Тестирование

Борисенко Глеб

ФТиАД2022

Что это такое?

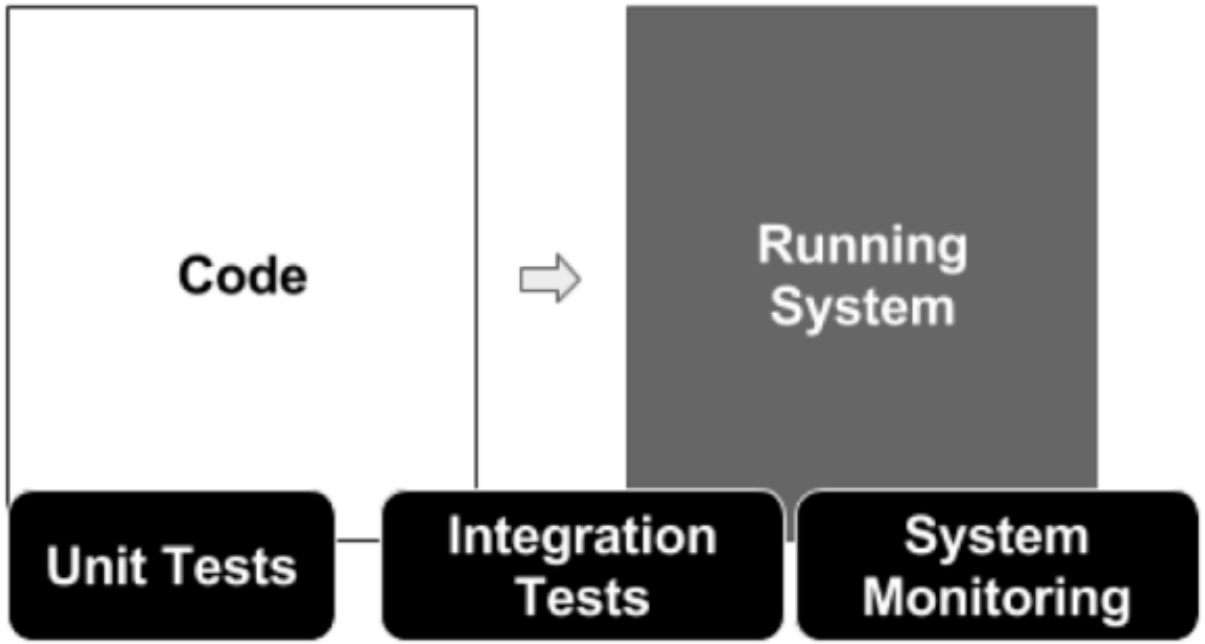
- Тестирование - инструмент, который позволяет проверять выполнение наших предположений / ожиданий о работе системы или ее части.
- В самом простом случае - мы задаем вход и ожидаемый выход для какого-то участка системы.

Что дает тестирование?

- Позволяет менять имплементацию существующего функционала
- Проверять, что новые изменения не ломают старую логику

Тестирование в классической разработке

Какое бывает тестирование



Traditional System Testing and Monitoring

Какое бывает тестирование

Выделяют несколько типов тестов:

- блочные / юнит / unit
- интеграционные / integration
- системные / system, end-2-end, e2e
- приемочные / acceptance

Юнит-тесты

- Юнит - тесты одного участка кода в изоляции от остальной системы.
- Есть вход, есть выход, проверяем совпадение.
- Как правило, покрывают небольшой участок кода / функцию.
- Примеры: сортировка массива, на вход - [1 ,4 ,0, 2], выход - [0, 1, 2, 4]

Интеграционные тесты

- Интеграционный тест - тест того, как разные компоненты системы взаимодействуют друг с другом.
- Пример: интеграция приложения с базой данных.
  - Инициализация - очищаем тестовую базу, кладем в нее значение.
  - Запускаем часть системы
  - Проверяем, что ответ соотвествует нашим ожиданиям.

