

## Лабораторная 1 — Настройка репозитория и окружения (scaffold)

**Цель:** подготовить репозиторий проекта, окружение разработки, базовую структуру файлов и CI-реплику.

**Требования:** Git, Conda/venv, Docker (локально), доступ к GitHub.

### Шаги:

1. Создать приватный репозиторий на GitHub (или форкнуть шаблон).
2. Клонировать репозиторий:
3. `git clone git@github.com:Irina-64/mlops-flight-delay.git`
4. `cd mlops-flight-delay`
5. Создать виртуальное окружение и установить зависимости:
6. `conda create -n mlops python=3.10 -y`
7. `conda activate mlops`
8. `pip install -r requirements.txt`

или `pip install` по `requirements.txt`.

9. Инициализировать DVC (локальный remote):
10. `dvc init`
11. `mkdir -p dvc_remote`
12. `dvc remote add -d local_remote dvc_remote`
13. `git add . && git commit -m "project scaffold"`
14. Создать шаблонные директории и файл `README.md`, `LICENSE`, `src/`, `data/`, `models/`, `tests/`.
15. Настроить базовый `.gitignore`, `dvc ignore`, и шаблон `environment.yml`.

**Артефакты:** работающий репозиторий с установленным окружением, dvc и шаблонами файлов.

**Критерии оценки:** репозиторий корректно клонируется, окружение запускается, DVC и Git инициализированы.

**Опционально:** добавить pre-commit hooks (black, isort, flake8).

## Лабораторная 2 — Версионирование данных с DVC

**Цель:** научиться хранить и версионировать датасеты с DVC, управлять remote хранилищем.

**Требования:** репозиторий из ЛР1, примерный CSV датасет (можно положить в `data/raw/`).

### Шаги:

1. Положить исходный CSV в `data/raw/flights_sample.csv`.
2. Добавить файл в DVC:
3. `dvc add data/raw/flights_sample.csv`
4. `git add data/.gitignore data/raw/flights_sample.csv.dvc`
5. `git commit -m "add raw flights dataset via dvc"`
6. Настроить локальный remote (если ещё не сделали) и выполнить `dvc push`:
7. `dvc remote add -d local_remote ./dvc_remote`
8. `dvc push -r local_remote`
9. Показать историю версий данных: создать небольшое изменение в датасете, повторно `dvc add` и закоммитить — показать, как вернуться к предыдущей версии:

10. `git checkout HEAD~1`
11. `dvc pull`
12. Показать `dvc status`, `dvc metrics` (если есть) и `dvc diff` (для изменений).

**Артефакты:** `data/*.dvc` файлы, локальный remote с данными, демонстрация восстановления старой версии.

**Критерии оценки:** набор данных контролируется DVC, remote работает, студенты демонстрируют `checkout/pull` старой версии.

**Опционально:** подключить облачный remote (S3/GCP) — для продвинутых.

## Лабораторная 3 — Предобработка данных и DVC пайплайн

**Цель:** написать скрипт предобработки и оформить его как stage в DVC-пайплайне.

**Требования:** `src/preprocess.py` шаблон, `data/raw/` с CSV.

**Шаги:**

1. Реализовать `src/preprocess.py`:
  - o чтение `data/raw/flights_sample.csv`,
  - o очистка NaN, приведение типов,
  - o создание простых признаков: `day_of_week`, `is_weekend`, `departure_hour_bucket`,
  - o сохранить результат в `data/processed/processed.csv`.
2. Добавить stage в DVC:
3. `dvc stage add -n preprocess \`
4. `-d src/preprocess.py \`
5. `-d data/raw/flights_sample.csv \`
6. `-o data/processed/processed.csv \`
7. `python src/preprocess.py`
8. Запустить stage и закоммитить:
9. `dvc repro`
10. `git add dvc.yaml dvc.lock`
11. `git commit -m "add preprocess stage"`
12. Проверить воспроизводимость: удалить `data/processed/` и запустить `dvc repro` — результат должен восстановиться.

