Machine Learning in Production

Automating and Testing ML Pipelines

Infrastructure Quality...

Fundamentals of Engineering AI-Enabled Systems

Holistic system view: AI and non-AI components, pipelines, stakeholders, environment interactions, feedback loops

Requirements:
System and model goals
User requirements
Environment assumptions
Quality beyond accuracy
Measurement
Risk analysis
Planning for mistakes

Architecture + design:
Modeling tradeoffs
Deployment architecture
Data science pipelines
Telemetry, monitoring
Anticipating evolution
Big data processing
Human-AI design

Quality assurance:
Model testing
Data quality
QA automation
Testing in production
Infrastructure quality
Debugging

Operations:
Continuous deployment
Contin. experimentation
Configuration mgmt.
Monitoring
Versioning
Big data
DevOps, MLOps

Teams and process: Data science vs software eng. workflows, interdisciplinary teams, collaboration points, technical debt

Responsible AI Engineering

Provenance, versioning, reproducibility

Safety

Security and privacy

Fairness

Interpretability and explainability

Transparency and trust

Ethics, governance, regulation, compliance, organizational culture

Readings

Required reading: Eric Breck, Shanqing Cai, Eric Nielsen, Michael Salib, D. Sculley. The ML Test Score: A Rubric for ML Production Readiness and Technical Debt Reduction. Proceedings of IEEE Big Data (2017)

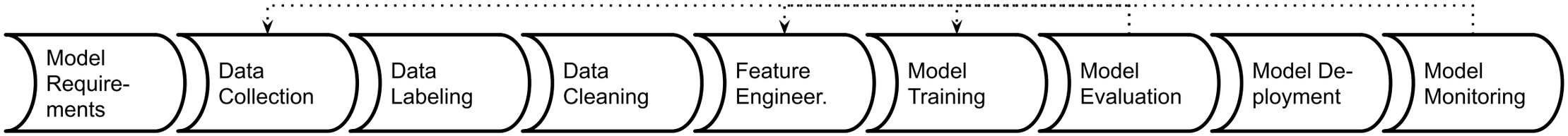
Recommended readings:

- O'Leary, Katie, and Makoto Uchida. "Common problems with Creating Machine Learning Pipelines from Existing Code." Proc. Conference on Machine Learning and Systems (MLSys) (2020).

Learning Goals

- Decompose an ML pipeline into testable functions
- Implement and automate tests for all parts of the ML pipeline
- Understand testing opportunities beyond functional correctness
- Describe the different testing levels and testing opportunities at each level
- Automate test execution with continuous integration

ML Pipelines



All steps to create (and deploy) the model

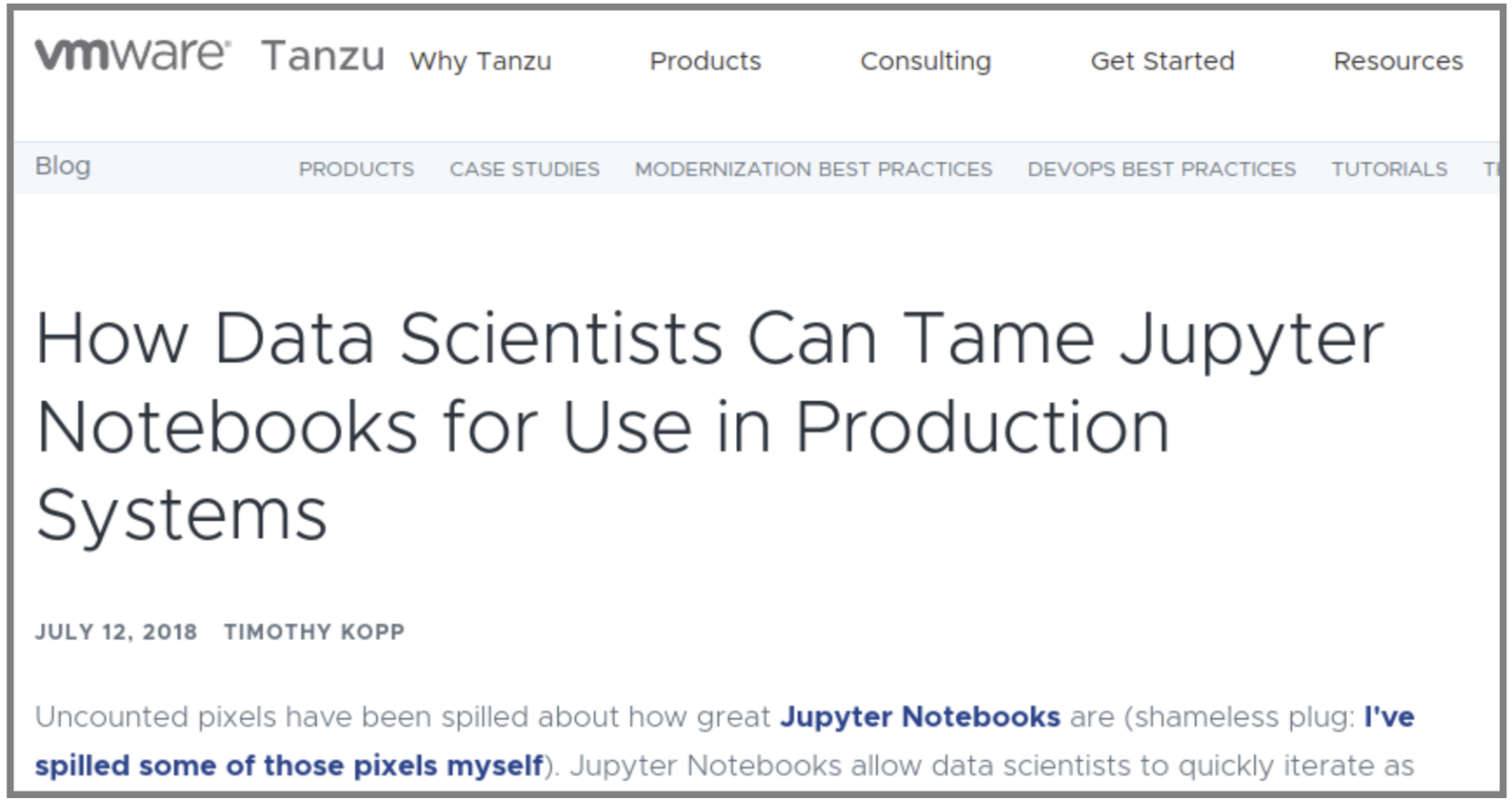
Common ML Pipeline

Speaker notes

Computational notebook

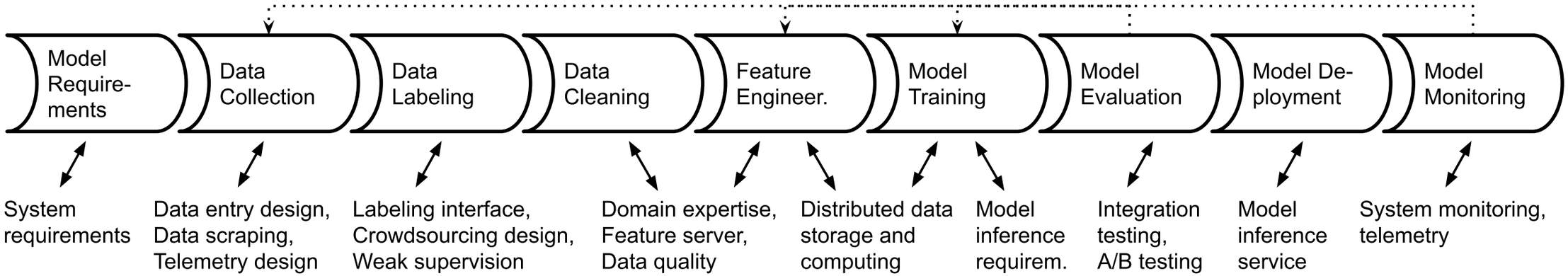
Containing all code, often also dead experimental code

Notebooks as Production Pipeline?



- Parameterize and use `nbconvert`?

Real Pipelines can be Complex



Real Pipelines can be Complex

Large arguments of data

Distributed data storage

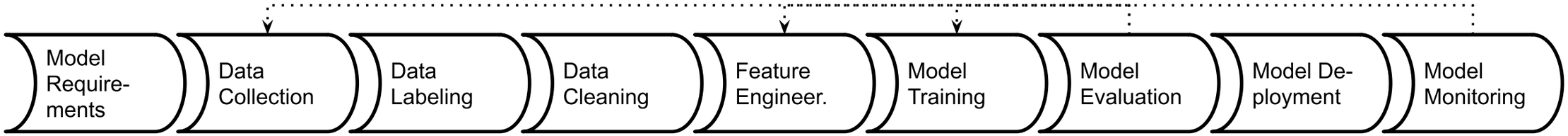
Distributed processing and learning

Special hardware needs

Fault tolerance

Humans in the loop

Possible Mistakes in ML Pipelines



Danger of "silent" mistakes in many phases

Examples?

