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 «Междисциплинартный проектный практикум-4» для студентов
направления подготовки 09.03.04 Программная инженерия
Направленность (профиль)
«Инженерия сложных программных систем»

Ставрополь 2026

Содержание

Введение

Современная программная инженерия всё чаще сталкивается с задачами создания распределённых, высоконагруженных и отказоустойчивых систем. Микросервисная архитектура, оркестрация контейнеров (Kubernetes), асинхронные коммуникации (Kafka), облачные технологии и инструменты наблюдаемости (мониторинг, логирование, трейсинг) стали стандартом индустрии. Выпускник направления 09.03.04 «Программная инженерия» должен не только уметь писать код, но и проектировать, развёртывать, тестировать и сопровождать сложные программные системы в условиях реальных нагрузок.

Цель методических указаний

Закрепить теоретические знания и сформировать практические компетенции в области:

1. оркестрации контейнеров (Kubernetes, minikube, k3s);
2. микросервисной архитектуры, API Gateway и Service Discovery;
3. отказоустойчивости (Circuit Breaker, Retry, Timeout, Bulkhead);
4. асинхронного обмена сообщениями (Kafka, exactly-once доставка);
5. распределённого кэширования (Redis Cluster);
6. нагрузочного тестирования (k6, JMeter);
7. профилирования и оптимизации (async-profiler, py-spy);
8. наблюдаемости (Prometheus, Grafana, Loki, Jaeger);
9. безопасности (SAST, DAST, OWASP ZAP);
10. GitOps (ArgoCD);
11. резервного копирования (PostgreSQL PITR);
12. облачного развёртывания (Yandex Cloud, VK Cloud);
13. полного цикла разработки и защиты комплексного highload-проекта.

Структура курса

Методические указания включают 27 лабораторных работ, объединённых в логические модули:

Модуль	**Лабораторные работы**	**Ключевые темы**
Базовый K8s и микросервисы	№1–2	minikube, deployment, service, rolling update, API Gateway, Service Discovery
Защита и устойчивость	№3–7	Rate limiting, Circuit Breaker, Retry, Bulkhead, Resilience4j
Асинхронность и данные	№4–5, №8–9	Kafka (at-least-once, exactly-once), Redis Cluster, сессии, кэширование
Наблюдаемость	№10–12, №14, №13	k6, JMeter, профилирование, Prometheus, Grafana, Loki, Jaeger
Безопасность и CI/CD	№15–17	SAST, DAST, SonarQube, OWASP ZAP, GitHub Actions
Автомасштабирование	№18–19	HPA, KEDA (event-driven)
DevOps и инфраструктура	№20–23	GitOps (ArgoCD), бэкап PostgreSQL, Docker Compose, облачный деплой
Комплексный проект	№24–27	Маркетплейс (архитектура, бизнес-логика, нагрузка,

Модуль	Лабораторные работы	Ключевые темы
		защита)

Требования к выполнению

Каждая лабораторная работа содержит:

краткое теоретическое содержание;

пошаговые задачи;

15 индивидуальных вариантов (по уровню сложности и технологическому стеку);

контрольные вопросы для самопроверки и защиты.

Формат отчёта (по каждой работе):

Тема и цель.

Используемые технологии.

Ход выполнения с листингами кода/манифестов и скриншотами.

Ответы на контрольные вопросы.

Вывод (что освоено, какие проблемы возникли и как решены).

Рекомендуемое программное обеспечение

1. Контейнеризация: Docker Desktop, Podman.
2. Kubernetes: minikube, k3s, kubectl, Helm.
3. Языки: Java (Spring Boot), Python (FastAPI/Flask), Go (Gin).
4. Брокеры: Apache Kafka (KRaft или Zookeeper), RabbitMQ.
5. Базы данных: PostgreSQL, ClickHouse, Redis.
6. Gateway: Spring Cloud Gateway, Kong, Traefik.
7. Нагрузка: k6, JMeter.
8. Мониторинг: Prometheus, Grafana, Loki, Jaeger.
9. Безопасность: SonarQube, OWASP ZAP, Semgrep.
10. CI/CD: GitHub Actions, GitLab CI.

11. Облака: Yandex Cloud, VK Cloud (при наличии гранта).

Ожидаемые результаты

После выполнения всех работ студент сможет:

1. самостоятельно разворачивать и настраивать Kubernetes-кластер;
2. проектировать и реализовывать микросервисные приложения с отказоустойчивостью;
3. настраивать rate limiting, circuit breaker, retry, bulkhead;
4. использовать Kafka с гарантиями exactly-once;
5. проводить нагрузочное тестирование и профилирование;
6. настраивать мониторинг, логирование и трейсинг;
7. внедрять SAST/DAST в CI/CD;
8. управлять инфраструктурой через GitOps (ArgoCD);
9. развёртывать систему в облачном managed-Kubernetes;
10. защитить комплексный highload-проект с документацией (ADR, OpenAPI, SLO

Лабораторная работа №1.

Настройка Kubernetes кластера и деплой первого микросервиса

1. Этап: Развёртывание локального кластера Kubernetes и деплой stateless-приложения.

2. Цель: Освоить установку и базовую настройку Kubernetes (minikube/k3s), научиться создавать Deployment и Service, выполнять rolling update.

3. Задачи:

1. Установить minikube или k3s на локальную машину.
2. Создать Docker-образ простого веб-приложения (например, «Hello, K8s!» с выводом hostname).
3. Написать манифесты Deployment (3 реплики) и Service типа ClusterIP.
4. Развернуть приложение в кластере, проверить доступность через port-forward.
5. Выполнить rolling update (изменить версию образа), откатить версию.
6. Настроить Ingress (опционально - для доступа извне).

4. Краткое теоретическое содержание

Kubernetes (K8s) - платформа для оркестрации контейнеров.

Компонент	Назначение
Pod	Группа контейнеров (обычно один), минимальная единица деплоя.
Deployment	Управляет репликами Pod, rolling update, откатом, масштабированием.
Service	Обеспечивает стабильный сетевой доступ к Pod’ам (ClusterIP, NodePort, LoadBalancer).

Компонент	Назначение
Ingress	Проксирует внешний HTTP/HTTPS трафик в Service.
minikube	Локальный K8s-кластер для разработки.

Rolling update: постепенная замена старых Pod'ов новыми без простоя. Команды kubectl set image deployment/<name> <container>=<new-image> и kubectl rollout undo.

Манифесты (YAML): декларативное описание ресурсов K8s.

5. Индивидуальные задания (15 вариантов)

Все варианты выполняются на minikube или k3s.

№ вар	**Образ приложения**	**Порт**	**Количество реплик**	**Особенность**
1	nginx:1.25	80	3	Rolling update на 1.26
2	httpd:2.4	80	2	Добавить ConfigMap с index.html
3	python:3.9-slim (простой HTTP-сервер)	8080	3	Написать Dockerfile самому
4	node:18-alpine (Express)	3000	2	Настроить livenessProbe
5	golang:1.21 (net/http)	8080	3	Собрать multi-stage Dockerfile
6	nginx:1.25	80	4	Настроить ресурсные

№ вар	Образ приложения	Порт	Количество реплик	Особенность
				лимиты (CPU, memory)
7	httpd:2.4	80	2	Ingress с TLS (самоподписанный сертификат)
8	python:3.9-slim	8080	3	Добавить readinessProbe
9	node:18-alpine	3000	3	ConfigMap с переменными окружения
10	golang:1.21	8080	2	Настроить HPA по CPU (имитация нагрузки)
11	nginx:1.25	80	3	PersistentVolume для логов
12	httpd:2.4	80	3	Rolling update с задержкой (maxSurge, maxUnavailable)
13	python:3.9-slim	8080	4	Service типа NodePort, статический порт
14	node:18-alpine	3000	3	canary-деплой через две версии Deployment

№ вар	Образ приложения	Порт	Количество реплик	Особенность
15	golang:1.21	8080	3	Blue-green деплой (старый + новый Service)

6. Контрольные вопросы

1. Что такое Pod, Deployment, Service? Какие у них обязанности?
2. В чём разница между ClusterIP, NodePort и LoadBalancer?
3. Как выполнить rolling update и откат?
4. Как посмотреть логи пода, подключиться к нему (kubectl exec)?
5. Что такое livenessProbe и readinessProbe? Зачем нужны?
6. Что такое ConfigMap и Secret? Как их смонтировать в Pod?
7. Как задать ресурсные лимиты (requests/limits)? Почему это важно?
8. Как увеличить количество реплик (kubectl scale)?
9. Что делает команда kubectl port-forward?
10. Как посмотреть события кластера (kubectl describe, kubectl get events)?
11. Чем отличается minikube от полноценного кластера (EKS, GKE, Yandex Managed K8s)?
12. Как настроить Ingress для доступа к сервису извне?
13. Что такое Helm? Чем полезен для деплоя?
14. Как временно приостановить развёртывание (kubectl rollout pause)?
15. Что такое оператор (operator) в K8s? Для чего используется?

Лабораторная работа №2.

API Gateway + Service Discovery

1. Этап: настройка API Gateway и Service Discovery в микросервисной архитектуре.

2. Цель: Освоить настройку API Gateway (Spring Cloud Gateway / Kong) для маршрутизации запросов и Service Discovery (Consul / Eureka) для динамического обнаружения сервисов.

3. Задачи:

1. Поднять Service Discovery (Consul или Eureka) в Docker/K8s.
2. Зарегистрировать два микросервиса (например, «Каталог» и «Корзина») в Service Discovery.
3. Настроить API Gateway, который маршрутизирует запросы /catalog/** и /cart/** в соответствующие сервисы.
4. Добавить фильтр (AddRequestHeader или логирование) на Gateway.
5. Убедиться, что Gateway автоматически переключается на работающую реплику при отказе одной из реплик.
6. Реализовать балансировку нагрузки (Round Robin / Weighted) на Gateway.

4. Краткое теоретическое содержание

Компонент	Назначение
API Gateway	Единая точка входа для всех клиентов, маршрутизация, агрегация запросов, аутентификация, rate limiting.
Service Discovery	Регистрация и обнаружение экземпляров микросервисов (адрес, порт, health status).
Consul	Service Discovery + KV-хранилище + health checks (агенты, серверы, интерфейс DNS).
Eureka	Service Discovery (Netflix), регистрация, аренда, пульс, защитный режим (self-preservation).
Spring Cloud Gateway	Реактивный API Gateway на Spring WebFlux, маршруты (route, predicate, filter).

Типовой маршрут в Spring Cloud Gateway:

yaml

```yaml
spring:
  cloud:
    gateway:
      routes:
        - id: catalog_service
          uri: lb://catalog-service    # lb:// = load balancer через Service Discovery
          predicates:
            - Path=/catalog/**
          filters:
            - AddRequestHeader=X-Service-Id, catalog
        - id: cart-service
          uri: lb://cart-service
          predicates:
            - Path=/cart/**
```

5. Индивидуальные задания (15 вариантов)

Все варианты предполагают работу с API Gateway и Service Discovery.

№ вар	Gateway	Service Discovery	Фильтр / особенность	Язык (бэкенд)
	Spring Cloud Gateway	Consul	AddRequestHeader	Java (Spring Boot)
	Spring Cloud Gateway	Eureka	RewritePath	Java (Spring Boot)

№ вар	Gateway	Service Discovery	Фильтр / особенность	Язык (бэкенд)
	Kong (DB-less)	Consul	Rate Limiting	Python (FastAPI)
	Kong (DB-less)	Consul	JWT-аутентификация	Go (Gin)
	Traefik	Consul	CORS	Python (FastAPI)
	Traefik	etcd	StripPrefix	Java (Spring Boot)
	Spring Cloud Gateway	Consul	Circuit Breaker	Java (Spring Boot)
	Spring Cloud Gateway	Eureka	Retry	Java (Spring Boot)
	Kong (с БД)	Consul	Request Transformer	Python (Django)
0	Kong (с БД)	Consul	Logging	Node.js (Express)
1	Traefik	Consul	Rate Limiting + IP whitelist	Go (Gin)

№ вар	Gateway	Service Discovery	Фильтр / особенность	Язык (бэкенд)
2	Spring Cloud Gateway	Consul	весовая балансировка (Weighted)	Java (Spring Boot)
3	Spring Cloud Gateway	Eureka	кастомный фильтр (логирование времени)	Java (Spring Boot)
4	Kong (DB-less)	Consul	Canary (10% трафика на новую версию)	Python (FastAPI)
5	Traefik	Consul	ForwardAuth (авторизация через внешний сервис)	Java (Spring Boot)

6. Контрольные вопросы

1. Зачем нужен API Gateway в микросервисной архитектуре?
2. Чем client-side discovery отличается от server-side discovery?
3. Как Consul проверяет здоровье сервисов? (health checks)
4. Что такое «защитный режим» (self-preservation) в Eureka?
5. Как Gateway использует Service Discovery для балансировки нагрузки (lb://)?
6. Что такое фильтр (filter) в Spring Cloud Gateway? Виды фильтров (Global, GatewayFilter).
7. Как добавить кастомный заголовок в каждый запрос через Gateway?
8. Как в Kong настроить маршрут с приоритетом?
9. Что такое «взвешенный Round Robin» и зачем он нужен?
10. Как Gateway обрабатывает ошибки при недоступности сервиса?
11. Чем отличается Traefik от Spring Cloud Gateway по модели настройки?

12. Можно ли обойтись без Service Discovery, используя только статические адреса? Какие минусы?
13. Как Consul синхронизирует состояние между своими агентами?
14. Что такое «сторонний» сервис-дискавери (например, etcd) и когда его используют?
15. Как в Kong реализовать canary-релиз (процент трафика)?

Лабораторная работа №3.

Rate Limiting (Token Bucket / GCRA) в API Gateway

1. Этап: Ограничение частоты запросов к API (rate limiting) на уровне Gateway.

2. Цель: Освоить реализацию rate limiting с использованием алгоритмов Token Bucket и GCRA, настроить лимиты для различных клиентов (по IP, по пользователю, по эндпоинту).

3. Задачи:

1. Развернуть API Gateway (Kong / Spring Cloud Gateway / Nginx).
2. Настроить rate limiting с использованием Redis (централизованное хранилище счётчиков).
3. Реализовать алгоритм Token Bucket (или GCRA) с ограничением RPS и burst.
4. Проверить, что при превышении лимита Gateway возвращает HTTP 429 (Too Many Requests).
5. Добавить разные лимиты для разных маршрутов (например, публичные – 100 req/min, авторизованные – 1000 req/min).
6. Выполнить нагрузочный тест (wrk / bombardier) для проверки работы лимитера.

4. Краткое теоретическое содержание

Алгоритм	**Принцип**	**Особенности**
Token Bucket	Ведро вместимостью B, пополняется с скоростью r токенов/сек. Запрос забирает 1 токен (или N).	Позволяет burst (кратковременное превышение

Алгоритм	Принцип	Особенности
		лимита).
GCRA	Время следующего разрешённого запроса (TAT). TAT = max(now, TAT) + T. Отклоняем если now < TAT - τ.	O(1) память, детерминированный, точный.
Fixed Window	Окно фиксированной длины (1 минута). Обнуляется в начале окна.	Проблема границ: burst в конце одного окна и начале следующего.
Sliding Window (счётчик)	Доля текущего окна + прерывание предыдущего окна.	Компромисс между точностью и памятью.

