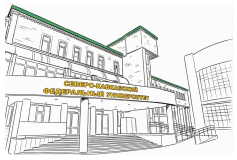

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное автономное образовательное учреждение высшего образования

«СЕВЕРО-КАВКАЗСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»

Институт перспективной инженерии

Департамент цифровых, робототехнических систем и электроники

ИНСТРУМЕНТАЛЬНЫЕ СРЕДСТВА ИНФОРМАЦИОННЫХ СИСТЕМ

Методические указания
к выполнению лабораторных работ

Направление подготовки: 09.03.02 «Информационные системы и технологии»
Профиль: «Прикладное программирование и интернет-технологии»

Квалификация (степень) выпускника: бакалавр
Форма обучения: очная

Ставрополь – 2026

УДК 004.41:004.7
ББК 32.973.2-018.2

ИНСТРУМЕНТАЛЬНЫЕ СРЕДСТВА ИНФОРМАЦИОННЫХ СИСТЕМ

Методические указания к лабораторным работам

Аннотация: Методические указания содержат комплект из 8 лабораторных работ по дисциплине «Инструментальные средства информационных систем». Каждая работа включает цель, задачи, теоретический минимум, пошаговый разбор примера, фрагменты программного кода для автоматизации конфигурирования и анализа, индивидуальное задание для самостоятельного выполнения, а также вопросы для самоконтроля. Материал охватывает полный цикл разработки современных ИС: от настройки среды и моделирования архитектуры до развёртывания контейнеризованных решений и организации мониторинга. Рекомендуемый объем — 32 часа лабораторных занятий. Работы выполняются с использованием современных IDE, CASE-средств, систем управления версиями, фреймворков разработки и инструментов CI/CD/Docker.
Составитель: к.т.н. Д.Л. Винокурский
Рецензент: к.ф.м.н. Е.В. Крахоткина

Содержание

Введение

Дисциплина «Инструментальные средства информационных систем» является ключевой в подготовке специалистов, способных проектировать, разрабатывать и сопровождать сложные программно-аппаратные комплексы. Современная разработка ИС невозможна без глубокого понимания экосистемы инструментов, обеспечивающих автоматизацию, стандартизацию и контроль качества на всех этапах жизненного цикла. **Актуальность дисциплины** обусловлена следующими факторами:

- Переход к Agile/DevOps-практикам требует владения инструментами непрерывной интеграции, тестирования и развёртывания;
- Рост сложности архитектуры ИС (микросервисы, распределённые системы) диктует необходимость использования CASE-средств и стандартизированных нотаций моделирования;
- Требования к надёжности и безопасности систем повышают роль автоматизированного тестирования, статического анализа и мониторинга;
- Контейнеризация и инфраструктура как код (IaC) становятся стандартом промышленного развёртывания.

Цель методических указаний — предоставить обучающимся структурированный практический материал для освоения современных инструментальных средств разработки, сопровождения и эксплуатации информационных систем. **Задачи лабораторного практикума:**

1. Сформировать навыки настройки профессиональных сред разработки и управления зависимостями;
2. Освоить методы визуального моделирования архитектуры и бизнес-процессов ИС;
3. Научить проектировать, реализовывать и мигрировать структуры данных с использованием ORM;
4. Развить компетенции в создании серверных и клиентских компонентов, интеграции через API;
5. Освоить практики совместной разработки, контроля версий и автоматизации сборки/развёртывания;
6. Подготовить к решению прикладных задач обеспечения качества, контейнеризации и мониторинга ИС.

Организация выполнения работ:

- Каждая лабораторная работа рассчитана на 2 академических часа;
- Перед началом работы студент обязан изучить теоретический материал и ответить на допусковые вопросы;
- Практическая часть выполняется под контролем преподавателя с соблюдением правил работы с вычислительной инфраструктурой;
- Отчёт по работе должен содержать: цель, схему/структуру проекта, логи выполнения, фрагменты кода/конфигураций, результаты тестов/мониторинга, выводы и ответы на контрольные вопросы;
- Защита работы включает демонстрацию рабочего прототипа и устный ответ по методологии применения инструментов.

Требования к программному обеспечению:

- Язык программирования: Python 3.10+;
- Среда разработки: VS Code / PyCharm / Jupyter Notebook;
- Инструменты: Git, Docker, Docker Compose, Postman/Insomnia, pytest,

PlantUML/Diagrams.net, SQLite/PostgreSQL, FastAPI/Flask, GitHub Actions/GitLab CI;
- Дополнительно: WSL2 (при работе в Windows), виртуальные окружения venv/conda.

