

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

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

Оглавление

Введение.....	3
Лабораторная работа №1.....	15
Лабораторная работа №2.....	47
Лабораторная работа №3.....	71
Лабораторная работа №4.....	94
Лабораторная работа №5.....	117
Лабораторная работа №6.....	144
Лабораторная работа №7.....	170
Лабораторная работа №8.....	194
Лабораторная работа №9.....	217
Лабораторная работа №10.....	Ошибка! Закладка не определена.
Лабораторная работа №11.....	Ошибка! Закладка не определена.
Лабораторная работа №12.....	Ошибка! Закладка не определена.
Лабораторная работа №13.....	Ошибка! Закладка не определена.
Лабораторная работа №14.....	Ошибка! Закладка не определена.
Лабораторная работа №15.....	Ошибка! Закладка не определена.
Лабораторная работа №16.....	Ошибка! Закладка не определена.
Лабораторная работа №17.....	Ошибка! Закладка не определена.
Лабораторная работа №18.....	Ошибка! Закладка не определена.

Введение

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

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

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

Задачи лабораторных работ:

1. Освоение инструментария:

- Контейнеризация и оркестрация (Docker, Docker Compose)
- Балансировка нагрузки (Nginx, API Gateway)
- Базы данных (PostgreSQL, MongoDB, Cassandra, Redis)
- Очереди сообщений (RabbitMQ, Kafka)
- Нагрузочное тестирование (k6)
- Документирование архитектуры (PlantUML, C4 model, arc42)

2. Применение архитектурных паттернов:

- Горизонтальное масштабирование stateless-сервисов
- Репликация и шардирование данных
- Circuit Breaker, Retry, Timeout
- CQRS, Saga (хореография и оркестрация)

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

3. Экспериментальное исследование:

- Анализ CAP-теоремы на примерах CP и AP систем
- Сравнение синхронной и асинхронной репликации БД
- Сравнение монолитной и микросервисной архитектур
- Сравнение SQL и NoSQL баз данных
- Нагрузочное тестирование с разными сценариями

4. Формирование компетенций (ПК-5):

- Проектировать эксперименты по проверке устойчивости систем
- Планировать и проводить нагрузочное тестирование
- Анализировать результаты экспериментов
- Формулировать и внедрять корректирующие меры
- Повышать отказоустойчивость и устойчивость к сбоям

Структура лабораторного практикума

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

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

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

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

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

№	Название работы	Формируемые компетенции
	Репликация баз данных: синхронная и асинхронная	ПК-5 _{ид-1} ПК-5 _{ид-4} ПК-5 _{ид-5} ПК-5 _{ид-6}

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

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

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

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

4. Уровни сложности

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

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

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

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

Требования к выполнению лабораторных работ

1. Общие требования

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

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

2. Технические требования

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

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

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

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

3. Рекомендуемое программное обеспечение

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

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

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

Система оценивания

1. Балльная структура каждой работы

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

2. Шкала перевода баллов в оценку

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

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

3. Итоговая оценка за лабораторный практикум

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

$$\text{Оценка} = (\Sigma \text{ оценок за ЛР1–ЛР18}) / 18$$

Порядок выполнения и сдачи работ

1. Этапы выполнения

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

Порядок защиты

Защита лабораторной работы включает:

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

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

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

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

Структура отчёта по лабораторной работе

Каждый отчёт должен содержать следующие разделы:

Лабораторная работа №X. [Название работы]

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

Литература для подготовки

Основная литература

1. Ньюмен, С. Создание микросервисов / С. Ньюмен. — 2-е изд. — СПб.: Питер, 2024. — 622 с.
2. Ричардсон, К. Микросервисы : паттерны разработки и рефакторинга / К. Ричардсон. — СПб.: Питер, 2024. — 544 с.

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

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

5. 9.2. Дополнительная литература

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

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

8. 9.3. Интернет-ресурсы

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

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

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

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

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

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

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

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

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

Лабораторная работа №1

Документирование архитектуры с использованием C4 model

Цель работы: Сформировать практические навыки документирования архитектуры программных систем на четырёх уровнях детализации с использованием методологии C4 model (Context, Containers, Components, Code) и инструментальных средств PlantUML.

Формируемые компетенции: ПК-5 (

1. ТЕОРЕТИЧЕСКОЕ ОБОСНОВАНИЕ

1.1. Понятие архитектурного документирования и его роль в разработке высоконагруженных систем

Архитектурное документирование - это процесс создания и поддержания описания архитектуры программной системы. В контексте высоконагруженных и отказоустойчивых систем качественная архитектурная документация становится критически важной, так как позволяет:

- Обеспечить единое понимание системы всеми участниками команды (архитекторами, разработчиками, тестировщиками, DevOps-инженерами).
- Выявить узкие места и точки отказа на ранних этапах проектирования.
- Обосновать архитектурные решения и компромиссы (trade-offs) для заинтересованных сторон.
- Снизить риски при внесении изменений в систему.

Основные проблемы отсутствия архитектурной документации:

- Размывание границ ответственности между компонентами.
- Невозможность оценить влияние изменений.
- Сложность онбординга новых членов команды.
- Повторное возникновение уже решённых проблем.

1.2. Методология C4 model

C4 model (Context, Containers, Components, Code) — это подход к визуализации архитектуры программного обеспечения, предложенный Саймоном Брауном (Simon Brown). Название отражает четыре уровня детализации, каждый из которых предназначен для своей аудитории (рисунок 1).

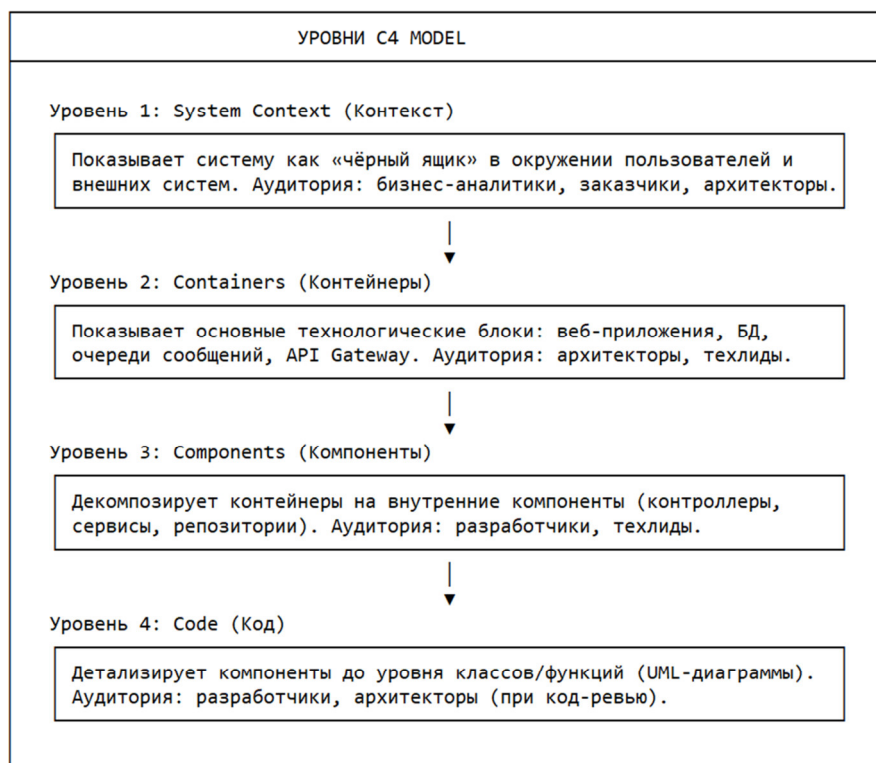


Рисунок 1 — Уровни детализации C4 model

1.3. Нотация C4 model

C4 model не предписывает строгую нотацию, но рекомендует использовать простые и понятные геометрические фигуры:

Элемент	Обозначение	Описание
	е	

Элемент	Обозначение	Описание
Пользователь / Актор	 или стик-фигура	Человек, использующий систему
Система / Контейнер	 Прямоугольник	Основной блок системы
Внешняя система	 Прямоугольник (серый фон)	Система, на которую мы не влияем
Компонент	 Прямоугольник с заголовком	Внутренний модуль контейнера
Взаимодействие	→ Стрелка (+ подпись)	Указывает протокол и действие

Принципы построения качественных диаграмм:

- Каждая диаграмма должна иметь заголовок, отражающий уровень и название системы.
- Все элементы должны быть подписаны (имя + краткое описание).
- Стрелки взаимодействия должны содержать аннотацию (протокол, действие).
- Цвета следует использовать минимально и осмысленно (например, внешние системы - серым).
-

1.4. Инструменты для реализации C4 model

Инструмент	Тип	Преимущества	Недостатки
PlantUML	Текстовый	Версионирuется	Портативный, требуется

Инструмент	Тип	Преимущества	Недостатки
		я в Git, интеграция с Markdown, бесплатно	установка или онлайн-сервис
Mermaid	Текстовый	Встроен в GitHub/GitLab Markdown, бесплатно	Меньше возможностей для сложных диаграмм
Draw.io	Графический	Интуитивный интерфейс, экспорт в XML	Сложнее версионировать
Structurizr	Специализированный	Нативная поддержка C4 model	Платная версия для продвинутого функционала

В данной работе используется PlantUML с библиотекой C4-PlantUML.

1.5. Синтаксис PlantUML для C4 model

Подключение библиотеки C4-PlantUML:

```
plantuml
```

```
' Для диаграммы контекста
```

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

```
' Для диаграммы контейнеров
```

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

' Для диаграммы компонентов

```
!include https://raw.githubusercontent.com/plantuml-stdlib/C4-PlantUML/master/C4_Component.puml
```

Базовые элементы:

Элемент	Синтаксис
Пользователь	Person(alias, "Имя", "Описание")
Система	System(alias, "Имя", "Описание")
Внешняя система	System_Ext(alias, "Имя", "Описание")
Контейнер	Container(alias, "Имя", "Технология", "Описание")
База данных	ContainerDb(alias, "Имя", "Технология", "Описание")
Компонент	Component(alias, "Имя", "Технология", "Описание")
Взаимодействие	Rel(от, к, "Описание", "Протокол")
Граница системы	System_Boundary(alias, "Название") { ... }

1.6. Примеры диаграмм C4 model

Пример 1. Диаграмма контекста (Level 1)

```
plantuml
```

```
@startuml
```

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

title Диаграмма контекста - Интернет-магазин

Person(customer, "Покупатель", "Клиент интернет-магазина")

Person(admin, "Администратор", "Управляет каталогом и заказами")

System(shop, "Интернет-магазин", "Позволяет оформлять заказы")

System_Ext(payment, "Платёжный шлюз", "Обрабатывает платежи")

System_Ext(email, "Email-сервер", "Отправляет уведомления")

Rel(customer, shop, "Оформляет заказ через", "HTTPS")

Rel(admin, shop, "Управляет через", "HTTPS")

Rel(shop, payment, "Отправляет запрос на списание", "API")

Rel(shop, email, "Отправляет email", "SMTP")

@enduml



Пример 2. Диаграмма контейнеров (Level 2)

plantuml

@startuml

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

title Диаграмма контейнеров — Интернет-магазин

Person(customer, "Покупатель")

System_Boundary(shop, "Интернет-магазин") {

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

Container(api, "API Gateway", "Kong", "Единая точка входа")

Container(orders, "Сервис заказов", "Java/Spring", "Бизнес-логика")

ContainerDb(db, "База данных", "PostgreSQL", "Хранилище заказов")

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

}

System_Ext(payment, "Платёжный шлюз")

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

Rel(web, api, "Вызывает", "REST")

Rel(api, orders, "Перенаправляет", "HTTP")

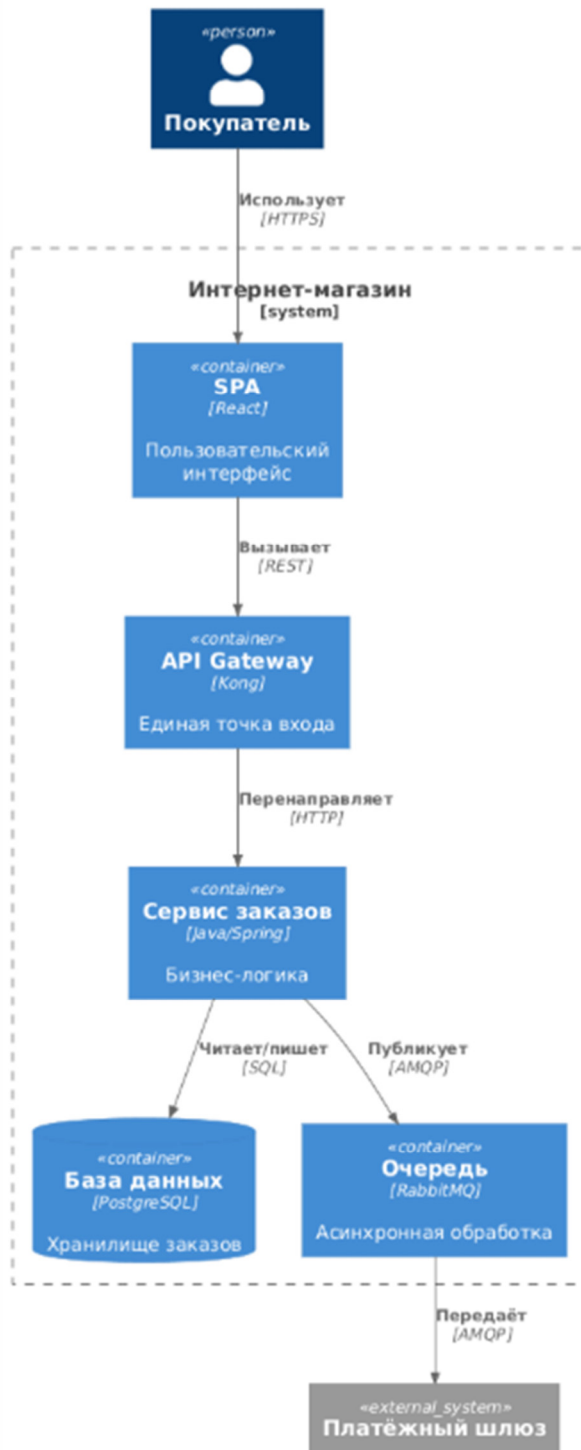
Rel(orders, db, "Читает/пишет", "SQL")

Rel(orders, queue, "Публикует", "AMQP")

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

@enduml

Диаграмма контейнеров — Интернет-магазин



Пример 3. Диаграмма компонентов (Level 3)

plantuml

@startuml

!include

https://raw.githubusercontent.com/plantuml-stdlib/C4-PlantUML/master/C4_Component.puml

PlantUML/master/C4_Component.puml

title Диаграмма компонентов — Сервис заказов

Container(orders, "Сервис заказов", "Java/Spring")

```
System_Boundary(orders_internal, "Сервис заказов") {  
    Component(controller, "OrderController", "REST", "Принимает HTTP-  
запросы")  
    Component(service, "OrderService", "Бизнес-логика", "Обрабатывает  
заказы")  
    Component(repo, "OrderRepository", "JPA", "Взаимодействие с БД")  
    Component(producer, "EventProducer", "RabbitMQ", "Публикует  
события")  
    Component(validator, "OrderValidator", "Валидация", "Проверка  
входных данных")  
}
```

Rel(controller, validator, "Вызывает")

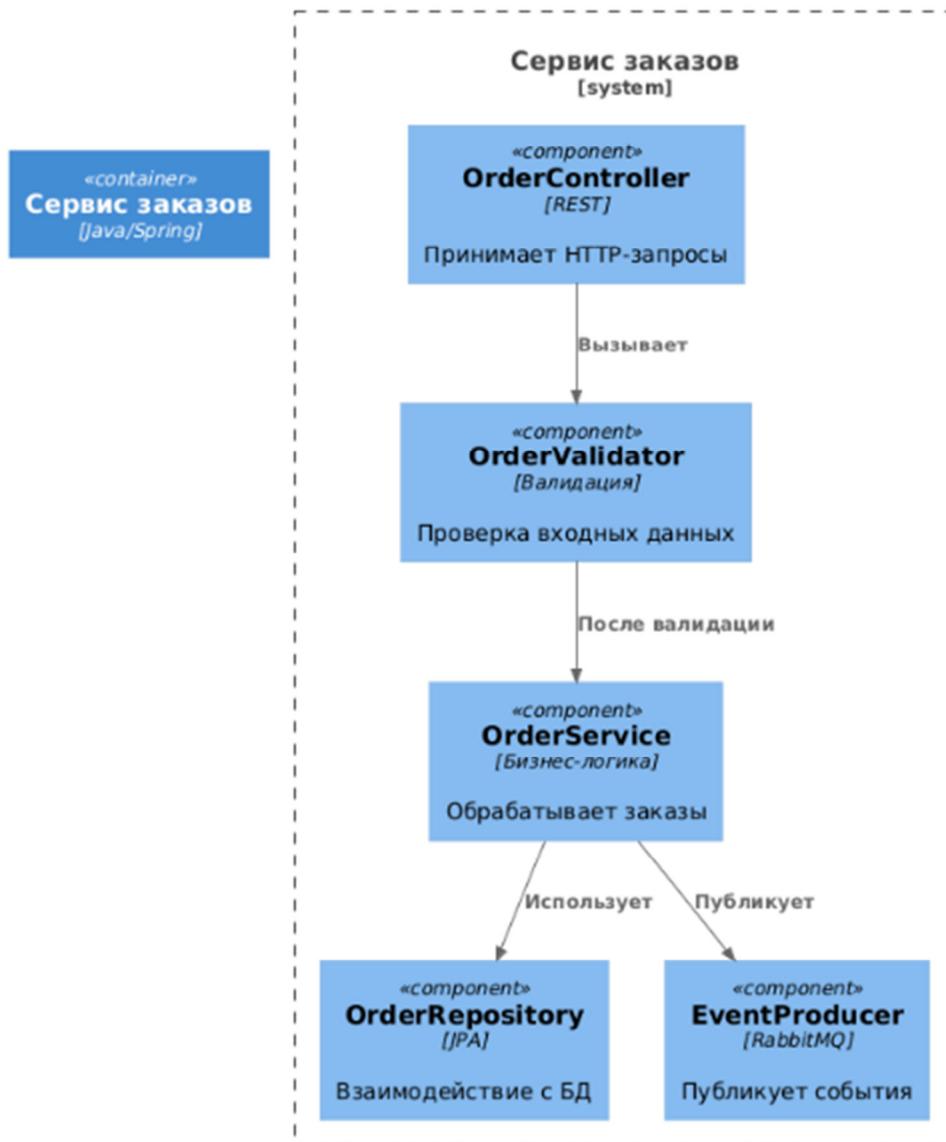
Rel(validator, service, "После валидации")

Rel(service, repo, "Использует")

Rel(service, producer, "Публикует")

@enduml

Диаграмма компонентов — Сервис заказов



1.7. Architecture Decision Record (ADR)

ADR (Architecture Decision Record) - это документ, фиксирующий важное архитектурное решение, его контекст, обоснование и последствия. ADR является неотъемлемой частью архитектурной документации.

Структура ADR (шаблон Мэддисона):

markdown

ADR-XXX: [Краткое название решения]

Статус

[Предложено / Принято / Устарело / Заменено]

Контекст

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

Решение

Описание выбранного подхода или технологии.

Обоснование

Почему выбрано именно это решение (trade-offs, сравнение альтернатив).