Token Bucket в Redis (Lua-скрипт):

```lua
local key = KEYS[1]
local rate = tonumber(ARGV[1])    -- r (токенов/сек)
local capacity = tonumber(ARGV[2]) -- B
local now = tonumber(ARGV[3])
local requested = tonumber(ARGV[4]) -- сколько токенов забираем (обычно 1)

local data = redis.call('HMGET', key, 'last_refill', 'tokens')
local last_refill = tonumber(data[1]) or now
local tokens = tonumber(data[2]) or capacity
```

```
    local delta = math.max(0, now - last_refill)
    local refill = delta * rate
    tokens = math.min(capacity, tokens + refill)

    if tokens < requested then
        return 0
    else
        redis.call('HMSET', key, 'last_refill', now, 'tokens', tokens
- requested)
        return 1
    end
```

5. Индивидуальные задания (15 вариантов)

№ вар	Gateway	Алгоритм	Хранилище	Тип лимита
	Kong	Token Bucket	Redis	по IP
	Kong	GCRA	Redis	по API-ключу
	Nginx + Lua	Token Bucket	Redis	по пользователю (JWT)
	Spring Cloud Gateway	Token Bucket	Redis	по маршруту
	Kong	Sliding Window	Redis	по IP + эндпоинт

№ вар	Gateway	Алгоритм	Хранилище	Тип лимита
	Kong	Fixed Window	Redis	по IP (burst)
	Nginx + Lua	GCRA	Redis	по IP + header X-User-Id
	Spring Cloud Gateway	GCRA	Redis	по API-ключу
	Kong	Token Bucket	in-memory	по пользователю (JWT)
0	Kong	GCRA	in-memory	по маршруту
1	Nginx + Lua	Token Bucket	Redis	лимиты для разных маршрутов (public / private)
2	Spring Cloud Gateway	Token Bucket	Redis	кастомный ключ (IP + метод + эндпоинт)
3	Kong	GCRA	Redis	с записью в лог при превышении
	Kong	Token	Redis + Local	гибрид (L1 –

№ вар	Gateway	Алгоритм	Хранилище	Тип лимита
4		Bucket	cache	локальный, L2 – Redis)
5	Nginx + Lua	Sliding Window	Redis	весовые лимиты (разный вес для разных методов)

6. Контрольные вопросы

1. Чем отличается Token Bucket от Leaky Bucket?
2. В чём преимущество GCRA перед Sliding Window?
3. Почему для rate limiting в распределённой системе нужно централизованное хранилище (Redis)?
4. Как обеспечить атомарность обновления счётчика в Redis?
5. Что такое burst и как его настроить в Token Bucket?
6. Какой код HTTP возвращается при превышении лимита?
7. Как добавить заголовки X-RateLimit-Limit, X-RateLimit-Remaining, X-RateLimit-Reset?
8. Как реализовать rate limiting для авторизованных пользователей (без передачи IP, только по JWT)?
9. Как влияет на лимитирование использование нескольких инстансов Gateway?
10. Что такое «проблема границ» (boundary problem) в Fixed Window?
11. Можно ли использовать Redis + Lua без блокировок? Как Lua помогает атомарности?
12. Как провести нагрузочное тестирование rate limiter’a?
13. Как отличить клиентскую ошибку (429) от атаки DoS?
14. Как динамически менять лимиты без перезапуска Gateway?
15. Что такое cluster-aware rate limiting в Kong?

Лабораторная работа №4.

Асинхронное взаимодействие через Kafka (producer / consumer)

1. Этап: Асинхронный обмен сообщениями между микросервисами с использованием Apache Kafka.

2. Цель: Настроить Kafka, реализовать producer для отправки событий и consumer для их обработки с гарантией доставки at-least-once.

3. Задачи:

1. Развернуть Kafka (один брокер, Zookeeper / KRaft) в Docker.
2. Создать топик orders с 3 партициями.
3. Реализовать producer на Spring Boot / Python / Go, отправляющий события о создании заказа.
4. Реализовать consumer, читающий события и логирующий их.
5. Настроить consumer group для параллельной обработки.
6. Обеспечить гарантию at-least-once (commit offset после обработки).
7. Запустить несколько экземпляров consumer и убедиться, что сообщения распределяются по партициям.

4. Краткое теоретическое содержание

Понятие	Описание
Топик	Логический канал, разделённый на партиции.
Партиция	Упорядоченная и неизменяемая последовательность сообщений.
Producer	Отправляет сообщения в топик (указывает ключ для выбора партиции).
Consumer	Читает сообщения из топика в составе consumer group.
Consumer group	Группа консьюмеров, читающих топик параллельно (каждая партиция закреплена за одним consumer).

Понятие	Описание
Offset	Уникальный идентификатор сообщения внутри партиции.
Гарантии доставки	at-most-once (автокоммит offset до обработки); at-least-once (коммит после обработки); exactly-once (транзакции, идемпотентность).

Producer (Spring Boot):

```
java
@Service
public class OrderEventProducer {
    @Autowired
    private KafkaTemplate<String, String> kafkaTemplate;

    public void sendOrderCreated(Order order) {
        kafkaTemplate.send("orders", order.getId(), order.toString());
    }
}
```

Consumer (Spring Boot):

```
java
@Component
public class OrderConsumer {
    @KafkaListener(topics = "orders", groupId = "order-processors")
    public void consume(String message) {
        // Обработка сообщения
        System.out.println("Получено: " + message);
    }
}
```

5. Индивидуальные задания (15 вариантов)

№ вар	Язык / платформа	Формат сообщения	Гарантия доставки	Дополнительно
1	Java (Spring Boot)	JSON (Jackson)	at-least-once	логирование ошибок
2	Java (Spring Boot)	Avro (схема)	at-least-once	Kafka Registry
3	Python (kafka-python)	JSON	at-least-once	ручной commit offset
4	Python (aiokafka)	JSON	at-least-once	асинхронный consumer
5	Go (Sarama)	JSON	at-least-once	consumer group
6	Java (Spring Boot)	Protobuf	at-least-once	запуск нескольких consumer (2–3)
7	Java (Spring Boot)	JSON	at-most-once	авто-коммит
8	Python (kafka-python)	Avro	at-least-once	Dead Letter Queue (имитация)
9	Go (Sarama)	Protobuf	at-least-once	partitioner кастомный
10	Java (Spring	JSON	at-least-once	Consumer rebalance

№ вар	Язык / платформа	Формат сообщения	Гарантия доставки	Дополнительно
	Boot)			listener
11	Python (aiokafka)	JSON + схема	at-least-once	обработка ошибок (retry)
12	Java (Spring Boot)	JSON	at-least-once	метрики (Prometheus)
13	Go (Sarama)	JSON	at-least-once	graceful shutdown
14	Java (Spring Boot)	Avro (без Registry)	at-least-once	два топика (orders, shipments)
15	Python (kafka-python)	JSON	at-least-once	запись offset в БД (не в Kafka)

6. Контрольные вопросы

1. Что такое партиция и зачем они нужны в Kafka?
2. Как consumer group влияет на распределение партиций?
3. Чем отличается at-least-once от at-most-once?
4. Как committed offset сохраняется? Где хранится?
5. Что происходит при ребалансировке (rebalance) consumer group?
6. Как обеспечить обработку сообщений без дублей?
7. Как выбрать количество партиций для топика?
8. Как producer выбирает партицию для сообщения (стратегии)?
9. Что такое ключ (key) сообщения и зачем он нужен?
10. Как масштабировать consumer’ы? Что будет, если consumer’ов больше, чем партиций?
11. Как обрабатывать сообщения, которые consumer не может обработать (DLQ)?
12. Как влияет retention на хранение сообщений в Kafka?
13. Как мониторить lag (отставание) consumer’а?
14. Как в Java-консьюмере остановить бесконечный цикл обработки?

15.Чем отличается ручной commit offset от автоматического?

Лабораторная работа №5.

Exactly-once доставка в Kafka

1. Этап: Идемпотентный producer и транзакции в Kafka для исключения дублирования сообщений.

2. Цель: Реализовать exactly-once доставку через идемпотентный producer и транзакционную запись с commit offset.

3. Задачи:

1. Включить enable.idempotence=true для producer.
2. Убедиться, что повторная отправка с одинаковым ключом и последовательностью не создаёт дубликат.
3. Реализовать транзакционную запись: initTransaction → send → commit (или abort).
4. Настроить consumer с isolation.level=read_committed.
5. Продемонстрировать, что сообщения из прерванной транзакции не читаются.
6. Дополнительно: транзакционно обновить consumer offset (sendOffsetsToTransaction).

4. Краткое теоретическое содержание

Идемпотентный producer (enable.idempotence=true): Каждый producer имеет PID (producer id) и sequence number для каждой партиции. Брокер отбрасывает сообщение, если sequence number не увеличился ровно на 1.

Транзакции:

initTransactions() – инициализация транзакционного producer.

beginTransaction() – начало транзакции.

send() – отправка сообщений (внутри транзакции).

commitTransaction() – фиксация (сообщения становятся видимыми для read_committed consumer'ов).

abortTransaction() – отмена (сообщения удаляются).

Consumer с транзакциями: isolation.level=read_committed (по умолчанию read_uncommitted). Consumer не увидит сообщения незафиксированной транзакции.

5. Индивидуальные задания (15 вариантов)

№ вар	**Язык/платформа**	**Транзакционная операция**	**Особенность**
1	Java (Spring Boot)	send + commit offset	Простая отправка заказа
2	Java (Spring Boot)	send, abort по ошибке	Имитация ошибки → abort
3	Python (kafka-python)	send + commit offset	Идемпотентность через ключ сообщения
4	Python (aiokafka)	send (транзакция)	Асинхронный producer
5	Go (Sarama)	send + commit offset	Транзакции через Sarama
6	Java (Spring Boot)	send в два топика в одной транзакции	orders и payments
7	Java (Spring Boot)	sendOffsetsToTransaction	Обновление offset внутри транзакции

№ вар	Язык/платформа	Транзакционная операция	Особенность
8	Python (kafka-python)	send + abort	Демонстрация read_committed vs read_uncommitted
9	Java (Spring Boot)	send с пользовательским transaction.id	Повторный запуск producer с тем же ID
10	Go (Sarama)	send в несколько топиков	Транзакционное чтение из order-events
11	Java (Spring Boot)	send, abort + retry	Обработка ошибок транзакции
12	Python (aiokafka)	send + commit offset + DLQ	Имитация DLQ для ошибочных сообщений
13	Java (Spring Boot)	транзакция с несколькими producer	Два разных producer (разные transaction.id)
14	Java (Spring Boot)	send + consumer с read_committed	Проверка, что abort-сообщения не видны
15	Go (Sarama)	send + commit offset +	Сравнение

№ вар	Язык/платформа	Транзакционная операция	Особенность
		idempotent	with/without idempotence

6. Контрольные вопросы

1. Какие два механизма обеспечивают exactly-once в Kafka?
2. Как работает идемпотентный producer? Что такое PID и sequence number?
3. Зачем нужен transactional.id?
4. Какие гарантии даёт идемпотентность при ретраях?
5. Как consumer должен быть настроен для чтения только зафиксированных транзакций?
6. Что произойдёт, если producer упадёт после commit? Как восстановить?
7. Влияют ли транзакции на производительность?
8. Можно ли использовать транзакции между разными producer'ами?
9. Как проверить, что сообщение было записано только один раз?
10. Как обрабатывать дубли на стороне consumer (идемпотентность бизнес-логики)?
11. Как мониторить транзакции (активные, завершённые)?
12. Можно ли сочетать идемпотентный producer и транзакции?
13. Как работает sendOffsetsToTransaction?
14. Что такое read_committed vs read_uncommitted на практике?
15. Что такое фантомные сообщения и почему read_committed их не решает?

Лабораторная работа №6.

Circuit Breaker + Retry + Timeout (Resilience4j)

1. Этап: Защита микросервиса от каскадных отказов внешних зависимостей.

2. Цель: Реализовать Circuit Breaker (три состояния: CLOSED, OPEN, HALF-OPEN) с настраиваемым порогом ошибок, добавить Retry с exponential backoff и Timeout, настроить fallback.

3. Задачи:

1. Написать микросервис, вызывающий нестабильный внешний API (например, погоду или платежи).
2. Включить Circuit Breaker: при 3 ошибках за 10 запросов перейти в OPEN на 5 секунд.
3. Добавить Retry: до 3 попыток, экспоненциальная задержка (1, 2, 4 сек).
4. Установить Timeout (500 ms) для вызова внешнего API.
5. Реализовать fallback (запасной ответ: статический или из кэша).
6. Протестировать поведение при сбоях (отключении внешнего API, долгом ответе, падении).
7. Экспортировать метрики Circuit Breaker (состояние, количество ошибок) в Prometheus.

4. Краткое теоретическое содержание

Компонент	**Описание**
Circuit Breaker	Защита от каскадных отказов. Состояния: CLOSED (запросы проходят, при достижении порога → OPEN), OPEN (запросы блокируются, fallback), HALF-OPEN (проверка восстановления).
Retry	Повторная попытка при временных ошибках. Экспоненциальный backoff + jitter (случайность) для разнесения пиков.
Timeout	Ограничение максимального времени ожидания ответа.
Bulkhead	Ограничение количества параллельных вызовов (изоляция пулов).

Компонент	Описание
Fallback	Запасной ответ (статика, кэш, предопределённый ответ).

Пример конфигурации Resilience4j (Spring Boot yml):

```yaml
resilience4j:
  circuitbreaker:
    instances:
      payment-service:
        sliding-window-size: 10
        failure-rate-threshold: 50
        wait-duration-in-open-state: 5s
  retry:
    instances:
      payment-retry:
        max-attempts: 3
        wait-duration: 1s
        exponential-backoff-multiplier: 2
  timelimiter:
    instances:
      payment-timeout:
        timeout-duration: 500ms
```

5. Индивидуальные задания (15 вариантов)

№ вар	Язык	Внешний сервис (имитация)	Параметры CB	Retry	Fallback
1	Java (Spring	Платежи (1 успех /	3 ошибки	3 попытки	Статика

№ вар	Язык	Внешний сервис (имитация)	Параметры CB	Retry	Fallback
	Boot)	2 ошибки)	→ OPEN (5 сек)		
2	Java (Spring Boot)	Погода (таймаут 1с)	50% ошибок → OPEN	2 попытки	Кэш последнего успеха
3	Java (Spring Boot)	Курсы валют (разные ошибки)	4 ошибки → OPEN (3 сек)	3 попытки	Статика + логирование
4	Python (resilience4j-py)	Платежи (HTTP 500)	30% ошибок → OPEN	Экспоненциальный	Кэш
5	Python (resilience4j-py)	Погода (socket timeout)	40% ошибок → OPEN (2 сек)	2 попытки	Статика

№ вар	Язык	Внешний сервис (имитация)	Параметры CB	Retry	Fallback
6	Go (gobreaker)	Платежи	5 ошибок → OPEN	3 попытки	Кэш
7	Go (gobreaker)	Погода	4 ошибки → OPEN	2 попытки	Статика
8	Java (Spring Boot)	Два внешних сервиса	CB отдельно на каждый	разные	комбинированный
9	Java (Spring Boot)	Платежи + логирование перехода состояния	3 ошибки → OPEN	3 попытки	Статика + Prometheus
10	Java (Spring	Платежи + экспорт	50% ошибок	3 попытки	Кэш (Redis)

№ вар	Язык	Внешний сервис (имитация)	Параметры CB	Retry	Fallback
	Boot)	метрик	→ OPEN		
11	Python (resilience4j-py)	Платежи + retry with jitter	30% ошибок → OPEN	5 попыток	Статика
12	Go (gobreaker)	Платежи + Timeout (2s)	3 ошибки → OPEN	3 попытки	Кэш
13	Java (Spring Boot)	Платежи + Bulkhead (max=2)	3 ошибки → OPEN	3 попытки	Статика
14	Java (Spring Boot)	Платежи + RateLimiter (5 запросов/сек)	3 ошибки → OPEN	3 попытки	Кэш + статика

№ вар	Язык	Внешний сервис (имитация)	Параметры CB	Retry	Fallback
15	Java (Spring Boot)	Платежи + комбинация всех паттернов (CB, Retry, Bulkhead, RateLimiter)	3 ошибки → OPEN	3 попытки	Статика + запись в лог

6. Контрольные вопросы

1. Какие три состояния Circuit Breaker и как они работают?
2. Как настроить порог открытия CB (sliding window, failure rate)?
3. Зачем нужен wait-duration-in-open-state?
4. Что такое полуоткрытое состояние? Как происходит проверка восстановления?
5. Как Retry с экспоненциальным backoff + jitter помогает при массовых сбоях?
6. В чём опасность retry без ограничения максимального числа попыток?
7. Как таймаут влияет на Circuit Breaker? (запрос, не уложившийся в таймаут, считается ошибкой)
8. Как реализовать fallback? Где он должен храниться (кэш, статика, последний успешный ответ)?
9. Как мониторить состояние Circuit Breaker (метрики, алерты)?
10. Чем отличается Retry от Circuit Breaker по назначению?
11. Можно ли комбинировать Retry и Circuit Breaker? В каком порядке?