1 Настройка интегрированной среды разработки и рабочего окружения ИС

Цель: Освоить установку, конфигурацию и верификацию базовых инструментальных платформ для создания информационных систем. **Задачи:**

- Изучить архитектуру современных IDE и редакторов кода, механизмы управления зависимостями;
- Создать изолированное виртуальное окружение и настроить менеджер пакетов;
- Подключить статические анализаторы, форматировщики и автодополнение;
- Верифицировать работоспособность toolchain на тестовом проекте.

Теоретическое описание: Современная среда разработки объединяет редактор, компилятор/интерпретатор, отладчик, систему контроля версий и расширения для линтинга. Виртуальное окружение обеспечивает изоляцию зависимостей проекта, предотвращая конфликты версий библиотек. Методология “Infrastructure as Code” для локальной среды предполагает декларативную настройку через файлы конфигурации (например, `requirements.txt`, `pyproject.toml`, `.prettierrc`). Стандартизация окружения сокращает время онбординга новых разработчиков и исключает ошибки типа “на моей машине работает”. **Разобранный пример:** 1. Установка VS Code, расширение Python, Pylance, Black, Ruff.
2. Создание окружения: `python -m venv .venv`, активация.
3. Инициализация проекта: `pip install fastapi uvicorn httpx`, генерация `requirements.txt`.
4. Настройка линтера и форматера через `pyproject.toml`.
5. Результат: проект запускается, линтер не выдаёт ошибок, форматирование применяется автоматически при сохранении. **Программный код (скрипт верификации окружения):**

```
import sys
import subprocess
import importlib

REQUIRED_PACKAGES = {"fastapi", "uvicorn", "httpx", "pytest", "ruff",
    "black"}

def check_environment():
    print(f"Python              : {sys.version}")
    print(f"                                  : {sys.executable}\n")

    missing = []
    for pkg in REQUIRED_PACKAGES:
        try:
            importlib.import_module(pkg)
            print(f"[OK] {pkg}")
        except ImportError:
            missing.append(pkg)
            print(f"[MISSING] {pkg}")

    if missing:
        print(f"\ n
                        : pip install {' '.join(missing)}")
    else:
```

```
23          print("\ n
                                    .")
24
25  if __name__ == "__main__":
26      check_environment()
```

Индивидуальное задание: Настройте автоматическое применение линтера и форматера через pre-commit hook (библиотека `pre-commit`). Добавьте проверку типов с помощью `mypy`. Убедитесь, что commit блокируется при наличии ошибок. **Вопросы для самопроверки:**

1. В чём принципиальное отличие виртуального окружения от глобальной установки пакетов?
2. Зачем в современных проектах используют `pyproject.toml` вместо разрозненных `setup.py` и `requirements.txt`?
3. Как статический анализ кода отличается от динамического тестирования?

2 Моделирование архитектуры и бизнес-процессов ИС с помощью CASE-средств

Цель: Научиться создавать концептуальные и логические модели ИС в стандартизированных нотациях с применением CASE-инструментов. **Задачи:**

- Изучить основы UML и BPMN, принципы model-driven разработки;
- Разработать диаграммы Use Case, Sequence и Component;
- Экспортировать модели в машиночитаемые форматы;
- Провести валидацию связности элементов модели.

Теоретическое описание: CASE-средства (Computer-Aided Software Engineering) автоматизируют создание, анализ и генерацию кода из моделей. Диаграммы UML описывают структуру и поведение системы: `Use Case` фиксирует требования пользователей, `Sequence` — взаимодействие компонентов во времени, `Component` — физическую организацию модулей. Современные инструменты (PlantUML, Draw.io, Enterprise Architect) поддерживают генерацию из текстовых спецификаций, что обеспечивает версионирование моделей вместе с кодом. Коэффициент связности модели оценивается как отношение realised relations к total possible relations. **Разобранный пример:** 1. Инструмент: PlantUML + VS Code расширение.
2. Создание `sequence.puml` с описанием регистрации пользователя.
3. Рендеринг в PNG/SVG, проверка полноты переходов.
4. Результат: диаграмма содержит 5 участников, 12 сообщений, 2 альтернативных ветвления; генерируется без ошибок. **Программный код (парсинг и валидация PlantUML):**

```
import re
from pathlib import Path

def validate_plantuml(filepath):
    text = Path(filepath).read_text(encoding='utf-8')
    errors = []
    if not re.search(r'@startuml\s*\n.*\n@enduml', text, re.DOTALL):
        errors.append("                          @startuml/@enduml")
    if re.search(r'actor\s+\w+', text) is None:
        errors.append("                                    actor")
    msg_count = len(re.findall(r'->', text)) + len(re.findall(r'-->',
        text))
    if msg_count < 3:
        errors.append("
                                        (<3)")
    return errors if errors else ["
                         "]

print(validate_plantuml("sequence.puml"))
```