Последствия

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

2. ПРИМЕРЫ ПРАКТИЧЕСКОГО ВЫПОЛНЕНИЯ

2.1. Пример для варианта 1 (базовый уровень)

Система: Онлайн-бронирование переговорных комнат (без дополнительных интеграций)

2.1.1. Диаграмма контекста (Level 1)

Код PlantUML:

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

title Диаграмма контекста — Система бронирования переговорных комнат
```

Person(employee, "Сотрудник", "Пользователь, бронирующий переговорные комнаты")

Person(admin, "Администратор", "Управляет комнатами и просматривает статистику")

System(booking_system, "Система бронирования", "Позволяет бронировать комнаты, управлять расписанием")

System_Ext(email, "Email-сервер", "Отправляет уведомления о бронировании")

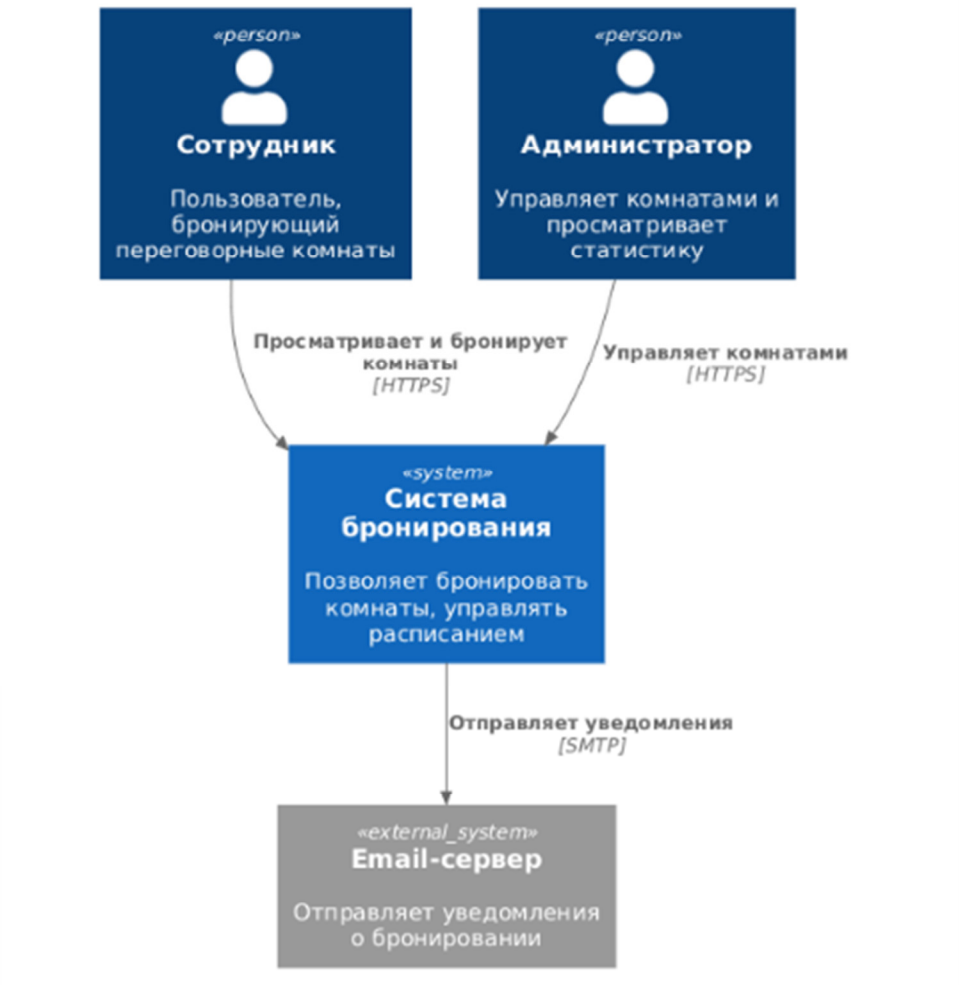
Rel(employee, booking_system, "Просматривает и бронирует комнаты", "HTTPS")

Rel(admin, booking_system, "Управляет комнатами", "HTTPS")

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

@enduml

Диаграмма контекста — Система бронирования переговорных комнат



Описание диаграммы:

Элемент	Тип	Описание
Сотрудник	Пользователь	Конечный пользователь, взаимодействует с системой для бронирования
Администратор	Пользователь	Управляет комнатами (CRUD)
Система бронирования	Разрабатываемая система	Основной функционал бронирования
Email-сервер	Внешняя	Отправляет уведомления (не

Элемент	Тип	Описание
	система	управляется нами)

Взаимодействия:

- Сотрудник → Система бронирования: просмотр и бронирование (HTTPS)
- Администратор → Система бронирования: управление комнатами (HTTPS)
- Система бронирования → Email-сервер: отправка уведомлений (SMTP)

2.1.2. Диаграмма контейнеров (Level 2)

Код PlantUML:

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

title Диаграмма контейнеров — Система бронирования переговорных комнат

Person(employee, "Сотрудник")
Person(admin, "Администратор")

System_Boundary(booking, "Система бронирования переговорных") {
    Container(web, "Веб-приложение", "React", "SPA для бронирования комнат")
    Container(api, "API Gateway", "NGINX", "Единая точка входа, маршрутизация")
    Container(backend, "Backend API", "Python/FastAPI", "Бизнес-логика бронирования")
}
```

ContainerDb(db, "База данных", "PostgreSQL", "Хранилище пользователей, комнат, бронирований")

Container(queue, "Очередь сообщений", "RabbitMQ", "Асинхронная отправка email")

Container(worker, "Worker", "Python", "Обработчик очереди email")
}

System_Ext(email, "Email-сервер")

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

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

Rel(web, api, "Вызывает API", "HTTPS/REST")

Rel(api, backend, "Перенаправляет запросы", "HTTP")

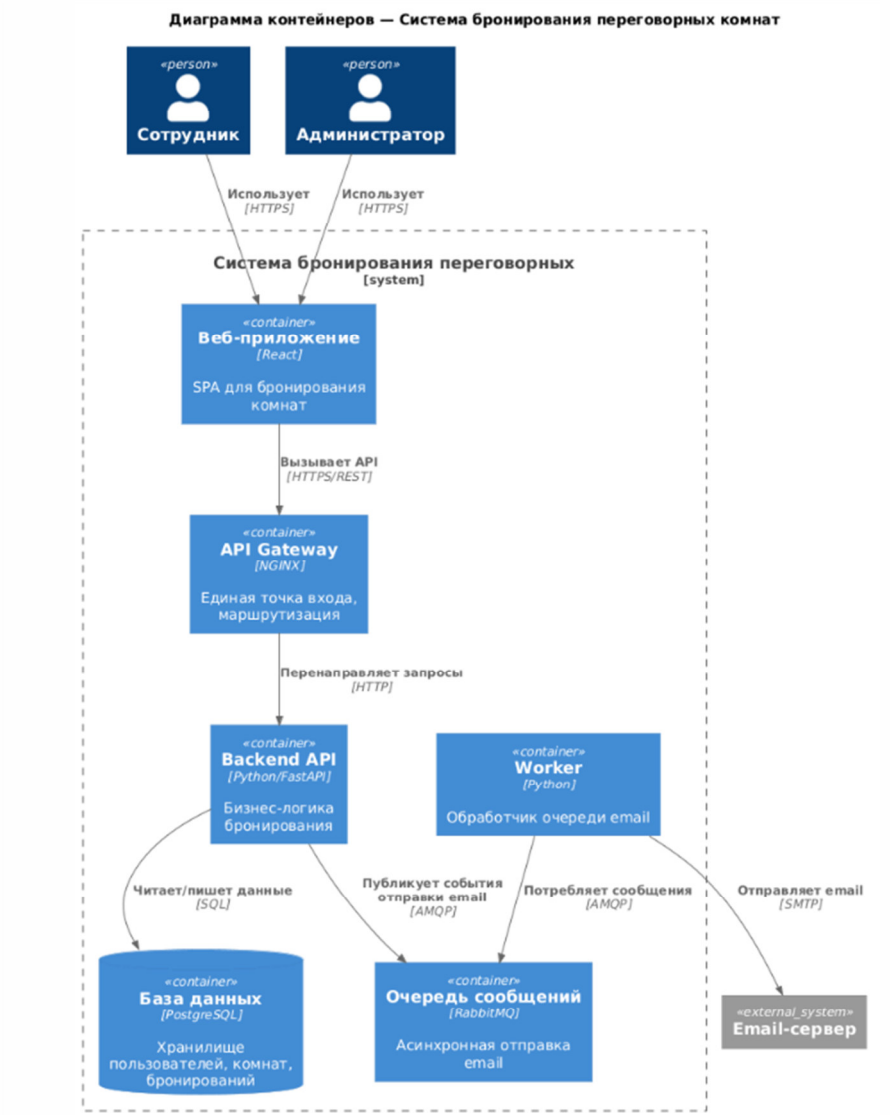
Rel(backend, db, "Читает/пишет данные", "SQL")

Rel(backend, queue, "Публикует события отправки email", "AMQP")

Rel(worker, queue, "Потребляет сообщения", "AMQP")

Rel(worker, email, "Отправляет email", "SMTP")

@enduml



Описание контейнеров:

Контейнер	Технология	Назначение
Веб-приложение	React	SPA для просмотра доступности и бронирования
API Gateway	NGINX	Маршрутизация, балансировка, rate limiting
Backend API	Python/FastAPI	Бизнес-логика: проверка доступности, бронирований, CRUD

Контейнер	Технология	Назначение
База данных	PostgreSQL	Хранение пользователей, комнат, бронирований
Очередь сообщений	RabbitMQ	Асинхронная передача задач на отправку email
Worker	Python	Потребление очереди, реальная отправка email

2.2. Пример для среднего уровня (добавляется Level 3)

Диаграмма компонентов (Level 3) для Backend API:

@startuml

!include https://raw.githubusercontent.com/plantuml-stdlib/C4-PlantUML/master/C4_Component.puml

title Диаграмма компонентов — Backend API (сервис бронирования)

Container(backend, "Backend API", "Python/FastAPI")

```
System_Boundary(api_boundary, "Backend API") {
    Component(router, "API Router", "FastAPI", "Маршрутизация HTTP-запросов")
    Component(controller, "BookingController", "Обработчик", "Принимает запросы по бронированию")
    Component(room_ctrl, "RoomController", "Обработчик", "Управление комнатами")
    Component(service, "BookingService", "Бизнес-логика", "Проверка доступности, создание брони")
}
```

Component(validator, "AvailabilityValidator", "Сервис", "Проверка
пересечения бронирований")

Component(repo, "BookingRepository", "SQLAlchemy", "CRUD
бронирований в БД")

Component(room_repo, "RoomRepository", "SQLAlchemy", "CRUD
комнат в БД")

Component(producer, "NotificationProducer", "aio-pika", "Публикация
сообщений в RabbitMQ")

}

Rel(router, controller, "Маршрутизирует")

Rel(router, room_ctrl, "Маршрутизирует")

Rel(controller, validator, "Вызывает")

Rel(controller, service, "Вызывает")

Rel(validator, repo, "Читает")

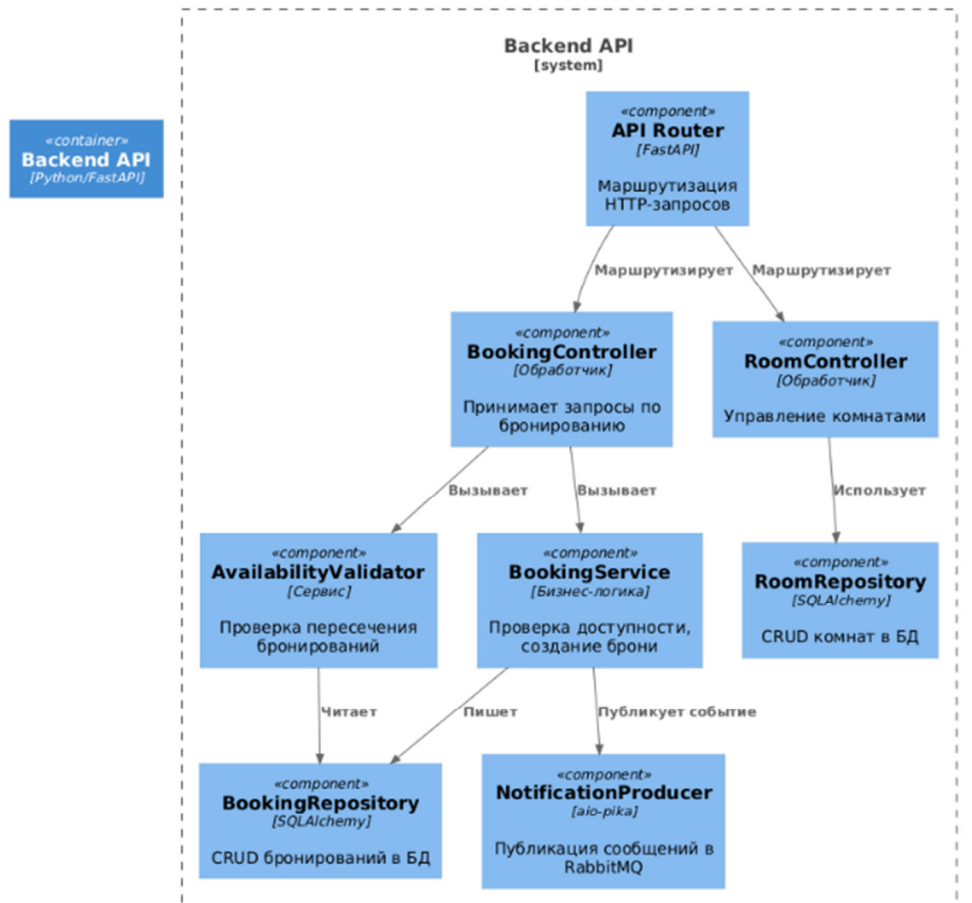
Rel(service, repo, "Пишет")

Rel(service, producer, "Публикует событие")

Rel(room_ctrl, room_repo, "Использует")

@enduml

Диаграмма компонентов — Backend API (сервис бронирования)



2.3. Пример для продвинутого уровня (добавляется ADR)

ADR-001: Выбор базы данных

markdown

ADR-001: Выбор базы данных для системы бронирования переговорных

Статус

Принято

Контекст

Система бронирования переговорных комнат должна хранить:

- Пользователей (до 1000 записей)
- Комнаты (до 50 записей)
- Бронирования (до 5000 активных записей, 50 000 в архиве)

Требования:

- Целостность данных (нельзя забронировать одну комнату дважды)
- Поддержка конкурентного доступа (до 500 RPS)
- Транзакционность при создании бронирования (проверка + запись)

Решение

Использовать PostgreSQL в качестве основной базы данных.

Обоснование

Рассматривались альтернативы:

1. ****MongoDB**** — не обеспечивает транзакционную целостность на уровне, требуемом для бронирования (риск двойного бронирования).
2. ****Cassandra**** — оптимизирована на запись, но слаба в сложных запросах (проверка пересечений по времени).
3. ****PostgreSQL**** — ACID-транзакции, мощные индексы для временных запросов, зрелый инструментарий.

Последствия

- ****Положительные:**** целостность данных, поддержка сложных запросов, надёжность.
- ****Отрицательные:**** сложнее горизонтальное масштабирование (потребуется репликация и шардирование в будущем).
- ****Компромисс:**** выбрана консистентность в ущерб лёгкости масштабирования.

МЕТОДИЧЕСКИЕ УКАЗАНИЯ ДЛЯ ВЫПОЛНЕНИЯ ПРАКТИЧЕСКОЙ ЧАСТИ РАБОТЫ

Задачи работы:

1. **Анализ предметной области:**

- 1.1. Изучить описание системы и дополнительные требования согласно варианту.
- 1.2. Определить границы системы (что входит в разрабатываемую систему, а что является внешним).
- 1.3. Выявить всех пользователей и внешние системы.
- 1.4. Построение диаграммы контекста (Level 1):
- 1.5. Определить основных пользователей (акторов) системы.
- 1.6. Определить внешние системы, с которыми взаимодействует разрабатываемая система.
- 1.7. Построить диаграмму в PlantUML с использованием библиотеки C4-PlantUML.
- 1.8. На каждой стрелке указать протокол взаимодействия (HTTPS, AMQP, SMTP и т.д.).

2. Построение диаграммы контейнеров (Level 2):

- 2.1. Выполнить декомпозицию системы на технологические блоки (контейнеры).
- 2.2. Для каждого контейнера указать технологию (React, FastAPI, PostgreSQL, RabbitMQ).
- 2.3. Показать взаимодействие между контейнерами с указанием протоколов.
- 2.4. Для базового уровня — достаточно 2-х контейнеров (Frontend + Backend + DB).
- 2.5. Для среднего и продвинутого уровня — добавить очередь сообщений, кэш, API Gateway.

3. Построение диаграммы компонентов (Level 3) — для среднего и продвинутого уровня:

- 3.1. Выбрать один ключевой контейнер (обычно Backend API).
- 3.2. Декомпонизировать его на внутренние компоненты (контроллеры, сервисы, репозитории).

3.3. Показать взаимодействие между компонентами.

4. **Построение диаграммы кода (Level 4) и ADR — для продвинутого уровня:**

- 4.1. Выбрать один ключевой компонент и построить диаграмму классов UML (в PlantUML).
- 4.2. Разработать минимум 2 ADR-документа для ключевых архитектурных решений.
- 4.3. Оформление отчёта:
- 4.4. Вставить все диаграммы в формате PlantUML (текстовое представление).
- 4.5. Добавить скриншоты визуализированных диаграмм (можно сгенерировать через онлайн-сервис PlantUML).
- 4.6. Дать текстовое описание каждого элемента диаграмм.
- 4.7. Ответить на контрольные вопросы.

3. ВАРИАНТЫ ЗАДАНИЙ

3.1. Базовый уровень (15 вариантов)

Общее описание системы: Система онлайн-бронирования переговорных комнат (базовая конфигурация).

Требования ко всем вариантам базового уровня:

- Построить диаграмму контекста (Level 1) с указанием всех пользователей и внешних систем.
- Построить диаграмму контейнеров (Level 2) с декомпозицией на технологические блоки.
- Минимальный набор контейнеров: Веб-приложение + Backend API + База данных.

№ варианта	Дополнительные требования	Внешние системы
1	Базовая система (без	Email-сервер

№ варианта	Дополнительные требования	Внешние системы
	доп. интеграций)	
2	Интеграция с Google Calendar	Email-сервер, Google Calendar API
3	Интеграция с Outlook	Email-сервер, Microsoft Graph API
4	Двухфакторная аутентификация	Email-сервер, SMS-шлюз
5	Уведомления в Telegram	Email-сервер, Telegram Bot API
6	Интеграция с Active Directory	Email-сервер, LDAP/AD
7	Платные бронирования	Email-сервер, Платёжный шлюз
8	Биллинг и отчётность	Email-сервер, BI-система
9	Резервное копирование	Email-сервер, S3-хранилище
0	1 Мониторинг	Email-сервер, Prometheus + Grafana
1	1 Кэширование свободных комнат	Email-сервер, Redis
2	1 Полнотекстовый поиск комнат	Email-сервер, Elasticsearch

№ варианта	Дополнительные требования	Внешние системы
3	1 Push-уведомления на мобильные	Email-сервер, FCM
4	1 Интеграция с видеоконференциями	Email-сервер, Zoom API
5	1 SLA-мониторинг доступности	Email-сервер, Uptime Kuma

3.2. Средний уровень (15 вариантов)

Общее описание системы: Система электронного документооборота (СЭД) для малого предприятия.

Требования ко всем вариантам среднего уровня:

- Построить диаграмму контекста (Level 1)
- Построить диаграмму контейнеров (Level 2) — минимум 4 контейнера
- Построить диаграмму компонентов (Level 3) для контейнера «Backend API»
- Предоставить текстовое описание всех компонентов

№	Тип СЭД	Дополнительные интеграции
1	Договорная	ЭЦП, Контур.Диадок
2	Кадровая	1С:Зарплата и управление персоналом
3	Счётная	Битрикс24
4	Архивная	Почта России (API)
5	Проектная	Jira, Confluence

№	Тип СЭД	Дополнительные интеграции
6	Клиентская	АmoCRM
7	Финансовая	МоегоСклад
8	Закупочная	ЕИС (zakupki.gov.ru)
9	Отчётная	Контур.Экстерн
10	Складская	WMS-система
11	Юридическая	ГАРАНТ / КонсультантПлюс
12	Транспортная	ГЛОНАСС / ЭРА-ГЛОНАСС
13	Производственная	MES-система
14	Аптечная	ЕГИСЗ
15	Ритейл	Меркурий (ФГИС)

3.3. Продвинутый уровень (15 вариантов)

Общее описание системы: Геораспределённая платформа доставки еды (аналог Delivery Hero / Яндекс.Еда).

