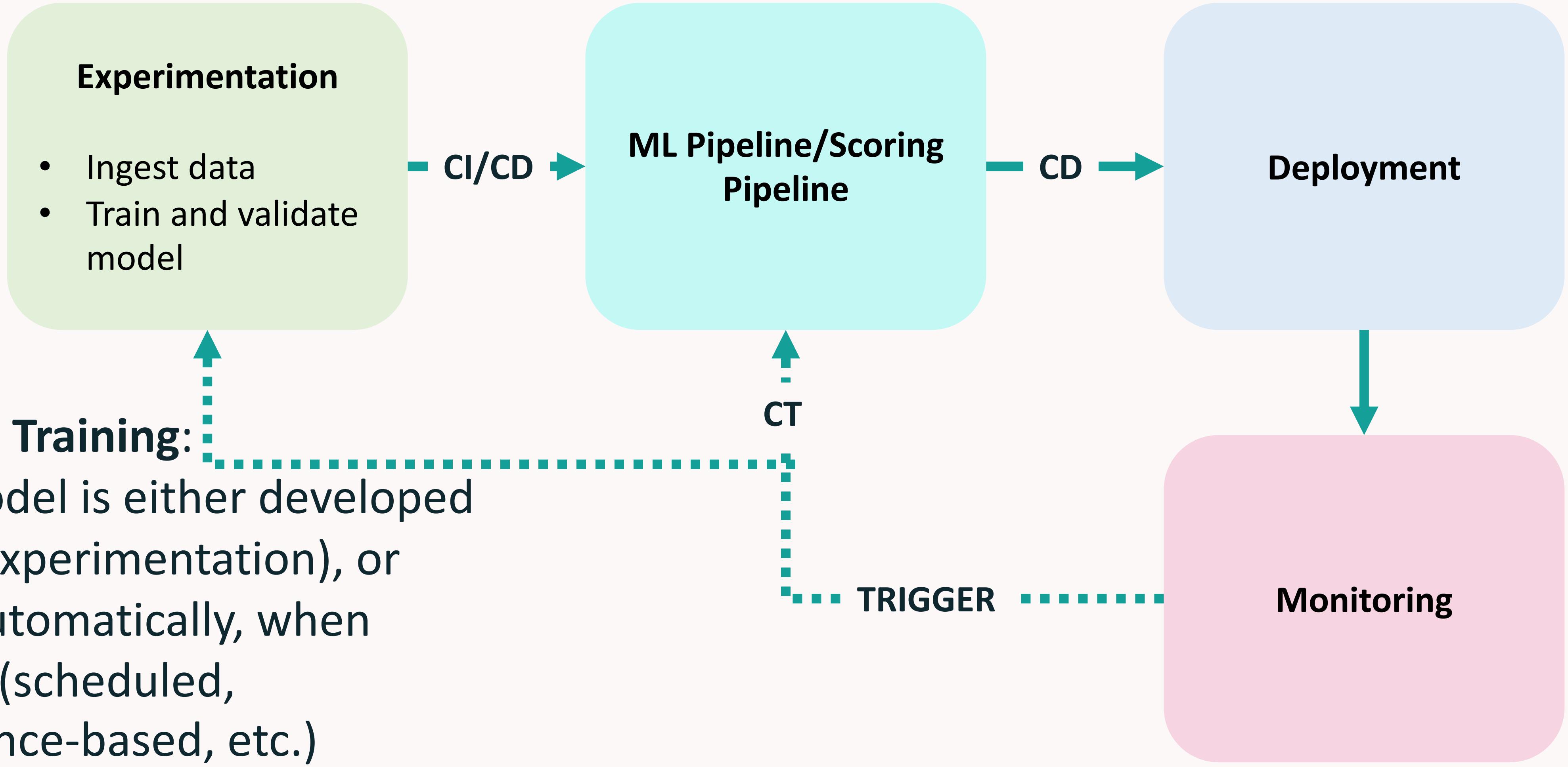
- build
- test
- package

Docker image triggered with push to Github.

CI/CD + CT (Continuous Training) for ML Revisited

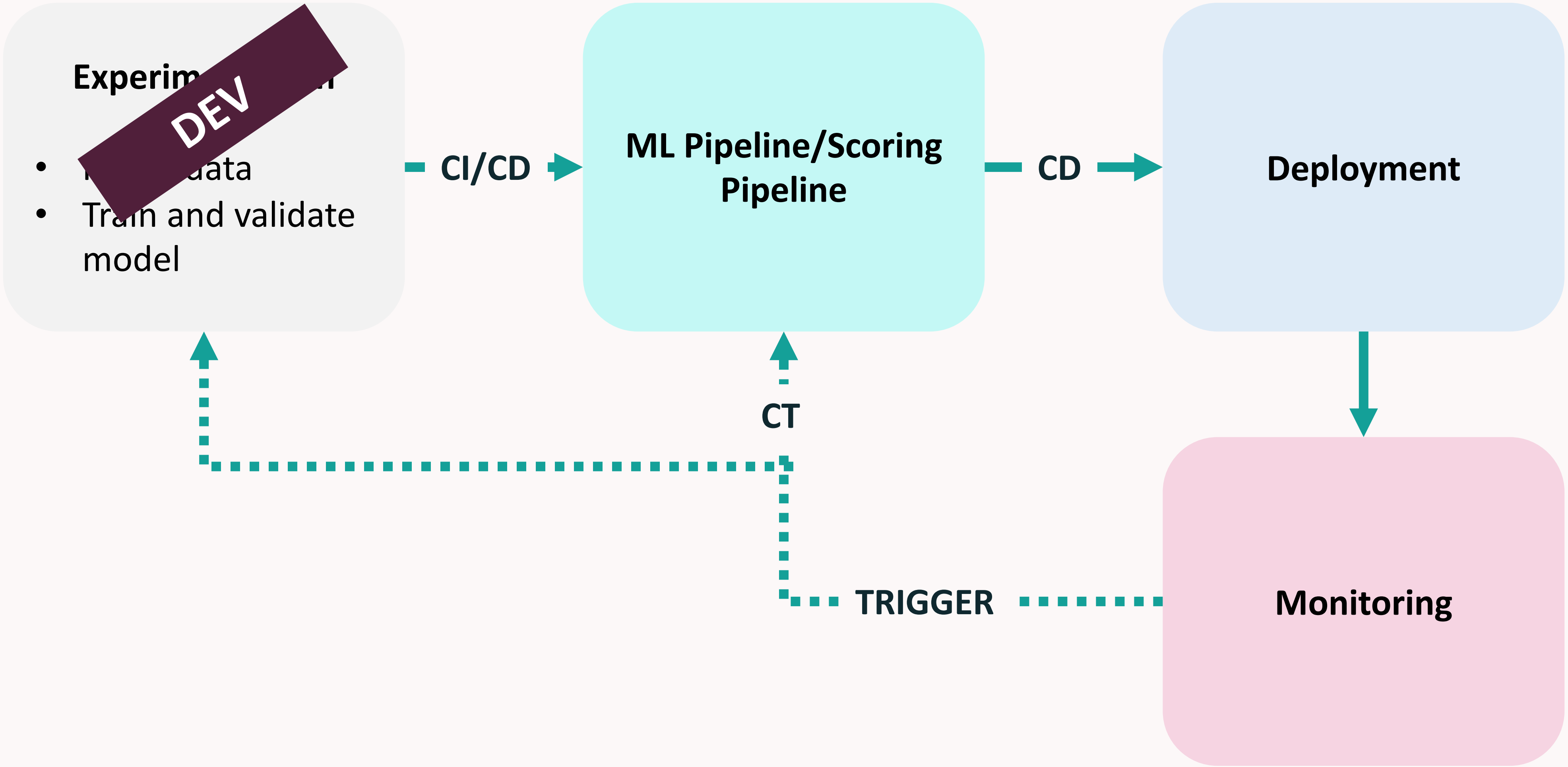
Continuous Delivery: automated deployment of model as a prediction service by registering, storing and serving ***latest version*** of model trained from the ML Pipeline stage.