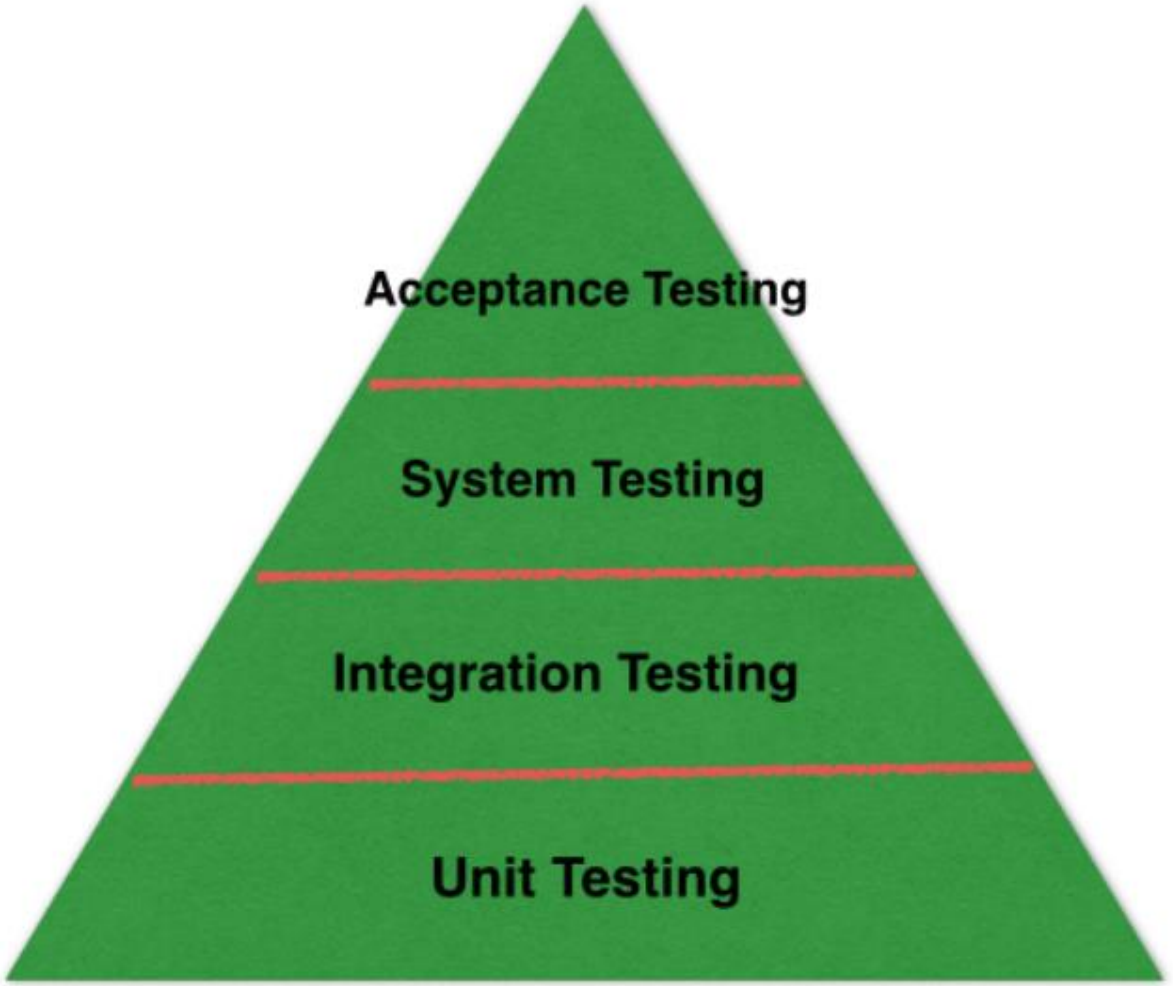
Системное тестирование

- Системное тестирование - проверяет, что система целиком работает.
- Пример:
  - Развернули приложение со всеми тестовыми базами,
  - Отправили запрос на выдачу рекомендаций,
  - Проверили что ответ пришел
  - И что он разумный

Приемочное тестирование

- Приемочное / продакшн тестирование - проверяет, что система работает “как ожидается” в бою.
- Пример:
  - Выдерживание нагрузки и времени ответа
  - Переключение реплик
  - Плавная деградация
  - И все то, что без реальной нагрузки проверить сложно