**Артефакты:** `data/processed/processed.csv`, `dvc.yaml`, `dvc.lock`, `src/preprocess.py`.

**Критерии оценки:** корректный pipeline stage, reproducibility через `dvc repro`.

**Опционально:** добавить базовый EDA (`matplotlib`), сохранять отчёт как `reports/eda.html`.

## Лабораторная 4 — Обучение модели и логирование экспериментов (MLflow)

**Цель:** написать тренировочный скрипт, логировать эксперименты в MLflow.

**Требования:** `src/train.py`, `data/processed/processed.csv`, MLflow установлен.

**Шаги:**

1. Реализовать `src/train.py`:
  - o загрузка `data/processed/processed.csv`,

- разделение train/test,
  - обучение RandomForestClassifier (или LogisticRegression),
  - вычисление метрик (accuracy, roc\_auc),
  - сохранение модели (joblib.dump(...)).
2. В коде использовать MLflow:
- ```
3. import mlflow
4. mlflow.set_experiment("flight_delay")
5. with mlflow.start_run():
6.     mlflow.log_param("model", "RandomForest")
7.     mlflow.log_metric("roc_auc", roc_auc)
8.     mlflow.sklearn.log_model(model, "model")
```
9. Запустить MLflow UI:
  10. mlflow ui --port 5000

и показать записанные рансы.

11. Добавить stage в DVC/или Airflow позже — для сейчас просто коммит.

**Артефакты:** сохранённая модель, записи в MLflow UI, src/train.py.

**Критерии оценки:** MLflow содержит ран с параметрами и метриками, модель корректно сохраняется и загружается.

**Опционально:** добавить гиперпараметрический поиск (GridSearchCV) и логирование каждого ранна.

## Лабораторная 5 — Оценка модели и регистрация в Model Registry

**Цель:** провести оценку модели (подробный отчёт), зарегистрировать версию модели в MLflow Model Registry.

**Требования:** MLflow (локальный tracking server или file store).

### Шаги:

1. Написать src/evaluate.py, который:
  - загружает тестовый набор,
  - вычисляет ROC AUC, precision/recall, confusion matrix,
  - сохраняет отчёт (reports/eval.json или reports/eval.html).
2. Зарегистрировать модель в MLflow Model Registry:
  - найти run\_id нужного ранна в MLflow UI,
  - выполнить:
 

```
mlflow models prepare-container -m "runs:<run_id>/model" -n flight_delay_model
```
  - # либо:
 

```
python -c "import mlflow;
mlflow.register_model('runs:<run_id>/model',
'flight_delay_model')"
```
3. В MLflow UI отметить описание модели (staging/prod) и комментировать.
4. Составить короткий отчет о метриках и выводах.

**Артефакты:** reports/eval.\*, модель зарегистрирована в Registry.

**Критерии оценки:** корректность метрик, модель есть в Registry, отчет читаем.

**Опционально:** написать скрипт promote\_model.py для автоматического продвижения версии по правилам (например, roc\_auc > 0.80 → staging).

