МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ
УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«СЕВЕРО-КАВКАЗСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»

МЕТОДИЧЕСКИЕ УКАЗАНИЯ

по выполнению практических работ
по дисциплине «Архитектура и проектирование высоконагруженных и отказоустойчивых систем» для студентов направления подготовки 09.04.04
Программная инженерия
Направленность (профиль)
«Разработка, мониторинг и управление жизненным циклом инженерных систем»

Ставрополь 2026

Оглавление

Введение

Дисциплина «Архитектура и проектирование высоконагруженных и отказоустойчивых систем» (Б1.В.02) является одной из ключевых дисциплин вариативной части образовательной программы магистратуры 09.04.04 Программная инженерия, профиль «Разработка, мониторинг и управление жизненным циклом инженерных систем».

Лабораторный практикум представляет собой последовательный набор из 18 лабораторных работ, охватывающих все основные темы дисциплины: от фундаментальных понятий распределённых систем (CAP-теорема, ACID, BASE) до практического проектирования архитектур реальных high-load систем (e-commerce, доставка еды) и их защиты.

Цель лабораторных работа: Формирование у магистрантов практических навыков проектирования, анализа, реализации и тестирования высоконагруженных и отказоустойчивых систем с использованием современных архитектурных паттернов, технологий и инструментов, а также закрепление теоретических знаний, полученных на лекционных занятиях.

Задачи лабораторных работ:

1. **Освоение инструментария:**
   - Контейнеризация и оркестрация (Docker, Docker Compose)
   - Балансировка нагрузки (Nginx, API Gateway)
   - Базы данных (PostgreSQL, MongoDB, Cassandra, Redis)
   - Очереди сообщений (RabbitMQ, Kafka)
   - Нагрузочное тестирование (k6)
   - Документирование архитектуры (PlantUML, C4 model, arc42)
2. **Применение архитектурных паттернов:**
   - Горизонтальное масштабирование stateless-сервисов
   - Репликация и шардирование данных
   - Circuit Breaker, Retry, Timeout
   - CQRS, Saga (хореография и оркестрация)

- Кэширование (Cache-Aside, Write-Through, Write-Behind)
- API Gateway (маршрутизация, агрегация, rate limiting, JWT)

3. **Экспериментальное исследование:**
   - Анализ CAP-теоремы на примерах CP и AP систем
   - Сравнение синхронной и асинхронной репликации БД
   - Сравнение монолитной и микросервисной архитектур
   - Сравнение SQL и NoSQL баз данных
   - Нагрузочное тестирование с разными сценариями
4. **Формирование компетенций (ПК-5):**
   - Проектировать эксперименты по проверке устойчивости систем
   - Планировать и проводить нагрузочное тестирование
   - Анализировать результаты экспериментов
   - Формулировать и внедрять корректирующие меры
   - Повышать отказоустойчивость и устойчивость к сбоям

Структура лабораторного практикума

Лабораторный практикум состоит из 18 работ, объединённых в три логических блока.

Блок 1. Основы архитектуры и масштабирования (работы №1–6)

	Название работы	**Формируемые компетенции**
	Документирование архитектуры с использованием C4 model	ПК-5 $_{ИД-1}$ ПК-5 $_{ИД-4}$
	Проектирование и развертывание горизонтально масштабируемого stateless-сервиса	ПК-5 $_{ИД-1}$ ПК-5 $_{ИД-2}$ ПК-5 $_{ИД-5}$
	Анализ CAP-теоремы на практике	ПК-5 $_{ИД-1}$

	Название работы	Формируемые компетенции
		ПК-5 ИД-2 ПК-5 ИД-3 ПК-5 ИД-4
	Реализация паттерна Circuit Breaker	ПК-5 ИД-2 ПК-5 ИД-4 ПК-5 ИД-5 ПК-5 ИД-6
	API Gateway: маршрутизация, агрегация и безопасность	ПК-5 ИД-1 ПК-5 ИД-2 ПК-5 ИД-5 ПК-5 ИД-7
	Сравнительный анализ монолитной и микросервисной архитектур	ПК-5 ИД-1 ПК-5 ИД-4 ПК-5 ИД-5 ПК-5 ИД-6 ПК-5 ИД-7

Блок 2. Управление данными и асинхронность (работы №7–12)

№	Название работы	Формируемые компетенции
	Репликация баз данных: синхронная и асинхронная	ПК-5 ИД-1 ПК-5 ИД-4 ПК-5 ИД-5 ПК-5 ИД-6

№	Название работы	Формируемые компетенции
	Шардирование данных (горизонтальное партиционирование)	ПК-5 ИД-1 ПК-5 ИД-4 ПК-5 ИД-5 ПК-5 ИД-6 ПК-5 ИД-7
	Асинхронная обработка с очередями сообщений (RabbitMQ)	ПК-5 ИД-1 ПК-5 ИД-4 ПК-5 ИД-5 ПК-5 ИД-7
0	Кэширование с использованием Redis	ПК-5 ИД-1 ПК-5 ИД-4 ПК-5 ИД-5 ПК-5 ИД-6
1	Сравнительный анализ SQL и NoSQL баз данных	ПК-5 ИД-1 ПК-5 ИД-4 ПК-5 ИД-5 ПК-5 ИД-6
2	Реализация паттерна Saga для распределённых транзакций	ПК-5 ИД-1 ПК-5 ИД-4 ПК-5 ИД-5 ПК-5 ИД-6 ПК-5 ИД-7

Блок 3. Анализ, тестирование и защита проекта (работы №13–18)

	Название работы	**Формируемые компетенции**
3	Архитектурный анализ методом ATAM	ПК-5 ИД-1 ПК-5 ИД-4 ПК-5 ИД-6 ПК-5 ИД-7
4	Нагрузочное тестирование с k6	ПК-5 ИД-1 ПК-5 ИД-2 ПК-5 ИД-3 ПК-5 ИД-4 ПК-5 ИД-5 ПК-5 ИД-6 ПК-5 ИД-7
5	Документирование архитектуры с использованием arc42	ПК-5 ИД-1 ПК-5 ИД-4 ПК-5 ИД-6 ПК-5 ИД-7
6	Архитектурный кейс: e-commerce платформа	ПК-5 ИД-1 ПК-5 ИД-2 ПК-5 ИД-3 ПК-5 ИД-4 ПК-5 ИД-5 ПК-5 ИД-6 ПК-5 ИД-7
7	Архитектурный кейс: система доставки	ПК-5 ИД-1 ПК-5 ИД-2

	Название работы	Формируемые компетенции
		ПК-5 ИД-3 ПК-5 ИД-4 ПК-5 ИД-5 ПК-5 ИД-6 ПК-5 ИД-7
8	Проектирование, защита и обзор курса	ПК-5 ИД-1 ПК-5 ИД-2 ПК-5 ИД-3 ПК-5 ИД-4 ПК-5 ИД-5 ПК-5 ИД-6 ПК-5 ИД-7

4. Уровни сложности

Каждая лабораторная работа (кроме работ №16–18) предусматривает **три уровня сложности**:

Уровень	Описание	Требования
Базовый (Basic)	Выполнение минимально необходимого набора заданий для получения зачёта	Реализация основного функционала, базовые сценарии
Средний (Intermediate)	Расширенное задание с дополнительными требованиями	Дополнительные сценарии, расширенная функциональность, использование

Уровень	Описание	Требования
		дополнительных инструментов
Продвинутый (Advanced)	Полная реализация с исследовательской или проектной составляющей	Дополнительные механизмы отказоустойчивости, мониторинг, автоматизация, интеграция с CI/CD

Студент выбирает уровень сложности самостоятельно или по согласованию с преподавателем. Уровень сложности влияет на максимальное количество баллов за работу и, соответственно, на итоговую оценку.

Требования к выполнению лабораторных работ

1. Общие требования

1. **Самостоятельность:** все работы выполняются индивидуально (или в парах — по согласованию с преподавателем).
2. **Работоспособность:** код должен успешно запускаться и выполнять заявленную функциональность.
3. **Документирование:** каждая работа сопровождается отчётом в формате Markdown, содержащим:
   - Цель работы
   - Теоретическое обоснование
   - Листинги кода (скриптов, конфигураций)
   - Скриншоты результатов выполнения
   - Анализ результатов
   - Выводы
   - Ответы на контрольные вопросы

4. **Версионирование:** рекомендуется хранить все работы в репозитории Git (GitHub / GitLab).

2. Технические требования

1. **Контейнеризация:** все многокомпонентные приложения должны быть описаны с использованием Docker Compose.

2. **Воспроизводимость:** должна быть предоставлена инструкция по запуску (команды, порядок шагов).

3. **Чистота кода:** код должен быть структурирован, снабжён комментариями (на английском или русском).

4. **Безопасность:** секреты (пароли, токены) не должны храниться в коде (использовать переменные окружения).

3. Рекомендуемое программное обеспечение

Для успешного выполнения лабораторных работ потребуется следующее ПО (всё доступно в РФ на законных основаниях):

Категория	Программное обеспечение	Лицензия
Контейнеризация	Docker Desktop / Docker Engine	Free for education
Оркестрация	Docker Compose, Kubernetes (Minikube/Kind)	Open Source
Редактор кода	VS Code, PyCharm Community	Free
Диаграммы	PlantUML, Mermaid, Draw.io	Open Source / Free
Языки программирования	Python 3.10+, Node.js 18+, Go 1.19+	Open Source
Базы данных	PostgreSQL, MongoDB, Cassandra,	Open Source

Категория	Программное обеспечение	Лицензия
	Redis	
Очереди	RabbitMQ, Apache Kafka	Open Source
Тестирование	k6, wrk	Open Source
Мониторинг	Prometheus, Grafana	Open Source
CI/CD	GitLab CI, GitHub Actions	Free plan

Система оценивания

1. Балльная структура каждой работы

Критерий	**Базовый (макс 10)**	**Средний (макс 14)**	**Продвинутый (макс 18)**
Работоспособность	5	5	4
Полнота реализации	3	4	4
Качество документации	2	3	4
Дополнительные требования	—	2	6
Итого	**10**	**14**	**18**

2. Шкала перевода баллов в оценку

Оценка	**Базовый**	**Средний**	**Продвинутый**
5 (отлично)	9–10	13–14	16–18

Оценка	Базовый	Средний	Продвинутый
4 (хорошо)	7–8	10–12	12–15
3 (удовлетворительно)	5–6	7–9	8–11
2 (неудовлетворительно)	<5	<7	<8

3. Итоговая оценка за лабораторный практикум

Итоговая оценка формируется как среднее арифметическое оценок за 18 лабораторных работ:

Оценка = (Σ оценок за ЛР1–ЛР18) / 18

Порядок выполнения и сдачи работ

1. Этапы выполнения

1. **Изучение теоретического материала** (лекции, рекомендованная литература).
2. **Выбор уровня сложности** (базовый / средний / продвинутый).
3. **Получение варианта задания** (согласно списку группы).
4. **Разработка и отладка** кода / конфигураций.
5. **Подготовка отчёта** в формате Markdown.
6. **Демонстрация работы** преподавателю (live демонстрация или видео-запись).
7. **Защита работы** (ответы на контрольные вопросы).

Порядок защиты

Защита лабораторной работы включает:

1. **Демонстрацию работоспособности** (запуск кода, демонстрация результатов).

2. **Ответы на контрольные вопросы** (2–3 вопроса из списка к работе).

3. **Обсуждение принятых решений** и альтернативных подходов.

Работа считается принятой только при положительной оценке за все три компонента защиты.

Структура отчёта по лабораторной работе

Каждый отчёт должен содержать следующие разделы:

Лабораторная работа №X. [Название работы]

1. Цель работы
2. Теоретическое обоснование
3. Практическая часть
   3.1. Исходные данные (вариант)
   3.2. Листинг кода / конфигураций
   3.3. Инструкция по запуску
4. Результаты выполнения
   4.1. Скриншоты
   4.2. Таблицы / графики
   4.3. Анализ результатов
5. Ответы на контрольные вопросы
6. Заключение

Литература для подготовки

Основная литература

1. Ньюмен, С. Создание микросервисов / С. Ньюмен. — 2-е изд. — СПб.: Питер, 2024. — 622 с.

2. Ричардсон, К. Микросервисы : паттерны разработки и рефакторинга / К. Ричардсон. — СПб.: Питер, 2024. — 544 с.

3. Клеппман, М. Проектирование систем, интенсивно работающих с данными / М. Клеппман. — Электронное издание (перевод сообщества Vonng). — Доступ: https://github.com/Vonng/ddia

4. Нюгор, М. Т. Release It! : проектирование и развертывание готовых к production программных систем / М. Т. Нюгор. — СПб.: Питер, 2025.

5. 9.2. Дополнительная литература

6. Beyer, B. Site Reliability Engineering : как Google управляет production-системами / B. Beyer, C. Jones, J. Petoff, N. R. Murphy. — Доступ: https://sre.google/books

7. Majors, C. Observability Engineering / C. Majors, L. Fong-Jones, G. Miranda. — Доступ: https://github.com/observability-engineering

8. 9.3. Интернет-ресурсы

9. C4 model documentation — https://c4model.com/

10. PlantUML C4 library — https://github.com/plantuml-stdlib/C4-PlantUML

11. k6 Load Testing — https://k6.io/docs/

12. Redis Documentation — https://redis.io/documentation

13. RabbitMQ Documentation — https://www.rabbitmq.com/documentation.html

14. Apache Kafka Documentation — https://kafka.apache.org/documentation/

15. Prometheus Documentation — https://prometheus.io/docs/

16. Grafana Documentation — https://grafana.com/docs/

17. Resilience4j Documentation — https://resilience4j.readme.io/

2. ПРИМЕРЫ ПРАКТИЧЕСКОГО ВЫПОЛНЕНИЯ

2.1. Пример системы заказов с уведомлениями

2.1.1. Структура проекта

```
rabbitmq-demo/
├── docker-compose.yml
├── producer_service/
│   ├── app.py          # FastAPI сервис заказов
│   ├── requirements.txt
│   └── Dockerfile
├── consumer_service/
│   ├── consumer.py     # Consumer уведомлений
│   ├── requirements.txt
│   └── Dockerfile
└── consumer_dlq.py     # Consumer DLQ
```

2.1.2. Docker Compose (docker-compose.yml)

```yaml
version: '3.8'

services:
  rabbitmq:
    image: rabbitmq:3-management
    container_name: rabbitmq
    ports:
      - "5672:5672"      # AMQP порт
      - "15672:15672"    # Management UI
    environment:
      RABBITMQ_DEFAULT_USER: guest
      RABBITMQ_DEFAULT_PASS: guest
    volumes:
      - rabbitmq_data:/var/lib/rabbitmq

  producer:
    build: ./producer_service
    container_name: order-service
    ports:
      - "8000:8000"
    environment:
      RABBITMQ_HOST: rabbitmq
```

```
    depends_on:
      - rabbitmq

  consumer:
    build: ./consumer_service
    container_name: notification-service
    environment:
      RABBITMQ_HOST: rabbitmq
    depends_on:
      - rabbitmq
    deploy:
      replicas: 2  # Два экземпляра consumer (competing consumers)

volumes:
  rabbitmq_data:
```

2.1.3. Producer сервис (producer_service/app.py)

```
import json
import pika
import uuid
from fastapi import FastAPI, HTTPException
from pydantic import BaseModel
from contextlib import contextmanager

app = FastAPI(title="Order Service")

# Подключение к RabbitMQ
RABBITMQ_HOST = "rabbitmq"
ORDER_QUEUE = "orders"
EXCHANGE_NAME = "orders_exchange"
ROUTING_KEY = "order.created"

@contextmanager
def get_rabbit_connection():
    connection = pika.BlockingConnection(
        pika.ConnectionParameters(host=RABBITMQ_HOST)
    )
    try:
        yield connection
    finally:
        connection.close()

class OrderRequest(BaseModel):
    user_id: int
    product: str
    amount: float

class OrderResponse(BaseModel):
    order_id: str
```

```python
    status: str

@app.post("/api/orders", response_model=OrderResponse)
async def create_order(order: OrderRequest):
    """Создание заказа с отправкой сообщения в очередь"""

    # Генерация ID заказа
    order_id = str(uuid.uuid4())

    # Формирование сообщения
    message = {
        "order_id": order_id,
        "user_id": order.user_id,
        "product": order.product,
        "amount": order.amount,
        "status": "created"
    }

    # Отправка в RabbitMQ
    with get_rabbit_connection() as connection:
        channel = connection.channel()

        # Объявление exchange (topic)
        channel.exchange_declare(
            exchange=EXCHANGE_NAME,
            exchange_type='topic',
            durable=True
        )

        # Отправка сообщения
        channel.basic_publish(
            exchange=EXCHANGE_NAME,
            routing_key=ROUTING_KEY,
            body=json.dumps(message),
            properties=pika.BasicProperties(
                delivery_mode=2,  # Persistent message
                content_type='application/json'
            )
        )

        print(f" [x] Sent order {order_id}")

    return OrderResponse(order_id=order_id, status="queued")

@app.get("/api/health")
async def health():
    return {"status": "ok"}

if __name__ == "__main__":
    import uvicorn
    uvicorn.run(app, host="0.0.0.0", port=8000)
```

2.1.4. Producer Dockerfile (producer_service/Dockerfile)

```
FROM python:3.11-slim

WORKDIR /app

COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt

COPY app.py .

CMD ["python", "app.py"]
```

2.1.5. Consumer сервис (consumer_service/consumer.py)

```
import json
import pika
import time
import smtplib
from email.mime.text import MIMEText

RABBITMQ_HOST = "rabbitmq"
ORDER_QUEUE = "orders_queue"
EXCHANGE_NAME = "orders_exchange"
ROUTING_KEYS = ["order.created", "order.paid", "order.shipped"]

# Настройки email (имитация)
SMTP_SERVER = "smtp.example.com"  # Заглушка

def send_email_notification(user_id: int, message: str):
    """Имитация отправки email"""
    print(f" [✉] Sending email to user {user_id}: {message}")
    # В реальности здесь был бы вызов SMTP
    time.sleep(0.5)  # Имитация работы

def process_order_created(message: dict):
    """Обработка заказа"""
    order_id = message.get("order_id")
    user_id = message.get("user_id")
    product = message.get("product")

    print(f" [✓] Processing order {order_id} for user {user_id}")

    # Имитация разных типов ошибок
    if order_id.endswith("0"):
        raise Exception(f"Random failure for order {order_id}")

    # Отправка уведомления
    send_email_notification(
```

```
        user_id,
        f"Order {order_id} created: {product}"
    )

    return True

def callback(ch, method, properties, body):
    """Функция обработки сообщения"""
    try:
        message = json.loads(body)
        print(f" [x] Received {message}")

        # Обработка в зависимости от routing key
        routing_key = method.routing_key

        if routing_key == "order.created":
            process_order_created(message)
        elif routing_key == "order.paid":
            # Обработка оплаты
            print(f" [✓] Payment confirmed for {message['order_id']}")
        elif routing_key == "order.shipped":
            # Обработка отгрузки
            print(f" [✓] Order {message['order_id']} shipped")

        # Подтверждение успешной обработки
        ch.basic_ack(delivery_tag=method.delivery_tag)
        print(f" [✓] Acknowledged {message['order_id']}")

    except Exception as e:
        print(f" [X] Error processing message: {e}")
        # Отказ от сообщения без перезагрузки (отправка в DLQ)
        ch.basic_nack(delivery_tag=method.delivery_tag, requeue=False)
        print(f" [💀] Message sent to Dead Letter Queue")

def main():
    print("Starting Notification Service Consumer...")

    connection = pika.BlockingConnection(
        pika.ConnectionParameters(host=RABBITMQ_HOST)
    )
    channel = connection.channel()

    # Объявление exchange (должен существовать)
    channel.exchange_declare(
        exchange=EXCHANGE_NAME,
        exchange_type='topic',
        durable=True
    )

    # Объявление очереди с поддержкой Dead Letter Exchange
    channel.queue_declare(
```

```
        queue=ORDER_QUEUE,
        durable=True,
        arguments={
            'x-dead-letter-exchange': 'dlx_exchange',
            'x-dead-letter-routing-key': 'dead.letter'
        }
    )

    # Привязка очереди к exchange с несколькими routing keys
    for key in ROUTING_KEYS:
        channel.queue_bind(
            exchange=EXCHANGE_NAME,
            queue=ORDER_QUEUE,
            routing_key=key
        )

    # Настройка QoS (prefetch count)
    channel.basic_qos(prefetch_count=1)

    # Начало потребления
    channel.basic_consume(
        queue=ORDER_QUEUE,
        on_message_callback=callback,
        auto_ack=False
    )

    print(" [*] Waiting for messages. To exit press CTRL+C")

    try:
        channel.start_consuming()
    except KeyboardInterrupt:
        print("\n [*] Stopping consumer...")
        connection.close()

if __name__ == "__main__":
    main()
```

2.1.6. Dead Letter Queue Consumer (consumer_dlq.py)

```
import json
import pika

RABBITMQ_HOST = "rabbitmq"
DLQ_QUEUE = "dead_letter_queue"
DLX_EXCHANGE = "dlx_exchange"

def callback(ch, method, properties, body):
    """Обработка сообщений из DLQ"""
    message = json.loads(body)
```

```
    print(f" [💀] Dead letter received: {message}")

    # Анализ причины ошибки (можно из заголовков)
    death_info = properties.headers.get('x-death', [])
    if death_info:
        print(f"     Reason: {death_info[0].get('reason')}")
        print(f"     Original exchange: {death_info[0].get('exchange')}")
        print(f"     Original routing key: {death_info[0].get('routing-keys')}")

    # Логирование в файл
    with open("/log/db_errors.log", "a") as log_file:
        log_file.write(f"{message}\n")

    # Подтверждение обработки
    ch.basic_ack(delivery_tag=method.delivery_tag)

def main():
    connection = pika.BlockingConnection(
        pika.ConnectionParameters(host=RABBITMQ_HOST)
    )
    channel = connection.channel()

    # Объявление Dead Letter Exchange
    channel.exchange_declare(
        exchange=DLX_EXCHANGE,
        exchange_type='direct',
        durable=True
    )

    # Объявление Dead Letter Queue
    channel.queue_declare(
        queue=DLQ_QUEUE,
        durable=True
    )

    # Привязка
    channel.queue_bind(
        exchange=DLX_EXCHANGE,
        queue=DLQ_QUEUE,
        routing_key='dead.letter'
    )

    channel.basic_consume(
        queue=DLQ_QUEUE,
        on_message_callback=callback,
        auto_ack=False
    )

    print(" [💀] Dead Letter Consumer started. Waiting for failed messages...")
    channel.start_consuming()
```

```
if __name__ == "__main__":
    main()
```

2.1.7. Тестовый скрипт (test_rabbitmq.py)

```
import requests
import time
import threading

BASE_URL = "http://localhost:8000"

def send_order(user_id: int, product: str, amount: float):
    """Отправка заказа"""
    response = requests.post(
        f"{BASE_URL}/api/orders",
        json={
            "user_id": user_id,
            "product": product,
            "amount": amount
        }
    )
    return response.json()

def load_test(num_orders: int = 100):
    """Нагрузочное тестирование"""
    print(f"Отправка {num_orders} заказов...")
    start = time.time()

    threads = []
    for i in range(num_orders):
        t = threading.Thread(
            target=send_order,
            args=(i % 10, f"Product_{i}", 100.0 * (i % 5))
        )
        threads.append(t)
        t.start()

    for t in threads:
        t.join()

    elapsed = time.time() - start
    print(f"Отправлено {num_orders} заказов за {elapsed:.2f} сек")
    print(f"RPS: {num_orders / elapsed:.2f}")

if __name__ == "__main__":
    print("=== Тестирование очередей RabbitMQ ===")

    # Проверка здоровья сервиса
    health = requests.get(f"{BASE_URL}/api/health")
```

```
print(f"Health check: {health.json()}")

# Отправка одного заказа
order = send_order(1, "Test Product", 99.99)
print(f"Создан заказ: {order}")
time.sleep(1)

# Нагрузочное тестирование
load_test(100)
```

МЕТОДИЧЕСКИЕ УКАЗАНИЯ ДЛЯ ВЫПОЛНЕНИЯ ПРАКТИЧЕСКОЙ ЧАСТИ РАБОТЫ

Задачи работы:

1. **Развёртывание RabbitMQ:**
   - Запустить RabbitMQ с Management UI
   - Создать очереди, exchanges, bindings
2. **Реализация Producer:**
   - Отправка сообщений в очередь
   - Подтверждения (publisher confirms)
   - Персистентные сообщения
3. **Реализация Consumer:**
   - Потребление сообщений из очереди
   - Подтверждение (ACK/NACK)
   - Обработка ошибок и DLQ
4. **Эксперимент 1: Competing Consumers:**
   - Запустить 2+ экземпляра consumer
   - Нагрузить очередь
   - Пронаблюдать распределение нагрузки
5. **Эксперимент 2: Dead Letter Queue:**
   - Сымитировать ошибку в consumer
   - Пронаблюдать попадание сообщения в DLQ
6. **Эксперимент 3: Publish/Subscribe (Fanout):**
   - Создать fanout exchange
   - Настроить несколько очередей (логирование, аналитика)

- Продемонстрировать доставку всем подписчикам

3. ВАРИАНТЫ ЗАДАНИЙ

3.1. Базовый уровень (15 вариантов)

Требования: Producer → Queue → Consumer (Work Queue)

№	Тип сообщения	Consumer операций	Дополнительно
1	Order (заказ)	Email уведомление	ACK
2	Order (заказ)	SMS уведомление	ACK
3	Order (заказ)	Telegram уведомление	ACK
4	Task (задача)	Обработка (sleep 2s)	ACK
5	Task (задача)	Обработка (sleep 5s)	ACK
6	Task (задача)	Обработка (sleep 10s)	ACK
7	Log (лог)	Запись в файл	Отсутствие ACK
8	Log (лог)	Запись в БД (SQLite)	Отсутствие ACK
9	Metric (метрика)	Prometheus push	NACK при ошибке
10	Metric (метрика)	InfluxDB write	NACK при ошибке
11	Stock (склад)	Update inventory	ACK

№	Тип сообщения	Consumer операций	Дополнительно
12	Stock (склад)	Reorder trigger	ACK
13	Image (изображение)	Resize + save	ACK
14	Image (изображение)	Thumbnail gen	ACK
15	Job (работа)	Archive to S3 (minio)	ACK

3.2. Средний уровень (15 вариантов)

Требования: как базовый + Dead Letter Queue + 2+ routing keys

№	Exchange тип	Routing keys	DLQ использование
1	Direct	order.created, order.paid	Да (NACK → DLQ)
2	Direct	user.signup, user.update	Да (NACK → DLQ)
3	Direct	stock.low, stock.out	Да (NACK → DLQ)
4	Topic	*.error, *.warning	Да (TTL → DLQ)
5	Topic	*.created.*, *.deleted.*	Да (TTL → DLQ)
6	Topic	log.*.error	Да (превышение длины)
7	Fanout	(не используется)	Да (NACK → DLQ)
8	Fanout	(не используется)	Да (TTL → DLQ)

№	Exchange тип	Routing keys	DLQ использование
9	Headers	user_country = RU/US	Да (NACK → DLQ)
10	Headers	priority = high/low	Да (превышение длины)
11	Direct + Topic	order.* (topic) + direct	Да (с отдельным DLX)
12	Fanout + Direct	fanout (все) + direct	Да (с отдельным DLX)
13	Topic + Dead Letter	*.payment, *.refund	Да (DLQ анализ ошибок)
14	Headers + DLQ	content_type = image/*	Да (DLQ аналитика)
15	Direct + Retry	order.created (retry 3x)	Да (после retry → DLQ)

3.3. Продвинутый уровень (15 вариантов)

Требования: средний уровень + competing consumers + publisher confirms + потребители на разных языках

№	Ролл	Сценарий	Требования
1	Order payment	Двухфазная обработка	Producer: confirms; Consumer: ack
2	Distributed Saga	Choreography (события)	3 сервиса, общий exchange

№	Ролл	Сценарий	Требования
3	Retry + Exponential backoff	Очередь retry (3 попытки)	После 3 попыток → DLQ
4	Priority Queue	Приоритет заказов (VIP / обычные)	2 очереди (high/low)
5	Request-Reply	Взаимодействие с ответом	Correlation ID + reply queue
6	RPC over RabbitMQ	Удалённый вызов с ожиданием ответа	Ожидание ответа (timeout 30s)
7	Stream Processing	Агрегация окон (10 сообщений)	Буферизация + batch в БД
8	Multi-language	Producer (Python) + Consumer (Node.js)	ACK, DLQ на Node.js
9	Multi-language	Producer (Go) + Consumer (Java)	ACK, DLQ на Java
10	Kubernetes + RabbitMQ	Развёртывание в K8s	RabbitMQ operator (cluster)
11	RabbitMQ Cluster	Кластер из 3 узлов	High availability queues
12	Consumer cc Circut Breaker	Защита при падении БД	Consumer отключает очередь

№	Ролл	Сценарий	Требования
13	Idempotent consumer	Повторная обработка (дубликаты)	Дедупликация по ID сообщения
14	Exactly-once	Транзакции RabbitMQ + БД	Транзакционный outbox
15	Stream (Kafka style)	Сохранение порядка (partition key)	Consistent hashing exchange

4. КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Что такое асинхронная обработка и для чего нужны очереди сообщений?
2. Какие компоненты входят в архитектуру RabbitMQ?
3. Чем отличается Direct exchange от Topic exchange?
4. Для чего нужен Fanout exchange?
5. Что такое Dead Letter Queue (DLQ) и как она настраивается?
6. В чём разница между ACK и NACK?
7. Что такое Publisher Confirms?
8. Как обеспечить сохранность сообщения при перезапуске RabbitMQ?
9. Как масштабировать consumer'ов горизонтально?
10. Что такое competing consumers?
11. Как ограничить количество необработанных сообщений на одного consumer (prefetch count)?
12. Как реализовать отложенную обработку сообщений (Delayed Message)?
13. Как обрабатывать сообщения в правильном порядке (FIFO) в распределённой очереди?

14. В чём разница между RabbitMQ и Kafka?
15. Как мониторить RabbitMQ (Management UI, метрики)?

5. ТРЕБОВАНИЯ К ОТЧЕТУ

1. **Титульный лист**
2. **Цель работы**
3. **Теоретическая часть:** очереди, RabbitMQ, exchanges, DLQ
4. **Практическая часть:**
   - Листинг docker-compose.yml
   - Листинг producer
   - Листинг consumer
   - Листинг DLQ consumer
5. **Результаты экспериментов:**
   - Скриншот RabbitMQ Management UI (очереди)
   - Распределение сообщений между несколькими consumer
   - Сообщение в DLQ (скриншот, лог)
6. **Анализ:** преимущества асинхронной обработки, отказоустойчивость
7. **Вывод**
8. **Ответы на контрольные вопросы**

6. КРИТЕРИИ ОЦЕНИВАНИЯ

Критерий	**Базовый (10)**	**Средний (14)**	**Продвинутый (18)**
RabbitMQ развёрнут	2	1	1
Producer работает	2	2	1

Критерий	Базовый (10)	Средний (14)	Продвинутый (18)
Consumer работает (ACK/NACK)	3	2	1
Dead Letter Queue настроена	—	3	2
Competing consumers	—	3	2
Publisher confirms	—	—	2
Multi-language	—	—	3
Idempotency/de-duplication	—	—	2
Качество отчёта	3	3	4
Максимум	**10**	**14**	**18**

Шкала перевода в оценку:

- **5 (отлично):** 90%+
- **4 (хорошо):** 75–89%
- **3 (удовлетворительно):** 60–74%
- **2 (неудовлетворительно):** <60%

7. ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ

Программа	Назначение	Доступность в РФ

Программа	Назначение	Доступность в РФ
Docker + Docker Compose	Контейнеризация	+
RabbitMQ (официальный образ)	Брокер сообщений	+
Python + pika	Producer/Consumer	+
Node.js + amqplib	Consumer (мультиязычный)	+
curl / Postman	Тестирование	+

Лабораторная работа №10

Кэширование с использованием Redis

Цель работы: Сформировать практические навыки применения различных стратегий кэширования (cache-aside, write-through, write-behind) с использованием Redis для повышения производительности высоконагруженных систем, а также освоить политики вытеснения данных, работу с TTL и решение проблемы «пробивания кэша» (cache stampede/thundering herd).

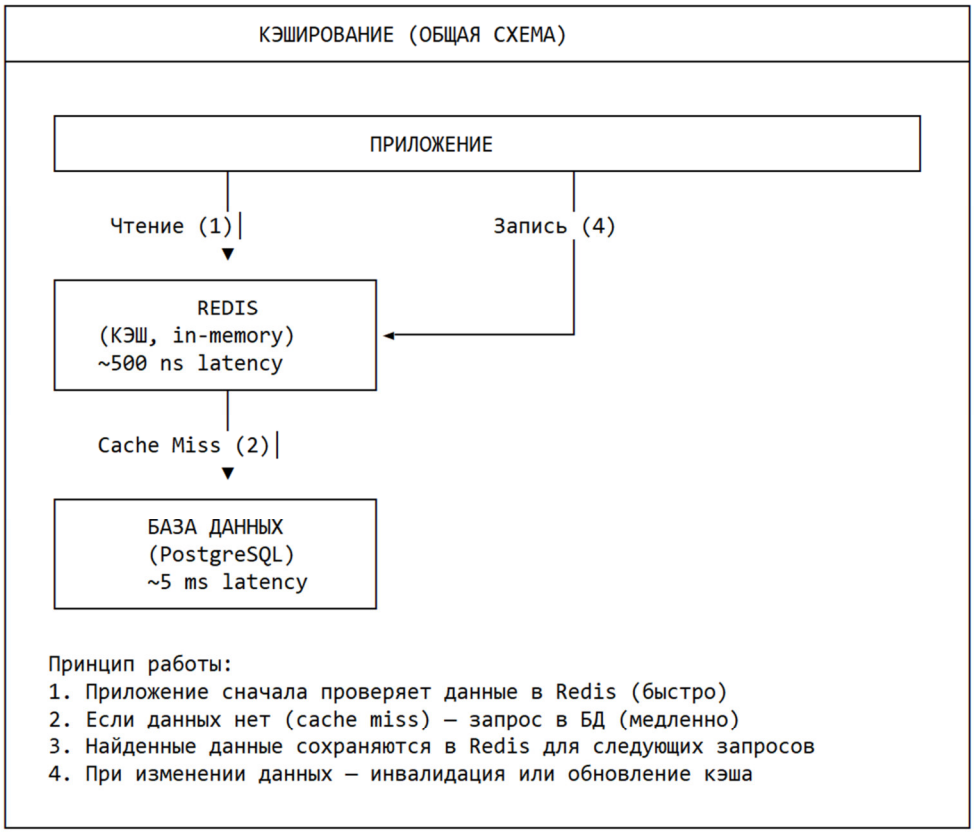
Формируемые компетенции: ПК-5

1. ТЕОРЕТИЧЕСКОЕ ОБОСНОВАНИЕ

1.1. Понятие кэширования и его роль в высоконагруженных системах

Кэширование - это техника временного хранения часто запрашиваемых данных в быстром хранилище (обычно in-memory) для

сокращения времени доступа и снижения нагрузки на основное хранилище (базу данных).



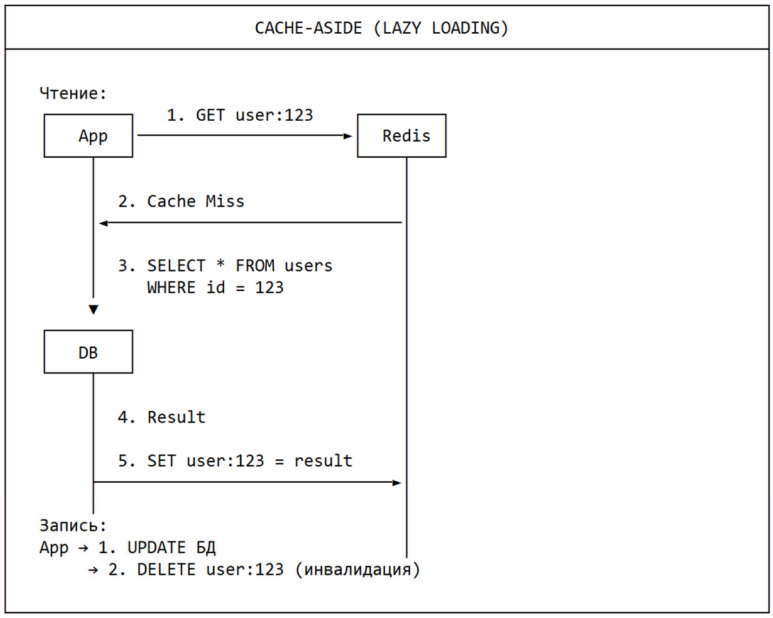
Преимущества кэширования:

Преимущество	**Описание**
Снижение latency	In-memory хранилища в 10-100 раз быстрее дисковых БД
Повышение throughput	Кэш выдерживает гораздо больше RPS (100k+ против 5-10k)
Снижение нагрузки на БД	БД занимается только записью и сложными запросами
Экономия ресурсов	Меньше CPU/IO на основном сервере БД

1.2. Стратегии кэширования

1.2.1. Cache-Aside (Lazy Loading)

Самая распространённая стратегия. Приложение само управляет кэшем: при чтении сначала проверяет кэш, при промахе — загружает из БД и сохраняет.



Преимущества	Недостатки
Простота реализации	Первый запрос всегда медленный (cache miss)
Кэширует только нужные данные	Требует ручного управления инвалидацией
Хорош для read-heavy сценариев	Данные могут быть неконсистентными

1.2.2. Read-Through

Кэш выступает как прокси: приложение всегда обращается к кэшу, который сам загружает данные из БД при промахе.