Требования ко всем вариантам продвинутого уровня:

1. Построить все 4 уровня C4 model (Context, Containers, Components, Code)
2. Разработать минимум 2 ADR-документа
3. Предоставить полную архитектурную документацию в формате Markdown
4. Обосновать выбор ключевых технологий

№	Особенность платформы	Количество регионов
1	Только курьерская доставка	1 регион
2	Самовывоз + курьерская	2 региона
3	Интеграция с ресторанными агрегаторами	3 региона
4	Дронодоставка (последняя миля)	1 регион
5	Подписка (Dark Store)	5 регионов
6	B2B доставка для офисов	3 региона
7	Прогнозирование времени доставки (ML)	2 региона
8	Динамическое ценообразование	4 региона
9	Роботы-доставщики	1 регион
10	Кроссплатформенное приложение (Flutter/RN)	3 региона
11	Офлайн-режим (буферизация на клиенте)	2 региона
12	Бонусная программа (Loyalty)	5 регионов
13	Маркетплейс (рестораны + магазины)	3 региона
14	Корпоративная доставка (интеграция с 1С)	2 региона
15	Платформа с готовой едой (dark kitchen)	4 региона

4. КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Что такое C4 model и для чего она используется в проектировании высоконагруженных систем?
2. Перечислите 4 уровня детализации C4 model и охарактеризуйте аудиторию каждого уровня.
3. Чем отличается диаграмма контекста (Level 1) от диаграммы контейнеров (Level 2)?
4. Какая информация обязательно должна быть указана на стрелках взаимодействия между элементами?
5. Почему для высоконагруженных систем важно документировать архитектуру с использованием C4 model?
6. Какие преимущества даёт использование текстовых инструментов (PlantUML) перед графическими редакторами для построения архитектурных диаграмм?
7. Что такое ADR (Architecture Decision Record) и какие разделы он должен содержать?
8. Как на диаграмме C4 отличить внешнюю систему от разрабатываемой системы?
9. В чём разница между контейнером и компонентом в C4 model?
10. Какие технологии могут выступать в роли контейнеров в диаграмме уровня 2?
11. Какую информацию несёт подпись на контейнере (например, «Python/FastAPI»)?
12. Что такое «границы системы» (System Boundary) и зачем они нужны?
13. Какие ошибки чаще всего встречаются при построении C4 диаграмм начинающими?
14. Почему не рекомендуется использовать строгую нотацию UML для всех уровней C4?
15. Как C4 model связана с документированием архитектурных рисков и компромиссов?

5. ТРЕБОВАНИЯ К ОТЧЕТУ

Отчёт должен быть оформлен в виде **Markdown-документа** и содержать следующие разделы:

1. **Титульный лист** с указанием:
 - ФИО студента
 - Номер группы
 - Номер варианта
 - Название работы
2. **Цель работы** — формулируется на основе задания.
3. **Теоретическая часть:**
 - Краткое описание методологии C4 model.
 - Объяснение выбранного инструмента (PlantUML).
 - Описание синтаксиса ключевых элементов.
4. **Практическая часть:**
 - Анализ предметной области (краткое описание системы по варианту).
 - Листинг кода диаграммы контекста (Level 1) с комментариями.
 - Листинг кода диаграммы контейнеров (Level 2) с комментариями.
 - *Для среднего и продвинутого уровня:* листинг диаграммы компонентов (Level 3).
 - *Для продвинутого уровня:* листинг диаграммы кода (Level 4) и ADR-документы.
5. **Результаты выполнения:**
 - Скриншоты визуализированных диаграмм (или встроенный рендер PlantUML).
 - Текстовое описание каждого элемента диаграмм.
 - Таблица взаимодействий (при необходимости).
6. **Анализ возможных ошибок** (если возникали при построении):
 - Описание ошибки.

- Причина возникновения.
- Способ устранения.

7. **Ответы на контрольные вопросы** (3-5 вопросов по выбору преподавателя).

8. **Вывод** - что удалось освоить в ходе работы, какие навыки приобретены, какие архитектурные компромиссы выявлены.

6. КРИТЕРИИ ОЦЕНИВАНИЯ

Шкала оценивания по уровням сложности

Критерий	Базовый уровень (макс. 10 баллов)	Средний уровень (макс. 14 баллов)	Продвинутый уровень (макс. 18 баллов)
Диаграмма контекста (Level 1) корректна	3 балла	3 балла	2 балла
Диаграмма контейнеров (Level 2) корректна	4 балла	4 балла	3 балла
Диаграмма компонентов (Level 3) корректна	—	4 балла	3 балла
Диаграмма кода (Level 4) корректна	—	—	3 балла

Критерий	Базовый уровень (макс. 10 баллов)	Средний уровень (макс. 14 баллов)	Продвинутый уровень (макс. 18 баллов)
ADR-документы (минимум 2)	—	3 балла	3 балла
Текстовое описание элементов диаграмм	1 балл	—	—
Качество отчёта и ответы на вопросы	2 балла	—	2 балла
Максимальный балл	10	14	18

Шкала перевода баллов в оценку

Оценка	Базовый уровень	Средний уровень	Продвинутый уровень
Отлично (5)	9-10 баллов	13-14 баллов	16-18 баллов
Хорошо (4)	7-8 баллов	10-12 баллов	12-15 баллов
Удовлетворитель- но (3)	5-6 баллов	7-9 баллов	8-11 баллов

Оценка	Базовый уровень	Средний уровень	Продвинутый уровень
Неудовлетворительно (2)	менее 5 баллов	менее 7 баллов	менее 8 баллов

Дополнительные критерии (штрафы)

Нарушение	Штраф
Отсутствие скриншотов визуализированных диаграмм	-1 балл
Отсутствие листингов PlantUML кода	-1 балл
Не указаны протоколы на стрелках взаимодействия	-2 балла
Синтаксические ошибки в PlantUML (диаграмма не рендерится)	-2 балла
Выбран неверный уровень детализации (контейнер вместо компонента и т.п.)	-2 балла
Отсутствие вывода по работе	-1 балл
Несоответствие варианту задания	работа не засчитывается

7. ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ

Программа	Версия	Назначение	Доступность в РФ
PlantUML	любая	Построение диаграмм	Open Source

Программа	Версия	Назначение	Доступность в РФ
		по текстовому описанию	
VS Code + плагин PlantUML	последняя	Редактор кода с предпросмотром диаграмм	+
Git	последняя	Система контроля версий (опционально)	+
Draw.io (опционально)	Desktop/Web	Альтернативный графический редактор диаграмм	+
Mermaid (опционально)	любой плагин	Альтернативный инструмент для диаграмм в Markdown	+

Примечание: Все указанные программные продукты являются открытыми (open-source) или бесплатными для использования в образовательных целях на территории Российской Федерации.

8. ПРИМЕР ЗАПУСКА PlantUML

Вариант 1: Онлайн-сервис (без установки)

1. Перейти на сайт: <https://www.plantuml.com/plantuml/uml/>
2. Вставить код диаграммы в левую панель.
3. Получить визуализированное изображение справа.
4. Сохранить скриншот или экспортировать изображение.

Вариант 2: Локальный запуск (VS Code)

1. Установить расширение «PlantUML» в VS Code.

2. Создать файл с расширением .puml или .plantuml.
3. Вставить код диаграммы.
4. Нажать Alt + D для предпросмотра.
5. Экспортировать изображение через контекстное меню.

Лабораторная работа №2

Проектирование и развертывание горизонтально масштабируемого stateless-сервиса

Цель работы: Сформировать практические навыки проектирования, разработки и развертывания горизонтально масштабируемого stateless-сервиса с использованием контейнеризации (Docker) и балансировки нагрузки (Nginx).

Формируемые компетенции: ПК-5

1. ТЕОРЕТИЧЕСКОЕ ОБОСНОВАНИЕ

1.1. Понятие stateless-сервиса и его роль в высоконагруженных системах

Stateless-сервис (сервис без сохранения состояния) — это приложение, которое не хранит информацию о клиенте между запросами. Каждый запрос обрабатывается независимо, как будто он первый и единственный.

Ключевые характеристики stateless-сервиса:

Характеристика	Описание
Отсутствие сессий	Сервер не сохраняет состояние клиента между запросами
Независимость запросов	Каждый запрос содержит всю необходимую информацию для обработки
Идемпотентность (опционально)	Повторное выполнение операции не изменяет результат

Характеристика	Описание
Легкость масштабирования	Можно добавлять/убирать экземпляры без синхронизации состояния

В отличие от stateful-сервиса: Stateful-сервис хранит состояние (например, сессии пользователей, временные данные, кэш), что требует синхронизации между экземплярами и усложняет горизонтальное масштабирование.

Примеры stateless vs stateful:

Stateless (хорошо для масштабирования)	Stateful (требуется специальных решений)
REST API (JWT-токены вместо сессий)	База данных (PostgreSQL, MySQL)
Обработчики веб-хуков	Кэш (Redis)
Функции в Serverless (AWS Lambda)	Очереди сообщений (RabbitMQ)
Микросервисы без локального кэша	Брокеры сообщений (Kafka)

1.2. Вертикальное vs горизонтальное масштабирование

Характеристика	Вертикальное масштабирование	Горизонтальное масштабирование
Суть	Увеличение мощности одного сервера (CPU, RAM, SSD)	Добавление новых серверов/экземпляров

Характеристика	Вертикальное масштабирование	Горизонтальное масштабирование
Stateless подходит?	Да, но быстро достигает предела	Идеально подходит
Stateful подходит?	Да, но дорого	Требует шардирования/репликации
Лимиты	Ограничено железом (max 64-128 core)	Практически безгранично
Отказоустойчивость	Нет (один сервер - SPOF)	Есть (остальные серверы работают)
Стоимость	Нелинейно растёт (суперкомпьютеры дороги)	Линейно растёт

1.3. Контейнеризация с Docker

Docker - платформа для контейнеризации приложений, позволяющая упаковывать приложения со всеми зависимостями в изолированные контейнеры.

Преимущества контейнеризации для масштабирования:

- **Идемпотентность:** контейнер полностью описывается Dockerfile (инфраструктура как код).
- **Лёгкость:** контейнеры делят ядро ОС, запускаются за секунды.
- **Воспроизводимость:** образ работает одинаково на всех окружениях (локальная разработка → CI/CD → production).

Базовая структура Dockerfile для Python:

```
dockerfile
```

```
# Базовый образ с Python
```

FROM python:3.11-slim

Установка рабочей директории

WORKDIR /app

Копирование файла зависимостей

COPY requirements.txt .

Установка зависимостей

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

Копирование исходного кода

COPY . .

Указание порта, который слушает приложение

EXPOSE 8000

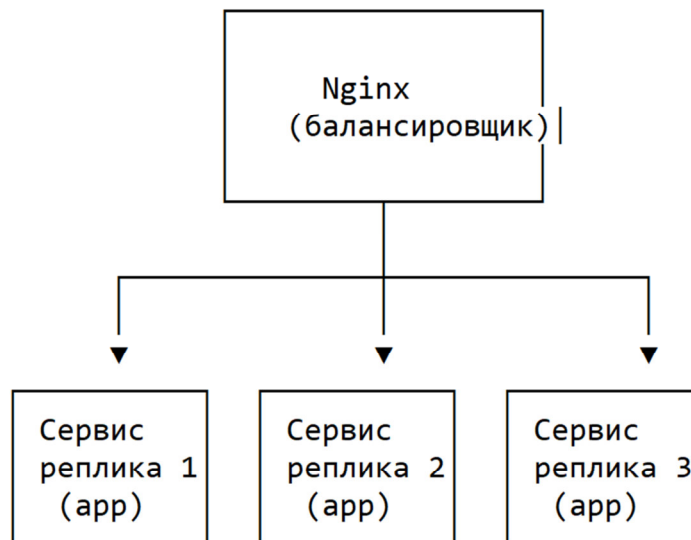
Команда запуска

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

1.4. Балансировка нагрузки с Nginx

Балансировка нагрузки — распределение входящих запросов между несколькими серверами (бэкендами).

Архитектура с Nginx:



Конфигурация балансировки в Nginx:

```
upstream backend {  
    # Алгоритм round-robin (по умолчанию)  
    server app:8000;  
    server app:8000;  
    server app:8000;  
}  
  
server {  
    listen 80;  
  
    location / {  
        proxy_pass http://backend;  
        proxy_set_header Host $host;  
        proxy_set_header X-Real-IP $remote_addr;  
    }  
}
```

Основные алгоритмы балансировки:

Алгоритм	Директива	Описание
Round-robin	(по умолчанию)	Циклическое распределение запросов

Алгоритм	Директива	Описание
Least connections	least_conn;	Запрос направляется серверу с наименьшим числом активных соединений
IP hash	ip_hash;	Один клиент всегда попадает на один сервер (для сессий)
Weighted	server app:8000 weight=3;	Учёт «веса» сервера (разная мощность)

1.5. Оркестрация контейнеров с Docker Compose

Docker Compose - инструмент для определения и запуска многоконтейнерных приложений. Позволяет описать все сервисы (реплики приложения, балансировщик, БД) в одном файле docker-compose.yml.

Пример docker-compose.yml для stateless-сервиса:

```
version: '3.8'

services:
  app:
    build: .
    ports:
      - "8000"
    # Без сохранения состояния – нет volumes для данных
    deploy:
      replicas: 3
    environment:
      - REDIS_URL=redis://cache:6379 # Важно: состояние вынесено в Redis!

  nginx:
    image: nginx:alpine
    ports:
      - "80:80"
    volumes:
      - ./nginx.conf:/etc/nginx/conf.d/default.conf
    depends_on:
      - app

  cache: # Пример stateful-сервиса (вынесен отдельно)
```

```
image: redis:alpine
ports:
  - "6379"
```

1.6. Проверка горизонтального масштабирования

Способы проверки распределения нагрузки:

1. **Логи реплик:** запустить несколько реплик с разным идентификатором (HOSTNAME).
2. **Нагрузочное тестирование:** утилиты wrk, ab (Apache Bench), k6.
3. **Метрики Nginx:** stub_status модуль показывает активные соединения.

Пример простого теста с wrk:

```
# Установка wrk
sudo apt install wrk

# Запуск теста на 10 секунд, 12 потоков, 400 соединений
wrk -t12 -c400 -d10s http://localhost/
```

1.7. Типовые ошибки при масштабировании

Ошибка	Симптом	Решение
Хранение состояния на диске в контейнере	После перезапуска реплики данные теряются	Вынести состояние в отдельную БД/кэш (PostgreSQL, Redis)
Локальный сессионный кэш	Пользователь «перескакивает» между репликами	Использовать Redis для кэша (Central Cache)
Жёстко заданные URL соседних	При контейнеризации	Использовать service discovery или переменные окружения

Ошибка	Симптом	Решение
сервисов (localhost:8000)	адреса меняются	
Слишком маленькое количество реплик (1)	При отказе одной реплики сервис недоступен	Минимум 2-3 реплики в production

2. ПРИМЕРЫ ПРАКТИЧЕСКОГО ВЫПОЛНЕНИЯ

2.1. Пример для Python (FastAPI) с Docker Compose

2.1.1. Структура проекта

text

myapp/

```

├── app.py          # Основное приложение
├── requirements.txt # Зависимости Python
├── Dockerfile      # Инструкция сборки образа
├── docker-compose.yml # Оркестрация сервисов
├── nginx.conf      # Конфигурация балансировщика
└── test.py         # Простой тест для проверки

```

2.1.2. Исходный код приложения (app.py)

```

import os
import socket
from fastapi import FastAPI
from pydantic import BaseModel

app = FastAPI(title="Stateless Service Demo")

class HealthResponse(BaseModel):
    status: str
    hostname: str

class InfoResponse(BaseModel):
    message: str

```

```

hostname: str

@app.get("/health", response_model=HealthResponse)
async def health_check():
    """Эндпоинт для проверки работоспособности"""
    return HealthResponse(
        status="ok",
        hostname=socket.gethostname()
    )

@app.get("/", response_model=InfoResponse)
async def root():
    """Главный эндпоинт, показывающий реплику"""
    return InfoResponse(
        message=f"Hello from stateless service!",
        hostname=socket.gethostname()
    )

@app.get("/compute/{number}")
async def compute(number: int):
    """Вычислительный эндпоинт без состояния"""
    result = number * number
    return {
        "input": number,
        "result": result,
        "processed_by": socket.gethostname()
    }

```

2.1.3. Зависимости (requirements.txt)

```

fastapi==0.104.1
uvicorn[standard]==0.24.0
pydantic==2.5.0

```

2.1.4. Dockerfile

```

FROM python:3.11-slim

WORKDIR /app

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

COPY . .

EXPOSE 8000

CMD ["uvicorn", "app:app", "--host", "0.0.0.0", "--port", "8000"]

```

2.1.5. Конфигурация Nginx (nginx.conf)

```
upstream backend {
    # Алгоритм round-robin (по умолчанию)
    server app:8000;
    server app:8000;
    server app:8000;
}

server {
    listen 80;

    location / {
        proxy_pass http://backend;
        proxy_set_header Host $host;
        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
    }

    # Статусная страница Nginx (опционально)
    location /nginx_status {
        stub_status on;
        allow 127.0.0.1;
        deny all;
    }
}
```

2.1.6. Docker Compose (docker-compose.yml)

```
version: '3.8'

services:
  app:
    build: .
    ports:
      - "8000"
    # Важно: нет volumes для сохранения состояния!
    environment:
      - PYTHONUNBUFFERED=1
    deploy:
      mode: replicated
      replicas: 3
    healthcheck:
      test: ["CMD", "curl", "-f", "http://localhost:8000/health"]
      interval: 10s
      timeout: 5s
      retries: 3

  nginx:
    image: nginx:alpine
    ports:
      - "80:80"
    volumes:
      - ./nginx.conf:/etc/nginx/conf.d/default.conf
    depends_on:
      - app
```

2.1.7. Запуск и проверка

```
# Запуск всех сервисов с масштабированием (3 реплики)
docker-compose up --scale app=3 -d

# Просмотр запущенных контейнеров
docker-compose ps

# Проверка: запрос должен распределяться между репликами
curl http://localhost/
curl http://localhost/
curl http://localhost/

# Просмотр логов (видно hostname каждого запроса)
docker-compose logs app

# Остановка всех сервисов
docker-compose down
```

Пример вывода трёх последовательных запросов:

```
// Запрос 1
{"message":"Hello from stateless service!","hostname":"myapp-app-1"}

// Запрос 2
{"message":"Hello from stateless service!","hostname":"myapp-app-2"}

// Запрос 3
{"message":"Hello from stateless service!","hostname":"myapp-app-3"}
```

2.2. Пример для Node.js (Express) с Docker Compose

2.2.1. Структура проекта

text

myapp-node/

├── app.js

├── package.json

├── Dockerfile

├── docker-compose.yml

└── nginx.conf

2.2.2. Исходный код (app.js)

```
const express = require('express');
const os = require('os');

const app = express();
const port = 8000;

app.get('/health', (req, res) => {
  res.json({ status: 'ok', hostname: os.hostname() });
});

app.get('/', (req, res) => {
  res.json({
    message: 'Hello from stateless service!',
    hostname: os.hostname()
  });
});

app.get('/compute/:number', (req, res) => {
  const number = parseInt(req.params.number);
  res.json({
    input: number,
    result: number * number,
    processed_by: os.hostname()
  });
});

app.listen(port, () => {
  console.log(`App listening on port ${port}`);
});
```

2.2.3. package.json

```
{
  "name": "stateless-service",
  "version": "1.0.0",
  "main": "app.js",
  "scripts": {
    "start": "node app.js"
  },
  "dependencies": {
    "express": "^4.18.2"
  }
}
```

2.2.4. Dockerfile

```
FROM node:18-alpine

WORKDIR /app

COPY package*.json ./
RUN npm ci --only=production

COPY . .

EXPOSE 8000

CMD ["node", "app.js"]
```

2.3. Тестирование отказоустойчивости

```
# Узнать PID контейнера с одной из реплик
docker ps

# Остановить одну реплику (эмуляция отказа)
docker stop <container_id>

# Проверить, что сервис продолжает работать (другие реплики активны)
curl http://localhost/

# Nginx автоматически перестанет направлять трафик на остановленную реплику
# Статус healthcheck в docker-compose.yml автоматически перезапустит её
```

МЕТОДИЧЕСКИЕ УКАЗАНИЯ ДЛЯ ВЫПОЛНЕНИЯ ПРАКТИЧЕСКОЙ ЧАСТИ РАБОТЫ

Задачи работы:

1. **Разработка stateless-сервиса:**
 - Реализовать REST API без хранения состояния между запросами.