CI/CD + CT (Continuous Training) for ML Revisited



Continual Training:
A new model is either developed (back to Experimentation), or trained automatically, when triggered (scheduled, performance-based, etc.)

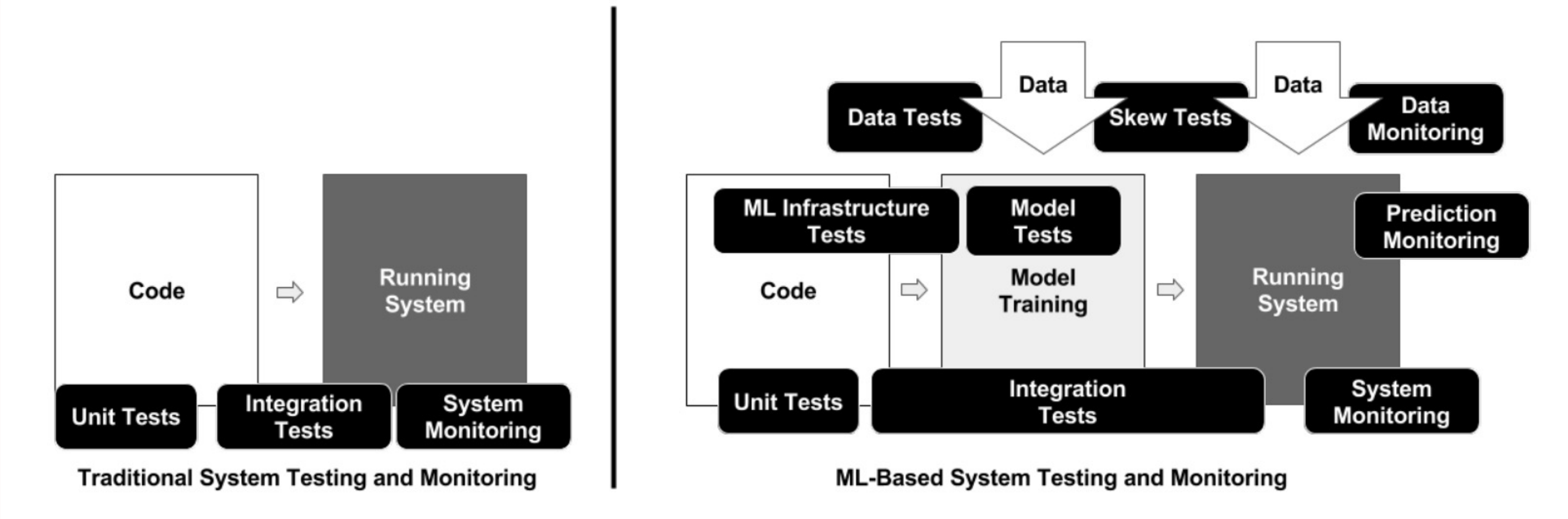
CI/CD + CT (Continuous Training) for ML Revisited



What exactly do we want to test?

Model Score

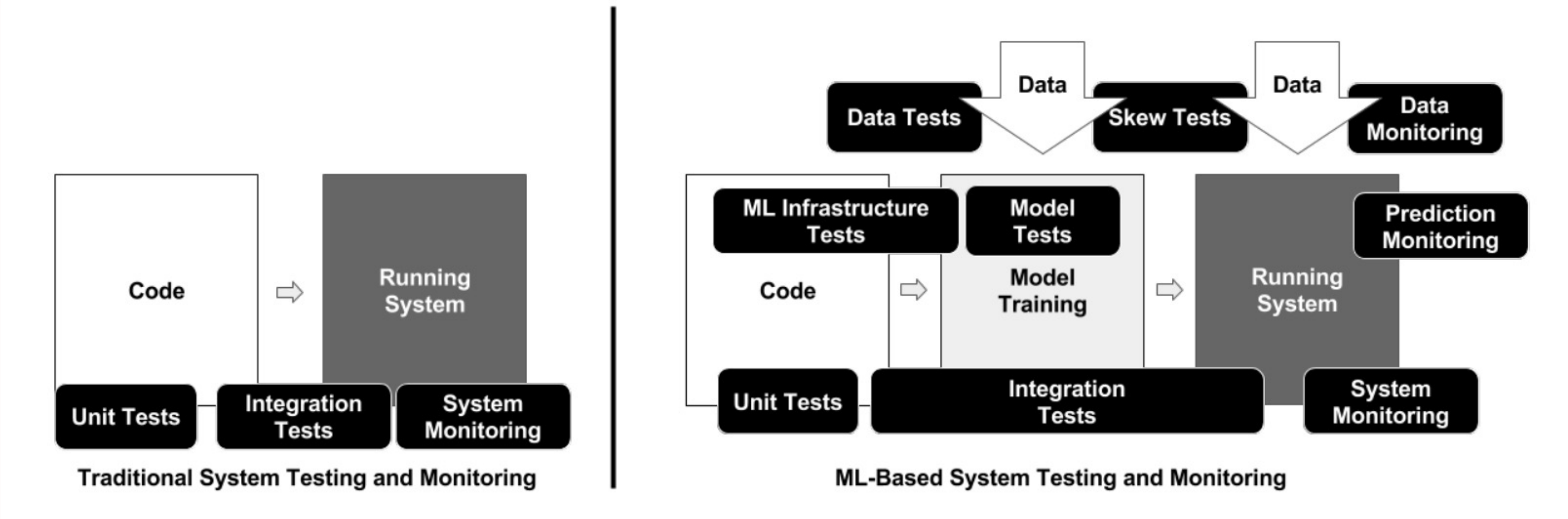
- 28 tests and monitoring needs to determine production readiness
- Developed at Google
- Published here