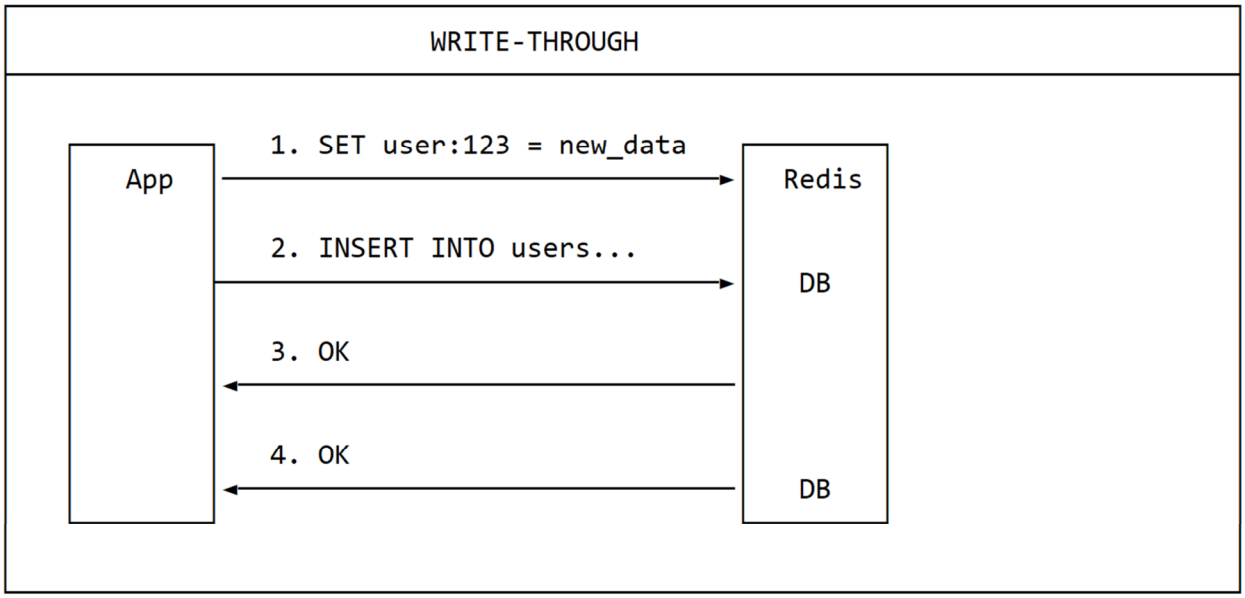
```
# Пример read-through
def get_user(user_id):
```

```
    # Кэш сам обрабатывает загрузку (если реализовано в библиотеке)
    return cache.get(f"user:{user_id}", loader=lambda: db.load_user(user_id))
```

Преимущества	Недостатки
Простое API для приложения	Кэш-библиотека должна поддерживать loader
Логика загрузки централизована	Сложнее отладка

1.2.3. Write-Through

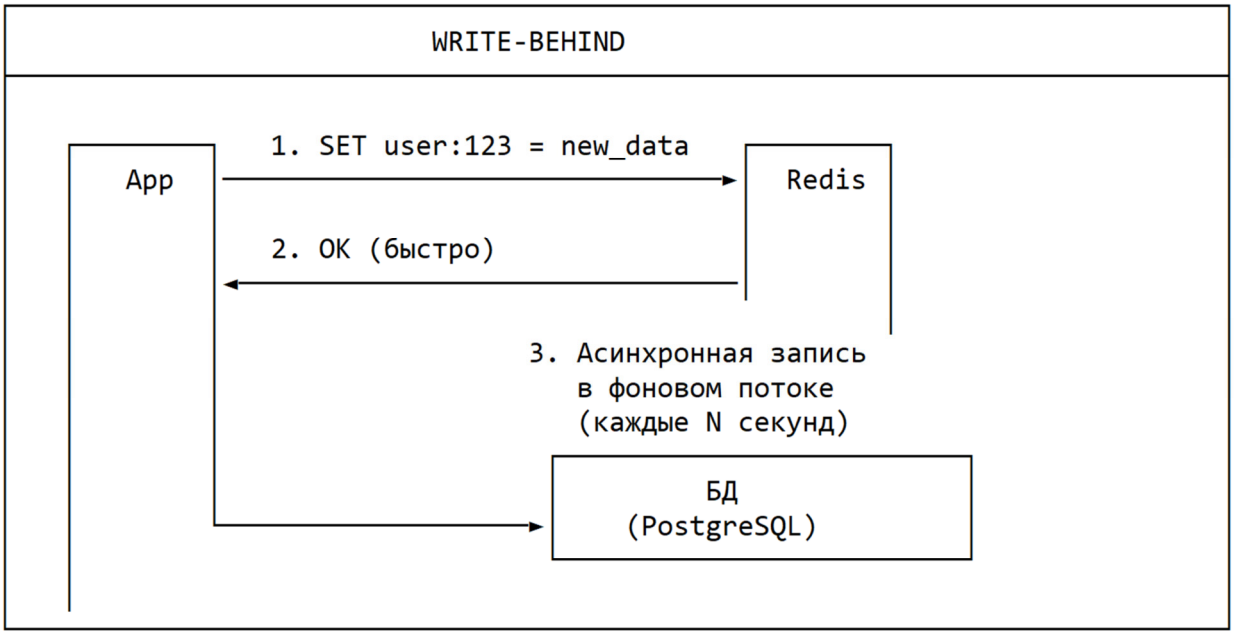
При записи данные одновременно записываются в БД и в кэш.



Преимущества	Недостатки
Кэш всегда актуален	Выше latency записи (две операции)
Нет проблем с инвалидацией	При отказе кэша запись может быть потеряна

1.2.4. Write-Behind (Write-Back)

Запись в БД происходит асинхронно, сначала данные попадают в кэш.



Преимущества	**Недостатки**
Очень низкая latency записи	Риск потери данных (при отказе кэша до записи в БД)
Пакетная запись в БД (batch)	Сложность реализации

1.2.5. Сравнение стратегий

Стратегия	**Latency чтения (hit)**	**Latency чтения (miss)**	**Latency записи**	**Консистентность**
Cache-Aside	Низкая	Средняя	Низкая	Eventual
Read-Through	Низкая	Средняя	—	Strong (read)
Write-Through	Низкая	—	Высокая	Strong

Стратегия	Latency чтения (hit)	Latency чтения (miss)	Latency записи	Консистентность
Write-Behind	Низкая	—	Очень низкая	Eventual

1.3. Политики вытеснения (Eviction Policies) в Redis

Redis поддерживает следующие политики вытеснения (eviction policies) при достижении лимита памяти maxmemory:

Политика	Описание
noeviction	Возвращает ошибку при записи новых данных (по умолчанию)
allkeys-lru	Удаляет наименее недавно использованные ключи (LRU) среди всех ключей
volatile-lru	Удаляет LRU среди ключей с TTL
allkeys-lfu	Удаляет наименее часто используемые ключи (LFU) среди всех ключей
volatile-lfu	Удаляет LFU среди ключей с TTL
allkeys-random	Удаляет случайный ключ
volatile-random	Удаляет случайный ключ среди ключей с TTL
volatile-ttl	Удаляет ключи с наименьшим оставшимся TTL

Рекомендации по выбору:

- **allkeys-lru** - общий случай, когда важны недавние данные
- **allkeys-lfu** - когда есть стабильно популярные ключи
- **volatile-ttl** - если данные естественным образом устаревают

1.4. TTL (Time To Live) и проактивная инвалидация

TTL - время жизни ключа в секундах/миллисекундах. Redis автоматически удаляет ключ по истечении TTL.

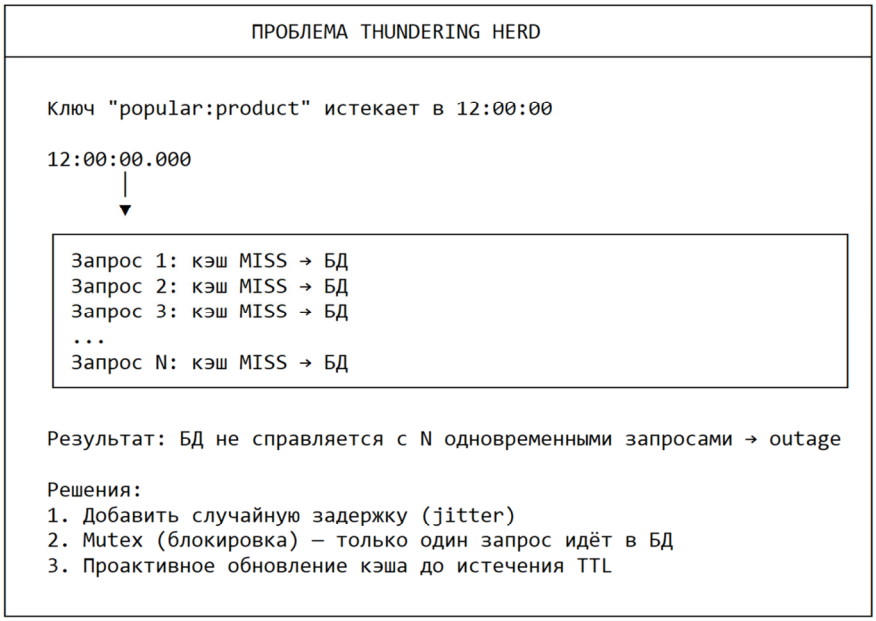
```
import redis
r = redis.Redis()

# Установка TTL при создании
r.setex("session:123", 3600, "user_data")  # 1 час

# Установка TTL для существующего ключа
r.expire("temp_data", 600)  # 10 минут

# Просмотр TTL
ttl = r.ttl("session:123")  # возвращает секунды до истечения
```

Типичные TTL по сценариям:

Сценарий	TTL
Сессии пользователей	1-24 часа
Каталог товаров	5-30 минут
Курсы валют	1-10 секунд
API ответы	1-60 секунд
Счётчики (rate limiting)	1-60 минут

1.5. Проблема «пробивания кэша» (Thundering Herd / Cache Stampede)

Проблема: При истечении TTL популярного ключа множество запросов одновременно обращаются к БД, вызывая пиковую нагрузку.



Решение с блокировкой (Mutex):

```python
import redis
import time

def get_with_mutex(key, ttl, loader):
    """Защита от thundering herd с использованием блокировки"""
    value = r.get(key)
    if value is not None:
        return value

    # Попытка захватить блокировку
    lock_key = f"{key}:lock"
    lock_acquired = r.setnx(lock_key, "locked")
    r.expire(lock_key, 5)  # Таймаут блокировки

    if lock_acquired:
        # Загружаем данные
        value = loader()
        r.setex(key, ttl, value)
        r.delete(lock_key)
        return value
    else:
        # Ждём, пока другой поток загрузит данные
        for _ in range(50):  # 5 секунд
            time.sleep(0.1)
            value = r.get(key)
            if value is not None:
                return value
        # Таймаут — загружаем сами
```

```
        return loader()
```

1.6. Redis как кэш: основные структуры данных

Тип	Команды	Сценарий использования
String	GET, SET, INCR, DECR	Простые значения, счётчики, сессии
Hash	HSET, HGET, HINCRBY	Объекты (user:id → поля)
List	LPUSH, RPUSH, LPOP, RPOP	Очереди, истории
Set	SADD, SREM, SISMEMBER, SINTER	Уникальные теги, отношения многие-ко-многим
Sorted Set	ZADD, ZRANGE, ZREVRANK	Лидерборды, приоритетные очереди
HyperLogLog	PFADD, PFCOUNT	Уникальные счётчики (приближённые)
Bitmap	SETBIT, GETBIT	Флаги признаков (check-in)

2. ПРИМЕРЫ ПРАКТИЧЕСКОГО ВЫПОЛНЕНИЯ

2.1. Cache-Aside с Redis и PostgreSQL

2.1.1. Структура проекта

text

redis-caching/

```
├── docker-compose.yml
├── app.py              # FastAPI приложение
├── requirements.txt
└── test_cache.py       # Тестирование производительности
```

2.1.2. Docker Compose (docker-compose.yml)

```yaml
version: '3.8'

services:
  postgres:
    image: postgres:15
    container_name: postgres
    environment:
      POSTGRES_USER: testuser
      POSTGRES_PASSWORD: testpass
      POSTGRES_DB: testdb
    ports:
      - "5432:5432"
    volumes:
      - postgres_data:/var/lib/postgresql/data

  redis:
    image: redis:7-alpine
    container_name: redis
    ports:
      - "6379:6379"
    command: redis-server --maxmemory 256mb --maxmemory-policy allkeys-lru
    volumes:
      - redis_data:/data

  app:
    build: .
    container_name: app
    ports:
      - "8000:8000"
    environment:
      POSTGRES_HOST: postgres
      REDIS_HOST: redis
    depends_on:
      - postgres
      - redis

volumes:
  postgres_data:
  redis_data:
```

2.1.3. Приложение с кэшированием (app.py)

```
import json
import time
import redis
import asyncpg
from fastapi import FastAPI, HTTPException
from pydantic import BaseModel
from contextlib import contextmanager
import asyncio

app = FastAPI(title="Caching Demo")

# Глобальные клиенты (singleton)
redis_client = None
db_pool = None

@app.on_event("startup")
async def startup():
    global redis_client, db_pool
    redis_client = redis.Redis(
        host="redis",
        port=6379,
        decode_responses=True,
        socket_connect_timeout=5
    )
    db_pool = await asyncpg.create_pool(
        host="postgres",
        user="testuser",
        password="testpass",
        database="testdb",
        min_size=5,
        max_size=20
    )
    await init_db()

@app.on_event("shutdown")
async def shutdown():
    await db_pool.close()
    redis_client.close()

async def init_db():
    async with db_pool.acquire() as conn:
        await conn.execute("""
            CREATE TABLE IF NOT EXISTS products (
                id SERIAL PRIMARY KEY,
                name TEXT NOT NULL,
                price DECIMAL(10,2) NOT NULL,
                stock INTEGER NOT NULL DEFAULT 0,
                updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
            )
```

```
        """)
        # Заполнение тестовыми данными
        result = await conn.fetchval("SELECT COUNT(*) FROM products")
        if result == 0:
            for i in range(1, 101):
                await conn.execute(
                    "INSERT INTO products (id, name, price, stock) VALUES ($1,
$2, $3, $4)",
                    i, f"Product_{i}", 100.0 * (i % 10 + 1), 100
                )
        print("Database initialized")

class ProductResponse(BaseModel):
    id: int
    name: str
    price: float
    stock: int
    source: str  # "cache" или "database"

async def get_product_from_db(product_id: int):
    """Загрузка продукта из БД"""
    async with db_pool.acquire() as conn:
        row = await conn.fetchrow(
            "SELECT id, name, price, stock FROM products WHERE id = $1",
            product_id
        )
        if not row:
            return None
        return dict(row)

# ========== Cache-Aside стратегия ==========
@app.get("/api/products/{product_id}", response_model=ProductResponse)
async def get_product_cache_aside(product_id: int):
    """Cache-Aside: сначала кэш, потом БД"""

    cache_key = f"product:{product_id}"

    # 1. Пытаемся получить из Redis
    cached = redis_client.get(cache_key)

    if cached:
        product = json.loads(cached)
        product["source"] = "cache"
        print(f"[CACHE HIT] Product {product_id}")
        return product

    # 2. Cache miss — загружаем из БД
    print(f"[CACHE MISS] Product {product_id} -> loading from DB")
    product = await get_product_from_db(product_id)

    if not product:
```

```
        raise HTTPException(status_code=404, detail="Product not found")

    # 3. Сохраняем в кэш с TTL 60 секунд
    redis_client.setex(cache_key, 60, json.dumps(product))

    product["source"] = "database"
    return product

# ========== Write-Through стратегия ==========
class ProductUpdateRequest(BaseModel):
    name: str = None
    price: float = None
    stock: int = None

@app.put("/api/products/{product_id}")
async def update_product_write_through(product_id: int, update:
ProductUpdateRequest):
    """Write-Through: обновляем БД и кэш одновременно"""

    cache_key = f"product:{product_id}"

    # 1. Обновляем БД
    async with db_pool.acquire() as conn:
        updates = []
        params = []
        if update.name:
            updates.append("name = $1")
            params.append(update.name)
        if update.price is not None:
            updates.append("price = $" + str(len(params) + 1))
            params.append(update.price)
        if update.stock is not None:
            updates.append("stock = $" + str(len(params) + 1))
            params.append(update.stock)

        if updates:
            params.append(product_id)
            query = f"""
                UPDATE products
                SET {', '.join(updates)}, updated_at = CURRENT_TIMESTAMP
                WHERE id = ${len(params)}
                RETURNING id, name, price, stock
            """
            row = await conn.fetchrow(query, *params)
            if not row:
                raise HTTPException(status_code=404, detail="Product not found")
            updated_product = dict(row)

    # 2. Обновляем кэш
    redis_client.setex(cache_key, 60, json.dumps(updated_product))
    print(f"[WRITE-THROUGH] Updated cache for product {product_id}")
```

```python
    return {"status": "updated", "product": updated_product}

# ========== Thundering Herd защита (Mutex) ==========
@app.get("/api/products/{product_id}/protected")
async def get_product_with_mutex(product_id: int):
    """Защита от thundering herd с использованием блокировки"""

    cache_key = f"product:{product_id}"
    lock_key = f"{cache_key}:lock"

    # Попытка получить из кэша
    cached = redis_client.get(cache_key)
    if cached:
        return {"source": "cache", "data": json.loads(cached)}

    # Попытка захватить блокировку
    lock_acquired = redis_client.setnx(lock_key, "locked")
    redis_client.expire(lock_key, 5)  # Таймаут 5 секунд

    if lock_acquired:
        # Загружаем из БД
        print(f"[MUTEX] Loading from DB (holder)")
        product = await get_product_from_db(product_id)
        if product:
            redis_client.setex(cache_key, 60, json.dumps(product))
        redis_client.delete(lock_key)
        return {"source": "database (lock holder)", "data": product}
    else:
        # Ждём результат
        print(f"[MUTEX] Waiting for cache...")
        for attempt in range(30):
            await asyncio.sleep(0.1)
            cached = redis_client.get(cache_key)
            if cached:
                return {"source": "cache (after wait)", "data":
json.loads(cached)}
        # Таймаут: загружаем сами
        product = await get_product_from_db(product_id)
        return {"source": "database (timeout fallback)", "data": product}

# ========== Статистика ==========
@app.get("/api/cache/stats")
async def get_cache_stats():
    """Получение статистики Redis"""
    info = redis_client.info("stats")
    return {
        "total_commands_processed": info.get("total_commands_processed"),
        "keyspace_hits": info.get("keyspace_hits"),
        "keyspace_misses": info.get("keyspace_misses"),
```

```
        "hit_rate": info.get("keyspace_hits") / (info.get("keyspace_hits") +
info.get("keyspace_misses") or 1)
    }

@app.delete("/api/cache/{product_id}")
async def invalidate_cache(product_id: int):
    """Принудительная инвалидация кэша"""
    cache_key = f"product:{product_id}"
    redis_client.delete(cache_key)
    return {"status": f"Cache for product {product_id} invalidated"}

if __name__ == "__main__":
    import uvicorn
    uvicorn.run(app, host="0.0.0.0", port=8000)
```

2.1.4. Тестирование производительности (test_cache.py)

```
import requests
import time
import threading
import statistics

BASE_URL = "http://localhost:8000"

def benchmark_cache(mode: str, product_id: int, num_requests: int = 1000):
    """Бенчмарк производительности кэша"""

    if mode == "cache-aside":
        url = f"{BASE_URL}/api/products/{product_id}"
    elif mode == "protected":
        url = f"{BASE_URL}/api/products/{product_id}/protected"
    else:
        return

    latencies = []

    # Инвалидация кэша перед тестом
    requests.delete(f"{BASE_URL}/api/cache/{product_id}")

    print(f"\n=== Бенчмарк: {mode} ({num_requests} запросов) ===")

    start_time = time.time()

    for i in range(num_requests):
        req_start = time.time()
        resp = requests.get(url)
        req_end = time.time()
        latencies.append((req_end - req_start) * 1000)  # ms

        if (i + 1) % 200 == 0:
```

```
            print(f"  Выполнено {i + 1}/{num_requests} запросов")

    end_time = time.time()

    print(f"\nРезультаты:")
    print(f"  Общее время: {end_time - start_time:.2f} сек")
    print(f"  RPS: {num_requests / (end_time - start_time):.2f}")
    print(f"  Средняя latency: {statistics.mean(latencies):.2f} ms")
    print(f"  Медианная latency: {statistics.median(latencies):.2f} ms")
    print(f"  p95 latency: {sorted(latencies)[int(len(latencies)*0.95)]:.2f} ms")
    print(f"  p99 latency: {sorted(latencies)[int(len(latencies)*0.99)]:.2f} ms")

def test_thundering_herd(num_workers: int = 10):
    """Тестирование защиты от thundering herd"""

    product_id = 1
    url = f"{BASE_URL}/api/products/{product_id}/protected"

    # Инвалидация кэша
    requests.delete(f"{BASE_URL}/api/cache/{product_id}")

    print(f"\n=== Thundering Herd Test ({num_workers} одновременных запросов)
===")

    def make_request():
        resp = requests.get(url)
        return resp.json()

    start_time = time.time()
    threads = []
    results = []

    for _ in range(num_workers):
        t = threading.Thread(target=lambda: results.append(make_request()))
        threads.append(t)
        t.start()

    for t in threads:
        t.join()

    end_time = time.time()

    print(f"Время выполнения: {end_time - start_time:.2f} сек")
    print(f"Источники ответов: {[r['source'] for r in results]}")

    # Проверка: только один запрос должен быть из БД
    db_sources = [r for r in results if "database" in r['source']]
    print(f"Запросы к БД: {len(db_sources)}")

if __name__ == "__main__":
    print("=== Тестирование кэширования Redis ===")
```

```
# Теплый запуск
requests.get(f"{BASE_URL}/api/products/1")
time.sleep(1)

# 1. Cache-Aside benchmark
benchmark_cache("cache-aside", 1, 500)

# 2. Повторный запуск (кэш прогрет)
benchmark_cache("cache-aside", 1, 2000)

# 3. Защита от thundering herd
test_thundering_herd(20)

# 4. Статистика кэша
stats = requests.get(f"{BASE_URL}/api/cache/stats").json()
print(f"\n=== Статистика Redis ===")
print(f"Hit rate: {stats['hit_rate']:.2%}")
print(f"Hits: {stats['keyspace_hits']}")
print(f"Misses: {stats['keyspace_misses']}")
```

МЕТОДИЧЕСКИЕ УКАЗАНИЯ ДЛЯ ВЫПОЛНЕНИЯ ПРАКТИЧЕСКОЙ ЧАСТИ РАБОТЫ

Задачи работы:

1. **Развёртывание Redis:**
   - Запустить Redis (Docker)
   - Настроить политику вытеснения (maxmemory-policy allkeys-lru)
   - Подключиться через redis-cli
2. **Реализация Cache-Aside:**
   - Настроить соединение с Redis и PostgreSQL
   - Реализовать логику: GET → кэш → БД → сохранить в кэш
   - Добавить TTL (30-60 секунд)
3. **Реализация Write-Through:**
   - Добавить эндпоинт обновления
   - Обновлять БД и кэш одновременно
   - Продемонстрировать консистентость
4. **Эксперимент 1: Производительность кэша:**
   - Измерить latency с кэшем (600+ RPS)

- Измерить latency без кэша (только БД)
- Сравнить hit rate

5. **Эксперимент 2: Политики вытеснения:**
   - Заполнить Redis данными до предела
   - Наблюдать вытеснение (LRU)
6. **Эксперимент 3: Защита от Thundering Herd:**
   - Инвалидировать популярный ключ
   - Сделать 20+ одновременных запросов
   - Пронаблюдать, что БД вызвана только один раз

3. ВАРИАНТЫ ЗАДАНИЙ

3.1. Базовый уровень (15 вариантов)

Требования: Cache-Aside + Write-Through

№	Тип данных	TTL	Политика вытеснения
1	Продукты	60 сек	allkeys-lru
2	Пользователи	300 сек	allkeys-lru
3	Сессии	3600 сек	volatile-lru
4	Счётчики просмотров	60 сек	allkeys-lfu
5	Курсы валют	10 сек	allkeys-lru
6	Новости	120 сек	allkeys-lru
7	Комментарии	30 сек	volatile-lru
8	Кэш API ответов	5 сек	allkeys-lru

№	Тип данных	TTL	Политика вытеснения
9	HTML фрагменты	300 сек	allkeys-lru
10	Словарь (key-value)	600 сек	allkeys-lfu
11	Рейтинг товаров (sorted set)	60 сек	volatile-lru
12	Список последних действий (list)	300 сек	allkeys-lru
13	Теги товаров (set)	120 сек	volatile-lfu
14	Лидерборд (sorted set)	10 сек	allkeys-lru
15	JSON документация	3600 сек	volatile-ttl

3.2. Средний уровень (15 вариантов)

Требования: как базовый + защита от Thundering Herd (Mutex)

№	Дополнительная функциональность	Тип защиты	Алгоритм вытеснения
1	Cache stampede (mutex)	Redis SETNX	allkeys-lru
2	Cache stampede (jitter 0-5s)	Случайная задержка	allkeys-lfu

№	Дополнительная функциональность	Тип защиты	Алгоритм вытеснения
3	Проактивное обновление	TTK (75%) + refresh	volatile-lru
4	Асинхронное обновление	Background task	allkeys-lru
5	Градуальное обновление	Постепенное обновление (ttl / 2)	volatile-ttl
6	Множественная блокировка	Redlock (2+ инстанса)	allkeys-lru
7	Добавить счётчик miss + alert	Prometheus метрики	volatile-lru
8	Bloom filter для miss	Фильтр отсутствующих ключей	allkeys-lfu
9	Если БД занята → дефолтные значения	Graceful degradation	allkeys-lru
10	Write-Behind + batch flush	Каждые 5 секунд или 100 записей	allkeys-lru
11	Write-Behind + Redis Stream	Буферизация в Redis Stream	volatile-lru
12	Self-healing кэша	При ошибке БД → возврат из кэша	allkeys-lru
1	Partition tolerant (c	Redis Sentinel (3	allkeys-lru

№	Дополнительная функциональность	Тип защиты	Алгоритм вытеснения
3	репликами)	ноды)	
14	Read-Through + loader	Библиотека загрузки данных	volatile-lfu
15	Batch cache (mget + pipeline)	Загрузка нескольких ключей	allkeys-lru

3.3. Продвинутый уровень (15 вариантов)

Требования: средний уровень + Redis Cluster + Lua-скрипты для атомарных операций

№	Кэшируемая сущность	Сложный сценарий	Требования
1	Каталог (лейзи-загрузка)	Иерархия (категория → товары)	Lua script: обновление дерева
2	Сессии с регионом	Геораспределение (multi region)	Redis Sentinel + geo
3	Rate limiter (токен-бакет)	Распределённый rate limit (счётчик)	Lua + множественные ключи
4	Инвентарь в реальном времени	Атомарное уменьшение stock (CAS)	Lua: get → decr → if <0 rollback

№	Кэшируемая сущность	Сложный сценарий	Требования
5	Leaderboard (игра)	Обновление рейтинга + отсечение 1000	Lua: ZADD → ZREMRANGEBYRANK
6	Очередь задач (Worker)	Приоритетная очередь с брокером	LPUSH + BRPOP
7	Поиск по тегам (множественный)	Пересечения множеств (SINTER) для фильтра	Set + Lua
8	Глобальный локатор (distributed lock)	Redlock (3 мастера)	Реализация блокировки
9	Кэш с версионированием	Кэш конкретной версии (Content-MD5)	Etag + версия в ключе
10	Асинхронная запись через Stream	Stream groups + retry	Redis Streams + отказоуст.
11	Временная инвалидация очереди	Задержка публикации (TTL)	TTL на сообщении
12	Bloom-filter + кэш	Контроль атакующих	Bloom filter

№	Кэшируемая сущность	Сложный сценарий	Требования
		запросов (нет в БД)	
13	HyperLogLog + кэш	Уникальные посетители (приближённо)	HyperLogLog
14	Кеш впереди БД (read-through + write-through)	Сервис слой + 2-х уровневое кэширование	L1 (локальный) + L2 (Redis)
15	Кэш на грани TTL + ожидание (preloading)	Пересчёт перед истечением (TTL 75%)	Cron + Distributed lock

4. КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Что такое кэширование и почему оно используется в высоконагруженных системах?
2. Какие стратегии кэширования вы знаете? В чём их различие?
3. Что такое Cache-Aside? Приведите пример.
4. В каких случаях используют Write-Through, а в каких - Write-Behind?
5. Что такое TTL (Time To Live) в Redis?
6. Какие политики вытеснения данных поддерживает Redis?
7. Какую политику вы выберете для кэша новостей и почему?
8. Что такое «пробивание кэша» (thundering herd / cache stampede)?
9. Как решить проблему thundering herd с помощью блокировки (mutex)?
10. В чём разница между LFU и LRU?

11. Как измерить эффективность кэша (hit rate, miss rate)?
12. Что такое Redis и почему он популярен для кэширования?
13. Какие структуры данных Redis подходят для лидерборда?
14. Как инвалидировать кэш (cache invalidation)?
15. Какие компромиссы при использовании кэша (consistency vs performance)?

5. ТРЕБОВАНИЯ К ОТЧЕТУ

1. **Титульный лист**
2. **Цель работы**
3. **Теоретическая часть:** стратегии кэширования, Redis, политики вытеснения, TTL, thundering herd
4. **Практическая часть:**
   - Листинг docker-compose.yml
   - Листинг app.py (основные эндпоинты)
   - Листинг бенчмарка
5. **Результаты экспериментов:**
   - График/таблица: latency с кэшем vs без кэша
   - Hit rate Redis
   - Результат теста thundering herd (один запрос в БД)
6. **Анализ:** влияние TTL, выбор политики, эффективность кэша
7. **Вывод**
8. **Ответы на контрольные вопросы**

6. КРИТЕРИИ ОЦЕНИВАНИЯ

Критерий	**Базовый (10)**	**Средний (14)**	**Продвинутый (18)**
Redis + PostgreSQL	2	1	1

Критерий	Базовый (10)	Средний (14)	Продвинутый (18)
развёрнуты			
Cache-Aside работает	3	2	1
Write-Through работает	2	2	1
Thundering Herd защита	—	3	2
Политики вытеснения (maxmemory)	—	2	2
Lua-скрипты / Redlock	—	—	3
Бенчмарк (latency + hit rate)	3	2	2
Качество отчёта	—	2	4
Максимум	**10**	**14**	**18**

Шкала перевода в оценку:

- **5 (отлично):** 90%+
- **4 (хорошо):** 75–89%
- **3 (удовлетворительно):** 60–74%
- **2 (неудовлетворительно):** <60%

7. ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ

Программа	Назначение	Доступность в РФ

Программа	Назначение	Доступность в РФ
Docker + Compose	Контейнеризация	+
Redis (официальный образ)	Кэш	+
PostgreSQL	Основная БД	+
Python + redis-py, asyncpg	Клиенты	+
curl / Postman	Тестирование	+

Лабораторная работа №11

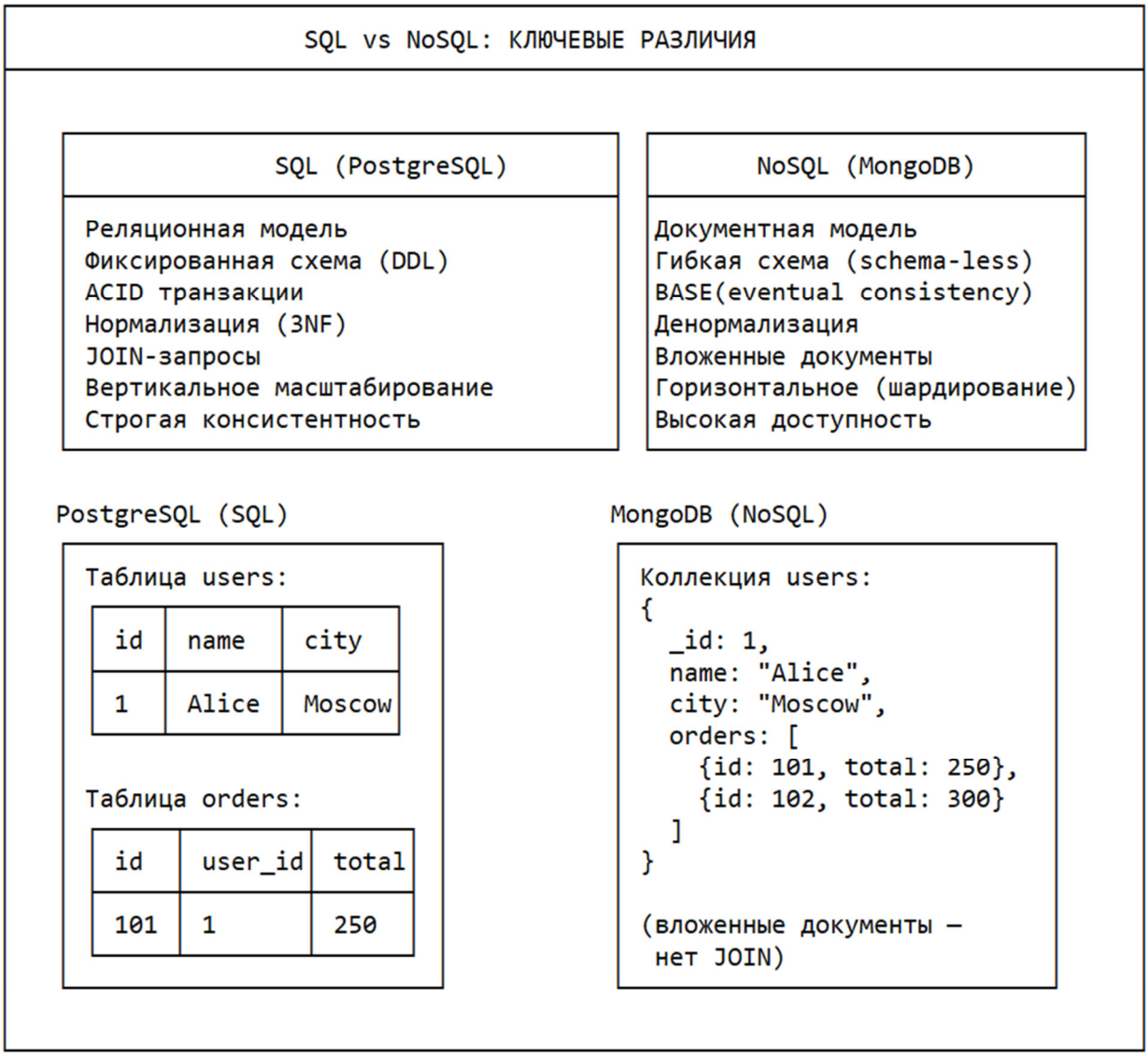
Сравнительный анализ SQL и NoSQL баз данных

Цель работы: Провести практическое сравнение реляционной (SQL) и документо-ориентированной (NoSQL) баз данных по критериям производительности, гибкости схемы, масштабируемости и сложности запросов на примере одного и того же приложения, реализованного на PostgreSQL и MongoDB, с последующим анализом trade-offs и выработкой рекомендаций по выбору СУБД для различных классов задач.

Формируемые компетенции: ПК-5

1. ТЕОРЕТИЧЕСКОЕ ОБОСНОВАНИЕ

1.1. SQL vs NoSQL: фундаментальные различия



1.2. Модели данных: нормализация vs денормализация

```sql
-- Отдельные таблицы с отношениями
CREATE TABLE users (
    id SERIAL PRIMARY KEY,
    name VARCHAR(100),
    email VARCHAR(100) UNIQUE
);

CREATE TABLE orders (
    id SERIAL PRIMARY KEY,
    user_id INTEGER REFERENCES users(id),
    total DECIMAL(10,2),
    status VARCHAR(20)
);

CREATE TABLE order_items (
    id SERIAL PRIMARY KEY,
    order_id INTEGER REFERENCES orders(id),
    product_id INTEGER,
```

```
    quantity INTEGER,
    price DECIMAL(10,2)
);

-- Запрос с JOIN
SELECT u.name, o.total, oi.product_id, oi.quantity
FROM users u
JOIN orders o ON u.id = o.user_id
JOIN order_items oi ON o.id = oi.order_id
WHERE u.id = 123;
```

1.3. Сравнение по характеристикам

Характеристика	PostgreSQL (SQL)	MongoDB (NoSQL)
Схема данных	Фиксированная, требуется миграция	Гибкая, schema-less
Транзакции	ACID (полная поддержка)	ACID на уровне документа (4.0+ multi-doc)
JOIN	Полноценные JOIN	$lookup (менее эффективен)
Агрегация	SQL (GROUP BY, оконные функции)	Aggregation Pipeline
Индексы	B-tree, GiST, GIN, BRIN	B-tree, гео, текстовые, TTL
Горизонтальное	Шардирование (Citus,	Нативное

Характеристика	PostgreSQL (SQL)	MongoDB (NoSQL)
масштабирование	pg_shard)	шардирование (по умолчанию)
Репликация	Streaming replication	Replica set
Типы данных	Богатая типизация (JSONB, массивы, гео)	BSON (бинарный JSON)
Полнотекстовый поиск	ts_vector, ts_query	Текстовые индексы
Хранимые процедуры	PL/pgSQL, PL/Python, PL/Perl	JavaScript (операции map/reduce)

1.4. Сценарии использования: когда что выбирать

Сценарий	Рекомендуемая БД	Обоснование
Финтех, платёжные системы	SQL (PostgreSQL)	ACID-транзакции, строгая консистентность
Каталог товаров (read-heavy)	MongoDB	Гибкая схема, высокая производительность

Сценарий	Рекомендуемая БД	Обоснование
		чтения
Логирование, метрики	MongoDB (time-series)	Высокая скорость записи, TTL-индексы
CRM с отчётами	SQL	Сложные аналитические запросы, JOIN
Сессии пользователей	MongoDB	Высокая доступность, шардирование
Блоги/комментарии	MongoDB	Денормализация (комментарии в посте)
Геоданные (maps)	PostgreSQL (PostGIS) или MongoDB	Оба поддерживают геоиндексы
Real-time аналитика	ClickHouse / TimescaleDB	Специализированные time-series БД
Графовые данные	Neo4j	Специализированная графовая БД
Кэш/сессии	Redis	In-memory, сверхнизкая latency

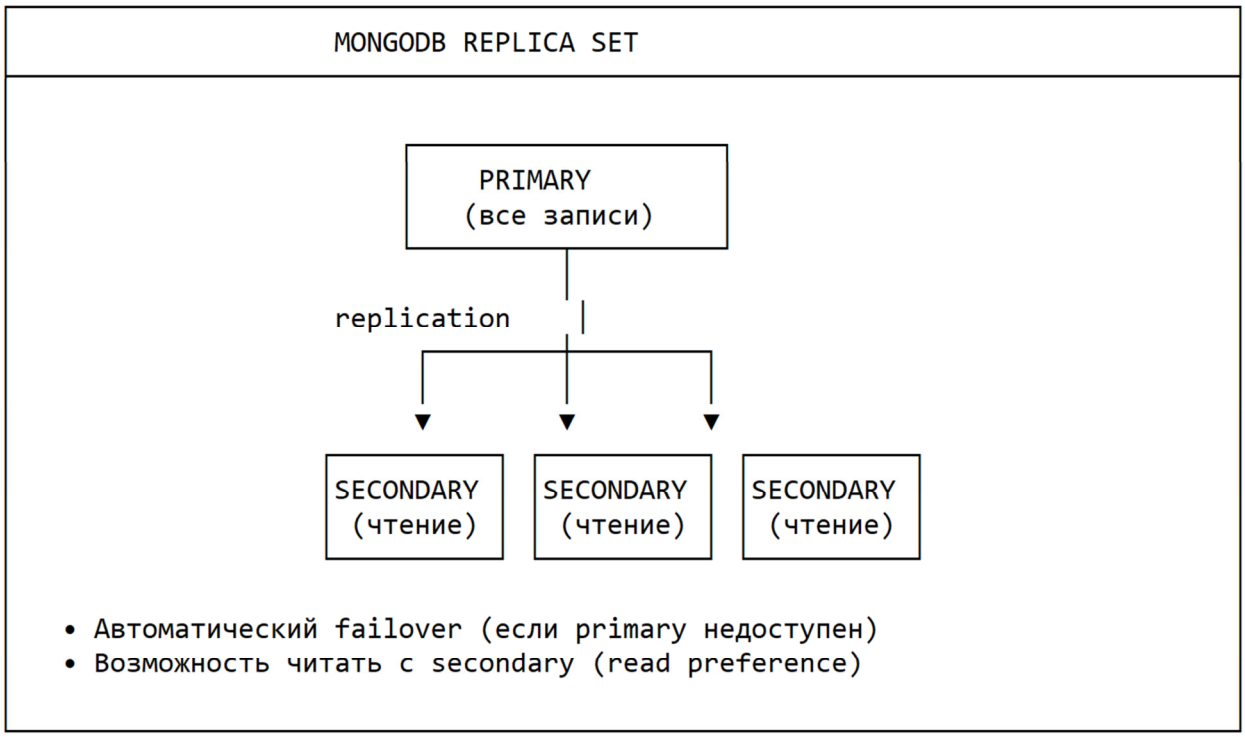
1.5. Производительность: чтение vs запись