Индивидуальное задание: Добавьте диаграмму состояний (State Machine) для сущности “Заказ” и диаграмму развёртывания (Deployment) для двухtier-архитектуры. Экспортируйте обе в SVG и опишите в отчёте точки интеграции с бизнес-логикой. **Вопросы для самопроверки:**

1. В каких случаях целесообразно использовать BPMN вместо UML Activity?
2. Как текстовое описание моделей (PlantUML/Mermaid) улучшает процесс code

review?

3. Что такое traceability matrix и как она связывает модели с тестами?

3 Проектирование и реализация базы данных информационной системы

Цель: Освоить инструменты создания, миграции и администрирования реляционных баз данных с применением ORM. **Задачи:**

- Изучить принципы реляционной модели, нормализацию до 3НФ;
- Реализовать схему данных через ORM (SQLAlchemy/Prisma);
- Настроить автоматические миграции структуры БД;
- Проанализировать планы выполнения запросов и индексы.

Теоретическое описание: ORM (Object-Relational Mapping) преобразует объектно-ориентированные сущности в реляционные таблицы, устраняя необходимость написания ручного SQL. Миграции обеспечивают контролируемое изменение схемы БД версионированными скриптами. Для повышения производительности применяются составные индексы, covering indexes и анализ запросов через `EXPLAIN ANALYZE`. ACID-свойства гарантируют атомарность транзакций, что критично для финансовых и учётных подсистем ИС. **Разобранный пример:** 1. Стек: SQLite + SQLAlchemy 2.0 + Alembic.
2. Модели: `User`, `Order`, `Item` (связи 1:N, M:N).
3. Миграция: `alembic revision -autogenerate -m "init"` `alembic upgrade head`.
4. Результат: схема создана, seed-данные загружены, сложные запросы (JOIN, subquery) выполняются за <5 мс. **Программный код (ORM-модели и seed-данные):**

```
from sqlalchemy import create_engine, Column, Integer, String,
    ForeignKey, Table
from sqlalchemy.orm import declarative_base, relationship,
    sessionmaker

Base = declarative_base()

order_items = Table(
    'order_items', Base.metadata,
    Column('order_id', ForeignKey('orders.id'), primary_key=True),
    Column('item_id', ForeignKey('items.id'), primary_key=True)
)

class Order(Base):
    __tablename__ = 'orders'
    id = Column(Integer, primary_key=True)
    status = Column(String(20), default='pending')
    items = relationship('Item', secondary=order_items,
        back_populates='orders')

class Item(Base):
    __tablename__ = 'items'
    id = Column(Integer, primary_key=True)
    name = Column(String(50), unique=True)
    price = Column(Integer)
    orders = relationship('Order', secondary=order_items,
        back_populates='items')

#
engine = create_engine('sqlite:///app.db', echo=False)
```

```
Base.metadata.create_all(engine)
Session = sessionmaker(engine)
with Session() as s:
    if not s.query(Order).first():
        s.add_all([Order(status='created')])
        s.commit()
```

Индивидуальное задание: Добавьте индекс по полю `status`, реализуйте пагинацию (limit/offset) и soft-delete. Сгенерируйте `EXPLAIN` план для запроса выбора активных заказов и прокомментируйте использование индекса. **Вопросы для самопроверки:**

1. Когда ORM снижает производительность по сравнению с чистым SQL?
2. Зачем нужны каскадные операции (`cascade="all, delete-orphan"`)?
3. Как миграции обеспечивают обратимость изменений схемы БД?

4 Разработка серверной части ИС на основе современных фреймворков

Цель: Сформировать навыки проектирования и реализации RESTful/GraphQL API для информационной системы. **Задачи:**

- Изучить принципы REST, маршрутизацию, middleware и валидацию данных;
- Реализовать CRUD-эндпоинты с асинхронной обработкой; генерацию OpenAPI-документации;
- Настроить обработку ошибок и логирование запросов.