Source: https://research.google/pubs/pub46555/

Model Score – Data Tests

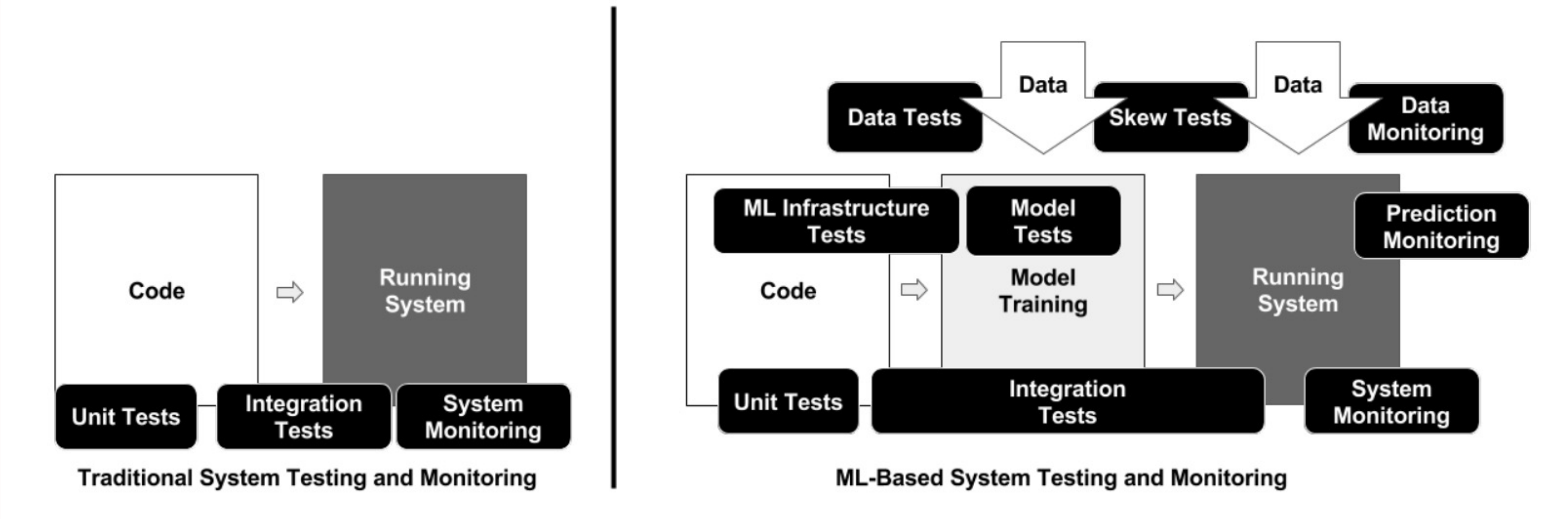
1. Feature expectations are captured in a schema.
2. All features are beneficial.
3. No feature's cost is too much.
4. Features adhere to meta-level requirements.
5. The data pipeline has appropriate privacy controls.
6. New features can be added quickly.
7. All input feature code is tested.



Source: https://research.google/pubs/pub46555/

Model Score – Model Tests

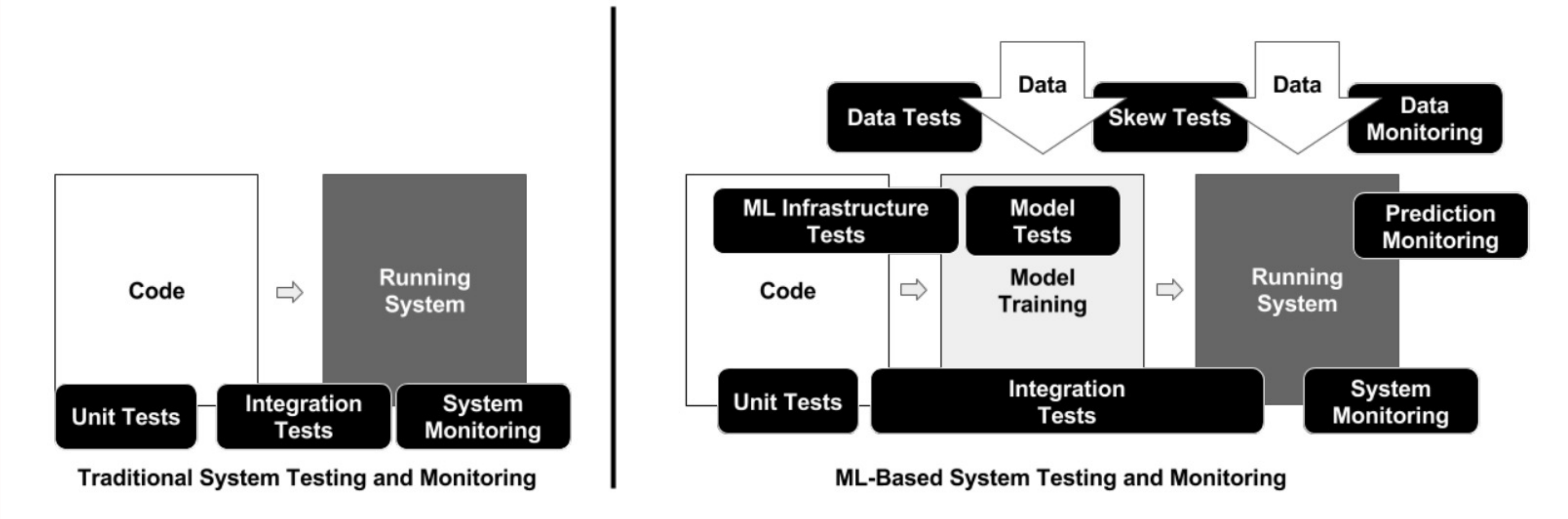
1. Model specs are reviewed and submitted.
2. Offline and online metrics correlate.
3. All hyperparameters have been tuned.
4. The impact of model staleness is known.
5. A simpler model is not better.
6. Model quality is sufficient on important data slices.
7. The model is tested for considerations of inclusion.



