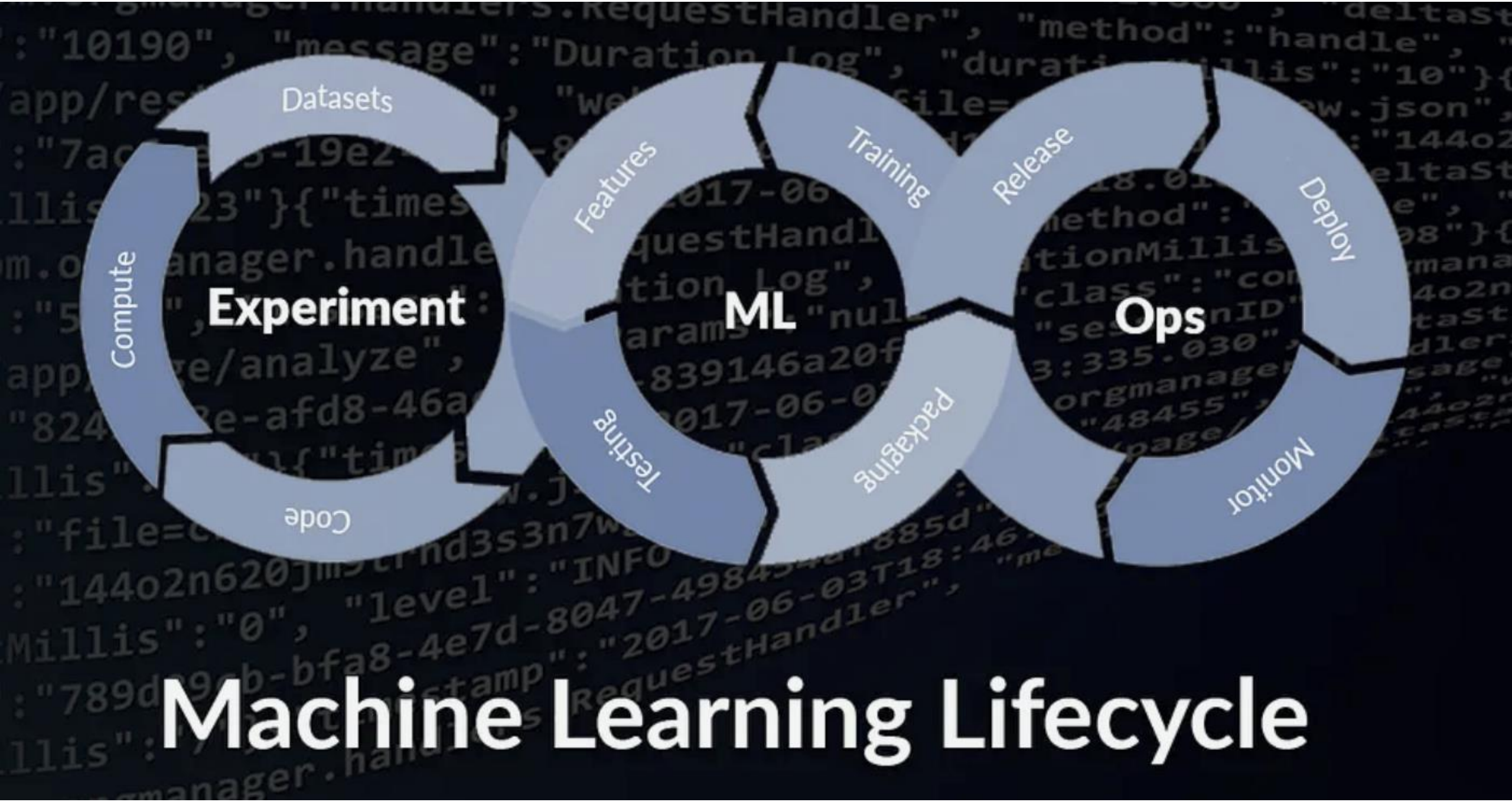
Лекция 4-5

практика организации проектов с моделями машинного обучения


Datasets
Features
Training
Release
Deploy
Compute
Experiment
ML
Ops
Testing
Packaging
Monitor
Code
Machine Learning Lifecycle