Теоретическое описание: Современные backend-фреймворки (FastAPI, Express, Spring) предоставляют декларативный роутинг, автоматическую валидацию через схемы (Pydantic/Zod) и встроенную поддержку OpenAPI/Swagger. Middleware позволяет выполнять кросс-режущие задачи: аутентификацию, CORS, rate-limiting, метрики. Асинхронность (async/await) повышает пропускную способность при I/O-bound операциях. Стандартизация API через спецификацию OpenAPI упрощает интеграцию с фронтендом и автоматизированным тестированием. **Разобранный пример:** 1. Фреймворк: FastAPI + Uvicorn.
2. Эндпоинты: `GET /items`, `POST /items`, `PUT /items/{id}`.
3. Валидация: Pydantic-модели `ItemCreate`, `ItemResponse`.
4. Результат: API доступен по `http://localhost:8000/docs`, возвращает корректные статусы, логирует запросы. **Программный код (FastAPI CRUD с валидацией):**

```python
from fastapi import FastAPI, HTTPException, Depends
from pydantic import BaseModel
from typing import List

app = FastAPI(title="Lab 4 API")

class ItemBase(BaseModel):
    name: str
    price: float

class ItemCreate(ItemBase): pass
class Item(ItemBase):
    id: int

fake_db = {}
next_id = 1

@app.post("/items/", response_model=Item)
def create_item(item: ItemCreate):
    global next_id
    new_item = Item(id=next_id, **item.model_dump())
    fake_db[next_id] = new_item
    next_id += 1
    return new_item

@app.get("/items/", response_model=List[Item])
def read_items():
    return list(fake_db.values())
```

Индивидуальное задание: Добавьте middleware для логирования времени выполнения запросов, реализуйте пагинацию (`skip`, `limit`) и endpoint для поиска по названию с регистронезависимым сравнением. **Вопросы для самопроверки:**

1. Чем отличается асинхронный сервер от многопоточного при обработке I/O операций?
2. Как OpenAPI-спецификация ускоряет разработку фронтенда?
3. Зачем в REST используют коды состояния 2xx, 4xx, 5xx раздельно?

5 Создание клиентского интерфейса и интеграция с API

Цель: Освоить инструменты разработки UI и обеспечения взаимодействия клиентской части с серверным API. **Задачи:**

- Изучить архитектуру SPA, принципы работы HTTP-клиентов и CORS;
- Реализовать компоненты интерфейса с состоянием загрузки и ошибок;
- Настроить автоматическое обновление данных (polling/webhooks);
- Протестировать интеграцию с mock-сервером.

Теоретическое описание: Клиентские приложения взаимодействуют с API через HTTP-клиенты (Fetch, Axios, httpx). CORS (Cross-Origin Resource Sharing) контролирует доступ к ресурсам с других доменов через заголовки `Access-Control-*`. State management (локальное или глобальное) обеспечивает синхронизацию UI с данными. Для устойчивости к сетевым сбоям применяются стратегии повторных попыток (exponential backoff) и кэширование на клиенте. Mock-серверы (WireMock, httpbin, FastAPI test client) позволяют разрабатывать фронтенд параллельно с бэкендом. **Разобранный пример:** 1. Стек: Python `httpx` + Jinja2 шаблоны (или простой HTTP-клиент для демонстрации).
2. Сценарий: запрос к `GET /items/`, отображение в консоли/файле.
3. Обработка ошибок: таймаут 5 с, повтор 3 раза с backoff.
4. Результат: данные успешно получены, при недоступности API выводится понятное сообщение. **Программный код (клиент с retry и таймаутом):**

```
import httpx
import time
from typing import List, Dict

API_URL = "http://127.0.0.1:8000/items/"
MAX_RETRIES = 3

def fetch_items_with_retry() -> List[Dict]:
    for attempt in range(MAX_RETRIES):
        try:
            response = httpx.get(API_URL, timeout=5.0)
            response.raise_for_status()
            return response.json()
        except httpx.RequestError as e:
            wait = 2 ** attempt
            print(f"                                    (                {
               attempt+1}).                        {wait}   ...")
            time.sleep(wait)
    raise RuntimeError("
                                                                      ")

if __name__ == "__main__":
    print(fetch_items_with_retry())
```

Индивидуальное задание: Реализуйте клиентское кэширование с TTL 60 секунд. Добавьте обработку специфичных HTTP-ошибок (404, 429 Too Many Requests) и логирование в JSON-формат. **Вопросы для самопроверки:**

1. Как механизм CORS предотвращает XSS-атаки на уровне браузера?
2. В чём отличие polling от WebSocket для real-time обновлений?