Source: https://research.google/pubs/pub46555/

Model Score – ML Infrastructure Tests

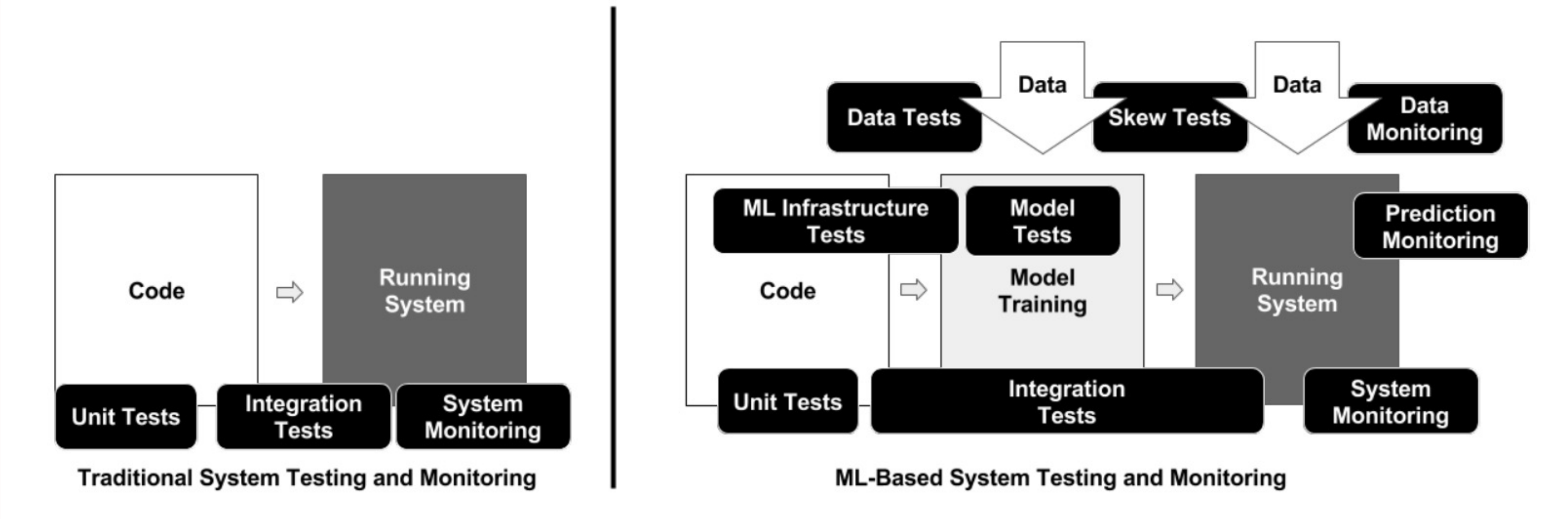
1. Training is reproducible.
2. Model specs are unit tested.
3. The ML pipeline is Integration tested.
4. Model quality is validated before serving.
5. The model is debuggable.
6. Models are canaried before serving.
7. Serving models can be rolled back.



Source: https://research.google/pubs/pub46555/

Model Score – Monitoring Tests

1. Dependency changes result in notification.
2. Data invariants hold for inputs.
3. Training and serving are not skewed.
4. Models are not too stale.
5. Models are numerically stable.
6. Computing performance has not regressed.
7. Prediction quality has not regressed.



Source: https://research.google/pubs/pub46555/

Linting and Styling Revisited

- Choose a linter (e.g. pylint) and set up configuration in pyproject.toml

- Choose a code style and corresponding formatter (e.g. black and isort), and set up configuration in pyproject.toml

```
pyproject.toml

[tool.pylint.messages_control]
disable = [
"missing-final-newline",
"missing-function-docstring”,
. . .
]

[tool.black]
line-length = 100
target-version = ['py39']
skip-string-normalization = true

[tool.isort]
length_sort = true
```

Automation with Git Pre-Commit Hooks

- Good for simple tests before committing code to git repo
- `pip install pre-commit`
- Check .git/hooks/
- Add these:
  - python black
  - pylint
  - isort
  - simple unit tests

.pre-commit-config.yaml

```yaml
repos:
  - repo: https://github.com/ambv/black
    rev: 22.3.0
    hooks:
      - id: black
        args: [--diff, --check]

  - repo: local
    hooks:
      - id: pylint
        name: pylint
        entry: pylint
        language: system
        types: [python]
        require_serial: true
```