Пирамида тестирования



Ремарки

- Важны тесты всех типов
- Чем ниже тип теста в пирамиде, тем их проще писать и тем больше их требуется
- Покрыть тестами все - очень сложно
- Метрика покрытия кода тестами - code coverage
- Хорошая практика - на любой баг заводить регрессионный тест.
  - Регрессия бага - когда починили, а он снова вернулся

Инструменты

- Для упрощения написания тестов есть разные подходы/библиотеки для разных языков программирования.
- Самыми известными инструментами для Python являются pytest и tox.

tox

- tox – python пакет, которые позволяет запускать ваши тесты в нескольких средах.
- tox создает несколько окружений по написанному вами небольшому конфиг файлу, и запускает в каждом из них ваши тесты

pytest

- pytest – python пакет для проведения тестов вашего кода, то есть их написания и запуска.
- Предоставляет удобный CLI-интерфейс для запуска тестов и дает удобный красивый отчет о результатах этого запуска.
- Позволяет писать **фикстуры** – специальные конструкции, которые запускаются перед и после ваших тестов.
- Также есть набор полезных функций для сравнения результатов (что-то вроде специальных assert-ов)

Нагрузочное тестирование: locust

Написание простого теста на pytest

- Создаем тест – просто питоновский файл с `test_` в начале названия.
- Пишем функцию, так же с `test_` в начале (например, test_one.py), а для сравнения результатов используем стандартный питоновский assert:

```
def test_passing():
        assert (1, 2, 3) == (1, 2, 3)
```

- Запускаем тест через pytest:

```
pytest test_one.py
```

Результат запуска рабочего кода

```
(venv33) c:\venv33\Scripts>pytest c:\BOOK\bopytest-code\code\ch1\test_one.py
============================= test session starts =============================
platform win32 -- Python 3.3.5, pytest-3.2.5, py-1.4.34, pluggy-0.4.0
rootdir: c:\, inifile:
collecting 0 items
collecting 1 item
collected 1 item


..\..\BOOK\bopytest-code\code\ch1\test_one.py .

========================== 1 passed in 0.09 seconds ===========================

(venv33) c:\venv33\Scripts>
```

Результат запуска рабочего кода с -v

```
(venv33) c:\venv33\Scripts>pytest -v c:\BOOK\bopytest-code\code\ch1\test_one.py
============================= test session starts =============================
platform win32 -- Python 3.3.5, pytest-3.2.5, py-1.4.34, pluggy-0.4.0 -- c:\venv
33\scripts\python.exe
cachedir: ..\..\.cache
rootdir: c:\, inifile:
collecting 0 items
collecting 1 item
collected 1 item


..\..\BOOK\bopytest-code\code\ch1\test_one.py::test_passing PASSED

========================== 1 passed in 0.04 seconds ===========================

(venv33) c:\venv33\Scripts>
```

Результат запуска **не**рабочего кода

```
(venv33) c:\venv33\Scripts>pytest  c:\BOOK\bopytest-code\code\ch1\test_two.py
============================= test session starts =============================
platform win32 -- Python 3.3.5, pytest-3.2.5, py-1.4.34, pluggy-0.4.0
rootdir: c:\, inifile:
collecting 0 items
collecting 1 item
collected 1 item


..\..\BOOK\bopytest-code\code\ch1\test_two.py F

================================== FAILURES ===================================
_________________________________ test_failing _________________________________

    def test_failing():
>       assert (1, 2, 3) == (3, 2, 1)
E       assert (1, 2, 3) == (3, 2, 1)
E         At index 0 diff: 1 != 3
E         Use -v to get the full diff

c:\BOOK\bopytest-code\code\ch1\test_two.py:2: AssertionError
========================== 1 failed in 0.10 seconds ===========================

(venv33) c:\venv33\Scripts>_
```

Полезные штучки

Отлов исключений:

```
with pytest.raises(ValueError, match=r".* 123 .*"):
    myfunc()
```

Параметризация тестов:

```
@pytest.mark.parametrize(arg1, arg2, [(a1, b1), (a2, b2])
```

Фикстуры

- Фикстура – это функция, которая выполняется перед и после вашего теста
- Тест ваш используется в фикстуре на моменте вызова yield
- Описывается через pytest.fixture декоратор
- Используется как аргумент в определении теста
- Есть разные scope-ы, то есть когда именно вызывается фикстура
- Хранить их лучше в отдельном файле conftest.py