## Лабораторная 6 — REST API (FastAPI) и контейнеризация (Docker)

**Цель:** сделать REST API для прогноза и упаковать приложение в Docker.

**Требования:** FastAPI, uvicorn, Docker, модель (joblib или mlflow model).

### Шаги:

1. Создать `src/api.py` (FastAPI) с endpoint `/predict`:
2. `from fastapi import FastAPI`
3. `import joblib, pandas as pd`
- 4.
5. `app = FastAPI()`
6. `model = joblib.load("models/model.joblib")`
- 7.
8. `@app.post("/predict")`
9. `def predict(payload: dict):`
10. `df = pd.DataFrame([payload])`
11. `preds = model.predict_proba(df)[0,1]`
12. `return {"delay_prob": float(preds[0])}`
13. Добавить Dockerfile в корень:
14. `FROM python:3.10-slim`
15. `WORKDIR /app`
16. `COPY requirements.txt .`
17. `RUN pip install -r requirements.txt`
18. `COPY . .`
19. `CMD ["uvicorn", "src.api:app", "--host", "0.0.0.0", "--port", "8080"]`
20. Собрать и запустить контейнер:
21. `docker build -t flight-delay-api:lab6 .`
22. `docker run -p 8080:8080 flight-delay-api:lab6`
23. Протестировать API:
24. `curl -X POST "http://localhost:8080/predict" -H "Content-Type: application/json" -d '{"carrier": "AA", "dep_hour": 9, ...}'`
25. Закоммитить Dockerfile и minimal README о запуске.

**Артефакты:** рабочий контейнер с API, пример запроса.

**Критерии оценки:** endpoint возвращает правдоподобные предсказания, контейнер стартует.

**Опционально:** поддержать `mlflow.pyfunc.load_model` вместо `joblib` и хранить модель в `models/`.

## Лабораторная 7 — Тестирование и базовый CI (GitHub Actions)

**Цель:** написать тесты для кода и настроить CI, который запускает тесты при пуше.

**Требования:** pytest, GitHub Actions.

### Шаги:

1. Реализовать тесты (`tests/test_preprocess.py`, `tests/test_api.py`) — mock входных данных и ожидаемый тип ответа. Пример:
2. `def test_predict_output_type(client):`
3. `resp = client.post("/predict", json={"carrier": "AA", "dep_hour": 9})`
4. `assert "delay_prob" in resp.json()`
5. Добавить `conftest.py` с тестовым FastAPI client (`TestClient`).
6. Создать `.github/workflows/ci.yml`:
7. `name: CI`

8. on: [push, pull\_request]
9. jobs:
10.     test:
11.         runs-on: ubuntu-latest
12.         steps:
13.             - uses: actions/checkout@v3
14.             - uses: actions/setup-python@v4
15.                 with: python-version: 3.10
16.             - run: pip install -r requirements.txt
17.             - run: pytest -q
18. Сделать пуш и проверить workflow в Actions.

**Артефакты:** тесты в tests/, успешный GitHub Actions run.

**Критерии оценки:** все тесты проходят в CI, покрытие основных функций.

**Опционально:** расширить CI — сбор Docker образа и публикация в registry (требуется секрет).

## Лабораторная 8 — Оркестрация пайплайна (Airflow)

**Цель:** организовать автоматический пайплайн (ETL → обучение → регистрация) в Airflow.

**Требования:** Docker Compose Airflow или локальная установка Airflow.

**Шаги:**

1. Запустить Airflow (рекомендуется docker-compose образ Apache Airflow).
2. Создать DAG airflow/dags/flight\_pipeline.py с задачами:
  - o download\_data (если нужно),
  - o preprocess (bash: python src/preprocess.py),
  - o train (python src/train.py),
  - o evaluate (python src/evaluate.py),
  - o register (при успешной оценке — register model).
3. Каждый task оформить как BashOperator или PythonOperator. Пример:
4. preprocess = BashOperator(  
5.     task\_id='preprocess',  
6.     bash\_command='python /opt/airflow/dags/../../src/preprocess.py'  
7. )
8. Запустить DAG вручную из UI, просмотреть логи, убедиться, что артефакты (модель, отчеты) появились.
9. Настроить переменные/конфигурации в Airflow для путей и порогов.