12. Как защитить систему от бесконечного потока вызовов в HALF-OPEN состоянии?
13. Что такое Bulkhead и зачем он нужен?
14. Как Timeout защищает ресурсы потока? Чем отличается @TimeLimiter от @Transactional timeout?
15. Как протестировать все переходы Circuit Breaker в unit-тестах (Mockito)?

Лабораторная работа №7.

Bulkhead и RateLimiter на уровне кода

1. Этап: Изоляция пулов ресурсов и ограничение частоты вызовов (Rate Limiter на уровне кода).

2. Цель: Реализовать Bulkhead (ограничение количества параллельных вызовов) и Rate Limiter (ограничение частоты вызовов) с использованием Resilience4j.

3. Задачи:

1. Написать сервис, вызывающий медленный внешний API (имитация задержки 1–2 секунды).
2. Ограничить количество параллельных вызовов к API с помощью SemaphoreBulkhead (максимум 3 вызова).
3. Настроить Rate Limiter: не более 10 вызовов в секунду.
4. При превышении лимитов выбрасывать исключение с последующим fallback.
5. Протестировать параллельные запросы (например, 20 одновременных вызовов) и убедиться, что часть запросов получает fallback.
6. Добавить метрики (количество отклонённых вызовов) в Prometheus.

4. Краткое теоретическое содержание

Механизм	Назначение	Пример
SemaphoreBulkhead	Ограничивает количество	Не более 3 параллельных

Механизм	Назначение	Пример
	одновременных вызовов (общий пул разрешений).	запросов.
ThreadPoolBulkhead	Выделяет отдельный пул потоков для изоляции.	Не блокирует основной пул (лучше, но ресурсоёмче).
RateLimiter	Ограничивает частоту вызовов (не более N запросов в секунду).	limitForPeriod=10, limitRefreshPeriod=1s.

Пример конфигурации:

yaml

```yaml
resilience4j:
  bulkhead:
    instances:
      external-api:
        max-concurrent-calls: 3
        max-wait-duration: 100ms
  ratelimiter:
    instances:
      external-api:
        limit-for-period: 10
        limit-refresh-period: 1s
        timeout-duration: 0
```

Java-код:

java

```java
@Bulkhead(name = "external-api", fallbackMethod = "fallback")
```

```
@RateLimiter(name = "external-api")
public String callExternalApi() { ... }
```

5. Индивидуальные задания (15 вариантов)

вар	Тип Bulkhead	Max параллельных вызовов	Rate Limiter (RPS)	Fallback
1	SemaphoreBulkhead	3	10	Статика
2	ThreadPoolBulkhead	3	10	Статика
3	SemaphoreBulkhead	5	20	Кэш
4	SemaphoreBulkhead	2	5	Кэш + логирование
5	ThreadPoolBulkhead	5	20	Статика + лог
6	SemaphoreBulkhead	3	10	Запись в очередь (имитация)
7	SemaphoreBulkhead	4	15	Кэш (Redis)
8	ThreadPoolBulkhead	4	15	Статика + метрики
9	SemaphoreBulkhead	3	10	Fallback с задержкой (retry)

вар	Тип Bulkhead	Max параллельных вызовов	Rate Limiter (RPS)	Fallback
10	ThreadPoolBulkhead	6	30	Статика
11	SemaphoreBulkhead	3	10	Два fallback (первичный + вторичный)
12	SemaphoreBulkhead	2	5	Кэш + отправка в DLQ (имитация)
13	ThreadPoolBulkhead	8	40	Статика + логирование
14	SemaphoreBulkhead	4	20	Кэш (Caffeine)
15	ThreadPoolBulkhead	3	10	Статика + Prometheus метрики

6. Контрольные вопросы

1. Чем SemaphoreBulkhead отличается от ThreadPoolBulkhead?
2. Когда следует использовать ThreadPoolBulkhead?
3. Как SemaphoreBulkhead влияет на основной пул потоков?
4. Что такое max-wait-duration и зачем оно нужно?
5. Как RateLimiter отличает свой лимит от лимита на Gateway?
6. Как тестировать параллельные вызовы с помощью параллельных потоков (CountDownLatch)?
7. Как комбинировать Circuit Breaker и Bulkhead?

8. Каковы метрики для Bulkhead (количество свободных разрешений, очередь)?
9. Как избежать голодания (starvation) при использовании Bulkhead?
10. Какие побочные эффекты при использовании ThreadPoolBulkhead (контекстный switch)?
11. Можно ли применять RateLimiter для защиты downstream сервиса?
12. Что произойдёт, если одновременно придут 100 запросов при max-parallel=3?
13. Как динамически изменить лимиты без перезапуска?
14. Как в Spring Boot экспортировать метрики Resilience4j в Prometheus?
15. Как в unit-тесте симулировать превышение лимита Bulkhead?

Лабораторная работа №8.

Распределённое кэширование с Redis Cluster

1. Этап: Стратегии кэширования (Cache-Aside, Read-Through, Write-Through) с использованием Redis Cluster.

2. Цель: Настроить Redis Cluster, реализовать кэширование данных с использованием паттерна Cache-Aside, добавить инвалидацию кэша при обновлении данных.

3. Задачи:

1. Запустить Redis Cluster (минимум 3 мастера) в Docker Compose.
2. Реализовать сервис, который сначала обращается к кэшу, при промахе читает из БД (PostgreSQL) и заполняет кэш.
3. Настроить TTL (time-to-live) для ключей (например, 5 минут).
4. При обновлении данных в БД инвалидировать (удалять) соответствующие ключи в Redis.
5. Измерить время ответа с кэшем и без кэша (нагрузочное тестирование).
6. Дополнительно: реализовать паттерн Write-Through (запись одновременно в БД и кэш).

4. Краткое теоретическое содержание

Стратегия	Описание	Плюсы / минусы
Cache-Aside	Приложение сначала ищет в кэше; при промахе читает из БД и записывает в кэш.	Простота; риск несогласованности.
Read-Through	Кэш сам загружает данные из БД при промахе (через специальный провайдер).	Прозрачность; нагрузка на кэш.
Write-Through	При записи сначала обновляется БД, затем кэш.	Согласованность; задержка записи.
Write-Behind	Запись сначала в кэш, потом асинхронно в БД.	Высокая скорость записи, риск потери данных.

Инвалидация кэша: при обновлении/удалении данных в БД – удалить (или обновить) соответствующий ключ в Redis.

Redis Cluster: автоматическое шардирование (хеш-слоты), репликация, failover.

5. Индивидуальные задания (15 вариантов)

№ вар	Стратегия кэширования	TTL	Инвалидация	Язык
1	Cache-Aside	5 мин	по ключу	Java (Spring

№ вар	Стратегия кэширования	TTL	Инвалидация	Язык
				Boot)
2	Cache-Aside	1 мин	по паттерну (удаление группы)	Java (Spring Boot)
3	Cache-Aside + Write-Through	10 мин	обновление кэша	Java (Spring Boot)
4	Read-Through (Spring Cache)	5 мин	@CacheEvict	Java (Spring Boot)
5	Cache-Aside	30 сек	по тегу (tag-based)	Python (FastAPI)
6	Write-Through	10 мин	удаление по ключу	Python (FastAPI)
7	Cache-Aside	2 мин	при обновлении -> инвалидация	Go
8	Read-Through	5 мин	по ключу + по шаблону	Java (Spring Boot)
9	Write-Behind	10	удаление через TTL	Java

№ вар	Стратегия кэширования	TTL	Инвалидация	Язык
	(асинхронный)	мин		(Spring Boot)
10	Cache-Aside + Local cache (Caffeine)	5 мин	двухуровневая инвалидация	Java (Spring Boot)
11	Cache-Aside	1 мин	инвалидация по событию (Kafka)	Java (Spring Boot)
12	Cache-Aside + Write-Through	5 мин	инвалидация + обновление	Python (FastAPI)
13	Cache-Aside	10 мин	инвалидация через паттерн keys (SCAN)	Go
14	Read-Through	2 мин	@CachePut для обновления	Java (Spring Boot)
15	Cache-Aside + статистика попаданий (hit ratio)	5 мин	–	Java (Spring Boot)

6. Контрольные вопросы

1. Что такое Cache-Aside? Нарисуйте диаграмму последовательности.

2. В чём отличие Read-Through от Cache-Aside?
3. Какие проблемы возникают при инвалидации кэша по паттерну?
4. Как гарантировать согласованность между БД и кэшем?
5. Что такое инвалидация по тегу (tag-based) и зачем она нужна?
6. Как TTL помогает избежать проблем с устаревшими данными?
7. Как Redis Cluster распределяет ключи по слотам?
8. Что происходит при отказе одного из мастеров в Redis Cluster?
9. Как измерить эффективность кэширования (hit ratio, latency)?
10. Что такое «горячий ключ» и как бороться?
11. Когда лучше использовать Write-Through, а когда Write-Behind?
12. Как в Spring Cache управлять TTL (@Cacheable + RedisCacheManager)?
13. Как избежать лавинного обновления кэша (thundering herd) при TTL?
14. Как в Redis реализовать SCAN для удаления ключей по паттерну (без KEYS)?
15. Как интегрировать Redis Cluster с Prometheus (экспорт метрик)?

Лабораторная работа №9.

Сессии и кэширование пользовательских данных

1. Этап: Управление сессиями пользователей в распределённой системе с использованием Redis.

2. Цель: Реализовать механизм хранения сессий в Redis, настроить инвалидацию сессий при выходе пользователя, сравнить с хранением сессий в памяти (один сервер) и с JWT.

3. Задачи:

1. Реализовать аутентификацию (логин/пароль).
2. При успешном входе генерировать уникальный session ID и сохранять в Redis (user-id, срок действия, данные пользователя).
3. Отправлять session ID клиенту (через cookie или заголовок).
4. На всех защищённых эндпоинтах проверять наличие сессии в Redis (middleware).
5. Реализовать logout (удаление сессии из Redis).

6. Запустить два экземпляра приложения с балансировщиком нагрузки и убедиться, что сессии работают (sticky sessions не нужны).
7. Сравнить с JWT (статистические данные, возможность отзыва).

4. Краткое теоретическое содержание

Механизм	Хранение	Отзыв	Масштабирование
Сессии в памяти	Память одного сервера	Ручное удаление	Не масштабируется
Сессии в Redis	Централизованное	DEL session:{id}	Да (горизонтально)
JWT	Клиент (токен)	Нет (до истечения TTL)	Да (stateless)

Redis-структура для сессии:

redis

SETEX session:{uuid} 3600 '{"userId": 123, "username": "ivan", "role": "USER"}'

Middleware проверки (Spring Boot):

java

```
@Component
public class SessionInterceptor implements HandlerInterceptor {
    @Autowired
    private RedisTemplate<String, String> redis;

    public boolean preHandle(HttpServletRequest request, ...) {
        String sessionId = request.getHeader("X-Session-Id");
```

```
        if (sessionId == null || !redis.hasKey("session:" + sessionId)) {
            throw new UnauthorizedException();
        }
        return true;
    }
}
```

5. Индивидуальные задания (15 вариантов)

вар	Язык/фреймворк	Хранение сессии	Время жизни сессии	Транспорт
1	Java Spring Boot	Redis	30 мин	Cookie
2	Java Spring Boot	Redis	1 час	Header X-Session-Id
3	Java Spring Boot	Redis + Local cache	15 мин	Cookie
4	Python (FastAPI)	Redis	1 час	Cookie
5	Python (Flask)	Redis	30 мин	Header
6	Go (Gin)	Redis	1 час	Cookie
7	Java Spring Boot	Redis + JWT как альтернатива	2 часа	Cookie + Header (сравнение)
8	Java Spring Boot	Redis + продление TTL при активности	1 час	Cookie

вар	Язык/фреймворк	Хранение сессии	Время жизни сессии	Транспорт
9	Java Spring Boot	Redis + отслеживание активных сессий пользователя	30 мин	Cookie
10	Python (FastAPI)	Redis + ограничение количества сессий на пользователя (макс 2)	1 час	Header
11	Go (Gin)	Redis + логирование входа/выхода	30 мин	Cookie
12	Java Spring Boot	Redis + обновление сессии (refresh) через специальный эндпоинт	1 час	Cookie
13	Java Spring Boot	Redis + Spring Session	30 мин	Cookie

вар	Язык/фреймворк	Хранение сессии	Время жизни сессии	Транспорт
14	Python (FastAPI)	Redis + Redis Cluster	1 час	Header
15	Java Spring Boot	Redis + двухфакторная сессия (первый фактор – пароль, второй – OTP)	15 мин	Cookie

6. Контрольные вопросы

1. Почему сессии в памяти не подходят для распределённой системы?
2. Как Redis помогает масштабировать сессии?
3. Как защитить session ID от кражи (HTTPS, HttpOnly, Secure флаги)?
4. Что такое session fixation attack и как защититься?
5. Как часто надо обновлять TTL сессии?
6. Как отозвать сессию администратором (через Redis)?
7. Чем отличаются сессии в Redis от JWT по безопасности?
8. Как ограничить количество активных сессий для одного пользователя?
9. Как добавить информацию о пользователе в сессию (роли, права)?
10. Как реализовать logout для всех устройств пользователя?
11. Что произойдёт, если Redis упадёт? (replica, sentinel, cluster)
12. Как в тестах имитировать сессию (MockHttpSession)?
13. Как хранить сессии при использовании Spring Session?
14. Как перенастроить приложение с сессий в памяти на Redis без изменения кода?

15. JWT vs сессии в Redis: что выбрать для высоконагруженного приложения?

Лабораторная работа №10.

Нагрузочное тестирование с k6 / JMeter

1. Этап: Оценка производительности API с помощью инструментов нагрузочного тестирования.

2. Цель: Научиться проводить нагрузочное тестирование, измерять RPS, p95/p99 latency и пропускную способность, анализировать узкие места.

3. Задачи:

1. Подготовить тестовый эндпоинт (GET /products, POST /orders, или любой другой).
2. Написать скрипт нагрузки на k6 (или JMeter) с параметризацией (разные пользователи, разные данные).
3. Провести тест с постепенным нарастанием нагрузки (ramp-up).
4. Замерить RPS, p95, p99 latency, количество ошибок (HTTP 5xx).
5. Проанализировать отчёт, найти узкое место (БД, кэш, CPU).
6. Повторить тест после оптимизации (добавление кэша, индекса).

4. Краткое теоретическое содержание

Метрики:

RPS (requests per second) – пропускная способность.

Latency (p50, p95, p99) – задержка. p99 означает: 99% запросов выполняются быстрее этого времени.