Пример

```
@pytest.fixture()
def tasks_db(tmpdir):
"""Подключение к БД перед тестами, отключение после."""
# Setup : start db
start_db(str(tmpdir), 'tiny’)

yield # здесь происходит тестирование

# Teardown : stop db
tasks.stop_tasks_db()
```

Mock-объекты

- При написании интеграционных тестов часто нужно создавать объект-заглушку, наример:
  - (псевдо-) базу, которая всегда возвращает один и тот же результат;
  - (псевдо-) внешний сервис, который всегда возвращает одинаковый ответ в рамках теста.
- Такие объекты называют mock-объектами.
- Они позволяют изолировать часть системы при интеграционном тестировании и явно управлять поведением вовлеченных компонентов.

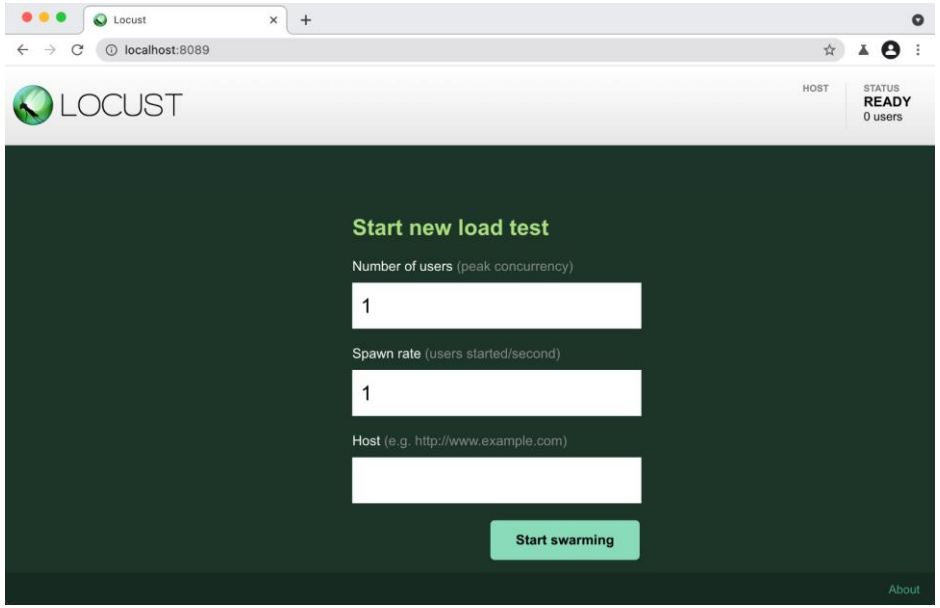
Пример мока на pytest-mock

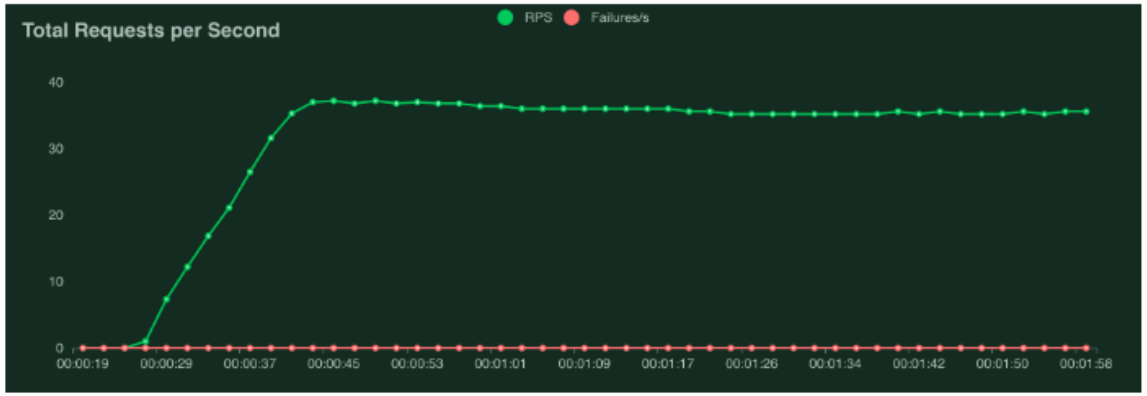
```
def test_slow_function_mocked_api_call(mocker):
    mocker.patch(
        # api_call is from slow.py but imported to main.py
        'mock_examples.main.api_call',
        return_value=5
    )

    expected = 5
    actual = slow_function()
    assert expected == actual
```