ПРОИЗВОДИТЕЛЬНОСТЬ: SQL vs NoSQL

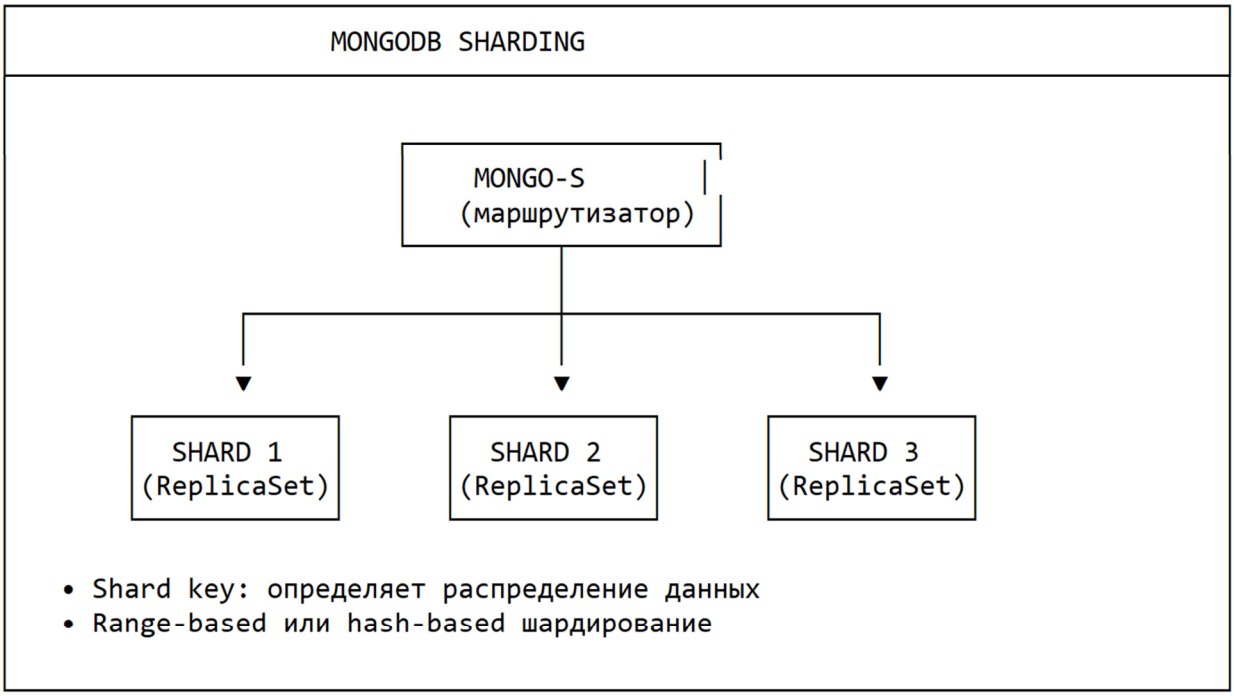
Операция	PostgreSQL	MongoDB
Простое чтение по ID (point select)	(~8-10k RPS)	(~12-15k RPS)
Сложный SELECT с JOIN (3 таблицы)	(~3-5k RPS)	(~1-2k RPS через $lookup)
Вставка (INSERT)	(~5-8k TPS)	(~15-20k TPS)
Вставка с транзакцией	(~2-4k TPS)	(~8-10k TPS)
Агрегация (SUM, AVG)	(~5-7k RPS)	(~4-6k RPS)

1.6. Особенности MongoDB для high-load

Replica Set (отказоустойчивость):



Шардирование в MongoDB:



2. ПРИМЕРЫ ПРАКТИЧЕСКОГО ВЫПОЛНЕНИЯ

2.1. Сравнение производительности: PostgreSQL vs MongoDB

2.1.1. Структура проекта

```
sql-nosql-comparison/
├── docker-compose.yml
├── postgres/
│   └── init.sql
├── mongo/
│   └── init.js
├── benchmark.py          # Сравнительный бенчмарк
├── app_postgres.py       # FastAPI + PostgreSQL
├── app_mongo.py          # FastAPI + MongoDB
└── requirements.txt
```

2.1.2. Docker Compose (docker-compose.yml)

```yaml
version: '3.8'

services:
```

```yaml
  postgres:
    image: postgres:15
    container_name: postgres
    environment:
      POSTGRES_USER: testuser
      POSTGRES_PASSWORD: testpass
      POSTGRES_DB: testdb
    ports:
      - "5432:5432"
    volumes:
      - ./postgres/init.sql:/docker-entrypoint-initdb.d/init.sql
      - postgres_data:/var/lib/postgresql/data

  mongo:
    image: mongo:6.0
    container_name: mongo
    ports:
      - "27017:27017"
    volumes:
      - ./mongo/init.js:/docker-entrypoint-initdb.d/init.js
      - mongo_data:/data/db

volumes:
  postgres_data:
  mongo_data:
```

2.1.3. Инициализация PostgreSQL (postgres/init.sql)

```sql
-- Схема для интернет-магазина
CREATE TABLE users (
    id SERIAL PRIMARY KEY,
    name VARCHAR(100),
    email VARCHAR(100) UNIQUE,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

CREATE TABLE categories (
    id SERIAL PRIMARY KEY,
    name VARCHAR(100),
    parent_id INTEGER REFERENCES categories(id)
);

CREATE TABLE products (
    id SERIAL PRIMARY KEY,
    name VARCHAR(200),
    price DECIMAL(10,2),
    category_id INTEGER REFERENCES categories(id),
    stock INTEGER DEFAULT 0
);
```

```sql
CREATE TABLE orders (
    id SERIAL PRIMARY KEY,
    user_id INTEGER REFERENCES users(id),
    total DECIMAL(10,2),
    status VARCHAR(20),
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

CREATE TABLE order_items (
    id SERIAL PRIMARY KEY,
    order_id INTEGER REFERENCES orders(id),
    product_id INTEGER REFERENCES products(id),
    quantity INTEGER,
    price DECIMAL(10,2)
);

-- Индексы для производительности
CREATE INDEX idx_orders_user_id ON orders(user_id);
CREATE INDEX idx_order_items_order_id ON order_items(order_id);
CREATE INDEX idx_products_category_id ON products(category_id);

-- Тестовые данные (1000 пользователей, 100 категорий, 10000 товаров)
INSERT INTO users (name, email)
SELECT
    'User_' || i,
    'user' || i || '@example.com'
FROM generate_series(1, 1000) AS i;

INSERT INTO categories (name, parent_id)
SELECT
    'Category_' || i,
    CASE WHEN i % 10 = 0 THEN i/10 ELSE NULL END
FROM generate_series(1, 100) AS i;

INSERT INTO products (name, price, category_id, stock)
SELECT
    'Product_' || i,
    (random() * 1000)::DECIMAL(10,2),
    (i % 100) + 1,
    (random() * 100)::INT
FROM generate_series(1, 10000) AS i;
```

2.1.4. Инициализация MongoDB (mongo/init.js)

```javascript
// Создание базы данных и коллекций
db = db.getSiblingDB('testdb');

// Коллекция users (денормализованная — заказы вложены)
db.createCollection('users');
```

```
// Коллекция продуктов
db.createCollection('products');

// Индексы
db.users.createIndex({ email: 1 });
db.products.createIndex({ category_id: 1 });
db.products.createIndex({ price: 1 });

// Тестовые данные (1000 пользователей, каждый с 3-5 заказами)
for (let i = 1; i <= 1000; i++) {
    const orders = [];
    const numOrders = Math.floor(Math.random() * 5) + 1;

    for (let j = 1; j <= numOrders; j++) {
        const items = [];
        const numItems = Math.floor(Math.random() * 3) + 1;

        for (let k = 1; k <= numItems; k++) {
            items.push({
                product_id: Math.floor(Math.random() * 10000) + 1,
                quantity: Math.floor(Math.random() * 5) + 1,
                price: (Math.random() * 1000).toFixed(2)
            });
        }

        orders.push({
            id: i * 1000 + j,
            total: items.reduce((sum, item) => sum + item.quantity *
item.price, 0),
            status: ['pending', 'completed',
'shipped'][Math.floor(Math.random() * 3)],
            items: items,
            created_at: new Date()
        });
    }

    db.users.insertOne({
        _id: i,
        name: 'User_' + i,
        email: 'user' + i + '@example.com',
        orders: orders,
        created_at: new Date()
    });
}

// Продукты
for (let i = 1; i <= 10000; i++) {
    db.products.insertOne({
        _id: i,
        name: 'Product_' + i,
```

```
        price: (Math.random() * 1000).toFixed(2),
        category_id: (i % 100) + 1,
        stock: Math.floor(Math.random() * 100)
    });
}

print('MongoDB initialized with 1000 users, 10000 products');
```

2.1.5. Бенчмарк (benchmark.py)

```
import time
import psycopg2
import pymongo
import statistics
import asyncio
from typing import Dict, List

class DatabaseBenchmark:
    def __init__(self):
        # PostgreSQL
        self.pg_conn = psycopg2.connect(
            host="localhost",
            port=5432,
            user="testuser",
            password="testpass",
            database="testdb"
        )

        # MongoDB
        self.mongo_client = pymongo.MongoClient("mongodb://localhost:27017/")
        self.mongo_db = self.mongo_client["testdb"]

    def benchmark_pg_read_point(self, num_queries: int = 1000) -> Dict:
        """Point-запрос (чтение пользователя по ID)"""
        latencies = []

        cursor = self.pg_conn.cursor()
        for i in range(1, num_queries + 1):
            user_id = (i % 1000) + 1
            start = time.perf_counter()
            cursor.execute("SELECT id, name, email FROM users WHERE id = %s",
(user_id,))
            cursor.fetchone()
            latencies.append((time.perf_counter() - start) * 1000)

        return {
            "avg_latency_ms": statistics.mean(latencies),
            "p95_latency_ms": sorted(latencies)[int(len(latencies) * 0.95)],
```

```
            "p99_latency_ms": sorted(latencies)[int(len(latencies) * 0.99)],
            "rps": num_queries / (sum(latencies) / 1000)
        }

    def benchmark_mongo_read_point(self, num_queries: int = 1000) -> Dict:
        """Point-запрос в MongoDB"""
        latencies = []
        collection = self.mongo_db["users"]

        for i in range(1, num_queries + 1):
            user_id = (i % 1000) + 1
            start = time.perf_counter()
            collection.find_one({"_id": user_id})
            latencies.append((time.perf_counter() - start) * 1000)

        return {
            "avg_latency_ms": statistics.mean(latencies),
            "p95_latency_ms": sorted(latencies)[int(len(latencies) * 0.95)],
            "p99_latency_ms": sorted(latencies)[int(len(latencies) * 0.99)],
            "rps": num_queries / (sum(latencies) / 1000)
        }

    def benchmark_pg_read_join(self, num_queries: int = 1000) -> Dict:
        """Сложный запрос с JOIN (пользователь + его заказы + товары)"""
        latencies = []
        cursor = self.pg_conn.cursor()

        query = """
            SELECT u.id, u.name, o.id as order_id, o.total,
                   oi.product_id, oi.quantity, p.name as product_name
            FROM users u
            JOIN orders o ON u.id = o.user_id
            JOIN order_items oi ON o.id = oi.order_id
            JOIN products p ON oi.product_id = p.id
            WHERE u.id = %s
            LIMIT 100
        """

        for i in range(1, num_queries + 1):
            user_id = (i % 1000) + 1
            start = time.perf_counter()
            cursor.execute(query, (user_id,))
            cursor.fetchall()
            latencies.append((time.perf_counter() - start) * 1000)

        return {
            "avg_latency_ms": statistics.mean(latencies),
            "p95_latency_ms": sorted(latencies)[int(len(latencies) * 0.95)],
            "p99_latency_ms": sorted(latencies)[int(len(latencies) * 0.99)],
            "rps": num_queries / (sum(latencies) / 1000)
        }
```

```python
    def benchmark_mongo_read_aggregate(self, num_queries: int = 1000) -> Dict:
        """Агрегация в MongoDB (замена JOIN)"""
        latencies = []
        collection = self.mongo_db["users"]

        pipeline = [
            {"$match": {"_id": 1}},
            {"$unwind": "$orders"},
            {"$unwind": "$orders.items"},
            {"$lookup": {
                "from": "products",
                "localField": "orders.items.product_id",
                "foreignField": "_id",
                "as": "product_info"
            }}
        ]

        for i in range(1, num_queries + 1):
            pipeline[0]["$match"]["_id"] = (i % 1000) + 1
            start = time.perf_counter()
            list(collection.aggregate(pipeline))
            latencies.append((time.perf_counter() - start) * 1000)

        return {
            "avg_latency_ms": statistics.mean(latencies),
            "p95_latency_ms": sorted(latencies)[int(len(latencies) * 0.95)],
            "p99_latency_ms": sorted(latencies)[int(len(latencies) * 0.99)],
            "rps": num_queries / (sum(latencies) / 1000)
        }

    def benchmark_pg_write(self, num_writes: int = 1000) -> Dict:
        """Вставка данных"""
        latencies = []
        cursor = self.pg_conn.cursor()

        for i in range(num_writes):
            start = time.perf_counter()
            cursor.execute(
                "INSERT INTO logs (message, created_at) VALUES (%s, NOW())",
                (f"Log message {i}",)
            )
            self.pg_conn.commit()
            latencies.append((time.perf_counter() - start) * 1000)

        return {
            "avg_latency_ms": statistics.mean(latencies),
            "tps": num_writes / (sum(latencies) / 1000)
        }

    def benchmark_mongo_write(self, num_writes: int = 1000) -> Dict:
```

```
        """Вставка в MongoDB"""
        latencies = []
        collection = self.mongo_db["logs"]

        for i in range(num_writes):
            start = time.perf_counter()
            collection.insert_one({"message": f"Log message {i}", "timestamp":
time.time()})
            latencies.append((time.perf_counter() - start) * 1000)

        return {
            "avg_latency_ms": statistics.mean(latencies),
            "tps": num_writes / (sum(latencies) / 1000)
        }

    def run_full_benchmark(self):
        """Запуск всех тестов"""
        print("\n" + "="*60)
        print("СРАВНИТЕЛЬНЫЙ БЕНЧМАРК: PostgreSQL vs MongoDB")
        print("="*60)

        # 1. Point-запросы
        print("\n📌 1. Point-запрос (чтение по ID):")
        pg_point = self.benchmark_pg_read_point(1000)
        mongo_point = self.benchmark_mongo_read_point(1000)
        print(f"   PostgreSQL: avg={pg_point['avg_latency_ms']:.2f}ms,
RPS={pg_point['rps']:.0f}")
        print(f"   MongoDB:    avg={mongo_point['avg_latency_ms']:.2f}ms,
RPS={mongo_point['rps']:.0f}")

        # 2. Сложные запросы (JOIN vs агрегация)
        print("\n🔗 2. Сложный запрос (JOIN/агрегация):")
        pg_join = self.benchmark_pg_read_join(500)
        mongo_agg = self.benchmark_mongo_read_aggregate(500)
        print(f"   PostgreSQL: avg={pg_join['avg_latency_ms']:.2f}ms,
RPS={pg_join['rps']:.0f}")
        print(f"   MongoDB:    avg={mongo_agg['avg_latency_ms']:.2f}ms,
RPS={mongo_agg['rps']:.0f}")

        # 3. Запись
        print("\n✏️ 3. Вставка записей:")
        pg_write = self.benchmark_pg_write(500)
        mongo_write = self.benchmark_mongo_write(500)
        print(f"   PostgreSQL: avg={pg_write['avg_latency_ms']:.2f}ms,
TPS={pg_write['tps']:.0f}")
        print(f"   MongoDB:    avg={mongo_write['avg_latency_ms']:.2f}ms,
TPS={mongo_write['tps']:.0f}")

        # Вывод рекомендаций
        print("\n" + "="*60)
        print("ВЫВОДЫ И РЕКОМЕНДАЦИИ:")
```

```
        if mongo_point['avg_latency_ms'] < pg_point['avg_latency_ms']:
            print("✅ MongoDB быстрее для point-запросов (чтение по ID)")
        else:
            print("✅ PostgreSQL быстрее для point-запросов")

        if pg_join['avg_latency_ms'] < mongo_agg['avg_latency_ms']:
            print("✅ PostgreSQL значительно быстрее для сложных запросов с
JOIN")
        else:
            print("✅ MongoDB быстрее для сложных запросов (денормализация)")

        if mongo_write['avg_latency_ms'] < pg_write['avg_latency_ms']:
            print("✅ MongoDB быстрее для операций записи")
        else:
            print("✅ PostgreSQL быстрее для операций записи")

if __name__ == "__main__":
    benchmark = DatabaseBenchmark()
    benchmark.run_full_benchmark()
```

МЕТОДИЧЕСКИЕ УКАЗАНИЯ ДЛЯ ВЫПОЛНЕНИЯ ПРАКТИЧЕСКОЙ ЧАСТИ РАБОТЫ

Задачи работы:

1. **Развёртывание PostgreSQL и MongoDB:**
   - Запустить оба контейнера через Docker Compose
   - Инициализировать схему и тестовые данные
2. **Эксперимент 1: Сравнение производительности:**
   - Замерить latency point-запросов (чтение по ID)
   - Замерить latency сложных запросов (JOIN vs aggregation)
   - Замерить TPS для операций записи
3. **Эксперимент 2: Анализ гибкости схемы:**
   - Добавить новое поле в обеих БД (например, phone)
   - В PostgreSQL — миграция (ALTER TABLE ADD COLUMN)
   - В MongoDB — просто добавление поля в документ
4. **Эксперимент 3: Поведение при масштабировании:**
   - Включить шардирование в MongoDB
   - Настроить репликацию в PostgreSQL

- Сравнить сложность и время настройки

5. **Оформление отчёта:** результаты, графики, выводы

3. ВАРИАНТЫ ЗАДАНИЙ

3.1. Базовый уровень (15 вариантов)

Требования: развернуть обе БД, провести бенчмарк чтения/записи

№	Сценарий нагрузки	Объём данных	Метрики
1	Чтение по ID (point)	10k записей	latency, RPS
2	Чтение по ID	100k записей	latency, RPS
3	Чтение по ID	1M записей	latency, RPS
4	Диапазонный запрос (WHERE price > X)	10k товаров	latency, RPS
5	Диапазонный запрос	100k товаров	latency, RPS
6	JOIN 2 таблиц	10k+ заказов	latency, RPS
7	JOIN 3 таблиц	10k+ заказов	latency, RPS
8	Агрегация (COUNT, SUM)	10k заказов	latency, RPS
9	Агрегация (AVG, GROUP BY)	100k заказов	latency, RPS
10	Вставка (INSERT)	5000 записей	avg latency, TPS
11	Вставка (bulk insert)	5000 записей	avg latency, TPS

№	Сценарий нагрузки	Объём данных	Метрики
12	Обновление (UPDATE)	5000 записей	avg latency, TPS
13	Удаление (DELETE)	5000 записей	avg latency, TPS
14	Смешанная нагрузка (50% read, 40% write, 10% update)	10k	throughput
15	Смешанная нагрузка с транзакциями	10k с транзакциями	throughput

3.2. Средний уровень (15 вариантов)

Требования: как базовый + миграция схемы + сравнение индексов

№	Дополнительная задача	Тип миграции	Индексы
1	Добавить поле phone	ALTER TABLE (SQL), просто поле (Mongo)	Индекс на phone
2	Добавить поле birth_date	С дефолтным значением (SQL)	B-tree индекс
3	Изменить тип поля (price INTEGER → DECIMAL)	Миграция с конвертацией	Перестроить индекс
4	Добавить внешний ключ (ON DELETE CASCADE)	SQL — FK, Mongo — вручную	Индекс на FK
5	Добавить составной индекс	idx_users_city_age (SQL)	Составной индекс

№	Дополнительная задача	Тип миграции	Индексы
6	Добавить текстовый поиск	tsvector (SQL) vs текстовый (Mongo)	GIN vs text index
7	Добавить гео-запросы (points within radius)	PostGIS (SQL)	2dsphere index
8	Добавить TTL (устаревание данных)	Выполнять вручную (SQL)	TTL index (Mongo)
9	Версионирование документов (schema version)	Добавить поле schema_version	Частичный индекс (SQL)
10	Партиционирование (range по дате)	Partition by RANGE (SQL)	Нативное (MongoDB)
11	Добавить триггер (при вставке → аудит)	PL/pgSQL trigger	Change stream (Mongo)
12	Оптимизация JOIN с помощью денормализации	Денормализованное поле	Индекс на поле
13	Материализованное представление	mat view (SQL)	—
14	Валидация данных (check constraint)	CHECK (SQL)	JSON schema (Mongo)
15	Аудит изменений (history table)	Триггер + history	Обновление через код

3.3. Продвинутый уровень (15 вариантов)

Требования: средний уровень + шардирование (MongoDB) + репликация (PostgreSQL)

№	Топология MongoDB	Топология PostgreSQL	Сценарий отказа
1	Sharded cluster (3 шарда)	Streaming replication (master + slave)	Отказ шарда
2	Sharded (hash на user_id)	Синхронная репликация	Отказ master
3	Sharded (range по дате)	Асинхронная репликация + Patroni	Partition network
4	Зональное шардирование (zone sharding)	Каскадная репликация	Split-brain
5	Replica set (3 узла) + read preference	Patroni + etcd	Failover время
6	Multi-region (Primary в US, secondary в EU)	Geo-replication (BDR)	Задержка cross-region
7	Смешанное шардирование (compound shard key)	Citus (distributed tables)	Перебалансировка

№	Топология MongoDB	Топология PostgreSQL	Сценарий отказа
8	Shard + replica set per shard	Patroni + HAProxy	Автоматический failover
9	Consistent hashing (shard key)	Sharding plugin (pg_shard)	Добавление шарда (resharding)
10	Холодное/горячее шардирование (архив)	Partition + read-only	Запрос к архиву
11	Использование load balancer (mongos) + retry	PgBouncer + DNS	Отказ балансировщика
12	Sharding + скрытый вторичный узел	PostgreSQL + repmgr	Ручной failover
13	Расширение: отказ shard + перестроение	Patroni + etcd (3 узла)	Восстановление после split-brain
14	ClickHouse / TimescaleDB (опционально)	Citus + columnar store	Аналитические запросы
15	Multi-model (Mongo + Redis кэш)	PostgreSQL + материализ. представления	Гибридное хранилище

4. КОНТРОЛЬНЫЕ ВОПРОСЫ

1. В чём ключевые различия между SQL (реляционными) и NoSQL (документными) базами данных?
2. Что такое нормализация/денормализация и как она влияет на производительность?
3. Почему в MongoDB нет JOIN и чем заменяются сложные связи?
4. Как транзакции работают в PostgreSQL и MongoDB (ACID vs eventual consistency)?
5. Какую БД выбрать для финансовой системы и почему?
6. Какую БД выбрать для каталога товаров с высокой нагрузкой чтения?
7. Что такое шардирование в MongoDB и как оно устроено?
8. Как репликация работает в PostgreSQL (master → slave)?
9. В каком сценарии NoSQL может быть быстрее SQL?
10. Что такое $lookup в MongoDB, в чём его ограничения по сравнению с JOIN?
11. Какое влияние на производительность оказывает количество индексов?
12. Как провести миграцию схемы в PostgreSQL без простоя?
13. Как в MongoDB добавить поле в существующий документ?
14. В чём отличие MongoDB от PostgreSQL при работе с JSON данными?
15. Какие компромиссы между консистентностью и доступностью в каждой системе?

5. ТРЕБОВАНИЯ К ОТЧЕТУ

1. **Титульный лист**
2. **Цель работы**
3. **Теоретическая часть:** SQL vs NoSQL, модели данных, CAP-теорема
4. **Практическая часть:**
   - Листинг docker-compose.yml

- Схемы данных (SQL и NoSQL)
- Скрипты для бенчмарка

5. **Результаты экспериментов:**
   - Таблица сравнения latency (point-запросы)
   - Таблица сравнения сложных запросов (JOIN/aggregation)
   - График TPS для вставок
6. **Анализ:** где какая БД выигрывает и почему
7. **Вывод:** рекомендации по выбору — когда SQL, когда NoSQL
8. **Ответы на контрольные вопросы**

6. КРИТЕРИИ ОЦЕНИВАНИЯ

Критерий	**Базовый (10)**	**Средний (14)**	**Продвинутый (18)**
PostgreSQL развёрнут и работает	2	1	1
MongoDB развёрнута и работает	2	1	1
Измерение point-запросов	2	2	1
Измерение JOIN/агрегации	2	2	2
Гибкость схемы (миграция)	—	2	1
Индексы и оптимизация	2	2	2
Шардирование/репликация	—	2	3
Качество отчёта	—	2	4
Максимум	**10**	**14**	**18**

Шкала перевода в оценку:

- **5 (отлично):** 90%+
- **4 (хорошо):** 75–89%
- **3 (удовлетворительно):** 60–74%
- **2 (неудовлетворительно):** <60%

7. ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ

Программа	Назначение	Доступность в РФ
Docker + Compose	Контейнеризация	+
PostgreSQL (официальный образ)	Реляционная БД	+
MongoDB (официальный образ)	Документная БД	+
Python + psycopg2, pymongo	Клиенты	+
pgAdmin (опционально)	Управление PostgreSQL	+
MongoDB Compass (опционально)	Управление MongoDB	+

Лабораторная работа №12

Реализация паттерна Saga для распределённых транзакций

Цель работы: Сформировать практические навыки реализации паттерна Saga для управления распределёнными транзакциями в микросервисной архитектуре без применения двухфазного коммита (2PC), включая два подхода: хореографию (choreography) на основе событий и

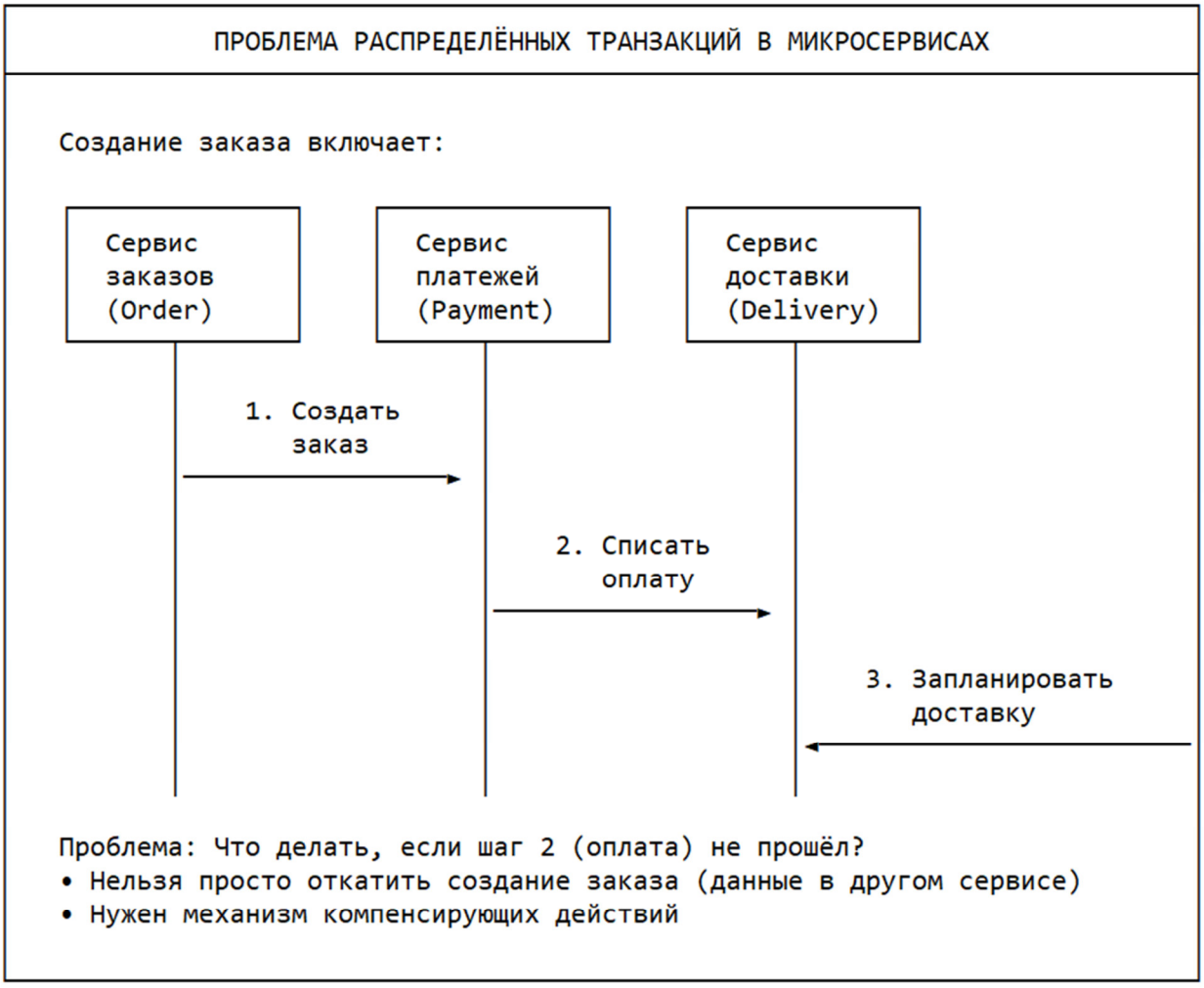
оркестрацию (orchestration) с центральным координатором, а также разработку компенсирующих транзакций для отката при сбоях.

Формируемые компетенции: ПК-5

1. ТЕОРЕТИЧЕСКОЕ ОБОСНОВАНИЕ

1.1. Проблема распределённых транзакций в микросервисах

В монолитной архитектуре ACID-транзакции обеспечивают атомарность, согласованность, изоляцию и долговечность операций с данными. В микросервисной архитектуре данные распределены по разным сервисам, и традиционные механизмы (2PC, XA) становятся неприменимыми из-за проблем с производительностью, масштабируемостью и отказоустойчивостью.



1.2. Паттерн Saga: определение и принципы

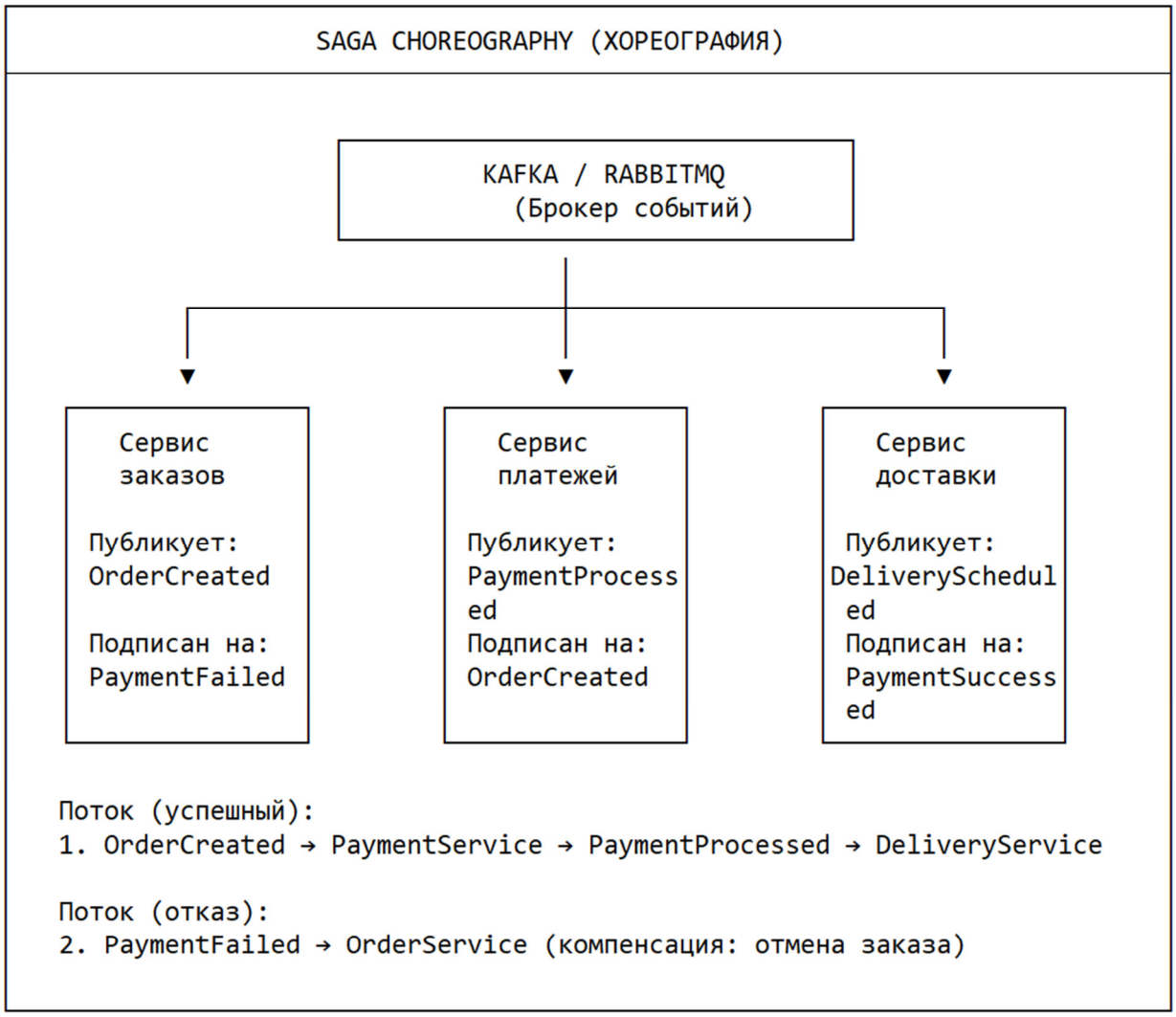
Saga - это паттерн для управления распределёнными транзакциями, который разбивает длинную транзакцию на последовательность локальных транзакций, каждая из которых может быть скомпенсирована при сбое.

Два подхода к реализации Saga:

Характеристика	**Хореография (Choreography)**	**Оркестрация (Orchestration)**
Координация	Децентрализованная (сервисы общаются через события)	Централизованная (оркестратор управляет шагами)
Брокер	Kafka / RabbitMQ	HTTP / gRPC / очередь
Сложность	Низкая (для 3-5 сервисов)	Средняя (прямолинейная логика)
Циклические зависимости	Риск есть	Отсутствуют
Отслеживание состояния	Сложное (размазано по логам)	Простое (в оркестраторе)
Масштабируемость	Высокая	Ограничена оркестратором
Когда выбирать	Много сервисов, слабые связи	Сложная бизнес-логика, нужен контроль

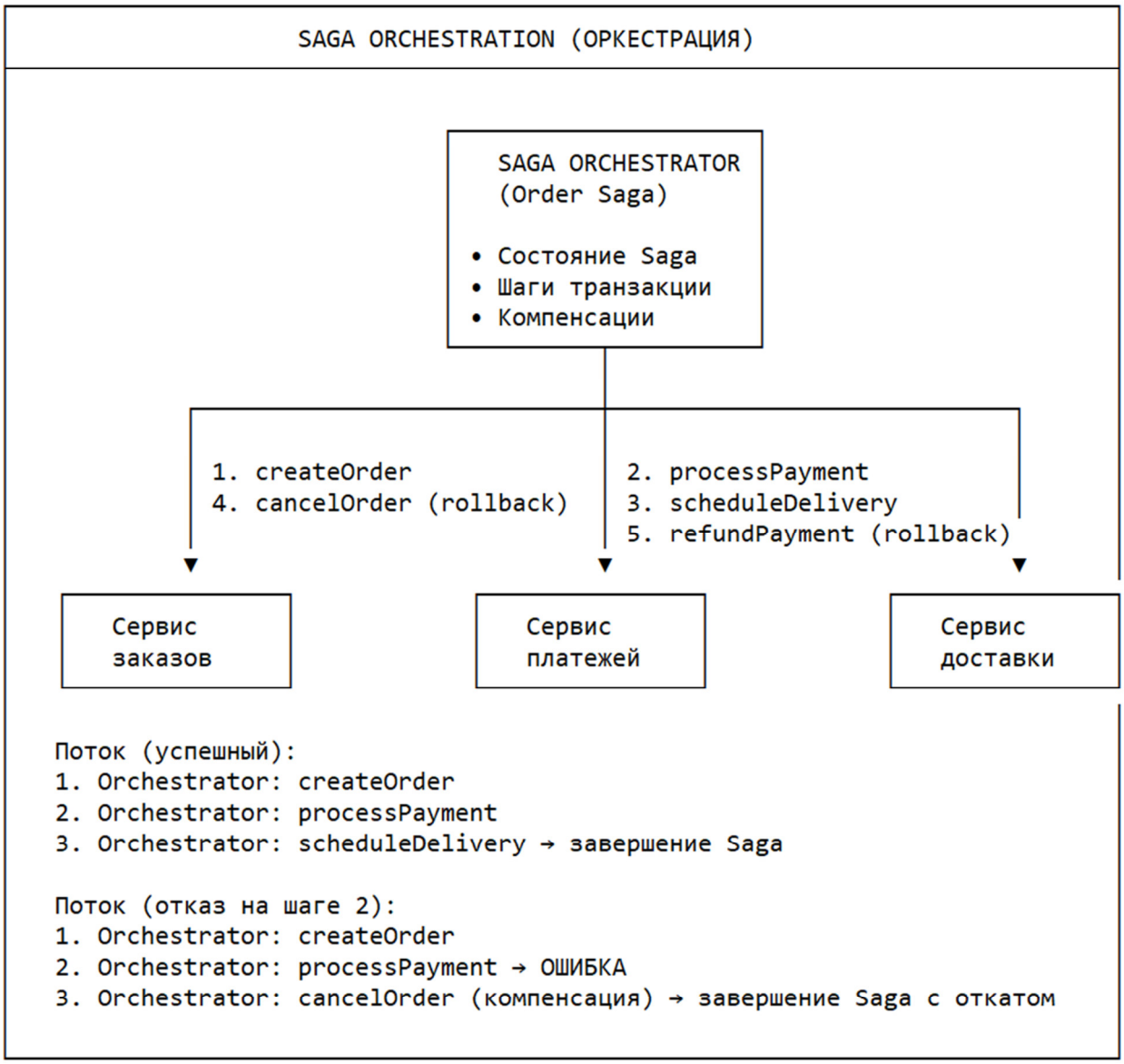
1.3. Saga Choreography (хореография)

Сервисы обмениваются событиями через брокер сообщений. Каждый сервис знает, какие события он должен обрабатывать и какие публиковать.



1.4. Saga Orchestration (оркестрация)

Центральный оркестратор (Saga Orchestrator) управляет последовательностью шагов, вызывает сервисы и обрабатывает сбои.



1.5. Компенсирующие транзакции

Каждый шаг Saga должен иметь компенсирующую транзакцию для отката.