**Артефакты:** рабочий DAG, выполненный run, сохранённые артефакты.

**Критерии оценки:** DAG выполняется без ошибок, лог действий читаем.

**Опционально:** настроить SLA, retry policy и оповещение (email/slack).

## Лабораторная 9 — Feature Store (Feast) — оффлайн материализация и использование в обучении

**Цель:** вынести признаки в feature store и интегрировать их с этапом тренировки/предсказаний.

**Требования:** Feast (локально), Redis (опционально) или файловый store для разработки.

## Шаги:

1. Инициализировать Feast репозиторий (`feature_repo/`) и файл `feature_store.yaml`.
2. Определить `entity` (например, `flight_id` или `carrier_id`) и `feature_view` (набор признаков: `avg_delay_by_carrier`, `dep_hour_bucket` и т.д.).
3. Подготовить оффлайн таблицу `features` (CSV) и `feast materialize`:
4. `feast apply`
5. `feast materialize 2020-01-01 2020-12-31`
6. В `src/train.py` заменить локальное чтение признаков на выборку из Feast offline store.
7. Продемонстрировать, что обучающая выборка теперь собирается через Feature Store.

**Артефакты:** `feature_repo/` с описанием фич, `materialize` успешен.

**Критерии оценки:** модель обучается на фичах из Feast, `code reproducible`.

**Опционально:** настроить online store и использовать его в `src/api.py` при получении реального запроса.

## Лабораторная 10 — Деплой в Kubernetes (Minikube)

**Цель:** развернуть API контейнер в локальном Kubernetes (Minikube/KIND) и обеспечить масштабирование.

**Требования:** Minikube или KIND, `kubectl`, образ Docker (см. ЛР6).

## Шаги:

1. Запустить Minikube:
2. `minikube start`
3. `eval $(minikube docker-env)` # чтобы собирать образы прямо в Minikube
4. `docker build -t flight-delay-api:lab10 .`
5. Создать манифесты в `k8s/`:
  - o `deployment.yaml` (Deployment с контейнером),
  - o `service.yaml` (NodePort или LoadBalancer),
  - o `hpa.yaml` (HorizontalPodAutoscaler по CPU).
6. Применить манифесты:
7. `kubectl apply -f k8s/`
8. `kubectl get pods,svc`
9. Проверить доступность сервиса:
10. `minikube service flight-delay-svc --url`
11. `curl <url>/predict ...`
12. Проверить autoscale: `kubectl scale --replicas=3 deployment/flight-delay` и нагрузить (`ab`, `hey`) — проверить увеличение реплик.

**Артефакты:** работающий Deployment + Service, доступный endpoint.

**Критерии оценки:** сервис доступен, можно масштабировать.

**Опционально:** использовать Helm chart и подготовить `values.yaml`.

## Лабораторная 11 — Мониторинг (Prometheus + Grafana) и метрики приложения

**Цель:** инструментировать сервис метриками, настроить Prometheus для сбора и Grafana — для визуализации.

**Требования:** prometheus\_client (Python), Prometheus, Grafana (Docker Compose или K8s).

#### Шаги:

1. Внедрить prometheus\_client в src/api.py:
  - добавить метрики: REQUEST\_COUNT, REQUEST\_LATENCY, PREDICTION\_DISTRIBUTION (гистограмма/summary).
  - endpoint /metrics автоматически отдается.
2. Настроить Prometheus config (prometheus.yml) для сканирования api:8080/metrics.
3. Запустить Prometheus и Grafana (локально через Docker Compose или в K8s).
4. В Grafana создать dashboard с графиками: запросы/с, latency, средняя вероятность задержки по времени.
5. Продемонстрировать случаи (много запросов / долгие запросы) и мониторинг.

**Артефакты:** dashboard Grafana, Prometheus targets up, метрики в /metrics.

**Критерии оценки:** метрики экспортируются, Grafana отображает данные.

**Опционально:** настроить alert (Prometheus Alertmanager) — например, если latency > X или ошибка > Y%.

## Лабораторная 12 — Детекция дрейфа и автоматическая реакция

**Цель:** реализовать простую систему обнаружения дрейфа (feature / performance drift) и запуск retraining.

**Требования:** библиотека для drift (Evidently или самописный PSI), Airflow или скрипт-расписание.

#### Шаги:

1. Написать скрипт src/drift\_check.py, который:
  - сравнивает распределения признаков (train vs production) через PSI/KS,
  - сравнивает текущую метрику (roc\_auc на контрольном наборе / recent requests).
2. Добавить задачу в Airflow DAG (или отдельный cron job), которая запускает drift\_check.py регулярно.
3. Если drift превышает порог — триггерить DAG retrain (Airflow TriggerDagRunOperator) или запуск python src/train.py и затем register.
4. Смоделировать дрейф (изменить входной набор запросов) и продемонстрировать, что retrain запускается автоматически.