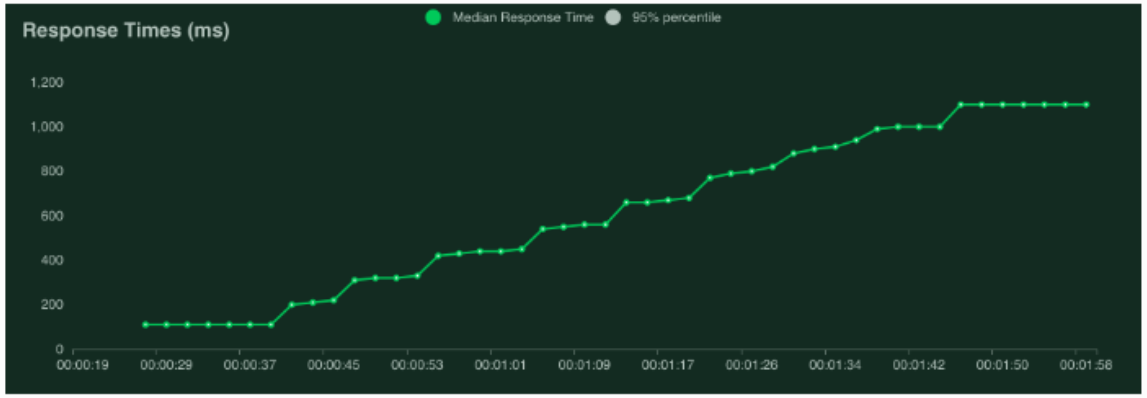
Нагрузочное тестирование

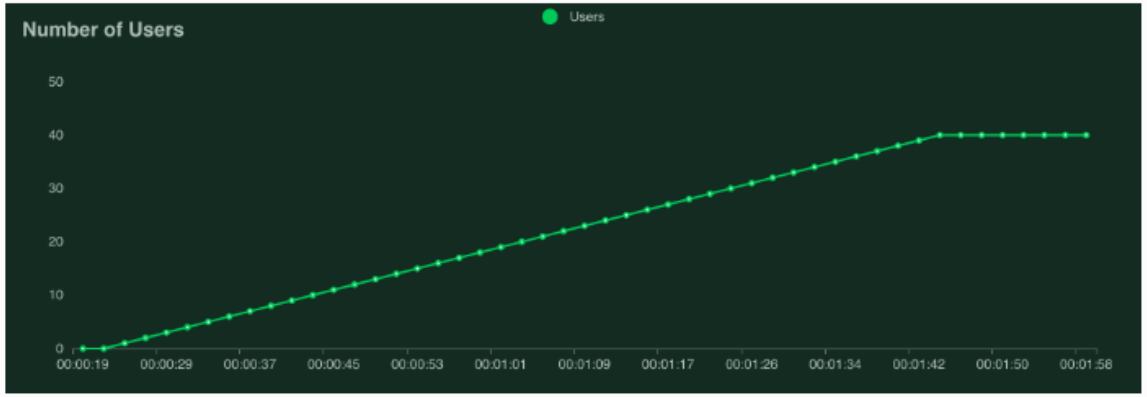
- SLA – соглашение, фиксирующее обязанности.
- В отношении НТ чаще всего SLA подразумевает определенное время ответа, количество запросов в секунду (RPS/TPS), количество используемых ресурсов.
- RPS – количество запросов, которое система может обработать за секунду.
- В инструментах всех обычно фиксируется не RPS, а количество пользователей и таймауты между сообщениями от одного пользователя.
- Инструменты: Jmeter самый популярный, Locust для нас самый простой (он на питоне)

Locust


Locust
localhost:8089
LOCUST
HOST
STATUS
READY
0 users
Start new load test
Number of users (peak concurrency)
1
Spawn rate (users started/second)
1
Host (e.g. http://www.example.com)
Start swarming
About