Проект: начало

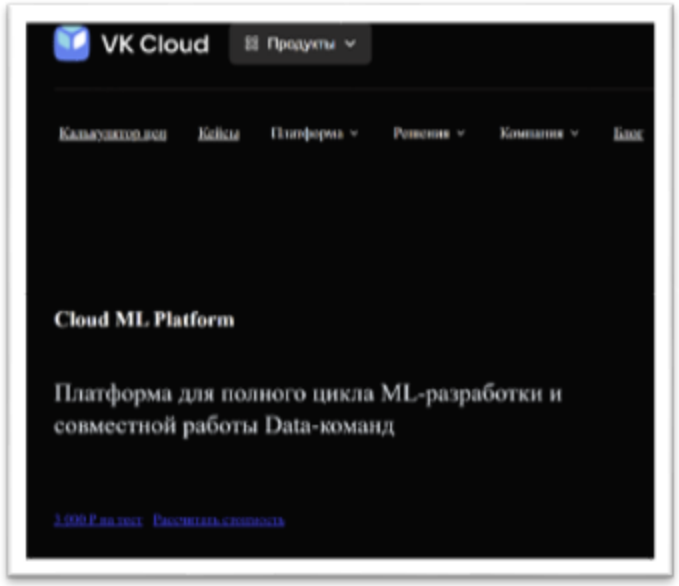
- Цели:
  воспроизводимость, масштабируемость, автоматизация.
- Ключевые аспекты:
  данные, документация, тестирование, CI/CD
- Важно: это работа с данными! инфраструктура! процессы!

Отличия от классического DevOps

- **DevOps** фокусируется на автоматизации разработки и доставки программного обеспечения
- **MLOps** добавляет специфические аспекты, такие как:
  - Управление данными и их версионирование
  - Эксперименты с моделями и их воспроизводимость
  - Мониторинг производительности моделей в реальном времени

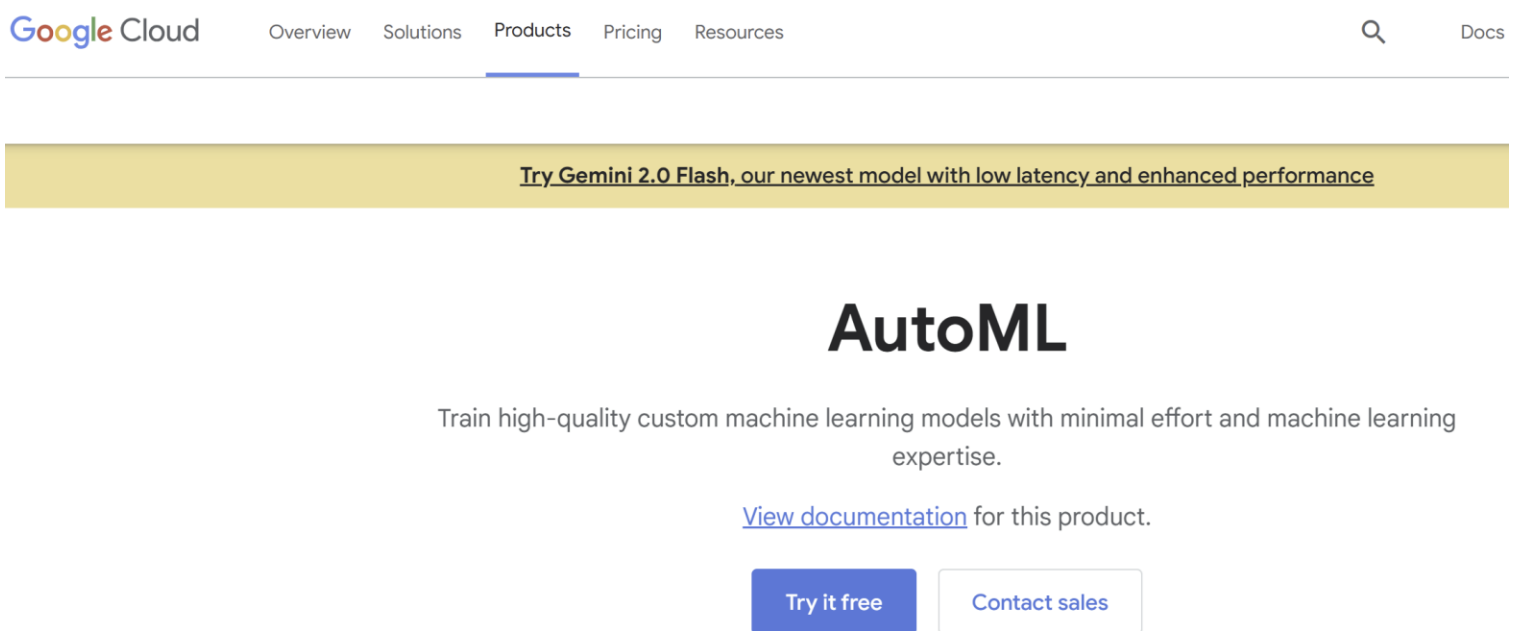
Основные компоненты MLOps

- **Данные:** Управление, версионирование и качество данных.
- **Код:** Автоматизация процессов обучения и развертывания моделей.
- **Инфраструктура:** Масштабируемые и надежные платформы для выполнения моделей