3. Зачем клиентские приложения используют интерцепторы HTTP-запросов?

6 Организация совместной разработки и базовые практики CI/CD

Цель: Научиться управлять версиями кода, проводить code review и автоматизировать сборку проекта. **Задачи:**

- Изучить модели ветвления (GitFlow, trunk-based), стратегии слияния;
- Настроить pipeline для lint, test, build;
- Реализовать автоматический release и деплой на тестовое окружение;
- Проанализировать метрики CI (время сборки, процент успешных runs).

Теоретическое описание: CI/CD автоматизирует интеграцию изменений в основную ветку и доставку в целевые среды. Pipeline состоит из stages: `install` → `lint` → `test` → `build` → `deploy`. Кэширование зависимостей (`actions/cache`) и матричные сборки ускоряют выполнение. Trunk-based development сокращает время feedback loop, требуя высокого покрытия тестами. Кодовые ревью через Pull Request повышают качество кода и распространение знаний в команде. **Разобранный пример:** 1. Платформа: GitHub Actions.
2. Workflow: триггер на push/pull-request в `main`.
3. Этапы: проверка Python, установка зависимостей, запуск pytest, сборка Docker-образа.
4. Результат: pipeline выполняется за 45 с, при падении тестов блокируется merge.
Программный код (генератор CI-конфигурации):

```
import yaml
from pathlib import Path

def generate_ci_config():
    workflow = {
        "name": "CI Pipeline",
        "on": {"push": {"branches": ["main"]}, "pull_request": {"
            branches": ["main"]}},
        "jobs": {
            "test": {
                "runs-on": "ubuntu-latest",
                "steps": [
                    {"uses": "actions/checkout@v4"},
                    {"name": "Set up Python", "uses": "actions/setup-
                        python@v5", "with": {"python-version": "3.11"
                        }},
                    {"run": "pip install -r requirements.txt"},
                    {"run": "pytest --cov=app --cov-report=xml"},
                    {"name": "Upload coverage", "uses": "codecov/
                        codecov-action@v4"}
                ]
            }
        }
    }
    Path(".github/workflows").mkdir(parents=True, exist_ok=True)
    with open(".github/workflows/ci.yml", "w", encoding="utf-8") as f
        :
        yaml.dump(workflow, f, default_flow_style=False, sort_keys=
            False)
```

```
    print("CI                                                    .")

generate_ci_config()
```

Индивидуальное задание: Добавьте этап сборки Docker-образа, кэширование pip и условие выполнения деплоя только при теге `v*`. Настройте уведомления в Telegram/Slack при падении pipeline. **Вопросы для самопроверки:**

1. В каких случаях trunk-based development предпочтительнее GitFlow?
2. Как кэширование зависимостей влияет на стоимость и время CI-запусков?
3. Зачем в pipeline выделяют отдельные stages вместо одного монолитного скрипта?

7 Автоматизированное тестирование, отладка и контроль качества ИС

Цель: Освоить инструменты обеспечения надёжности, поиска дефектов и анализа покрытия кода. **Задачи:**

- Изучить тестовую пирамиду, принципы unit, integration и e2e тестирования;
- Написать параметризованные тесты с фикстурами и mock-объектами;
- Настроить отладчик для пошагового выполнения и инспекции состояния;
- Сгенерировать отчёт о покрытии и статическом анализе.

Теоретическое описание: Тестовая пирамида рекомендует баланс: много unit-тестов, меньше integration, ещё меньше e2e. Mocking изолирует тестируемый модуль от внешних зависимостей (БД, API, ФС). Coverage измеряет процент выполненных строк/ветвей, но не заменяет качество тестов. Статический анализ (mypy, pylint, sonarqube) выявляет потенциальные дефекты до запуска. Отладка с breakpoints позволяет анализировать стеки вызовов, переменные и память в реальном времени. **Разобранный пример:** 1. Фреймворк: pytest + coverage.
2. Тесты: проверка CRUD-логики, обработка ошибок 404/422.
3. Mock: замена реального DB-клиента на in-memory dict.
4. Результат: покрытие 85%, все тесты проходят за <1 с, отчёт в HTML. **Программный код (pytest с фикстурами и mock):**

```
import pytest
from unittest.mock import patch

#
def get_user(db, user_id):
    return db.get(user_id)

def create_order(db, user_id, item_price):
    user = get_user(db, user_id)
    if not user:
        raise ValueError("User not found")
    return {"user_id": user_id, "total": item_price * 1.2}

#                    mock-
@pytest.fixture
def mock_db():
    return {1: {"name": "Alice"}, 2: {"name": "Bob"}}

def test_get_user_exists(mock_db):
    assert get_user(mock_db, 1)["name"] == "Alice"

def test_get_user_not_found(mock_db):
    assert get_user(mock_db, 99) is None

def test_create_order_valid(mock_db):
    order = create_order(mock_db, 1, 100)
    assert order["total"] == 120.0

def test_create_order_invalid_user(mock_db):
    with pytest.raises(ValueError, match="User not found"):
```

```
31          create_order(mock_db, 99, 50)
```