Possible Mistakes in ML Pipelines

Danger of "silent" mistakes in many phases:

- Dropped data after format changes
- Failure to push updated model into production
- Incorrect feature extraction
- Use of stale dataset, wrong data source
- Data source no longer available (e.g web API)
- Telemetry server overloaded
- Negative feedback (telemtr.) no longer sent from app
- Use of old model learning code, stale hyperparameter
- Data format changes between ML pipeline steps

Pipeline Thinking

After exploration and prototyping build robust pipeline

One-off model creation -> repeatable automateable process

Enables updates, supports experimentation

Explicit interfaces with other parts of the system (data sources, labeling infrastructure, training infrastructure, deployment, ...)

Design for change

Building Robust Pipeline Automation

- Support experimentation and evolution
  - Automate
  - Design for change
  - Design for observability
  - Testing the pipeline for robustness
- Thinking in pipelines, not models
- Integrating the Pipeline with other Components

Pipeline Testability and Modularity

Pipelines are Code

From experimental notebook code to production code

Each stage as a function or module

Well tested in isolation and together

Robust to changes in inputs (automatically adapt or crash, no silent mistakes)

Use good engineering practices (version control, documentation, testing, naming, code review)

Sequential Data Science Code in Notebooks

```
# typical data science code from a notebook
df = pd.read_csv('data.csv', parse_dates=True)


# data cleaning
# ...


# feature engineering
df['month'] = pd.to_datetime(df['datetime']).dt.month
df['dayofweek']= pd.to_datetime(df['datetime']).dt.dayofweek
df['delivery_count'] = boxcox(df['delivery_count'], 0.4)
df.drop(['datetime'], axis=1, inplace=True)
```

How to test??

Pipeline restructed into separate function

```
def encode_day_of_week(df):
    if 'datetime' not in df.columns: raise ValueError("Column datetime missing")
    if df.datetime.dtype != 'object': raise ValueError("Invalid type for column datetime")
    df['dayofweek']= pd.to_datetime(df['datetime']).dt.day_name()
    df = pd.get_dummies(df, columns = ['dayofweek'])
    return df


# ...


def prepare_data(df):
    df = clean_data(df)
```

Orchestrating Functions

```
def pipeline():
    train = pd.read_csv('train.csv', parse_dates=True)
    test = pd.read_csv('test.csv', parse_dates=True)
    X_train, y_train = prepare_data(train)
    X_test, y_test = prepare_data(test)
    model = learn(X_train, y_train)
    accuracy = eval(model, X_test, y_test)
    return model, accuracy
```

Dataflow frameworks like Luigi, DVC, Airflow, d6tflow, and Ploomber support distribution, fault tolerance, monitoring, ...

Hosted versions like DataBricks and AWS SageMaker Pipelines

Test the Modules

```
def encode_day_of_week(df):
    if 'datetime' not in df.columns: raise ValueError("Column d
    if df.datetime.dtype != 'object': raise ValueError("Invalid
    df['dayofweek']= pd.to_datetime(df['datetime']).dt.day_name
    df = pd.get_dummies(df, columns = ['dayofweek'])
    return df
```

```
def test_day_of_week_encoding():
  df = pd.DataFrame({'datetime': ['2020-01-01','2020-01-02','2
  encoded = encode_day_of_week(df)
  assert "dayofweek_Wednesday" in encoded.columns
  assert (encoded["dayofweek_Wednesday"] == [1, 0, 1]).all()
```

Subtle Bugs in Data Wrangling Code

```
df['Join_year'] = df.Joined.dropna().map(
    lambda x: x.split(',')[1].split(' ')[1])
```

```
df.loc[idx_nan_age,'Age'].loc[idx_nan_age] =
    df['Title'].loc[idx_nan_age].map(map_means)
```

```
df["Weight"].astype(str).astype(int)
```

Subtle Bugs in Data Wrangling Code (continued)

```
df['Reviws'] = df['Reviews'].apply(int)
```

```
df["Release Clause"] =
    df["Release Clause"].replace(regex=['k'], value='000')
df["Release Clause"] =
    df["Release Clause"].astype(str).astype(float)
```

Speaker notes

1 attempting to remove na values from column, not table

2 loc[] called twice, resulting in assignment to temporary column only

3 astype() is not an in-place operation

4 typo in column name

5&6 modeling problem (k vs K)

Modularity fosters Testability

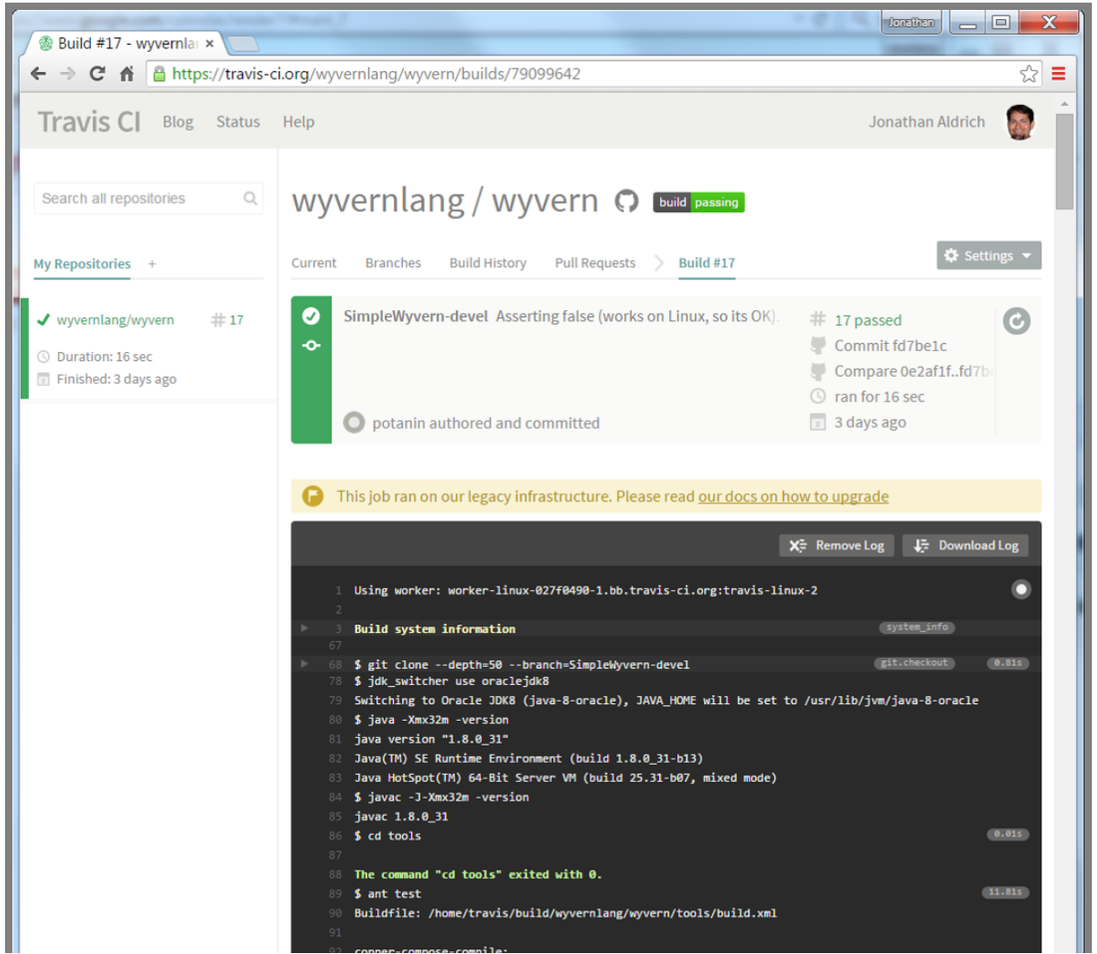
Breaking code into functions/modules

Supports reuse, separate development, and testing

Can test individual parts

Excursion: Test Automation

From Manual Testing to Continuous Integration



Anatomy of a Unit Test

```
import org.junit.Test;
import static org.junit.Assert.assertEquals;

public class AdjacencyListTest {
    @Test
    public void testSanityTest(){
        // set up
        Graph g1 = new AdjacencyListGraph(10);
        Vertex s1 = new Vertex("A");
        Vertex s2 = new Vertex("B");
        // check expected results (oracle)
```

Ingredients to a Test

Specification

Controlled environment

Test inputs (calls and parameters)

Expected outputs/behavior (oracle)

Unit Testing Pitfalls

Working code, failing tests

"Works on my machine"

Tests break frequently

How to avoid?