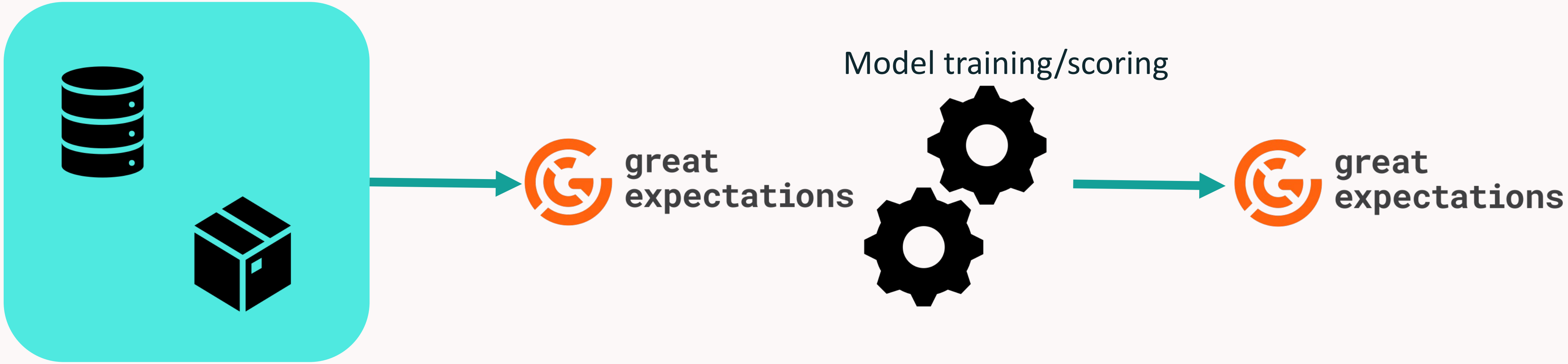
Code Tests

- Use pytest for testing code with unit tests
  - Does your data ingest/processing code produce expected features?
  - Does your ML training code produce expected models?
  - Does your ML scoring/serving code produce expected scores?
- Use doctests for testing code ***in documentation***
- In Notebooks:
  - Add assert statements
  - Run using nbconvert

```
jupyter nbconvert --to notebook --execute mynotebook.ipynb
```

Data Tests

- Use Great Expectations or an alternative for data tests
- Keep tests relevant – what are the most likely causes of errors?
  - E.g. Shape of data, introduction of NaNs
- Set up alerts



Model Training Tests

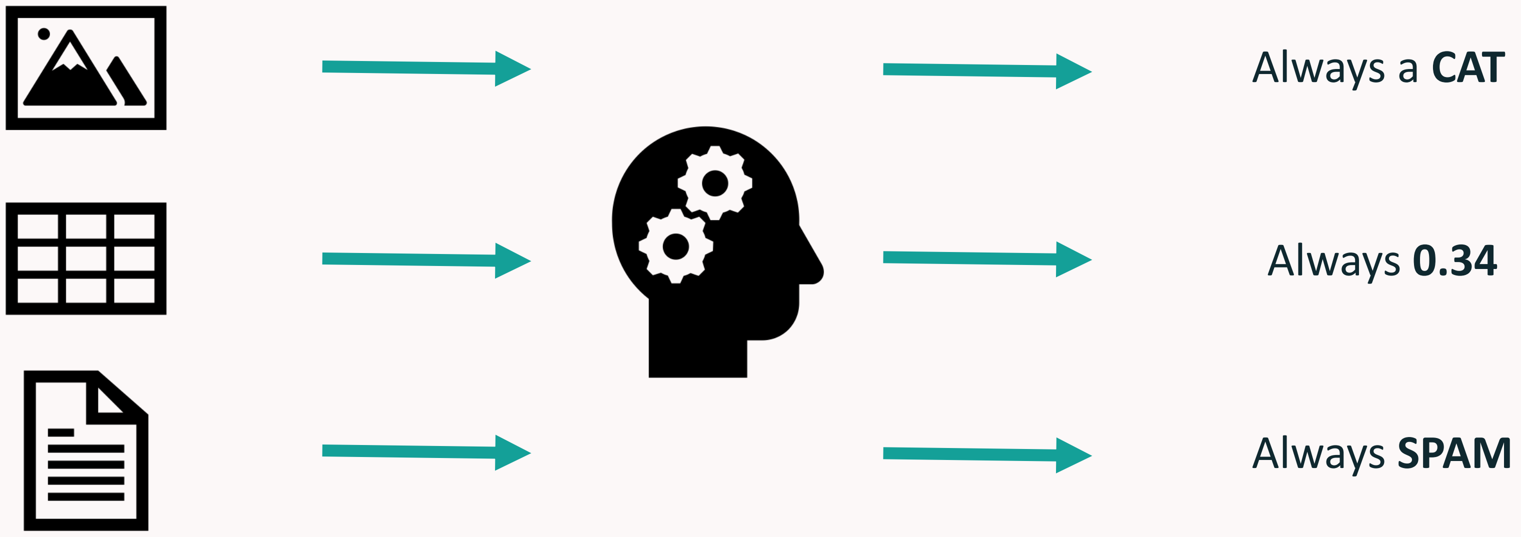
Mainly used for novel models/algorithms

- Run a memorization test – if model is training correctly, it should memorize your training set if you overfit it enough
- Train for a few iterations and check that the loss is decreasing
- Run old training jobs (old training data) with the new training pipeline