Error rate – процент ошибок (≥ 1% – плохо).

k6 (JS) пример скрипта:

javascript

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

export let options = {
```

```
  stages: [
    { duration: '30s', target: 50 },  // ramp-up до 50 VU
    { duration: '1m', target: 50 },   // стабильная нагрузка
    { duration: '10s', target: 0 },   // ramp-down
  ],
};

export default function () {
  let res = http.get('http://localhost:8080/products');
  check(res, { 'status is 200': (r) => r.status === 200 });
  sleep(1);
}
```

5. Индивидуальные задания (15 вариантов)

№ вар	**Целевой эндпоинт**	**Инструмент**	**Профиль нагрузки**	**Метрики для анализа**
1	GET /products	k6	ramp-up до 100 VU	RPS, p95, ошибки
2	POST /orders	k6	постоянная 50 VU	p99, ошибки
3	GET /products + параметры	JMeter	ступенчатый: 50,100,200	RPS, CPU
4	GET /search?q=term	k6	пик (100 VU → 300 VU → 100 VU)	p95, БД

№ вар	Целевой эндпоинт	Инструмент	Профиль нагрузки	Метрики для анализа
5	GET /products (кэшируемый)	k6	длительный (10 мин)	hit ratio, latency
6	POST /auth/login	k6	ramp-up до 50 VU	ошибки авторизации
7	GET /user/profile	JMeter	200 VU постоянная	p99, ошибки
8	GET /products + POST /cart	k6	смешанный (80% GET, 20% POST)	RPS, latency
9	GET /products (без кэша) vs с кэшем	k6	100 VU	сравнение p95
10	GET /products with DB	k6	ramp-up до 200 VU	RPS, загрузка БД
11	GET /products with Redis	k6	300 VU	hit ratio, redis latency
12	WebSocket (чат)	k6	100 одновременных соединений	сообщений/сек
13	GET /products	k6 (xk6-grpc)	50 VU	RPS, latency

№ вар	Целевой эндпоинт	Инструмент	Профиль нагрузки	Метрики для анализа
	(gRPC)			
14	POST /upload (файлы)	JMeter	20 VU	пропускная способность
15	GET /products + лимитирование	k6	200 VU, обход rate limiter	429 ошибки

6. Контрольные вопросы

1. Какие метрики являются ключевыми в нагрузочном тестировании?
2. Что означает p99 latency и почему он важен?
3. Как отличить узкое место: CPU, БД, сеть, IO?
4. Какой процент ошибок считается допустимым для production?
5. Как k6 моделирует виртуальных пользователей (VU)? Чем отличается от потоков?
6. Как в JMeter настроить ramp-up и ступенчатую нагрузку?
7. Как анализировать результаты теста: какие графики смотреть?
8. Как нагрузочное тестирование влияет на саму систему (искажение результатов)?
9. Что такое тест устойчивости (soak test) и когда его проводить?
10. Как измерять p95 latency в k6?
11. Как проверять SLA (service level agreement) в тесте?
12. Как тестировать кэширование (замерять hit ratio)?
13. Как обнаружить утечку памяти через нагрузочный тест?
14. Как в k6 передавать разные входные данные (CSV)?
15. Какие инструменты позволяют распределённое нагрузочное тестирование?

Лабораторная работа №11.

Профилирование и оптимизация (async profiler)

1. Этап: Поиск узких мест производительности кода с помощью профилирования.

2. Цель: Освоить асинхронный профилировщик (async-profiler / JProfiler / py-spy) для выявления горячих точек (CPU-intensive, I/O-bound) и оптимизации.

3. Задачи:

1. Написать CPU-интенсивную задачу (например, сортировка большого массива, шифрование) и выполнить профилирование.
2. Написать I/O-bound задачу (много вызовов БД или HTTP) и профилировать.
3. С помощью флейм-графа (flame graph) найти функцию, потребляющую больше всего CPU.
4. Оптимизировать код (улучшить алгоритм, добавить кэш, асинхронность) и повторить профилирование.
5. Сравнить результаты.

4. Краткое теоретическое содержание

Async profiler (Java): собирает стек-трейсы семплированием, малая нагрузка, можно собирать данные по CPU, аллокациям, локу.

JProfiler: коммерческий профилировщик (30 дней триал), позволяет детально анализировать потоки, мониторы, JDBC.

py-spy: профилировщик для Python (семплирование, работает без изменения кода).

Flame graph (флейм-граф): визуализация стека вызовов: ширина = времени, вертикаль = глубина стека.

5. Индивидуальные задания (15 вариантов)

№ вар	**Язык**	**Тип нагрузки**	**Профилировщик**	**Ожидаемая оптимизация**
1	Java	CPU (сортировка 10 млн	async-profiler	улучшить алгоритм

№ вар	Язык	Тип нагрузки	Профилировщик	Ожидаемая оптимизация
		элементов)		(QuickSort vs Bubble)
2	Java	I/O (1000 SELECT)	async-profiler	добавить кэш, batch
3	Python	CPU (обработка изображения)	py-spy	оптимизировать циклы (NumPy)
4	Python	I/O (1000 HTTP запросов)	py-spy	asyncio, connection pooling
5	Java	CPU + I/O (шифрование + БД)	async-profiler	профилировать отдельно
6	Java	Локи (deadlock профилирование)	JProfiler	исправить блокировки
7	Go	CPU (JSON сериализация)	pprof	заменить на быструю библиотеку
8	Go	I/O (много gRPC вызовов)	pprof	добавить pool соединений
9	Java	Аллокации	async-profiler	уменьшить

№ вар	Язык	Тип нагрузки	Профилировщик	Ожидаемая оптимизация
		памяти	(alloc)	создание объектов
10	Python	CPU (regex на больших текстах)	py-spy	оптимизировать регулярное выражение
11	Java	Блокировки (synchronized)	JProfiler	заменить на ReentrantLock
12	Java	CPU (парсинг XML)	async-profiler	заменить на StAX
13	Python	I/O (много обращений к файлам)	py-spy	кэш, asyncio
14	Go	CPU + GC	pprof	уменьшить аллокации
15	Java	Смешанная нагрузка (сервер)	async-profiler + JMC	оптимизировать hot method

6. Контрольные вопросы

1. Чем отличается профилирование на основе семплов от инструментирования?
2. Как асинхронный профилировщик снимает показания без существенных накладных расходов?
3. Как читать флейм-граф? Что означает ширина и высота?

4. Как в async-profiler отдельно профилировать CPU, аллокации, локи?
5. Как профилировать Java приложение без изменения кода (-agentpath)?
6. Как найти метод, который вызывает много аллокаций и GC?
7. Как в py-spy профилировать уже работающий процесс (attach)?
8. В чём преимущество JProfiler перед async-profiler?
9. Как профилировать I/O-bound приложение? Какие метрики смотреть?
10. Как измерять время блокировки (lock contention)?
11. Как оптимизировать горячую функцию: что делать в первую очередь?
12. Как профилирование помогает отыскать неэффективный алгоритм ($O(N^2)$ против $O(N \log N)$)?
13. Какой инструмент профилирования можно использовать в CI (без GUI)?
14. Как профилировать код в Kubernetes (sidecar, shared volume)?
15. Как сравнить два профиля (до и после оптимизации)?

Лабораторная работа №12.

Мониторинг (Prometheus + Grafana)

1. Этап: Сбор метрик приложения (RPS, ошибки, latency) с экспортом в Prometheus и визуализацией в Grafana.

2. Цель: Настроить экспорт метрик из приложения (Micrometer / Prometheus client), собрать их в Prometheus и построить дашборд в Grafana.

3. Задачи:

1. Добавить в приложение метрики: количество запросов (RPS), количество ошибок (HTTP 5xx), гистограмму времени ответа.
2. Экспортировать метрики в формате Prometheus (эндпоинт /actuator/prometheus или /metrics).
3. Настроить Prometheus (scrape config) для сбора метрик.
4. Построить дашборд в Grafana: панель с RPS, панель с p95 latency, панель с количеством ошибок.
5. Настроить алерт (например, при ошибках >1% за минуту).
6. Провести нагрузочный тест и наблюдать за изменением метрик.
7. 4. Краткое теоретическое содержание

8. Метрики (RED – Rate, Errors, Duration):
9. Rate – количество запросов в секунду.
10. Errors – количество или процент ошибочных ответов.
11. Duration – распределение времени ответа (p50, p95, p99).
12. Prometheus: pull-модель (HTTP scrape), временная БД, язык запросов PromQL.
13. Grafana: визуализация, дашборды, алерты.

5. Индивидуальные задания (15 вариантов)

№ вар	**Метрики**	**Алерт**	**Дополнительная визуализация**
	RPS, p95 latency, ошибки (count)	ошибки > 5% за 1 мин	топ эндпоинтов по latency
	RPS, p99 latency, ошибки (%)	ошибки > 1% за 2 мин	количество активных сессий
	RPS, p90 latency, ошибки (count)	latency > 500ms на 50% запросов	бизнес-метрика (количество заказов)
	RPS, p95 latency, ошибки (%)	отсутствие метрик > 1 мин	размер очереди Kafka
	RPS, p99 latency, ошибки	высокое использование CPU (>80%)	количество потоков

№ вар	Метрики	Алерт	Дополнительная визуализация
	(count)		
	RPS, p95 latency, ошибки (%)	ошибки БД (таймауты) > 1%	количество запросов к БД
	RPS, p99 latency, ошибки (count)	Redis ошибки > 0	hit ratio кэша
	RPS, p95 latency, ошибки (%)	ошибки платежного шлюза > 0.5%	latency внешнего сервиса
	RPS, p90 latency, ошибки (count)	ошибки авторизации > 10 в минуту	количество заблокированных пользователей
0	RPS, p99 latency, ошибки (%)	долгое время ответа для конкретного пользователя	топ медленных запросов
1	RPS, p95 latency, ошибки	превышение лимита подключений к БД	количество открытых соединений

№ вар	Метрики	Алерт	Дополнительная визуализация
	(count)		
2	RPS, p99 latency, ошибки (%)	отказ Circuit Breaker	состояние Circuit Breaker (closed/open)
3	RPS, p95 latency, ошибки (count)	ошибки валидации входных данных	количество валидационных ошибок
4	RPS, p90 latency, ошибки (%)	низкий hit ratio кэша (<50%)	размер кэша (количество ключей)
5	RPS, p99 latency, ошибки (count)	задержка в обработке сообщения Kafka	lag consumer’a

6. Контрольные вопросы

1. Какие типы метрик бывают (Counters, Gauges, Histograms, Summaries)?
2. Как в Prometheus моделируется RPS (rate() или irate())?
3. Для чего нужны гистограммы? Как вычислить p95 через histogram_quantile?
4. Как отличить клиентские ошибки (4xx) от серверных (5xx) в метриках?

5. Как настроить алерт на отсутствие метрик (аларм сработает, если scrape не происходит)?
6. Как в Grafana построить дашборд с переменными (namespace, pod, endpoint)?
7. Как экспортировать метрики из приложения на Python? (prometheus_client)
8. Как экспортировать метрики из приложения на Java (Micrometer + Actuator)?
9. Как настроить scrape config для динамических целей (kubernetes_sd_config)?
10. Как бороться с высоким cardinality метрик (слишком много уникальных лейблов)?
11. Что такое rate() vs increase() в PromQL?
12. Как в Grafana создать дашборд с оповещениями (alerting)?
13. Как в Prometheus хранить метрики долгосрочно (retention, remote storage)?
14. Как добавить бизнес-метрики (количество заказов, выручка) в мониторинг?
15. Как агрегировать метрики по разным экземплярам приложения (sum by)?

Лабораторная работа 13

Централизованное логирование (Loki + Promtail)

1. Этап: Сбор, централизация и анализ логов микросервисов с помощью Loki и Promtail.

2. Цель: Настроить сбор логов (Promtail) со всех экземпляров приложения, централизованное хранение в Loki и поиск логов в Grafana.

3. Задачи:

1. Поднять Loki и Promtail (в Docker или в K8s).
2. Настроить Promtail для сбора логов из stdout и из файлов.
3. Добавить структурирование логов (JSON) с полями: timestamp, level, service, message, correlation_id.
4. Выполнить несколько запросов к API, чтобы сгенерировать логи.
5. В Grafana (через Data Source Loki) выполнить поиск логов по correlation ID.
6. Настроить алерт (например, при появлении ERROR в логах).

4. Краткое теоретическое содержание

Loki: горизонтально масштабируемая сисЭтап агрегации логов (как Prometheus для логов). Не индексирует содержимое, только лейблы (метки).

Promtail: агент, который собирает логи, добавляет лейблы (например, job, instance, pod) и отправляет в Loki.

Структурированные логи (JSON):

json

{"timestamp":"2025-01-01T12:00:00Z","level":"INFO","service":"payment","message":"Payment approved","correlation_id":"abc-123"}

LogQL: язык запросов к Loki, похож на PromQL (например, {service="payment"} |= "ERROR").

5. Индивидуальные задания (15 вариантов)

№ вар	Структура лога	Источник логов	Лейблы	Алерт
1	JSON (уровень, сообщение)	stdout сервиса	job, instance	наличие ERROR
2	JSON (уровень, correlation_id)	файл app.log	service, env	наличие WARN
3	logfmt	stdout	job, pod	частота ошибок > 10 в минуту
4	JSON (level, message,	stdout + K8s	namespace, pod	появление panic

№ вар	Структура лога	Источник логов	Лейблы	Алерт
	stacktrace)			
5	JSON (level, service, user_id)	файл access.log	service, endpoint	много 5xx (>5)
6	logfmt	stdout	job, instance	долгое выполнение (>2s)
7	JSON (с корреляцией)	stdout + файл	app, version	любое сообщение с FATAL
8	JSON (без корреляции)	stdout	service, region	ошибки БД
9	logfmt	stdout + K8s	pod, container	rate limit превышен
10	JSON (уровень, сообщение, duration)	файл metrics.log	job	долгий запрос (>3s)
11	JSON (correlation_id, step)	stdout + микросервис	service, userId	ошибка валидации
12	logfmt	stdout	app, pod	login неудача > 5

№ вар	Структура лога	Источник логов	Лейблы	Алерт
				раз
13	JSON (уровень, сообщение)	stdout + системные логи	host, app	out of memory
14	JSON (correlation_id, user)	stdout	service, pod	подозрительные логи (SQL injection)
15	JSON (уровень, сообщение, stack)	stdout + error.log	job, level	любой сработавший assert

6. Контрольные вопросы

1. Чем отличается Loki от ELK (Elasticsearch)?
2. Как Loki обрабатывает логи, не индексируя содержимое?
3. Что такое лейблы в Loki и как их настроить?
4. Как добавить correlation ID в логи и использовать для трассировки?
5. Как настроить Promtail для чтения логов из stdout в K8s?
6. Как в Grafana переключиться с дашборда метрик (Prometheus) на логи (Loki) по correlation ID?
7. Что такое LogQL? Как написать запрос для поиска ошибок?
8. Как агрегировать логи (количество ошибок в минуту) в Loki?
9. Как бороться с высоким объёмом логов (ограничения, сжатие)?
10. Как в Loki хранить логи долгосрочно (retention)?
11. Как настроить алерт на логи (например, ERROR) в Loki через Ruler?
12. Как избежать попадания чувствительных данных в логи?
13. Как структурировать логи в приложении (JSON, logfmt)?
14. Как интегрировать Loki с Grafana Alerting?

15.Как в Loki искать логи между двумя временными метками?

Лабораторная работа №14.

Распределённый трейсинг (Jaeger)

1. Этап: Отслеживание запросов через несколько микросервисов с помощью распределённого трейсинга.

2. Цель: Инструментировать микросервисы с помощью OpenTelemetry, настроить Jaeger для сбора трейсов и визуализации зависимостей.

3. Задачи:

1. Добавить в приложение OpenTelemetry SDK (Java / Python / Go).
2. Настроить экспорт трейсов в Jaeger (через HTTP или gRPC).
3. Сгенерировать несколько запросов, проходящих через 2–3 сервиса.
4. В интерфейсе Jaeger найти выполненную трассировку и проанализировать спаны (задержки, вызовы).
5. Определить узкое место (медленный вызов) в цепочке сервисов.
6. Добавить атрибуты в спаны (например, http.method, http.status, user.id).

4. Краткое теоретическое содержание

Трассировка (trace): цепочка спанов (span), отражающая жизненный цикл запроса.

Span: единичный шаг (операция) с именем, временем начала/конца, атрибутами.

Корреляция через контекст: заголовок traceparent (W3C Trace Context) передаётся между сервисами.

Jaeger: UI для поиска трейсов, анализа лагов, построения диаграммы зависимостей сервисов.

5. Индивидуальные задания (15 вариантов)

№ вар	Язык	Количество сервисов	Дополнительные атрибуты	Особенность
1	Java	2	http.method, http.status	синхронные вызовы
2	Java	3	user.id, payment.amount	асинхронный Kafka
3	Python	2	db.statement	замеры времени БД
4	Python	3	user.role	цепочка с ошибкой
5	Go	2	http.url	асинхронные вызовы (goroutine)
6	Java	3	message.id	запись лога в span (log)
7	Java	2	error	intentional slow query
8	Python	2	cache.hit	Redis + БД
9	Go	3	client.id	gRPC вызовы
10	Java	2	payment.gateway	внешний HTTP вызов
11	Java	3	order.id, product.id	несколько атрибутов
12	Python	2	request.body	запись большого сообщения

№ вар	Язык	Количество сервисов	Дополнительные атрибуты	Особенность
13	Go	3	kafka.topic, consumer.group	контекст через Kafka
14	Java	2	thread.name	параллельные спаны
15	Java	4	микросервисная цепочка	полный набор (gateway, auth, order, payment)

6. Контрольные вопросы

1. Что такое distributed tracing и зачем он нужен?
2. Что такое span, trace, context propagation?
3. Как передаётся контекст между сервисами через HTTP?
4. Как передаётся контекст через Kafka?
5. Как добавить кастомный атрибут в span?
6. Как отметить ошибку в span (error flag)?
7. Как измерить продолжительность вызова внешнего API?
8. Как в Jaeger увидеть длинные операции (critical path)?
9. Как анализировать медленные запросы с помощью Flame Graph в Jaeger?
10. Как связать логи (Loki) с трейсами (Jaeger) через correlation ID?
11. Какие есть альтернативы Jaeger (Zipkin, Tempo)?
12. Как использовать OpenTelemetry вместо платформозависимых библиотек?
13. Как влияет трейсинг на производительность (сэмплирование)?
14. Как в Jaeger отслеживать ошибки по всему распределённому запросу?
15. Как в Java автоматически инструментировать Spring Boot (opentelemetry-javaagent)?

Лабораторная работа №15.

CI/CD пайплайн с SAST/DAST (GitHub Actions / GitLab CI)

1. Этап: Настройка конвейера непрерывной интеграции (CI) с включением статического (SAST) и динамического (DAST) анализа безопасности.

2. Цель: Автоматизировать сборку, тестирование, статический анализ кода и динамическое сканирование безопасности в едином CI/CD пайплайне.

3. Задачи:

1. Создать репозиторий (GitHub/GitLab) с простым веб-приложением (Java/Python/Go).
2. Настроить CI пайплайн, который при каждом push выполняет:
3. Сборку проекта
4. Запуск модульных тестов
5. SAST (SonarQube / Semgrep)
6. DAST (OWASP ZAP) против поднятого тестового стенда
7. Установить пороги качества (например, покрытие кода >80%, отсутствие критических уязвимостей).
8. Настроить автоматический деплой на staging при успешном прохождении пайплайна.
9. Протестировать пайплайн, внеся уязвимость (например, SQL инъекцию) и убедиться, что SAST/DAST ловит её.

4. Краткое теоретическое содержание

SAST (Static Application Security Testing): анализ исходного кода без выполнения программы (SonarQube, Semgrep, Checkmarx).

DAST (Dynamic Application Security Testing): анализ работающего приложения через отправку запросов (OWASP ZAP, Burp Suite).

CI/CD (GitHub Actions): workflow в файле .github/workflows/main.yml:

yaml