Total Requests per Second
RPS
Failures/s
40
30
20
10
0
00:00:19
00:00:29
00:00:37
00:00:45
00:00:53
00:01:01
00:01:09
00:01:17
00:01:26
00:01:34
00:01:42
00:01:50
00:01:58


Response Times (ms)
Median Response Time
95% percentile
1,200
1,000
800
600
400
200
0
00:00:19
00:00:29
00:00:37
00:00:45
00:00:53
00:01:01
00:01:09
00:01:17
00:01:26
00:01:34
00:01:42
00:01:50
00:01:58


Number of Users
Users
50
40
30
20
10
0
00:00:19
00:00:29
00:00:37
00:00:45
00:00:53
00:01:01
00:01:09
00:01:17
00:01:26
00:01:34
00:01:42
00:01:50
00:01:58

Locustfile.py

```python
from locust import HttpUser, task

class HelloWorldUser(HttpUser):
    @task
    def hello_world(self):
        self.client.get("/hello")
        self.client.get("/world")
```

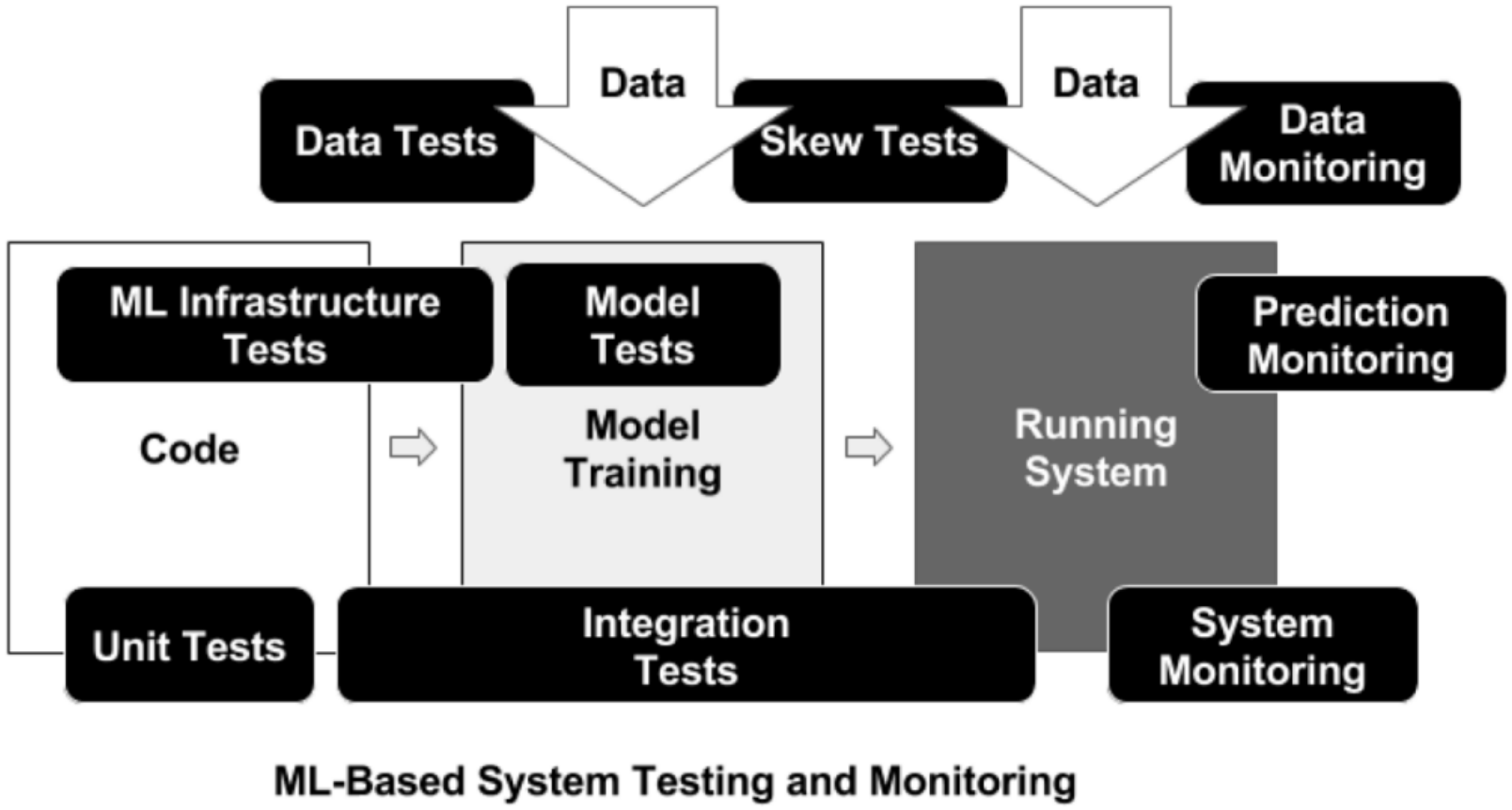
```python
from locust import HttpUser, task, between

class WebsiteTestUser(HttpUser):
    wait_time = between(0.5, 3.0)

    def on_start(self):
        """ on_start is called when a Locust start before any task is scheduled """
        pass

    def on_stop(self):
        """ on_stop is called when the TaskSet is stopping """
        pass

    @task(1)
    def hello_world(self):
        self.client.get("http://localhost:5000")
```