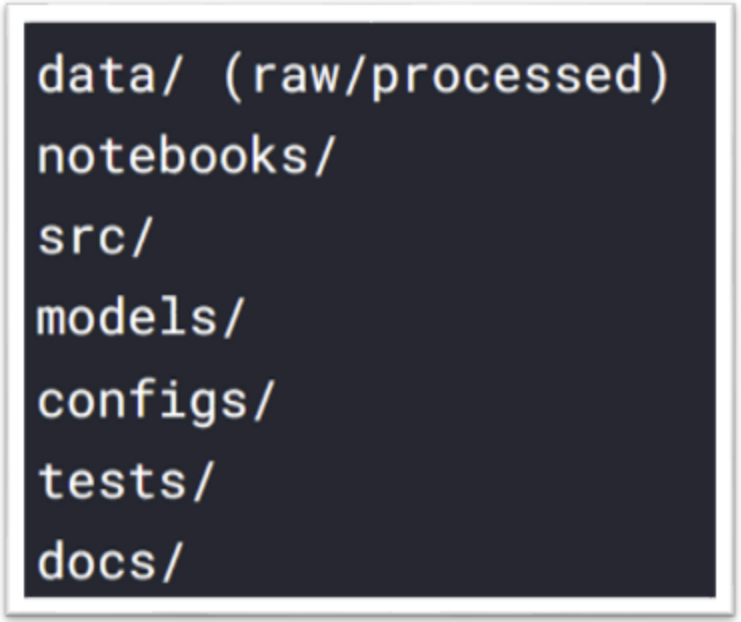


IBM Watson® Studio empowers data scientists, developers and analysts to build, run and manage AI models, and optimize decisions anywhere on IBM Cloud Pak® for Data. Unite teams, automate AI lifecycles and speed time to value on an open multicloud architecture.

Get started with IBM Cloud Pak →



Структура проекта: кратко



Хранение данных

- **data/**: Основная директория для хранения данных
- **raw/**: Исходные, необработанные данные
- **processed/**: Обработанные данные, готовые для использования в моделях
- разделение данных на raw и processed для обеспечения прозрачности и воспроизводимости

Исследовательские блокноты и исходный код

- **notebooks/**: Блокноты для исследования данных, экспериментов и визуализации.
- **src/**: Исходный код Python, включающий модули для обработки данных, обучения моделей и утилиты.