Testable Code

Think about testing when writing code

Unit testing encourages you to write testable code

Separate parts of the code to make them independently testable

Abstract functionality behind interface, make it replaceable

Bonus: Test-Driven Development is a design and development method in which you *always* write tests *before* writing code

Build systems & Continuous Integration

Automate all build, analysis, test, and deployment steps from a command line call

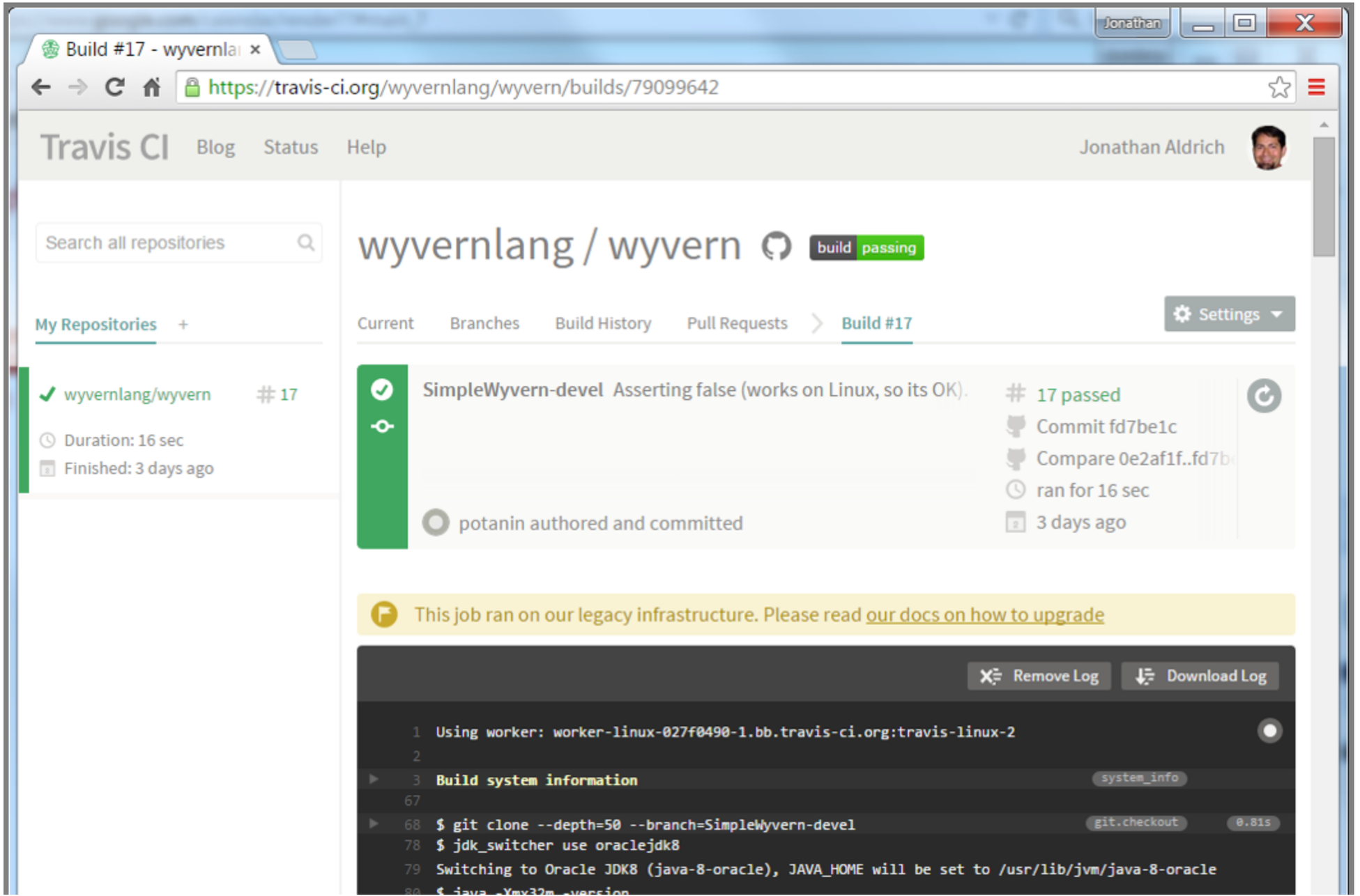
Ensure all dependencies and configurations are defined

Ideally reproducible and incremental

Distribute work for large jobs

Track results

Key CI benefit: Tests are regularly executed, part of process

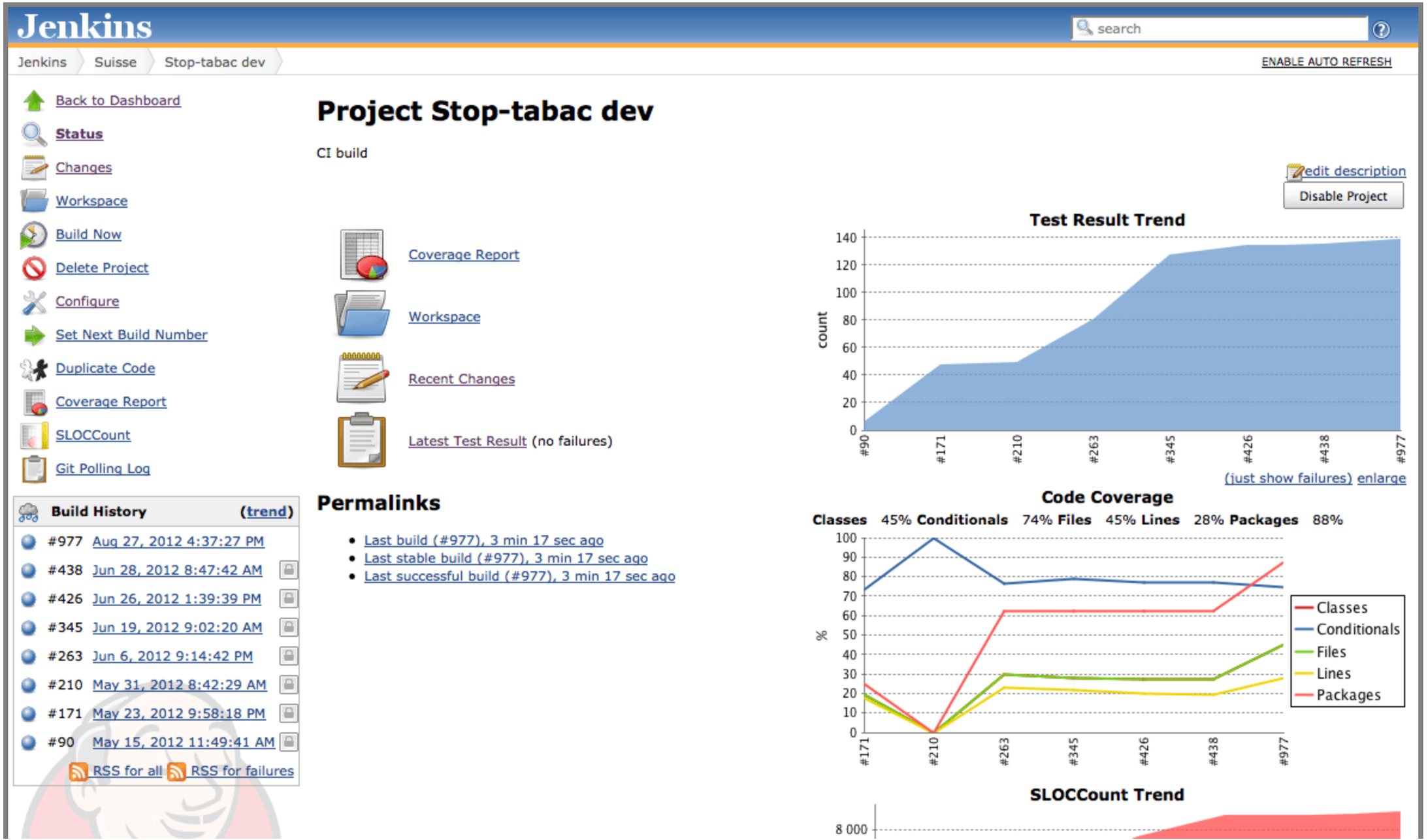

Build #17 - wyvernla
https://travis-ci.org/wyvernlang/wyvern/builds/79099642
Travis CI
Blog
Status
Help
Jonathan Aldrich
Search all repositories
wyvernlang / wyvern
build passing
My Repositories
wyvernlang/wyvern
17
Duration: 16 sec
Finished: 3 days ago
Current
Branches
Build History
Pull Requests
Build #17
Settings
SimpleWyvern-devel Asserting false (works on Linux, so its OK).
17 passed
Commit fd7be1c
Compare 0e2af1f..fd7b
ran for 16 sec
3 days ago
potanin authored and committed
This job ran on our legacy infrastructure. Please read our docs on how to upgrade
Remove Log
Download Log
1 Using worker: worker-linux-027f0490-1.bb.travis-ci.org:travis-linux-2
3 Build system information
system_info
68 $ git clone --depth=50 --branch=SimpleWyvern-devel
git.checkout
0.81s
78 $ jdk_switcher use oraclejdk8
79 Switching to Oracle JDK8 (java-8-oracle), JAVA_HOME will be set to /usr/lib/jvm/java-8-oracle