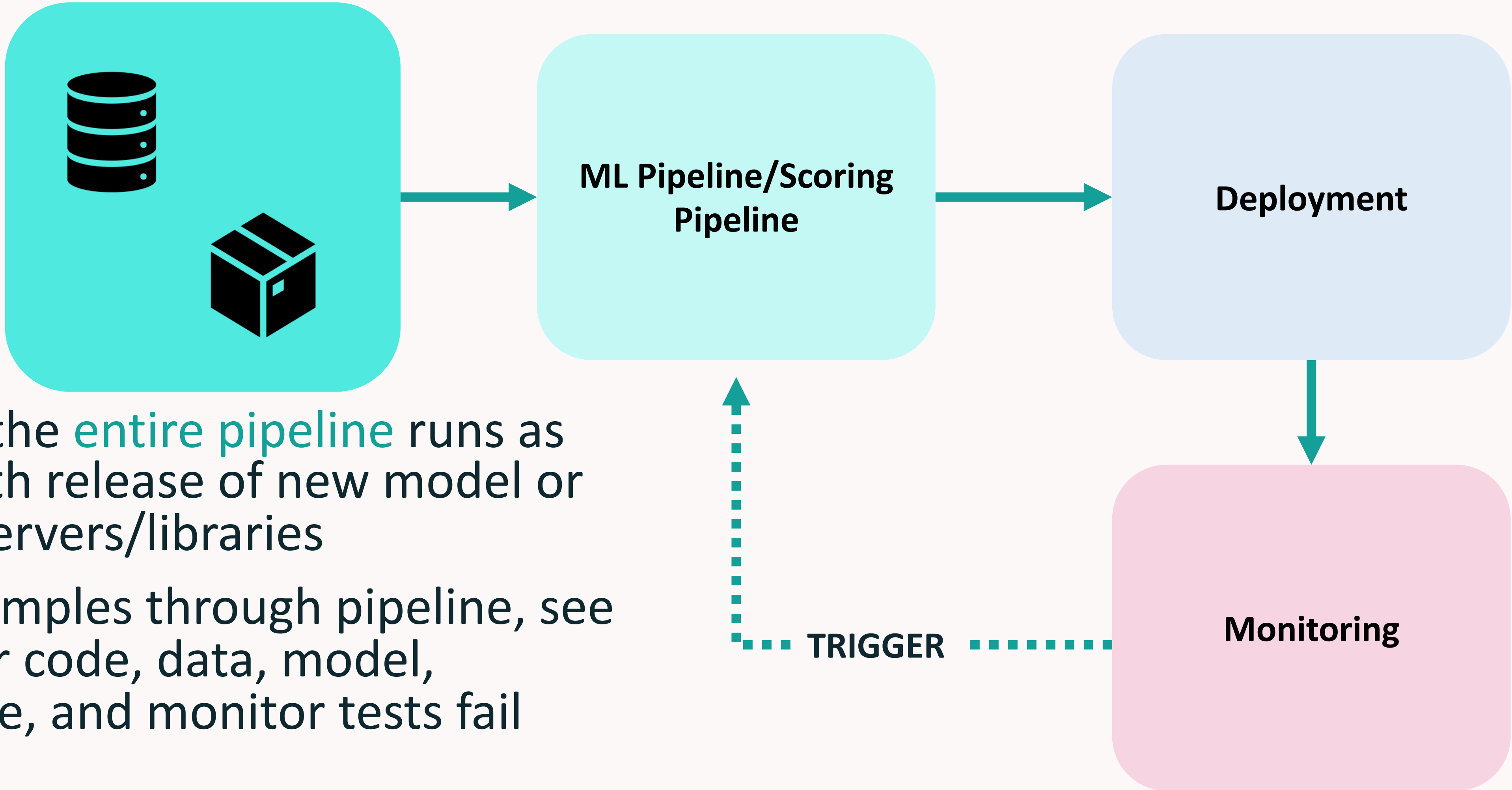
ML Pipeline

Regression Testing for Models

- Models are essentially ***functions*** with inputs and outputs
- Write tests to ensure that, given the same input, model produces the same output