```
name: CI
on: [push]
jobs:
```

```
build:
  runs-on: ubuntu latest
  steps:
    - uses: actions/checkout@v3
    - name: Build with Maven
      run: mvn compile
    - name: SonarQube Scan
      run: mvn sonar sonar
    - name: ZAP Scan
      run: docker run --network host owasp/zap2docker stable zap-baseline.py -t http //localhost 8080
```

5. Индивидуальные задания (15 вариантов)

№ вар	Язык	SAST-инструмент	DAST-инструмент	Особенность пайплайна
1	Java (Spring Boot)	SonarQube	OWASP ZAP	деплой в staging (Docker)
2	Java (Spring Boot)	Semgrep	ZAP	conditional step (если SAST пройден)
3	Python (Flask)	SonarQube	ZAP	coverage порог (>70%)

№ вар	Язык	SAST-инструмент	DAST-инструмент	Особенность пайплайна
4	Python (FastAPI)	Semgrep	ZAP	arifacts: логов, отчётов
5	Go (Gin)	SonarQube	ZAP	параллельные джобы
6	Java (Spring Boot)	CodeQL	ZAP	matrix build (JDK 11,17)
7	Python (Django)	SonarQube	ZAP	отдельный job для DAST
8	Node.js (Express)	Semgrep	ZAP	кэширование зависимостей
9	Go (gorilla/mux)	SonarQube	ZAP	сканирование image на уязвимости (Trivy)
10	Java (Spring Boot)	Semgrep	ZAP	генерация артефакта (report.html)
11	Python (FastAPI)	SonarQube	ZAP	интеграция со Slack

№ вар	Язык	SAST-инструмент	DAST-инструмент	Особенность пайплайна
				(уведомления)
12	Java (Spring Boot)	CodeQL	ZAP	проверка лицензий зависимостей
13	Go (Gin)	Semgrep	ZAP	стадия production approve
14	Python (Flask)	SonarQube	ZAP	отправка отчёта в Telegram
15	Java (Spring Boot)	Semgrep	ZAP	запуск тестовой среды через docker-compose

6. Контрольные вопросы

1. В чём разница между SAST и DAST? Когда каждый применять?
2. Какой SAST-инструмент лучше для Java? Почему?
3. Как настроить SonarQube для проверки покрытия тестами?
4. Как выполнить ZAP-сканирование в CI без графического интерфейса?

5. Как в GitHub Actions смоделировать среду (docker-compose) для DAST?
6. Какие пороги качества стоит установить для статического анализа?
7. Как зафиксировать артефакты (отчёты) в GitHub Actions?
8. Как в Semgrep написать правило для поиска SQL-инъекций?
9. Как в CI обрабатывать уязвимости: fail или только предупреждение?
10. Как автоматически проверять зависимости на известные CVE (SCA)?
11. Как отделить stage «сборка» от stage «деплой»?
12. Как в пайплайне поднять тестовое окружение, прогнать DAST и остановить?
13. Как в GitHub Actions кэшировать зависимости для ускорения билда?
14. Как в ZAP отключить определённые сканеры (ложные срабатывания)?
15. Как интегрировать SonarQube с GitHub (Quality Gate, комментарии в PR)?

Лабораторная работа №16.

SAST-анализ кода с SonarQube

1. Этап: Статический анализ качества и безопасности кода с помощью SonarQube.

2. Цель: Настроить SonarQube для проверки кода на наличие «запахов» (code smells), уязвимостей (vulnerabilities) и оценки технического долга.

3. Задачи:

1. Запустить SonarQube (Docker или локально).
2. Подключить проект (Java/Python/Go/JS) к SonarQube.
3. Настроить Quality Gate (пороги: покрытие >80%, дублирование <3%, критических уязвимостей = 0).
4. Проанализировать код, исправить найденные проблемы.
5. Интегрировать SonarQube в CI (GitHub Actions / GitLab CI) для автоматической проверки.

4. Краткое теоретическое содержание

SonarQube Metrics:

Coverage – процент кода, покрытого тестами.

Duplications – процент дублированного кода.

Code Smells – нарушения лучших практик.

Bugs – потенциальные ошибки (логические).

Vulnerabilities – уязвимости безопасности.

Security Hotspots – потенциально опасные места.

Quality Gate: автоматический пропуск сборки, если метрики не соответствуют порогу.

5. Индивидуальные задания (15 вариантов)

№ вар	Язык	Особые правила	Порог
1	Java	только критический уровень	покрытие >80%
2	Python	игнорировать дублирование в тестах	дублирование <5%
3	Go	проверка на SQL-инъекции	уязвимости = 0
4	JavaScript	проверка на XSS	code smells <20
5	TypeScript	проверка на использование уязвимых библиотек	покрытие >70%
6	Java + Spring	проверка на циклические зависимости	tech debt <5%
7	Python	проверка на assert в коде	покрытие >85%
8	Go	проверка на неиспользуемые	дублирование

№ вар	Язык	Особые правила	Порог
		переменные	<3%
9	Java	проверка на System.out.println	code smells <15
10	Python	проверка на broad exception	уязвимости = 0
11	Java	проверка на @Autowired в private	покрытие >90%
12	TypeScript	проверка на any тип	code smells <10
13	Java	проверка на цикломатическую сложность (<10)	tech debt <3%
14	Go	проверка на забытый error handling	покрытие >75%
15	Python	проверка на использование exec	code smells <5

6. Контрольные вопросы

1. Что такое технический долг (Technical Debt) в SonarQube?
2. Как SonarQube вычисляет покрытие тестами?
3. Как настроить Quality Gate для блокировки сборки?
4. Как исключить из анализа тестовые файлы?
5. Как интерпретировать Security Hotspots?
6. Чем отличаются Vulnerabilities от Security Hotspots?
7. Как убрать ложные срабатывания (false positive)?

8. Как интегрировать SonarQube с GitHub (Pull Request decoration)?
9. Как запустить локальный SonarQube через Docker?
10. Как использовать SonarQube для мониторинга технического долга в динамике?
11. Как в Java проекте измерить покрытие тестами с JaCoCo?
12. Как в Python проекте измерить покрытие с coverage.py?
13. Как в Go измерить покрытие (go test -cover)?
14. Как настроить CI на отправку отчёта в SonarQube?
15. Как восстановить пропущенный Quality Gate (manual override)?

Лабораторная работа №17.

DAST-тестирование с OWASP ZAP

1. Этап: Динамический анализ безопасности веб-приложения с помощью OWASP ZAP.

2. Цель: Научиться проводить автоматизированное DAST-сканирование, анализировать отчёт и исправлять типовые уязвимости.

3. Задачи:

1. Развернуть тестовое веб-приложение (с намеренными уязвимостями: SQL injection, XSS, CSRF).
2. Настроить OWASP ZAP (Docker или GUI).
3. Провести базовое сканирование (ZAP Baseline Scan) и анализ отчёта.
4. Провести активное сканирование (Active Scan) с подстановкой параметров.
5. Исправить уязвимости (добавить параметризованные запросы, экранирование, CSRF-токен).
6. Повторить сканирование и убедиться, что уязвимости исчезли.

4. Краткое теоретическое содержание

Уязвимости OWASP Top 10 (основные для тестирования):

SQL Injection – параметризованные запросы (prepared statements).

XSS (Cross-Site Scripting) – экранирование вывода (HTML-entity).

CSRF (Cross-Site Request Forgery) – CSRF-токен, SameSite cookie.

ZAP режимы:

Baseline Scan – быстрое сканирование, минимальное количество запросов.

Full Scan – более глубокое сканирование с перебором параметров.

API Scan – сканирование OpenAPI (Swagger) спецификаций.

5. Индивидуальные задания (15 вариантов)

№ вар	**Приложение (стек)**	**Уязвимости (заложенные)**	**Режим сканирования**
1	Java (Spring Boot)	SQLi, XSS	Baseline
2	Python (Flask)	SQLi, CSRF	Active
3	Node.js (Express)	XSS (Reflected), NoSQLi	Baseline
4	Go (Gin)	SQLi (raw query), XSS (Stored)	Active
5	Java (Spring Boot)	CSRF, Path traversal	Baseline
6	Python (FastAPI)	SQLi, Open Redirect	Active
7	PHP (простое)	SQLi, XSS, File inclusion	Full scan
8	Java (Spring Boot)	SQLi, Command injection	Baseline
9	Node.js	NoSQLi, XSS	Active
10	Go	SQLi, CSRF	Baseline
1	Python (Django)	SQLi, XSS (DOM)	Active

№ вар	Приложение (стек)	Уязвимости (заложенные)	Режим сканирования
1			
12	Java (JSP)	SQLi, Path traversal, XSS	Full scan
13	JavaScript (React + API)	XSS (через API), SQLi (API)	API Scan
14	Java (Spring Boot)	CSRF, SQLi, Insecure headers	Baseline + Active
15	Python (Flask)	SQLi, XSS, CORS misconfiguration	Baseline + API Scan

6. Контрольные вопросы

1. Чем отличается DAST от SAST?
2. Какие уязвимости может найти ZAP в веб-приложении?
3. Как ZAP обнаруживает SQL injection?
4. Как защититься от XSS? Что такое экранирование вывода?
5. Как работают CSRF-токены и как их проверить в ZAP?
6. Что такое Baseline Scan и сколько времени он занимает?
7. Чем опасен Active Scan (может нагружать приложение)?
8. Как в ZAP исключить ложные срабатывания (false positive)?
9. Как протестировать API (OpenAPI) с помощью ZAP?
10. Как автоматизировать ZAP в CI/CD?
11. Как в Java (Spring Boot) параметризовать запросы (PreparedStatement)?
12. Как в Python (Flask) экранировать вывод (Markup)?
13. Как настроить CSRF-защиту в Spring Security?
14. Как в React избежать XSS при использовании dangerouslySetInnerHTML?

15.Как интегрировать ZAP с отчётом в GitLab CI (артефакты)?

Лабораторная работа №18.

Автомасштабирование в Kubernetes (HPA)

1. Этап: Горизонтальное автомасштабирование подов (Horizontal Pod Autoscaler) по CPU и кастомным метрикам.

2. Цель: Настроить автомасштабирование Deployment в K8s на основе использования CPU и кастомной метрики (RPS).

3. Задачи:

1. Развернуть приложение с известной нагрузкой (Web-API) в K8s.
2. Настроить Metrics Server для сбора метрик CPU.
3. Создать HPA с target CPU = 50%.
4. Запустить нагрузочный тест (k6 / JMeter) и наблюдать за увеличением числа подов.
5. (Дополнительно) Настроить HPA по кастомной метрике (RPS через Prometheus adapter).

4. Краткое теоретическое содержание

HPA: контроллер, который периодически (по умолчанию 15 секунд) проверяет метрики и изменяет количество реплик.

Формула желаемых реплик:

desiredReplicas = ceil(currentReplicas * (currentMetricValue / desiredMetricValue))

Metrics Server: сбор метрик CPU и памяти из K8s (требуется для HPA).

Custom Metrics (Prometheus Adapter): позволяет использовать метрики приложения (например, RPS, длина очереди).

5. Индивидуальные задания (15 вариантов)

№ вар	Тип метрики	Таргет	Min / Max реплик	Нагрузка
	CPU	50%	2–10	k6

№ вар	Тип метрики	Таргет	Min / Max реплик	Нагрузка
2	CPU	70%	1–5	JMeter
3	Memory	80%	2–8	своя
4	RPS (Prometheus)	50 RPS	3–12	k6
5	CPU + RPS (комбинация)	50% CPU, 50 RPS	2–10	k6
6	CPU	60%	1–6	JMeter
7	Memory	75%	2–7	своя (утечка)
8	RPS (Prometheus)	100 RPS	2–15	k6
9	CPU	50%	3–10	k6
10	Custom (queue length)	10	1–8	симулятор очереди
11	CPU + Memory	50% CPU, 70% Memory	2–10	смешанная
12	RPS (Prometheus)	30 RPS	2–6	k6
13	CPU	40%	2–20	JMeter
14	CPU	80%	1–5	k6

№ вар	Тип метрики	Таргет	Min / Max реплик	Нагрузка
4				
5	Custom (response time p95)	200ms	2–12	k6

6. Контрольные вопросы

1. Как работает Horizontal Pod Autoscaler (HPA)?
2. Какие метрики могут использоваться для автомасштабирования?
3. Что такое Metrics Server и зачем он нужен?
4. Как рассчитать желаемое количество реплик?
5. Что происходит, если HPA не может получить метрику?
6. Как ограничить максимальное количество реплик (maxReplicas)?
7. Как настроить HPA по кастомной метрике через Prometheus adapter?
8. Как проверить состояние HPA (kubectl describe hpa)?
9. Как влияет стабилизация (stabilization window) на масштабирование?
10. Как HPA ведёт себя при часто меняющейся нагрузке?
11. Как связаны HPA и VPA (Vertical Pod Autoscaler)?
12. Как протестировать HPA с помощью нагрузочного теста?
13. Какие параметры можно настроить для HPA (поведение)?
14. Как отслеживать историю масштабирований?
15. Как HPA взаимодействует с Cluster Autoscaler (масштабирование узлов)?

Лабораторная работа №19.

Event-driven автомасштабирование (KEDA)

1. Этап: Масштабирование на основе событий (длина очереди Kafka / RabbitMQ) с помощью KEDA.

2. Цель: Настроить KEDA (Kubernetes Event-driven Autoscaler) для масштабирования consumer’а по длине очереди Kafka.

3. Задачи:

1. Установить KEDA в кластер (через Helm или оператор).
2. Развернуть Kafka (или RabbitMQ) и consumer.
3. Написать ScaledObject, который:
4. отслеживает отставание (lag) в Kafka (max threshold = 50)
5. масштабирует consumer от 1 до 10 подов.
6. Сгенерировать нагрузку (отправить много сообщений) и наблюдать за масштабированием.
7. Убедиться, что при уменьшении сообщений поды уменьшаются.

4. Краткое теоретическое содержание

KEDA: компонент K8s, масштабирующий реплики по внешним источникам событий (Kafka, RabbitMQ, Prometheus, cron и др.)

ScaledObject: CRD, описывающий:

scaleTargetRef – какой Deployment масштабировать.

triggers – тип триггера (kafka, rabbitmq) и параметры.

minReplicaCount / maxReplicaCount.

Пример ScaledObject (Kafka):

yaml