Tracking Build Quality

Track quality indicators over time, e.g.,

- Build time
- Coverage
- Static analysis warnings
- Performance results
- Model quality measures
- Number of TODOs in source code

Coverage


Jenkins
search
Jenkins
Suisse
Stop-tabac dev
ENABLE AUTO REFRESH
Back to Dashboard
Status
Changes
Workspace
Build Now
Delete Project
Configure
Set Next Build Number
Duplicate Code
Coverage Report
SLOCCount
Git Polling Log
Build History
(trend)
#977 Aug 27, 2012 4:37:27 PM
#438 Jun 28, 2012 8:47:42 AM
#426 Jun 26, 2012 1:39:39 PM
#345 Jun 19, 2012 9:02:20 AM
#263 Jun 6, 2012 9:14:42 PM
#210 May 31, 2012 8:42:29 AM
#171 May 23, 2012 9:58:18 PM
#90 May 15, 2012 11:49:41 AM
RSS for all
RSS for failures
Project Stop-tabac dev
CI build
edit description
Disable Project
Coverage Report
Workspace
Recent Changes
Latest Test Result (no failures)
Permalinks
Last build (#977), 3 min 17 sec ago
Last stable build (#977), 3 min 17 sec ago
Last successful build (#977), 3 min 17 sec ago
Test Result Trend
count
(just show failures) enlarge
Code Coverage
Classes 45% Conditionals 74% Files 45% Lines 28% Packages 88%
%
Classes
Conditionals
Files
Lines
Packages
SLOCCount Trend

Tracking Model Qualities

Many tools: MLFlow, ModelDB, Neptune, TensorBoard, Weights & Biases, Comet.ml, ...

ModelDB Example

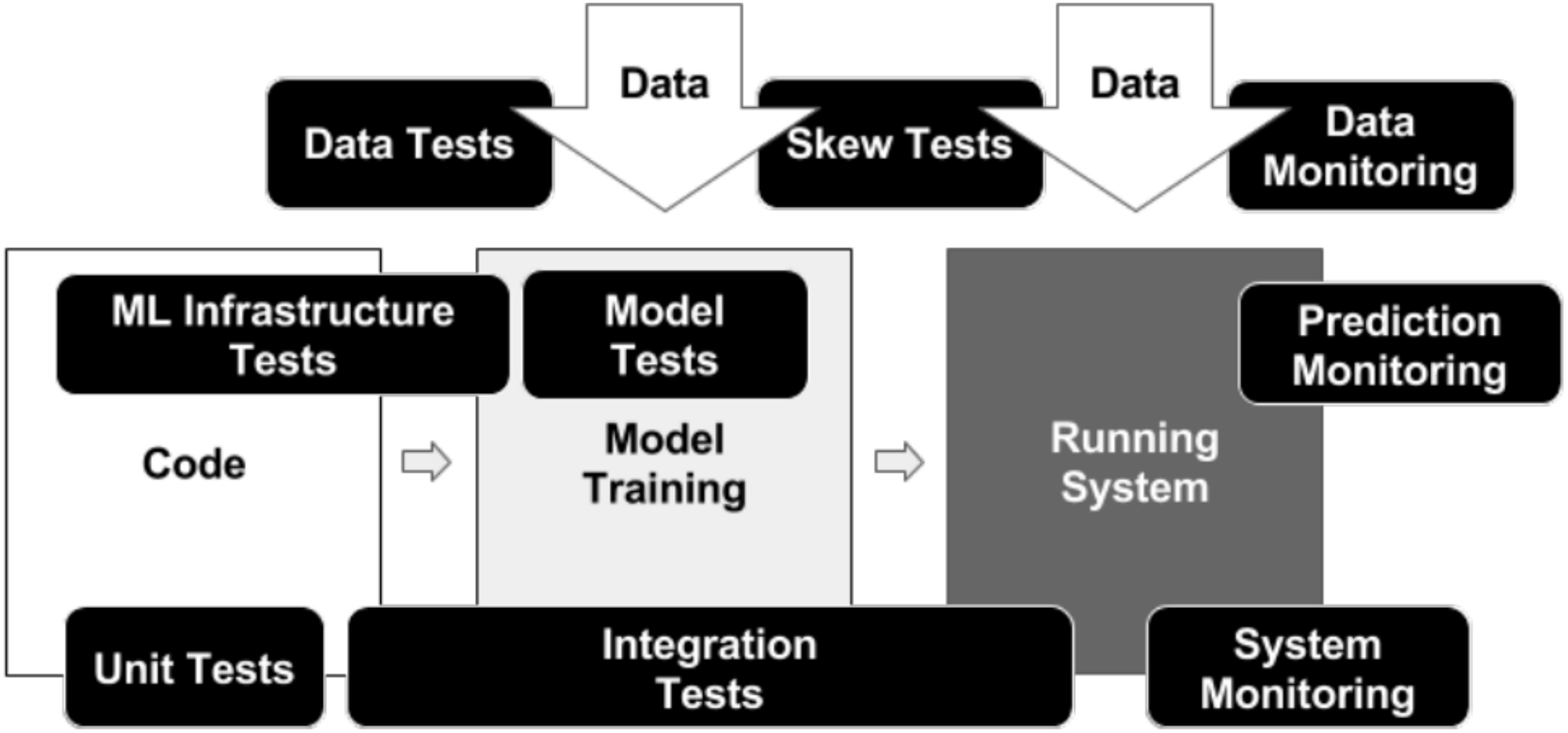
```
from verta import Client
client = Client("http://localhost:3000")

proj = client.set_project("My first ModelDB project")
expt = client.set_experiment("Default Experiment")

# log a training run
run = client.set_experiment_run("First Run")
run.log_hyperparameters({"regularization" : 0.5})
model1 = # ... model training code goes here
run.log_metric('accuracy', accuracy(model1, validationData))
```

Testing Maturity

Eric Breck, Shanqing Cai, Eric Nielsen, Michael Salib, D. Sculley. The ML Test Score: A Rubric for ML Production Readiness and Technical Debt Reduction. Proceedings of IEEE Big Data (2017)



Source: Eric Breck, Shanqing Cai, Eric Nielsen, Michael Salib, D. Sculley. The ML Test Score: A Rubric for ML Production Readiness and Technical Debt Reduction. Proceedings of IEEE Big Data (2017)

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.

Eric Breck, Shanqing Cai, Eric Nielsen, Michael Salib, D. Sculley. The ML Test Score: A Rubric for ML Production Readiness and Technical Debt Reduction. Proceedings of IEEE Big Data (2017)

Tests for Model Development

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.

Eric Breck, Shanqing Cai, Eric Nielsen, Michael Salib, D. Sculley. The ML Test Score: A Rubric for ML Production Readiness and Technical Debt Reduction. Proceedings of IEEE Big Data (2017)

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.