Тестирование в ML

Отличие от классического

- Тестирование разработок с использованием ML включает все те же приемы, что и в классической разработке, но добавляет дополнительную степень сложности.
- Причина - зависимость кода от модели, а модели - от данных. Это увеличивает пространство потенциальных ошибок.
- Это привело к появлению новых типов тестов:
  - Тесты данных (data tests)
  - Обнаружение смещения распределений (skew tests)
  - Вероятностные тесты (fuzzy tests)

Где эти тесты находятся



ML-Based System Testing and Monitoring

Тесты данных

- Цель тестов данных - убедиться, что наши ожидания от данных выполняются.
- Чаще всего это требуется на входе в систему и при обучении.
- Такие тесты часто включают в себя проверку на:
  - наличие необходимых полей в данных
  - формат (тестовый/числовой)
  - пропуски
  - количество записей
  - ожидаемые возможные значения (если речь о enum-like полях).
- Иными словами, это валидация схемы данных .

Инструмент для тестов данных

- Для этой задачи нередко используется Great Expectations в пайплайнах
- Это не очень простой инструмент в насйтройке, но очень простой в использовании
- Валидация данных происходит путем простых конфигурируемых правил
- По итогам каждой валидации создается отчетик в их собственном UI

Обнаружение смещения распределений

- Теория машинного обучения работает при условии, что распределение данных на обучении и на тесте совпадает.
- Проверка этого условия - и есть обнаружение смещения распределений.
- Оно позволяет отловить баги при обучении (признаки при обучении и в проде вычисляются по-разному), баги при изменении кода, “отвалившиеся” признаки и др.

Вероятностные тесты

- Тесты, которые проверяют вероятностные свойства части системы.
- Пример: хитрая функция сэмплирования коэффициента.
- Запускаем ее 1000 раз и проверяем среднее/мин/макс сэмпла.

А что по моделькам?

- Тесты моделек – это по сути их валидация во всех ее сущностях ☺
- Во CT (cont. testing) пайплайн можно включить, но это прям пик тестирования, для которого нужно много сделать:
  - Методология золотых валидационных датасетов
  - Методология версионирования моделей, с тегами, корректными названиями и т.п.
  - Универсальная и гибкая пайплайнизация запуска моделей
- Но если это все есть – это пик MLOps, которого крайне сложно достигнуть и нередко не всегда нужен из-за увеличения времени разработки

Инструменты

- Инструментов много и разных, большинство можно реализовать самостоятельно с помощью базовой статистики.
- TensorFlow имеет инструментарий для тестов данных и обнаружения смещения распределений: TensorFlow Data Validation (TFDV)
- Для больших данных и Spark есть библиотека с похожим функционалом - dequ

Заключение

- В реальных системах покрывать тестами стоит все хрупкие части или те, где возникают ошибки.
- Хрупкое – почти все :D
- Набор тестов стоит собирать под конкретный продукт и требования. “Классический набор” можно описать так:
  - юнит-тесты на код расчета признаков и все внутренние преобразования данных
  - интеграционные тесты на сервис с моделью
  - тесты данных перед обучением
  - детекция смещения между трейном и тестом
  - тесты качества модели (определение устаревания)