```
apiVersion: keda.sh/v1alpha1
kind: ScaledObject
metadata:
  name: kafka-consumer-scaler
spec:
  scaleTargetRef:
    name: consumer-deployment
  triggers:
  - type: kafka
    metadata:
      bootstrapServers: my-kafka:9092
```

```
consumerGroup: my-group
topic: orders
lagThreshold: "50"
```

5. Индивидуальные задания (15 вариантов)

№ вар	Триггер	Lag threshold	Min/Max	Дополнительно
1	Kafka	50	1–10	
2	Kafka	100	2–15	
3	RabbitMQ (queue length)	20	1–8	
4	RabbitMQ (message rate)	100 msg/s	2–10	
5	Kafka (lag)	30	1–5	
6	Kafka + CPU (комбинация)	50 lag + 70% CPU	1–10	
7	Azure Queue (симуляция)	10	1–6	
8	AWS SQS (симуляция)	50	1–8	
9	Kafka (lag) с auth	50	1–10	SSL

№ вар	Триггер	Lag threshold	Min/Max	Дополнительно
10	RabbitMQ (queue length)	30	2–12	vhost
11	Kafka (lag)	200	2–20	3 топика
12	Cron (по расписанию)	–	1–1	
13	Prometheus (metric)	>100	1–10	custom
14	Kafka (lag) + external auth	50	1–10	SASL
15	Kafka (lag) + dead letter queue	50	1–15	автоматическое

6. Контрольные вопросы

1. Чем отличается KEDA от HPA?
2. Какие типы триггеров поддерживает KEDA?
3. Как KEDA отслеживает lag в Kafka?
4. Что такое ScaledObject и ScaledJob?
5. Как проверить, что KEDA масштабирует поды?
6. Как задать минимальное и максимальное количество реплик?
7. Как KEDA обрабатывает ошибки подключения к брокеру?
8. Как настроить KEDA для RabbitMQ (queue length)?
9. Как KEDA ведёт себя при неактивности (scale-to-zero)?
10. Как комбинировать несколько триггеров в одном ScaledObject?
11. Как влияет cooldownPeriod на уменьшение реплик?
12. Как мониторить KEDA (метрики, логи)?
13. Как принудительно масштабировать (не через KEDA)?

14. Как защитить KEDA от резких скачков нагрузки (maxReplicaCount)?
15. Как протестировать KEDA локально (без кластера)?

Лабораторная работа №20.

GitOps с ArgoCD

1. Этап: Принципы GitOps и синхронизация кластера с Git-репозиторием с помощью ArgoCD.

2. Цель: Установить ArgoCD, настроить автоматическую синхронизацию приложения из Git-репозитория (манифесты K8s), освоить принцип «pull-модели».

3. Задачи:

1. Установить ArgoCD (в minikube/k3s или через оператор).
2. Создать Git-репозиторий с манифестами (Deployment, Service, ConfigMap).
3. Зарегистрировать приложение (Application) в ArgoCD, указав репозиторий и путь.
4. Настроить автоматическую синхронизацию (sync policy: automated).
5. Изменить манифест (новый тег образа) в Git и убедиться, что ArgoCD применил изменения в кластере.
6. Включить self-heal (при ручном изменении ресурсов вернуть состояние из Git).

4. Краткое теоретическое содержание

GitOps: декларативное описание инфраструктуры и приложений хранится в Git как единый источник истины. Агент в кластере (ArgoCD) синхронизирует состояние.

ArgoCD Application: CRD с указанием:

1. source – репозиторий, путь, Helm values.
2. destination – целевой кластер и namespace.
3. syncPolicy – автоматическая синхронизация, prune, self-heal.

5. Индивидуальные задания (15 вариантов)

№ вар	Тип манифестов	Количество ресурсов	Sync policy	Особенность
1	plain YAML	2 (Deployment, Service)	auto	
2	plain YAML	3 (Deployment, Service, ConfigMap)	auto + prune	
3	Helm chart	1 (чарт)	auto	
4	Kustomize	2 (base + overlay)	auto	
5	Helm + values	1	auto	разные окружения
6	plain YAML	4 (Deployment, Service, Ingress, Secret)	auto + self-heal	
7	Kustomize + Helm	2	auto	
8	Helm + зависимость	2 чарта	auto	
9	plain YAML	2	manual	
10	Helm + values (staging)	1	auto	

№ вар	Тип манифестов	Количество ресурсов	Sync policy	Особенность
11	plain YAML + JSON patches	2	auto + prune	
12	Kustomize (компоненты)	3	auto	
13	Helm + подстановка CI/CD	1	auto	
14	plain YAML + скрытые зависимости	3	auto + self-heal	
15	Helm (multi-service)	3 чарта	aut o	репозиторий Helm

6. Контрольные вопросы

1. Что такое GitOps и чем он отличается от традиционного CI/CD (push-модель)?
2. Как ArgoCD отслеживает изменения в Git?
3. Что такое синхронизация (Sync), prune, self-heal?
4. Как ArgoCD сравнивает состояние в Git и в кластере?
5. Как в ArgoCD настроить несколько окружений (staging, production)?
6. Что такое ApplicationSet и зачем он нужен?
7. Как ArgoCD управляет секретами (Secrets)?
8. Как ArgoCD роллит новую версию (rollout strategies)?
9. Как восстановить предыдущее состояние (rollback) в ArgoCD?
10. Как интегрировать ArgoCD с CI (например, обновить тег образа)?

11. Как в ArgoCD задать порядок синхронизации (sync waves)?
12. Как мониторить состояние приложений в ArgoCD UI?
13. Как настроить нотификации (Telegram, Slack) о синхронизации?
14. Чем ArgoCD отличается от Flux?
15. Как проверить, что self-heal работает (вручную подправить под)?

Лабораторная работа №21.

Резервное копирование и восстановление PostgreSQL

1. Этап: Организация резервного копирования (логического и физического) и восстановления данных в PostgreSQL.

2. Цель: Настроить автоматический бэкап (pg_dump и WAL-архивацию), выполнить восстановление базы данных на определённый момент времени (Point-in-Time Recovery).

3. Задачи:

1. Развернуть PostgreSQL (Docker или K8s).
2. Настроить бэкап через pg_dump и cron (ежедневно).
3. Включить WAL-архивацию (архивировать в S3 или локальную папку).
4. Сымитировать сбой (удаление таблицы) и восстановить базу на момент до сбоя.
5. Настроить автоматическую ротацию бэкапов (удаление старых).

4. Краткое теоретическое содержание

Логический бэкап (pg_dump): дамп схемы и данных, можно восстановить частично, не зависит от архитектуры.

Физический бэкап (WAL-архивация): архив журналов предзаписи (WAL), позволяет восстановиться на любой момент времени (PITR).

Point-in-Time Recovery (PITR): восстановление базы на заданное время (например, до удаления таблицы).

5. Индивидуальные задания (15 вариантов)

№ вар	Тип бэкапа	Хранилище	Ротация	Особенность
1	pg_dump	локальная папка	7 дней	
2	pg_dump	S3 (MinIO)	30 дней	
3	WAL (PITR)	локальная папка	14 дней	восстановление на момент
4	WAL (PITR)	S3	7 дней	
5	pg_dump + WAL	S3	7 дней/30 дней	
6	pg_dump	локальная папка	10 дней	сжатие (gzip)
7	WAL (PITR)	NFS	7 дней	восстановление на конкретную транзакцию
8	pg_dump	Git LFS (имитация)	1 день	для разработки
9	pg_dump + шифрование	локальная папка	7 дней	пароль
10	WAL (PITR) + скрипт	S3	14 дней	автоматическое удаление

№ вар	Тип бэкапа	Хранилище	Ротация	Особенность
11	pg_dump (разделение схем)	S3	30 дней	дамп только схемы
12	WAL (PITR) + pg_basebackup	S3	14 дней	
13	pg_dump (custom)	локальная папка	5 дней	параллельный дамп
14	WAL (PITR) + восстановление в другой кластер	S3	7 дней	
15	pg_dump + Telegram уведомление	S3	7 дней	бот при успехе/ошибке

6. Контрольные вопросы

1. Чем отличается логический бэкап от физического?
2. В каких случаях нужен PITR?
3. Как настроить WAL-архивацию в PostgreSQL?
4. Какие параметры PostgreSQL важны для PITR?
5. Как восстановить базу на момент времени (recovery_target_time)?
6. Как проверить целостность бэкапа?
7. Как часто нужно делать бэкапы?
8. Какие риски при длительном хранении бэкапов?
9. Как автоматизировать бэкап в Kubernetes?
10. Как восстановить один сбойный объект (например, таблицу) без полной остановки БД?
11. Как сжать дамп (pg_dump + gzip)?

12. Как зашифровать дамп перед отправкой в S3?
13. Как бэкапить БД без простоя (pg_dump не блокирует чтение)?
14. Как восстанавливать в другой K8s-кластер из S3?
15. Как настроить ротацию бэкапов (удаление старых)?

Лабораторная работа №22.

Полный Highload-стек в Docker Compose

1. Этап: Локальный запуск многоконтейнерного высоконагруженного приложения с использованием Docker Compose.

2. Цель: Научиться описывать инфраструктуру (бэкенд, БД, кэш, очередь, веб-сервер) в docker-compose.yml, управлять сетями и volumes.

3. Задачи:

Создать docker-compose.yml для:

1. Веб-приложения (бэкенд) на выбранном языке (Java/Python/Go).
2. PostgreSQL / ClickHouse.
3. Redis (кэш).
4. Kafka + Zookeeper (или RabbitMQ).
5. Nginx (reverse proxy).
6. Настроить сети (bridge) и volumes для персистенции данных.
7. Запустить стек и проверить, что все сервисы работают и видят друг друга (по именам контейнеров).
8. Добавить healthcheck для каждого сервиса.
9. Выполнить нагрузочный тест через k6 (дополнительный контейнер).

4. Краткое теоретическое содержание

Docker Compose: инструмент для описания и запуска многоконтейнерных приложений.

Пример docker-compose.yml:

yaml