- Сервис должен возвращать идентификатор реплики (hostname) в каждом ответе.

- Подготовить код, Dockerfile и зависимости.

2. Контейнеризация:

- Написать Dockerfile для приложения.
- Собрать образ и проверить его работу локально.

3. Настройка оркестрации:

- Создать docker-compose.yml для запуска нескольких реплик сервиса.

- Настроить переменные окружения для конфигурации (при необходимости).

4. Настройка балансировки нагрузки:

- Создать конфигурацию Nginx для балансировки.
- Настроить проксирование на реплики сервиса.
- Запустить связку Docker Compose + Nginx.

5. Проверка масштабирования:

- Увеличить количество реплик через --scale.
- Убедиться, что запросы распределяются между репликами.
- Протестировать отказоустойчивость (остановка одной реплики).

6. Оформление отчёта:

- Зафиксировать листинги кода (app, Dockerfile, compose, nginx.conf).

- Приложить скриншоты результатов запросов.

- Сделать выводы о работе горизонтального масштабирования.

3. ВАРИАНТЫ ЗАДАНИЙ

3.1. Базовый уровень (15 вариантов)

Требования ко всем вариантам базового уровня:

- Реализовать простой REST API (3-4 эндпоинта).

- Сервис должен быть stateless (не хранить данные между запросами).
- Каждый ответ должен содержать hostname обработавшей реплики.
- Dockerfile, docker-compose.yml (3 реплики), nginx.conf.

№	Язык/Фреймворк	Функциональность сервиса
1	Python / FastAPI	Калькулятор: +, -, ×, /
2	Python / Flask	Генерация случайного числа от 1 до 100
3	Python / FastAPI	Конвертер валют (USD → EUR, RUB, CNY)
4	Python / Flask	Проверка числа на простоту
5	Python / FastAPI	Вычисление факториала
6	Python / Flask	Генерация Fibonacci-последовательности до N
7	Node.js / Express	Калькулятор: +, -, ×, /
8	Node.js / Express	Генерация UUID v4
9	Node.js / Express	Проверка email на корректность (регулярное выражение)
10	Node.js / Express	Конвертер температур (C→F, F→C)
11	Node.js / Express	Вычисление площади фигуры (круг/прямоугольник)

№	Язык/Фреймворк	Функциональность сервиса
12	Node.js / Express	Хеширование строки (SHA-256)
13	Node.js / Express	Валидация пароля (длина, спецсимволы, цифры)
14	Node.js / Express	Разница между двумя датами (в днях)
15	Node.js / Express	Склонение слова по числу (например, "1 яблоко", "2 яблока")

3.2. Средний уровень (15 вариантов)

Требования ко всем вариантам среднего уровня:

- Базовый уровень + дополнительно:
 - Добавить эндпоинт /stats, возвращающий количество обработанных запросов за сессию жизни реплики (счётчик в памяти — возможен т.к. на одной реплике).
 - Настроить healthcheck для Docker Compose.
 - Добавить эндпоинт /error с имитацией ошибки для проверки отказоустойчивости.
 - Реализовать идемпотентность для одного из эндпоинтов (POST).

№	Язык/Фреймворк	Статистика	Дополнительный эндпоинт
1	Python / FastAPI	Счётчик запросов к каждому эндпоинту	/error/500

№	Язык/Фреймворк	Статистика	Дополнительный эндпоинт
2	Python / Flask	Время работы реплики (uptime)	/error/timeout
3	Python / FastAPI	Счётчик уникальных IP	/error/crash
4	Python / Flask	Счётчик ошибок 4xx/5xx	/error/404
5	Python / FastAPI	Распределение типов запросов (GET/POST)	/stats/reset
6	Python / Flask	Общее время обработки запросов	/slow/3 (задержка 3 сек)
7	Node.js / Express	Счётчик запросов	/error/500
8	Node.js / Express	Uptime	/error/timeout
9	Node.js / Express	Счётчик ошибок	/stats/detailed
10	Node.js / Express	Счётчик по IP	/slow/2
11	Node.js / Express	Распределение по User-Agent	/error/503
12	Node.js / Express	Время отклика	/stress/cpu

№	Язык/Фреймворк	Статистика	Дополнительный эндпоинт
		(среднее, p50, p95)	
13	Node.js / Express	Счётчик по endpoint	/slow/5 (имитация долгой операции)
14	Node.js / Express	Свободная память (при запуске реплики)	/error/random
15	Node.js / Express	Количество активных соединений	/health/detailed

3.3. Продвинутый уровень (15 вариантов)

Требования ко всем вариантам продвинутого уровня:

- Средний уровень + дополнительно:
 - Поддержка нескольких алгоритмов балансировки (round-robin, least connections, ip_hash) - переключаемых через переменную окружения Nginx.
 - Развернуть Redis как отдельный контейнер для хранения счётчика статистики (состояние вынесено из реплик).
 - Продемонстрировать, что после остановки реплики счётчик не сбрасывается (хранится в Redis).
 - Добавить интеграцию с Prometheus (эндпоинт /metrics в формате Prometheus).

	Язык/Фреймворк	Вынесенное состояние (через Redis)	Особенность масштабирования
--	----------------	---------------------------------------	--------------------------------

	Язык/Фреймворк	Вынесенное состояние (через Redis)	Особенность масштабирования
	Python / FastAPI	Глобальный счётчик запросов	least_conn + Redis
	Python / Flask	Время последнего запроса	ip_hash + Redis
	Python / FastAPI	Список последних 10 запросов (буфер)	round-robin + Redis
	Python / Flask	Rate limiting (ограничение по IP)	least_conn + скрипт логики
	Python / FastAPI	Хранение состояния сессии (JWT не подходит)	sticky sessions через ip_hash
	Python / Flask	Простая очередь задач (с помощью Redis Queue)	round-robin + Redis Queue
	Node.js / Express	Глобальный счётчик	least_conn + Redis
	Node.js / Express	Сессионная корзина (shopping cart)	ip_hash + Redis
	Node.js / Express	Примитивный leader election (Distributed Lock)	round-robin + Redis Lock
0	Node.js / Express	Асинхронная фоновая задача через Bull	least_conn + Redis (Bull)
1	Node.js / Express	Кэширование результатов (GET /compute)	round-robin + Redis Cache
2	Node.js / Express	Rate limiting с глобальным счётчиком	least_conn + Redis

	Язык/Фреймворк	Вынесенное состояние (через Redis)	Особенность масштабирования
3	Node.js / Express	Потоковые метрики (WebSocket к Redis)	ip_hash + Redis Pub/Sub
4	Node.js / Express	Таймаут сессии бездействия	round-robin + Redis TTL
5	Node.js / Express	Сравнение stateful (сессия) vs stateless (JWT)	least_conn + Redis

4. КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Что такое stateless-сервис? В чём его отличие от stateful?
2. Почему stateless-сервисы легче масштабировать горизонтально?
3. В чём разница между вертикальным и горизонтальным масштабированием? Приведите примеры.
4. Какую роль выполняет Nginx в описанной архитектуре?
5. Назовите четыре алгоритма балансировки нагрузки в Nginx.
6. Как проверить, что запросы действительно распределяются между разными репликами?
7. Что такое Docker и Docker Compose? В чём их различие?
8. Почему в Dockerfile используется EXPOSE 8000, а в docker-compose отдельно указывается ports?
9. Какие данные нельзя хранить внутри контейнера stateless-приложения?
10. Как в Docker Compose увеличить количество реплик сервиса?
11. Как эмулировать отказ реплики и проверить отказоустойчивость системы?
12. Почему после остановки одной реплики Nginx продолжает отправлять запросы на живые?

13. Что входит в понятие «Healthcheck» в контексте балансировщика?

14. Что такое idempotency (идемпотентность) и зачем она нужна в распределённых системах?

15. Какую роль играет Redis для вынесения состояния (state) из stateless-реплик?

5. ТРЕБОВАНИЯ К ОТЧЕТУ

Отчёт должен быть оформлен в виде Markdown-документа и содержать:

1. **Титульный лист** (ФИО, группа, вариант).
2. **Цель работы.**
3. **Теоретическая часть:**
 - Понятие stateless-сервиса.
 - Горизонтальное масштабирование.
 - Docker, Docker Compose, Nginx как балансировщик.
4. **Практическая часть:**
 - Листинг кода приложения (app.py / app.js).
 - Листинг Dockerfile.
 - Листинг docker-compose.yml.
 - Листинг nginx.conf.
 - Инструкция по запуску (команды).
5. **Результаты выполнения:**
 - Скриншот запуска docker-compose up --scale app=3.
 - Скриншот docker-compose ps.
 - Три последовательных запроса curl http://localhost/ с разными hostname.
 - Скриншот логов docker-compose logs app.
 - Скриншот теста отказоустойчивости (остановка одной реплики).
6. **Анализ ошибок** (если возникали).

7. **Ответы на контрольные вопросы (3-5 по выбору преподавателя).**

8. **Вывод** (что освоено, как работает горизонтальное масштабирование).

6. КРИТЕРИИ ОЦЕНИВАНИЯ

Шкала оценивания по уровням сложности

Критерий	Базовый уровень (макс. 10)	Средний уровень (макс. 14)	Продвинутый уровень (макс. 18)
Сервис работает (stateless, возвращает hostname)	3	2	2
Dockerfile корректен, образ собирается	2	2	2
docker-compose.yml с 3 репликами работает	2	2	2
Nginx балансирует запросы к репликам	3	2	2
Healthcheck настроен	—	2	2
Альтернативные алгоритмы балансировки	—	2	2
Вынесение состояния в Redis	—	—	3

Критерий	Базовый уровень (макс. 10)	Средний уровень (макс. 14)	Продвинутый уровень (макс. 18)
Prometheus-метрики (/metrics)	—	—	2
Качество отчёта и ответы на вопросы	—	2	1
Максимальный балл	10	14	18

Шкала перевода баллов в оценку

Оценка	Базовый уровень	Средний уровень	Продвинутый уровень
Отлично (5)	9-10	13-14	16-18
Хорошо (4)	7-8	10-12	12-15
Удовлетворительно (3)	5-6	7-9	8-11
Неудовлетворительно (2)	менее 5	менее 7	менее 8

Штрафы

Нарушение	Штраф
Отсутствие скриншотов успешного выполнения	-1
Отсутствие листингов кода	-1

Нарушение	Штраф
В ответе на запрос не возвращается hostname	-2
Реплик менее 3 в docker-compose	-2
Nginx не настроен / не проксирует	-3
Работа не запускается	Работа не засчитывается

7. ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ

Программа	Версия	Назначение	Доступность в РФ
Docker Desktop / Docker Engine	последняя	Контейнеризация	+ (бесплатно для образования)
Docker Compose	2.x	Оркестрация многоконтейнерных приложений	+ (входит в Docker)
Nginx (официальный образ)	alpine	Балансировка нагрузки	+ (Open Source)
Python + FastAPI/Flask	3.10 +	Разработка сервиса	+
Node.js + Express	18+	Разработка сервиса	+

Программа	Версия	Назначение	Доступность в РФ
curl	любая	Тестирование API	+
wrk / ab (Apache Bench)	последняя	Нагрузочное тестирование	+(Open Source)
Visual Studio Code	последняя	Редактор кода	+

Лабораторная работа №3

Анализ CAP-теоремы на практике

Цель работы: Экспериментально проверить ограничения CAP-теоремы на примере распределённых систем (Apache Cassandra и PostgreSQL) и научиться обоснованно выбирать компромиссы между согласованностью (Consistency), доступностью (Availability) и устойчивостью к сетевым разделам (Partition Tolerance).

Формируемые компетенции: ПК-5

1. ТЕОРЕТИЧЕСКОЕ ОБОСНОВАНИЕ

1.1. Теорема CAP: формулировка и история

Теорема CAP (также известная как теорема Брюера) - фундаментальное ограничение распределённых систем, сформулированное Эриком Брюером в 2000 году и доказанное Сетом Гилбертом и Нэнси Линч в 2002 году.

Формулировка: В распределённой системе с репликацией данных невозможно одновременно гарантировать все три свойства:

Свойство	Расшифровка	Описание
C Consistency (Согласованность)	Все узлы видят одни и те же данные одновременно	После успешной записи любой последующий чтение вернёт записанное значение
A Availability (Доступность)	Каждый запрос получает ответ (успешный или неуспешный)	Система остаётся работоспособной, даже если часть узлов вышла из строя
P Partition Tolerance (Устойчивость к разделам)	Система продолжает работать при потере связи между узлами	Сетевой раздел не приводит к полному отказу системы

Ключевой вывод: При возникновении сетевого раздела (P) система вынуждена выбирать между C (согласованностью) и A (доступностью). После устранения раздела система должна восстановить согласованность.

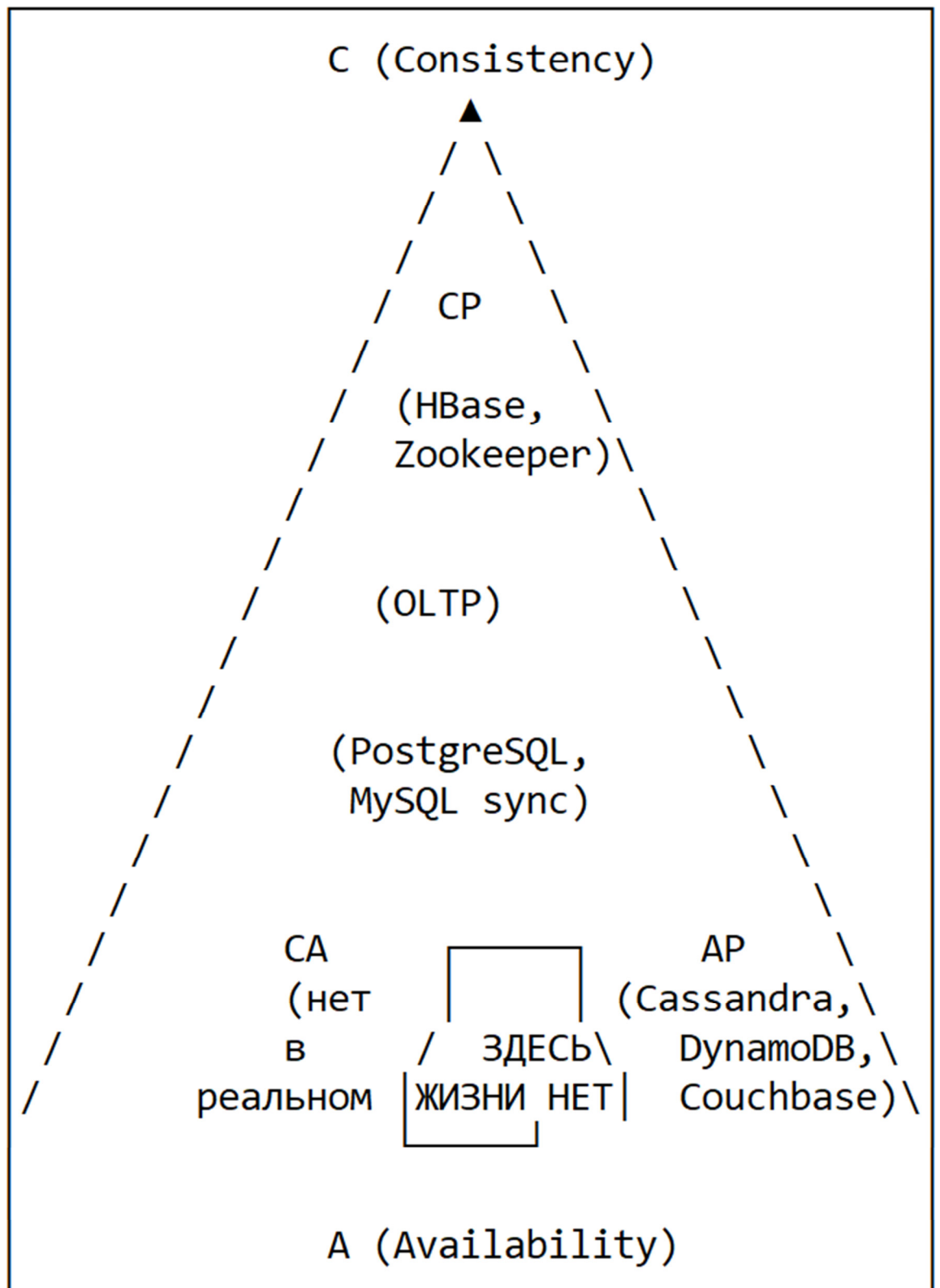


Рисунок 1 — CAP-треугольник (системы могут быть только CP или AP)

1.2. Типы систем по CAP

Тип	Описание	Примеры	Компромисс
CP (Consistency + Partition Tolerance)	Приоритет согласованности над доступностью. При сетевом разделе система может стать недоступной, чтобы не допустить рассинхронизации данных	HBase, Zookeeper, etcd, PostgreSQL (синхронная репликация), MongoDB (с настройкой majority)	Жертвует доступностью во имя целостности
AP (Availability + Partition Tolerance)	Приоритет доступности над согласованностью. При сетевом разделе система продолжает работу, но допускает временную рассинхронизацию данных (eventual consistency)	Cassandra, DynamoDB (Amazon), Couchbase, Riak, Voldemort	Жертвует строгой согласованностью во имя доступности
CA (Consistency + Availability)	Теоретически возможна только в отсутствие сетевых разделов, что в	(Не существует в распределённом виде)	Невозможна в распределённой системе

Тип	Описание	Примеры	Компромисс
	распределённых системах недостижимо		

1.3. Модели согласованности

Строгая согласованность (Linearizability):

- После успешной записи все последующие чтения (с любого узла) возвращают это значение.
- Требуется синхронная репликация (запись подтверждается только после записи на все реплики).

Eventual consistency (конечная согласованность):

- Система гарантирует, что при отсутствии новых записей все реплики в конечном итоге сойдутся к одному значению.
- Интервал рассогласования (inconsistency window) зависит от задержек сети и настроек репликации.

Другие модели согласованности (градации):

Модель	Описание
Strong consistency	Строгая согласованность (linearizable)
Sequential consistency	Порядок операций сохраняется, но время не важно
Causal consistency	Причинно-следственная согласованность
Read-your-writes	Читаешь свои записи
Monotonic reads	Монотонные чтения (значения не

Модель	Описание
	«откатываются»)
Monotonic writes	Монотонные записи (порядок записей гарантирован)
Eventual consistency	Конечная согласованность (самая «слабая»)

1.4. ACID vs BASE

Характеристика	ACID (SQL)	BASE (NoSQL)
Расшифровка	Atomicity, Consistency, Isolation, Durability	Basically Available, Soft state, Eventual consistency
Тип по CAP	CP (приоритет согласованию)	AP (приоритет доступности)
Согласованность	Строгая (в момент коммита транзакции)	Конечная (eventual)
Доступность	Может снижаться при сетевых проблемах	Высокая (система отвечает всегда)
Транзакции	Поддерживаются (ACID)	Ограниченно (lightweight transactions, batch)
Примеры задач	Финансовые системы, заказы, инвентаризация	Логи, сессии, каталоги, социальные сети

1.5. Механизмы консистентности в распределённых БД

Quorum (кворум):

Настройка $W + R > N$, где:

1. N - количество реплик
2. W - количество реплик, подтверждающих запись
3. R - количество реплик, опрашиваемых при чтении

Соотношени е	Гарантии
$W + R > N$	Гарантируется, что чтение получит как минимум одну актуальную реплику
$W = N$	Максимальная консистентность (минимальная доступность)
$W = 1$	Максимальная доступность (минимальная консистентность)

Hinted handoff: Если реплика недоступна при записи, координатор сохраняет запись локально и отправляет её, когда реплика возвращается.

Read repair: При чтении обнаруживаются расхождения между репликами и исправляются.

Anti-entropy (Merkle-деревья): Фоновый процесс сравнения реплик и синхронизации расходящихся данных.