**Артефакты:** reports/drift\_report.json, автоматический триггер retrain.

**Критерии оценки:** детекция дрейфа с корректным срабатыванием триггера.

**Опционально:** интеграция с MLflow для сравнения новых версий и автоматической промоции.

## Лабораторная 13 — Полный CI/CD: от кода до обновления кластера

**Цель:** настроить CI/CD, который при мердже в main собирает образ, прогоняет тесты и деплоит в Minikube/kubernetes (или обновляет манифесты).

**Требования:** GitHub Actions (или GitLab CI), секреты (Registry), kubectl в runner (или использовать self-hosted runner).

### Шаги:

1. Расширить `.github/workflows/ci.yml` → `deploy.yml`:
  - По push в main:
    - запуск тестов,
    - сборка Docker image,
    - push в registry (GHCR/DockerHub) — требует `DOCKER_USERNAME`, `DOCKER_PASSWORD` в Secrets,
    - деплой в Kubernetes: `kubectl set image deployment/flight-delay api=ghcr.io/your-org/flight-delay:latest`.
2. Альтернативно — реализовать GitOps: обновление image в `k8s/deployment.yaml` и push в infra repo, где ArgoCD/Flux автоматом применит изменения.
3. Проверить workflow (пуш в feature branch → PR → merge → автоматический деплой).
4. В логах CI проверить, что образ появился и кластера обновлён.

**Артефакты:** рабочий pipeline CI/CD + задеплоенное приложение после merge.

**Критерии оценки:** автоматический деплой после merge, CI тесты и Build успешны.

**Опционально:** привязать деплой к версии модели в MLflow (например, deploy только если модель в Registry промотирована в Production).

## Лабораторная 14 — Итоговая интеграция, демонстрация и отчёт

**Цель:** собрать и задеплоить end-to-end систему; подготовить презентацию и документацию.

**Требования:** все предыдущие лабораторные выполнены/интегрированы.

### Шаги:

1. Собрать чек-лист: DVC → Airflow → MLflow Registry → Feature Store → API (Docker/K8s) → Monitoring → CI/CD → Drift detection.
2. Подготовить README.md с инструкцией «как воспроизвести» (кроки: `git clone` → `dvc pull` → `dvc repro` → `mlflow ui` → `minikube start` → `kubectl apply -f k8s/` и т.д.).
3. Сделать демонстрацию (живой или записанный ролик 3–5 минут):
  - показать данные → preprocessing → запуск DAG → создание модели → регистрация → деплой → тестовый /predict → мониторинг.
4. Подготовить краткий отчёт (markdown или pdf) с описанием архитектуры, принятых решений, проблем и путей улучшения.
5. Защитить проект устно (5–10 минут) — показать ключевые артефакты.

**Артефакты:** работающий E2E проект, README, demo, финальный отчет.

**Критерии оценки:** полнота интеграции, воспроизводимость, качество кода и документации, убедительность демонстрации.



## Критерии оценивания

- **Каждая лабораторная:** проверяем минимум по 3 пунктам:
  1. Работоспособность (выполняются основные шаги).
  2. Репродуцируемость (корректные коммиты, DVC/Git история).
  3. Документация/код (readme + чистый код).
- **Итоговый проект:** оцениваем интеграцию систем (E2E), качество презентации, качество кода/тестов, мониторинг и политика реагирования на дрейф.
- используем check-list для каждой ЛР (готовность артефактов, CI green, MLflow run visible, Docker image доступен и т.д.).