```yaml
version: '3.8'
services:
  app:
```

```
    build: ./backend
    ports:
      - "8080:8080"
    depends_on:
      - postgres
      - redis
  postgres:
    image: postgres:15
    environment:
      POSTGRES_DB: app
      POSTGRES_USER: user
      POSTGRES_PASSWORD: pass
    volumes:
      - pgdata:/var/lib/postgresql/data
  redis:
    image: redis:7-alpine
  nginx:
    image: nginx:alpine
    ports:
      - "80:80"
    volumes:
      - ./nginx.conf:/etc/nginx/nginx.conf
volumes:
  pgdata:
```

5. Индивидуальные задания (15 вариантов)

№ вар	Бэкенд	БД	Кэш	Очередь	Reverse proxy

№ вар	Бэкенд	БД	Кэш	Очередь	Reverse proxy
1	Spring Boot	PostgreSQL	Redis	–	Nginx
2	FastAPI	PostgreSQL	–	RabbitMQ	Nginx
3	Gin	PostgreSQL	Redis	Kafka	–
4	Django	PostgreSQL	Redis	–	Nginx
5	Spring Boot	PostgreSQL	Redis	Kafka	Nginx
6	Flask	PostgreSQL	Redis	–	Nginx
7	Express (Node)	PostgreSQL	Redis	RabbitMQ	Nginx
8	Go (net/http)	ClickHouse	Redis	Kafka	Caddy
9	Spring Boot	PostgreSQL	Redis	–	Traefik
10	FastAPI	ClickHouse	Redis	Kafka	Nginx
11	Spring Boot	PostgreSQL + ClickHouse (dual)	Redis	RabbitMQ	Nginx
12	Gin	PostgreSQL	–	Kafka	Nginx

№ вар	Бэкенд	БД	Кэш	Очередь	Reverse proxy
13	FastAPI	PostgreSQL	Redis	–	Traefik
14	Spring Boot	PostgreSQL	Redis	Kafka	Apache
15	Django	PostgreSQL	Redis	RabbitMQ	Nginx

6. Контрольные вопросы

1. Как в Docker Compose сервисы находят друг друга по имени?
2. Для чего нужны volumes в Docker Compose?
3. Как настроить зависимость между сервисами (depends_on)?
4. Что делает restart: unless-stopped?
5. Как задать переменные окружения для контейнера?
6. Как пробросить порты (ports vs expose)?
7. Как настроить сеть (bridge) и в чем отличие от network_mode: host?
8. Как добавить healthcheck для сервиса?
9. Как поднять несколько экземпляров одного сервиса (scale)?
10. Как логировать stdout контейнера в файл?
11. Как пересобрать образ без кэша (docker-compose build --no-cache)?
12. Как выполнить команду в запущенном контейнере?
13. Как проверить, что все сервисы готовы (wait-for-it)?
14. Как обновить конфигурацию без пересоздания контейнера?
15. Как ограничить ресурсы (CPU, память) в docker-compose.yml?

Лабораторная работа №23.

Развёртывание в облаке (Yandex Cloud / VK Cloud)

1. Этап: Деплой высоконагруженного приложения в облачном managed-кластере Kubernetes с использованием Yandex Cloud / VK Cloud.

2. Цель: Научиться создавать управляемый кластер K8s, подключаться к нему, разворачивать приложение, настраивать Ingress с SSL (Let's Encrypt) и внешний LoadBalancer.

3. Задачи:

1. Зарегистрироваться в Yandex Cloud / VK Cloud (использовать грант или пробный период).
2. Создать managed-кластер K8s (один node group, 2–3 узла).
3. Настроить kubectl для доступа к кластеру.
4. Развернуть приложение (из ЛР 22) с помощью манифестов или Helm.
5. Настроить Ingress-контроллер (Nginx Ingress) и получить SSL-сертификат (Let's Encrypt) через cert-manager.
6. Создать Service типа LoadBalancer для публичного доступа.
7. Проверить, что приложение доступно по HTTPS.

4. Краткое теоретическое содержание

Managed Kubernetes (Yandex Cloud, VK Cloud, AWS EKS, GCP GKE): предоставляется control plane (master) как услуга, управление worker-узлами.

Ingress Controller (Nginx Ingress): реализует Ingress API, маршрутизирует трафик по правилам, может завершать TLS.

Cert-Manager: автоматически получает и обновляет SSL-сертификаты от Let's Encrypt.

Service LoadBalancer: создаёт облачный балансировщик для доступа к приложению из интернета.

5. Индивидуальные задания (15 вариантов)

№ вар	**Облачный провайдер**	**Тип приложения**	**Особенность**

№ вар	Облачный провайдер	Тип приложения	Особенность
1	Yandex Cloud	Spring Boot	Ingress + cert-manager
2	Yandex Cloud	FastAPI	LoadBalancer (без Ingress)
3	VK Cloud	Gin	Ingress + SSL
4	Yandex Cloud	Django + PostgreSQL (Managed)	отдельная managed БД
5	Yandex Cloud	Spring Boot + Redis (Managed)	отдельный managed Redis
6	VK Cloud	FastAPI	NodePort + cloud firewall
7	Yandex Cloud	Spring Boot	Ingress + cert-manager + HPA
8	Yandex Cloud	Go	LoadBalancer + Cloud DNS
9	VK Cloud	Django + PostgreSQL (Managed)	отказоустойчивый кластер
10	Yandex Cloud	Spring Boot + Kafka (Managed)	отдельный managed Kafka
11	Yandex Cloud	FastAPI + ClickHouse	отдельный Managed ClickHouse

№ вар	Облачный провайдер	Тип приложения	Особенность
12	VK Cloud	Gin + Redis	отдельный Redis cluster
13	Yandex Cloud	Spring Boot	Ingress + HTTP -> HTTPS redirect
14	Yandex Cloud	Django	Object Storage для статики
15	VK Cloud	Spring Boot	автопривязка домена (Cloud DNS)

6. Контрольные вопросы

1. Чем отличается managed-кластер от собственного (on-prem) кластера K8s?
2. Как получить доступ к кластеру через kubectl?
3. Как создать ингресс-контроллер в облачном кластере?
4. Как работает cert-manager с Let's Encrypt в облаке?
5. Как создать LoadBalancer в Yandex Cloud (аннотация service.beta.kubernetes.io/yandex-cloud/...)?
6. Как подключить внешнюю managed БД к приложению внутри кластера?
7. Как настроить security groups (firewall) для доступа к кластеру?
8. Как собирать логи из облачного кластера (Cloud Logging)?
9. Как настроить мониторинг (managed Prometheus) в облаке?
10. Как деплоить приложения через Helm в облаке?
11. Как установить cert-manager в кластер (Helm)?
12. Как включить HTTP-to-HTTPS редирект на Ingress?
13. Как привязать свой домен к облачному кластеру (DNS)?
14. Как настроить HPA в облачном кластере?
15. Как управлять расходами в облаке (бюджеты, оповещения)?

Лабораторная работа №24.

Комплексный проект: Highload-маркетплейс (часть 1 – архитектура и инфраструктура)

1. Этап: Проектирование микросервисной системы, создание репозиториев, CI/CD, развёртывание инфраструктуры.

2. Цель: Создать основу для сквозного проекта: выбрать технологический стек, спроектировать архитектуру, настроить CI/CD и облачную инфраструктуру.

3. Задачи:

1. Спроектировать систему (микросервисы: каталог, корзина, заказы, оплата, нотификации).
2. Выбрать технологический стек (Java/Spring Boot / Python / Go).
3. Создать Git-репозитории (монорепозиторий или несколько).
4. Настроить CI/CD пайплайн (GitHub Actions / GitLab CI) для сборки и публикации Docker-образов.
5. Развернуть инфраструктуру (K8s, БД, Redis, Kafka) в облаке или локально.
6. Написать базовые манифесты K8s для каждого микросервиса.

4. Краткое теоретическое содержание

Микросервисная архитектура: набор слабосвязанных сервисов, каждый отвечает за свой домен.

Рекомендуемый стек: Spring Boot (каталог, заказы), Python (FastAPI) для оплаты, Go для нотификаций. Kafka для событий, Redis для сессий/кэша, PostgreSQL для хранения, ClickHouse для аналитики.

5. Индивидуальные задания (15 вариантов)

№ вар	**Стек (бэкенд)**	**Количество микросервисов**	**БД**	**Сообщения**	**Кэш**
	Java	3	Postg	Kafka	R

№ вар	Стек (бэкенд)	Количество микросервисов	БД	Сообщения	Кэш
	(Spring Boot)		reSQL		edis
	Python (FastAPI)	3	PostgreSQL	RabbitMQ	Redis
	Go (Gin)	3	PostgreSQL	Kafka	Redis
	Java + Python	4	PostgreSQL	Kafka	Redis
	Java + Go	4	PostgreSQL	RabbitMQ	Redis
	Python + Go	4	PostgreSQL	Kafka	Redis
	Java (Spring Boot)	3	PostgreSQL + ClickHouse	Kafka	Redis
	Python (FastAPI)	3	PostgreSQL + ClickHouse	RabbitMQ	Redis
	Go (Gin)	3	PostgreSQL + ClickHouse	Kafka	Redis

№ вар	Стек (бэкенд)	Количество микросервисов	БД	Сообщения	Кэш
10	Java (Spring Boot)	5	PostgreSQL	Kafka	Redis
11	Python (FastAPI)	5	PostgreSQL	RabbitMQ	Redis
12	Go (Gin)	5	PostgreSQL	Kafka	Redis
13	Kotlin (Ktor)	3	PostgreSQL	Kafka	Redis
14	.NET (C#)	3	PostgreSQL	RabbitMQ	Redis
15	Node.js (Express)	3	PostgreSQL	Kafka	Redis

6. Контрольные вопросы (по проекту)

1. Почему выбран именно такой стек?
2. Как коммуницируют микросервисы (синхронно – REST/gRPC, асинхронно – Kafka)?
3. Как организовано обнаружение сервисов (Service Discovery)?
4. Как управляются транзакции между сервисами (Saga)?
5. Как хранятся данные разных сервисов (Database per Service)?
6. Какие паттерны отказоустойчивости используются?
7. Как мониторить и логировать микросервисы?
8. Как обеспечивается безопасность (JWT, OAuth2, mTLS)?
9. Как организовано CI/CD?

10. Как тестировать интеграцию микросервисов (контрактное тестирование)?

Лабораторная работа №25.

Комплексный проект: Highload-маркетплейс (часть 2 – бизнес-логика)

1. Этап: Реализация основных микросервисов и их интеграция через REST/gRPC и Kafka.

2. Цель: Написать код микросервисов, реализовать API (каталог, корзина, заказы), настроить взаимодействие через Kafka и синхронные вызовы.

3. Задачи:

1. Реализовать сервис «Каталог» (GET /products, GET /products/{id}).
2. Реализовать сервис «Корзина» (POST /cart, GET /cart, DELETE /cart).
3. Реализовать сервис «Заказы» (POST /orders, GET /orders).
4. Интегрировать через Kafka: событие «заказ создан» отправляется в сервис оплаты.
5. Добавить JWT-аутентификацию (для доступа к корзине и заказам).
6. Написать docker-compose для локального запуска всех сервисов.

4. Краткое теоретическое содержание

Сценарий:

Пользователь добавляет товар в корзину (REST).

Оформляет заказ → сервис заказов отправляет событие в Kafka order-created.

Сервис оплаты читает событие, обрабатывает оплату (имитация), отправляет событие payment-completed.

Сервис заказов обновляет статус заказа.

5. Индивидуальные задания (по выбору из части 1)

Варианты стека и количество сервисов – из ЛР 24. Задания не имеют дополнительных вариантов, так как они привязаны к проекту, выбранному в части 1.

6. Контрольные вопросы (по проекту)

1. Как сервис заказов узнаёт о завершении оплаты? (Kafka consumer)
2. Как обеспечить идемпотентность обработки событий?
3. Как восстанавливаться после падения consumer'а?
4. Как гарантировать доставку сообщения exactly-once?
5. Как синхронизировать состояние корзины и сессии?
6. Как обновлять каталог товаров (CRUD для администратора)?
7. Как обрабатывать одновременное оформление одного заказа (race condition)?
8. Как реализовать отмену заказа (Saga)?
9. Как тестировать API микросервисов (postman, contract tests)?
10. Как обновлять схему API без остановки?

Лабораторная работа №26.

Комплексный проект: Highload-маркетплейс (часть 3 – нагрузочное тестирование и наблюдаемость)

1. Этап: Нагрузочное тестирование, мониторинг, трейсинг и алерты для высоконагруженного маркетплейса.

2. Цель: Проверить систему на прочность (RPS, latency, ошибки), настроить наблюдаемость (Prometheus, Grafana, Loki, Jaeger) и алерты.

3. Задачи:

1. Написать сценарий нагрузочного тестирования (k6) для основных эндпоинтов (каталог, корзина, заказы).
2. Провести нагрузочный тест (ramp-up до 200 VU) и замерить p95/p99 latency.

3. Собрать метрики приложений (Micrometer / prometheus_client) и построить дашборды в Grafana.
4. Настроить централизованный сбор логов (Loki) и трейсинг (Jaeger) для микросервисов.
5. Создать алерты (например, p99 latency > 500ms, ошибки > 1%).
6. Провести повторный тест после оптимизации и сравнить результаты.

4. Краткое теоретическое содержание

Нагрузочный сценарий (k6):

javascript