1.6. Эмуляция сетевого раздела с помощью Docker

Для эмуляции сетевого раздела между узлами можно использовать:

```

# Получение ID контейнера
docker ps

# Отключение сетевого интерфейса (разрыв связи)
docker exec -it <container_id> ip link set eth0 down

# Подключение обратно
docker exec -it <container_id> ip link set eth0 up

# Более сложные сценарии (задержка, потеря пакетов) через tc (traffic control)
docker exec -it <container_id> tc qdisc add dev eth0 root netem delay 1000ms
docker exec -it <container_id> tc qdisc add dev eth0 root netem loss 50%

```

2. ПРИМЕРЫ ПРАКТИЧЕСКОГО ВЫПОЛНЕНИЯ

2.1. Пример для CP-системы (PostgreSQL с синхронной репликацией)

2.1.1. Структура проекта

text

postgres-cp/

```

├── docker-compose.yml    # Мастер + 2 реплики
├── init.sql              # Инициализация таблиц
├── test_queries.sql      # Тестовые запросы
└── simulate_network.sh   # Скрипт эмуляции раздела

```

2.1.2. Docker Compose для PostgreSQL CP-кластера

```

version: '3.8'

services:
  master:
    image: postgres:15
    container_name: pg-master
    environment:
      POSTGRES_USER: testuser
      POSTGRES_PASSWORD: testpass
      POSTGRES_DB: testdb
    ports:
      - "5432:5432"
    volumes:
      - ./init.sql:/docker-entrypoint-initdb.d/init.sql
    command: |
      postgres -c wal_level=replica
               -c max_wal_senders=10
               -c max_replication_slots=10

```

```

        -c hot_standby=on
        -c synchronous_commit=on
        -c synchronous_standby_names='replica1,replica2'

replica1:
  image: postgres:15
  container_name: pg-replica1
  environment:
    POSTGRES_USER: testuser
    POSTGRES_PASSWORD: testpass
    POSTGRES_DB: testdb
  command: |
    bash -c "
      until pg_basebackup -h master -U testuser -D /var/lib/postgresql/data -R -
P; do
        echo 'Waiting for master...'
        sleep 2
      done
      postgres -c hot_standby=on
               -c primary_conninfo='host=master user=testuser password=testpass'
    "
  depends_on:
    - master

replica2:
  image: postgres:15
  container_name: pg-replica2
  environment:
    POSTGRES_USER: testuser
    POSTGRES_PASSWORD: testpass
    POSTGRES_DB: testdb
  command: |
    bash -c "
      until pg_basebackup -h master -U testuser -D /var/lib/postgresql/data -R -
P; do
        echo 'Waiting for master...'
        sleep 2
      done
      postgres -c hot_standby=on
               -c primary_conninfo='host=master user=testuser password=testpass'
    "
  depends_on:
    - master

```

2.1.3. Тестовый сценарий для CP-системы

```
-- test_queries.sql

-- Создание тестовой таблицы
CREATE TABLE IF NOT EXISTS test_data (
    id SERIAL PRIMARY KEY,
    value TEXT NOT NULL,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

-- Запись данных
INSERT INTO test_data (value) VALUES ('Test record 1');
INSERT INTO test_data (value) VALUES ('Test record 2');

-- Чтение данных (проверка консистентности)
SELECT * FROM test_data ORDER BY id;
```

2.1.4. Эмуляция сетевого раздела для CP-системы

```
#!/bin/bash
# simulate_network_cp.sh

echo "=== CP-система (PostgreSQL с синхронной репликацией) ==="

# Запись данных (должна работать)
echo "Запись данных..."
docker exec pg-master psql -U testuser -d testdb -c "INSERT INTO test_data
(value) VALUES ('Before partition');"

# Проверка чтения реплики 1
echo "Чтение с replica1 ДО раздела..."
docker exec pg-replica1 psql -U testuser -d testdb -c "SELECT * FROM test_data
ORDER BY id;"

# Эмуляция сетевого раздела (отключение replica1)
echo "Эмуляция сетевого раздела: отключение replica1..."
docker exec pg-replica1 ip link set eth0 down

# Попытка записи (должна заблокироваться, т.к. replica1 не подтверждает)
echo "Попытка записи во время раздела..."
timeout 5 docker exec pg-master psql -U testuser -d testdb -c "INSERT INTO
test_data (value) VALUES ('During partition');"
if [ $? -eq 124 ]; then
    echo "ЗАПИСЬ ЗАБЛОКИРОВАНА (ожидаемое поведение CP-системы)"
else
    echo "ЗАПИСЬ УДАЛАСЬ (нарушение CP)"
fi

# Возврат replica1 в сеть
echo "Восстановление replica1..."
```

```
docker exec pg-replica1 ip link set eth0 up
sleep 5

echo "Чтение с replica1 ПОСЛЕ раздела (данные должны синхронизироваться)..."
docker exec pg-replica1 psql -U testuser -d testdb -c "SELECT * FROM test_data
ORDER BY id;"

echo "Готово!"
```

2.2. Пример для AP-системы (Apache Cassandra)

2.2.1. Docker Compose для Cassandra AP-кластера

```
version: '3.8'

services:
  cassandra-seed:
    image: cassandra:4.1
    container_name: cassandra-seed
    environment:
      - CASSANDRA_CLUSTER_NAME=CAPCluster
      - CASSANDRA_DC=DC1
      - CASSANDRA_RACK=RACK1
      - CASSANDRA_ENDPOINT_SNITCH=GossipingPropertyFileSnitch
      - CASSANDRA_NUM_TOKENS=256
    ports:
      - "9042:9042"
    healthcheck:
      test: ["CMD", "cqlsh", "-e", "describe keyspaces"]
      interval: 15s
      timeout: 10s
      retries: 10

  cassandra-node1:
    image: cassandra:4.1
    container_name: cassandra-node1
    environment:
      - CASSANDRA_SEEDS=cassandra-seed
      - CASSANDRA_CLUSTER_NAME=CAPCluster
      - CASSANDRA_DC=DC1
      - CASSANDRA_RACK=RACK1
      - CASSANDRA_ENDPOINT_SNITCH=GossipingPropertyFileSnitch
    depends_on:
      cassandra-seed:
        condition: service_healthy

  cassandra-node2:
    image: cassandra:4.1
    container_name: cassandra-node2
```

```

environment:
  - CASSANDRA_SEEDS=cassandra-seed
  - CASSANDRA_CLUSTER_NAME=CAPCluster
  - CASSANDRA_DC=DC1
  - CASSANDRA_RACK=RACK1
  - CASSANDRA_ENDPOINT_SNITCH=GossipingPropertyFileSnitch
depends_on:
  cassandra-seed:
    condition: service_healthy

```

2.2.2. Настройка Keyspace с разными уровнями согласованности

```

-- cql_commands.cql

-- AP-ориентированная настройка (высокая доступность, eventual consistency)
CREATE KEYSPACE IF NOT EXISTS cap_demo_ap
WITH REPLICATION = {
  'class': 'SimpleStrategy',
  'replication_factor': 3
};

-- CP-ориентированная настройка (строгая согласованность через настройку quorum)
CREATE KEYSPACE IF NOT EXISTS cap_demo_cp
WITH REPLICATION = {
  'class': 'SimpleStrategy',
  'replication_factor': 3
};

-- Использование AP keyspace
USE cap_demo_ap;

CREATE TABLE users (
  user_id UUID PRIMARY KEY,
  name TEXT,
  email TEXT
);

-- Настройка консистенции для AP (ONE = высокая доступность)
CONSISTENCY ONE;
INSERT INTO users (user_id, name, email) VALUES (uuid(), 'Alice',
'alice@example.com');

-- Настройка консистенции для CP (QUORUM = баланс)
CONSISTENCY QUORUM;
SELECT * FROM users;

```

2.2.3. Эмуляция сетевого раздела для AP-системы

```
#!/bin/bash
# simulate_network_ap.sh

echo "=== AP-система (Apache Cassandra) ==="

# Вход в контейнер seed
SEED_CONTAINER="cassandra-seed"

# Проверка статуса кластера
echo "Статус кластера ДО раздела..."
docker exec $SEED_CONTAINER nodetool status

# Запись данных
echo "Запись данных с уровнем консистенции ONE..."
docker exec $SEED_CONTAINER cqlsh -e "CONSISTENCY ONE; INSERT INTO
cap_demo_ap.users (user_id, name, email) VALUES (uuid(), 'Bob',
'bob@example.com');"

# Эмуляция отказа одного узла (отключение node1)
echo "Эмуляция раздела: отключение cassandra-node1..."
docker stop cassandra-node1

# Проверка, что запись всё ещё возможна (AP-система должна быть доступна)
echo "Попытка записи во время отказа узла..."
docker exec $SEED_CONTAINER cqlsh -e "CONSISTENCY ONE; INSERT INTO
cap_demo_ap.users (user_id, name, email) VALUES (uuid(), 'Charlie',
'charlie@example.com');"
if [ $? -eq 0 ]; then
    echo "ЗАПИСЬ УДАЛАСЬ (AP-система остаётся доступной)"
fi

# Чтение (может не увидеть последние записи – eventual consistency)
echo "Чтение во время отказа узла (может не увидеть Charlie)..."
docker exec $SEED_CONTAINER cqlsh -e "CONSISTENCY ONE; SELECT * FROM
cap_demo_ap.users;"

# Восстановление узла
echo "Восстановление cassandra-node1..."
docker start cassandra-node1
sleep 10

# Чтение после восстановления (должны увидеть все записи)
echo "Чтение ПОСЛЕ восстановления..."
docker exec $SEED_CONTAINER cqlsh -e "CONSISTENCY ONE; SELECT * FROM
cap_demo_ap.users;"

echo "Готово!"
```

МЕТОДИЧЕСКИЕ УКАЗАНИЯ ДЛЯ ВЫПОЛНЕНИЯ ПРАКТИЧЕСКОЙ ЧАСТИ РАБОТЫ

Задачи работы:

- 1. Развёртывание CP-системы (PostgreSQL с синхронной репликацией):**
 - Запустить кластер PostgreSQL с одним мастером и двумя репликами.
 - Настроить синхронную репликацию (`synchronous_commit = on`).
 - Убедиться, что запись требует подтверждения от реплик.
- 2. Развёртывание AP-системы (Apache Cassandra):**
 - Запустить кластер Cassandra из 3 узлов.
 - Создать keyspace с replication factor = 3.
 - Проверить возможность записи при уровне консистенции ONE.
- 3. Эмуляция сетевого раздела:**
 - Отключить один из узлов CP-системы и наблюдать блокировку записи.
 - Отключить один из узлов AP-системы и наблюдать сохранение доступности.
- 4. Анализ поведения:**
 - Сравнить время записи в CP и AP системах.
 - Зафиксировать рассинхронизацию данных (eventual consistency) в AP-системе.
 - Сделать вывод о применимости каждой системы для разных классов задач.
- 5. Оформление отчёта:**
 - Представить листинги конфигураций и команд.
 - Сравнительная таблица поведения CP и AP систем.
 - Обоснование выбора типа системы для конкретной предметной области.

3. ВАРИАНТЫ ЗАДАНИЙ

3.1. Базовый уровень (15 вариантов)

Требования ко всем вариантам базового уровня:

- Развернуть PostgreSQL кластер (мастер + 1 реплика) с синхронной репликацией.
- Развернуть Cassandra кластер (3 узла) с replication factor = 3.
- Провести эксперимент по эмуляции сетевого раздела (отключение узла).
- Зафиксировать результаты в таблице.

	Особенность СР-системы	Особенность АР-системы	Дополнительное задание
1	synchronous_commit = on	CONSISTENCY ONE	Сравнить время записи
2	synchronous_commit = off	CONSISTENCY QUORUM	Сравнить время записи
3	1 реплика	replication factor = 2	Измерить RTO после failover
4	2 реплики	replication factor = 3	Измерить RTO после failover
5	synchronous_standby_names = '1'	CONSISTENCY LOCAL_QUORUM	Оценить процент потерь данных
6	Приоритет согласованности	Приоритет доступности	Нарисовать CAP-треугольник для системы
7	Тест на отказ мастера	Тест на отказ одного узла	Время восстановления (MTTR)
8	Тест на потерю кворума	Тест на split-brain	Анализ сценариев

	Особенность СР-системы	Особенность АР-системы	Дополнительное задание
9	PostgreSQL + pgpool-II	Cassandra + tunable consistency	Сравнить сложность настройки
10	Patroni + etcd для HA	Cassandra + Gossip	Сравнение механизмов обнаружения
11	Асинхронная репликация как baseline	CONSISTENCY EACH_QUORUM	Сравнить три уровня консистенции
12	Синхронная vs асинхронная	QUORUM vs ALL	Скорость записи vs надёжность
13	PostgreSQL + repmgr	Cassandra + rack awareness	Влияние топологии на CAP
14	Логическая репликация	CONSISTENCY LOCAL_ONE + QUORUM	Сравнить модели консистенции
15	Только master (без реплик)	replication factor = 1	Baseline (нет отказоустойчивости)

3.2. Средний уровень (15 вариантов)

Требования ко всем вариантам среднего уровня:

- Всё из базового уровня + дополнительно:
 - Подготовить автоматизированный скрипт эмуляции сетевого раздела.
 - Измерить время записи в различных режимах консистенции.
 - Создать график зависимости доступности от согласованности.

№	Эксперимент 1	Эксперимент 2	Метрика для сравнения
---	---------------	---------------	-----------------------

№	Эксперимент 1	Эксперимент 2	Метрика для сравнения
1	Задержка сети 100ms	Задержка сети 500ms	p99 latency записи
2	Потеря пакетов 10%	Потеря пакетов 30%	Успешность операций
3	Отказ 1 узла	Отказ 2 узлов	Доступность записи/чтения
4	Split-brain (два раздела)	Partition + восстановление	Время согласования данных
5	CONSISTENCY ONE	CONSISTENCY ALL	Скорость vs надёжность
6	CONSISTENCY LOCAL_ONE	CONSISTENCY EACH_QUORUM	Влияние топологии
7	Запись во время раздела	Чтение во время раздела	Расхождение данных
8	Постепенное отключение узлов	Постепенное включение	Восстановление согласованности
9	Синхронная vs асинхронная репликация	CONSISTENCY QUORUM	Время блокировки записи
10	PostgreSQL + Patroni	Cassandra + Gossip	Время failover
11	3 узла, потеря 1 узла	5 узлов, потеря 2 узлов	Масштабирование и CAP
12	Тест с нагрузкой (100 RPS)	Тест с нагрузкой (500 RPS)	Поведение при нагрузке
13	Сетевой раздел между DC	Сетевой раздел внутри DC	Геораспределение и CAP

№	Эксперимент 1	Эксперимент 2	Метрика для сравнения
14	Strong consistency	Eventual consistency	Сравнить оба подхода
15	ACID транзакция	Lightweight transaction (Cassandra)	Сравнить механизмы

3.3. Продвинутый уровень (15 вариантов)

Требования ко всем вариантам продвинутого уровня:

- Всё из среднего уровня + дополнительно:
 - Реализовать тестовое приложение на Python/Java, которое демонстрирует эффект eventual consistency (чтение «старого» значения).
 - Настроить разные уровни консистенции для разных типов операций в Cassandra (read/write consistency levels).
 - Подготовить отчёт с выводами о применимости CP/AP систем для трёх различных предметных областей.

	Основная БД	Альтернативная БД	Предметная область для анализа
	PostgreSQL	Cassandra	Банковская транзакция
	PostgreSQL	MongoDB	Каталог товаров
	PostgreSQL	CockroachDB	Система бронирования билетов
	PostgreSQL	YugabyteDB	Платёжный шлюз
	Cassandra	ScyllaDB	Логиование и мониторинг
	Cassandra	ClickHouse	Аналитика больших данных

	Основная БД	Альтернативная БД	Предметная область для анализа
			данных
	Cassandra	Redis Cluster	Сессии пользователей
	MongoDB	PostgreSQL	Социальная сеть (лента постов)
	MongoDB	Cassandra	Система рекомендаций
0	CockroachDB	PostgreSQL	Распределённые финансы
11	YugabyteDB	Cassandra	ERP-система
12	TiDB	MySQL Cluster	OLTP + OLAP (HTAP)
13	FoundationDB	Cassandra	Сложные транзакции
14	FaunaDB	PostgreSQL	Глобальное приложение
5	Spanner (эмуляция)	CockroachDB	Глобальный банк

4. КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Сформулируйте теорему CAP. Какие три свойства она описывает?
2. Почему СА-системы не существуют в распределённом виде?
3. В чём разница между CP- и AP-системами? Приведите по два примера для каждого типа.

4. Как настройка `synchronous_commit` в PostgreSQL влияет на место системы в CAP-треугольнике?
5. Что такое `CONSISTENCY ONE` и `CONSISTENCY QUORUM` в Cassandra?
6. Какое соотношение $W + R > N$ гарантирует в распределённой БД?
7. Что такое «eventual consistency»? Каков типичный «inconsistency window»?
8. Как эмулировать сетевой раздел с помощью Docker?
9. Что происходит с СР-системой при потере кворума реплик?
10. Что происходит с АР-системой при потере узла?
11. Какие механизмы используются для восстановления согласованности в АР-системах (hinted handoff, read repair)?
12. Выберите: для системы бронирования авиабилетов - СР или АР? Почему?
13. Выберите: для системы хранения логов посещаемости сайта - СР или АР? Почему?
14. Выберите: для системы кэширования новостной ленты - СР или АР? Почему?
15. Что такое PACELC-теорема и чем она дополняет CAP?

5. ТРЕБОВАНИЯ К ОТЧЕТУ

Отчёт должен содержать:

1. **Титульный лист** (ФИО, группа, вариант).
2. **Цель работы.**
3. **Теоретическая часть:**
 - Теорема CAP (формулировка, диаграмма).
 - ACID vs BASE.
 - СР и АР системы (примеры).
4. **Практическая часть:**
 - Листинг `docker-compose.yml` для PostgreSQL (СР).

- Листинг docker-compose.yml для Cassandra (AP).
- Скрипты эмуляции сетевого раздела.
- Команды для проверки консистенции.

5. Результаты выполнения:

- Скриншот успешного запуска кластеров.
- Таблица сравнения поведения СР и АР систем при отказе узла.
- График времени записи в разных режимах.
- Скриншоты логов и ошибок.

6. Анализ результатов:

- Объяснение, почему СР-система заблокировала запись.
- Объяснение, почему АР-система осталась доступной, но показала

старые данные.

7. Вывод о применимости:

- Для каких задач выбирать СР.
- Для каких задач выбирать АР.
- Примеры из реальной практики.

8. Ответы на контрольные вопросы (3-5 по выбору).

6. КРИТЕРИИ ОЦЕНИВАНИЯ

Шкала оценивания по уровням сложности

Критерий	Базовый (10)	Средний (14)	Продвинутый (18)
Кластер PostgreSQL развёрнут и работает	2	2	1
Кластер Cassandra развёрнут и работает	2	2	1
Эмуляция сетевого раздела выполнена	2	2	2
Зафиксировано поведение СР при	2	2	2

Критерий	Базовый (10)	Средний (14)	Продвинутый (18)
отказе			
Зафиксировано поведение АР при отказе	2	2	2
Сравнительная таблица составлена	—	2	2
Автоматизация экспериментов (скрипт)	—	2	2
Измерены и визуализированы метрики	—	—	2
Выводы о применимости СР/АР	—	—	2
Качество отчёта	—	—	2
Максимум	10	14	18

Шкала перевода баллов в оценку

Оценка	Базовый	Средний	Продвинутый
5	9-10	13-14	16-18
4	7-8	10-12	12-15
3	5-6	7-9	8-11

О ценка	Ба зовый	Ср едний	Продви нутый
2	< 5	<7	<8

Штрафы

Нарушение	Штра ф
Отсутствие скриншотов	-1
Отсутствие таблицы сравнения	-2
Не объяснено почему СР заблокировала запись	-2
Не объяснена eventual consistency в АР	-2
Работа не запускается	не засчитывается

7. ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ

Программа	Назначение	Доступность в РФ
Docker Desktop / Docker Engine	Контейнеризация	+
Docker Compose	Оркестрация	+
PostgreSQL (официальный образ)	СР-система	+
Apache	АР-система	+

Программа	Назначение	Доступность в РФ
Cassandra (официальный образ)		
cqlsh (входит в образ Cassandra)	Управление Cassandra	+
psql (входит в образ PostgreSQL)	Управление PostgreSQL	+
Python (опционально)	Автоматизация экспериментов	+
curl / wget	Проверка доступности	+

Лабораторная работа №4

Реализация паттерна Circuit Breaker для обеспечения отказоустойчивости

Цель работы: Сформировать практические навыки реализации паттерна Circuit Breaker для предотвращения каскадных отказов в распределённых системах, настройки параметров автоматического восстановления и организации graceful degradation (fallback-логики).