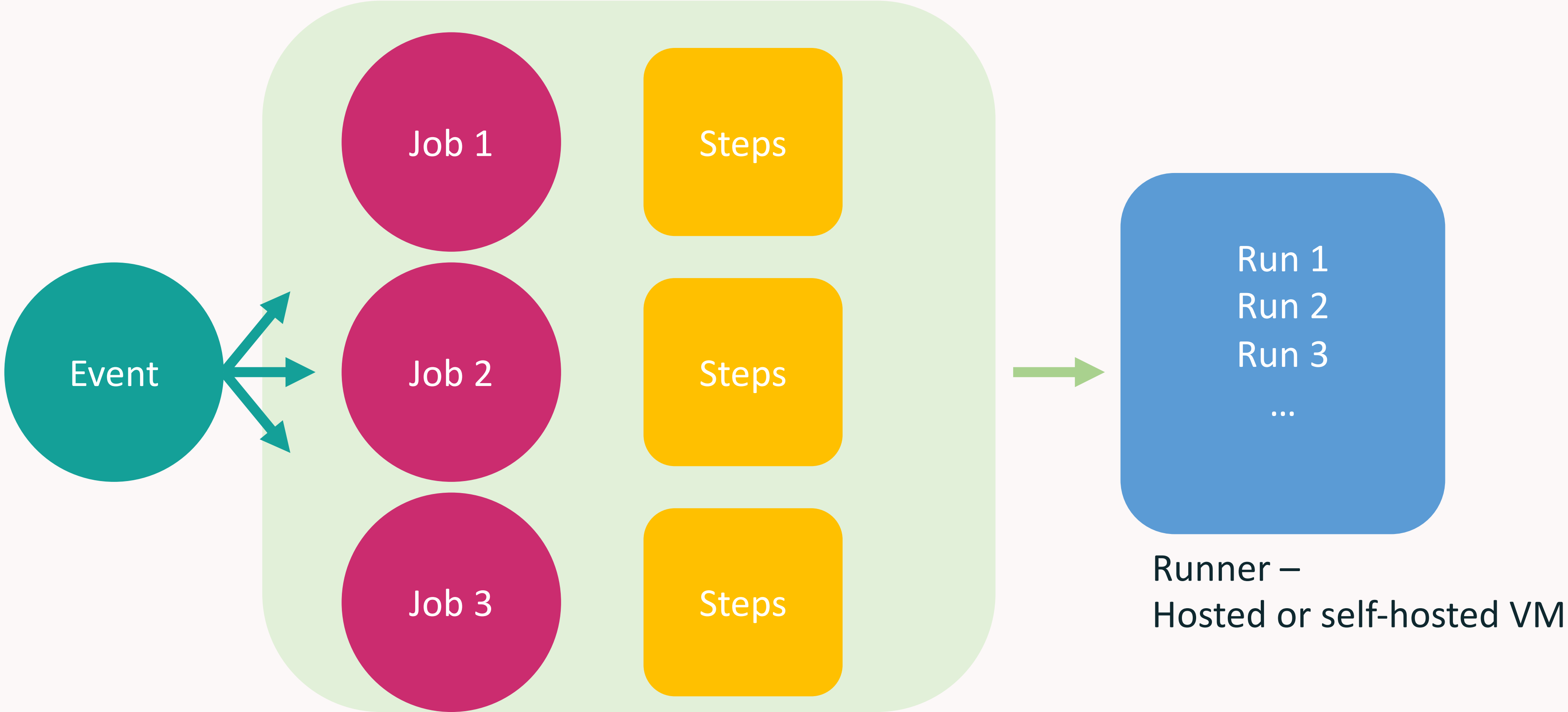
Integration Testing



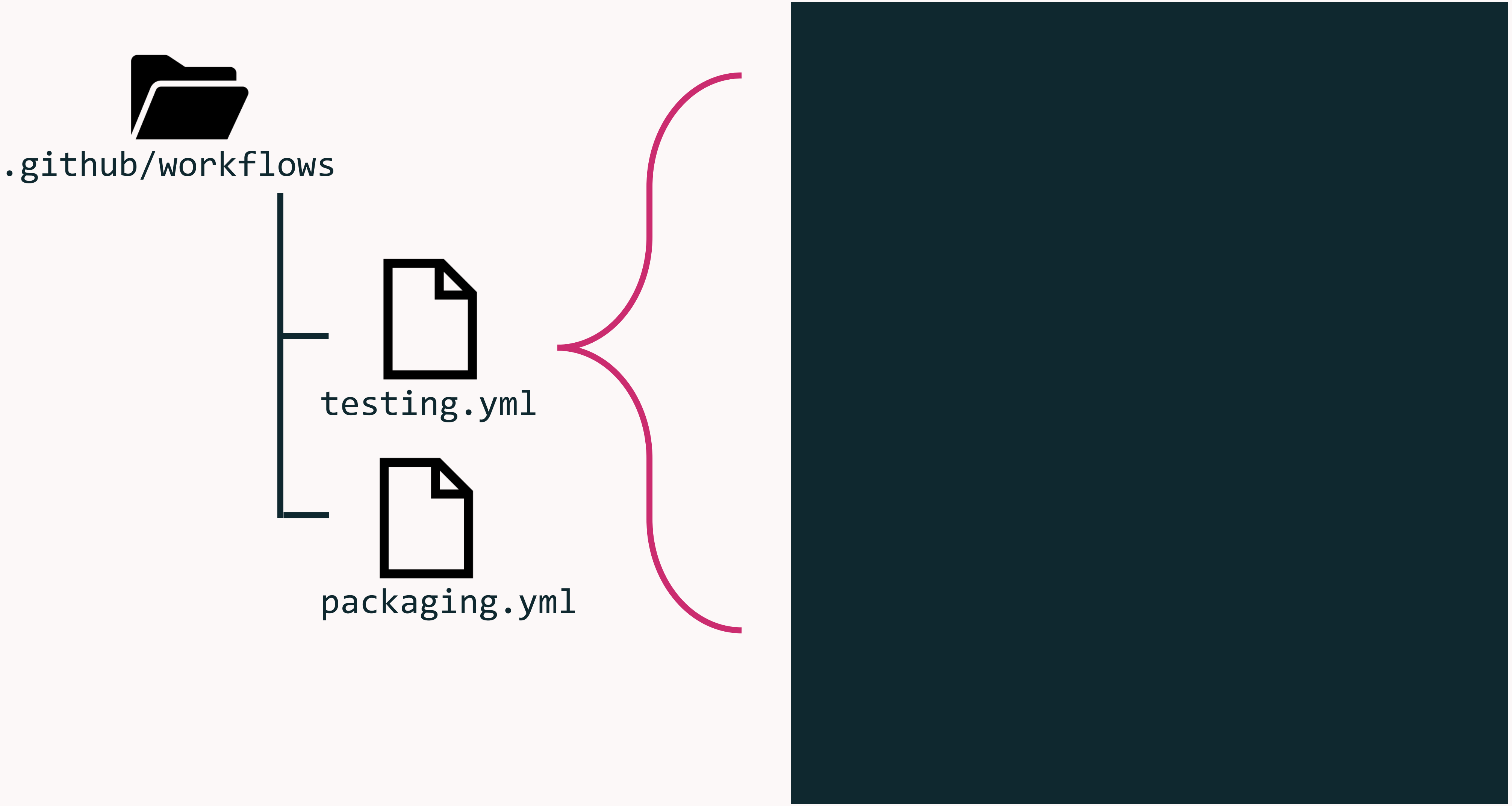
- Ensure that the entire pipeline runs as expected with release of new model or changes in servers/libraries
- Send data samples through pipeline, see if any of your code, data, model, infrastructure, and monitor tests fail

Automation with Github Actions

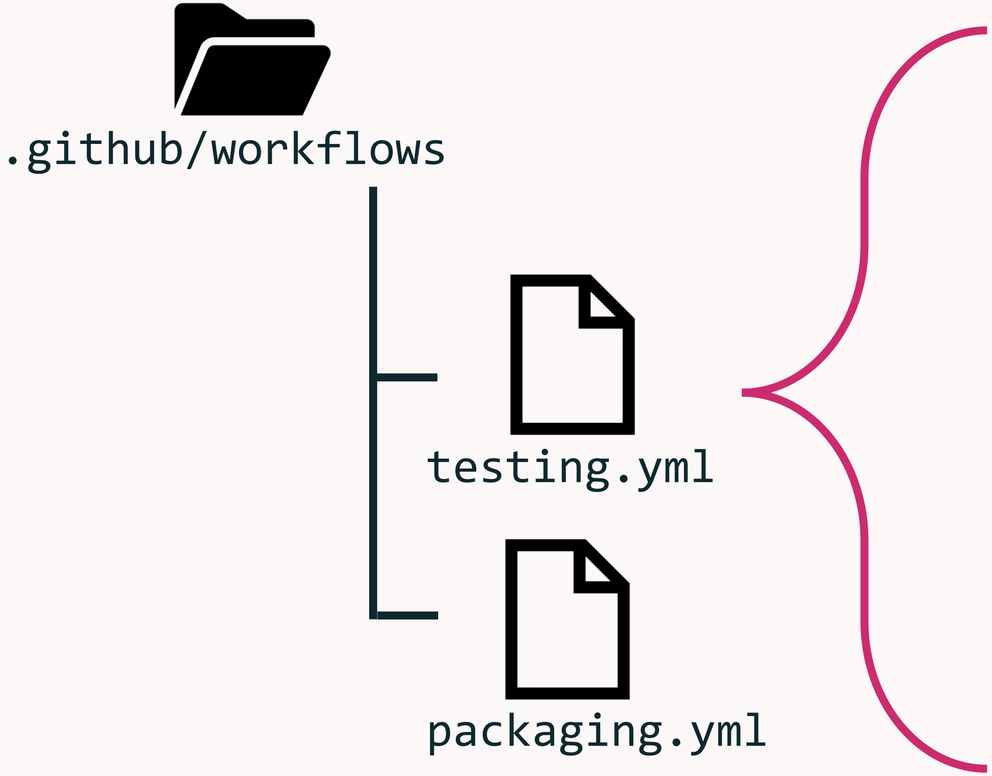
- Use Github Actions to enable event-triggered workflows to run jobs consisting of steps