Шаг (действие)	**Компенсация (откат)**
Создать заказ	Отменить заказ
Зарезервировать товар	Освободить резерв

Шаг (действие)	Компенсация (откат)
Списать оплату	Вернуть оплату
Отправить уведомление	(игнорировать)
Запланировать доставку	Отменить доставку

Идемпотентность - критически важное свойство для Saga. Команды и события должны быть идемпотентными, чтобы их можно было безопасно повторять.

```
# Пример идемпотентной операции
def process_payment(order_id, amount, idempotency_key):
    # Проверка, была ли уже обработана операция
    if redis.exists(f"payment:{idempotency_key}"):
        return {"status": "already_processed", "transaction_id":
redis.get(f"payment:{idempotency_key}")}

    # Выполнение операции
    transaction_id = perform_payment(order_id, amount)

    # Сохранение идемпотентности
    redis.setex(f"payment:{idempotency_key}", 86400, transaction_id)

    return {"status": "success", "transaction_id": transaction_id}
```

1.6. Инструменты для реализации Saga

Инструмент	Подход	Платформа	Особенности
Apache Kafka	Хореография	JVM	Высокая производительность, durable
RabbitMQ	Хореография	Erlang	Простая настройка, ACK

Инструмент	Подход	Платформа	Особенности
Camunda	Оркестрация	JVM	BPMN 2.0, визуальный дизайнер
Temporal	Оркестрация	Go/JVM	Долговременные workflow, отказоуст.
AWS Step Functions	Оркестрация	Cloud	Визуальный редактор, serverless
NATS	Хореография	Go	Лёгкий, высокая скорость
Eventuate Tram	Оба	JVM	Специализированная Saga библиотека
Axon Framework	Оба	JVM	CQRS + Event Sourcing + Saga

2. ПРИМЕРЫ ПРАКТИЧЕСКОГО ВЫПОЛНЕНИЯ

2.1. Saga Choreography с Kafka

2.1.1. Структура проекта

```
saga-choreography/
├── docker-compose.yml
├── order-service/
│   ├── app.py
│   └── Dockerfile
├── payment-service/
│   ├── app.py
│   └── Dockerfile
```

```
├── delivery-service/
│   ├── app.py
│   └── Dockerfile
├── kafka-setup/
│   └── init.sh
└── test_saga.py
```

2.1.2. Docker Compose (docker-compose.yml)

```yaml
version: '3.8'

services:
  zookeeper:
    image: confluentinc/cp-zookeeper:latest
    container_name: zookeeper
    environment:
      ZOOKEEPER_CLIENT_PORT: 2181
      ZOOKEEPER_TICK_TIME: 2000

  kafka:
    image: confluentinc/cp-kafka:latest
    container_name: kafka
    depends_on:
      - zookeeper
    environment:
      KAFKA_BROKER_ID: 1
      KAFKA_ZOOKEEPER_CONNECT: zookeeper:2181
      KAFKA_ADVERTISED_LISTENERS: PLAINTEXT://kafka:9092
      KAFKA_OFFSETS_TOPIC_REPLICATION_FACTOR: 1
    ports:
      - "9092:9092"

  order-service:
    build: ./order-service
    container_name: order-service
    ports:
      - "8001:8001"
    environment:
      KAFKA_HOST: kafka:9092
    depends_on:
      - kafka

  payment-service:
    build: ./payment-service
    container_name: payment-service
    ports:
```

```
      - "8002:8002"
    environment:
      KAFKA_HOST: kafka:9092
    depends_on:
      - kafka

  delivery-service:
    build: ./delivery-service
    container_name: delivery-service
    ports:
      - "8003:8003"
    environment:
      KAFKA_HOST: kafka:9092
    depends_on:
      - kafka
```

2.1.3. Order Service (choreography)

```
# order-service/app.py
import json
import uuid
from kafka import KafkaProducer, KafkaConsumer
from fastapi import FastAPI, HTTPException
from pydantic import BaseModel

app = FastAPI(title="Order Service (Saga Choreography)")

KAFKA_HOST = "kafka:9092"

# Producer для отправки событий
producer = KafkaProducer(
    bootstrap_servers=KAFKA_HOST,
    value_serializer=lambda v: json.dumps(v).encode('utf-8')
)

# Хранилище заказов (в реальности — БД)
orders = {}

class OrderRequest(BaseModel):
    user_id: int
    product: str
    amount: float

class OrderResponse(BaseModel):
    order_id: str
    status: str

@app.post("/api/orders", response_model=OrderResponse)
async def create_order(order: OrderRequest):
    order_id = str(uuid.uuid4())
```

```
    # Сохраняем заказ в статусе PENDING
    orders[order_id] = {
        "id": order_id,
        "user_id": order.user_id,
        "product": order.product,
        "amount": order.amount,
        "status": "PENDING"
    }

    # Отправляем событие OrderCreated в Kafka
    event = {
        "event_type": "OrderCreated",
        "order_id": order_id,
        "user_id": order.user_id,
        "product": order.product,
        "amount": order.amount
    }
    producer.send("orders", value=event)
    producer.flush()

    print(f"[OrderService] Order {order_id} created, event published")

    return OrderResponse(order_id=order_id, status="PENDING")

@app.get("/api/orders/{order_id}")
async def get_order(order_id: str):
    return orders.get(order_id, {"error": "Order not found"})

# Consumer для компенсирующих событий
def start_compensation_consumer():
    consumer = KafkaConsumer(
        "order_compensation",
        bootstrap_servers=KAFKA_HOST,
        value_deserializer=lambda v: json.loads(v.decode('utf-8')),
        auto_offset_reset='earliest'
    )

    for msg in consumer:
        event = msg.value
        if event["event_type"] == "PaymentFailed":
            order_id = event["order_id"]
            if order_id in orders and orders[order_id]["status"] == "PENDING":
                orders[order_id]["status"] = "CANCELLED"
                print(f"[OrderService] Order {order_id} cancelled due to payment
failure")

if __name__ == "__main__":
    import threading
    threading.Thread(target=start_compensation_consumer, daemon=True).start()
    import uvicorn
```

```
    uvicorn.run(app, host="0.0.0.0", port=8001)
```

2.1.4. Payment Service (choreography)

```
# payment-service/app.py
import json
import random
from kafka import KafkaProducer, KafkaConsumer
from fastapi import FastAPI

app = FastAPI(title="Payment Service (Saga Choreography)")

KAFKA_HOST = "kafka:9092"

producer = KafkaProducer(
    bootstrap_servers=KAFKA_HOST,
    value_serializer=lambda v: json.dumps(v).encode('utf-8')
)

# Хранилище платежей
payments = {}

def start_order_consumer():
    consumer = KafkaConsumer(
        "orders",
        bootstrap_servers=KAFKA_HOST,
        value_deserializer=lambda v: json.loads(v.decode('utf-8')),
        auto_offset_reset='earliest'
    )

    for msg in consumer:
        event = msg.value

        if event["event_type"] == "OrderCreated":
            order_id = event["order_id"]
            amount = event["amount"]

            # Имитация обработки платежа (90% успеха)
            success = random.random() > 0.1

            if success:
                payment_id = f"pay_{order_id}"
                payments[order_id] = {"status": "SUCCESS", "payment_id":
payment_id}

                result_event = {
                    "event_type": "PaymentProcessed",
                    "order_id": order_id,
                    "payment_id": payment_id
                }
                print(f"[PaymentService] Payment for {order_id} SUCCESS")
            else:
```

```
                result_event = {
                    "event_type": "PaymentFailed",
                    "order_id": order_id,
                    "reason": "Insufficient funds"
                }
                print(f"[PaymentService] Payment for {order_id} FAILED")

            producer.send("payments", value=result_event)
            producer.flush()

if __name__ == "__main__":
    import threading
    threading.Thread(target=start_order_consumer, daemon=True).start()
    import uvicorn
    uvicorn.run(app, host="0.0.0.0", port=8002)
```

2.2. Saga Orchestration (центральный оркестратор)

```
# orchestrator.py
import httpx
import asyncio
from enum import Enum
from typing import Dict, List

class SagaStepStatus(Enum):
    PENDING = "PENDING"
    COMPLETED = "COMPLETED"
    FAILED = "FAILED"

class SagaOrchestrator:
    def __init__(self):
        self.saga_steps = [
            {
                "name": "create_order",
                "action_url": "http://order-service:8001/api/orders",
                "compensation_url": "http://order-
service:8001/api/orders/{order_id}/cancel",
                "method": "POST",
                "status": SagaStepStatus.PENDING
            },
            {
                "name": "process_payment",
                "action_url": "http://payment-service:8002/api/payments",
                "compensation_url": "http://payment-
service:8002/api/payments/{payment_id}/refund",
                "method": "POST",
                "status": SagaStepStatus.PENDING
            },
            {
                "name": "schedule_delivery",
```

```
                "action_url": "http://delivery-service:8003/api/deliveries",
                "compensation_url": None,  # Некритичный шаг, компенсация не
нужна
                "method": "POST",
                "status": SagaStepStatus.PENDING
            }
        ]
        self.saga_id = None
        self.context = {}
        self.completed_steps = []

    async def execute_step(self, step: Dict) -> Dict:
        """Выполнение одного шага Saga"""
        url = step["action_url"]

        # Подстановка параметров из контекста
        if "{order_id}" in url and "order_id" in self.context:
            url = url.replace("{order_id}", str(self.context["order_id"]))
        if "{payment_id}" in url and "payment_id" in self.context:
            url = url.replace("{payment_id}", str(self.context["payment_id"]))

        async with httpx.AsyncClient(timeout=10.0) as client:
            if step["method"] == "POST":
                response = await client.post(url, json=self.context)
            else:
                response = await client.get(url)

            if response.status_code >= 400:
                raise Exception(f"Step {step['name']} failed: {response.text}")

            return response.json()

    async def compensate_step(self, step: Dict) -> bool:
        """Компенсация шага при отказе"""
        if not step["compensation_url"]:
            print(f"[Saga] No compensation for {step['name']}, skipping")
            return True

        url = step["compensation_url"]
        if "{order_id}" in url and "order_id" in self.context:
            url = url.replace("{order_id}", str(self.context["order_id"]))

        async with httpx.AsyncClient(timeout=10.0) as client:
            try:
                response = await client.post(url, json=self.context)
                return response.status_code < 400
            except Exception as e:
                print(f"[Saga] Compensation failed for {step['name']}: {e}")
                return False

    async def run_saga(self, saga_id: str, initial_context: Dict) -> Dict:
```

```
        """Запуск Saga оркестрации"""
        self.saga_id = saga_id
        self.context = initial_context
        self.completed_steps = []

        print(f"[Saga {saga_id}] Starting orchestration")

        for idx, step in enumerate(self.saga_steps):
            step["status"] = SagaStepStatus.PENDING
            print(f"[Saga {saga_id}] Executing step: {step['name']}")

            try:
                result = await self.execute_step(step)
                self.context.update(result)
                step["status"] = SagaStepStatus.COMPLETED
                self.completed_steps.append(step["name"])
                print(f"[Saga {saga_id}] Step {step['name']} completed")

            except Exception as e:
                print(f"[Saga {saga_id}] Step {step['name']} FAILED: {e}")
                step["status"] = SagaStepStatus.FAILED

                # Откат (компенсация) пройденных шагов в обратном порядке
                print(f"[Saga {saga_id}] Initiating compensation...")
                for prev_step in reversed(self.saga_steps[:idx]):
                    if prev_step["status"] == SagaStepStatus.COMPLETED:
                        print(f"[Saga {saga_id}] Compensating:
{prev_step['name']}")
                        await self.compensate_step(prev_step)

                return {
                    "status": "FAILED",
                    "failed_step": step["name"],
                    "error": str(e)
                }

        print(f"[Saga {saga_id}] Saga completed successfully")
        return {
            "status": "SUCCESS",
            "saga_id": saga_id,
            "context": self.context
        }

# FastAPI endpoints for orchestrator
from fastapi import FastAPI
app = FastAPI(title="Saga Orchestrator")

orchestrator = SagaOrchestrator()

@app.post("/api/saga/order")
async def start_order_saga(order_data: dict):
```

```
    saga_id = str(uuid.uuid4())
    result = await orchestrator.run_saga(saga_id, order_data)
    return result
```

МЕТОДИЧЕСКИЕ УКАЗАНИЯ ДЛЯ ВЫПОЛНЕНИЯ ПРАКТИЧЕСКОЙ ЧАСТИ РАБОТЫ

Задачи работы:

1. **Реализация Saga Choreography:**
   - Развернуть Kafka/RabbitMQ
   - Реализовать 3 микросервиса (Order, Payment, Delivery)
   - Настроить обмен событиями
   - Реализовать компенсацию при отказе платежа
2. **Реализация Saga Orchestration:**
   - Создать центральный оркестратор
   - Определить шаги транзакции
   - Реализовать компенсацию при сбое на любом шаге
3. **Эксперимент 1: Успешный сценарий:**
   - Запустить Saga с корректными данными
   - Проследить выполнение всех шагов
4. **Эксперимент 2: Сценарий с отказом:**
   - Инициировать сбой на шаге 2 (платёж)
   - Проследить выполнение компенсации (отмена заказа)
5. **Сравнение подходов:**
   - Сложность реализации
   - Прозрачность отладки
   - Возможность переиспользования

3. ВАРИАНТЫ ЗАДАНИЙ

3.1. Базовый уровень (15 вариантов)

Требования: Saga Choreography, 3 сервиса, компенсация при отказе

№	Бизнес-процесс	Шаг 1	Шаг 2	Шаг 3	Компенсация
1	Заказ товара	Создать заказ	Списать оплату	Зарезервировать товар	Отмена заказа
2	Бронирование отеля	Забронировать номер	Списать депозит	Отправить подтверждение	Отмена брони
3	Доставка еды	Создать заказ	Списать оплату	Назначить курьера	Возврат оплаты
4	Регистрация на курс	Создать запись	Списать оплату	Выдать доступ	Отмена записи
5	Аренда авто	Забронировать авто	Списать оплату	Застраховать	Отмена брони
6	Покупка билетов	Забронировать места	Списать оплату	Сгенерировать билеты	Возврат билетов
7	Подписка на сервис	Создать подписку	Списать оплату	Активировать доступ	Отмена подписки
8	Перевод средств	Заблокировать счёт A	Перевести средства	Разблокировать счёт B	Возврат средств
9	Заказ такси	Создать заказ	Назначить водителя	Списать оплату	Отмена заказа
10	Онлайн-консультация	Записаться на время	Списать оплату	Отправить ссылку на звонок	Возврат оплаты
11	Страхование	Создать полис	Списать оплату	Активировать покрытие	Отмена полиса
12	Хостинг (VM)	Создать VM	Списать оплату	Настроить DNS	Удалить VM
13	Облачное хранилище	Создать bucket	Списать оплату	Настроить ключи	Удалить bucket
14	CDN-сервис	Создать	Списать	Настроить правила	Удалить origin

№	Бизнес-процесс	Шаг 1	Шаг 2	Шаг 3	Компенсация
		origin	оплату		
15	Маркетплейс	Создать заказ	Зарезервировать товар у продавца	Списать оплату	Отмена заказа

3.2. Средний уровень (15 вариантов)

Требования: как базовый + Idempotency + Dead Letter Queue + Retry

№	Транзакция	Дополнительный шаг 4	Компенсация 4	Механизм Retry
1	Бронирование путешествия	Забронировать трансфер	Отменить трансфер	Экспоненциальный backoff
2	Страхование	Зарегистрироваться в реестре	Удалить из реестра	Фиксированные интервалы
3	Ипотека	Создать заявку	Отменить заявку	Retry с разными серверами
4	Оплата с кэшбэком	Начислить кэшбэк	Отозвать кэшбэк	Retry с ограничением времени
5	Видеоконференция	Создать meeting room	Удалить room	Retry с учётом idempotency
6	Заказ с доставкой	Отправить уведомление	(нет)	Retry с dead letter
7	Платеж с флагами антифрода	Проверить anti-fraud	(нет)	Retry в DLQ после 3 попыток
8	Оформление	Создать	Отменить	Exponential backoff + jitter

№	Транзакция	Дополнительный шаг 4	Компенсация 4	Механизм Retry
	рассрочки	рассрочку	рассрочку	
9	Оплата частями (Split payment)	Создать суб-платеж (часть)	Отменить суб-платеж	Распределённый retry
10	Кэшбэк + бонус	Начислить бонусные баллы	Списать баллы	Retry с переключением партиции
11	Агрегация заказов (bulk)	Обновить агрегированную таблицу	Откатить агрегацию	Retry + batch
12	Подписка с пробным периодом	Активировать пробный период	Деактивировать пробный	Retry с учётом TTL
13	Конвертация валюты	Заблокировать курс	Разблокировать курс	Retry с учётом истечения курса
14	Бронирование подарка	Зарезервировать подарочную упаковку	Освободить упаковку	Retry с Dead Letter Topic
15	Комплексный заказ	Резерв всех товаров из разных магазинов	Освободить резерв	Retry с учётом split-brain

3.3. Продвинутый уровень (15 вариантов)

Требования: Orchestration + State persistence + Monitoring

№	Тип оркестрации	Хранение состояния	Мониторинг	Сценарий отказа

№	Тип оркестрации	Хранение состояния	Мониторинг	Сценарий отказа
1	BPMN (Camunda)	PostgreSQL	Camunda Cockpit	Отказ на шаге 2 → компенсация
2	Temporal	MySQL	Temporal UI	Отказ на шаге 2 → компенсация
3	AWS Step Functions (эмуляция)	DynamoDB (эмул.)	CloudWatch	Отказ на шаге 3 → компенсация
4	Custom (FastAPI + Redis)	Redis	Prometheus + Grafana	Отказ на шаге 2 → компенсация
5	Custom + Event Sourcing	Event store (Kafka)	Kafka Lag monitoring	Отказ на шаге 2 → компенсация
6	Axon Framework (Java)	Axon Server	Axon Dashboard	Отказ на шаге 2 → компенсация
7	Camunda + Kafka Connector	PostgreSQL + Kafka	Grafana + Loki	Отказ на шаге 2 → компенсация
8	Temporal + gRPC	CockroachDB	Jaeger + Prometheus	Отказ на шаге 2 → компенсация

№	Тип оркестрации	Хранение состояния	Мониторинг	Сценарий отказа
9	Custom + SAGA tx log	PostgreSQL	Graphana	Отказна шаге 3 → компенсация
10	HTTP orchestration + компенсация	Redis (состояние)	ELK	Отказ на шаге 2 → компенсация
11	Step Functions + retry	DynamoDB (состояние)	X-Ray	Отказ на шаге 2 с ручным вмешательством
12	Orchestrator + dead letter queue	RabbitMQ + PostgreSQL	Prometheus + Alertmanager	Отказ на шаге 2 → dead letter
13	Choreography + event-carried state	Kafka Streams	Confluent Control Center	Отказ на шаге 2 → компенсация
14	Temporal + saga tracking	Cassandra	Grafana + Loki	Отказ на шаге 2 (сбой оркестратора)
15	AWS Step Functions + Lambda	DynamoDB	CloudWatch + X-Ray	Отказ на шаге 2 → компенсация

4. КОНТРОЛЬНЫЕ ВОПРОСЫ

1. В чём суть паттерна Saga и какую проблему он решает?
2. Чем отличается хореография от оркестрации в Saga?
3. Что такое компенсирующая транзакция?
4. Почему необходим механизм идемпотентности в Saga?
5. Какой подход проще для отладки: хореография или оркестрация?
6. Какие инструменты поддерживают Saga из коробки?
7. Чем Saga отличается от 2PC (двухфазного коммита)?
8. Как реализовать состояние Saga в оркестраторе?
9. Что происходит, если компенсирующая транзакция сама не удалась?
10. Как обрабатывать долгоидущие Saga (hours/days)?
11. В чём роль брокера сообщений в хореографии?
12. Как транзакции влияют на производительность по сравнению с 2PC?
13. Как обеспечить видимость промежуточных состояний Saga?
14. Как избежать циклических зависимостей в хореографии?
15. Почему обычные ACID-транзакции не подходят для микросервисов?

5. ТРЕБОВАНИЯ К ОТЧЕТУ

1. **Титульный лист**
2. **Цель работы**
3. **Теоретическая часть:** паттерн Saga, хореография vs оркестрация
4. **Практическая часть:**
   - Листинг кода сервисов (хореография)
   - Листинг оркестратора
   - Листинг Docker Compose
5. **Результаты экспериментов:**
   - Логи успешного выполнения Saga
   - Логи выполнения компенсации при отказе

- Демонстрация идемпотентности

6. **Сравнение подходов:** хореография vs оркестрация
7. **Вывод**
8. **Ответы на контрольные вопросы**

6. КРИТЕРИИ ОЦЕНИВАНИЯ

Критерий	**Базовый (10)**	**Средний (14)**	**Продвинутый (18)**
Микросервисы (3 шт) развёрнуты	2	1	1
Хореография (события) работает	4	2	1
Компенсация при отказе работает	2	2	1
Idempotency реализована	—	3	2
Оркестратор реализован	—	2	2
State persistence	—	2	2
Мониторинг (Prometheus + Grafana)	—	—	2
Dead Letter обработка	—	—	2
Качество отчёта	2	2	4
Максимум	**10**	**14**	**18**

Шкала перевода в оценку:

- **5 (отлично):** 90%+
- **4 (хорошо):** 75–89%
- **3 (удовлетворительно):** 60–74%
- **2 (неудовлетворительно):** <60%

7. ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ

Программа	Назначение	Доступность в РФ
Docker + Compose	Контейнеризация	+
Apache Kafka (Confluent образ)	Брокер событий (хореография)	+
RabbitMQ (альтернатива)	Брокер	+
Python + kafka-python, httpx	Сервисы и оркестратор	+
FastAPI	Фреймворк	+
Redis (опционально)	Состояние Saga	+
PostgreSQL (опционально)	State persistence	+

Лабораторная работа №13

Архитектурный анализ методом ATAM (Architecture Tradeoff Analysis Method)

Цель работы: Сформировать практические навыки применения метода ATAM (Architecture Tradeoff Analysis Method) для формальной оценки архитектуры программных систем, выявления архитектурных рисков, чувствительных точек и компромиссов между атрибутами качества, а также

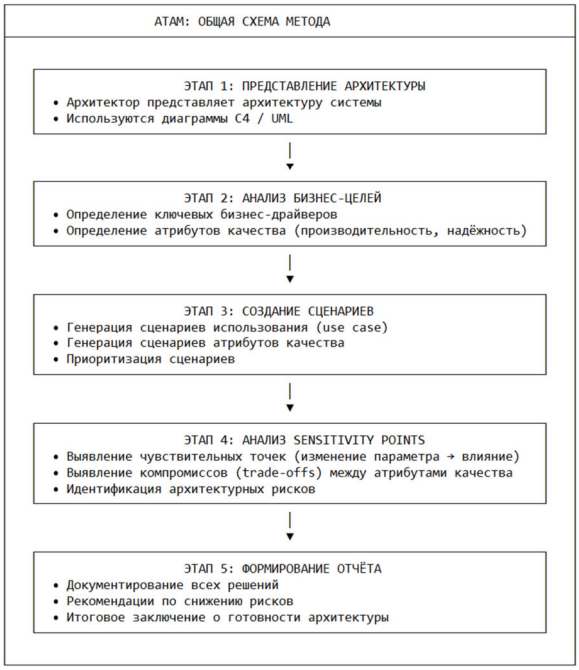
документирования результатов анализа для принятия обоснованных архитектурных решений.

Формируемые компетенции: ПК-5

1. ТЕОРЕТИЧЕСКОЕ ОБОСНОВАНИЕ

1.1. Что такое ATAM

ATAM (Architecture Tradeoff Analysis Method) - это метод структурированного анализа архитектуры программных систем, разработанный в Институте программной инженерии Университета Карнеги-Меллон (SEI CMU). ATAM позволяет оценить архитектуру на соответствие бизнес-целям и атрибутам качества, а также выявить компромиссы (trade-offs) между ними.



1.2. Ключевые понятия ATAM

Термин	Определение	Пример
Атрибут качества (Quality Attribute)	Характеристика системы, влияющая на её функционирование	Производительность, доступность, безопасность, модифицируемость

Термин	Определение	Пример
Сценарий (Scenario)	Конкретное описание взаимодействия с системой от заинтересованного лица	«При 1000 одновременных запросов время отклика < 200 мс»
Чувствительная точка (Sensitivity Point)	Архитектурное решение, которое существенно влияет на атрибут качества	Количество реплик сервиса (влияет на доступность)
Компромисс (Trade-off)	Ситуация, когда одно архитектурное решение позитивно влияет на один атрибут качества, но негативно — на другой	Кэширование ↑ производительность, но ↓ консистентность
Риск (Risk)	Архитектурное решение, которое может привести к проблемам в будущем	Использование одной БД для всех сервисов (SPOF)
Нерешаемый момент (Non-negotiable)	Требование или ограничение, которое не может быть изменено	Законодательные требования (GDPR, ФЗ-152)

1.3. Атрибуты качества (Quality Attributes)

Атрибут	Описание	Метрики
Производительность	Способность системы обрабатывать нагрузку	RPS, latency (p50, p95, p99), throughput

Атрибут	Описание	Метрики
Доступность	Процент времени, когда система доступна	Uptime %, MTBF, MTTR, SLA (99.9%, 99.99%)
Надёжность	Вероятность безотказной работы	Частота отказов, время между отказами
Безопасность	Защита от несанкционированного доступа	Количество уязвимостей, время взлома
Модифицируемость	Лёгкость внесения изменений	Время реализации изменения, объём кода
Масштабируемость	Способность расти с нагрузкой	Максимальное число пользователей, горизонтальное расширение
Тестируемость	Лёгкость проверки системы	Покрытие тестами, время выполнения тестов
Развертываемость	Лёгкость установки и обновления	Время развёртывания, частота релизов

1.4. Структура сценария ATAM

Каждый сценарий в ATAM должен содержать 6 элементов:

```
                    СТРУКТУРА СЦЕНАРИЯ ATAM

1. ИСТОЧНИК (Source of stimulus)
   Кто или что инициирует сценарий?
   Пример: "Пользователь", "Внешняя система", "Администратор"

2. СТИМУЛ (Stimulus)
   Какое действие происходит?
   Пример: "Отправляет запрос", "Выполняет вход"

3. СРЕДА (Environment)
   В каких условиях происходит сценарий?
   Пример: "При 1000 активных пользователей", "В ночное время"

4. АРТЕФАКТ (Artifact)
   Какой компонент системы задействован?
   Пример: "API Gateway", "Сервис заказов"

5. ОТВЕТ (Response)
   Как система должна отреагировать?
   Пример: "Вернуть результат за < 200 мс"

6. ИЗМЕРЕНИЕ ОТВЕТА (Response measure)
   Как измерить качество ответа?
   Пример: "p95 latency измеряется в миллисекундах"
```

1.5. Примеры сценариев ATAM

Сценарий производительности (Performance):

```
Источник:   Пользователь
Стимул:     Отправляет запрос на получение списка заказов
Среда:      В часы пик (19:00-21:00)
Артефакт:   Сервис заказов
Ответ:      Система возвращает список заказов
Измерение:  Время ответа (p95) ≤ 500 мс
```

Сценарий доступности (Availability):

```
Источник:   Отказ оборудования
Стимул:     Один из узлов базы данных выходит из строя
Среда:      В рабочее время
Артефакт:   Кластер базы данных
Ответ:      Система продолжает обрабатывать запросы
Измерение:  Доступность ≥ 99.99%, RTO ≤ 60 с, RPO ≤ 5 с
```

Сценарий безопасности (Security):

```
Источник:   Злоумышленник
Стимул:     Пытается выполнить SQL-инъекцию в API
Среда:      Через общедоступный интернет
Артефакт:   API Gateway
Ответ:      Запрос отклоняется, событие логируется
Измерение:  100% неавторизованных запросов отклоняются
```

Сценарий модифицируемости (Modifiability):

```
Источник:   Разработчик
Стимул:     Добавляет новое поле в модель заказа
Среда:      В процессе разработки
Артефакт:   Сервис заказов
Ответ:      Изменение вносится без каскадных правок
Измерение:  Время реализации ≤ 1 дня, затронуто ≤ 3 модулей
```

1.6. Чувствительные точки и компромиссы

Чувствительная точка (Sensitivity Point):

Определённый параметр или свойство архитектуры, изменение которого напрямую влияет на атрибут качества.

Архитектурное решение	Чувствительная точка	Влияние на атрибут качества
Количество реплик	Число реплик N	Доступность (N↑ →

Архитектурное решение	Чувствительная точка	Влияние на атрибут качества
		доступность↑)
Размер кэша	Объём памяти Redis	Производительность (hit rate)
Параметры Circuit Breaker	Таймаут, порог ошибок	Доступность, стабильность

Компромисс (Trade-off):

Ситуация, когда улучшение одного атрибута качества ухудшает другой.

```
ПРИМЕРЫ КОМПРОМИССОВ (TRADE-OFFS)

1. КЭШИРОВАНИЕ
   Плюс: производительность (ответ из кэша быстрее)
   Минус: консистентность (данные могут быть устаревшими)

2. СИНХРОННАЯ vs АСИНХРОННАЯ РЕПЛИКАЦИЯ
   Синхронная: высокая консистентность, низкая производительность записи
   Асинхронная: высокая производительность записи, риск потери данных

3. МОНОЛИТ vs МИКРОСЕРВИСЫ
   Монолит: простота разработки, сложность масштабирования
   Микросервисы: масштабируемость, сложность распределённых транзакций

4. ЗАПИСЬ В НЕСКОЛЬКОХ ШАРДАХ
   Плюс: горизонтальное масштабирование, распределение нагрузки
   Минус: сложные JOIN и глобальные агрегации

5. НОРМАЛИЗАЦИЯ vs ДЕНОРМАЛИЗАЦИЯ
   Нормализация: консистентность, сложность запросов (JOIN)
   Денормализация: скорость чтения, избыточность данных, риск аномалий
```

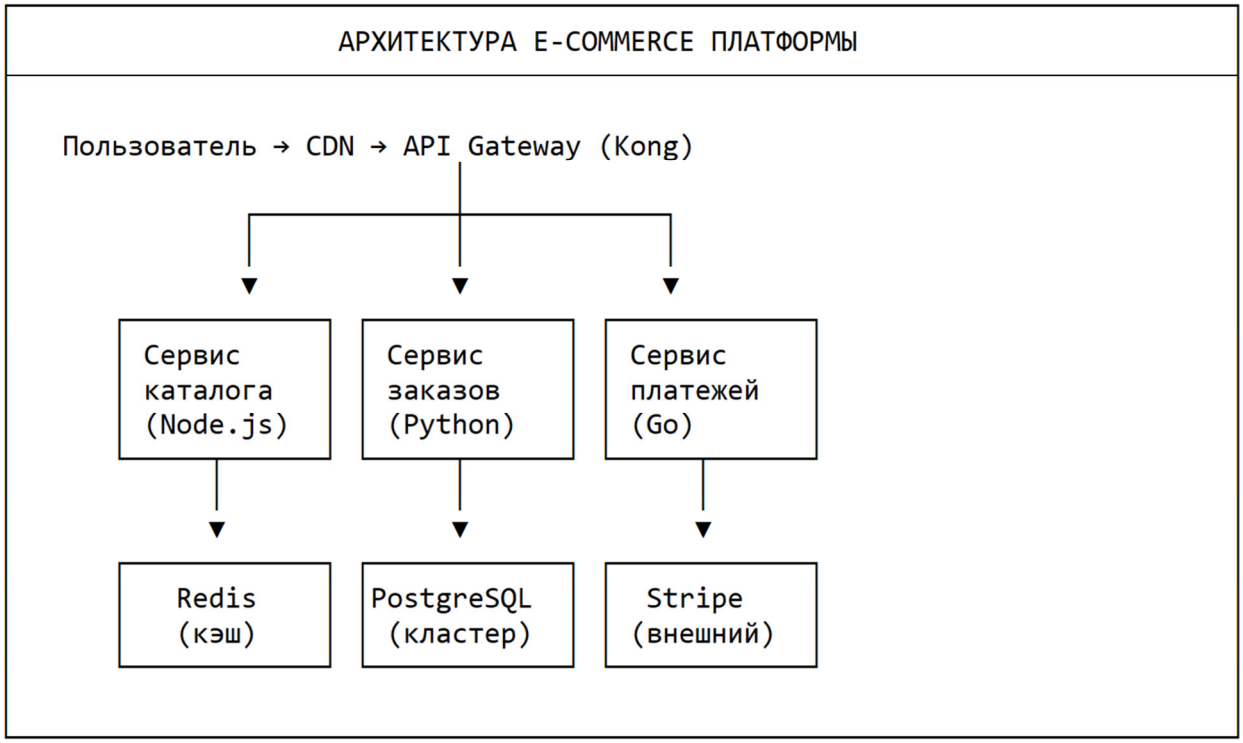
2. ПРИМЕРЫ ПРАКТИЧЕСКОГО ВЫПОЛНЕНИЯ

2.1. Система для АТАМ-анализа: E-commerce платформа

Описание системы:

- Интернет-магазин с 1 млн товаров и 100 тыс. заказов в день
- Пиковая нагрузка: 5000 RPS
- Требования к доступности: 99.99% (≈ 52 минуты даунтайма в год)
- Требования к отказоустойчивости: быстрый failover (< 1 минуты)

Архитектура (упрощённо):



2.2. Проведение ATAM-анализа

2.2.1. Этап 1: Представление архитектуры

Архитектура представлена с использованием C4 model (см. ЛР №1). Диаграммы включают:

- Контекстную диаграмму (пользователи, внешние платежные системы)
- Диаграмму контейнеров (API Gateway, микросервисы, БД, кэш)

2.2.2. Этап 2: Определение бизнес-драйверов

ID	Бизнес-драйвер	Приоритет

ID	Бизнес-драйвер	Приоритет
BD-1	Увеличение конверсии: скорость загрузки страниц < 1 сек	Высокий
BD-2	Рост базы пользователей: масштабирование до 10 млн пользователей	Высокий
BD-3	Стабильность в пики (Чёрная пятница): выдерживать нагрузку 10 000 RPS	Критический
BD-4	Соответствие PCI DSS для обработки платежей	Критический
BD-5	Время восстановления при сбое < 5 минут	Высокий

2.2.3. Этап 3: Создание сценариев

Сценарий производительности (S-PERF-1):

Источник: Пользователь
Стимул: Поиск товаров по категории
Среда: Час пик (Black Friday)
Артефакт: Сервис каталога + Redis
Ответ: Система возвращает результаты поиска
Измерение: p95 latency ≤ 500 мс при 5000 RPS

Сценарий доступности (S-AVAIL-1):

Источник: Отказ оборудования

Стимул: Узел PostgreSQL выходит из строя

Среда: Рабочее время

Артефакт: Кластер PostgreSQL

Ответ: Система продолжает обрабатывать запросы на чтение

Измерение: Доступность записи восстанавливается через ≤ 60 с

Сценарий безопасности (S-SEC-1):

Источник: Злоумышленник

Стимул: Попытка подбора пароля (brute force)

Среда: Через API аутентификации

Артефакт: API Gateway

Ответ: Блокировка IP после 5 неудачных попыток

Измерение: 100% заблокированных IP, события в логах

Сценарий модифицируемости (S-MOD-1):

Источник: Разработчик

Стимул: Добавление поля "скидка" в модель заказа

Среда: Спринт разработки

Артефакт: Сервис заказов + БД

Ответ: Изменение вносится без каскадных правок

Измерение: Затронуто ≤ 2 микросервисов, время ≤ 4 часов

2.2.4. Этап 4: Выявление Sensitivity Points и Trade-offs

Таблица чувствительных точек:

ID	Архитектурное решение	Чувствительная точка	Влияние на атрибут качества

ID	Архитектурное решение	Чувствительная точка	Влияние на атрибут качества
SP-1	Количество реплик сервиса заказов	N (число реплик)	Доступность (увеличение N → доступность↑)
SP-2	Размер кэша Redis	Объём памяти (MB)	Производительность (hit rate)
SP-3	Уровень консистенции Cassandra (W+R > N)	Значения W и R	Доступность/Согласованность
SP-4	Таймаут Circuit Breaker	Таймаут (сек)	Доступность/Производительность
SP-5	Стратегия шардирования	Ключ шардирования	Производительность (горячие ключи)

Таблица компромиссов (Trade-offs):

ID	Компромисс	Положительный эффект	Отрицательный эффект	Решение
TO-1	Кэширование каталога	↑ Производительность (cache hits)	↓ Консистентность (устаревшие цены)	TTL 60 сек, при обновлении - инвалидация
TO-2	Асинхронная	↑	↓	DLQ +

ID	Компромисс	Положительный эффект	Отрицательный эффект	Решение
	отправка email	Производительность заказа (не блокирует)	Доставка не гарантирована сразу	повторные попытки
ТО-3	Синхронная репликация БД	↑ Согласованность (RPO=0)	↓ Производительность записи	Критические данные — синхронно, логи — асинхронно
ТО-4	Шардирование заказов	↑ Горизонтальное масштабирование	↓ Сложность глобальных отчётов	OLTP — шардирование, OLAP — отдельное хранилище

2.2.5. Этап 5: Идентификация рисков

Таблица рисков:

ID	Риск	Вероятность	Влияние	Митигация
R-1	База данных становится узким местом (single point of failure)	Средняя	Высокое	Репликация + шардирование + Read Replicas
R-2	Некорректная настройка таймаутов приводит к каскадным	Средняя	Высокое	Circuit Breaker + Bulkhead + мониторинг

ID	Риск	Вероятность	Влияние	Митигация
	отказам			
R-3	Отказ платёжного шлюза останавливает оформление заказов	Низкая	Критическое	Альтернативный шлюз (fallback)
R-4	Недостаточное покрытие тестами нового функционала	Высокая	Средняя	Shift-left testing, обязательный code review
R-5	Зависимость от внешнего CDN (если недоступен)	Низкая	Высокое	Fallback на локальные статические файлы

2.2.6. Итоговый отчёт ATAM

markdown

 # ATAM Report: E-commerce Platform

1. Executive Summary

Архитектура признана соответствующей бизнес-целям.

Выявлены 5 рисков, 4 компромисса, 5 чувствительных точек.

2. Key Findings

- Основной риск: SPOF — шардирование для заказов критично

- Ключевой компромисс: кэширование каталога (производительность vs консистенция)

- Принятые решения: асинхронная обработка email, синхронная репликация БД

3. Recommendations

1. Внедрить мониторинг hit rate кэша Redis
2. Настроить автоматический failover для PostgreSQL (Patroni)
3. Добавить альтернативный платёжный шлюз
4. Провести нагрузочное тестирование перед Black Friday

4. Conclusion

Архитектура готова к реализации при условии выполнения рекомендаций.