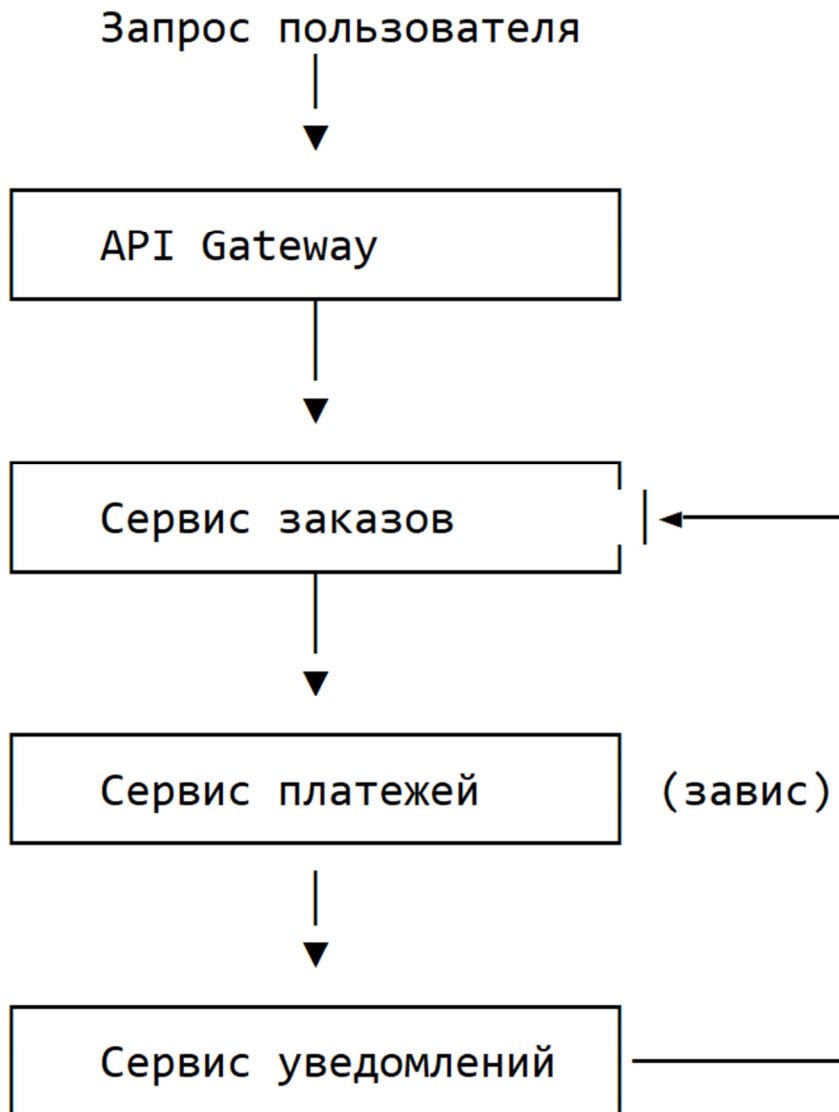
Формируемые компетенции: ПК-5

1. ТЕОРЕТИЧЕСКОЕ ОБОСНОВАНИЕ

1.1. Проблема каскадных отказов в распределённых системах

В микросервисной архитектуре один сервис часто вызывает другой. При недоступности зависимого сервиса возникают проблемы:

Каскадный отказ (Cascade Failure):



Механизм каскадного отказа:

1. Сервис платежей начинает работать медленно или зависает
2. Сервис заказов удерживает соединения, ожидая ответа
3. Пул соединений сервиса заказов исчерпывается
4. Новые запросы к сервису заказов ждать
5. Сервис заказов перестаёт отвечать
6. API Gateway начинает ждать ответ от сервиса заказов
7. Вся система «падает»

Последствия без Circuit Breaker:

- Исчерпание пулов соединений (connection pool exhaustion)
- Утечка памяти из-за накопления ожидающих запросов
- Увеличение времени отклика (response time)

- Полная недоступность системы

1.2. Паттерн Circuit Breaker

Circuit Breaker (автоматический выключатель) - паттерн, предотвращающий повторные вызовы заведомо неработающего сервиса, позволяя системе gracefully деградировать и восстанавливаться.

Три состояния Circuit Breaker:

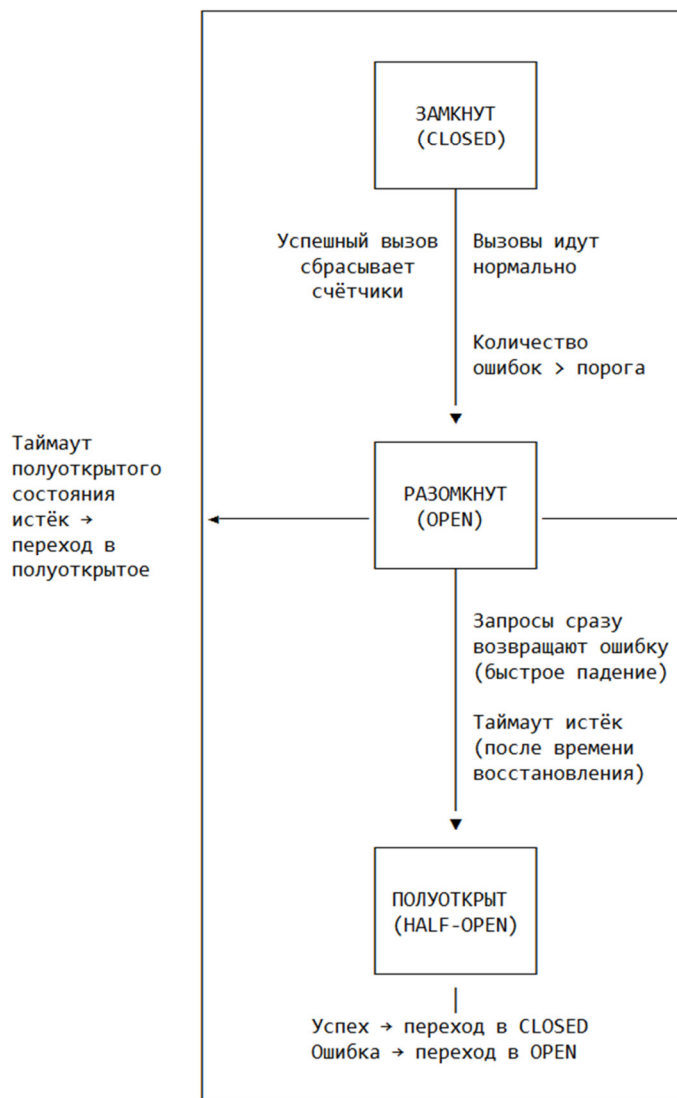


Таблица состояний:

Состояние	Описание	Поведение
CLOSED (замкнуто)	Нормальная работа	Все запросы проходят, счётчики ошибок

Состояние	Описание	Поведение
		накапливаются
OPEN (разомкнут)	Сервис считается недоступным	Запросы мгновенно отклоняются (fail fast)
HALF-OPEN (полуоткрыт)	Проверка восстановления	Пропускается ограниченное число запросов для проверки

1.3. Ключевые параметры Circuit Breaker

Параметр	Описание	Типичное значение
failureThreshold	Количество ошибок для перехода в OPEN	5-10 ошибок
timeout	Таймаут ожидания ответа от сервиса	3-5 секунд
waitDurationInOpenState	Время в состоянии OPEN перед переходом в HALF-OPEN	30-60 секунд
permittedNumberOfCallsInHalfOpenState	Количество тестовых вызовов в HALF-OPEN	3-5 вызовов
slidingWindowSize	Размер окна для подсчёта	10-100 запросов

Параметр	Описание	Типичное значение
	ошибок	
slowCallThreshold	Порог «медленного вызова»	5 секунд

1.4. Стратегии Fallback (graceful degradation)

Fallback - альтернативное действие при срабатывании Circuit Breaker.

Стратегия	Описание	Пример
Static fallback	Возврат статического значения	return {"status": "service unavailable, try later"}
Cached fallback	Возврат кэшированных данных	Последний известный курс валют из Redis
Default fallback	Возврат значения по умолчанию	return [] (пустой список)
Stubbed fallback	Заглушка с ограниченной логикой	return mock_data
Alternative service	Вызов альтернативного сервиса	Резервный платёжный шлюз

1.5. Инструменты реализации Circuit Breaker

	Платформы	Особенности

	Платформ а	Особенности
Resilience4j	Java	Лёгкий, функциональный, интеграция с Spring Boot
Hystrix (устарел)	Java	Netflix Hystrix (режим обслуживания)
PyBreaker	Python	Простая реализация на Python
circuitbreaker (PyPI)	Python	Гибкая настройка, декораторы
Polly	.NET	Очень богатая библиотека
go-resiliency	Go	Простые паттерны на Go
Istio	Service Mesh	Circuit Breaker на уровне инфраструктуры

1.6. Пример реализации на Python (resilience4j-подобный)

```
import time
import threading
from enum import Enum
from datetime import datetime, timedelta

class CircuitBreakerState(Enum):
    CLOSED = "CLOSED"
    OPEN = "OPEN"
    HALF_OPEN = "HALF_OPEN"

class CircuitBreaker:
    def __init__(self, failure_threshold=5, timeout=60, half_open_calls=3):
        self.failure_threshold = failure_threshold
        self.timeout = timeout # время в OPEN состоянии (сек)
        self.half_open_calls = half_open_calls
        self.state = CircuitBreakerState.CLOSED
```

```

self.failure_count = 0
self.last_failure_time = None
self.half_open_success_count = 0
self.lock = threading.Lock()

def call(self, func, fallback=None, *args, **kwargs):
    with self.lock:
        if self.state == CircuitBreakerState.OPEN:
            # Проверка, не пора ли перейти в HALF_OPEN
            if datetime.now() - self.last_failure_time >
timedelta(seconds=self.timeout):
                self.state = CircuitBreakerState.HALF_OPEN
                self.half_open_success_count = 0
                print(f"[CircuitBreaker] Переход в HALF_OPEN")
            else:
                print(f"[CircuitBreaker] Запрос отклонён (состояние OPEN)")
                return fallback() if fallback else None

        try:
            result = func(*args, **kwargs)
            self._on_success()
            return result
        except Exception as e:
            self._on_failure()
            print(f"[CircuitBreaker] Ошибка: {e}")
            return fallback() if fallback else None

def _on_success(self):
    with self.lock:
        if self.state == CircuitBreakerState.HALF_OPEN:
            self.half_open_success_count += 1
            if self.half_open_success_count >= self.half_open_calls:
                self._reset()
                print(f"[CircuitBreaker] Переход в CLOSED (успешное
восстановление)")
            elif self.state == CircuitBreakerState.CLOSED:
                self.failure_count = 0

def _on_failure(self):
    with self.lock:
        if self.state == CircuitBreakerState.HALF_OPEN:
            self.state = CircuitBreakerState.OPEN
            self.last_failure_time = datetime.now()
            print(f"[CircuitBreaker] Переход в OPEN (ошибка в HALF_OPEN)")
        elif self.state == CircuitBreakerState.CLOSED:
            self.failure_count += 1
            if self.failure_count >= self.failure_threshold:
                self.state = CircuitBreakerState.OPEN
                self.last_failure_time = datetime.now()
                print(f"[CircuitBreaker] Переход в OPEN (достигнут порог
ошибок)")

```

```
def _reset(self):
    self.state = CircuitBreakerState.CLOSED
    self.failure_count = 0
    self.last_failure_time = None
    self.half_open_success_count = 0
```

2. ПРИМЕРЫ ПРАКТИЧЕСКОГО ВЫПОЛНЕНИЯ

2.1. Пример на Python (FastAPI + PyBreaker)

2.1.1. Структура проекта

text

circuit-breaker-demo/

- |— app.py # Основное приложение с Circuit Breaker
- |— unhealthy_service.py # Имитация нестабильного сервиса
- |— requirements.txt # Зависимости
- |— docker-compose.yml # Запуск двух сервисов
- |— test_requests.py # Скрипт тестирования

2.1.2. Зависимости (requirements.txt)

text

```
fastapi==0.104.1
uvicorn[standard]==0.24.0
pybreaker==0.6.0
httpx==0.25.0
```

2.1.3. Нестабильный сервис (unhealthy_service.py)

```
import random
import time
from fastapi import FastAPI

app = FastAPI(title="Unstable Service Example")

# Счётчики для эмуляции разных режимов сбоев
call_count = 0

@app.get("/health")
async def health():
```

```

        return {"status": "ok"}

@app.get("/unstable")
async def unstable():
    global call_count
    call_count += 1

    # Эмуляция разных типов сбоев
    error_type = random.choice(["ok", "timeout", "error", "slow", "error"])

    # 20% ошибок
    if error_type == "error":
        raise Exception("Internal server error")

    # 20% медленных ответов
    if error_type == "slow":
        time.sleep(5)
        return {"status": "slow", "count": call_count}

    # 10% таймаутов (без ответа)
    if error_type == "timeout":
        time.sleep(15) # Дольше, чем таймаут клиента
        return {"status": "timeout", "count": call_count}

    # 50% успешных запросов
    return {"status": "ok", "count": call_count}

@app.post("/reset")
async def reset():
    global call_count
    call_count = 0
    return {"status": "reset"}

```

2.1.4. Основной сервис с Circuit Breaker (app.py)

```

import pybreaker
import httpx
from fastapi import FastAPI, HTTPException
import asyncio

# Настройка Circuit Breaker
breaker = pybreaker.CircuitBreaker(
    fail_max=3,                # 3 ошибки → OPEN
    reset_timeout=30,          # 30 секунд в OPEN
    half_open_timeout=5,       # 5 секунд на проверку в HALF_OPEN
    exclude=[],
    name="payment_service"
)

app = FastAPI(title="Service with Circuit Breaker")

```

```

# URL нестабильного сервиса
UNSTABLE_SERVICE_URL = "http://unstable:8001"

@breaker
async def call_unstable_service():
    """Вызов нестабильного сервиса с защитой Circuit Breaker"""
    async with httpx.AsyncClient(timeout=5.0) as client:
        response = await client.get(f"{UNSTABLE_SERVICE_URL}/unstable")
        response.raise_for_status()
        return response.json()

async def fallback_response():
    """Fallback при срабатывании Circuit Breaker"""
    return {
        "status": "fallback",
        "message": "Сервис временно недоступен, попробуйте позже",
        "data": {"cached": True, "value": "default response"}
    }

@app.get("/call-unstable")
async def call_with_circuit_breaker():
    """Эндпоинт с использованием Circuit Breaker и fallback"""
    try:
        result = await call_unstable_service()
        return {"status": "success", "data": result}
    except pybreaker.CircuitBreakerError:
        # Circuit Breaker разомкнут
        return await fallback_response()
    except Exception as e:
        # Другие ошибки (сеть, таймаут и т.д.)
        return await fallback_response()

@app.get("/breaker-status")
async def get_breaker_status():
    """Получение статуса Circuit Breaker"""
    status = {
        "state": str(breaker.current_state).split(".")[1],
        "fail_counter": breaker.fail_counter,
        "opened_at": str(breaker.opened_at) if breaker.opened_at else None,
    }
    return status

@app.post("/breaker-reset")
async def reset_breaker():
    """Сброс Circuit Breaker (принудительное закрытие)"""

```

```
breaker.close()
return {"status": "reset"}
```

2.1.5. Docker Compose (docker-compose.yml)

```
version: '3.8'

services:
  main:
    build: .
    ports:
      - "8000:8000"
    environment:
      - UNSTABLE_URL=http://unstable:8001
    depends_on:
      - unstable
    command: uvicorn app:app --host 0.0.0.0 --port 8000 --reload

  unstable:
    build: .
    ports:
      - "8001:8001"
    command: uvicorn unhealthy_service:app --host 0.0.0.0 --port 8001 --reload
```

2.1.6. Тестовый скрипт (test_requests.py)

```
import httpx
import time
import asyncio

BASE_URL = "http://localhost:8000"

async def test_circuit_breaker():
    async with httpx.AsyncClient() as client:
        print("=== Тестирование Circuit Breaker ===\n")

        for i in range(1, 16):
            try:
                response = await client.get(f"{BASE_URL}/call-unstable")
                data = response.json()

                if data.get("status") == "fallback":
                    print(f"[{i:2d}] FALLBACK: {data.get('message')}")
                else:
                    print(f"[{i:2d}] SUCCESS: {data.get('data')}")

            # Статус Circuit Breaker
            status = await client.get(f"{BASE_URL}/breaker-status")
            print(f"      → Состояние CB: {status.json()['state']}")
```

```

except Exception as e:
    print(f"[{i:2d}] ОШИБКА: {e}")

    time.sleep(1)

print("\n=== Завершение теста ===")

if __name__ == "__main__":
    asyncio.run(test_circuit_breaker())

```

Ожидаемый вывод:

```

[ 1] SUCCESS: {'status': 'ok', 'count': 1}
      → Состояние CB: CLOSED
[ 2] SUCCESS: {'status': 'ok', 'count': 2}
      → Состояние CB: CLOSED
[ 3] FALLBACK: Сервис временно недоступен, попробуйте позже
      → Состояние CB: OPEN
[ 4] FALLBACK: Сервис временно недоступен, попробуйте позже
      → Состояние CB: OPEN
...
Через 30 секунд (таймаут) → переход в HALF-OPEN
...
[11] SUCCESS: ...
      → Состояние CB: CLOSED

```

2.2. Пример реализации на Java (Spring Boot + Resilience4j)

2.2.1. Зависимость Maven (pom.xml)

```

<dependency>
  <groupId>io.github.resilience4j</groupId>
  <artifactId>resilience4j-spring-boot3</artifactId>
  <version>2.2.0</version>
</dependency>
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-web</artifactId>
</dependency>

```

2.2.2. Контроллер с Circuit Breaker

```

@RestController
public class DemoController {

    @Autowired
    private ExternalServiceClient externalClient;

    @GetMapping("/call")

```

```

    @CircuitBreaker(name = "externalService", fallbackMethod =
"fallbackResponse")
    public ResponseEntity<?> callExternalService() {
        return ResponseEntity.ok(externalClient.call());
    }

    public ResponseEntity<?> fallbackResponse(Exception e) {
        return ResponseEntity.status(503).body(
            Map.of("status", "fallback",
                "message", "Service is temporarily unavailable")
        );
    }
}

```

2.2.3. Настройка application.yml

```

resilience4j:
  circuitbreaker:
    instances:
      externalService:
        failureRateThreshold: 50           # Порог ошибок 50%
        slowCallRateThreshold: 100
        slowCallDurationThreshold: 5s
        permittedNumberOfCallsInHalfOpenState: 3
        slidingWindowSize: 10
        minimumNumberOfCalls: 5
        waitDurationInOpenState: 30s

```

МЕТОДИЧЕСКИЕ УКАЗАНИЯ ДЛЯ ВЫПОЛНЕНИЯ ПРАКТИЧЕСКОЙ ЧАСТИ РАБОТЫ

Задачи работы:

1. **Разработка нестабильного сервиса:**
 - Реализовать сервис с эмуляцией сбоев (ошибки, медленные ответы, таймауты).
 - Добавить эндпоинты для сброса счётчиков и изменения сценария сбоев.
2. **Реализация Circuit Breaker:**
 - Внедрить библиотеку Circuit Breaker (PyBreaker / Resilience4j).
 - Настроить параметры: порог ошибок, таймаут полуоткрытого состояния.
 - Оборачивать вызовы зависимого сервиса в Circuit Breaker.

3. **Реализация Fallback-логики:**
 - Добавить graceful degradation при срабатывании Circuit Breaker.
 - Реализовать несколько стратегий fallback.
4. **Эмуляция различных сценариев отказов:**
 - Постоянные ошибки (500)
 - Медленные ответы (превышение таймаута)
 - Временные сбои (перемежающиеся ошибки)
 - Полное восстановление сервиса
5. **Мониторинг состояний:**
 - Добавить эндпоинт для получения состояния Circuit Breaker.
 - Отслеживать переходы CLOSED → OPEN → HALF-OPEN → CLOSED.
6. **Оформление отчёта:**
 - Листинги кода и конфигураций.
 - Логи переходов состояний.
 - Демонстрация fail fast и восстановления.