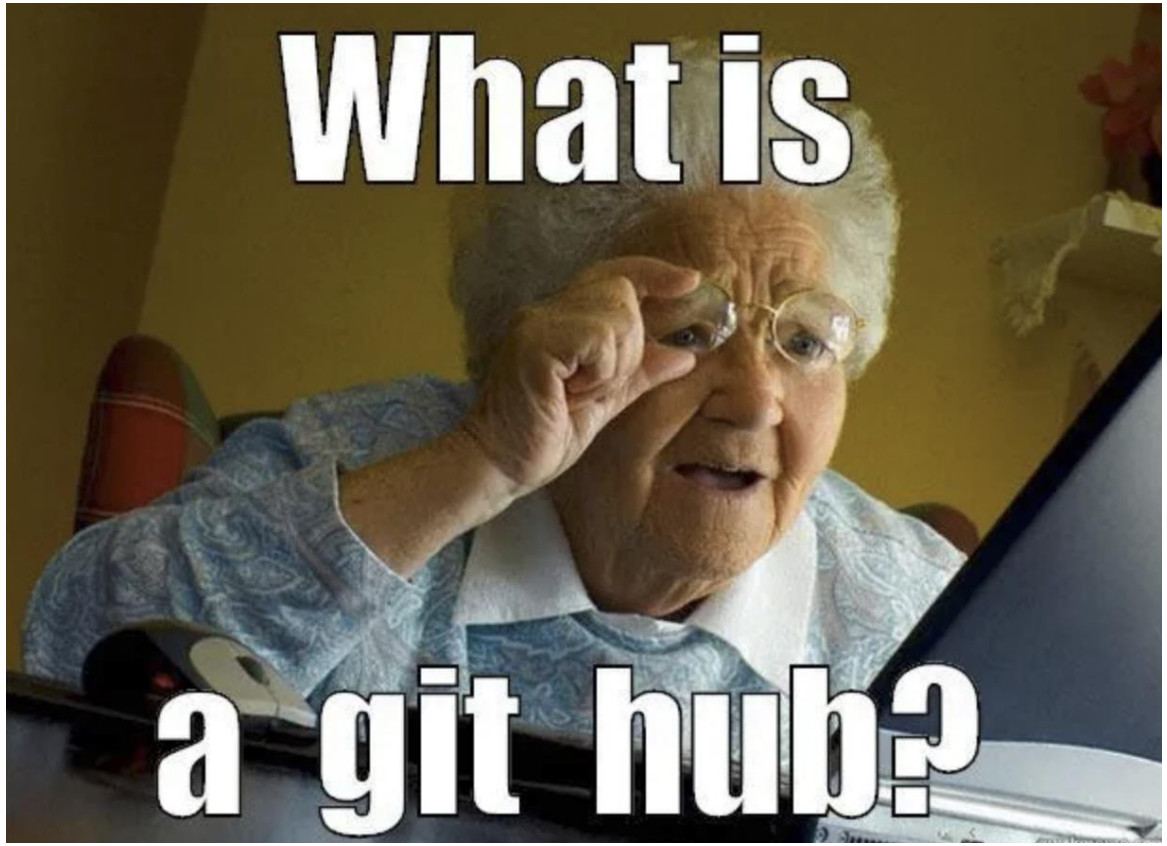
Модели и конфигурации

- **models/**: Артефакты моделей, включая обученные модели и их метаданные.
- **configs/**: Конфигурационные файлы для настройки параметров обучения, данных и инфраструктуры.
- версионирование моделей и конфигураций

Тестирование и документация

- **tests/**: Модульные и интеграционные тесты для проверки корректности кода и моделей.
- **docs/**: Документация проекта, включая руководства, API и описание архитектуры.

Организация GitHub-репозитория



README.md и .gitignore

- **README.md**: Визитная карточка проекта, содержащая описание, инструкции по установке и использованию.
- **.gitignore**: Шаблон для исключения ненужных файлов (например, временных файлов, данных, кэша) из репозитория.

Лицензия и CONTRIBUTING.md

- **Лицензия**: Определяет права и ограничения на использование проекта.
- **CONTRIBUTING.md**: Руководство для контрибьюторов, включая процесс внесения изменений и стандарты кода.

Использование веток

- **Git Flow**: Традиционная модель с основными ветками (master, develop) и вспомогательными (feature, release, hotfix).
- **GitHub Flow**: Упрощенная модель с основной веткой (main) и ветками для функций (feature branches).

Примеры хороших репозиториев

nin-jin 18 фев 2021 в 18:39

Грубая оценка проблемности GitHub-проектов

4 мин 4K

GitHub*, Анализ и проектирование систем*, Программирование*, Веб-разработка*

Здравствуйте, меня зовут Дмитрий Карловский и я… практикую терморектальную ароматерапию. Понимаю, что каждый любит своё болото, и будет защищать его до последней капли жижи. Тем не менее, высокая инженерная культура требует объективности в оценке инструментов.

Часто для решения одной и той же задачи существует более чем один подходящий по функциональности вариант. При прочих равных, хотелось бы выбрать такой проект, который доставит меньше всего проблем. Но как навскидку оценить объём этих проблем, не тратя несколько человеколет для собственноручного набивания всевозможных шишек?

Что ж, давайте разберём, какие бывают проблемы, как их оценить и посравниваем некоторые популярные проекты.

```
#
def f(x): return x*2+3

#
def calculate_velocity(mass: float) -> float:
    """
    GRAVITY = 9.81
    return mass * GRAVITY * 2 + 3
```

- Соблюдение PEP8, использование flake8 и black
- Типизация (mypy)

```
def f(x):return xx2+3
```

```
def calculate_velocity(mass):
    GRAVITY = 9.81
    return mass * GRAVITY * 2 + 3
```

```
def calc(a,b):return a*b+10
```

```
def calculate_force(mass: float, acceleration: float) -> float:
    return mass * acceleration

def calculate_velocity(mass: float) -> float:
    GRAVITY = 9.81
    return mass * GRAVITY * 2 + 3

def calculate_energy(mass: float, speed: float) -> float:
    return 0.5 * mass * speed ** 2
```