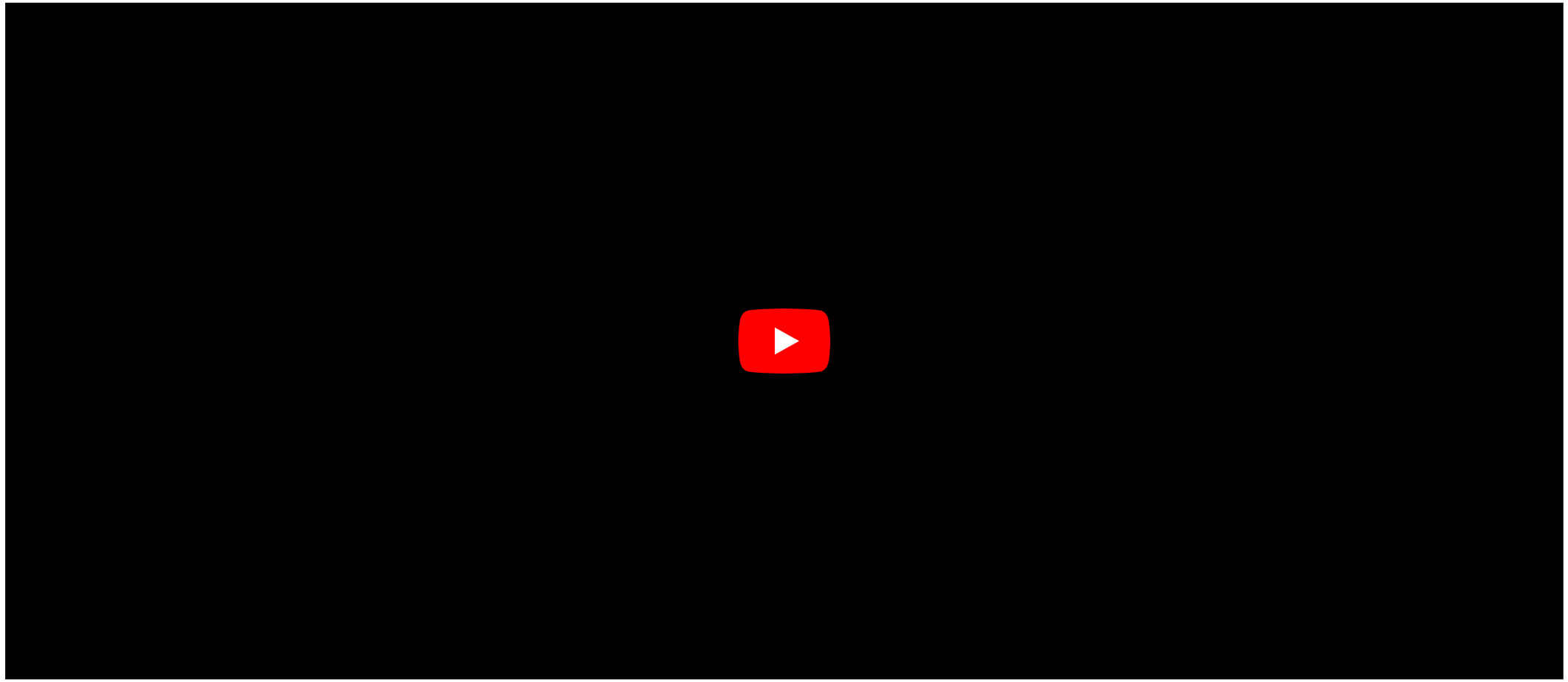
Eric Breck, Shanqing Cai, Eric Nielsen, Michael Salib, D. Sculley. The ML Test Score: A Rubric for ML Production Readiness and Technical Debt Reduction. Proceedings of IEEE Big Data (2017)

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.

Eric Breck, Shanqing Cai, Eric Nielsen, Michael Salib, D. Sculley. The ML Test Score: A Rubric for ML Production Readiness and Technical Debt Reduction. Proceedings of IEEE Big Data (2017)

Case Study: Covid-19 Detection



(from S20 midterm; assume cloud or hybrid deployment)

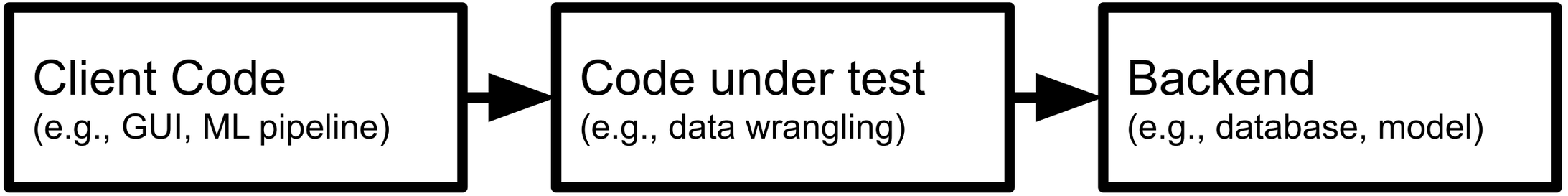
Breakout Groups

- In the Smartphone Covid Detection scenario
- Discuss in groups:
  - Back left: data tests
  - Back right: model dev. tests
  - Front right: infrastructure tests
  - Front left: monitoring tests
- For 8 min, discuss some of the listed point in the context of the Covid-detection scenario: what would you do?
- In `#lecture`, tagging group members, suggest what top 2 tests to implement and how

Minimizing and Stubbing Dependencies

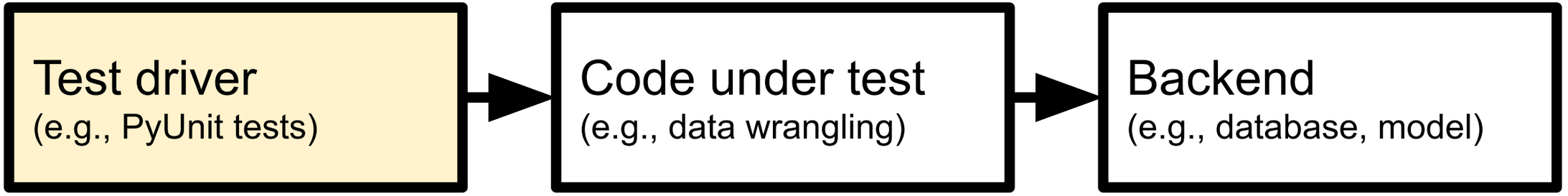
How to unit test component with dependency on other code?

How to Test Parts of a System?




```
# original implementation hardcodes external API
def clean_gender(df):
  def clean(row):
    if pd.isnull(row['gender']):
      row['gender'] = gender_api_client.predict(row['firstname
    return row
  return df.apply(clean, axis=1)
```

Automating Test Execution



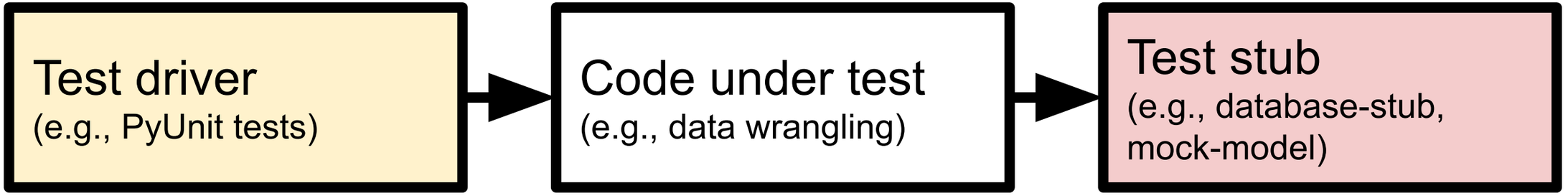

```
def test_do_not_overwrite_gender():
    df = pd.DataFrame({'firstname': ['John', 'Jane', 'Jim'],
                       'lastname': ['Doe', 'Doe', 'Doe'],
                       'location': ['Pittsburgh, PA', 'Rome, It
                       'gender': [np.nan, 'F', np.nan]})
    out = clean_gender(df, model_stub)
    assert(out['gender'] ==['M', 'F', 'M']).all()
```

Decoupling from Dependencies

```
def clean_gender(df, model):
  def clean(row):
    if pd.isnull(row['gender']):
      row['gender'] = model(row['firstname'],
                            row['lastname'],
                            row['location'])
    return row
  return df.apply(clean, axis=1)
```

Replace concrete API with an interface that caller can parameterize

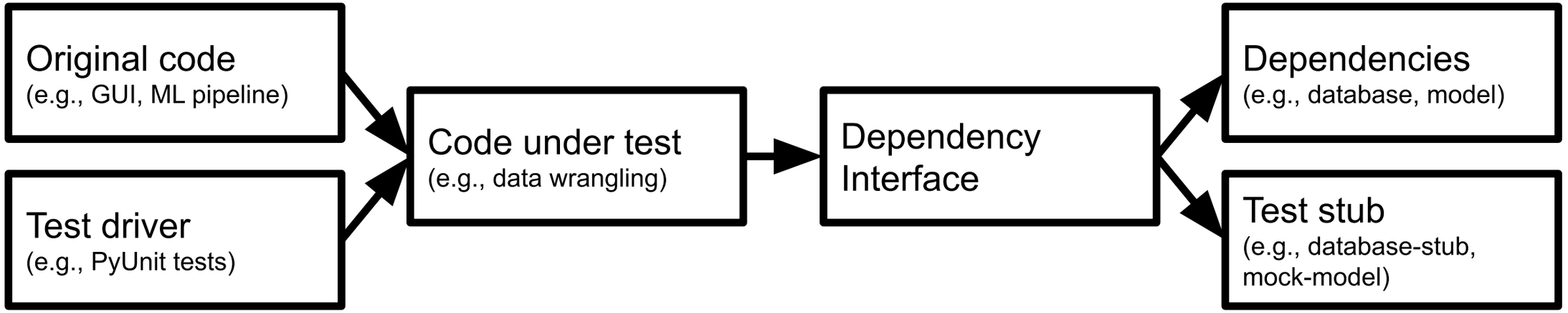
Stubbing the Dependency




```
def test_do_not_overwrite_gender():
  def model_stub(first, last, location):
    return 'M'

  df = pd.DataFrame({'firstname': ['John', 'Jane', 'Jim'], 'la
  out = clean_gender(df, model_stub)
  assert(out['gender'] ==['M', 'F', 'M']).all()
```

General Testing Strategy: Decoupling Code Under Test



(Mocking frameworks provide infrastructure for expressing such tests compactly.)