3. ВАРИАНТЫ ЗАДАНИЙ

3.1. Базовый уровень (15 вариантов)

Требования ко всем вариантам базового уровня:

- Реализовать клиентский сервис и нестабильный сервис.
- Настроить Circuit Breaker с базовой конфигурацией (3 ошибки → OPEN, 30 секунд → HALF-OPEN).
- Реализовать fallback (статическое сообщение).

	Язык/Фреймворк	Тип сбоя в нестабильном сервисе	Fallback
	Python / FastAPI	Ошибка 50%	Статическое сообщение

	Язык/Фреймворк	Тип сбоя в нестабильном сервисе	Fallback
	Python / Flask	Таймаут 5 секунд	Статическое сообщение
	Python / FastAPI	Медленный ответ (3 сек)	Статическое сообщение
	Python / Flask	Чередование (ok/error)	Статическое сообщение
	Python / FastAPI	Ошибка 80%	Статическое сообщение
	Python / Flask	Таймаут + ошибка	Статическое сообщение
	Node.js / Express	Ошибка 50%	Статическое сообщение
	Node.js / Express	Таймаут 5 секунд	Статическое сообщение
	Node.js / Express	Медленный ответ (3 сек)	Статическое сообщение
0	Node.js / Express	Чередование (ok/error)	Статическое сообщение
1	Node.js / Express	Ошибка 80%	Статическое сообщение
2	Node.js / Express	Таймаут + ошибка	Статическое сообщение

	Язык/Фреймворк	Тип сбоя в нестабильном сервисе	Fallback
3	Java / Spring Boot	Ошибка 50%	Статическое сообщение
4	Java / Spring Boot	Таймаут 5 секунд	Статическое сообщение
5	Java / Spring Boot	Медленный ответ (3 сек)	Статическое сообщение

3.2. Средний уровень (15 вариантов)

Требования ко всем вариантам среднего уровня:

- Всё из базового уровня + дополнительно:
 - Добавить эндпоинт для получения состояния Circuit Breaker.
 - Реализовать fallback с использованием Redis (кэшированный ответ).
 - Настроить разные параметры для разных зависимостей.

	Fallback с кэшем (Redis)	Дополнительная зависимость	Тип мониторинга
	Кэширование курса валют	2-й нестабильный сервис	Логирование переходов
	Кэширование списка товаров	2-й нестабильный сервис	Prometheus метрики
	Кэширование профиля пользователя	База данных	Webhook уведомления
	Кэширование цен (price)	2-й нестабильный	Логирование

	Fallback с кэшем (Redis)	Дополнительная зависимость	Тип мониторинга
	cache)	сервис	переходов
	Кэширование погоды	2-й нестабильный сервис	Prometheus метрики
	Кэширование новостей	База данных	Webhook уведомления
	Кэширование сессий	2-й нестабильный сервис	Логирование переходов
	Кэширование топ-10 (leaderboard)	2-й нестабильный сервис	Prometheus метрики
	Кэширование рейтинга	База данных	Webhook уведомления
0	Кэширование stock уровня	2-й нестабильный сервис	Логирование переходов
1	Кэширование с TTL 60 сек	2-й нестабильный сервис	Prometheus метрики
2	Кэширование с обновлением фона	База данных	Webhook уведомления
3	Кэширование с версионированием	2-й нестабильный сервис	Логирование переходов
	Кэширование с tenant	2-й нестабильный	Prometheus

	Fallback с кэшем (Redis)	Дополнительная зависимость	Тип мониторинга
4	изоляция	сервис	метрики
5	Кэширование с fallback цепочкой (Redis → файл → дефолт)	База данных	Webhook уведомления

3.3. Продвинутый уровень (15 вариантов)

Требования ко всем вариантам продвинутого уровня:

- Всё из среднего уровня + дополнительно:
 - Реализовать Bulkhead изоляцию (ограничение количества одновременных вызовов).
 - Реализовать альтернативный сервис (backup service) при срабатывании Circuit Breaker.
 - Настроить интеграцию с Prometheus + Grafana для визуализации состояний СВ.
 - Провести нагрузочное тестирование с разными сценариями сбоев.

№	Service Mesh (Istio / Linkerd)	Bulkhead	Альтернативный сервис	Нагрузочное тестирование
1	Istio VirtualService	Thread pool	Резервный API (mock)	wrk / k6
2	Istio DestinationRule	Semaphore	Резервная БД	wrk / k6
3	Linkerd	Connection pool	Кэш CDN	wrk / k6

№	Service Mesh (Istio / Linkerd)	Bulkhead	Альтернативный сервис	Нагрузочное тестирование
4	Istio + Envoy	Fixed thread pool	Офлайн режим	wrk / k6
5	Istio VirtualService	BoundedElastic	Mock-сервис (JSON)	wrk / k6
6	Linkerd + Viz	Semaphore + таймаут	Статический fallback	wrk / k6
7	Istio + Kiali	Fixed + queue	Асинхронная очередь	wrk / k6
8	Istio DestinationRule	Thread pool (размер 5)	Резервный API (альт. URL)	wrk / k6
9	Linkerd + Jaeger	Semaphore (лимит 10)	Кэш (Redis) + статика	wrk / k6
10	Istio + Grafana	Connection pool + timeout	Стратегия retry → fallback	wrk / k6
11	Istio VirtualService	Bulkhead + rate limiter	K8s readiness probe	wrk / k6
12	Linkerd + Envoy	Thread pool (7)	K8s headless service	wrk / k6
13	Istio + Prometheus	Semaphore (15)	Circuit Breaker chain	wrk / k6
14	Istio	Connection	Адаптивный fallback	wrk / k6

№	Service Mesh (Istio / Linkerd)	Bulkhead	Альтернативный сервис	Нагрузочное тестирование
	DestinationRule	pool (20)		
15	Linkerd + Flagger	Fixed pool (25)	Аварийный дата-центр	wrk / k6

4. КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Что такое каскадный отказ и как Circuit Breaker его предотвращает?
2. Перечислите и опишите три состояния Circuit Breaker.
3. Какие параметры важны при настройке Circuit Breaker?
4. Что такое Fallback и какие стратегии Fallback существуют?
5. В чём разница между Circuit Breaker и Retry?
6. Что значит «fail fast» в контексте Circuit Breaker?
7. Как Circuit Breaker помогает при временных сбоях зависимых сервисов?
8. Что такое Bulkhead и чем он отличается от Circuit Breaker?
9. Как проверить, что Circuit Breaker перешёл в состояние OPEN?
10. Как реализовать мониторинг Circuit Breaker (метрики, логи)?
11. В каких случаях следует использовать Circuit Breaker на стороне клиента, а в каких — на стороне сервера?
12. Что такое Timeout и как он сочетается с Circuit Breaker?
13. В чём отличие HALF-OPEN состояния от OPEN?
14. Как выбрать порог ошибок для Circuit Breaker?
15. Как тестировать систему с Circuit Breaker (Chaos Engineering)?

5. ТРЕБОВАНИЯ К ОТЧЕТУ

Отчёт должен содержать:

1. **Титульный лист** (ФИО, группа, вариант).
2. **Цель работы.**
3. **Теоретическая часть:**
 - Понятие каскадного отказа.
 - Паттерн Circuit Breaker (состояния, параметры).
 - Стратегии Fallback.
4. **Практическая часть:**
 - Листинг кода нестабильного сервиса.
 - Листинг кода клиентского сервиса с Circuit Breaker.
 - Листинг Fallback-функции.
 - Конфигурация Circuit Breaker (параметры).
5. **Результаты выполнения:**
 - Логи переходов состояний Circuit Breaker (CLOSED → OPEN → HALF-OPEN → CLOSED).
 - Скриншоты успешных вызовов.
 - Скриншоты fallback-ответов при OPEN состоянии.
 - Демонстрация восстановления после сбоя.
6. **Анализ результатов:**
 - Объяснение, почему при достижении порога ошибок сработал Circuit Breaker.
 - Объяснение механизма проверки восстановления (HALF-OPEN).
7. **Вывод** (как паттерн повышает отказоустойчивость).
8. **Ответы на контрольные вопросы** (3-5 по выбору).

6. КРИТЕРИИ ОЦЕНИВАНИЯ

Шкала оценивания по уровням сложности

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

Критерий	Базовый (10)	Средний (14)	Продвинутый (18)
Нестабильный сервис работает	2	2	1
Circuit Breaker настроен	3	2	1
Fallback реализован	2	2	1
Переходы состояний зафиксированы	3	2	2
Мониторинг состояний (эндпоинт)	—	2	1
Кэширование fallback (Redis)	—	2	1
Bulkhead изоляция	—	—	2
Альтернативный сервис	—	—	2
Prometheus + Grafana	—	—	2
Нагрузочное тестирование	—	—	2
Качество отчёта	—	2	3
Максимум	10	14	18

Шкала перевода баллов в оценку

Оценка	Базовый	Средний	Продвинутый
--------	---------	---------	-------------

О ценка	Ба зовый	Ср едний	Продви нутый
5	9- 10	13- 14	16-18
4	7-8	10- 12	12-15
3	5-6	7-9	8-11
2	<5	<7	<8

Штрафы

Нарушение	Штра ф
Отсутствие логов переходов состояний	-2
Отсутствие fallback при OPEN	-3
Работа не запускается	не засчитывается

7. ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ

Программа	Назначение	Доступность в РФ
Python + FastAPI/Flask	Разработка сервисов	+
Node.js + Express	Разработка сервисов	+
Java + Spring Boot	Разработка сервисов	+

Программа	Назначение	Доступность в РФ
PyBreaker / circuitbreaker	Библиотека Circuit Breaker (Python)	+
Resilience4j	Библиотека Circuit Breaker (Java)	+
opossum	Библиотека Circuit Breaker (Node.js)	+
Redis	Кэширование fallback	+
Docker (опционально)	Контейнеризация	+
Prometheus + Grafana (продвинутый)	Мониторинг	+
wrk / k6 (продвинутый)	Нагрузочное тестирование	+

Лабораторная работа №5

API Gateway: маршрутизация, агрегация и безопасность

Цель работы: Сформировать практические навыки настройки API Gateway как единой точки входа в микросервисную архитектуру, включая маршрутизацию запросов, агрегацию данных от нескольких микросервисов, ограничение частоты запросов (rate limiting) и аутентификацию/авторизацию на основе JWT.

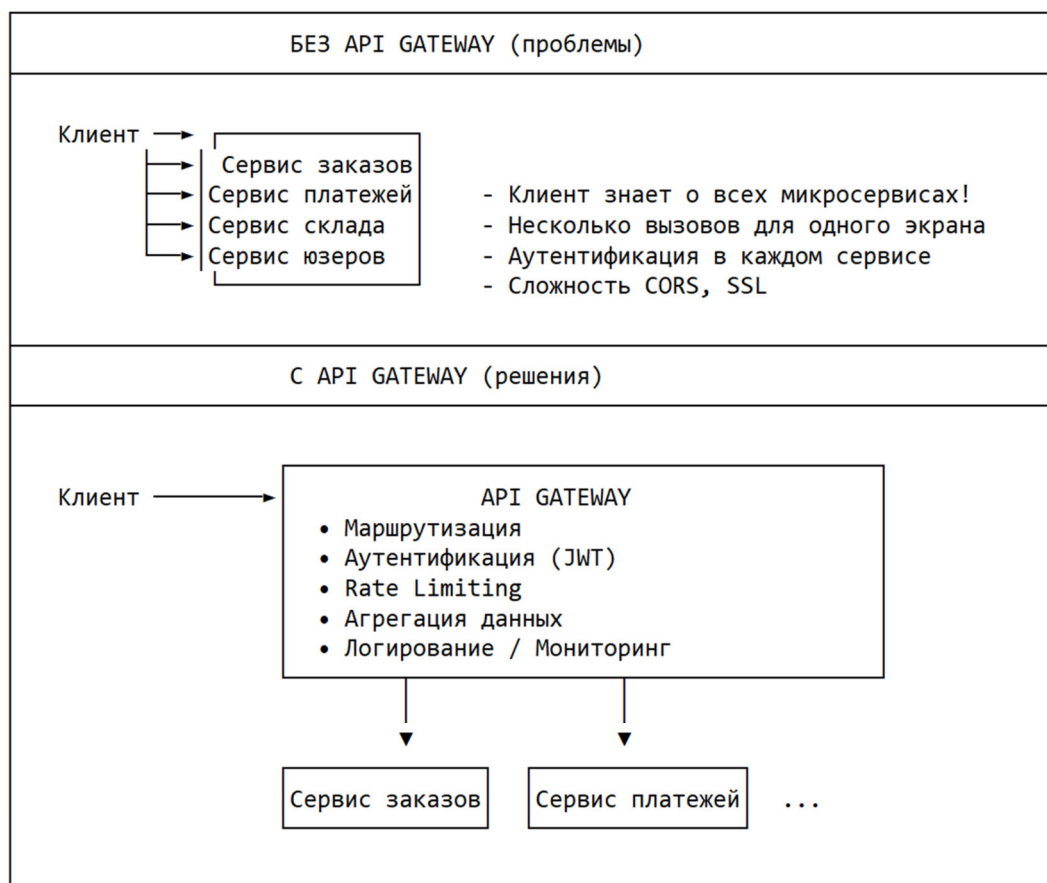
Формируемые компетенции: ПК-5

1. ТЕОРЕТИЧЕСКОЕ ОБОСНОВАНИЕ

1.1. Роль API Gateway в микросервисной архитектуре

API Gateway - это шаблон проектирования, при котором все внешние запросы к микросервисной системе проходят через единую точку входа, выполняющую функции маршрутизации, агрегации, аутентификации и мониторинга.

Проблемы, решаемые API Gateway:



Основные функции API Gateway:

Функция	Описание	Пример
Маршрутизация	Направление запросов к соответствующему микросервису	<code>/api/users/*</code> → сервис пользователей
Агрегация	Объединение данных из нескольких сервисов в один ответ	Профиль пользователя + его заказы

Функция	Описание	Пример
Аутентификация	Проверка JWT/API ключей перед передачей запроса	Валидация токена, проверка ролей
Rate Limiting	Ограничение количества запросов от одного клиента	100 запросов в минуту на IP
CORS	Настройка кросс-доменных запросов	Access-Control-Allow-Origin: *
Логирование	Централизованный сбор метрик и логов	Запись всех запросов в Elasticsearch
Трансформация	Изменение формата запроса/ответа	XML → JSON
Кэширование	Кэширование ответов на уровне Gateway	Кэш GET-запросов на 60 секунд

1.2. Популярные реализации API Gateway

Инструмент	Стек	Преимущества	Недостатки
NGINX	C	Высокая производительность, низкий порог входа, надёжность	Сложная агрегация данных
Kong	Lua/OpenResty	Плагины, поддержка	Сложность

Инструмент	Стек	Преимущества	Недостатки
		gRPC, админка	настройки плагинов
Spring Cloud Gateway	Java	Интеграция с Spring, флюентный API	Тяжёлый, подходит только для Java
Traefik	Go	Автоматическое обнаружение сервисов в Kubernetes, Let's Encrypt	Меньше гибкости
Envoy	C++	Service Mesh (Istio), большое сообщество	Высокий порог входа
Apache APISIX	Lua/Nginx	High performance, rich plugins	Относительно новое
Ambassador	Envoy/K8s	Нативная интеграция с Kubernetes	Только K8s
Ocelot	.NET	Простота для .NET-стека	Только .NET

1.3. Маршрутизация в API Gateway

Концепция маршрутизации:

Клиентский запрос → API Gateway → Правило маршрутизации → Целевой сервис

Пример конфигурации NGINX для маршрутизации:

Пример маршрутов для Spring Cloud Gateway (YAML):

```
spring:
  cloud:
    gateway:
      routes:
        - id: user-service
          uri: lb://user-service
          predicates:
            - Path=/api/users/**
          filters:
            - StripPrefix=1
        - id: order-service
          uri: lb://order-service
          predicates:
            - Path=/api/orders/**
          filters:
            - StripPrefix=1
```

1.4. Агрегация данных

Проблема: Клиенту нужны данные из нескольких микросервисов.

Без агрегации (N запросов):

Клиент → /api/users/123

Клиент → /api/orders?user_id=123

Клиент → /api/payments?user_id=123

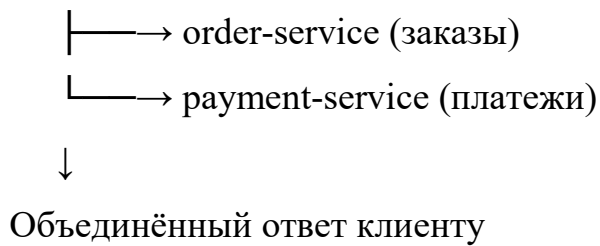
С агрегацией (1 запрос):

Клиент → /api/aggregate/user-dashboard/123

↓

API Gateway

└──→ user-service (профиль)



Пример реализации агрегации на Python (FastAPI Gateway):

```
import httpx
from fastapi import FastAPI
import asyncio

app = FastAPI()

USER_SERVICE = "http://user-service:8001"
ORDER_SERVICE = "http://order-service:8002"
PAYMENT_SERVICE = "http://payment-service:8003"

@app.get("/api/aggregate/dashboard/{user_id}")
async def get_user_dashboard(user_id: int):
    async with httpx.AsyncClient() as client:
        # Параллельные вызовы к трём сервисам
        user_task = client.get(f"{USER_SERVICE}/users/{user_id}")
        orders_task = client.get(f"{ORDER_SERVICE}/orders?user_id={user_id}")
        payments_task = client.get(f"{PAYMENT_SERVICE}/payments?user_id={user_id}")

        user_resp, orders_resp, payments_resp = await asyncio.gather(
            user_task, orders_task, payments_task,
            return_exceptions=True
        )

    return {
        "user": user_resp.json() if not isinstance(user_resp, Exception) else
        {"error": "User service unavailable"},
        "orders": orders_resp.json() if not isinstance(orders_resp, Exception)
        else [],
        "payments": payments_resp.json() if not isinstance(payments_resp,
        Exception) else []
    }
```

1.5. Rate Limiting (ограничение частоты запросов)

Основные алгоритмы Rate Limiting:

Алгоритм	Описание	Pros / Cons
----------	----------	-------------

Алгоритм	Описание	Pros / Cons
Fixed Window	Счётчик запросов за фиксированный интервал (например, минуту)	Прост, но имеет всплески на границах окон
Sliding Window	Скользящее окно с учётом более точной временной шкалы	Точнее, но сложнее
Token Bucket	«Ведро» токенов пополняется с постоянной скоростью	Гибкий, поддерживает всплески
Leaky Bucket	Запросы ставятся в очередь с постоянной скоростью обработки	Плавная нагрузка, но может накапливаться очередь

Пример конфигурации NGINX для Rate Limiting:

Определение зоны ограничения (10 запросов в секунду, всплеск до 20)

```
limit_req_zone $binary_remote_addr zone=mylimit:10m rate=10r/s;
```

```
server {
    location /api/ {
        # Применение rate limiting
        limit_req zone=mylimit burst=20 nodelay;
        proxy_pass http://backend;
    }
}
```

Пример Rate Limiting в Spring Cloud Gateway (YAML):

```
spring:
  cloud:
    gateway:
      routes:
        - id: api-route
          uri: lb://service
          predicates:
            - Path=/api/**
          filters:
            - name: RequestRateLimiter
              args:
                key-resolver: "#{@userKeyResolver}"
                redis-rate-limiter.replenishRate: 10
                redis-rate-limiter.burstCapacity: 20
```

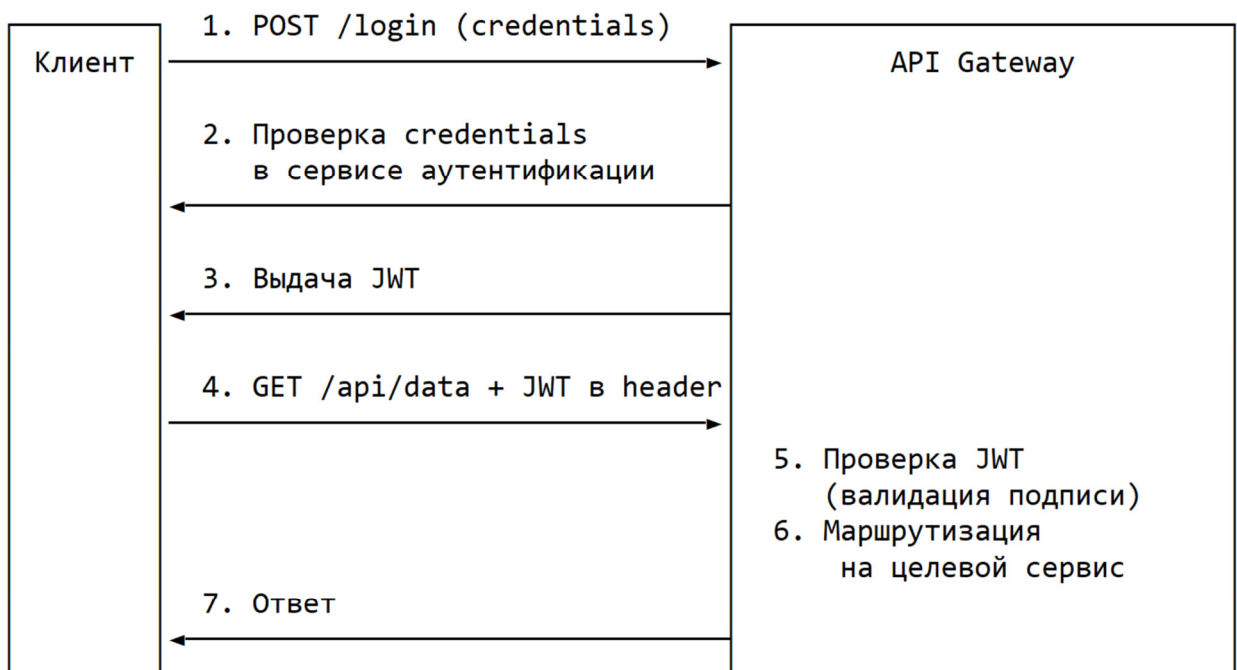
1.6. Аутентификация и авторизация (JWT)

JWT (JSON Web Token) - компактный, URL-безопасный способ передачи утверждений между сторонами.