Использование docstrings

- **Docstrings** — это строки документации, которые описывают назначение и использование функций, классов и модулей
- **Google Style**:
  - Простой и читаемый формат
- **NumPy Style**:
  - Более детализированный формат, часто используемый в научных проектах

Автоматическая генерация документации

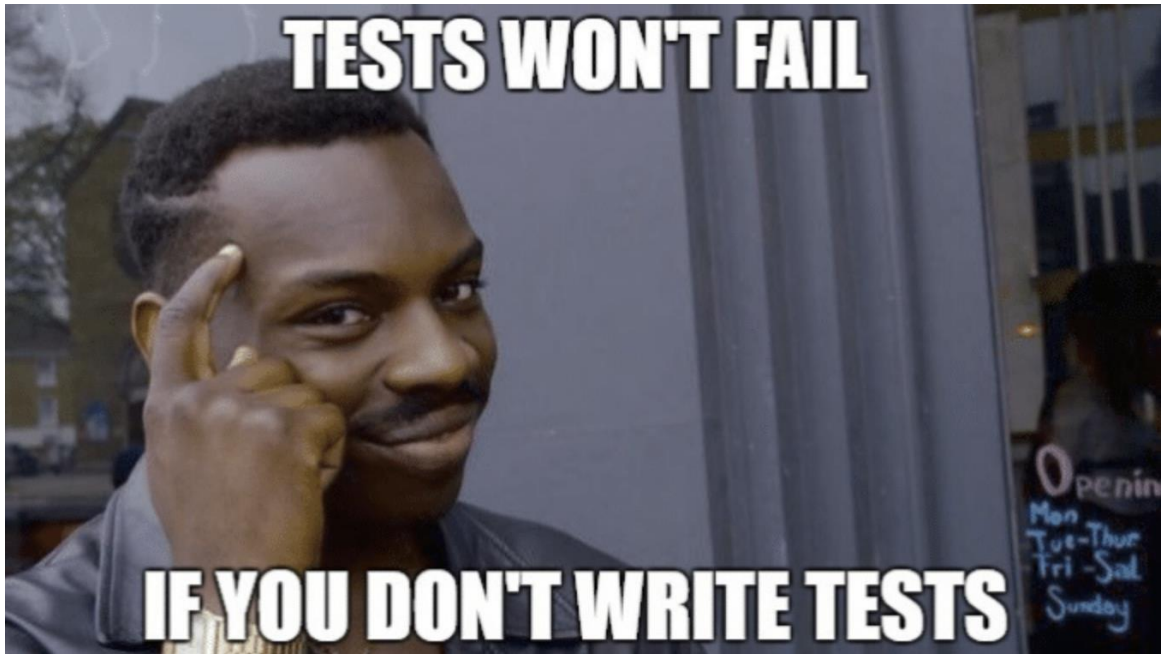
Sphinx

```
sphinx-build -b html sourcedir builddir
```

MkDocs

```
mkdocs build
```

Тестирование



Модульные и интеграционные тесты

- **Модульные тесты**:
  - Проверяют отдельные компоненты или функции изолированно.
  - Используют **pytest** для написания и запуска тестов.

```
def add(a: int, b: int) -> int:
    return a + b

def test_add():
    assert add(1, 2) == 3
    assert add(-1, 1) == 0
```

  - Преимущества: быстрое выполнение, простота отладки.

Интеграционные тесты:

- Проверяют взаимодействие между несколькими компонентами или системами.

```
def test_integration():
result = some_complex_operation()
assert result == expected_value
```

- Преимущества: проверка реального взаимодействия компонентов.

Покрытие кода с использованием coverage.py

- **Покрытие кода** — это метрика, показывающая, какая часть кода была выполнена во время тестов
- **coverage.py** — инструмент для измерения покрытия кода в Python

Взаимодействие данных, кода и моделей

- **Данные**: Исходные и обработанные данные хранятся в директориях `data/raw/` и `data/processed/`.
- **Код**: Скрипты и модули в `src/` обрабатывают данные и обучают модели.
- **Модели**: Обученные модели сохраняются в `models/` и могут быть загружены для предсказаний.
- Пример взаимодействия:
  - Данные загружаются и предобрабатываются.
  - Код обучает модель на обработанных данных.
  - Модель сохраняется и используется для предсказаний