Testing Error Handling / Infrastructure Robustness

General Error Handling Strategies

Avoid silent errors

Recover locally if possible, propagate error if necessary -- fail entire task if needed

Explicitly handle exceptional conditions and mistakes

Test correct error handling

If logging only, is anybody analyzing log files?

Test for Expected Exceptions

```
def test_invalid_day_of_week_data():
  df = pd.DataFrame({'datetime_us': ['01/01/2020'],
                     'delivery_count': [1]})
  with pytest.raises(ValueError):
    encode_day_of_week(df)
```

Test for Expected Exceptions

```
def test_learning_fails_with_missing_data():
  df = pd.DataFrame({})
  with pytest.raises(NoDataError):
    learn(df)
```

Test Recovery Mechanisms with Stub

Use stubs to inject artificial faults

```
## testing retry mechanism
from retry.api import retry_call
import pytest

# stub of a network connection, sometimes failing
class FailedConnection(Connection):
  remaining_failures = 0
  def __init__(self, failures):
    self.remaining_failures = failures
  def get(self, url):
    print(self.remaining_failures)
```

Test Error Handling throughout Pipeline

Is invalid data rejected / repaired?

Are missing data updates raising errors?

Are unavailable APIs triggering errors?

Are failing deployments reported?

Log Error Occurrence

Even when reported or mitigated, log the issue

Allows later analysis of frequency and patterns

Monitoring systems can raise alarms for anomalies

Example: Error Logging

```
from prometheus_client import Counter
connection_timeout_counter = Counter(
            'connection_retry_total',
            'Retry attempts on failed connections')

class RetryLogger():
 def warning(self, fmt, error, delay):
   connection_timeout_counter.inc()

retry_logger = RetryLogger()
```

Test Monitoring

- Inject/simulate faulty behavior
- Mock out notification service used by monitoring
- Assert notification

```
class MyNotificationService extends NotificationService {
    public boolean receivedNotification = false;
    public void sendNotification(String msg) {
        receivedNotification = true; }
}
@Test void test() {
    Server s = getServer();
    MyNotificationService n = new MyNotificationService();
    Monitor m = new Monitor(s, n);
    s.stop();
    s.request(); s.request();
```

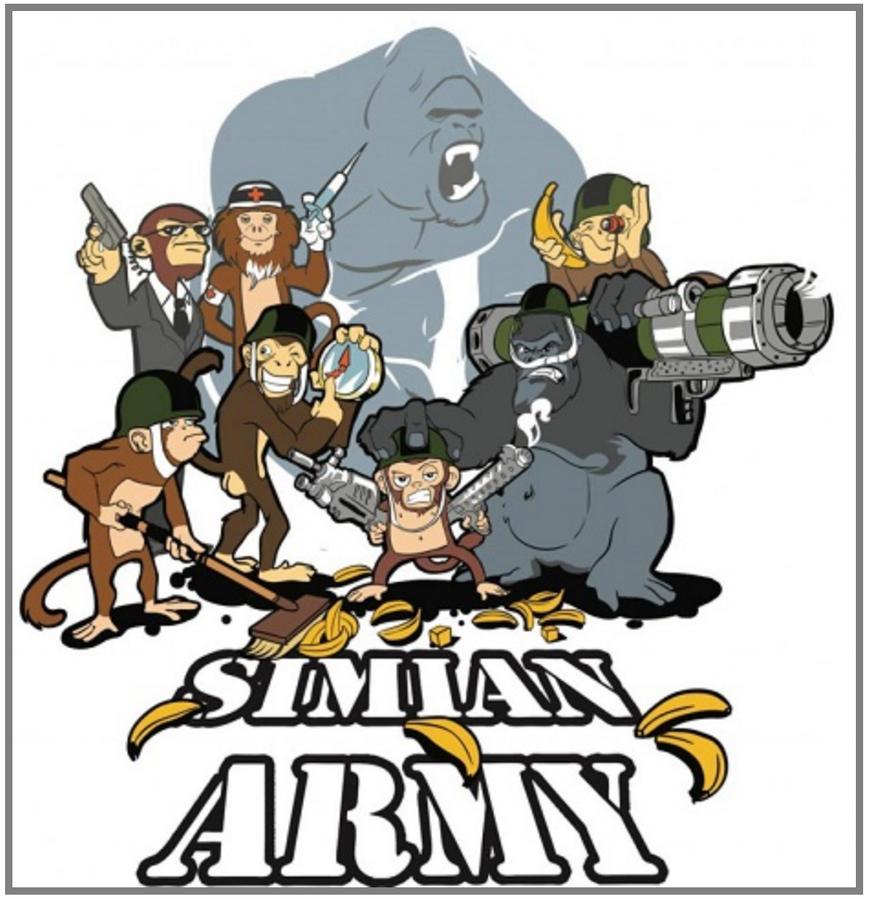
Test Monitoring in Production

Like fire drills (manual tests may be okay!)

Manual tests in production, repeat regularly

Actually take down service or trigger wrong signal to monitor

Chaos Testing



- http://principlesofchaos.org

Speaker notes

Chaos Engineering is the discipline of experimenting on a distributed system in order to build confidence in the system’s capability to withstand turbulent conditions in production. Pioneered at Netflix

Chaos Testing Argument

- Distributed systems are simply too complex to comprehensively predict
  - experiment to learn how it behaves in the presence of faults
- Base corrective actions on experimental results because they reflect real risks and actual events

- Experimentation != testing -- Observe behavior rather then expect specific results
- Simulate real-world problem in production (e.g., take down server, inject latency)
- *Minimize blast radius:* Contain experiment scope

Netflix's Simian Army

- Chaos Monkey: randomly disable production instances
- Latency Monkey: induces artificial delays in our RESTful client-server communication layer
- Conformity Monkey: finds instances that don’t adhere to best-practices and shuts them down
- Doctor Monkey: monitors external signs of health to detect unhealthy instances
- Janitor Monkey: ensures cloud environment is running free of clutter and waste
- Security Monkey: finds security violations or vulnerabilities, and terminates the offending instances
- 10–18 Monkey: detects problems in instances serving customers in multiple geographic regions
- Chaos Gorilla is similar to Chaos Monkey, but simulates an outage of an entire Amazon availability zone.

Chaos Toolkit

- Infrastructure for chaos experiments
- Driver for various infrastructure and failure cases
- Domain specific language for experiment definitions

```
{
    "version": "1.0.0",
    "title": "What is the impact of an expired certificate on
    "description": "If a certificate expires, we should gracef
    "tags": ["tls"],
    "steady-state-hypothesis": {
        "title": "Application responds",
        "probes": [
            {
                "type": "probe",
                "name": "the-astre-service-must-be-running",
```

Chaos Experiments for ML Infrastructure?

Speaker notes

Fault injection in production for testing in production. Requires monitoring and explicit experiments.

Where to Focus Testing?

Testing in ML Pipelines

Usually assume ML libraries already tested (pandas, sklearn, etc)

Focus on custom code

- data quality checks
- data wrangling (feature engineering)
- training setup
- interaction with other components

Consider tests of latency, throughput, memory, ...