Automation with Github Actions



Automation with Github Actions



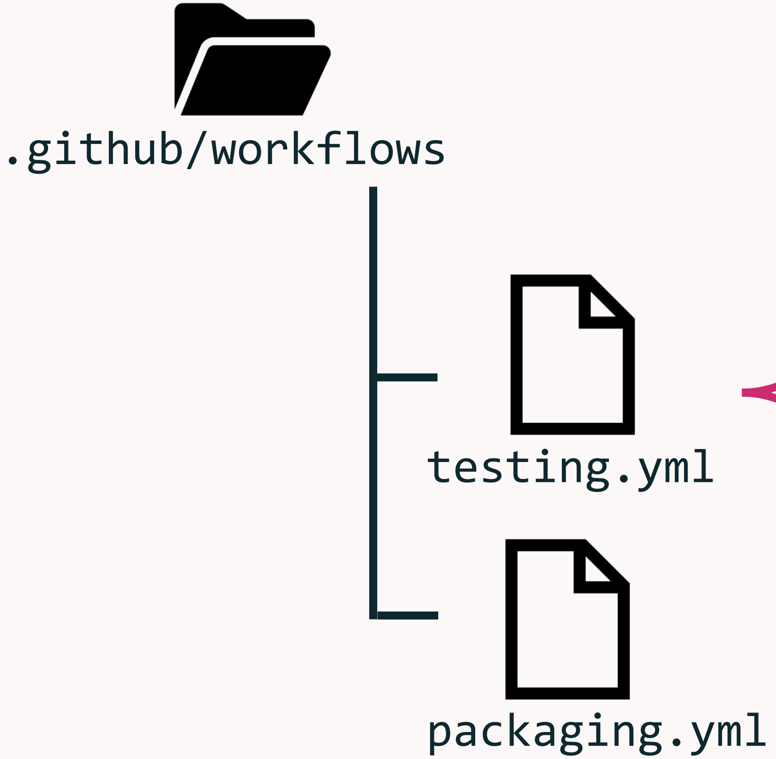

```yaml
on:
  push:
    branches:
    - main
    - master
```

Other common options:

- pull_request
- schedule

All options here

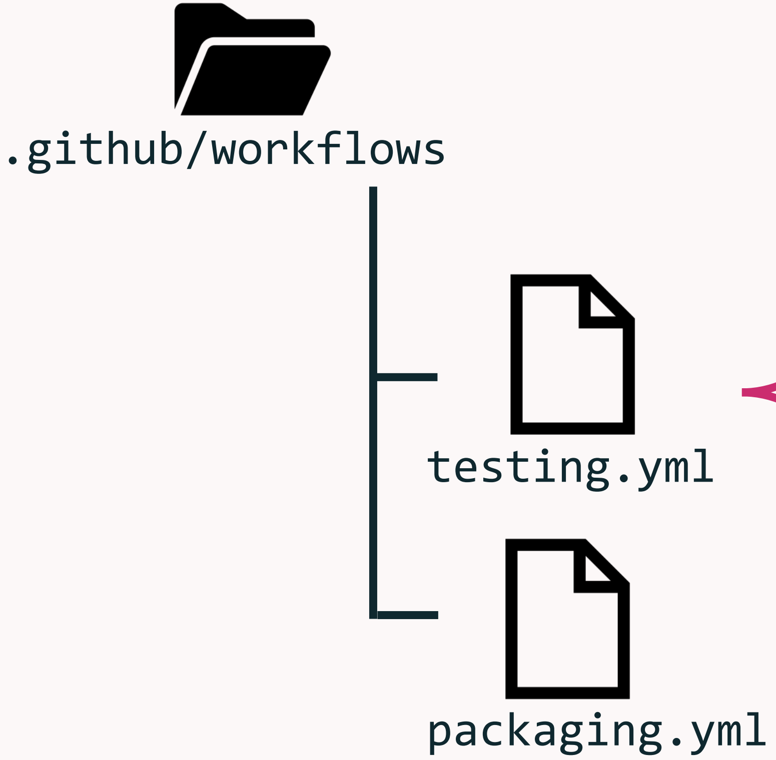
Automation with Github Actions




```yaml
on:
  push:
    branches:
    - main
    - master
jobs:
  test-code:
    runs-on: ubuntu-latest
```

Jobs will run in parallel

Automation with Github Actions




```yaml
on:
  push:
    branches:
    - main
    - master
jobs:
  test-code:
    runs-on: ubuntu-latest
    steps:
      - name: Checkout repo
        uses: actions/checkout@v2
      - name: Set up Python
        uses: actions/setup-python@v2
        with:
          python-version: 3.9.1
      - name: Install dependencies
        run: |
          python3 -m pip install …
      - name: Execute tests
        run: pytest tests
```

Note: Github may not have access to data and model registries, so not all tests can be run.

Automation with Github Actions

- Github Actions marketplace has actions for:
  - Collecting results from your experiment tracking server
  - Containerizing your ml pipeline
  - Deploying your container to a cloud service
  - Creating Argo Workflows
- Use actions/cache@v2 to cache dependencies
- Use secrets for reusable configuration data
- Set branch protection rules (Settings -> Branches) to avoid merging branches before tests are passed
- Set environment variables in yml file using env :

Alternatives to Github Actions

[CircleCI]

[Jenkins]

CI/CD/CT Demo

Demo

- Git pre-commit hooks for linting (pylint), styling (black and isort)
- 2 source files: training flow; scoring flow
- Tests:
    - Training flow: test input data is correct; test model performs better than some threshold on test set
    - Scoring flow: test whether a batch of input results in a proper prediction
- Github Actions:
    - 2 workflows: tests; deployment
    - Tests: run training tests; run scoring tests
    - Deployment: dockerize