МЕТОДИЧЕСКИЕ УКАЗАНИЯ ДЛЯ ВЫПОЛНЕНИЯ ПРАКТИЧЕСКОЙ ЧАСТИ РАБОТЫ

Задачи работы:

1. **Выбор системы для анализа:**
   - Выбрать архитектуру (реальную или из кейса) для ATAM-анализа
   - Подготовить описание системы и её диаграммы
2. **Определение бизнес-драйверов:**
   - Сформулировать 3-5 ключевых бизнес-целей системы
   - Определить приоритет атрибутов качества
3. **Создание сценариев:**
   - Сгенерировать минимум 5 сценариев для разных атрибутов качества
   - Приоритизировать сценарии (высокий/средний/низкий приоритет)

4. **Анализ архитектуры:**
   - Выявить чувствительные точки
   - Выявить компромиссы между атрибутами качества
   - Идентифицировать архитектурные риски
5. **Формирование отчёта:**
   - Оформить результаты ATAM-анализа в структурированный отчёт
   - Подготовить рекомендации по улучшению архитектуры
6. **Защита результатов:**
   - Представить анализ перед группой (ролевая игра: архитектор + стейкхолдеры)

3. ВАРИАНТЫ ЗАДАНИЙ

3.1. Базовый уровень (15 вариантов)

Требования: выполнить ATAM-анализ для заданной архитектуры

- Представить архитектуру (C4 диаграммы)
- 10 сценариев (по 2 на атрибут качества)
- Таблица чувствительных точек
  - Таблица рисков

№	Система для анализа	Атрибуты качества (фокус)
	E-commerce платформа	Производительность, доступность
	Система бронирования авиабилетов	Доступность, консистентность
	Платформа онлайн-кинотеатра	Производительность, масштабируемость

№	Система для анализа	Атрибуты качества (фокус)
	Система мгновенных сообщений (чат)	Доступность, latency
	Банковское приложение (mobile)	Безопасность, доступность
	Система управления логистикой	Надёжность, производительность
	Платформа для видеоконференций	Масштабируемость, latency
	CRM для крупного предприятия	Модифицируемость, безопасность
	Социальная сеть с лентой новостей	Производительность, масштабируемость
0	Система онлайн-образования	Доступность, тестируемость
1	Маркетплейс (аналог Avito)	Масштабируемость, безопасность
2	Система управления складами (WMS)	Надёжность, модифицируемость
3	Платформа доставки еды	Производительность, доступность
4	Система заказа такси (Uber clone)	Latency, масштабируемость

№	Система для анализа	Атрибуты качества (фокус)
5	IoT-платформа для сенсоров	Производительность, надёжность

3.2. Средний уровень (15 вариантов)

Требования: как базовый + количественные метрики + сценарии для всех 6 атрибутов качества

№	Система	Дополнительные атрибуты	SLA требований
1	E-commerce + рекомендации	Безопасность, тестируемость	99.99% доступности
2	Бронирование авиабилетов + лояльность	Производительность, надёжность	99.999% (5 девяток)
3	Онлайн-кинотеатр + live events	Безопасность, модифицируемость	50 ms p99 latency
4	Чат + видеозвонки	Производительность, масштабируемость	100k одновременных пользователей
5	Банк + инвестиции	Безопасность, доступность	RTO 30 с, RPO 0
6	Логистика +	Надёжность,	99.99% uptime

№	Система	Дополнительные атрибуты	SLA требований
	трекинг	производительность	
7	Видеоконференции + запись	Модифицируемость, безопасность	250 ms max latency
8	CRM + маркетинг	Масштабируемость, тестируемость	10 млн пользователей
9	Соцсеть + лента + stories	Производительность, доступность	5000 RPS на пике
10	Образование + live вебинары	Масштабируемость, безопасность	1000 групп одновременно
11	Маркетплейс + оплата	Безопасность, надёжность	99.95% доступности
12	WMS + роботизация	Модифицируемость, производительность	10k операций/сек
13	Доставка + трекинг курьеров (live)	Latency, доступность	3 сек max latency
14	Такси + динамическое ценообразование	Производительность, масштабируемость	50k RPS
15	IoT + real-time аналитика	Надёжность, производительность	1 млн устройств

3.3. Продвинутый уровень (15 вариантов)

Требования: как средний + полный ATAM (все шаги) + ролевая игра (защита)

№	Система	Архитектурное решение (для Trade-off)	Стейкхолдеры
1	Fintech платформа	ACID vs BASE (согласованность)	CTO, risk-manager
2	Глобальный стриминг (Netflix-like)	CDN vs edge caching	product, infra leads
3	Банковский core-banking	Репликация синхронная vs асинхронная	compliance, devops
4	Телемедицина	Монолит vs микросервисы	мед. эксперт, CISO
5	Роботизированная доставка (дроны)	Cloud vs edge computing	operation lead, CTO
6	Цифровая платформа для биометрии	Centralised vs federated storage	security, legal
7	Система госуслуг (портал)	Kafka vs REST (межсервисное общение)	product owner, infra
8	Промышленный IoT	MQTT vs AMQP	head of industrial,

№	Система	Архитектурное решение (для Trade-off)	Стейкхолдеры
			dev
9	Игровая платформа (real-time)	gRPC vs WebSocket	game director, CTO
10	SaaS CRM с AI (chatbot)	Встраивание модели (sync/async)	AI lead, product
11	Автоматизация склада (AGV)	Centralized orchestrator vs distributed	ops, architect
12	Авиационная платформа (предписание техобслуживания)	In-memory grid vs relational DB	aircraft engineer, data
13	Цифровая платформа для университета	LTI vs custom API	IT director, academic
14	Умный город (камеры, датчики)	Kafka vs Redis Streams	city IT lead
15	Средство мониторинга (observability)	Push vs pull метрики	SRE, development team

4. КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Что такое ATAM и для чего он используется?
2. Какие этапы включает ATAM?
3. Что такое атрибуты качества? Перечислите основные.
4. Из каких элементов состоит сценарий в ATAM?
5. Что такое чувствительная точка (sensitivity point)?
6. Что такое архитектурный компромисс (trade-off)? Приведите пример.
7. Как выявляются архитектурные риски в ATAM?
8. Кто участвует в ATAM-сессии?
9. Чем ATAM отличается от других методов оценки архитектуры (SAAM, CBAM)?
10. Какую роль играют диаграммы C4 в ATAM?
11. Как приоритизировать сценарии в ATAM?
12. Как документируются результаты ATAM?
13. Как часто следует проводить ATAM?
14. Можно ли проводить ATAM для существующей системы?
15. Какие преимущества даёт ATAM для высоконагруженных систем?

5. ТРЕБОВАНИЯ К ОТЧЕТУ

1. **Титульный лист**
2. **Цель работы**
3. **Теоретическая часть:** ATAM, атрибуты качества, сценарии, чувствительные точки, компромиссы
4. **Практическая часть:**
   - Описание системы (C4 диаграммы)
   - Бизнес-драйверы (BD)
   - Сценарии (10–15 штук)
   - Таблица чувствительных точек
   - Таблица компромиссов

- Таблица рисков

5. **Рекомендации:** по снижению рисков и улучшению архитектуры
6. **Вывод:** готова ли архитектура к реализации?
7. **Ответы на контрольные вопросы**

---

6. КРИТЕРИИ ОЦЕНИВАНИЯ

Критерий	**Базовый (10)**	**Средний (14)**	**Продвинутый (18)**
Описание системы	2	1	1
Бизнес-драйверы (3–5)	2	2	1
Сценарии (5/10/15 шт)	2	2	2
Чувствительные точки	2	2	2
Компромиссы	2	2	2
Риски (таблица с митигацией)	2	2	2
ATAM отчёт (структура)	—	2	2
Защита (ролевая игра)	—	1	2
Качество отчёта	—	—	3
Максимум	**10**	**14**	**18**

Шкала перевода в оценку:

- **5 (отлично):** 90%+
- **4 (хорошо):** 75–89%

- **3 (удовлетворительно):** 60–74%
- **2 (неудовлетворительно):** <60%

7. ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ

Программа	Назначение	Доступность в РФ
Draw.io / PlantUML	Диаграммы C4	+
Mermaid	Альтернативный инструмент	+
Markdown-редактор	Оформление отчёта	+
Excel / Google Sheets	Таблицы рисков	+
Any Презентация	Защита результатов	+

Лабораторная работа №14

Нагрузочное тестирование с k6

Цель работы: Сформировать практические навыки проведения нагрузочного и стресс-тестирования высоконагруженных систем с использованием современного инструментария k6 (Grafana Labs), включая разработку сценариев тестирования (smoke, spike, stress, soak), анализ результатов и интеграцию тестов в CI/CD-конвейер для автоматического контроля производительности.

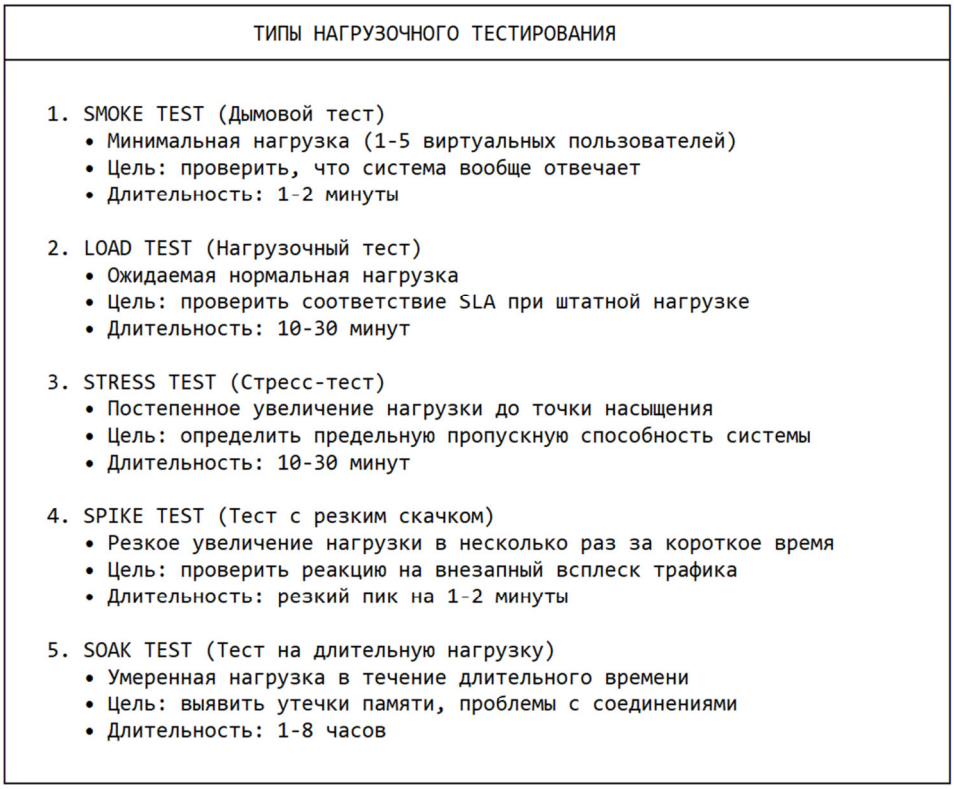
Формируемые компетенции: ПК-5

1. ТЕОРЕТИЧЕСКОЕ ОБОСНОВАНИЕ

1.1. Значение нагрузочного тестирования

Нагрузочное тестирование (Performance Testing) — это процесс оценки поведения системы под определённой нагрузкой для выявления узких

мест, определения предельной пропускной способности и проверки соответствия требованиям к производительности.



1.2. Ключевые метрики производительности

Метрика	Описание	Хорошее значение
RPS (Requests Per Second)	Количество запросов в секунду	Зависит от системы
Latency (p50)	Медианное время ответа	< 100 мс (API), < 2 с (страница)
Latency (p95)	95-й перцентиль	< 200 мс (API)
Latency (p99)	99-й перцентиль	< 500 мс (API)

Метрика	Описание	Хорошее значение
Error Rate	Процент ошибочных ответов	< 0.1%
Throughput	Объём обработанных данных	Зависит от системы

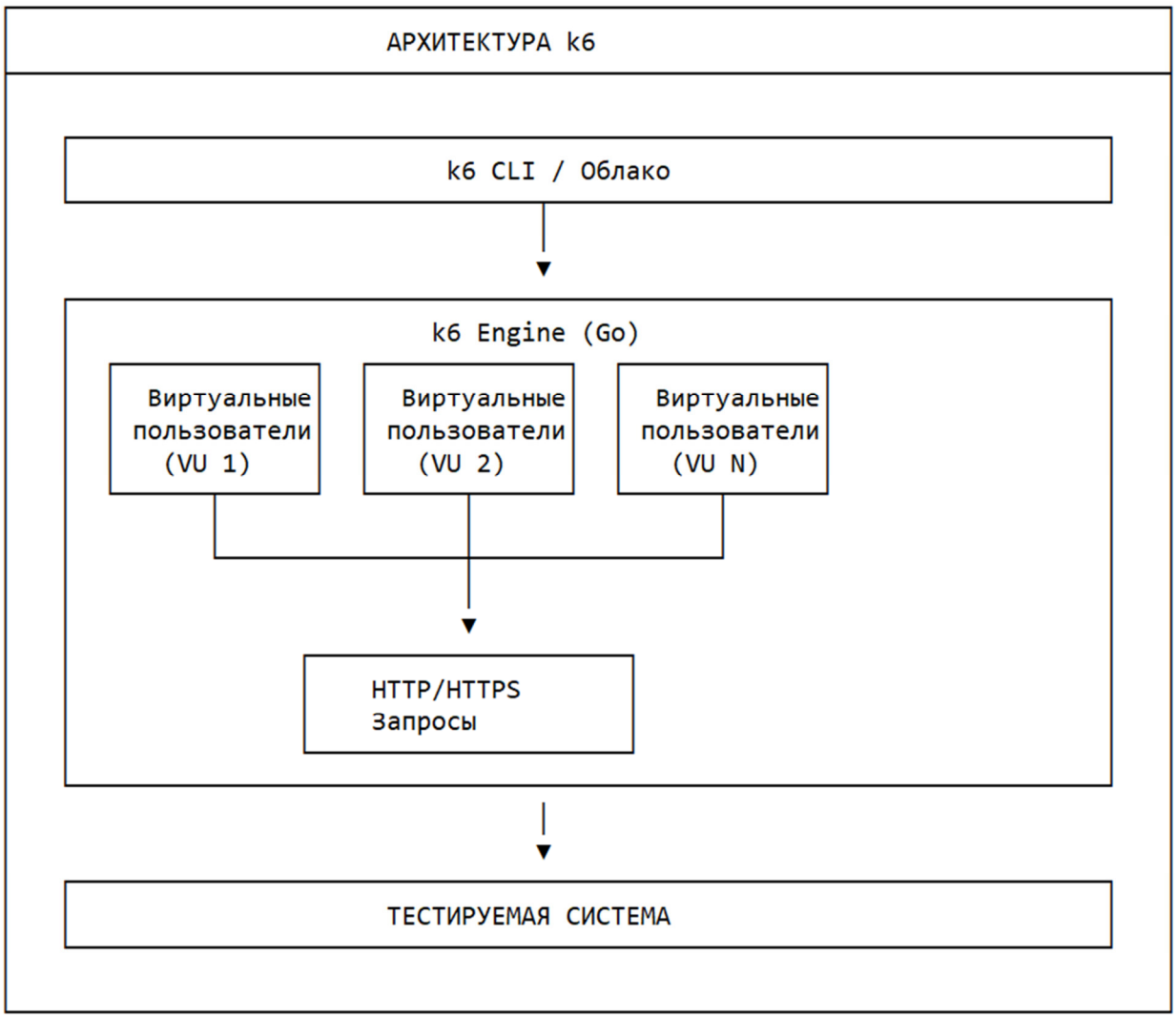
1.3. Обзор k6

k6 - это инструмент нагрузочного тестирования с открытым исходным кодом, разработанный компанией Grafana Labs.

Ключевые особенности:

Характеристика	Описание
Сценарии на JavaScript	Тесты пишутся на JS/ES6
Высокая производительность	Написан на Go, один экземпляр генерирует десятки тысяч RPS
Интеграция с Grafana	Встроенная поддержка Prometheus, Grafana Cloud
CI/CD	Легко интегрируется в пайплайны (GitHub Actions, GitLab CI)
Checks и Thresholds	Встроенные проверки для автоматического контроля SLA
Экспорт результатов	JSON, CSV, Prometheus, InfluxDB

Архитектура k6:



1.4. Основные концепции k6

Виртуальные пользователи (VU — Virtual Users):

```
import http from 'k6/http';

export const options = {
    vus: 10,        // 10 виртуальных пользователей
    duration: '30s', // Тест длится 30 секунд
};

export default function () {
    // Каждый VU выполняет этот код
    http.get('https://test-api.example.com/health');
}
```

Сценарии (Scenarios):

```
export const options = {
    scenarios: {
        // Сценарий 1: постоянная нагрузка
        constant_load: {
            executor: 'constant-vus',
            vus: 50,
            duration: '5m',
        },
        // Сценарий 2: растущая нагрузка (ramp-up)
        ramping_load: {
            executor: 'ramping-vus',
            startVUs: 0,
            stages: [
                { duration: '2m', target: 100 }, // 2 минуты до 100 VU
                { duration: '5m', target: 100 }, // 5 минут на 100 VU
                { duration: '2m', target: 0 },   // 2 минуты снижение до 0
            ],
        },
    },
};
```

Thresholds (пороговые значения):

```
export const options = {
    thresholds: {
        // p95 latency < 500 мс
        http_req_duration: ['p(95) < 500'],
        // error rate < 1%
        http_req_failed: ['rate < 0.01'],
    },
};
```

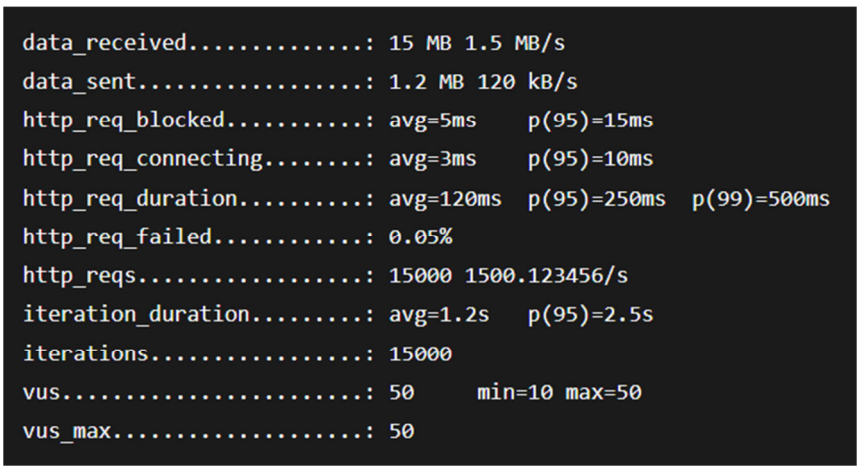
1.5. Типы сценариев k6

Executor	Описание	Когда использовать
constant-vus	Постоянное число VU	Базовый нагрузочный тест
ramping-vus	VU растут/падают по этапам	Стресс-тест, поиск порога

Executor	Описание	Когда использовать
shared-iterations	Общее число итераций	Тесты с фиксированным объёмом
per-vu-iterations	Каждый VU выполняет N итераций	Измерение производительности
constant-arrival-rate	Фиксированное число RPS	Тесты с контролем RPS

1.6. Анализ результатов

k6 выдаёт результаты в следующем формате:



2. ПРИМЕРЫ ПРАКТИЧЕСКОГО ВЫПОЛНЕНИЯ

2.1. Структура проекта

k6-tests/

├── test_smoke.js # Дымовой тест

├── test_load.js # Нагрузочный тест

├── test_stress.js # Стресс-тест (ramp-up)

├── test_spike.js # Spike-тест (резкий скачок)

├── test_soak.js # Soak-тест (длительная нагрузка)

```
├── test_api.js          # API с проверками и порогами
├── package.json         # (опционально) для Node.js проектов
├── docker-compose.yml   # Prometheus + Grafana
├── thresholds.json      # Пороговые значения
└── results/             # Папка с результатами
```

2.2. Установка k6

```
# Debian/Ubuntu
sudo apt-key adv --keyserver hkp://keyserver.ubuntu.com:80 --recv-keys
C5AD17C747E3415A3642D57D77C6C491D6AC1D69
echo "deb https://dl.k6.io/deb stable main" | sudo tee
/etc/apt/sources.list.d/k6.list
sudo apt-get update
sudo apt-get install k6

# macOS
brew install k6

# Windows (choco)
choco install k6
```

Docker:

```
docker run --rm -i grafana/k6 run - <test.js
```

2.3. Пример 1: Дымовой тест (Smoke Test)

```
// test_smoke.js
import http from 'k6/http';
import { check, sleep } from 'k6';

export const options = {
    vus: 1,          // 1 виртуальный пользователь
    duration: '30s', // 30 секунд
};

export default function () {
    // 1. Проверка health
    const healthRes = http.get('http://localhost:8000/health');
    check(healthRes, {
        'health status is 200': (r) => r.status === 200,
        'health response has ok': (r) => r.json('status') === 'ok',
    });

    // 2. Получение списка продуктов
    const productsRes = http.get('http://localhost:8000/api/products');
```

```
    check(productsRes, {
        'products status is 200': (r) => r.status === 200,
        'products array not empty': (r) => r.json().length > 0,
    });

    // 3. Получение конкретного продукта
    const productRes = http.get('http://localhost:8000/api/products/1');
    check(productRes, {
        'product status is 200': (r) => r.status === 200,
        'product id is 1': (r) => r.json('id') === 1,
    });

    sleep(1);
}
```

2.4. Пример 2: Нагрузочный тест (Load Test)

```
// test_load.js
import http from 'k6/http';
import { check, sleep } from 'k6';

export const options = {
    stages: [
        { duration: '1m', target: 50 },  // Разогрев: 1 минута до 50 VU
        { duration: '3m', target: 50 },  // Стабильная нагрузка: 50 VU в
течение 3 минут
        { duration: '1m', target: 0 },   // Охлаждение: снижение до 0
    ],
    thresholds: {
        http_req_duration: ['p(95) < 500'],  // 95% запросов < 500 мс
        http_req_failed: ['rate < 0.01'],    // Ошибки < 1%
    },
    tags: { scenario: 'load_test' },
};

export default function () {
    const url = 'http://localhost:8000/api/products';

    // GET запрос к каталогу
    const res = http.get(url, {
        headers: { 'Content-Type': 'application/json' },
        tags: { endpoint: 'catalog' },
    });

    // Проверка ответа
    check(res, {
        'status is 200': (r) => r.status === 200,
        'response time < 500ms': (r) => r.timings.duration < 500,
```

```
    });

    // Имитация времени обдумывания (think time)
    sleep(Math.random() * 2);
}
```

2.5. Пример 3: Стресс-тест с этапами нагрузки

```
// test_stress.js
import http from 'k6/http';
import { check, sleep } from 'k6';

export const options = {
    stages: [
        { duration: '2m', target: 100 },   // 0 → 100 VU за 2 минуты
        { duration: '5m', target: 100 },   // 100 VU в течение 5 минут
        { duration: '2m', target: 200 },   // 100 → 200 VU за 2 минуты
        { duration: '5m', target: 200 },   // 200 VU в течение 5 минут
        { duration: '2m', target: 500 },   // 200 → 500 VU за 2 минуты
        { duration: '5m', target: 500 },   // 500 VU в течение 5 минут
        { duration: '2m', target: 1000 },  // 500 → 1000 VU за 2 минуты
        { duration: '5m', target: 1000 },  // 1000 VU в течение 5 минут
        { duration: '2m', target: 0 },     // Снижение до 0
    ],
    thresholds: {
        http_req_duration: ['p(99) < 1000'],  // 99% запросов < 1 сек
        http_req_failed: ['rate < 0.05'],     // Ошибки < 5%
    },
};

export default function () {
    const payload = JSON.stringify({
        user_id: Math.floor(Math.random() * 1000),
        product_id: Math.floor(Math.random() * 100),
        quantity: Math.floor(Math.random() * 5) + 1,
    });

    const params = {
        headers: { 'Content-Type': 'application/json' },
    };

    // POST запрос на создание заказа
    const res = http.post('http://localhost:8000/api/orders', payload,
params);

    check(res, {
        'order created (201)': (r) => r.status === 201,
        'order created with id': (r) => r.json('order_id') !== undefined,
    });
```

```
    sleep(0.5);
}
```

2.6. Пример 4: Spike-тест (резкий скачок нагрузки)

```
// test_spike.js
import http from 'k6/http';
import { check } from 'k6';

export const options = {
    scenarios: {
        spike: {
            executor: 'ramping-vus',
            startVUs: 0,
            stages: [
                { duration: '30s', target: 10 },   // Нормальная нагрузка
                { duration: '30s', target: 10 },   // Фоновая нагрузка
                { duration: '10s', target: 500 },  // РЕЗКИЙ СКАЧОК (500 VU)
                { duration: '30s', target: 10 },   // Возврат к норме
                { duration: '30s', target: 0 },    // Охлаждение
            ],
        },
    },
    thresholds: {
        http_req_duration: ['p(95) < 1000'],
        http_req_failed: ['rate < 0.02'],
    },
};

export default function () {
    // Имитация тяжёлой операции
    const res = http.get('http://localhost:8000/api/search?q=test');

    check(res, {
        'search status is 200': (r) => r.status === 200,
    });
}
```

2.7. Пример 5: Soak-тест (длительная нагрузка)

```
// test_soak.js
import http from 'k6/http';
import { check, sleep } from 'k6';

export const options = {
    stages: [
        { duration: '5m', target: 100 },   // Разогрев
        { duration: '8h', target: 100 },   // 8 часов стабильной нагрузки
        { duration: '5m', target: 0 },     // Охлаждение
```

```
    ],
    thresholds: {
        http_req_duration: ['p(99) < 800'],
        http_req_failed: ['rate < 0.001'],   // Ошибки < 0.1%
        iterations: ['count > 100000'],
    },
};

export default function () {
    const res = http.get('http://localhost:8000/api/products');

    check(res, {
        'status is 200': (r) => r.status === 200,
    });

    // 2-5 секунд между запросами
    sleep(Math.random() * 3 + 2);
}
```

2.8. Пример 6: Тестирование API с данными из CSV

```
// test_csv_data.js
import http from 'k6/http';
import { check, sleep } from 'k6';
import { SharedArray } from 'k6/data';

// Загрузка тестовых данных из CSV
const users = new SharedArray('users', function () {
    const data = open('./users.csv');
    return data.split('\n').slice(1).map(line => {
        const [id, name, email] = line.split(',');
        return { id, name, email };
    });
});

export const options = {
    vus: 10,
    duration: '1m',
};

export default function () {
    const user = users[Math.floor(Math.random() * users.length)];

    const payload = JSON.stringify({
        user_id: user.id,
        name: user.name,
        email: user.email,
    });
```

```
    const res = http.post('http://localhost:8000/api/users', payload, {
        headers: { 'Content-Type': 'application/json' },
    });

    check(res, {
        'user created': (r) => r.status === 201,
    });

    sleep(1);
}
```

2.9. Docker Compose для мониторинга (k6 + Prometheus + Grafana)

```yaml
# docker-compose.yml
version: '3.8'

services:
  prometheus:
    image: prom/prometheus:latest
    ports:
      - "9090:9090"
    volumes:
      - ./prometheus.yml:/etc/prometheus/prometheus.yml
      - prometheus_data:/prometheus
    command:
      - '--config.file=/etc/prometheus/prometheus.yml'
      - '--storage.tsdb.path=/prometheus'

  grafana:
    image: grafana/grafana:latest
    ports:
      - "3000:3000"
    environment:
      - GF_SECURITY_ADMIN_PASSWORD=admin
    volumes:
      - grafana_data:/var/lib/grafana
      - ./dashboards/:/etc/grafana/provisioning/dashboards/

  k6:
    image: grafana/k6:latest
    volumes:
      - ./tests:/scripts
    environment:
      - K6_PROMETHEUS_RW_SERVER_URL=http://prometheus:9090/api/v1/write
    command: run /scripts/test_load.js

volumes:
```

```
  prometheus_data:
  grafana_data:
```

2.10. Интеграция с CI/CD (GitLab CI)

```
# .gitlab-ci.yml
performance-test:
  stage: test
  image: grafana/k6:latest
  script:
    - k6 run --out json=results.json test_load.js
  artifacts:
    paths:
      - results.json
    expire_in: 1 week
  only:
    - main
    - merge_requests
  when: manual  # Ручной запуск для performance тестов
```

2.11. Интеграция с GitHub Actions

```
# .github/workflows/performance.yml
name: Performance Tests

on:
  workflow_dispatch:  # Ручной запуск

jobs:
  k6-test:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4

      - name: Run k6 test
        uses: grafana/k6-action@v0.3.0
        with:
          filename: test_load.js
          flags: --out json=results.json

      - name: Upload results
        uses: actions/upload-artifact@v4
        with:
          name: k6-results
          path: results.json
```

МЕТОДИЧЕСКИЕ УКАЗАНИЯ ДЛЯ ВЫПОЛНЕНИЯ ПРАКТИЧЕСКОЙ ЧАСТИ РАБОТЫ

Задачи работы:

1. **Установка и настройка k6:**
   - Установить k6 (локально или через Docker)
   - Настроить целевое окружение для тестирования
2. **Разработка сценариев тестирования:**
   - Smoke test - минимальная проверка работоспособности
   - Load test - проверка поведения при нормальной нагрузке
   - Stress test - поиск точки насыщения
   - Spike test - реакция на резкий скачок
   - Soak **test** - выявление проблем с памятью/соединениями
3. **Настройка Thresholds (порогов):**
   - Установить пороги для p95 latency
   - Установить пороги для error rate
   - Настроить автоматический провал теста при нарушении
4. **Анализ результатов:**
   - Экспорт результатов в JSON
   - Построение графиков в Grafana (через Prometheus)
   - Сравнение с предыдущими запусками
5. **Интеграция в CI/CD:**
   - Добавить job в GitLab CI / GitHub Actions
   - Автоматический запуск нагрузочных тестов при релизе
6. **Оформление отчёта:** скриншоты, выводы, рекомендации по оптимизации

3. ВАРИАНТЫ ЗАДАНИЙ

3.1. Базовый уровень (15 вариантов)

Требования: Smoke Test + Load Test (постоянная нагрузка)

	Тестируемый эндпоинт	VU (макс)	Порог p95
	GET /api/products	50	< 200 мс
	GET /api/products/{id}	50	< 100 мс
	POST /api/orders	30	< 300 мс
	GET /api/orders/{id}	50	< 150 мс
	POST /api/users	30	< 200 мс
	GET /api/search?q=test	50	< 300 мс
	POST /api/login	20	< 500 мс
	GET /api/health	100	< 50 мс
	GET /api/restaurants	50	< 200 мс
0	GET /api/restaurants/{id}	50	< 150 мс
1	POST /api/cart/items	30	< 200 мс
2	DELETE /api/cart/items/{id}	30	< 150 мс
3	POST /api/payments	20	< 500 мс

	Тестируемый эндпоинт	VU (макс)	Порог p95
4	GET /api/delivery/status/{id}	50	< 100 мс
5	GET /api/notifications	40	< 200 мс

3.2. Средний уровень (15 вариантов)

Требования: Load Test + Stress Test (ramp-up) + Thresholds

№	Сложный сценарий	RPS целевой	RPS пик (stress)	Пороги
1	Просмотр каталога + добавление в корзину	500	2000	p95<300, err<1%
2	Оформление заказа + платёж (через мок)	200	800	p95<1000, err<0.5%
3	Поиск + фильтрация + сортировка	400	1500	p95<500, err<1%
4	Регистрация + аутентификация	100	400	p95<600, err<0.5%
5	Админ-панель (списки, отчёты)	100	400	p95<800, err<1%
6	API Gateway с rate limiting	1000	5000	p95<200, err<2%
7	Загрузка изображений	50	200	p95<2000, err<1%

№	Сложный сценарий	RPS целевой	RPS пик (stress)	Пороги
	(multipart)			
8	WebSocket соединения	500 conn	2000 conn	handshake<200ms, err<1%
9	GraphQL запросы (query)	300	1000	p95<400, err<0.5%
10	GraphQL мутации (mutation)	100	400	p95<500, err<0.5%
11	gRPC unary вызовы	1000	5000	p95<100, err<0.1%
12	gRPC server streaming	200	800	p95<200, err<0.5%
13	Elasticsearch поиск	500	2000	p95<150, err<0.5%
14	Kafka producer (через прокси)	5000	20000	p95<50, err<0.1%
15	Redis команды (через API)	10000	50000	p95<10, err<0.01%

3.3. Продвинутый уровень (15 вариантов)

Требования: все 5 типов тестов + интеграция с Prometheus + Grafana + CI/CD

№	Тип системы	Сценарий тестирования	Метрики в Grafana	CI/CD (branch)
1	E-commerce полный	Поток пользователя (home → каталог → товар → корзина → заказ)	RPS, latency, hit rate, error rate	GitLab CI
2	Доставка	Создание заказа → поиск курьера → трекинг	p50/p95/p99, saturation	GitHub Actions
3	Социальная сеть	Лента постов → комментарии → лайки	CPU, memory, network	Jenkins
4	Стриминг	Загрузка видео → стриминг фрагментов	throughput, bitrate, start time	GitLab CI
5	Бронирование (билеты)	Поиск → бронирование → оплата	transaction rate, lock time	GitHub Actions

№	Тип системы	Сценарий тестирования	Метрики в Grafana	CI/CD (branch)
6	Fintech	API платежей → вебхуки → подтверждение	idempotency, retry rate	GitLab CI
7	Медицина (телемедицина)	Запись к врачу → видео-звонок → рецепт	jitter, packet loss, latency	GitHub Actions
8	IoT платформа	Приём телеметрии (1k devices) → агрегация → алерты	ingestion rate, lag	Jenkins
9	Логистика (WMS)	Приём заказа → резервирование → отгрузка	concurrency, lock wait	GitLab CI
10	Чат (WebSocket)	Отправка сообщений → чтение → уведомления	message latency, fanout delay	GitHub Actions

№	Тип системы	Сценарий тестирования	Метрики в Grafana	CI/CD (branch)
11	Аналитика (ClickHouse)	Вставка событий → агрегация → запрос	insert rate, query duration, memory	GitLab CI
12	CDN	Запрос статики (CSS/JS/images)	cache hit ratio, origin load	GitHub Actions
13	Кэш (Redis)	SET/GET операции + TTL инвалидация	hit/miss, eviction rate, memory	Jenkins
14	База данных (PostgreSQL)	SELECT/INSERT/UPDATE сложные транзакции	deadlocks, locks, tx duration	GitLab CI
15	Система очередей (Kafka)	Производитель → consumer → commit	lag, throughput, rebalance time	GitHub Actions

4. КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Какие существуют типы нагрузочного тестирования? В чём различия?
2. Что такое виртуальные пользователи (VU) в k6?
3. Как настроить постепенное увеличение нагрузки (ramp-up) в k6?
4. Что такое thresholds в k6 и для чего они нужны?
5. Как в k6 проверить, что p95 latency меньше заданного значения?
6. В чём разница между constant-vus и ramping-vus?
7. Как в k6 передать тестовые данные из CSV-файла?
8. Как интегрировать k6 в GitLab CI / GitHub Actions?
9. Какие метрики по умолчанию собирает k6?
10. Как экспортировать результаты k6 в Prometheus?
11. Какой тип теста лучше всего подходит для выявления утечек памяти?
12. Какой тип теста используется для проверки реакции на внезапный всплеск трафика (например, Чёрная пятница)?
13. Как отличить проблему сети от проблем приложения по результатам k6?
14. Что такое «think time» и зачем он нужен?
15. Как настроить в k6 разные сценарии для разных эндпоинтов (например, read-heavy и write-heavy)?

5. ТРЕБОВАНИЯ К ОТЧЕТУ

Отчёт должен содержать:

1. **Титульный лист**
2. **Цель работы**
3. **Теоретическая часть:** типы тестирования, метрики, концепции k6
4. **Практическая часть:**
   - Листинг всех скриптов (smoke, load, stress, spike, soak)
   - Листинг конфигурации thresholds

- Инструкция по запуску тестов

5. **Результаты выполнения:**
   - Скриншоты результатов smoke теста
   - Таблица с метриками для всех типов тестов
   - Графики зависимости latency от нагрузки
   - Точка насыщения (из stress-теста)
6. **Анализ результатов:**
   - Соответствие SLA (пороги пройдены/не пройдены)
   - Выявленные узкие места (bottlenecks)
   - Рекомендации по оптимизации
7. **Вывод**
8. **Ответы на контрольные вопросы**

6. КРИТЕРИИ ОЦЕНИВАНИЯ

Критерий	**Базовый (10)**	**Средний (14)**	**Продвинутый (18)**
k6 установлен и работает	2	1	1
Smoke test (скрипт + результат)	2	1	1
Load test (скрипт + результат)	3	2	1
Stress test (ramp-up)	3	2	2
Spike test	—	3	2
Soak test	—	2	2
Thresholds настроены	—	2	2

Критерий	Базовый (10)	Средний (14)	Продвинутый (18)
Интеграция с Prometheus/Grafana	—	1	2
CI/CD интеграция	—	—	3
Качество отчёта	—	—	2
Максимум	**10**	**14**	**18**

Шкала перевода в оценку:

- **5 (отлично):** 90%+
- **4 (хорошо):** 75–89%
- **3 (удовлетворительно):** 60–74%
- **2 (неудовлетворительно):** <60%

7. ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ

Программа	Назначение	Доступность в РФ
k6 (grafana/k6)	Инструмент нагрузочного тестирования	(Open Source)
Docker + Docker Compose	Контейнеризация	+
Prometheus (опционально)	Сбор метрик	+
Grafana (опционально)	Визуализация	+

Программа	Назначение	Доступность в РФ
	результатов	
Python (опционально)	Подготовка тестовых данных	+
GitLab CI / GitHub Actions	CI/CD интеграция	+

Лабораторная работа №15

Документирование архитектуры с использованием arc42

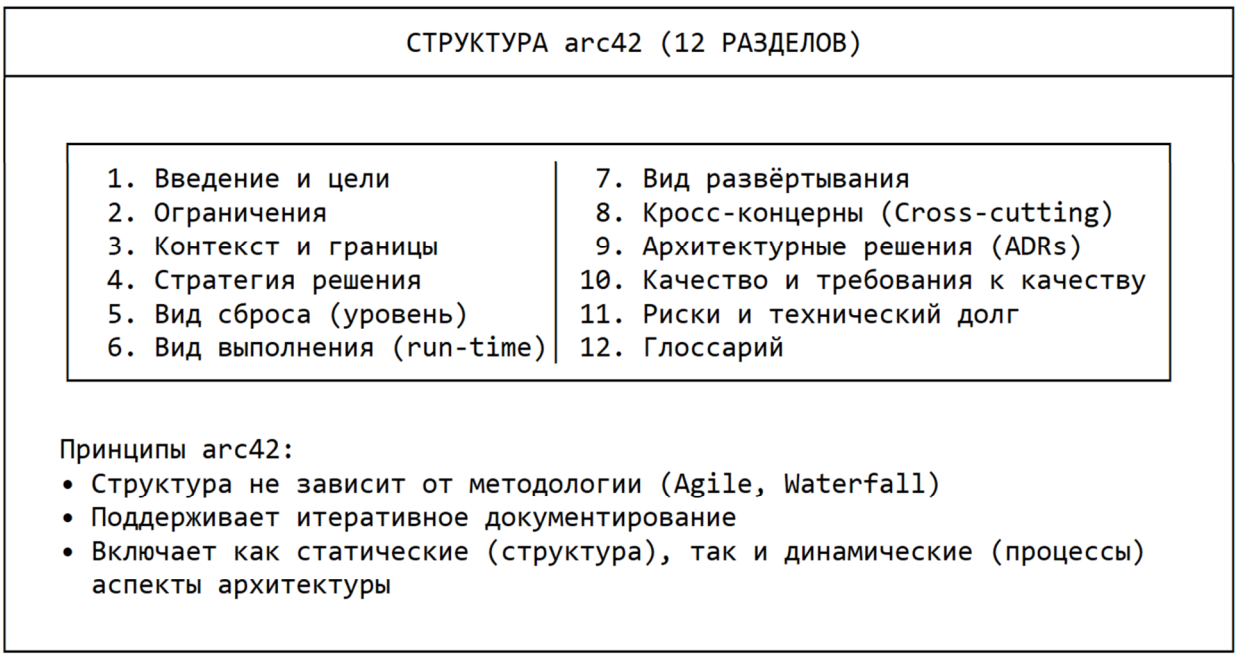
Цель работы: Сформировать практические навыки структурированного документирования архитектуры программных систем с использованием методологии arc42, включая заполнение всех 12 разделов шаблона, создание архитектурных диаграмм и документирование архитектурных решений в формате ADR (Architecture Decision Record).

Формируемые компетенции: ПК-5

1. ТЕОРЕТИЧЕСКОЕ ОБОСНОВАНИЕ

1.1. Что такое arc42

arc42 -это шаблон (template) для документирования архитектуры программных систем, разработанный на основе многолетнего опыта промышленной разработки. arc42 предоставляет структурированный подход к описанию архитектуры, охватывающий как технические, так и организационные аспекты.