Testing Data Quality Checks

Test correct detection of problems

```
def test_invalid_day_of_week_data():
   ...
```

Test correct error handling or repair of detected problems

```
def test_fill_missing_gender():
   ...
def test_exception_for_missing_data():
   ...
```

Test Data Wrangling Code

```
num = data.Size.replace(r'[kM]+$', '', regex=True).
      astype(float)
factor = data.Size.str.extract(r'[\d\.]+([KM]+)',
                               expand =False)
factor = factor.replace(['k','M'], [10**3, 10**6]).fillna(1)
data['Size'] = num*factor.astype(int)
```

```
data["Size"]= data["Size"].
      replace(regex =['k'], value='000')
data["Size"]= data["Size"].
      replace(regex =['M'], value='000000')
data["Size"]= data["Size"].astype(str). astype(float)
```

Speaker notes

both attempts are broken:

- Variant A, returns 10 for “10k”
- Variant B, returns 100.5000000 for “100.5M”

Test Model Training Setup?

Execute training with small sample data

Ensure shape of model and data as expected (e.g., tensor dimensions)

Test Interactions with Other Components

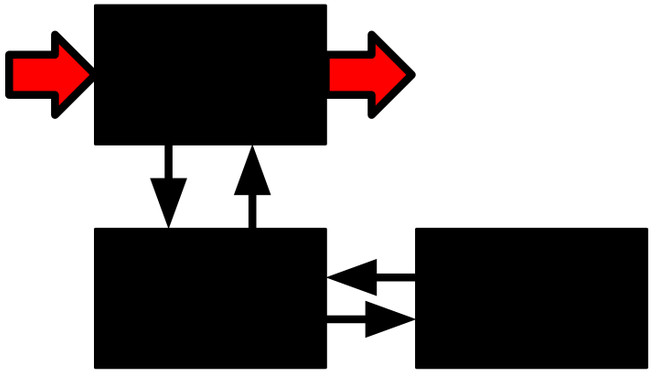
Test error handling for detecting connection/data problems

- loading training data
- feature server
- uploading serialized model
- A/B testing infrastructure

Integration and system tests

Unit test

Integration test



System Test

Speaker notes

Software is developed in units that are later assembled. Accordingly we can distinguish different levels of testing.

Unit Testing - A unit is the "smallest" piece of software that a developer creates. It is typically the work of one programmer and is stored in a single file. Different programming languages have different units: In C++ and Java the unit is the class; in C the unit is the function; in less structured languages like Basic and COBOL the unit may be the entire program.

Integration Testing - In integration we assemble units together into subsystems and finally into systems. It is possible for units to function perfectly in isolation but to fail when integrated. For example because they share an area of the computer memory or because the order of invocation of the different methods is not the one anticipated by the different programmers or because there is a mismatch in the data types. Etc.

System Testing - A system consists of all of the software (and possibly hardware, user manuals, training materials, etc.) that make up the product delivered to the customer. System testing focuses on defects that arise at this highest level of integration. Typically system testing includes many types of testing: functionality, usability, security, internationalization and localization, reliability and availability, capacity, performance, backup and recovery, portability, and many more.

Acceptance Testing - Acceptance testing is defined as that testing, which when completed successfully, will result in the customer accepting the software and giving us their money. From the customer's point of view, they would generally like the most exhaustive acceptance testing possible (equivalent to the level of system testing). From the vendor's point of view, we would generally like the minimum level of testing possible that would result in money changing hands. Typical strategic questions that should be addressed before acceptance testing are: Who defines the level of the acceptance testing? Who creates the test scripts? Who executes the tests? What is the pass/fail criteria for the acceptance test? When and how do we get paid?

Integration and system tests

Test larger units of behavior

Often based on use cases or user stories -- customer perspective

```
@Test void gameTest() {
    Poker game = new Poker();
    Player p = new Player();
    Player q = new Player();
    game.shuffle(seed)
    game.add(p);
    game.add(q);
    game.deal();
    p.bet(100);
    q.bet(100);
    p.call();
```

Integration tests

Test combined behavior of multiple functions

```
def test_cleaning_with_feature_eng() {
    d = load_test_data();
    cd = clean(d);
    f = feature3.encode(cd);
    assert(no_missing_values(f["m"]));
    assert(max(f["m"]) <= 1.0);
}
```

Test Integration of Components

```
// making predictions with an ensemble of models
function predict_price(data, models, timeoutms) {
    // send asynchronous REST requests all models
    const requests = models.map(model => rpc(model, data, {time
    // collect all answers and return average if at least two m
    return Promise.all(requests).then(predictions => {
        const success = predictions.filter(v => v >= 0)
        if (success.length < 2) throw new Error("Too many model
        return success.reduce((a, b) => a + b, 0) / success.len
    })
}
```

End-To-End Test of Entire Pipeline

```
def test_pipeline():
  train = pd.read_csv('pipelinetest_training.csv', parse_dates
  test = pd.read_csv('pipelinetest_test.csv', parse_dates=True
  X_train, y_train = prepare_data(train)
  X_test, y_test = prepare_data(test)
  model = learn(X_train, y_train)
  accuracy = eval(model, X_test, y_test)
  assert accuracy > 0.9
```

System Testing from a User Perspective

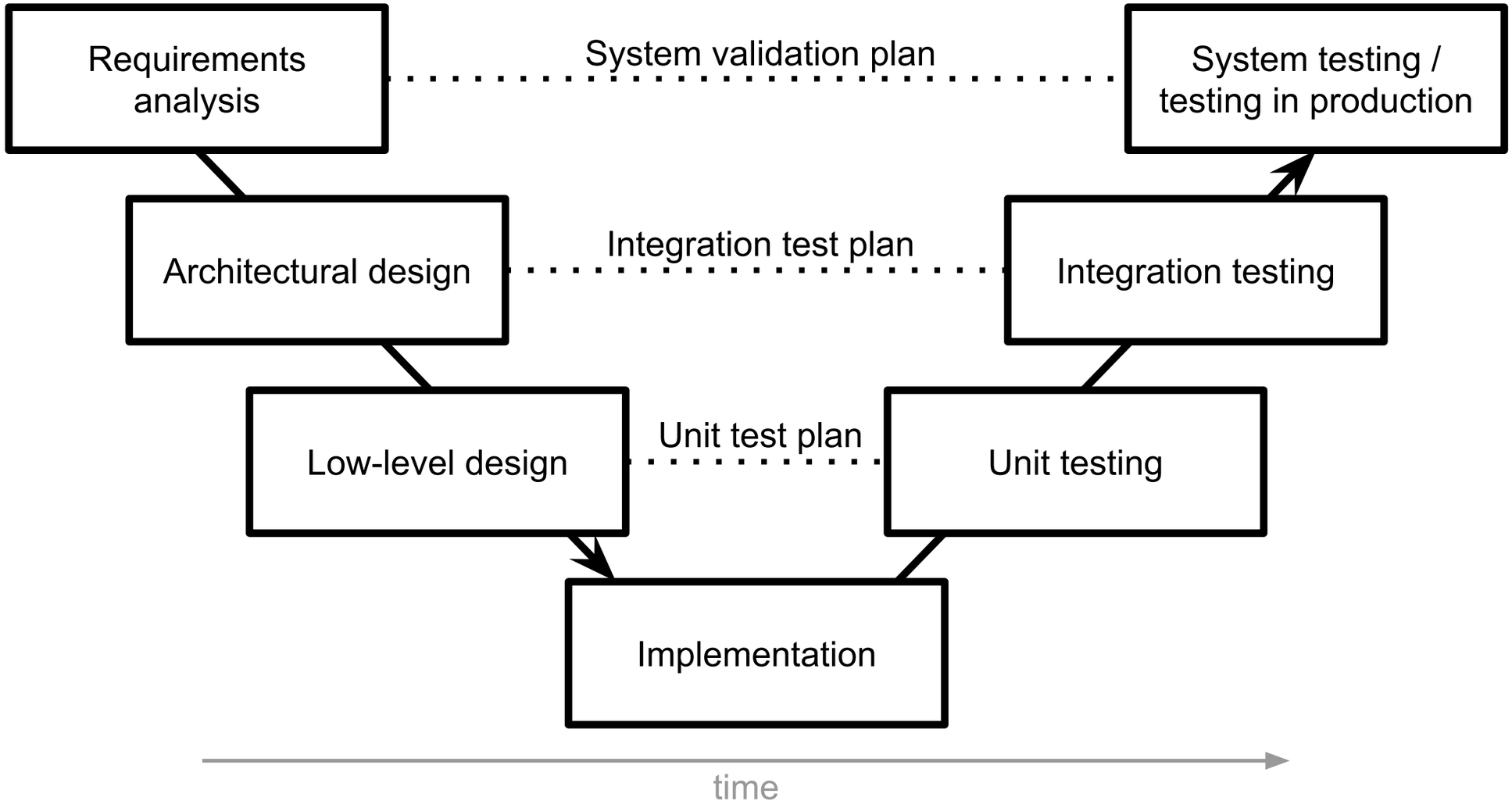
Test the product as a whole, not just components