```
export default function () {
  // этапы:
  // 1 – просмотр каталога
  // 2 – добавление в корзину
  // 3 – оформление заказа
}
```

Метрики: RPS (rate), ошибки (errors), latency (histogram).

Алерты: на основе пороговых значений (threshold) или аномалий.

5. Индивидуальные задания (вариации по тем же стекам)

Сценарий нагрузки должен включать все ключевые эндпоинты (каталог, корзина, заказы), а также одновременное выполнение (mix).

6. Контрольные вопросы (по проекту)

1. Какие эндпоинты самые нагруженные? Почему?
2. Какая p95 latency у каталога и у оформления заказа?
3. Как влияет кэширование на результаты теста?
4. Какие ошибки возникают при высокой нагрузке?
5. Как проверить, что Circuit Breaker сработал?
6. Как latency в Kafka влияет на общее время?
7. Как профилирование (async-profiler) помогает найти узкое место?
8. Какой процент ошибок считается допустимым (SLO)?
9. Как после оптимизации изменились метрики?
10. Какие выводы сделаны и какие улучшения внесены?

Лабораторная работа №27.

Комплексный проект: защита и документация

1. Этап: Подготовка итоговой документации и презентация комплексного проекта перед комиссией.

2. Цель: Оформить проект в виде полноценного отчёта (Architecture Decision Records, OpenAPI, руководство по развёртыванию) и защитить его.

3. Задачи

1. Оформить Архитектурные решения (ADR) – минимум 3:
2. Выбор микросервисной архитектуры
3. Выбор технологий (брокер сообщений, БД)
4. Паттерны отказоустойчивости
5. Спецификация API (OpenAPI / Swagger) для каждого микросервиса.
6. Документация по развёртыванию (локальный docker-compose и облачный K8s).
7. Подготовить презентацию (7–10 слайдов): архитектура, технологии, нагрузочное тестирование, результаты оптимизации.
8. Защитить проект перед комиссией (демонстрация работающего приложения и дашбордов мониторинга).

4. Структура отчёта

Введение – цели и задачи проекта, предметная область.

Архитектура – диаграмма C4 (Context, Container), список микросервисов.

Технологический стек – выбор технологий и обоснование.

ADR – ключевые архитектурные решения.

API – спецификация (OpenAPI).

Инфраструктура и CI/CD – описание деплоя, пайплайнов.

Нагрузочное тестирование – сценарии, метрики, сравнение.

Наблюдаемость – метрики, логи, трейсинг.

Заключение – достигнутые результаты, выводы, перспективы.

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 «Междисициплинарный проектный практикум-4» для студентов направления подготовки
09.03.04 Программная инженерия Направленность (профиль)
«Инженерия сложных программных систем»

Ставрополь 2026

ВВЕДЕНИЕ

Актуальность дисциплины

Дисциплина **«Междисциплинарный проектный практикум-4»** является завершающим этапом проектной подготовки бакалавров по профилю «Инженерия сложных программных систем». Она направлена на интеграцию знаний, полученных при изучении профессиональных дисциплин, и их применение при реализации комплексного программного проекта.

В соответствии с учебным планом направления 09.03.04, дисциплина реализуется в **7 семестре**, объём составляет **3 зачётные единицы (108 академических часов)**, из которых значительная часть отведена на самостоятельную работу. Практикум завершается **курсовым проектом (КП)**.

Цель методических указаний

Систематизировать самостоятельную работу студента, обеспечить методическую поддержку при выполнении индивидуального проекта, подготовке к защите и формировании портфолио.

Задачи самостоятельной работы студента

1. Закрепить и углубить теоретические знания в области проектирования высоконагруженных и распределённых систем.
2. Сформировать практические навыки разработки, тестирования и развёртывания программного обеспечения.
3. Развить способность управлять проектом: планировать сроки, распределять задачи, документировать решения.
4. Освоить инструменты командной разработки (Git, CI/CD, система управления проектами).
5. Подготовить и защитить курсовой проект в соответствии с требованиями.

Связь с предшествующими и последующими дисциплинами

Предшествующие дисциплины	Последующие дисциплины
Междисциплинарный проектный практикум-2, -3	Выполнение выпускной квалификационной работы (ВКР)
Высоконагруженные и распределённые системы	Защита ВКР
Контейнеризация и DevOps / Микросервисные технологии	Государственный экзамен
Тестирование и отладка ПО	

Формируемые компетенции (ПК)

Код	Содержание компетенции
ПК-1	Управление программными проектами (планирование, контроль сроков и ресурсов, системы контроля версий, инструменты управления задачами)
ПК-2	Проектирование архитектуры программных систем среднего и крупного масштаба с применением современных паттернов
ПК-3	Разработка ПО (алгоритмы, современные языки программирования, веб- и desktop-приложения, работа с БД)
ПК-4	Тестирование, отладка и оценка качества ПО (автоматизированное тестирование, надежность, безопасность)
ПК-5	Сопровождение, настройка, развёртывание и эксплуатация ПО (включая контейнеризацию и облачные платформы)

Код	Содержание компетенции
ПК-6	Работа с данными (проектирование БД, анализ и обработка данных, интеграция с внешними системами)
ПК-7	Применение DevOps-практик, облачных сервисов и средств автоматизации
ПК-8	Адаптация и применение методов ИИ и машинного обучения
ПК-9	Оформление технической документации, презентаций, отчётов, деловая коммуникация
ПК-10	Соблюдение правовых норм, стандартов и требований информационной безопасности

1. ОБЩАЯ ХАРАКТЕРИСТИКА САМОСТОЯТЕЛЬНОЙ РАБОТЫ

1.1. Объём самостоятельной работы

Вид самостоятельной работы	Трудоёмкость (акад. часы)	% от СРС
Изучение теоретического материала	18	30%
Выполнение практической части проекта	24	40%
Подготовка документации и отчёта	12	20%
Подготовка к защите	6	10%
Итого	**60**	**100%**

Примечание: общая трудоёмкость дисциплины – 108 часов, из них 48 часов – контактная работа (включая лекции, лабораторные, консультации), 60 часов – самостоятельная работа.

1.2. Структура самостоятельной работы по темам

	Тема	Часы	Форма отчётности
	Анализ требований и проектирование архитектуры	8	Техническое задание, ADR
	Разработка компонентов проекта (бэкенд, БД, API)	16	Исходный код в репозитории
	Контейнеризация и CI/CD пайплайн	10	Dockerfile, docker-compose, workflow CI
	Тестирование (модульное, интеграционное, нагрузочное)	8	Отчёт о тестировании
	Документирование (OpenAPI, Readme, пользовательская документация)	10	Документация в репозитории
	Подготовка презентации и защита проекта	8	Презентация, доклад

1.3. Рекомендуемый график выполнения (по неделям семестра)

Неделя	Задание	Форма контроля
1-2	Выбор темы, утверждение ТЗ, выбор технологического стека	Собеседование с руководителем

Неделя	Задание	Форма контроля
3-4	Проектирование архитектуры (диаграммы, ADR)	Проверка документации
5-8	Разработка основных модулей, написание кода	Промежуточный просмотр кода
9-10	Контейнеризация, настройка CI/CD	Демонстрация пайплайна
11-12	Тестирование (unit, integration, e2e)	Отчёт по тестированию
13-14	Документирование (OpenAPI, README, руководство пользователя)	Проверка документации
15	Подготовка презентации, доклада, видео-демонстрации	Предзащита
16	Защита курсового проекта	Дифференцированный зачёт

2. МЕТОДИЧЕСКИЕ РЕКОМЕНДАЦИИ ПО ИЗУЧЕНИЮ ТЕОРЕТИЧЕСКОГО МАТЕРИАЛА

Для успешного выполнения проектного практикума студенту необходимо самостоятельно изучить следующие темы:

2.1. Перечень тем для самостоятельного изучения

	Тема	Источники	Часы
	Паттерны проектирования микросервисной архитектуры (SAGA, CQRS, API Gateway, Circuit Breaker)	[1,2, 3]	4
	Контейнеризация: Docker, Docker Compose, оркестрация (minikube/k3s)	[4,5]	4
	CI/CD: GitHub Actions / GitLab CI, артефакты, переменные окружения	[6]	2
	Базы данных: проектирование схем, индексы, транзакции, ORM	[7]	2
	Безопасность: JWT, OAuth2, HTTPS, защита от OWASP Top 10	[8]	2
	Документирование API: OpenAPI (Swagger), Markdown	[9]	2
	Методологии управления проектами (Scrum, Kanban), инструменты (Trello, YouTrack)	[10]	2

2.2. Рекомендуемая литература

Основная:

1. Ньюмен С. «Проектирование микросервисной архитектуры». – СПб.: Питер, 2022.
2. Ричардсон К. «Микросервисы. Паттерны разработки и рефакторинга». – М.: ДМК Пресс, 2021.
3. Клеппманн М. «Высоконагруженные приложения». – СПб.: Питер, 2022.

4. Кьюб Д. «Kubernetes в действии». – М.: ДМК Пресс, 2020.
5. Кохлс К. «Docker на практике». – СПб.: Питер, 2021.
6. Доугал А. «GitHub Actions». – O'Reilly, 2022 (на англ.).
7. Грубер М. «SQL. Полное руководство». – М.: Вильямс, 2021.
8. Штайнер Й. «Безопасность веб-приложений». – СПб.: Питер, 2020.
9. OpenAPI Initiative. «OpenAPI Specification» (документация онлайн).
10. Кноберг А. «SCRUM. Революционный метод управления проектами». – М.: Манн, Иванов и Фербер, 2020.

2.3. Рекомендации по работе с литературой

Для каждой темы составьте **конспект-схему** (майнд-карту) ключевых понятий.

Отмечайте практические примеры из литературы, которые можно применить в вашем проекте.

Используйте официальную документацию инструментов (Docker, Kubernetes, FastAPI, Spring Boot) как основной источник.

Создайте личный репозиторий для **шаблонов кода** (docker-compose для базы данных, пример CI/CD, пример тестов).

3. МЕТОДИЧЕСКИЕ УКАЗАНИЯ ПО ВЫПОЛНЕНИЮ ПРАКТИЧЕСКИХ РАБОТ

Практическая часть самостоятельной работы представляет собой выполнение **курсового проекта** по индивидуальной теме.

3.1. Примерные темы проектов (соответствуют лабораторным работам курса)

Студент выбирает одну из тем на основе стека, изученного в предыдущих практикумах:

	Тема	Технологический стек
	Маркетплейс (каталог,	Java Spring Boot + PostgreSQL +

	Тема	Технологический стек
	корзина, заказы, оплата)	Kafka + Redis
	Сервис заказа такси	Python FastAPI + PostgreSQL + RabbitMQ + Redis
	Система управления задачами (Task tracker)	Go Gin + PostgreSQL + WebSocket
	API Gateway для микросервисного приложения	Spring Cloud Gateway + Consul + Resilience4j
	Платформа для онлайн-обучения с рекомендательной системой	Django + PostgreSQL + Redis + ML-модель

3.2. Этапы выполнения проекта

Этап 1. Техническое задание (ТЗ)

Самостоятельная работа (8 часов):

Определить целевое назначение системы.

Собрать и проанализировать требования (функциональные и нефункциональные).

Выбрать технологический стек (язык, фреймворки, БД, брокер сообщений).

Составить план-график выполнения работ.

Результат: документ «Техническое задание» (1-2 стр.).

Этап 2. Проектирование архитектуры (8 часов)

Разработать диаграмму компонентов (C4 model).

Определить структуру базы данных (ER-диаграмма).

Описать API в формате OpenAPI 3.0 (Swagger).

Задокументировать ключевые архитектурные решения (ADR).

Результат: раздел «Архитектура» в отчёте, OpenAPI-спецификация.

Этап 3. Разработка кода (16 часов)

Создать репозиторий на GitHub/GitLab.

Реализовать бэкенд (REST API или gRPC).

Реализовать асинхронные обработчики (Kafka/RabbitMQ).

Настроить подключение к БД и кэшу (Redis).

Внедрить аутентификацию (JWT).

Рекомендации:

Следуйте принципам чистой архитектуры.

Пишите unit-тесты на критически важные модули (покрытие ≥70%).

Фиксируйте промежуточные результаты через Git (осмысленные коммиты).

Этап 4. Контейнеризация и CI/CD (10 часов)

Написать Dockerfile для каждого сервиса.

Составить docker-compose.yml для локального запуска всех сервисов (БД, брокер, кэш, приложение).

Настроить CI-пайплайн (GitHub Actions / GitLab CI):

сборка проекта;

запуск тестов;

сборка Docker-образов;

(опционально) публикация в Docker Hub / registry.

Настроить базовый мониторинг (Prometheus + Grafana) через docker-compose.

Результат: работающий docker-compose.yml, проходящий CI/CD пайплайн.

Этап 5. Тестирование и нагрузка (8 часов)

Написать интеграционные тесты (проверка взаимодействия сервисов).

Написать сценарий нагрузочного тестирования на k6 (или JMeter) – не менее 3 эндпоинтов.

Провести тест с ramp-up до 50-100 VU, замерить p95, p99 latency, ошибки.

Проанализировать результаты, при необходимости оптимизировать.

Результат: отчёт о тестировании (скриншоты графиков, выводы).

Этап 6. Документирование (10 часов)

Оформить README.md:

описание проекта;

инструкция по установке и запуску (docker-compose);

примеры API-запросов (curl или Postman-коллекция).

Сгенерировать и сохранить Swagger UI (OpenAPI).

Написать краткое руководство пользователя (1-2 страницы).

Подготовить слайды презентации (7-10 слайдов).

3.3. Требования к оформлению отчёта о проекте

Отчёт (текстовая часть) должен содержать:

Титульный лист (с указанием темы, ФИО студента, группы, руководителя).

Введение (цель, задачи, актуальность).

Анализ требований (функциональные, нефункциональные).

Архитектура (диаграммы, ADR, выбор стека).

Реализация (структура проекта, описание основных модулей, листинги кода не более 30% от объёма).

Тестирование (сценарии, результаты, метрики).

Развёртывание (инструкция по запуску, CI/CD).

Заключение (достигнутые результаты, выводы, перспективы).

Список использованных источников.

Приложение (исходный код ссылкой на репозиторий).

Объём отчёта: **20-30 страниц** (без учёта листингов).

4. ВОПРОСЫ ДЛЯ СОБЕСЕДОВАНИЯ ПО САМОСТОЯТЕЛЬНОМУ ИЗУЧЕНИЮ ЛИТЕРАТУРЫ

Ниже представлен перечень вопросов для проверки глубины освоения теоретического материала. Ответы на них должны быть продемонстрированы при сдаче курсового проекта (на защите или на собеседовании).

Раздел «Архитектура и проектирование»

1. Какие архитектурные паттерны вы использовали в проекте и почему?
2. В чём разница между монолитом и микросервисами? Когда что выбирать?
3. Что такое C4 model? Какой уровень диаграмм вы построили?
4. Какие паттерны обеспечения согласованности данных между сервисами вы знаете (SAGA, двухфазная фиксация)? Какой применили?
5. Что такое API Gateway? Какие задачи он решает?

Раздел «Технологический стек»

6. Почему вы выбрали именно этот язык/фреймворк для реализации проекта?
7. Как вы обеспечиваете безопасность (аутентификацию/авторизацию)?
8. Какие типы баз данных вы использовали (реляционные / документоориентированные / колоночные)? Почему?
9. Зачем в проекте нужен брокер сообщений (Kafka/RabbitMQ)? Какую задачу он решает?
10. Как вы управляете конфигурацией и секретами (переменные окружения, .env, Vault)?

Раздел «Разработка и тестирование»

11. Какими принципами чистого кода вы руководствовались?
12. Какие виды тестов вы написали (unit, integration, e2e)? Приведите пример.
13. Как вы проверяли производительность системы (нагрузочное тестирование)?

14. Какие инструменты статического анализа кода использовали (SonarQube, Checkstyle, ESLint)?
15. Как вы обеспечиваете качество API (контрактное тестирование, валидация OpenAPI)?

Раздел «DevOps и эксплуатация»

16. Опишите ваш CI/CD пайплайн. Какие стадии он включает?
17. Зачем нужна контейнеризация? Чем Docker отличается от виртуальной машины?
18. Как вы поднимаете все сервисы локально (docker-compose)?
19. Как вы мониторите работу приложения (метрики, логи, трейсинг)?
20. Какие риски при развёртывании в облаке вы учитывали?

Раздел «Управление проектом»

21. Как вы планировали работу над проектом? Использовали ли Scrum/Kanban?
22. Какой инструмент управления задачами применяли (Trello, YouTrack, Jira)?
23. Как вы документировали API (Swagger, Postman)?
24. Какую систему контроля версий использовали? Опишите стратегию ветвления (Git Flow / GitHub Flow).
25. Какие риски возникли в ходе выполнения проекта и как вы их смягчили?

Форма проверки: собеседование с преподавателем во время защиты или отдельная аттестация. Оценивается полнота, аргументированность и связь ответов с реальным проектом студента.

5. КРИТЕРИИ ОЦЕНИВАНИЯ КОМПЕТЕНЦИЙ

Итоговая оценка по дисциплине (дифференцированный зачёт) формируется на основе:

качества выполненного курсового проекта (код, документация);

результатов защиты (презентация, доклад, ответы на вопросы);

отчёта о самостоятельной работе (портфолио);

своевременности сдачи промежуточных этапов.

5.1. Оценка курсового проекта

Критерий	Максимальный балл
Соответствие техническому заданию	10
Архитектура и дизайн (обоснованность решений, масштабируемость)	15
Качество кода (стиль, тесты, документация в коде)	15
Функциональность (все запланированные возможности реализованы)	20
Контейнеризация и CI/CD (работоспособность)	10
Тестирование (наличие unit-тестов, нагрузочных сценариев, анализ)	10
Документация (README, OpenAPI, руководство пользователя)	10
Презентация и защита (доклад, слайды, ответы на вопросы)	10
Итого	**100**

5.2. Шкала перевода баллов в оценку

Баллы	Оценка (5-балльная)	Уровень сформированности компетенций
85 – 100	Отлично	Продвинутый (компетенции сформированы полностью, проект готов к промышленному использованию)
70 – 84	Хорошо	Базовый (компетенции сформированы, но есть незначительные недостатки)
56 – 69	Удовлетворительно	Пороговый (минимально допустимый уровень, проект работает с ограничениями)
0 – 55	Неудовлетворительно	Компетенции не сформированы (проект не сдан, не работает или не соответствует ТЗ)

5.3. Оценка формирования профессиональных компетенций (ПК)

Компетенция	Что проверяется	Критерий успешности
ПК-1 (управление проектами)	План-график, использование трекера задач, Git-	План выполнен, коммиты регулярны, задачи

Компетенция	Что проверяется	Критерий успешности
	flow	декомпозированы
ПК-2 (архитектура)	Диаграммы, ADR, выбор стека, паттерны	Архитектура соответствует требованиям, обоснована
ПК-3 (разработка)	Исходный код, API, БД	Код работает, покрыт тестами, соответствует стилю
ПК-4 (тестирование)	Unit-тесты, интеграционные тесты, нагрузочное тестирование	Покрытие ≥70%, есть нагрузочный сценарий, анализ
ПК-5 (сопровождение, DevOps)	Docker, CI/CD, развёртывание	docker-compose запускается, CI проходит успешно
ПК-6 (работа с данными)	Схема БД, запросы, индексы, интеграция данных	Схема нормализована, запросы оптимизированы
ПК-7 (DevOps, автоматизация)	Использование облачных сервисов, IaC, оркестрация	Есть автоматизация сборки/развёртывания
ПК-8 (ИИ/МО)	Применение ML-модели или	Модель интегрирована,

Компетенция	Что проверяется	Критерий успешности
	алгоритмов анализа данных (при наличии)	демонстрирует результат
ПК-9 (документирование, коммуникация)	README, OpenAPI, презентация, доклад	Документация полная, презентация понятная
ПК-10 (безопасность, право)	JWT, HTTPS, защита от инъекций, соблюдение лицензий	Нет критических уязвимостей, указаны зависимости

5.4. Типовые замечания, снижающие оценку

Отсутствие тестов или их низкое покрытие (менее 50%).

Отсутствие CI/CD пайплайна.

Документация неполная или отсутствует.

Проект не собирается или не запускается по инструкции.

На защите студент не может объяснить архитектурные решения.

Использование уязвимых версий библиотек (не устранены известные CVE).

5.5. Критерии получения оценки «отлично»

Проект полностью соответствует ТЗ.

Реализованы все заявленные микросервисы (не менее 2-3).

Есть асинхронное взаимодействие (Kafka/RabbitMQ).

Docker-контейнеризация всех компонентов.

CI/CD пайплайн с тестами и сборкой образов.

Покрытие unit-тестами ≥80%, нагрузочный тест проведён.

Документация включает OpenAPI, README, руководство пользователя.

Защита прошла успешно, на все вопросы даны полные ответы.

Дополнительно: реализована одна из сложных опций (мониторинг, трейсинг, canary-деплой, автогенерация клиента по OpenAPI).