1.2. Назначение каждого раздела

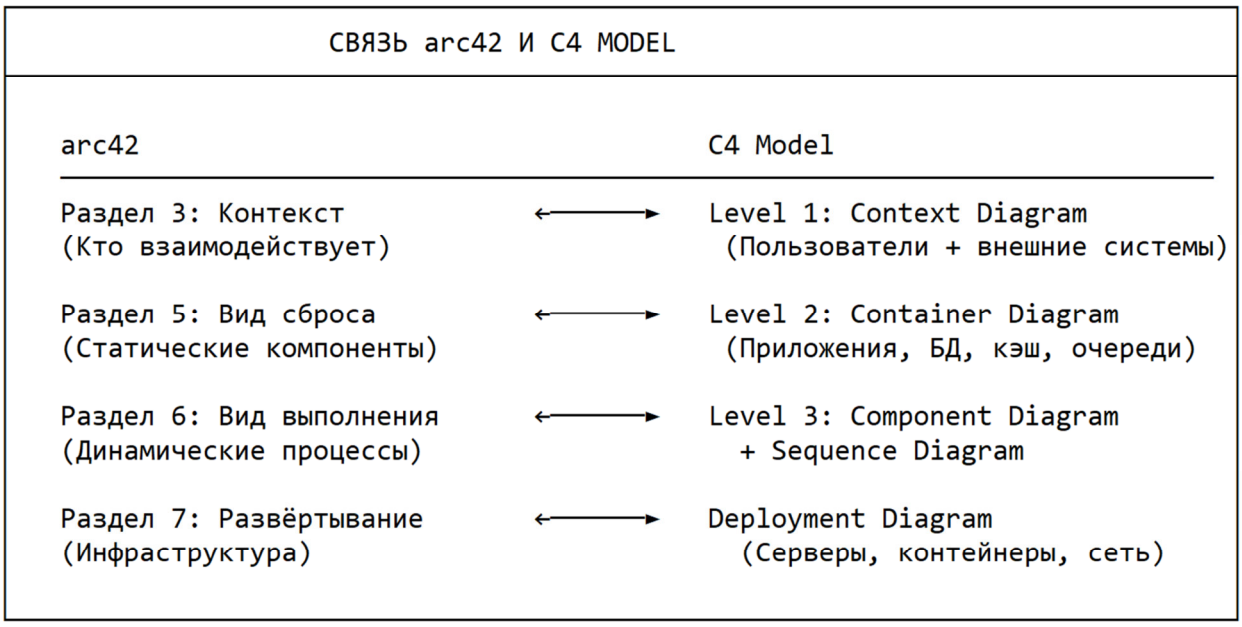
Раздел	Назначение	Ключевые вопросы
1. Введение и цели	Описание целей системы, заинтересованных сторон, требований	Зачем нужна система? Кто её будет использовать?
2. Ограничения	Технические, организационные, регуляторные ограничения	Какие технологии обязательны? Есть ли бюджетные ограничения?
3. Контекст и границы	Взаимодействие системы с внешним миром	С какими системами взаимодействует? Где границы системы?
4. Стратегия решения	Ключевые архитектурные решения и обоснования	Какие паттерны выбраны? Почему?

Раздел	Назначение	Ключевые вопросы
5. Вид сброса	Статическая структура системы (компоненты, модули)	Из каких компонентов состоит система?
6. Вид выполнения	Динамическая структура (процессы, взаимодействия)	Как компоненты общаются в рантайме?
7. Вид развёртывания	Инфраструктура и развёртывание	Где и как запускается система?
8. Кросс-концерны	Сквозные концепции (логирование, безопасность, мониторинг)	Как реализованы общие сквозные функции?
9. Архитектурные решения	Документирование ADR (Architecture Decision Record)	Какие важные решения приняты и почему?
10. Качество	Требования к качеству (SLA, SLO, SLI)	Каковы количественные требования к качеству?
11. Риски	Архитектурные риски и способы их митигации	Какие риски есть? Как их снизить?
12. Глоссарий	Определение ключевых терминов	Что означают используемые термины?

1.3. Взаимосвязь arc42 и C4 model

arc42 и C4 model дополняют друг друга:

arc42 Раздел	C4 уровень	Инструмент
Раздел 3 (Контекст)	Level 1 (Context)	C4 диаграмма контекста
Раздел 5 (Вид сброса)	Level 2 (Containers)	C4 диаграмма контейнеров
Раздел 6 (Вид выполнения)	Level 3 (Components) + диаграммы последовательности	C4 диаграмма компонентов + UML Sequence
Раздел 7 (Развёртывание)	Дополнительные диаграммы развёртывания	Диаграмма развёртывания UML



1.4. Architecture Decision Record (ADR)

ADR - это документ, фиксирующий важное архитектурное решение. ADR является частью раздела 9 arc42.

Структура ADR (шаблон Мэддисона):

markdown

 # ADR-001: Выбор базы данных

Статус

[Предложено | Принято | Устарело | Заменено]

Контекст

Описание проблемы, требований и ограничений, которые привели к необходимости принятия решения.

Решение

Описание выбранного подхода или технологии.

Обоснование

Почему выбрано именно это решение. Сравнение альтернатив.

Последствия

- ****Положительные:**** что улучшится
- ****Отрицательные:**** какие возникают компромиссы
- ****Нейтральные:**** что остаётся неизменным

1.5. Лучшие практики документирования

Практика	Описание
Документация как код	Хранить документацию в репозитории рядом с кодом
Актуальность	Обновлять документацию при каждом изменении архитектуры

Практика	Описание
Диаграммы из кода	Использовать PlantUML/Mermaid для диаграмм (версионирование)
Краткость	Документировать только то, что меняется редко
Аудитория	Писать для разных аудиторий (разделы 1-3 для менеджмента, 4-9 для разработчиков)
ADR	Фиксировать важные решения, а не всё подряд

2. ПРИМЕРЫ ПРАКТИЧЕСКОГО ВЫПОЛНЕНИЯ

2.1. Шаблон документа arc42 (заполненный пример)

Ниже представлен заполненный шаблон arc42 для системы онлайн-бронирования переговорных комнат.

Раздел 1: Введение и цели

markdown

 # 1. Введение и цели

1.1. Цели системы

Система онлайн-бронирования переговорных комнат предназначена для:

- Просмотра доступных комнат по времени и дате
- Бронирования комнат на определённый период
- Управления комнатами (CRUD) администраторами
- Отправки уведомлений о бронировании

1.2. Заинтересованные стороны

| **Сторона** | **Интересы** |

|---------|----------|

| Сотрудники | Удобный интерфейс, быстрый поиск, мгновенное подтверждение |

| Администраторы | Простое управление комнатами, отмена бронирований |

| IT-отдел | Простота развёртывания, мониторинг, отказоустойчивость |

| Руководство | Эффективное использование офисных ресурсов |

1.3. Ключевые требования

- Доступность: 99.9% (≤ 8.76 часов даунтайма в год)
- Производительность: p95 latency ≤ 500 мс при 500 RPS
- Безопасность: аутентификация через корпоративный SSO

Раздел 2: Ограничения

markdown

2. Ограничения

2.1. Технические ограничения

- Использовать PostgreSQL в качестве основной БД
- Развёртывание в Kubernetes (on-premise)
- Аутентификация через корпоративный Keycloak

2.2. Организационные ограничения

- Команда разработки: 5 человек
- Бюджет на инфраструктуру: ограничен
- Релизный цикл: 2 недели

2.3. Регуляторные ограничения

- ФЗ-152 (персональные данные)
- Политика безопасности компании

Раздел 3: Контекст и границы

markdown

 # 3. Контекст и границы

3.1. Бизнес-контекст

plantuml

@startuml

!include https://raw.githubusercontent.com/plantuml-stdlib/C4-PlantUML/master/C4_Context.puml

title Контекстная диаграмма — Система бронирования переговорных

Person(employee, "Сотрудник", "Бронирует комнаты")

Person(admin, "Администратор", "Управляет комнатами")

System(booking, "Система бронирования", "Управление бронированиями")

System_Ext(sso, "Keycloak", "Аутентификация")

System_Ext(email, "Email-сервер", "Уведомления")

Rel(employee, booking, "Использует", "HTTPS")

Rel(admin, booking, "Использует", "HTTPS")

Rel(booking, sso, "Проверяет токен", "OAuth2")

Rel(booking, email, "Отправляет уведомления", "SMTP")

@enduml

Раздел 4: Стратегия решения

markdown

 # 4. Стратегия решения

4.1. Ключевые архитектурные решения

| **Решение** | **Обоснование** |

|---------|-------------|

| Микросервисная архитектура | Независимое масштабирование сервиса бронирований |

| API Gateway (Kong) | Единая точка входа, аутентификация, rate limiting |

| PostgreSQL | ACID-транзакции для бронирований |

| Redis | Кэширование доступности комнат |

| RabbitMQ | Асинхронная отправка уведомлений |

4.2. Принципы

1. **Stateless сервисы** — состояние вынесено в БД и кэш

2. **Отказоустойчивость** — Circuit Breaker, Retry, Timeout

3. **Наблюдаемость** — метрики (Prometheus), логи (ELK), трассировка (Jaeger)

Раздел 5: Вид сброса (уровень контейнеров)

markdown

5. Вид сброса

5.1. Диаграмма контейнеров

plantuml

@startuml

!include https://raw.githubusercontent.com/plantuml-stdlib/C4-PlantUML/master/C4_Container.puml

title Диаграмма контейнеров — Система бронирования

Person(employee, "Сотрудник")

System_Boundary(booking, "Система бронирования") {
Container(web, "SPA", "React", "UI для бронирования")
Container(gateway, "API Gateway", "Kong", "Маршрутизация + JWT validation")
Container(api, "API", "Python/FastAPI", "Бизнес-логика")
ContainerDb(db, "БД", "PostgreSQL", "Хранилище")
Container(cache, "Кэш", "Redis", "Кэширование доступности")
Container(queue, "Очередь", "RabbitMQ", "Асинхронные задачи")
Container(worker, "Worker", "Python", "Обработка очереди")
}

System_Ext(email, "Email-сервер")

Rel(employee, web, "Использует", "HTTPS")
Rel(web, gateway, "Вызывает API", "HTTPS")
Rel(gateway, api, "Перенаправляет", "HTTP")
Rel(api, db, "Читает/пишет", "SQL")
Rel(api, cache, "Читает", "Redis protocol")
Rel(api, queue, "Публикует", "AMQP")
Rel(worker, queue, "Потребляет", "AMQP")
Rel(worker, email, "Отправляет", "SMTP")
@enduml

Раздел 6: Вид выполнения

markdown

 # 6. Вид выполнения

6.1. Диаграмма последовательности (создание бронирования)

plantuml

@startuml

actor "Сотрудник" as User

```
participant "SPA" as SPA
participant "API Gateway" as Gateway
participant "API Сервис" as API
database "PostgreSQL" as DB
participant "RabbitMQ" as MQ
participant "Worker" as Worker
participant "Email-сервер" as Email

User -> SPA: Заполняет форму бронирования
SPA -> Gateway: POST /api/bookings
Gateway -> Gateway: Проверка JWT
Gateway -> API: POST /api/bookings

API -> DB: Начать транзакцию
API -> DB: Проверить доступность комнаты
alt Комната свободна
    API -> DB: Создать бронирование
    API -> DB: Закоммитить транзакцию
    API -> MQ: Опубликовать событие BookingCreated
    API --> Gateway: 201 Created
    Gateway --> SPA: 201 Created
    SPA --> User: "Бронирование создано"

    MQ -> Worker: Доставить событие
    Worker -> Email: Отправить уведомление
    Worker --> MQ: ACK
else Комната занята
    API -> DB: Откат транзакции
    API --> Gateway: 409 Conflict
    Gateway --> SPA: 409 Conflict
```

SPA --> User: "Комната уже занята"
end
@enduml

Раздел 7: Вид развёртывания

markdown

 # 7. Вид развёртывания

7.1. Инфраструктура

- Оркестрация: Kubernetes (3 узла)
- База данных: PostgreSQL Operator (Patroni)
- Кэш: Redis Sentinel (3 узла)
- Очередь: RabbitMQ Cluster (3 узла)
- Мониторинг: Prometheus + Grafana

7.2. Диаграмма развёртывания

plantuml

@startuml
node "K8s Cluster" {
  node "Node 1" {
    container "Pod: API Gateway"
    container "Pod: API"
    container "Pod: Worker"
  }
  node "Node 2" {
    container "Pod: API"
    container "Pod: Redis Slave"
  }
  node "Node 3" {
    container "Pod: API"
    container "Pod: Redis Slave"

```
    }

    database "PostgreSQL Cluster" {
        file "Master"
        file "Replica 1"
        file "Replica 2"
    }

    database "RabbitMQ Cluster" {
        file "Node 1"
        file "Node 2"
        file "Node 3"
    }
}
@enduml
```

Раздел 8: Кросс-концерны

```
markdown
# 8. Кросс-концерны (Cross-cutting concepts)

## 8.1. Логирование
- Уровни: ERROR, WARN, INFO, DEBUG
- Формат: JSON (структурированные логи)
- Агрегация: ELK Stack (Elasticsearch, Logstash, Kibana)
- Correlation ID: передаётся через все сервисы

## 8.2. Мониторинг
- Метрики: Prometheus (сбор каждые 15 секунд)
- Дашборды: Grafana
- Алерты: Alertmanager (PagerDuty)
```

8.3. Безопасность

- Аутентификация: JWT (через Keycloak)
- Авторизация: RBAC (роли: user, admin)
- Шифрование: TLS 1.3 для всех сервисов
- Secrets: Kubernetes Secrets (зашифрованы etcd)

8.4. Трассировка

- Инструмент: Jaeger
- Сэмплирование: 10% трафика
- Хранение: Elasticsearch

Раздел 9: Архитектурные решения (ADRs)

markdown

9. Архитектурные решения

ADR-001: Выбор базы данных

Статус

Принято

Контекст

Системе требуется хранить данные о комнатах, пользователях и бронированиях.

Ключевое требование — целостность данных при конкурентном бронировании.

Решение

Использовать PostgreSQL.

Обоснование

- ACID-транзакции гарантируют отсутствие двойных бронирований

- Хорошая поддержка конкурентного доступа (MVCC)

- Зрелый инструментарий и большое сообщество

Последствия

- ****Положительные:**** целостность данных, надёжность

- ****Отрицательные:**** сложнее горизонтальное масштабирование (шардирование)

ADR-002: Асинхронная отправка уведомлений

Статус

Принято

Контекст

При создании бронирования необходимо отправить email-уведомление.

Решение

Использовать RabbitMQ для асинхронной обработки.

Обоснование

- Не блокирует основной процесс создания бронирования

- Позволяет повторять неудачные отправки

- Лучшая отказоустойчивость

Последствия

- ****Положительные:**** высокая производительность записи, отказоустойчивость

- ****Отрицательные:**** eventual consistency (письмо может прийти с задержкой)

Раздел 10: Качество и требования к качеству

markdown

 # 10. Качество и требования к качеству

| **Атрибут** | **Требование** | **Метрика** |

|---|---|---|

| Производительность | Ответ API бронирования | p95 ≤ 500 мс |

| Доступность | Система доступна | 99.9% uptime |

| Масштабируемость | Поддержка пиковой нагрузки | 1000 RPS |

| Безопасность | Аутентификация | JWT, TLS 1.3 |

| Надёжность | Отказоустойчивость | RTO ≤ 5 мин, RPO ≤ 1 мин |

Раздел 11: Риски и технический долг

markdown

 # 11. Риски и технический долг

Риски

| **ID** | **Риск** | **Вероятность** | **Влияние** | **Митигация** |

|---|---|---|---|---|

| R-1 | PostgreSQL — single point of failure | Средняя | Высокое | Репликация + Patroni + автоматический failover |

| R-2 | Сложность отладки распределённой системы | Средняя | Средняя | Внедрить распределённую трассировку (Jaeger) |

| R-3 | Зависимость от внешнего Keycloak | Низкая | Высокое | Локальный fallback (резервная БД пользователей) |

Технический долг

- Отсутствие автоматических тестов отказоустойчивости (Chaos Engineering)

- Нет документации по API (OpenAPI)

- Мониторинг только базовых метрик

Раздел 12: Глоссарий

markdown

 # 12. Глоссарий

| **Термин** | **Определение** |

|--------|------------|

| ****Бронирование**** | Запись о том, что комната занята в определённый период |

| ****Комната**** | Переговорная комната с уникальным идентификатором, названием, вместимостью |

| ****ADR**** | Architecture Decision Record — документ, фиксирующий архитектурное решение |

| ****SPOF**** | Single Point Of Failure — единая точка отказа |

| ****RTO**** | Recovery Time Objective — целевое время восстановления |

| ****RPO**** | Recovery Point Objective — целевая точка восстановления (потеря данных) |

МЕТОДИЧЕСКИЕ УКАЗАНИЯ ДЛЯ ВЫПОЛНЕНИЯ ПРАКТИЧЕСКОЙ ЧАСТИ РАБОТЫ

Задачи работы:

1. **Изучение arc42:**
   - Ознакомиться со структурой из 12 разделов
   - Понять назначение каждого раздела
2. **Выбор системы для документирования:**
   - Использовать систему из предыдущих работ (e-commerce, доставка, бронирование)
   - Или выбрать новую систему по согласованию с преподавателем
3. **Заполнение разделов arc42:**
   - Раздел 1–3: используя бизнес-требования
   - Раздел 4–7: используя архитектурные решения

- Раздел 8: сквозные концепции (логирование, мониторинг, безопасность)
- Раздел 9: минимум 3 ADR
- Раздел 10–12: на основе анализа

4. **Создание диаграмм:**
   - Диаграмма контекста (C4 Level 1)
   - Диаграмма контейнеров (C4 Level 2)
   - Диаграмма последовательности / компонентов
   - Диаграмма развёртывания (опционально)
5. **Формирование итогового документа:**
   - Оформить в формате Markdown
   - Встроить диаграммы (PlantUML / Mermaid)
   - Убедиться в полноте и связности

3. ВАРИАНТЫ ЗАДАНИЙ

3.1. Базовый уровень (15 вариантов)

Требования: заполнить разделы 1–7 arc42, минимум 1 ADR

№	Система для документирования	Фокус
1	API Gateway для микросервисов	Маршрутизация, аутентификация
2	Сервис аутентификации (Keycloak alternative)	JWT, OAuth2, RBAC
3	Система логирования (ELK stack)	Сбор, хранение, визуализация логов
4	Платформа	Метрики, алерты, дашборды

№	Система для документирования	Фокус
	мониторинга (Prometheus + Grafana)	
5	Система очередей (RabbitMQ cluster)	Высокая доступность, DLQ
6	Кэширующий слой (Redis Sentinel)	Репликация, failover
7	База данных (PostgreSQL cluster)	Репликация, Patroni
8	CI/CD платформа (GitLab CI)	Пайплайны, параллелизация
9	Система бронирования (simple)	Конкурентный доступ
10	E-commerce каталог	Кэширование, полнотекстовый поиск
11	Корзина товаров (stateful)	Redis, сессии, таймауты
12	Платёжный шлюз (интеграция со Stripe)	Безопасность, идемпотентность
13	Сервис доставки	Геоданные, трекинг
14	Система уведомлений	Асинхронность, очереди

№	Система для документирования	Фокус
	(email/SMS/Push)	
15	Сервис аналитики (ClickHouse)	Агрегация, большие данные

3.2. Средний уровень (15 вариантов)

Требования: базовый + разделы 8–12, минимум 3 ADR

№	Система	Дополнительные требования	ADR темы
1	E-commerce полная платформа	Мониторинг, безопасность, CI/CD	БД, кэш, очередь
2	Платформа доставки еды	Геораспределение, трекинг	БД, API Gateway, fallback
3	Онлайн-кинотеатр	CDN, рекомендации	CDN, кэш, видеостриминг
4	Система управления складом (WMS)	Интеграция с 1С, роботы	БД, синхронизация, надёжность
5	Платформа такси	Геоданные, динамическое ценообразование	Балансировка, очереди, координация
6	Социальная сеть	Новостная лента,	Feed, кэш,

№	Система	Дополнительные требования	ADR темы
		подписки	хранение медиа
7	CRM для крупного бизнеса	BI-отчёты, интеграция с почтой	Безопасность, отчёты, интеграции
8	Медицинская платформа (телемедицина)	Видеозвонки, хранение записей	Безопасность, видео, хранилище
9	Банковское приложение	Платежи, Push, двухфакторная аутентификация	Безопасность, транзакции, уведомления
10	Финтех (инвестиции)	Реальное время, аналитика	Транзакции, стриминг, безопасность
11	IoT-платформа	Масштабирование (1M устройств), аналитика	Приём данных, агрегация, хранилище
12	Система госуслуг (портал)	Высокая нагрузка, доступность, аудит	Доступность, аудит, безопасность
13	Образовательная платформа (LMS)	Видео, тесты, сертификаты	Хранилище видео, масштабирование,

№	Система	Дополнительные требования	ADR темы
			безопасность
14	Платформа бронирования отелей	Интеграция с внешними системами	Богатые интеграции, конкурентность
15	Система управления строительством	Проекты, задачи, BIM	Хранение файлов, совместная работа

3.3. Продвинутый уровень (15 вариантов)

Требования: полный arc42 (12 разделов) + документация в репозитории (Git) + автоматическая генерация из маркдауна

№	Система	Сложность	Расширения arc42
1	Глобальная CDN (edge-caching)	Высокая	Многорегиональность, geo-DNS
2	Распределённая NoSQL БД (Cassandra-like)	Очень высокая	Шардирование, консистенция, repair
3	Service Mesh (Istio) для K8s	Высокая	Наблюдаемость, безопасность (mTLS)
4	Платформа обработки событий (Kafka)	Высокая	Stream processing, delivery guarantees

№	Система	Сложность	Расширения arc42
5	Система рекомендаций (ML-сервис)	Высокая	Обучение/инференс, feature store
6	Платформа видеозвонков (WebRTC)	Высокая	TURN/STUN серверы, сигнализация
7	Дронодоставка (управление дронами)	Очень высокая	Коммуникация, телеметрия, зарядка
8	Умный город (сенсоры + аналитика)	Очень высокая	Приём данных, сложные события
9	Фиатная платёжная система (real-time gross settlement)	Критическая	Безопасность, транзакции, аудит
10	Цифровая биржа (высокочастотный трейдинг)	Экстремальная	Низкая задержка, консистенция ордеров
11	Симуляция робототехники (Gazebo + ROS)	Высокая	Распределённая симуляция, координация
12	Платформа автономного	Экстремальная	Real-time сенсоры, безопасность

№	Система	Сложность	Расширения arc42
	вождения (симулятор)		
13	Операционная система облачной платформы (OpenStack-like)	Очень высокая	Оркестрация, сеть, хранилище
14	P2P-платформа (BitTorrent-like)	Высокая	Децентрализация, DHT
15	Блокчейн (уровень консенсуса)	Экстремальная	Распределённый консенсус, виртуальная машина

4. КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Что такое arc42 и каковы цели использования этого шаблона?
2. Перечислите 12 разделов arc42.
3. Как arc42 связан с C4 model?
4. Для чего нужен раздел «Контекст и границы»?
5. Что такое ADR (Architecture Decision Record)? Какие разделы он содержит?
6. В чём различие между статическим (view) и динамическим (runtime) аспектами архитектуры?
7. Какой раздел arc42 описывает требования к качеству?
8. Где в arc42 документируются архитектурные риски?
9. Как часто следует обновлять архитектурную документацию?
10. Почему раздел «Ограничения» важен для архитектуры?

11. Что такое «кросс-концерны» (cross-cutting concepts)? Приведите примеры.

12. Какую информацию содержит раздел «Глоссарий»?

13. Как документировать принятые архитектурные решения (ADR)?

14. Почему arc42 подходит как для Agile, так и для Waterfall?

15. Как проверить полноту архитектурной документации?

5. ТРЕБОВАНИЯ К ОТЧЕТУ

1. **Титульный лист**
2. **Цель работы**
3. **Теоретическая часть:** arc42, 12 разделов, ADR
4. **Практическая часть:** полный arc42-документ (12 разделов) в формате Markdown с встроенными диаграммами
5. **Скриншоты** (если диаграммы рендерятся отдельно)
6. **Вывод:** оценка полноты документации
7. **Ответы на контрольные вопросы**

6. КРИТЕРИИ ОЦЕНИВАНИЯ

Критерий	**Базовый (10)**	**Средний (14)**	**Продвинутый (18)**
Раздел 1–3 заполнены	2	1	1
Раздел 4–7 (диаграммы C4)	2	2	1
Раздел 8 (кросс-концерны)	—	2	2

Критерий	Базовый (10)	Средний (14)	Продвинутый (18)
Раздел 9 (ADR, мин 1/3/5)	2	2	2
Раздел 10–12	2	2	2
Полнота (12 разделов)	2	2	2
Диаграммы (PlantUML)	—	2	3
Git (документация как код)	—	—	2
Качество отчёта	—	1	3
Максимум	**10**	**14**	**18**

Шкала перевода в оценку:

- **5 (отлично):** 90%+
- **4 (хорошо):** 75–89%
- **3 (удовлетворительно):** 60–74%
- **2 (неудовлетворительно):** <60%

7. ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ

Программа	Назначение	Доступность в РФ
Markdown-редактор (VS Code, Obsidian, Typora)	Оформление документа	+

Программа	Назначение	Доступность в РФ
Git	Версионирование документации	+
PlantUML / Mermaid	Диаграммы в Markdown	+
Draw.io	Альтернативный инструмент диаграмм	+
GitHub / GitLab (опционально)	Хостинг документации	+

Лабораторная работа №16

Архитектурный кейс: e-commerce платформа

Цель работы: Провести полный цикл архитектурного проектирования высоконагруженной e-commerce платформы с обоснованием всех решений на основе требований к нагрузке, отказоустойчивости и масштабируемости, а также представить итоговое архитектурное решение в формате, готовом для защиты перед техническими стейкхолдерами.

Формируемые компетенции: ПК-5

1. ТЕОРЕТИЧЕСКОЕ ОБОСНОВАНИЕ

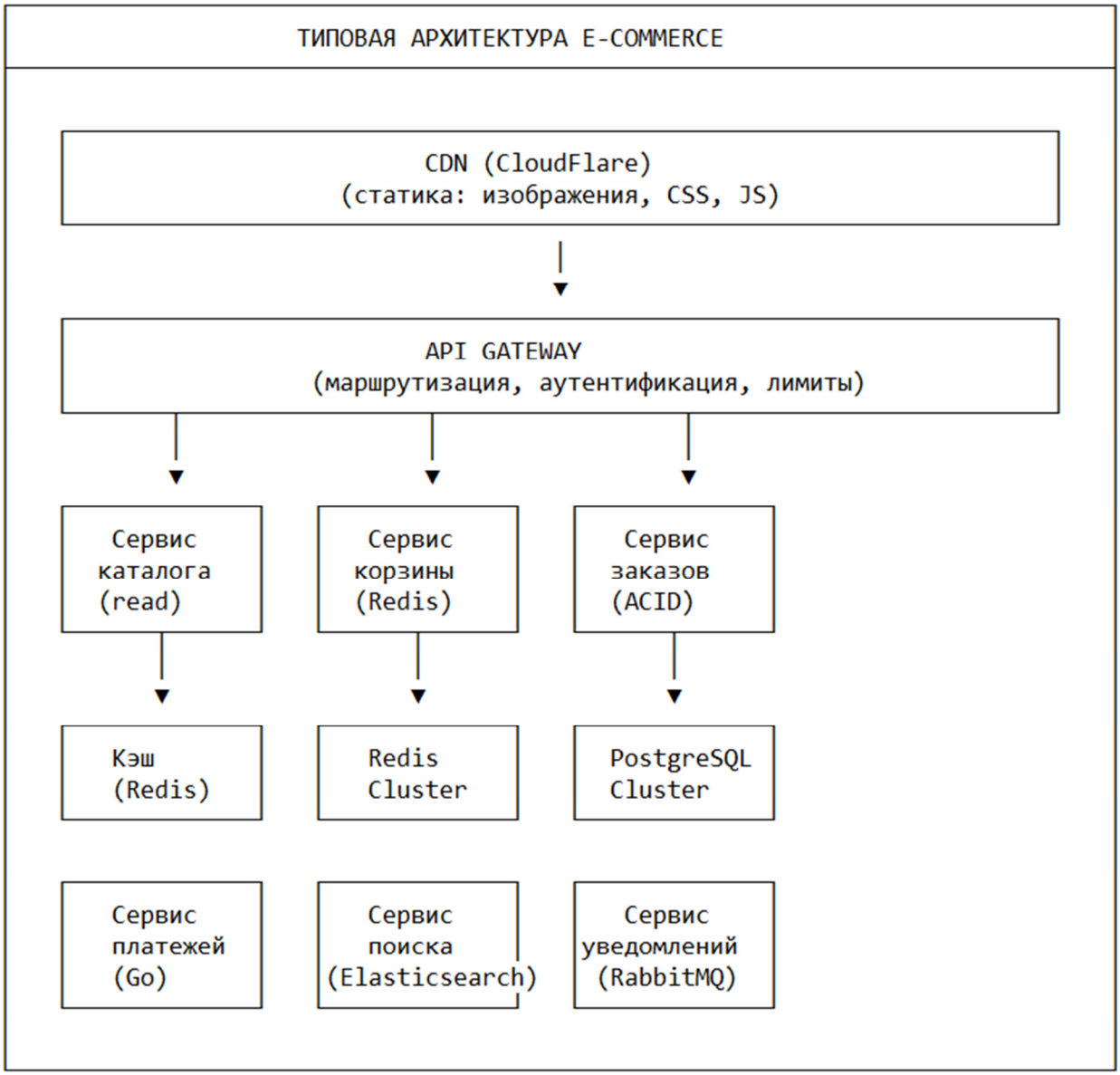
1.1. Особенности e-commerce платформ

E-commerce платформы предъявляют специфические требования к архитектуре:

Характеристика	Требование	Типичные значения

Характеристика	Требование	Типичные значения
Сезонные пики	Выдерживать всплески нагрузки (Чёрная пятница)	10x от обычной нагрузки
Каталог товаров	Высокая скорость чтения, гибкость схемы	1 млн+ товаров, 10k+ категорий
Корзина	Временное хранение, возможна потеря	Хранение до 30 дней
Заказы	ACID-транзакции, консистентность	Нельзя потерять или задвоить
Платежи	Безопасность, идемпотентность	Соответствие PCI DSS
Поиск	Полнотекстовый, фильтрация, сортировка	Поиск < 100 мс
Персонализация	Рекомендации, история	Онлайн/офлайн вычисления

1.2. Типовая архитектура e-commerce



1.3. Ключевые архитектурные решения для e-commerce

Компонент	Решение	Обоснование
Каталог товаров	Денормализованные документы в MongoDB + Redis кэш	Высокая скорость чтения, гибкая схема
Корзина	Redis (TTL 30 дней)	Временное хранилище,

Компонент	Решение	Обоснование
		высокая скорость
Заказы	PostgreSQL (шардирование по user_id)	ACID-транзакции, целостность
Платежи	Отдельный сервис (Go) + идемпотентность	Безопасность, обработка больших объёмов
Поиск	Elasticsearch (индексация каталога)	Полнотекстовый поиск, фильтрация
Уведомления	RabbitMQ + Workers	Асинхронность, отказоустойчивость

1.4. Нагрузочные профили e-commerce

Сценарий	Обычная нагрузка	Чёрная пятница	RPS	Latency (p95)
Просмотр каталога	50%	60%	5000	< 200 мс
Поиск	20%	25%	2000	< 300 мс
Добавление в корзину	15%	10%	1500	< 100 мс
Оформление	10%	4%	5	< 500 мс

Сценарий	Обычная нагрузка	Чёрная пятница	RPS	Latency (p95)
заказа			00	
Платёж	5%	1%	2 00	< 1000 мс

1.5. Стратегии масштабирования для e-commerce

```
СТРАТЕГИИ МАСШТАБИРОВАНИЯ E-COMMERCE

1. КАТАЛОГ (read-heavy)
   • Горизонтальное масштабирование сервиса каталога
   • Redis-кэш: 3 мастер-узла + реплики
   • CDN для статики (изображения, CSS, JS)

2. КОРЗИНА (stateful)
   • Redis Cluster (шардирование по session_id)
   • Репликация для отказоустойчивости
   • TTL для автоматической очистки

3. ЗАКАЗЫ (write-heavy)
   • Шардирование PostgreSQL по user_id
   • Чтение с реплик (read replicas)
   • Асинхронная валидация (stock)

4. ПЛАТЕЖИ (critical)
   • Идемпотентность (по idempotency_key)
   • Очередь retry при временных сбоях
   • Saga для компенсации
```

2. ПРИМЕРЫ ПРАКТИЧЕСКОГО ВЫПОЛНЕНИЯ

2.1. Полный архитектурный проект e-commerce платформы

2.1.1. Требования к системе

Бизнес-требования:

- Интернет-магазин с 1 млн товаров в каталоге
- 100 000 заказов в день в обычный день
- Поддержка пика в Чёрную пятницу (10× нагрузка)

- Поддержка разных типов оплаты (карты, SBP, PayPal)
- Отправка уведомлений (email, SMS, Push)

Технические требования:

- Доступность: 99.99% (≈ 52 минуты даунтайма в год)
- RPS пиковый: 5000 RPS (каталог), 500 RPS (заказы)
- Время ответа: p95 < 500 мс (API), < 200 мс (кэш)
- RTO: < 5 минут, RPO: < 1 минута

2.1.2. Диаграмма контекста (C4 Level 1)

```
plantuml
@startuml
!include https://raw.githubusercontent.com/plantuml-stdlib/C4-PlantUML/master/C4_Context.puml

title Контекстная диаграмма — E-commerce платформа

Person(customer, "Покупатель", "Заказывает товары")
Person(seller, "Продавец", "Управляет товарами")
Person(admin, "Администратор", "Управляет платформой")

System(ecommerce, "E-commerce платформа", "Онлайн-магазин")

System_Ext(payment, "Платёжный шлюз", "Stripe / SBP")
System_Ext(delivery, "Служба доставки", "Отслеживание отправлений")
System_Ext(crm, "CRM", "Управление клиентами")
System_Ext(logistics, "Склад", "Управление остатками")

Rel(customer, ecommerce, "Покупает через", "HTTPS")
Rel(seller, ecommerce, "Управляет через", "HTTPS")
Rel(admin, ecommerce, "Администрирует", "HTTPS")
Rel(ecommerce, payment, "Обрабатывает платежи", "API")
Rel(ecommerce, delivery, "Передаёт заказы", "API")
Rel(ecommerce, crm, "Синхронизирует клиентов", "gRPC")
Rel(ecommerce, logistics, "Проверяет остатки", "API")
```

@enduml

2.1.3. Диаграмма контейнеров (C4 Level 2)

plantuml

@startuml

!include https://raw.githubusercontent.com/plantuml-stdlib/C4-PlantUML/master/C4_Container.puml

title Диаграмма контейнеров — E-commerce платформа

Person(customer, "Покупатель")

System_Boundary(ecommerce, "E-commerce платформа") {

Container(web, "Web SPA", "React", "Пользовательский интерфейс")

Container(gateway, "API Gateway", "Kong", "Маршрутизация, аутентификация, rate limiting")

Container(catalog, "Сервис каталога", "Node.js", "Управление товарами")

Container(cart, "Сервис корзины", "Python/FastAPI", "Временное хранение")

Container(order, "Сервис заказов", "Python/FastAPI", "Оформление и хранение заказов")

Container(payment, "Сервис платежей", "Go", "Проведение платежей")

Container(search, "Сервис поиска", "Node.js", "Полнотекстовый поиск")

Container(notify, "Сервис уведомлений", "Python", "Email/SMS/Push")

ContainerDb(catalog_db, "Каталог БД", "MongoDB", "Товары, категории")

ContainerDb(order_db, "Заказы БД", "PostgreSQL", "Заказы, клиенты")

ContainerDb(cache, "Кэш", "Redis Cluster", "Кэш каталога, корзины")

ContainerDb(search_engine, "Поисковый движок", "Elasticsearch", "Индекс товаров")

ContainerQueue(queue, "Очередь", "RabbitMQ", "Асинхронная обработка")

}

System_Ext(payment_gw, "Платёжный шлюз")

Rel(customer, web, "Использует", "HTTPS")

Rel(web, gateway, "Вызывает API", "HTTPS")

Rel(gateway, catalog, "Маршрутизирует", "HTTP")

Rel(gateway, cart, "Маршрутизирует", "HTTP")

Rel(gateway, order, "Маршрутизирует", "HTTP")

Rel(gateway, payment, "Маршрутизирует", "HTTP")

Rel(gateway, search, "Маршрутизирует", "HTTP")

Rel(catalog, catalog_db, "Читает/пишет", "MongoDB wire")

Rel(catalog, cache, "Читает/обновляет", "Redis protocol")

Rel(cart, cache, "Читает/пишет (TTL 30d)", "Redis protocol")

Rel(order, order_db, "Читает/пишет (ACID)", "SQL")

Rel(order, queue, "Публикует события", "AMQP")

Rel(payment, queue, "Потребляет", "AMQP")

Rel(payment, payment_gw, "Выполняет платеж", "HTTPS API")

Rel(search, search_engine, "Индексирует/ищет", "Elasticsearch API")

Rel(notify, queue, "Потребляет", "AMQP")

@enduml

2.1.4. Схема базы данных (заказы — PostgreSQL)

```
-- Шарды по user_id
CREATE TABLE orders (
    id BIGSERIAL,
    user_id BIGINT NOT NULL,
    total DECIMAL(12,2) NOT NULL,
    status VARCHAR(20) NOT NULL,  -- pending, paid, shipped, cancelled
    payment_id VARCHAR(255),
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    PRIMARY KEY (user_id, id)
) PARTITION BY HASH (user_id);

-- 4 шарда
```

```
CREATE TABLE orders_0 PARTITION OF orders FOR VALUES WITH (MODULUS 4, REMAINDER
0);
CREATE TABLE orders_1 PARTITION OF orders FOR VALUES WITH (MODULUS 4, REMAINDER
1);
CREATE TABLE orders_2 PARTITION OF orders FOR VALUES WITH (MODULUS 4, REMAINDER
2);
CREATE TABLE orders_3 PARTITION OF orders FOR VALUES WITH (MODULUS 4, REMAINDER
3);

-- Индексы
CREATE INDEX idx_orders_user_id ON orders(user_id);
CREATE INDEX idx_orders_status ON orders(status);
CREATE INDEX idx_orders_created_at ON orders(created_at);
```

2.1.5. Архитектурные решения (ADR)

markdown

 # ADR-001: Выбор базы данных для каталога

****Статус:**** Принято

****Контекст:****

Каталог содержит 1 млн+ товаров с разными полями (разные категории).

Частота чтения: 5000 RPS, записи: 100 RPS.

Схема товара часто меняется (добавляются новые характеристики).

****Решение:**** Использовать MongoDB

****Обоснование:****

- Гибкая схема позволяет добавлять поля без миграций
- Высокая производительность чтения (индексы)
- Шардирование из коробки (по _id)

****Последствия:****

- (+): Быстрая разработка, лёгкое изменение схемы

- (-): Нет JOIN с заказами (денормализация копий данных)

 # ADR-002: Асинхронная обработка платежей

****Статус:**** Принято

****Контекст:****

Платёжный шлюз может отвечать медленно (1-5 секунд) или быть временно недоступен.

****Решение:**** Saga с хореографией через RabbitMQ

****Обоснование:****

- Не блокирует оформление заказа
- Позволяет компенсировать отказ (отмена заказа)
- Retry с экспоненциальной задержкой

****Последствия:****

- (+): Отказоустойчивость, высокая доступность оформления
- (-): Платеж не подтверждается мгновенно (eventual consistency)

 # ADR-003: Кэширование каталога

****Статус:**** Принято

****Контекст:****

Каталог товаров — самый частый запрос (5000 RPS).