Click through user interface, achieve task (often manually performed)

Derived from requirements (use cases, user stories)

Testing in production

The V-Model of Testing



Code Review and Static Analysis

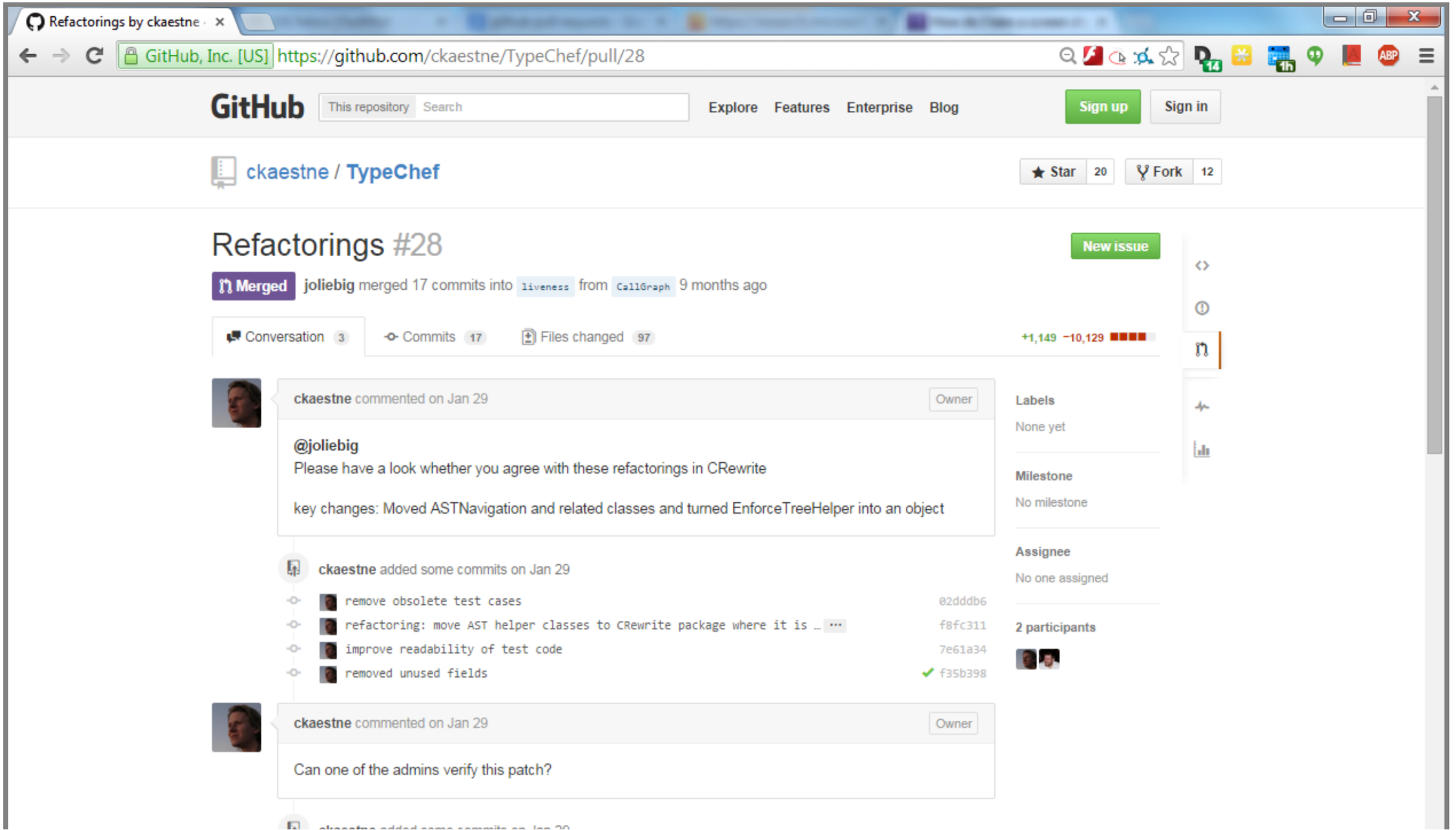
Code Review

Manual inspection of code

- Looking for problems and possible improvements
- Possibly following checklists
- Individually or as group

Modern code review: Incremental review at checking

- Review individual changes before merging
- Pull requests on GitHub
- Not very effective at finding bugs, but many other benefits: knowledge transfer, code imporvement, shared code ownership, improving testing


Refactorings by ckaestne
GitHub, Inc. [US] https://github.com/ckaestne/TypeChef/pull/28
GitHub
This repository Search
Explore Features Enterprise Blog
Sign up Sign in
ckaestne / TypeChef
Star 20 Fork 12
Refactorings #28
New issue
Merged joliebig merged 17 commits into liveness from CallGraph 9 months ago
Conversation 3 Commits 17 Files changed 97 +1,149 −10,129
ckaestne commented on Jan 29 Owner
@joliebig
Please have a look whether you agree with these refactorings in CRewrite
key changes: Moved ASTNavigation and related classes and turned EnforceTreeHelper into an object
ckaestne added some commits on Jan 29
remove obsolete test cases 02dddb6
refactoring: move AST helper classes to CRewrite package where it is … f8fc311
improve readability of test code 7e61a34
removed unused fields f35b398
ckaestne commented on Jan 29 Owner
Can one of the admins verify this patch?
Labels
None yet
Milestone
No milestone
Assignee
No one assigned
2 participants

Subtle Bugs in Data Wrangling Code

```
df['Join_year'] = df.Joined.dropna().map(
    lambda x: x.split(',')[1].split(' ')[1])
```

```
df.loc[idx_nan_age,'Age'].loc[idx_nan_age] =
    df['Title'].loc[idx_nan_age].map(map_means)
```

```
df["Weight"].astype(str).astype(int)
```

```
df['Reviws'] = df['Reviews'].apply(int)
```

Speaker notes

We did code review earlier together

Static Analysis, Code Linting

Automatic detection of problematic patterns based on code structure

```
if (user.jobTitle = "manager") {
    ...
}
```

```
function fn() {
    x = 1;
    return x;
    x = 3;
}
```

Static Analysis for Data Science Code

- Lots of research
- Style issues in Python
- Shape analysis of tensors in deep learning
- Analysis of flow of datasets to detect data leakage
- ...

Examples:

- Yang, Chenyang, et al.. "Data Leakage in Notebooks: Static Detection and Better Processes." Proc. ASE (2022).
- Lagouvardos, S. et al. (2020). Static analysis of shape in TensorFlow programs. In Proc. ECOOP.
- Wang, Jiawei, et al. "Better code, better sharing: on the need of analyzing jupyter notebooks." In Proc. ICSE-NIER. 2020.

Process Integration: Static Analysis Warnings during Code Review

Speaker notes

Social engineering to force developers to pay attention. Also possible with integration in pull requests on GitHub.

Bonus: Data Linter at Google

Miscoding

- Number, date, time as string
- Enum as real
- Tokenizable string (long strings, all unique)
- Zip code as number

Outliers and scaling

- Unnormalized feature (varies widely)
- Tailed distributions
- Uncommon sign

Packaging

- Duplicate rows
- Empty/missing data

Further readings: Hynes, Nick, D. Sculley, and Michael Terry. The data linter: Lightweight, automated sanity checking for ML data sets. NIPS MLSys Workshop. 2017.

Summary

- Beyond model and data quality: Quality of the infrastructure matters, danger of silent mistakes
- Automate pipelines to foster testing, evolution, and experimentation
- Many SE techniques for test automation, testing robustness, test adequacy, testing in production useful for infrastructure quality

Further Readings

- O'Leary, Katie, and Makoto Uchida. "Common problems with Creating Machine Learning Pipelines from Existing Code." Proc. Third Conference on Machine Learning and Systems (MLSys) (2020).
- Eric Breck, Shanqing Cai, Eric Nielsen, Michael Salib, D. Sculley. The ML Test Score: A Rubric for ML Production Readiness and Technical Debt Reduction. Proceedings of IEEE Big Data (2017)
- 📰 Zinkevich, Martin. Rules of Machine Learning: Best Practices for ML Engineering. Google Blog Post, 2017
- Serban, Alex, Koen van der Blom, Holger Hoos, and Joost Visser. "Adoption and Effects of Software Engineering Best Practices in Machine Learning." In Proc. ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (2020).