Индивидуальное задание: Добавьте параметризованные тесты (`@pytest.mark.parametrize`), фикстуру scope="session"для инициализации тестового окружения и интеграцию с pytest-html для генерации отчёта. **Вопросы для самопроверки:**

1. Почему 100% code coverage не гарантирует отсутствие багов?
2. В чём разница между mock, stub и fake?
3. Как фикстуры `autouse=True` упрощают поддержку тестов?

8 Контейнеризация, развёртывание и мониторинг информационной системы

Цель: Научиться деплоить ИС в изолированной среде и настраивать базовый мониторинг работоспособности. **Задачи:**

- Изучить принципы контейнеризации, слои образов, сети и volumes;
- Написать Dockerfile и docker-compose.yml для multi-container приложения;
- Настроить healthcheck, resource limits и log rotation; сбор метрик (CPU, RAM, HTTP-latency) через Prometheus/Grafana.

Теоретическое описание: Контейнеры изолируют процессы через namespaces и cgroups, обеспечивая переносимость сред. Multi-stage builds уменьшают размер образов. Docker Compose оркестрирует локальные сервисы. Healthcheck позволяет orchestrator'у принимать решения о restart. Мониторинг делится на metrics (количественные данные), logs (события) и traces (распределённые вызовы). Prometheus собирает time-series данные, Grafana визуализирует. Alerting предупреждает о деградации до инцидента. **Разобранный пример:** 1. Стек: FastAPI + Redis + Docker Compose.
2. Dockerfile: multi-stage, non-root user, healthcheck.
3. Compose: сервисы app, redis, prometheus.
4. Результат: `docker compose up` поднимает стек, /health возвращает 200, метрики доступны на :9090. **Программный код (скрипт валидации docker-compose и healthcheck):**

```
import subprocess
import json
import httpx
import time

def run_compose_up():
    subprocess.run(["docker", "compose", "up", "-d", "--build"],
       check=True)
    time.sleep(5)  #

def check_health(endpoint="http://localhost:8000/health"):
    try:
        resp = httpx.get(endpoint, timeout=3)
        return resp.status_code == 200
    except Exception:
        return False

def collect_metrics():
    #                                    Prometheus API
    url = "http://localhost:9090/api/v1/query?query=up{job='app'}"
    resp = httpx.get(url)
    return resp.json()

if __name__ == "__main__":
    run_compose_up()
    print("Health OK:", check_health())
    print("Metrics:", collect_metrics())
```

Индивидуальное задание: Добавьте в Dockerfile multi-stage сборку, настройте `deploy.resources.limits` в compose для CPU/RAM, реализуйте ротацию логов через `json-file` driver и настройте alert в Prometheus на $\texttt{http}_r equests_t otal > 100/s$.Вопросы для самопроверки:

Чем контейнер отличается от виртуальной машины на уровне ядра ОС?

Зачем в production-образах используют non-root пользователя?

Как healthcheck влияет на стратегию zero-downtime деплоя?

Список литературы

[1] Martin R.C. Clean Architecture: A Craftsman's Guide to Software Structure and Design. Prentice Hall, 2017. 432 p.

[2] Fowler M. Patterns of Enterprise Application Architecture. Addison-Wesley, 2002. 560 p.

[3] Kernighan B.W., Pike R. The Practice of Programming. Addison-Wesley, 1999. 288 p.

[4] Summers D.C. Software Engineering. Pearson, 2021. 700 p.

[5] Docker Inc. Docker Documentation. URL: `https://docs.docker.com/`.

[6] FastAPI Documentation. URL: `https://fastapi.tiangolo.com/`.

[7] pytest Documentation. URL: `https://docs.pytest.org/`.

[8] GitHub Actions Documentation. URL: `https://docs.github.com/en/actions`.

[9] Prometheus Monitoring System. URL: `https://prometheus.io/docs/`.

[10] SQLAlchemy Documentation. URL: `https://docs.sqlalchemy.org/`.