Прямое обращение к MongoDB даёт latency ~10 мс.

****Решение:**** Redis Cluster с политикой allkeys-lru

****Обоснование:****

- Latency из кэша: ~1 мс
- Hit rate > 95%
- Равномерное распределение по ключам

****Последствия:****

- (+): Высокая производительность
- (-): Eventual consistency (есть шанс увидеть устаревшие цены на 1-2 секунды)

2.1.6. Расчёт ресурсов

Компонент	Обычная нагрузка	Пиковое масштабирование	vCPU	RAM
API Gateway (Kong)	3 реплики	10 реплик	2	4 GB
Сервис каталога	5 реплик	20 реплик	4	8 GB
Сервис корзины (Redis)	3 мастер + 3 реплики	6 мастер + 6 реплик	4	16 GB
Сервис заказов	4 реплики	10 реплик	4	8 GB
PostgreSQL (шарды)	4 шарда × 3	8 шардов × 3 реплики	8	32 GB

Компонент	Обычная нагрузка	Пиковое масштабирование	vCPU	RAM
	реплики			
Elasticsearch	3 горячих + 2 холодных	5 горячих + 3 холодных	8	32 GB
RabbitMQ	3 узла	5 узлов	2	8 GB
Итого	**~60 узлов**	**~130 узлов**	—	—

МЕТОДИЧЕСКИЕ УКАЗАНИЯ ДЛЯ ВЫПОЛНЕНИЯ ПРАКТИЧЕСКОЙ ЧАСТИ РАБОТЫ

Задачи работы:

1. **Анализ требований:**
   - Изучить бизнес-требования к e-commerce платформе
   - Определить ключевые метрики (RPS, latency, объёмы данных)
   - Уточнить SLA, SLO, SLI
2. **Проектирование архитектуры:**
   - Создать диаграмму контекста (C4 Level 1)
   - Создать диаграмму контейнеров (C4 Level 2)
   - Создать диаграмму компонентов для ключевых сервисов
   - Создать диаграмму развёртывания
3. **Выбор технологий и обоснование:**

- Для каждого компонента выбрать технологию
- Сравнить альтернативы (если есть)
- Документировать в ADR

4. **Проектирование данных:**
   - Схема базы данных (SQL-шарды, NoSQL-коллекции)
   - Стратегии индексации
   - Политики хранения и архивации
5. **Расчёт ресурсов и масштабирования:**
   - Оценить необходимые ресурсы для нагрузки
   - Определить стратегию горизонтального масштабирования
   - Оценить стоимость (облако / on-premise)
6. **Защита проекта:**
   - Презентация архитектуры перед группой
   - Ответы на вопросы «стейкхолдеров»

3. ВАРИАНТЫ ЗАДАНИЙ

3.1. Базовый уровень (15 вариантов)

Требования: спроектировать e-commerce платформу с RPS ≤ 500, 3-5 микросервисов

	Особенность	**Каталог (кол-во товаров)**	**Заказы (день)**
	Товары (физические)	100 000	1000
	Цифровые товары (скачивание)	50 000	2000
	Продукты питания (срок годности)	10 000	5000

	Особенность	**Каталог (кол-во товаров)**	**Заказы (день)**
	Одежда (размеры, цвета)	200 000	800
	Электроника (характеристики)	50 000	1500
	Книги (ISBN, авторы)	500 000	500
	Мебель (габариты, доставка)	30 000	300
	Автозапчасти (совместимость)	1 000 000	200
	Косметика (состав, бренды)	100 000	1000
0	Игрушки (возраст)	150 000	2000
1	Спорттовары	80 000	800
2	Строительные материалы	40 000	400
3	Зоотовары	60 000	600
4	Ювелирные изделия	20 000	100
	Аптека (рецепты)	30 000	3000

	Особенность	Каталог (кол-во товаров)	Заказы (день)
5			

3.2. Средний уровень (15 вариантов)

Требования: RPS ≤ 2000, 6-8 микросервисов, шардирование заказов

	Масштаб	RPS пик	Особые требования
	1 млн товаров	2000	Рекомендации (ML)
	500 тыс товаров	1500	Интеграция с 1С (остатки)
	2 млн товаров	3000	Маркетплейс (много продавцов)
	1 млн товаров	2500	Видео-обзоры (CDN)
	800 тыс товаров	2000	Бонусная программа
	1.5 млн товаров	2000	Покупка в кредит
	2 млн товаров	3500	Аукцион (лоты)
	500 тыс товаров	1500	Подписка (товары по расписанию)

	Масштаб	RPS пик	Особые требования
	1 млн товаров	2000	Рассрочка (платежи частями)
0	3 млн товаров	4000	Промокоды, купоны
1	1 млн товаров	2000	Кэшбэк-сервис
2	2 млн товаров	3000	Аналитика продавцам
3	500 тыс товаров	1500	Чат с поддержкой
4	800 тыс товаров	1800	Отзывы с фото
5	1.2 млн товаров	2200	Гарантия, расширенная гарантия

3.3. Продвинутый уровень (15 вариантов)

Требования: RPS ≥ 5000, 10+ микросервисов, мультирегиональность

	География	RPS пик	Особенности
	Россия + СНГ	10 000	Geo-шардирование заказов
	Европа (EU)	15 000	GDPR compliance

	География	**RPS пик**	**Особенности**
	Глобальная (US/EU/AS)	20 000	3 дата-центра
	Китай (CN)	25 000	Интеграция с Alipay, WeChat
	Ближний Восток	8 000	Локализация (ar)
	Юго-Восточная Азия	12 000	Интеграция с Grab/Lalamove
	Латинская Америка	7 000	Офлайн-платежи (boleto)
	Индия	30 000	UPI платежи, кастомизация
	Африка (регион)	5 000	Мобильный first (USSD)
0	Глобальная (4 континента)	40 000	Active-Active во всех регионах
1	Европа + US	18 000	Низкая latency (50ms p99)
2	Россия (вся страна)	15 000	DDoS защита от RuNet
3	Глобал + умные рекомендации	25 000	RAG / LLM для поиска
4	Глобал + real-time инвентарь	35 000	Очереди Kafka, стриминг
	Маркетплейс + логистика	50 000	Сеть фулфилмент-центров

	География	RPS пик	Особенности
5			

4. КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Какие компоненты входят в типовую архитектуру e-commerce?
2. Почему каталог товаров часто реализуют на NoSQL, а заказы — на SQL?
3. Какую базу данных вы выберете для каталога товаров и почему?
4. Как решить проблему конкурентного бронирования товара (last unit)?
5. Как обеспечить идемпотентность платежей?
6. Какую стратегию кэширования каталога вы примените?
7. Как шардировать заказы по user_id?
8. Какие метрики SLO необходимо определить для e-commerce?
9. Как обеспечить отказоустойчивость корзины?
10. Как спроектировать поиск с фильтрацией по цене, категории, бренду?
11. Как масштабировать сервис каталога в Чёрную пятницу?
12. Какую роль играет API Gateway в e-commerce?
13. Как синхронизировать остатки товаров со складом (warehouse)?
14. Как интегрировать внешние платёжные системы с сохранением идемпотентности?
15. Как спроектировать систему рекомендаций (офлайн/онлайн)?

5. ТРЕБОВАНИЯ К ОТЧЕТУ

1. **Титульный лист**
2. **Цель работы**
3. **Теоретическая часть:** e-commerce архитектура, компоненты, стратегии масштабирования

4. **Практическая часть:**
   - Диаграммы C4 (Level 1, 2, 3)
   - Схема БД (SQL + NoSQL)
   - Таблица технологий с обоснованием
   - ADR (3-5 документов)
   - Расчёт ресурсов
5. **Результаты:** описание архитектуры, интеграций, мониторинг
6. **Рекомендации:** по улучшению, по тестированию
7. **Вывод**
8. **Ответы на контрольные вопросы**

---

6. КРИТЕРИИ ОЦЕНИВАНИЯ

Критерий	**Базовый (10)**	**Средний (14)**	**Продвинутый (18)**
Анализ требований	2	1	1
Диаграммы (C4 Level 1,2)	3	2	2
Схема БД (SQL+NoSQL)	2	2	2
Выбор технологий (обоснование)	3	2	2
ADR (1/3/5 штук)	—	3	2
Расчёт ресурсов	—	2	2
Презентация / защита	—	2	3
Качество отчёта	—	—	4

Критерий	Базовый (10)	Средний (14)	Продвинутый (18)
Максимум	**10**	**14**	**18**

Шкала перевода в оценку:

- **5 (отлично):** 90%+
- **4 (хорошо):** 75–89%
- **3 (удовлетворительно):** 60–74%
- **2 (неудовлетворительно):** <60%

7. ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ

Программа	Назначение	Доступность в РФ
Draw.io / PlantUML	Диаграммы C4	+
Mermaid	Диаграммы в Markdown	+
Markdown-редактор	Оформление отчёта	+
Excel / Google Sheets	Расчёты ресурсов	+
Презентационное ПО	Защита проекта	+

Лабораторная работа №17

Архитектурный кейс: система доставки (Delivery Hero-style)

Цель работы: Спроектировать архитектуру георасределённой системы доставки с управлением курьерами и трекингом в реальном времени (на примере Delivery Hero / Яндекс.Еда), включая обработку геоданных, алгоритмы назначения заказов ближайшим курьерам, обеспечение

отказоустойчивости на уровне регионов и оценку полной стоимости владения (TCO).

Формируемые компетенции: ПК-5

1. ТЕОРЕТИЧЕСКОЕ ОБОСНОВАНИЕ

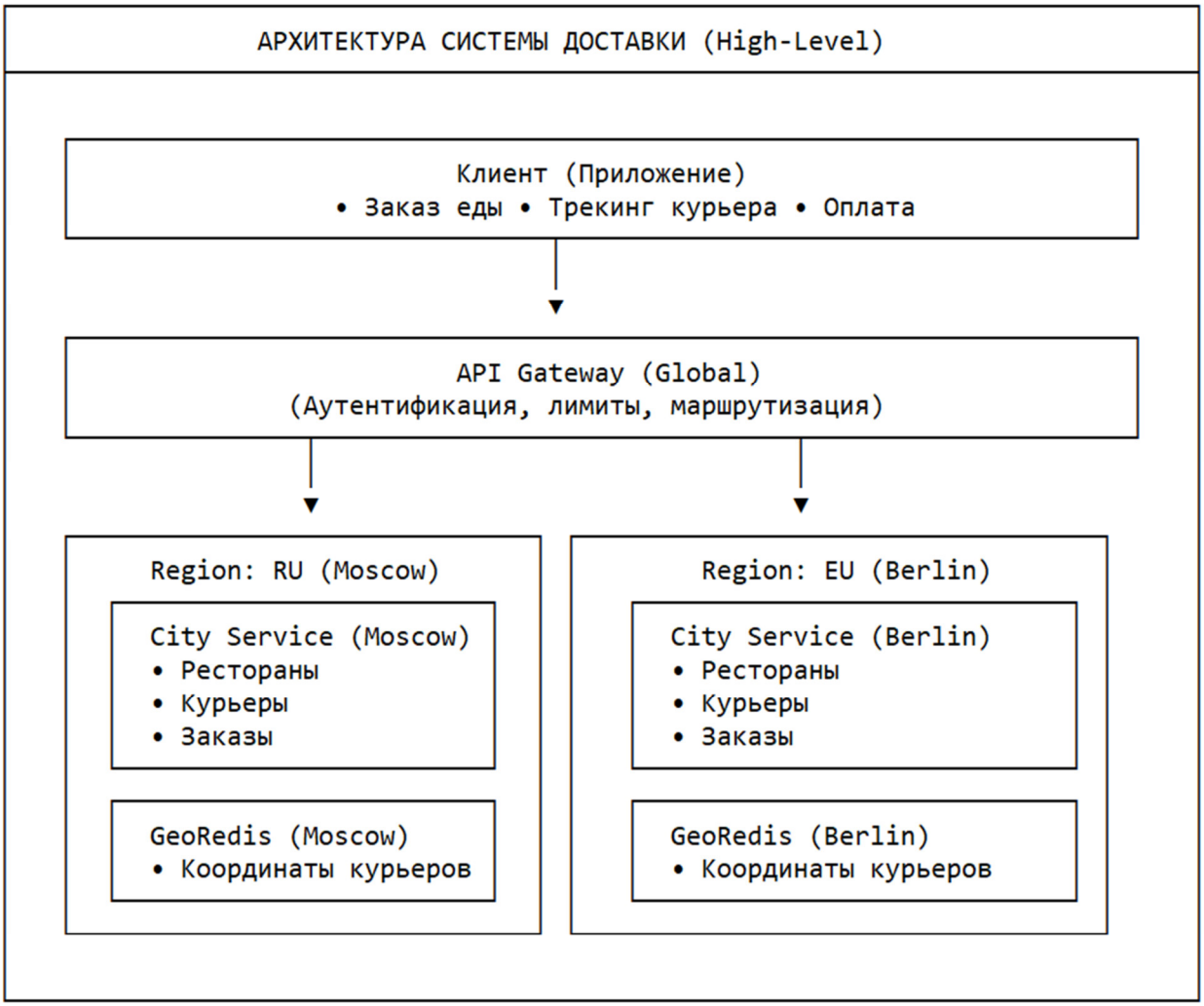
1.1. Особенности систем доставки (Delivery Hero-style)

Системы доставки еды предъявляют специфические требования к архитектуре, отличающие их от классических e-commerce платформ:

Характеристика	Требование	Типичные значения
Геораспределение	Низкая latency в разных регионах	< 100 мс до ближайшего дата-центра
Real-time трекинг	Обновление позиции курьера	1-3 секунды
Динамическое ценообразование	Изменение цены в реальном времени	Отклик < 1 секунды
Матчинг (назначение курьера)	Оптимальное распределение заказов	< 5 секунд
Сезонные пики	Праздники, непогода	5-10× от обычной нагрузки
Гео-шардирование	Курьер в регионе → данные в том же регионе	Гарантия фоллбека

1.2. Geo-шардирование (региональная изоляция)

Главный принцип масштабирования системы доставки — **гео-шардирование**: данные и вычисления локализованы в том регионе, где происходит заказ.



1.3. Хранение геоданных (Redis Geo)

Redis предоставляет встроенную поддержку геопространственных индексов через структуру данных **GeoSet** (реализована на основе Sorted Set).

Команды Redis Geo:

Команда	**Описание**	**Пример**
GEOADD	Добавление	GEOADD couriers 37.6176

Команда	Описание	Пример
	координат	55.7558 "courier_123"
GEORADIUS	Поиск в радиусе	GEORADIUS couriers 37.6176 55.7558 5 km WITHDIST
GEORADIUSBYMEMBER	Поиск относительно точки	GEORADIUSBYMEMBER couriers "restaurant_456" 3 km
GEODIST	Расстояние между точками	GEODIST couriers "courier_123" "restaurant_456" km

Пример базы Redis (Python):

```python
import redis

r = redis.Redis(host='localhost', port=6379)

# Добавление курьеров с координатами
r.geoadd("couriers:moscow", 37.6176, 55.7558, "courier_1")  # Москва
r.geoadd("couriers:moscow", 37.6800, 55.7700, "courier_2")  # Москва
r.geoadd("couriers:moscow", 37.5000, 55.7000, "courier_3")  # Москва

# Поиск ближайших курьеров к ресторану
restaurant_lon, restaurant_lat = 37.6200, 55.7600
nearest = r.georadius(
    "couriers:moscow",
    restaurant_lon, restaurant_lat,
    5, "km",
    withdist=True,
    sort="ASC",
    count=3
)
print(f"Nearest couriers: {nearest}")
# Вывод: [('courier_1', 0.8), ('courier_2', 5.2), ('courier_3', 8.1)]
```

1.4. Алгоритмы назначения курьеров

Проблема: нужно назначить заказ оптимальному курьеру с учётом расстояния, статуса курьера, рейтинга, загруженности.

Базовый алгоритм (Greedy):

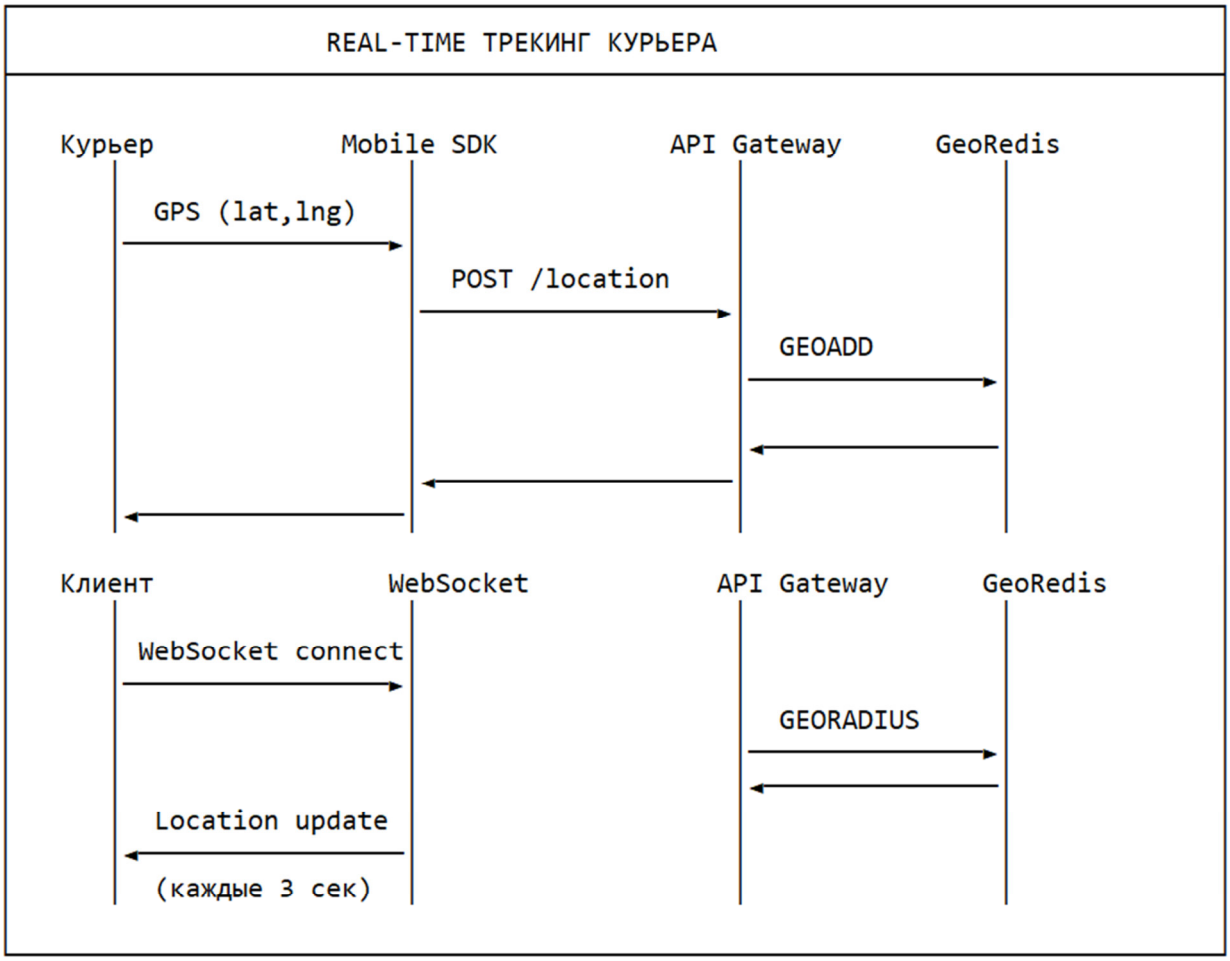
1. При поступлении заказа:
   - Получить все свободные курьеры в радиусе 5 км от ресторана
   - Отсортировать по расстоянию (ближайший → первый)
   - Выбрать ближайшего доступного курьера
   - Назначить заказ

Расширенный алгоритм (Dynamic Matching):

```
def match_order_to_courier(order_id, restaurant_coords):
    # 1. Поиск свободных курьеров в радиусе
    available_couriers = find_free_couriers(restaurant_coords, radius_km=5)

    # 2. Расчёт скора для каждого курьера
    scored_couriers = []
    for courier in available_couriers:
        distance = geodist(restaurant_coords, courier['coords'])

        score = (
            - distance * 0.4              # Чем ближе, тем лучше
            + courier['rating'] * 0.3     # Чем выше рейтинг, тем лучше
            - courier['load'] * 0.2        # Чем меньше заказов, тем лучше
            + courier['speed'] * 0.1       # Чем быстрее, тем лучше
        )
        scored_couriers.append((courier, score))

    # 3. Выбор лучшего
    best = max(scored_couriers, key=lambda x: x[1])

    # 4. Назначение заказа
    assign_order(best[0]['id'], order_id)

    # 5. Обновление статуса курьера
    update_courier_status(best[0]['id'], 'busy')
```

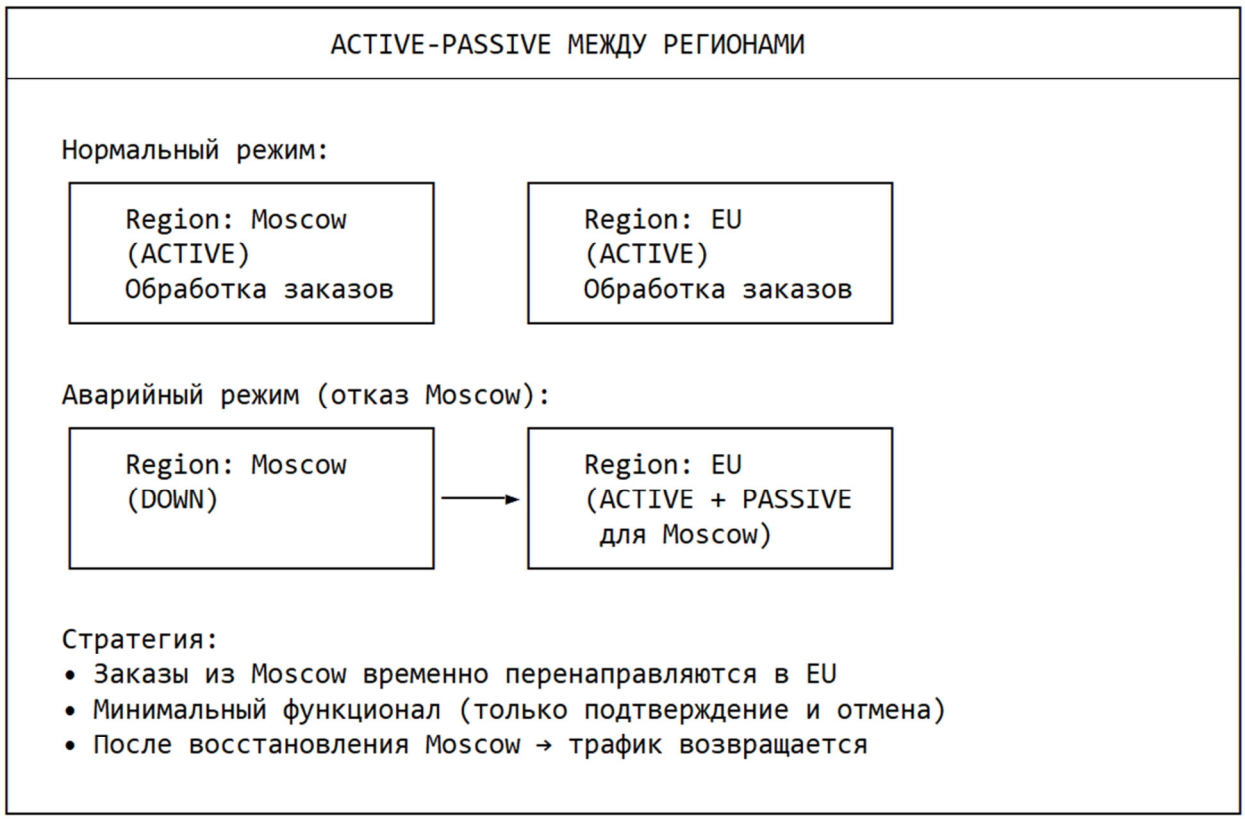
1.5. Real-time трекинг курьеров

Поток обновлений:



1.6. Отказоустойчивость на уровне региона

Active-Passive с георазпределением:



2. ПРИМЕРЫ ПРАКТИЧЕСКОГО ВЫПОЛНЕНИЯ

2.1. Полный архитектурный проект системы доставки

2.1.1. Требования к системе

Бизнес-требования:

- 10 млн заказов в день в пик (Чёрная пятница, новогодние праздники)
- 3 млн активных курьеров
- 500 000 ресторанов
- Время доставки: цель < 45 минут (p95)

Технические требования:

- Доступность: 99.99% (регионально), 99.9% (глобально)
- Пиковый RPS: 50 000 (API Gateway), 100 000 (гео-запросы)
- Задержка трекинга: < 3 секунды
- RTO (межрегиональный): < 5 минут
- RPO (межрегиональный): < 1 минута

2.1.2. Диаграмма контекста (C4 Level 1)

```
plantuml
@startuml
!include https://raw.githubusercontent.com/plantuml-stdlib/C4-PlantUML/master/C4_Context.puml

title Контекстная диаграмма — Система доставки (Delivery Hero-style)

Person(customer, "Клиент", "Заказывает еду")
Person(courier, "Курьер", "Доставляет заказы")
Person(restaurant, "Ресторан", "Готовит еду")
Person(admin, "Администратор", "Управляет платформой")

System(delivery, "Система доставки", "Платформа заказа и доставки еды")

System_Ext(payment, "Платёжный шлюз", "Обработка платежей")
System_Ext(maps, "Картографический сервис", "Google Maps / 2GIS")
System_Ext(push, "Push-сервис", "FCM / APNS")

Rel(customer, delivery, "Оформляет заказ", "HTTPS/WebSocket")
Rel(courier, delivery, "Обновляет статус", "HTTPS")
Rel(restaurant, delivery, "Принимает заказы", "HTTPS")
Rel(admin, delivery, "Администрирует", "HTTPS")

Rel(delivery, payment, "Списывает оплату", "API")
Rel(delivery, maps, "Строит маршруты", "API")
Rel(delivery, push, "Отправляет уведомления", "API")

@enduml
```

2.1.3. Диаграмма контейнеров (C4 Level 2)

```
plantuml
@startuml
!include https://raw.githubusercontent.com/plantuml-stdlib/C4-PlantUML/master/C4_Container.puml

title Диаграмма контейнеров — Система доставки (Geo-распределённая)

Person(customer, "Клиент")
Person(courier, "Курьер")
```

```
Person(restaurant_owner, "Ресторан")

System_Boundary(delivery, "Система доставки") {

    Container(gateway, "API Gateway", "Kong", "Маршрутизация, auth, rate limiting")

    Container(order, "Сервис заказов", "Python/FastAPI", "Управление заказами")
    Container(matching, "Матчинг сервис", "Go", "Назначение курьеров")
    Container(tracking, "Трекинг сервис", "Node.js", "Обновление позиций курьеров")
    Container(payment, "Сервис платежей", "Go", "Проведение платежей")
    Container(notification, "Уведомления", "Python", "Push/WebSocket")

    ContainerDb(geo, "Geo-хранилище", "Redis Cluster", "Координаты курьеров (GeoSet)")
    ContainerDb(order_db, "Заказы БД", "PostgreSQL", "Заказы, транзакции")
    ContainerDb(courier_db, "Курьеры БД", "PostgreSQL", "Профили, статусы")
    ContainerDb(restaurant_db, "Рестораны БД", "MongoDB", "Меню, часы работы")

    ContainerQueue(queue, "Очередь", "Kafka", "События (заказы, трекинг)")
}

System_Ext(payment_gw, "Платёжный шлюз")
System_Ext(push_gw, "Push-сервис")

Rel(customer, gateway, "Вызывает API", "HTTPS")
Rel(courier, gateway, "Вызывает API", "HTTPS")
Rel(restaurant_owner, gateway, "Вызывает API", "HTTPS")

Rel(gateway, order, "Маршрутизирует")
Rel(gateway, matching, "Маршрутизирует")
Rel(gateway, tracking, "Маршрутизирует")
Rel(gateway, payment, "Маршрутизирует")

Rel(order, order_db, "Читает/пишет", "SQL")
Rel(order, queue, "Публикует OrderCreated", "Kafka")

Rel(matching, geo, "Поиск ближайших", "Redis Geo")
Rel(matching, courier_db, "Статусы курьеров", "SQL")
Rel(matching, queue, "Публикует CourierAssigned", "Kafka")
```

Rel(tracking, geo, "Обновляет координаты", "GEOADD")

Rel(tracking, queue, "Публикует LocationUpdate", "Kafka")

Rel(queue, notification, "Потребляет", "Kafka")

Rel(notification, push_gw, "Отправляет уведомления", "API")

@enduml

2.1.4. Сервис матчинга (назначение курьеров) — Python пример

```
# matching_service.py
import redis
import asyncpg
from kafka import KafkaProducer
import json
import math

class CourierMatcher:
    def __init__(self):
        self.redis = redis.Redis(
            host='geo-redis',
            port=6379,
            decode_responses=True
        )
        self.producer = KafkaProducer(
            bootstrap_servers='kafka:9092',
            value_serializer=lambda v: json.dumps(v).encode('utf-8')
        )

    def _haversine_distance(self, lat1, lon1, lat2, lon2):
        """Расстояние между двумя точками (формула гаверсинуса)"""
        R = 6371  # Радиус Земли в км
        dlat = math.radians(lat2 - lat1)
        dlon = math.radians(lon2 - lon1)
        a = math.sin(dlat/2)**2 + math.cos(math.radians(lat1)) *
math.cos(math.radians(lat2)) * math.sin(dlon/2)**2
        c = 2 * math.asin(math.sqrt(a))
        return R * c

    async def find_and_assign_courier(self, order_id, restaurant_id,
restaurant_lat, restaurant_lon):
        """Поиск и назначение ближайшего курьера"""

        # 1. Поиск свободных курьеров в радиусе 5 км
        couriers = self.redis.georadius(
            f"couriers:location",
            restaurant_lon, restaurant_lat,
```

```
            5, "km",
            withdist=True,
            withcoord=True,
            sort="ASC"
        )

        if not couriers:
            # Расширяем радиус до 10 км
            couriers = self.redis.georadius(
                f"couriers:location",
                restaurant_lon, restaurant_lat,
                10, "km",
                withdist=True,
                withcoord=True,
                sort="ASC"
            )

        if not couriers:
            return {"error": "No couriers available"}

        # 2. Фильтрация по статусу и загрузке
        available = []
        for courier_data in couriers[:20]:  # Берём 20 ближайших
            courier_id = courier_data[0]
            distance = courier_data[1]
            coords = courier_data[2]

            # Проверка статуса курьера в отдельной БД
            # (здесь упрощённо)
            status = await self.get_courier_status(courier_id)
            if status == 'free':
                available.append({
                    'id': courier_id,
                    'distance': distance,
                    'lat': coords[1],
                    'lon': coords[0]
                })

        if not available:
            return {"error": "No free couriers"}

        # 3. Выбор ближайшего
        best = min(available, key=lambda x: x['distance'])

        # 4. Назначение заказа
        assignment = {
            'order_id': order_id,
            'courier_id': best['id'],
            'restaurant_id': restaurant_id,
            'assigned_at': self._now_iso()
        }
```

```
        # Обновление статуса курьера
        await self.update_courier_status(best['id'], 'assigned', order_id)

        # Отправка события в Kafka
        self.producer.send('courier_assigned', value=assignment)

        # Обновление метрики (в Redis)
        self.redis.zadd('metrics:assignments', {order_id: best['distance']})

        return {
            'courier_id': best['id'],
            'distance_km': round(best['distance'], 2),
            'eta_min': round(best['distance'] / 0.3)  # 18 км/ч = 300 м/мин
        }

    async def get_courier_status(self, courier_id):
        """Получение статуса курьера из PostgreSQL"""
        # Здесь был бы asyncpg запрос
        # Упрощённо:
        return 'free'

    async def update_courier_status(self, courier_id, status, order_id=None):
        """Обновление статуса в PostgreSQL"""
        pass

    def _now_iso(self):
        import datetime
        return datetime.datetime.now().isoformat()
```

2.1.5. Архитектурные решения (ADR)

markdown

 # ADR-001: Гео-шардирование (региональная изоляция)

****Статус:**** Принято

****Контекст:****

Система работает в 50+ странах с разными законодательными требованиями.

Многие данные не могут покидать страну (GDPR, ФЗ-152).

****Решение:****

Разделить систему на независимые региональные кластеры.
Каждый кластер обрабатывает заказы только своего региона.

Обоснование:
- Соответствие законодательству
- Низкая latency (данные близко к пользователям)
- Изоляция сбоев

Последствия:
- (+): Отказ одного региона не влияет на другие
- (-): Сложность кросс-региональной отчётности

ADR-002: Redis для трекинга курьеров

Статус: Принято

Контекст:
3 млн курьеров обновляют позицию каждые 3 секунды.
Требуется быстрый поиск ближайших (геозапросы).

Решение:
Использовать Redis GeoSet для хранения текущих координат курьеров.

Обоснование:
- Встроенные гео-команды (GEORADIUS)
- Высокая производительность (in-memory)
- TTL для устаревших данных

Последствия:
- (+): 100k+ запросов/сек, latency ~1-2 мс

- (-): Данные теряются при отказе Redis (потеря последней позиции)

 # ADR-003: Kafka для событий

****Статус:**** Принято

****Контекст:****

Сервисы должны обмениваться событиями: OrderCreated, CourierAssigned, DeliveryStarted.

****Решение:****

Использовать Apache Kafka для событийной архитектуры.

****Обоснование:****

- Сохранение порядка событий по ключу (partition by order_id)
- replay событий
- Высокая пропускная способность

****Последствия:****

- (+): Надёжность, replay, масштабируемость
- (-): Сложность настройки кластера

2.1.6. Расчёт ресурсов (на один регион с 5 млн заказов/день)

Компонент	Реплик (обычная/пик)	vCPU	RAM	Хранилище
API Gateway (Kong)	5 / 20	2	4 GB	—
Сервис заказов	10 / 40	4	8 GB	—
Матчинг сервис	15 / 60	4	8 GB	—

Компонент	Реплик (обычная/пик)	vCPU	RAM	Хранилище
Трекинг сервис	5 / 20	4	8 GB	—
Сервис платежей	8 / 30	4	8 GB	—
Уведомления	10 / 30	2	4 GB	—
Redis Geo	6 / 6 (шарды)	8	64 GB	—
PostgreSQL (заказы)	5 шардов × 3 реплики	16	64 GB	2 TB
PostgreSQL (курьеры)	3 шарда × 2 реплики	8	32 GB	500 GB
MongoDB (меню)	3 шарда × 2 реплики	8	32 GB	1 TB
Kafka	5 узлов	8	32 GB	5 TB

Итого на регион: ~250 pods в обычное время, ~800 pods в пик

МЕТОДИЧЕСКИЕ УКАЗАНИЯ ДЛЯ ВЫПОЛНЕНИЯ ПРАКТИЧЕСКОЙ ЧАСТИ РАБОТЫ

Задачи работы:

1. **Анализ требований к системе доставки:**
   - Определить ключевые метрики (заказы/день, курьеры, города)
   - Уточнить географию — один регион или глобальный запуск
   - Определить SLA/SLO (доставка 45 минут, трекинг 3 секунды)
2. **Проектирование архитектуры:**
   - Разработать диаграммы C4 (Level 1, 2, 3)
   - Региональная изоляция (как минимум 2+ региона)
   - Geo-шардирование ключа (данные не покидают регион)
3. **Реализация трекинга на Redis Geo:**
   - Продемонстрировать GEOADD/GEORADIUS
   - Измерить latency запросов (≤ 2 мс)

- Настроить TTL на ключи курьеров (например, 10 секунд)

4. **Алгоритм матчинга:**
   - Реализовать базовый greedy (ближайший курьер)
   - Добавить скоринг (расстояние, рейтинг, загрузка)
   - Выбрать критерии + веса
5. **Обеспечение отказоустойчивости:**
   - Active-Active в разных регионах
   - Или Active-Passive с гео-балансировкой
   - Рассчитать RTO / RPO для регионального отказа
6. **Расчёт TCO (полная стоимость владения):**
   - Подсчёт ресурсов (CPU, RAM, storage)
   - Оценка облака (модель spot/ondemand)
   - Бюджет на межрегиональный трафик
7. **Защита проекта:**
   - Презентация архитектуры (10-15 минут)
   - Ответы на вопросы стейкхолдеров

3. ВАРИАНТЫ ЗАДАНИЙ

3.1. Базовый уровень (15 вариантов)

Требования: 1 регион, RPS ≤ 5000, 3-5 микросервисов

№	Город (регион)	Заказов в день	Курьеров	Ресторанов
1	Москва	100 000	10 000	5 000
2	Санкт-Петербург	50 000	5 000	2 000
3	Новосибирск	20 000	2 000	1 000
4	Екатеринбург	25 000	2 500	1 200

№	Город (регион)	Заказов в день	Курьеров	Ресторанов
5	Казань	15 000	1 500	800
6	Нижний Новгород	12 000	1 200	600
7	Челябинск	10 000	1 000	500
8	Самара	10 000	1 000	500
9	Омск	8 000	800	400
10	Ростов-на-Дону	12 000	1 200	600
11	Уфа	8 000	800	400
12	Красноярск	6 000	600	300
13	Пермь	6 000	600	300
14	Воронеж	5 000	500	250
15	Волгоград	5 000	500	250

3.2. Средний уровень (15 вариантов)

Требования: 2-3 региона, RPS до 20 000, 6-8 микросервисов

№	Регионы	Пик (заказы/день)	Тип доставки	Сложность
1	Москва + СПб + Казань	200 000	Курьерская	Средняя

№	Регионы	Пик (заказы/день)	Тип доставки	Сложность
2	Европа (Лондон, Париж, Берлин)	500 000	Курьерская	Высокая
3	США (NY, LA, Chicago)	1 000 000	Курьерская	Высокая
4	Китай (Шанхай, Пекин)	2 000 000	Курьерская + дроны	Высокая
5	ОАЭ (Дубай, Абу-Даби)	100 000	Курьерская	Средняя
6	Турция (Стамбул, Анкара)	80 000	Курьерская	Средняя
7	Индия (Мумбаи, Дели)	500 000	Курьерская + вело	Высокая
8	Бразилия (Сан-Паулу, Рио)	150 000	Курьерская	Средняя
9	Австралия (Сидней, Мельбурн)	100 000	Курьерская	Средняя
10	Канада (Торонто, Ванкувер)	80 000	Курьерская	Средняя
11	Япония (Токио, Осака)	200 000	Курьерская	Высокая
12	Корея (Сеул)	150 000	Курьерская	Высокая
13	Мексика (Мехико)	100 000	Курьерская	Средняя

№	Регионы	Пик (заказы/день)	Тип доставки	Сложность
14	Индонезия (Джакарта)	80 000	Курьерская	Средняя
15	Нигерия (Лагос)	50 000	Курьерская	Низкая/высокая

3.3. Продвинутый уровень (15 вариантов)

Требования: глобальная (5+ регионов), RPS ≥ 50 000, 10+ микросервисов + ML

№	География	RPS пик	Особые технологии	Интеграции
1	Глобальная (6 регионов)	100 000	Kafka Streams, Redis Geo	ML для ETA
2	Евразия (5 регионов)	80 000	Flink для аналитики, ClickHouse	Биржа курьеров (аукцион)
3	Северная Америка + EU	120 000	NATS, ScyllaDB, TimescaleDB	Динамическое ценообразование
4	Азиатско-Тихоокеанский	200 000	Pulsar, YugabyteDB	Многоязычная поддержка
5	Латинская Америка	60 000	Celery, Redis Cluster	Офлайн-платежи
6	Ближний Восток	70 000	RabbitMQ, MongoDB Atlas	Арабская локализация
7	Африка	40 000	MQTT, MinIO	Лёгкие клиенты (USSD)
8	Индийский	250	gRPC, Cassandra +	UPI платежи

№	География	RPS пик	Особые технологии	Интеграции
	субконтинент	000	Redis	
9	Россия + СНГ	150 000	Tarantool, Postgres Pro	2GIS, СБП
10	Юго-Восточная Азия	180 000	Kafka + Flink, TiDB	Grab/Lalamove интеграция
11	Океания	40 000	NATS, CockroachDB	Интеграция с Uber Direct
12	Центральная Европа	150 000	RabbitMQ + Redis, InfluxDB	ML (предсказание времени)
13	Северная Европа	100 000	Pulsar, Elasticsearch	Приоритеты устойчивости
14	Южная Европа	120 000	Redis + Kafka, Trino	Сезонная динамика
15	Глобал (low-latency)	300 000	gRPC + Envoy, ScyllaDB, Apache Pinot	Real-time ML для матчинга

4. КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Чем архитектура системы доставки отличается от e-commerce?
2. Что такое гео-шардирование и почему оно критично для Delivery Hero?
3. Как Redis GeoSet позволяет искать ближайших курьеров?
4. Какие алгоритмы матчинга курьеров существуют? Сравните их.
5. Как обеспечить eventual consistency при трекинге курьеров?
6. Какие данные лучше хранить в Redis, а какие в PostgreSQL?
7. Как синхронизировать состояние заказа и статус курьера?