Работа с конфигурационными файлами

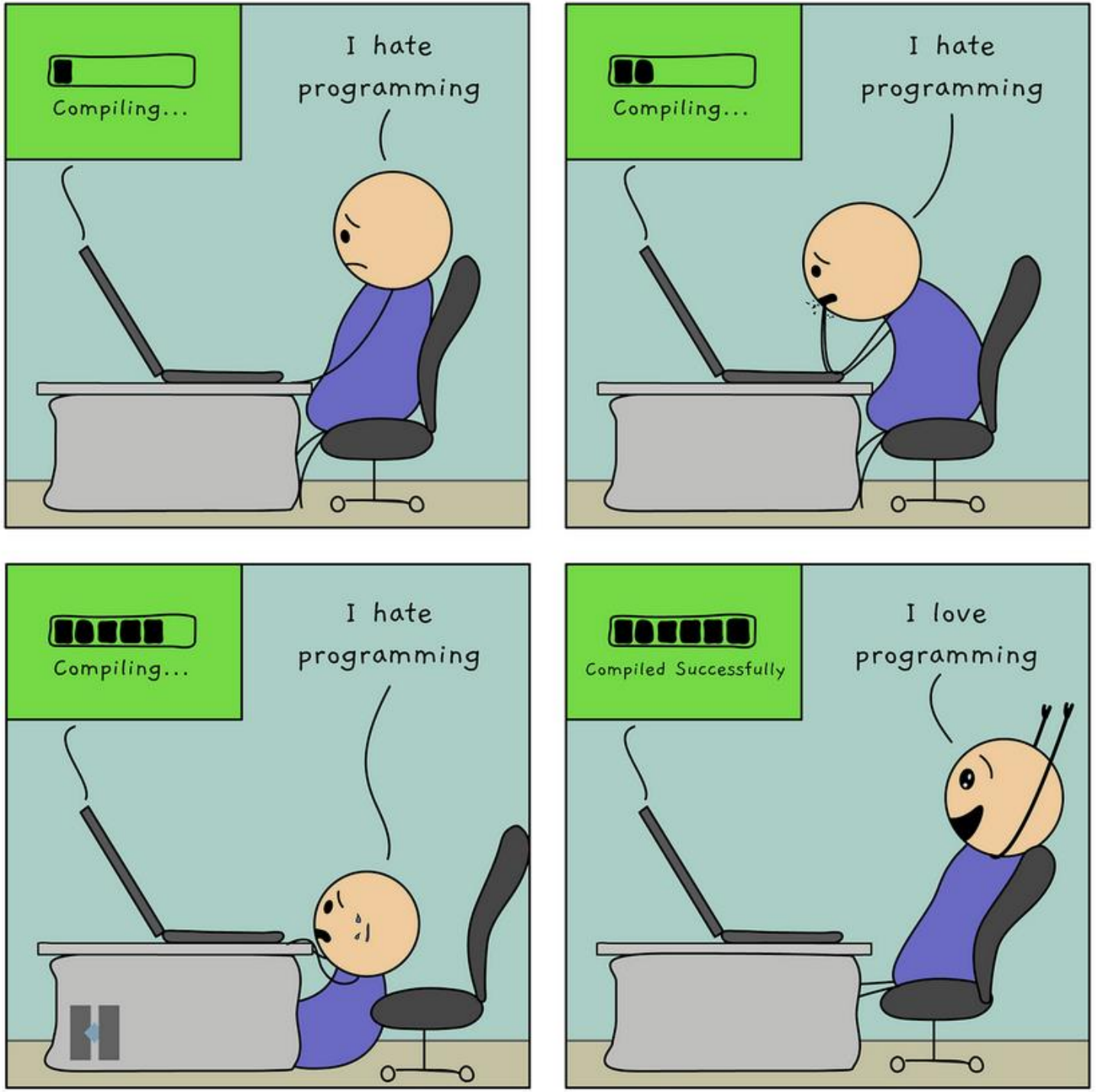
Конфигурационные файлы хранятся в configs/ и содержат параметры для настройки процессов

Управление зависимостями

- **requirements.txt**: Простой текстовый файл для списка зависимостей.
- numpy==1.21.0 pandas==1.3.0 scikit-learn==0.24.2
- pip install -r requirements.txt
- **Poetry**: Современный инструмент для управления зависимостями и виртуальными окружениями

Важные хинты по выкатке кода

- Автоматизация CI/CD
- Контейнеризация
- Мониторинг и логирование
- Версионирование данных и моделей
- Откат и канареечные релизы


Compiling...
I hate programming
Compiling...
I hate programming
Compiling...
I hate programming
Compiled Successfully
I love programming

Удачи с ДЗ!