8. Как тестировать алгоритмы назначения курьеров (A/B тесты)?

9. Что такое «мёртвая петля» (dead loop) в трекинге и как её избежать?

10. Как архитектура системы доставки влияет на время доставки (ETD)?

11. Как спроектировать региональную отказоустойчивость (RTO/RPO)?

12. Как обрабатывать перегрузку сервиса матчинга в часы пик?

13. Как интегрировать внешние картографические API (Google Maps, 2GIS)?

14. Как строить маршрут для курьера с несколькими заказами (batch)?

15. Какие метрики SLO важны для системы доставки?

5. ТРЕБОВАНИЯ К ОТЧЕТУ

Отчёт должен содержать:

1. **Титульный лист**
2. **Цель работы**
3. **Теоретическая часть:** особенности delivery-систем, гео-шардирование, алгоритмы матчинга
4. **Практическая часть:**
   - Диаграммы C4 (Level 1, 2, 3)
   - Схема базы данных (Redis Geo, PostgreSQL)
   - Реализация матчинга (псевдокод или код)
   - Время отклика поиска курьера
   - ADR (3-5 документов)
   - Расчёт ресурсов и TCO
5. **Рекомендации:** мониторинг, аварийное восстановление
6. **Вывод**
7. **Ответы на контрольные вопросы**

6. КРИТЕРИИ ОЦЕНИВАНИЯ

Критерий	Базовый (10)	Средний (14)	Продвинутый (18)
Анализ требований	2	1	1
Диаграммы C4 (1,2,3)	3	2	2
Redis Geo (GEOADD/GEORADIUS)	2	2	1
Алгоритм матчинга (код/псевдокод)	3	2	2
ADR (1/3/5)	—	3	2
Региональная отказоустойчивость	—	2	2
Расчёт TCO (ресурсы, бюджет)	—	1	2
Защита (презентация)	—	1	3
Качество отчёта	—	—	3
Максимум	**10**	**14**	**18**

Шкала перевода в оценку:

- **5 (отлично):** 90%+
- **4 (хорошо):** 75–89%
- **3 (удовлетворительно):** 60–74%
- **2 (неудовлетворительно):** <60%

7. ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ

Программа	Назначение	Доступность в РФ
Redis (с Geo-модулем)	Хранение координат, геопоиск	+
PostgreSQL	Заказы, курьеры (ACID)	+
Apache Kafka	Событийная шина	+
Python / Go	Сервисы матчинга, трекинга	+
PlantUML / Draw.io	Диаграммы C4	+
Excel / Google Sheets	Расчёт TCO	+

Лабораторная работа №18

Проектирование, защита и обзор курса

Цель работы: Комплексная защита итогового архитектурного проекта высоконагруженной и отказоустойчивой системы по выбранному кейсу (e-commerce, система доставки или другой согласованный вариант) с демонстрацией всех компонентов архитектуры, обоснованием принятых решений и ответами на вопросы стейкхолдеров, а также подведение итогов освоения дисциплины.

Формируемые компетенции: ПК-5

1. ТЕОРЕТИЧЕСКОЕ ОБОСНОВАНИЕ

1.1. Цель итоговой защиты

Итоговая защита является завершающим этапом лабораторного практикума, на котором студент демонстрирует:

- **Системное мышление** - способность видеть архитектуру целиком, а не отдельные компоненты.
- **Обоснованность решений** - умение объяснить, почему выбрана та или иная технология или паттерн.
- **Понимание компромиссов** - осознание trade-offs между атрибутами качества.
- **Навыки презентации** - способность донести сложные архитектурные идеи до разных аудиторий.

1.2. Структура защиты

```
СТРУКТУРА ЗАЩИТЫ (15 минут)

1. ВСТУПЛЕНИЕ (1-2 минуты)
   • Название проекта
   • Проблемная область
   • Ключевые требования

2. АРХИТЕКТУРА ВЫСОКОГО УРОВНЯ (3-4 минуты)
   • C4 Level 1 (контекст) — кто взаимодействует
   • C4 Level 2 (контейнеры) — основные сервисы и БД
   • Ключевые технологические решения

3. КЛЮЧЕВЫЕ АСПЕКТЫ (5-6 минут)
   • Масштабирование и отказоустойчивость
   • Управление данными (шардирование, репликация)
   • Асинхронность (очереди, события)
   • Наблюдаемость (мониторинг, трассировка)

4. КОМПРОМИССЫ И РИСКИ (2-3 минуты)
   • Какие trade-offs были выбраны
   • Какие риски остаются
   • План митигации рисков

5. ЗАКЛЮЧЕНИЕ И ВОПРОСЫ (2-3 минуты)
   • Основные выводы
   • Ответы на вопросы стейкхолдеров
```

1.3. Ролевая игра: стейкхолдеры

При защите преподаватель и группа выступают в роли заинтересованных лиц:

Стейкхолдер	Типичные вопросы

Стейкхолдер	Типичные вопросы
CTO (технический директор)	Почему вы выбрали эту технологию? Как вы оцениваете риски?
Product Owner	Как архитектура влияет на скорость вывода новых фич?
DevOps инженер	Как вы разворачиваете систему? Есть ли документация по мониторингу?
Security officer	Как защищены данные пользователей? Есть ли шифрование?
FinOps / CFO	Какова полная стоимость владения (TCO)? Какие облачные затраты?
Разработчик	Как устроен обмен данными между сервисами? Есть ли API документация?

1.4. Критерии успешной защиты

Критерий	Что оценивается
Полнота	Охвачены ли все аспекты: масштабирование, отказоустойчивость, данные, безопасность?
Обоснованность	Каждое решение имеет аргументацию «почему, а не альтернатива»
Осознание компромиссов	Признаются ли недостатки выбранных

Критерий	Что оценивается
	решений?
Качество презентации	Структура, наглядность, уверенность ответов
Техническая глубина	Понимание деталей (не только «картинки»)

1.5. Перечень обязательных элементов итогового проекта

	Элемент	Формат	ес
	Диаграмма контекста (C4 Level 1)	PlantUML / Draw.io	%
	Диаграмма контейнеров (C4 Level 2)	PlantUML / Draw.io	10%
	Диаграмма компонентов для 1-2 ключевых сервисов (C4 Level 3)	PlantUML / Draw.io	10%
	Диаграмма развёртывания	PlantUML / Draw.io	5%
	Схема базы данных (SQL + NoSQL)	Текст + диаграмма	10%
	Таблица выбора технологий с обоснованием	Таблица в отчёте	10%
	ADR (минимум 3 документа)	Markdown	10%

	Элемент	Формат	ес
	Расчёт ресурсов и TCO	Таблица + расчёты	10%
	Презентация (10-15 слайдов)	PPT / Google Slides	15%
0	Устная защита + ответы на вопросы	Устно	15%

2. ПРИМЕРЫ ПРАКТИЧЕСКОГО ВЫПОЛНЕНИЯ

2.1. Пример презентации (структура слайдов)

Слайд 1: Титульный

Архитектурный проект

Система доставки еды (Delivery Hero-style)

Студент: Иванов И.И.

Группа: ПИ-М-01

Вариант: 17 (глобальная доставка)

Слайд 2: Проблемная область и требования

Проблема:

• Миллион заказов в день

• 10 000 курьеров онлайн

• Время доставки < 45 минут

Ключевые требования:

• Доступность: 99.99%

• RPS пиковый: 50 000

• Latency API: < 200 мс (p95)

• Трекинг курьеров: < 3 сек

Слайд 3: Контекст (C4 Level 1)

[Вставка диаграммы контекста]

Слайд 4: Контейнеры (C4 Level 2)

[Вставка диаграммы контейнеров]

Слайд 5: Технологический стек

Компонент	Технология	Обоснование
API Gateway	Kong	Высокая производительность, плагины, LUA
Сервис заказов	FastAPI (Python)	Быстрая разработка, async
Матчинг	Go	Высокая производительность, конкурентность
Трекинг	Node.js	WebSocket, событийная модель
Гео-хранилище	Redis	GeoSet, in-memory, < 1 мс latency
БД заказов	PostgreSQL (шарды)	ACID, транзакции, надёжность
БД меню	MongoDB	Гибкая схема, денормализация

Компонент	Технология	Обоснование
Очередь	Kafka	Долговременное хранение, replay

Слайд 6: Масштабирование и отказоустойчивость

Масштабирование:

• Каталог: горизонтальное (5 → 20 реплик в пик)

• Заказы: шардирование по user_id (4 → 8 шардов)

• Корзина: Redis Cluster (3 мастер-узла)

Отказоустойчивость:

• Active-Active в регионах

• PostgreSQL: Patroni + automatic failover

• Redis: Sentinel / Cluster failover

• Kafka: replication factor = 3

Слайд 7: Управление данными

PostgreSQL (заказы):

• Шардирование по user_id (HASH, 4 шарда)

• Read replicas для отчётности

• WAL архивация для PITR

Redis (гео-координаты):

• GeoSet: couriers:{region}

• TTL: 10 секунд (очистка старых записей)

• Репликация master-slave

MongoDB (меню):

• Шардирование по restaurant_id

• Денормализация: вложенные блюда

Слайд 8: Асинхронность

Kafka топики:

• order.created (заказ создан)

• payment.processed (платёж проведён)

• courier.assigned (курьер назначен)

• delivery.started / completed

Почему Kafka?

• Сохранение порядка (ключ = order_id)

• Replay для восстановления

• Consumer groups (масштабирование)

Слайд 9: Наблюдаемость

Метрики (Prometheus):

• RPS по эндпоинтам

• p99 latency

• Hit rate (Redis)

• Lag consumer (Kafka)

Логи (ELK):

• Структурированные логи (JSON)

• Correlation ID через все сервисы

Трассировка (Jaeger):

• Анализ задержек между сервисами

• Дашборды в Grafana

Слайд 10: ADR (пример)

ADR-001: Redis Geo для трекинга

• Плюс: 100k+ RPS, 1-2 мс latency

• Минус: потеря последней позиции при отказе

ADR-002: Шардирование заказов по user_id

• Плюс: горизонтальное масштабирование

• Минус: сложные глобальные отчёты

ADR-003: Kafka для событий

• Плюс: replay, порядок, масштабируемость

• Минус: сложность настройки кластера

Слайд 11: Компромиссы и риски

Основные компромиссы (Trade-offs):

• MongoDB (гибкость) vs PostgreSQL (консистентность) → гибрид

• Redis (latency) vs durability → TTL и репликация

• Шардирование (масштаб) vs сложность JOIN → денормализация

Риски и митигация:

• R-1: Отказ Redis Geo → 2-я реплика, автоматический failover

• R-2: Kafka consumer lag → мониторинг + авто-масштабирование

• R-3: Перегрузка PostgreSQL в пик → шардирование + read replicas

Слайд 12: Расчёт ресурсов и TCO

Обычный день (Москва):

• API Gateway: 5 реплик × 2 vCPU, 4GB

• Сервис заказов: 10 реплик × 4 vCPU, 8GB

• PostgreSQL: 4 шарда × (8 vCPU, 32GB) × 3 реплики

• Redis Geo: 3 узла × (4 vCPU, 16GB)

• Kafka: 3 узла × (4 vCPU, 16GB, 2TB)

Пик (Чёрная пятница):

• Масштабирование всех компонентов ×2-×3

TCO (месяц, Yandex Cloud):

• Обычный день: ~$15 000

• Пик (5 дней): +$10 000

• Итого среднемесячный: ~$17 000

Слайд 13: Итоги и выводы

text

Что спроектировано:

• Гео-распределённая система доставки

• Поддержка 5 млн заказов в день

• Доступность 99.99%

Ключевые решения:

• Redis Geo для низкой задержки трекинга

• Шардирование заказов по user_id

• Kafka как центральная событийная шина

Планы развития:

• ML для оптимизации матчинга

• Автоматическое масштабирование на базе прометея

• Chaos Engineering тесты

Слайд 14: Спасибо за внимание

text

Вопросы?

Контакты:

email: ivanov@example.com

МЕТОДИЧЕСКИЕ УКАЗАНИЯ ДЛЯ ВЫПОЛНЕНИЯ ПРАКТИЧЕСКОЙ ЧАСТИ РАБОТЫ

Подготовка к защите:

1. **Завершение документации:**
   - Проверить полноту arc42-документации (12 разделов)
   - Убедиться, что все диаграммы актуальны
   - Проверить ADR (не менее 3 документов)
2. **Подготовка презентации:**
   - 10-15 слайдов
   - Не перегружать текст (ключевые тезисы)
   - Включить диаграммы C4
   - Подготовить «запасные слайды» для сложных вопросов
3. **Репетиция:**
   - Уложиться в 10-15 минут
   - Проговорить основные аргументы вслух
   - Попросить одногруппников задать каверзные вопросы
4. **Демонстрация кода/прототипа (опционально):**
   - Если реализован прототип — подготовить live demo
   - Если нет — скриншоты и архитектурные схемы

3. ВАРИАНТЫ ИТОГОВЫХ ПРОЕКТОВ

3.1. Базовый уровень (15 вариантов)

Требования: система на 1 регион, 3-5 микросервисов, базовые ADR

	Тип системы	**Ключевая особенность**
	Интернет-магазин (e-commerce)	Товары, корзина, заказы
	Система доставки (1 город)	Курьеры, трекинг в реальном времени
	Платформа бронирования (отели, билеты)	Конкурентное

	Тип системы	Ключевая особенность
		бронирование
	Система управления задачами (Task manager)	Пользователи, проекты, уведомления
	Платформа онлайн-кинотеатра	Каталог, рекомендации, стриминг
	CRM для малого бизнеса	Контакты, сделки, задачи
	Платформа блогов (Medium-клон)	Посты, комментарии, лайки
	Система вопросов и ответов (StackOverflow-клон)	Вопросы, ответы, голосования
	Платформа для онлайн-обучения (LMS)	Курсы, уроки, сертификаты
0	Маркетплейс услуг (YouDo-клон)	Заказы, отклики, рейтинги
1	Платформа для конференций	Расписание, спикеры, билеты
2	Система учёта личных финансов	Транзакции, категории, отчёты

	Тип системы	Ключевая особенность
3	Платформа для совместной работы (Trello-клон)	Доски, карточки, комментарии
4	Система заказа такси (1 город)	Водители, заказы, трекинг
5	Платформа для доставки еды (0 город)	Рестораны, корзина, курьеры

3.2. Средний уровень (15 вариантов)

Требования: 2-3 региона, 6-8 микросервисов, 3+ ADR, расчёт TCO

	Тип системы	Регионы	Сложность
	E-commerce с международной доставкой	RU, EU, US	Высокая
	Доставка еды с гео-шардированием	3 города РФ	Средняя
	Платформа бронирования (международная)	RU, EU, AS	Высокая
	CRM для enterprise	RU, KZ	Средняя
	Платформа онлайн-кинотеатра (глобальная)	RU, EU, US, BR	Высокая
	Система управления логистикой	RU, CN, KZ	Средняя
	Финтех-платформа (платежи)	RU, GE, AM	Высокая

	Тип системы	Регионы	Сложность
	Health-tech (телемедицина)	RU, BY	Средняя
	Ed-tech (обучение)	RU, UA, KZ	Средняя
0	HR-платформа	RU, KZ, BY	Средняя
1	Маркетплейс недвижимости	RU, GE, AM	Средняя
2	Система для фриланса	RU, UA, KZ	Средняя
3	Платформа для инвесторов	RU	Средняя
4	Система контроля версий документов	RU, BY	Средняя
5	Агрегатор новостей	RU, EU, US	Высокая

3.3. Продвинутый уровень (15 вариантов)

Требования: глобальная (5+ регионов), 10+ микросервисов, ML/Streaming, полный ATAM-анализ

№	Тип системы	Сложность	Особые требования
	Глобальная доставка (Delivery Hero)	Экстремальная	Гео-шардирование, Kafka, ML

№	Тип системы	Сложность	Особые требования
	E-commerce с персонализацией (Amazon-подобный)	Экстремальная	ML рекомендации, аналитика в реальном времени
	Платформа видео-стриминга (Twitch)	Экстремальная	Низкая задержка, CDN, чат
	Банковский core-banking	Критическая	ACID, 2PC, безопасность
	Система HFT (high-frequency trading)	Экстремальная	Низкая задержка (microseconds)
	IoT для умного города (1M устройств)	Экстремальная	MQTT, стриминг, time-series
	Платформа такси (Uber clone)	Экстремальная	Geo-шардирование, матчинг
	Маркетплейс криптовалют	Критическая	Безопасность, аудит, латентность
	Платформа автономного вождения (симулятор)	Экстремальная	Real-time, сенсоры, безопасность
10	Роботизированная доставка (дроны)	Экстремальная	Управление дронами, телеметрия
11	Система онлайн-голосования (на выборах)	Критическая	Безопасность, аудит, прозрачность

№	Тип системы	Сложность	Особые требования
12	Платформа видеоконференций (Zoom/Meet)	Экстремальная	Низкая задержка, серверная сторона
13	Система обработки платежей (Stripe-клон)	Критическая	Идемпотентность, высокие транзакции
14	Глобальный CDN (CloudFront-клон)	Экстремальная	Edge-серверы, кэширование, geo-routing
15	Платформа для вычислений (Serverless)	Экстремальная	Высокая масштабируемость, изоляция вызовов

4. КОНТРОЛЬНЫЕ ВОПРОСЫ (для защиты)

Архитектура высокого уровня

1. Какие атрибуты качества были приоритетными в вашем проекте?
2. Почему выбрана микросервисная архитектура, а не монолит?
3. Как вы обеспечиваете отказоустойчивость? Какие механизмы failover?

Масштабирование и данные

4. Как вы шардируете данные? По какому ключу? Почему?
5. Почему выбрана именно база данных для основного хранилища?
6. Как вы решаете проблему распределённых транзакций (Saga vs 2PC)?

Производительность

7. Какие стратегии кэширования вы используете?
8. Как вы измеряете производительность? Какие SLO и SLI?

9. Какие узкие места (bottlenecks) вы видите в своей архитектуре?

Безопасность

10. Как защищены данные в покое и в транзите?
11. Как обеспечивается аутентификация и авторизация?
12. Как защититься от DDoS-атак?

Отказоустойчивость

13. Каковы RTO и RPO для вашей системы?
14. Как вы восстанавливаетесь после сбоя целого дата-центра?
15. Какие сценарии тестирования отказоустойчивости (Chaos Engineering) вы планируете?

Разработка и эксплуатация

16. Как устроен CI/CD для вашей системы?
17. Как вы мониторите систему (метрики, логи, трассировка)?
18. Как часто вы выпускаете релизы? Как обеспечиваете zero-downtime?

Компромиссы (Trade-offs)

19. Какие компромиссы вы сделали между согласованностью и доступностью?
20. Какой компромисс был самым сложным, и почему выбрали именно так?

5. ТРЕБОВАНИЯ К ОТЧЕТУ (итоговому)

Итоговый отчёт должен объединять все предыдущие лабораторные работы в единый документ:

1. **Титульный лист**
2. **Введение:** цели, задачи, предметная область
3. **Анализ требований:**
   - Бизнес-требования
   - Технические требования (RPS, latency, SLA)
   - Атрибуты качества с приоритетами
4. **Архитектура:**

- C4 Level 1, 2, 3 диаграммы
- Диаграмма развёртывания
- Описание ключевых сценариев (sequence diagrams)

5. **Технологический стек:**
   - Таблица выбора технологий с обоснованием
   - Сравнение альтернатив
6. **Управление данными:**
   - Схема БД (SQL/NoSQL)
   - Шардирование, репликация, индексы
7. **Масштабирование и отказоустойчивость:**
   - Стратегии масштабирования (горизонтальное, шардирование)
   - Active-Passive / Active-Active
   - RTO, RPO, резервное копирование
8. **ADR (не менее 3 документов)**
9. **Наблюдаемость:**
   - Метрики (Prometheus)
   - Логи (ELK)
   - Трассировка (Jaeger)
10. **Безопасность:** Auth, TLS, Secrets
11. **Расчёт ресурсов и TCO**
12. **Риски и компромиссы:**
    - Таблица рисков с митигацией
    - Таблица trade-offs с обоснованием
13. **Заключение:** выводы, что удалось спроектировать, планы развития
14. **Приложения:** полные листинги кода прототипа (если есть)

6. КРИТЕРИИ ОЦЕНИВАНИЯ ЗАЩИТЫ

Критерий	Вес (макс. 30 баллов)	Оценка

Критерий	Вес (макс. 30 баллов)	Оценка
Полнота архитектурной документации (C4, схемы)	5	
Обоснованность технологических решений	5	
Качество ADR (3+ документов)	4	
Осознание компромиссов (trade-offs)	4	
Расчёт TCO и ресурсов	4	
Презентация (структура, наглядность)	4	
Ответы на вопросы (глубина понимания)	4	
Итого	**30**	

Шкала перевода баллов в оценку

Оценка	Базовый уровень	Средний уровень	Продвинутый уровень
5 (отлично)	27-30	25-30	23-30
4 (хорошо)	23-26	21-24	19-22
3 (удовлетворительно)	18-22	16-20	14-18
2 (неудовлетворительно)	<18	<16	<14

7. ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ

Программа	Назначение	Доступность в РФ
Markdown-редактор	Оформление отчёта	+
PlantUML / Mermaid	Диаграммы	+
Draw.io	Диаграммы	+
Презентационное ПО	Слайды	+
Git	Версионирование	+
Python / Go / Node.js (опционально)	Прототип	+
Docker (опционально)	Контейнеризация	+

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

«СЕВЕРО-КАВКАЗСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»

МЕТОДИЧЕСКИЕ УКАЗАНИЯ

по организации и проведению самостоятельной работы по дисциплине «Архитектура и проектирование высоконагруженных и отказоустойчивых систем» для студентов направления подготовки

09.04.04 Программная инженерия Направленность (профиль)

«Разработка, мониторинг и управление жизненным циклом инженерных систем»

Ставрополь 2026

ВВЕДЕНИЕ

Методические указания для самостоятельной работы по дисциплине «Архитектура и проектирование высоконагруженных и отказоустойчивых систем» предназначены для студентов магистратуры, обучающихся по направлению подготовки **09.04.04 «Программная инженерия»** (профиль **«Разработка, мониторинг и управление жизненным циклом инженерных систем»**).

Дисциплина «Архитектура и проектирование высоконагруженных и отказоустойчивых систем» является одной из ключевых дисциплин вариативной части образовательной программы и направлена на формирование у магистрантов системных знаний и практических навыков в области проектирования распределённых систем, выбора архитектурных паттернов, масштабирования, шардирования, репликации, обеспечения отказоустойчивости и анализа архитектурных решений.

Настоящие методические указания разработаны в соответствии с:

- Федеральным государственным образовательным стандартом высшего образования по направлению подготовки 09.04.04 «Программная инженерия»;
- Учебным планом направления подготовки 09.04.04 «Программная инженерия» (профиль «Разработка, мониторинг и управление жизненным циклом инженерных систем»);
- Рабочей программой дисциплины «Архитектура и проектирование высоконагруженных и отказоустойчивых систем»;

- Положением об организации образовательного процесса на основе рейтинговой системы оценки знаний студентов в ФГАОУ ВО «СКФУ».

Целью самостоятельной работы является закрепление и углубление теоретических знаний о высоконагруженных и отказоустойчивых системах, полученных на лекционных занятиях, а также формирование практических навыков проектирования архитектур, анализа компромиссов (CAP, ACID/BASE), реализации паттернов масштабирования, шардирования, репликации, кэширования и распределённых транзакций.

Задачи самостоятельной работы:

В процессе выполнения самостоятельной работы студент должен решить следующие задачи:

1. Изучить теоретический материал по темам дисциплины в соответствии с учебным планом (9 тем лекций).
2. Подготовиться к выполнению лабораторных работ (№1–18) и осмыслению их результатов.
3. Выполнить индивидуальный сквозной проект по проектированию архитектуры высоконагруженной системы (e-commerce или система доставки) с обоснованием всех решений (Лабораторная работа №16, №17, №18).
4. Сформировать комплексный отчёт, включающий диаграммы C4 model, документацию arc42, анализ ATAM, расчёт ресурсов и TCO, а также защиту проекта.
5. Подготовиться к промежуточной аттестации (экзамену) по дисциплине.
6. Развить навыки самостоятельного поиска и анализа научно-технической информации в области высоконагруженных распределённых систем.

Настоящие методические указания призваны помочь студентам в организации самостоятельной работы по дисциплине «Архитектура и проектирование высоконагруженных и отказоустойчивых систем». Систематическое выполнение всех видов самостоятельной работы, своевременная подготовка к лабораторным занятиям и выполнение сквозного проекта позволят успешно освоить дисциплину и сформировать необходимые профессиональные компетенции.

1. ОБЩАЯ ХАРАКТЕРИСТИКА САМОСТОЯТЕЛЬНОЙ РАБОТЫ

1.1. Цель самостоятельной работы

Целью самостоятельной работы является закрепление и углубление теоретических знаний, полученных на лекциях, а также формирование практических навыков проектирования высоконагруженных и отказоустойчивых систем, анализа архитектурных решений, выбора стратегий масштабирования и управления данными в распределённых системах.

1.2. Общая характеристика сквозного проекта

Сквозной проект выполняется студентом индивидуально в течение всего семестра параллельно с изучением дисциплины и выполнением лабораторных работ (№1–17). Защита проекта проводится на 18-м лабораторном занятии.

Тема проекта: «Проектирование архитектуры высоконагруженной и отказоустойчивой системы для предметной области [выбрать: e-commerce / доставка / бронирование / fintech]»

Результат проекта: Комплексный отчёт, включающий:

- Диаграммы C4 model (Level 1, 2, 3, Deployment)
- Документацию arc42 (12 разделов)

- Анализ ATAM (чувствительные точки, компромиссы, риски)
- ADR (Architecture Decision Record) — минимум 3 документа
- Расчёт ресурсов и TCO (Total Cost of Ownership)
- Презентацию для защиты

1.3. Формируемые компетенции

Компетенция	Индикаторы
ПК-5. Способен проектировать, планировать и проводить эксперименты по обеспечению и проверке устойчивости систем, анализировать их результаты, формулировать и внедрять корректирующие меры для повышения отказоустойчивости и устойчивости к сбоям	**ИД-1 ПК-5.** Определяет показатели устойчивого состояния (steady state) системы и формулирует гипотезы о ее поведении при различных типах отказов.
	ИД-2 ПК-5. Проектирует эксперименты по внедрению сбоев с контролируемой зоной поражения (blast radius).
	ИД-3 ПК-5. Проводит эксперименты по тестированию устойчивости с использованием инструментов chaos engineering.
	ИД-4 ПК-5. Анализирует результаты экспериментов, выявляет уязвимости и слабые места архитектуры.
	ИД-5 ПК-5. Разрабатывает и внедряет корректирующие меры по итогам экспериментов.

Компетенция	Индикаторы
	ИД-6 ПК-5. Проводит повторные эксперименты для подтверждения эффективности изменений.
	ИД-7 ПК-5. Интегрирует практики тестирования устойчивости в CI/CD-пайплайн.

1.4. Трудоёмкость самостоятельной работы

Вид работы	**Часы**
Изучение теоретического материала (9 тем)	20
Подготовка к лабораторным работам (№1–17)	35
Выполнение сквозного проекта (ЛР №16–18)	25
Подготовка к экзамену	10
Итого	**90**

2. МЕТОДИЧЕСКИЕ РЕКОМЕНДАЦИИ ПО ИЗУЧЕНИЮ ТЕОРЕТИЧЕСКОГО МАТЕРИАЛА

2.1. Темы для самостоятельного изучения

№	Тема	Литература	Часы

№	Тема	Литература	Часы
1	Введение в архитектуру высоконагруженных систем (CAP, ACID, BASE)	[1, гл. 1-2]	2
2	Паттерны масштабирования (горизонтальное, шардирование, CQRS)	[1, гл. 3-4]	2
3	Отказоустойчивость на уровне архитектуры (Active-Passive, Active-Active, георазпределение)	[2, гл. 5-6]	2
4	Сетевые паттерны и Load Balancing	[2, гл. 7]	2
5	Архитектура микросервисов (DDD, Saga, 2PC)	[3, гл. 4-6]	2
6	Базы данных в высоконагруженных системах (репликация, шардирование, NoSQL)	[1, гл. 5-6]	2
7	Производительность и оптимизация (кэширование, асинхронность)	[4, гл. 3-4]	2
8	Анализ архитектурных решений (ATAM, C4 model)	[5]	2
9	Мониторинг, наблюдаемость и инцидент-менеджмент	[6]	2
Итого			**18**

2.2. Рекомендуемая литература

Основная литература

1. Ньюмен, С. Создание микросервисов / С. Ньюмен. — 2-е изд. — СПб.: Питер, 2024. — 622 с.
2. Ричардсон, К. Микросервисы: паттерны разработки и рефакторинга / К. Ричардсон. — СПб.: Питер, 2024. — 544 с.
3. Клеппман, М. Проектирование систем, интенсивно работающих с данными / М. Клеппман. — Электронное издание (перевод сообщества Vonng). — Режим доступа: https://github.com/Vonng/ddia
4. Нюгор, М. Т. Release It! : проектирование и развертывание готовых к production программных систем / М. Т. Нюгор. — СПб.: Питер, 2025. — 400 с.

Дополнительная литература

5. Beyer, B. Site Reliability Engineering: как Google управляет production-системами / B. Beyer, C. Jones, J. Petoff, N. R. Murphy. — Режим доступа: https://sre.google/books (лицензия CC BY-NC-ND 4.0)
6. Majors, C. Observability Engineering / C. Majors, L. Fong-Jones, G. Miranda. — Режим доступа: https://github.com/observability-engineering (лицензия CC BY-ND)

2.3. Рекомендации по работе с литературой

- При изучении теоретического материала рекомендуется вести конспект в виде структурированных заметок.
- Для каждой темы выделить ключевые термины и составить глоссарий.
- Выполнять практические примеры из литературы в тестовой среде.
- Использовать официальную документацию как основной источник для выполнения лабораторных работ.

---

3. МЕТОДИЧЕСКИЕ УКАЗАНИЯ ПО ВЫПОЛНЕНИЮ ЛАБОРАТОРНЫХ РАБОТ

3.1. Структура лабораторного практикума

Лабораторный практикум состоит из 18 работ, объединённых в 3 тематических блока:

Блок	№ ЛР	Тема
Блок 1. Основы архитектуры и масштабирования	1–6	C4 model, stateless-сервисы, CAP-теорема, Circuit Breaker, API Gateway, монолит vs микросервисы
Блок 2. Управление данными и асинхронность	7–12	Репликация, шардирование, RabbitMQ, Redis, SQL vs NoSQL, Saga
Блок 3. Анализ, тестирование и защита проекта	13–18	ATAM, нагрузочное тестирование k6, arc42, e-commerce кейс, доставка кейс, защита проекта

3.2. Порядок выполнения лабораторных работ

1. Ознакомиться с теоретическим обоснованием работы.
2. Выбрать уровень сложности (базовый / средний / продвинутый).
3. Выбрать вариант задания в соответствии с номером, выданным преподавателем.
4. Разработать решение (код, конфигурации, диаграммы, отчёты).
5. Выполнить проверку в тестовой среде (Docker, локальный кластер, инструменты).
6. Оформить отчёт по установленной форме (Markdown).
7. Защитить работу перед преподавателем (демонстрация + ответы на вопросы).

3.3. Рекомендации по подготовке к лабораторным работам

- Изучить соответствующий раздел теоретического материала перед выполнением работы.
- Подготовить тестовую среду (установленные Docker, Python/Node.js, Redis, PostgreSQL, RabbitMQ, k6, PlantUML).
- Ознакомиться с примерами выполнения из методических указаний.
- При возникновении вопросов — консультироваться с преподавателем.

3.4. Рекомендации по выполнению сквозного проекта (ЛР №16–18)

Сквозной проект выполняется на протяжении всего семестра параллельно с лабораторными работами. Каждая лабораторная работа добавляет новый артефакт в проект.

Требования к проекту:

- Выбрать предметную область (e-commerce, система доставки, fintech, маркетплейс).
- Разработать диаграммы C4 model (Level 1, 2, 3, Deployment).
- Заполнить документацию arc42 (минимум разделы 1–7, 9, 10, 11).
- Провести ATAM-анализ (сценарии, чувствительные точки, компромиссы, риски).
- Разработать минимум 3 ADR (Architecture Decision Record).
- Выполнить расчёт ресурсов и TCO.
- Подготовить презентацию (10–15 слайдов).
- Представить итоговый отчёт и защитить проект.

4. ВОПРОСЫ ДЛЯ СОБЕСЕДОВАНИЯ ПО САМОСТОЯТЕЛЬНОМУ ИЗУЧЕНИЮ ЛИТЕРАТУРЫ

1. Сформулируйте теорему CAP. Какие три свойства она описывает?
2. В чём разница между ACID и BASE?
3. Какие стратегии шардирования вы знаете? Приведите примеры.
4. Что такое consistent hashing и в чём его преимущество?
5. Как выбирается ключ шардирования (shard key)?
6. В чём разница между синхронной и асинхронной репликацией?
7. Какие метрики RTO и RPO характеризуют?
8. Что такое паттерн Circuit Breaker и какие у него состояния?
9. Алгоритмы балансировки нагрузки: round-robin, least_conn, ip_hash.
10. Функции API Gateway.
11. Что такое Saga (хореография vs оркестрация)?
12. Что такое компенсирующая транзакция?
13. Стратегии кэширования: Cache-Aside, Write-Through, Write-Behind.
14. Какие политики вытеснения в Redis вы знаете?
15. Что такое «пробивание кэша» (thundering herd) и как его избежать?
16. Чем отличается монолитная архитектура от микросервисной?
17. Какие преимущества и недостатки у каждого подхода?
18. Что такое ATAM? Опишите этапы метода.
19. Что такое чувствительная точка (sensitivity point) и компромисс (trade-off)?
20. Что такое ADR и какие разделы он содержит?

21. Что такое C4 model? Опишите 4 уровня детализации.
22. Почему для high-load систем важна документация arc42?
23. Как выбрать базу данных: SQL или NoSQL? Какие критерии?
24. Каковы типичные значения RPS и latency для e-commerce платформы?
25. Как архитектура доставки еды отличается от классического e-commerce?

5. КРИТЕРИИ ОЦЕНИВАНИЯ КОМПЕТЕНЦИЙ

5.1. Критерии оценивания лабораторных работ

Оценка	**Критерии**
5 (отлично)	Работа выполнена в полном объёме, код/конфигурации работают без ошибок, студент демонстрирует понимание всех этапов, отвечает на дополнительные вопросы, отчёт оформлен по требованиям, диаграммы и документация полные.
4 (хорошо)	Работа выполнена полностью, но есть небольшие замечания (неоптимальная конфигурация, неполный отчёт). Студент отвечает на большинство вопросов.
3 (удовлетворительно)	Работа выполнена с помощью преподавателя, есть ошибки, которые студент исправляет с наводящими вопросами. Отчёт оформлен минимально.
2 (неудовлетворительно)	Работа не выполнена или выполнена неверно, студент не понимает принципов, не может объяснить свои действия.

5.2. Критерии оценивания сквозного проекта (ЛР №18)

Критерий	**Вес, %**	**5 (отлично)**	**4 (хорошо)**	**3 (удовлетворительно)**	**2 (неудовлетворительно)**
Полнота реализации архитектуры	20	Реализованы все 4 уровня C4, arc42 заполнена полностью	Реализованы 3 уровня C4, arc42 с пропусками	Реализованы 2 уровня C4, arc42 минимально	Менее 2 уровней C4, arc42 отсутствует
Качество ADR	15	3+ ADR, полное обоснование, сравнение альтернатив	2 ADR, обоснование есть	1 ADR, обоснование минимально	ADR отсутствуют
Анализ ATAM (сценарии, риски, trade-offs)	15	ATAM полный (сп, си, сц, компромиссы, риски)	ATAM частичный (3 из 5 компонентов)	ATAM минимальный (2 компонента)	ATAM не проведён
Расчёт ресурсов и TCO	10	Полный расчёт, таблица ресурсов, бюджет	Расчёт есть, но неполный	Минимальная оценка	Отсутствует

Критерий	**Вес, %**	**5 (отлично)**	**4 (хорошо)**	**3 (удовлетворительно)**	**2 (неудовлетворительно)**
		на месяц/пик			
Качество диаграмм C4	10	Все 4 уровня, PlantUML в коде, нотация корректна	3 уровня, нотация с ошибками	2 уровня, ошибки в нотации	Отсутствуют
Ответы на вопросы защиты	15	Уверенные, полные ответы, понимание trade-offs	Хорошие ответы, неполные обоснования	Минимальные ответы, нет понимания	Нет ответов
Качество отчёта и презентации	15	Отчёт полный, презентация готова, скриншоты есть	Отчёт с небольшими недочётами	Отчёт минимальный, презентация отсутствует	Отчёт отсутствует

5.3. Единая таблица для вставки в документ

Критерий	**Вес, %**	**5 (отлично)**	**4 (хорошо)**	**3 (удовл.)**	**2 (неуд.)**
Полнота архитектуры (C4 + arc42)	20	++	+	-	--
Качество ADR (3+/2/1/0)	15	++	+	-	--
Анализ ATAM (полный/частичный/мини м./нет)	15	++	+	-	--
Расчёт ресурсов и TCO	10	++	+	-	--
Качество диаграмм C4 (4/3/2/1 ур.)	10	++	+	-	--
Ответы на вопросы защиты	15	++	+	-	--
Качество отчёта и презентации	15	++	+	-	--

Расшифровка знаков:

- «++» — критерий выполнен полностью, без ошибок
- «+» — критерий выполнен с незначительными замечаниями
- «-» — критерий выполнен минимально или с серьёзными ошибками
- «--» — критерий не выполнен

5.4. Шкала перевода процента баллов в оценку

Процент баллов	**Оценка**

Процент баллов	**Оценка**
90–100%	5 (отлично)
75–89%	4 (хорошо)
60–74%	3 (удовлетворительно)
менее 60%	2 (неудовлетворительно)