Структура JWT:

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9 ← Header (алгоритм подписи, тип)
.eyJzdWIiOiIxMjMiLCJyb2xlIjoieWRtaW4ifQ ← Payload (claims: user_id, role)
.SflKxwRJSMeKKF2QT4fwpMeJf36POk6yJV_adQssw5c ← Signature
(подпись)

Поток аутентификации через API Gateway:



Пример проверки JWT в NGINX (с модулем njs):

```
location /api/ {
    # Проверка наличия JWT в заголовке Authorization
    if ($http_authorization = "") {
        return 401 '{"error": "Missing Authorization header"}';
    }

    # Передача JWT в бэкенд после проверки
    proxy_set_header Authorization $http_authorization;
    proxy_pass http://backend;
}
```

Пример JWT-валидации в Spring Cloud Gateway:

```
@Component
public class JwtAuthenticationFilter implements GlobalFilter {
    @Override
    public Mono<Void> filter(ServerWebExchange exchange, GatewayFilterChain
chain) {
        String authHeader =
exchange.getRequest().getHeaders().getFirst("Authorization");

        if (authHeader == null || !authHeader.startsWith("Bearer ")) {
            exchange.getResponse().setStatusCode(HttpStatus.UNAUTHORIZED);
        }
    }
}
```

```

        return exchange.getResponse().setComplete();
    }

    String token = authHeader.substring(7);
    try {
        // Валидация JWT
        Claims claims = Jwts.parser()
            .verifyWith(secretKey)
            .build()
            .parseSignedClaims(token)
            .getPayload();

        // Добавление claims в заголовки для downstream сервисов
        exchange = exchange.mutate()
            .request(r -> r.header("X-User-Id", claims.getSubject()))
            .build();

    } catch (Exception e) {
        exchange.getResponse().setStatusCode(HttpStatus.UNAUTHORIZED);
        return exchange.getResponse().setComplete();
    }

    return chain.filter(exchange);
}
}

```

2. ПРИМЕРЫ ПРАКТИЧЕСКОГО ВЫПОЛНЕНИЯ

2.1. Пример для NGINX + Python микросервисы

2.1.1. Структура проекта

```

api-gateway-demo/
├── gateway/
│   └── nginx.conf      # Конфигурация API Gateway NGINX
├── services/
│   ├── user-service/
│   │   ├── app.py      # Сервис пользователей
│   │   └── Dockerfile
│   ├── order-service/
│   │   ├── app.py      # Сервис заказов
│   │   └── Dockerfile
│   └── auth-service/
│       ├── app.py      # Сервис аутентификации (выдача JWT)
│       └── Dockerfile
├── docker-compose.yml  # Оркестрация всех сервисов
├── test_requests.py    # Тестовые запросы
└── requirements.txt

```

2.1.2. Сервис аутентификации (auth-service/app.py)

```
import jwt
import datetime
from fastapi import FastAPI, HTTPException
from pydantic import BaseModel

app = FastAPI(title="Auth Service")

SECRET_KEY = "your-secret-key-change-in-production"
ALGORITHM = "HS256"

class LoginRequest(BaseModel):
    username: str
    password: str

class LoginResponse(BaseModel):
    access_token: str
    token_type: str

# Простая имитация пользователей (в реальности — из БД)
USERS = {
    "admin": "admin123",
    "user": "user123"
}

@app.post("/login", response_model=LoginResponse)
async def login(request: LoginRequest):
    if USERS.get(request.username) != request.password:
        raise HTTPException(status_code=401, detail="Invalid credentials")

    # Создание JWT
    payload = {
        "sub": request.username,
        "role": "admin" if request.username == "admin" else "user",
        "exp": datetime.datetime.utcnow() + datetime.timedelta(hours=1)
    }
    token = jwt.encode(payload, SECRET_KEY, algorithm=ALGORITHM)

    return LoginResponse(access_token=token, token_type="bearer")

@app.get("/verify")
async def verify_token(token: str):
    try:
        payload = jwt.decode(token, SECRET_KEY, algorithms=[ALGORITHM])
        return {"valid": True, "username": payload.get("sub"), "role":
payload.get("role")}
    except jwt.InvalidTokenError:
        return {"valid": False}
```

2.1.3. Сервис пользователей (user-service/app.py)

```
from fastapi import FastAPI

app = FastAPI(title="User Service")

# Имитация БД пользователей
USERS = {
    1: {"id": 1, "name": "Alice", "email": "alice@example.com"},
    2: {"id": 2, "name": "Bob", "email": "bob@example.com"},
    3: {"id": 3, "name": "Charlie", "email": "charlie@example.com"}
}

@app.get("/users")
async def get_users():
    return list(USERS.values())

@app.get("/users/{user_id}")
async def get_user(user_id: int):
    return USERS.get(user_id, {"error": "User not found"})
```

2.1.4. Сервис заказов (order-service/app.py)

```
from fastapi import FastAPI

app = FastAPI(title="Order Service")

# Имитация БД заказов
ORDERS = {
    1: [
        {"id": 101, "product": "Laptop", "amount": 1200},
        {"id": 102, "product": "Mouse", "amount": 25}
    ],
    2: [
        {"id": 201, "product": "Keyboard", "amount": 75}
    ],
    3: []
}

@app.get("/orders")
async def get_orders(user_id: int = None):
    if user_id:
        return ORDERS.get(user_id, [])
    return ORDERS

@app.get("/orders/{order_id}")
async def get_order(order_id: int):
    # Поиск заказа во всех пользователях
    for user_id, orders in ORDERS.items():
```

```

    for order in orders:
        if order["id"] == order_id:
            return {"user_id": user_id, **order}
    return {"error": "Order not found"}

```

2.1.5. Конфигурация NGINX Gateway (nginx.conf)

```

# Ограничение частоты запросов
limit_req_zone $binary_remote_addr zone=api_limit:10m rate=10r/s;
limit_req_zone $binary_remote_addr zone=auth_limit:10m rate=5r/m;

upstream auth_upstream {
    server auth-service:8000;
}

upstream user_upstream {
    server user-service:8001;
}

upstream order_upstream {
    server order-service:8002;
}

server {
    listen 80;

    # Логирование
    access_log /var/log/nginx/access.log;
    error_log /var/log/nginx/error.log;

    # CORS настройки
    add_header Access-Control-Allow-Origin * always;
    add_header Access-Control-Allow-Methods "GET, POST, PUT, DELETE, OPTIONS"
always;
    add_header Access-Control-Allow-Headers "Authorization, Content-Type" always;

    # Обработка preflight запросов (OPTIONS)
    if ($request_method = 'OPTIONS') {
        return 204;
    }

```

Публичный эндпоинт для логина (без аутентификации)

```
location /api/auth/ {  
    limit_req zone=auth_limit burst=2 nodelay;  
    proxy_pass http://auth_upstream/;  
    proxy_set_header Host $host;  
    proxy_set_header X-Real-IP $remote_addr;  
}
```

Приватные эндпоинты — проверка JWT

```
location /api/users/ {  
    # Rate limiting для API  
    limit_req zone=api_limit burst=20 nodelay;
```

Проверка JWT

```
auth_request /api/auth/verify-jwt;  
auth_request_set $auth_status $upstream_status;
```

```
    proxy_pass http://user_upstream/;  
    proxy_set_header Host $host;  
    proxy_set_header X-Real-IP $remote_addr;  
}
```

```
location /api/orders/ {  
    limit_req zone=api_limit burst=20 nodelay;
```

```
    auth_request /api/auth/verify-jwt;
```

```
    proxy_pass http://order_upstream/;  
    proxy_set_header Host $host;  
    proxy_set_header X-Real-IP $remote_addr;  
}
```

Внутренний эндпоинт для валидации JWT (используется auth_request)

```
location = /api/auth/verify-jwt {  
    internal;  
    proxy_pass http://auth_upstream/verify?token=$http_authorization;  
    proxy_pass_request_body off;
```

```

        proxy_set_header Content-Length "";
    }

    # Агрегированный эндпоинт (дашборд пользователя)
    location /api/dashboard/ {
        limit_req zone=api_limit burst=10 nodelay;

        auth_request /api/auth/verify-jwt;

        # Агрегация через Lua (упрощённо: перенаправление на агрегатор)
        proxy_pass http://user-service:8001/dashboard-aggregate;
        proxy_set_header Host $host;
    }
}

```

2.1.6. Сервис-агрегатор (user-service с агрегацией)

```

# Добавим в user-service/app.py эндпоинт для агрегации
@app.get("/dashboard-aggregate/{user_id}")
async def get_dashboard_aggregate(user_id: int):
    async with httpx.AsyncClient() as client:
        # Параллельные вызовы к другим сервисам
        user_task = client.get(f"http://user-service:8001/users/{user_id}")
        orders_task = client.get(f"http://order-
service:8002/orders?user_id={user_id}")

        user_resp, orders_resp = await asyncio.gather(user_task, orders_task)

    return {
        "user": user_resp.json(),
        "orders": orders_resp.json()
    }

```

2.1.7. Docker Compose (docker-compose.yml)

```

version: '3.8'

networks:
  gateway-network:
    driver: bridge

services:
  nginx-gateway:
    image: nginx:alpine
    container_name: api-gateway
    ports:

```

```

    - "80:80"
volumes:
    - ./gateway/nginx.conf:/etc/nginx/conf.d/default.conf
depends_on:
    - auth-service
    - user-service
    - order-service
networks:
    - gateway-network

auth-service:
    build: ./services/auth-service
    container_name: auth-service
    environment:
        - SECRET_KEY=your-secret-key
    networks:
        - gateway-network

user-service:
    build: ./services/user-service
    container_name: user-service
    networks:
        - gateway-network

order-service:
    build: ./services/order-service
    container_name: order-service
    networks:
        - gateway-network

```

2.1.8. Тестовый скрипт (test_requests.py)

```

import httpx
import asyncio

BASE_URL = "http://localhost"

async def test_api_gateway():
    async with httpx.AsyncClient() as client:
        print("=== 1. Аутентификация ===")
        login_resp = await client.post(
            f"{BASE_URL}/api/auth/login",
            json={"username": "admin", "password": "admin123"}
        )
        token = login_resp.json()["access_token"]
        headers = {"Authorization": token}
        print(f"Получен JWT: {token[:50]}...\n")

        print("=== 2. Запрос пользователей ===")
        users_resp = await client.get(

```

```

        f"{BASE_URL}/api/users/users",
        headers=headers
    )
    print(f"Пользователи: {users_resp.json()}\n")

    print("=== 3. Запрос заказов ===")
    orders_resp = await client.get(
        f"{BASE_URL}/api/orders/orders?user_id=1",
        headers=headers
    )
    print(f"Заказы пользователя 1: {orders_resp.json()}\n")

    print("=== 4. Проверка Rate Limiting ===")
    for i in range(15):
        resp = await client.get(
            f"{BASE_URL}/api/users/users",
            headers=headers
        )
        if resp.status_code == 429:
            print(f"Достигнут лимит на {i+1}-м запросе!")
            break
        print(f"{i+1}: Успешно")

    print("\n=== 5. Доступ без токена (должен быть заблокирован) ===")
    resp = await client.get(f"{BASE_URL}/api/users/users")
    print(f"Статус без токена: {resp.status_code}")

if __name__ == "__main__":
    asyncio.run(test_api_gateway())

```

МЕТОДИЧЕСКИЕ УКАЗАНИЯ ДЛЯ ВЫПОЛНЕНИЯ ПРАКТИЧЕСКОЙ ЧАСТИ РАБОТЫ

Задачи работы:

1. **Разработка микросервисов:**
 - Микросервис аутентификации (выдача и валидация JWT)
 - Микросервис пользователей (CRUD пользователей)
 - Микросервис заказов (CRUD заказов)
2. **Настройка API Gateway:**
 - Настроить маршрутизацию запросов к соответствующим микросервисам
 - Настроить Rate Limiting (ограничение частоты запросов)
 - Настроить CORS для поддержки веб-клиентов

3. **Реализация аутентификации:**

- Настроить проверку JWT на уровне Gateway
- Блокировать запросы без валидного токена
- Передавать информацию о пользователе в downstream сервисы

4. **Реализация агрегации:**

○ Создать агрегированный эндпоинт, объединяющий данные из нескольких сервисов

○ Использовать параллельные вызовы для повышения производительности

5. **Тестирование:**

- Проверить маршрутизацию через Gateway
- Проверить аутентификацию (доступ без токена = 401)
- Проверить Rate Limiting (ограничение запросов)
- Проверить агрегацию данных

3. ВАРИАНТЫ ЗАДАНИЙ

3.1. Базовый уровень (15 вариантов)

Требования ко всем вариантам базового уровня:

- API Gateway на базе NGINX
- 2-3 микросервиса (один из них - аутентификация)
- Маршрутизация (без агрегации)
- Базовая проверка JWT

№	Микросервис 1	Микросервис 2	Микросервис 3 (опционально)
1	Пользователи	Заказы	—
2	Пользователи	Товары	—

№	Микросервис 1	Микросервис 2	Микросервис 3 (опционально)
3	Заказы	Платежи	—
4	Товары	Корзина	—
5	Пользователи	Отзывы	—
6	Заказы	Доставка	—
7	Платежи	Уведомления	—
8	Пользователи	Сессии	—
9	Каталог	Цены	Склад
10	Бронирования	Отмены	Возвраты
11	Билеты	Места	Схемы зала
12	Курсы	Студенты	Оценки
13	Статьи	Комментарии	Лайки
14	Файлы	Преобразования	Сжатие

№	Микросервис 1	Микросервис 2	Микросервис 3 (опционально)
15	Логи	Алерты	Инциденты

3.2. Средний уровень (15 вариантов)

Требования ко всем вариантам среднего уровня:

- Всё из базового + дополнительно:
 - Агрегация данных (один эндпоинт, объединяющий вызовы к 2-3 сервисам)
 - Rate Limiting на уровне Gateway (с настраиваемыми лимитами)
 - Разные лимиты для разных эндпоинтов (например, /login — 5/мин, /api — 100/мин)

№	Агрегируемые сервисы	Rate Limiting (эндпоинт → лимит)
1	Пользователи + Заказы	/api/auth → 5r/m, /api/* → 100r/m
2	Пользователи + Товары	/api/auth → 5r/m, /api/* → 200r/m
3	Заказы + Платежи	/api/auth → 10r/m, /api/* → 50r/m
4	Товары + Корзина + Цены	/api/auth → 5r/m, /api/* → 150r/m
5	Пользователи + Отзывы + Рейтинг	/api/auth → 3r/m, /api/* → 300r/m
6	Заказы + Доставка + Статус	/api/auth → 5r/m, /api/* → 100r/m
7	Платежи + Уведомления + Квитанции	/api/auth → 2r/m, /api/* → 60r/m
8	Каталог + Цены + Наличие	/api/auth → 10r/m, /api/* → 500r/m

№	Агрегируемые сервисы	Rate Limiting (эндпоинт → лимит)
9	Бронирования + Отмены + возвраты	/api/auth → 5r/m, /api/* → 80r/m
10	Билеты + Места + Схемы зала	/api/auth → 5r/m, /api/* → 120r/m
11	Курсы + Студенты + Оценки	/api/auth → 5r/m, /api/* → 200r/m
12	Статьи + Комментарии + Лайки	/api/auth → 10r/m, /api/* → 250r/m
13	Файлы + Преобразования + Сжатие	/api/auth → 5r/m, /api/* → 60r/m
14	Логи + Алерты + Инциденты	/api/auth → 2r/m, /api/* → 30r/m
15	Кэш + Счётчики + Лидерборд	/api/auth → 10r/m, /api/* → 400r/m

3.3. Продвинутый уровень (15 вариантов)

Требования ко всем вариантам продвинутого уровня:

- Всё из среднего + дополнительно:
 - Использование специализированного Gateway (Kong, Spring Cloud Gateway, Traefik)
 - Интеграция с Prometheus для сбора метрик (количество запросов, latency, коды ошибок)
 - А/В-тестирование канареечных релизов (canary deployment) через Gateway
 - Поддержка WebSocket/gRPC через Gateway

№	Gateway	Канареечные релизы (canary)	Метрики/Мониторинг
1	Kong + Postgres	10% трафика на v2, 90% на v1	Prometheus + Grafana
2	Kong + Cassandra	5% → 20% → 50% → 100% пошагово	Prometheus + Grafana
3	Spring Cloud Gateway	По аргументу (User-Agent)	Micrometer + Prometheus
4	Spring Cloud Gateway	По IP-адресу (белый список)	Micrometer + Prometheus
5	Traefik	По заголовку (X- Canary-Version)	Prometheus + Grafana
6	Traefik	По cookie (canary=1)	Prometheus + Grafana
7	Envoy	По проценту трафика (10% / 90%)	Prometheus + Grafana
8	Envoy + Istio	По весу сервиса (weighted routing)	Prometheus + Grafana
9	Apache APISIX	По географической зоне (ru/us/eu)	Prometheus + Grafana
10	Apache APISIX	По типу клиента (web/mobile/api)	Prometheus + Grafana

№	Gateway	Канареечные релизы (canary)	Метрики/Мониторинг
1	Ambassador (K8s)	По времени суток (day/night)	Prometheus + Grafana
2	Ambassador + Consul	По значению feature flag	Prometheus + Grafana
3	NGINX + Lua	Регулируемый процент (User-Agent regex)	Prometheus + Grafana
4	NGINX + njs	Ручное управление через API Gateway	Prometheus + Grafana
5	Envoy + Consul	Адаптивный canary (на базе метрик ошибок)	Prometheus + Grafana

4. КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Какие функции выполняет API Gateway в микросервисной архитектуре?
2. В чём отличие API Gateway от балансировщика нагрузки (Load Balancer)?
3. Что такое маршрутизация в API Gateway? Приведите пример конфигурации NGINX.
4. Что такое агрегация данных и зачем она нужна?
5. Какие алгоритмы Rate Limiting вы знаете? В чём их особенности?
6. Как настроить Rate Limiting в NGINX?

7. Что такое JWT и как он используется для аутентификации через API Gateway?
8. Как API Gateway проверяет JWT без передачи его в каждый микросервис?
9. Что такое CORS и как настроить его в API Gateway?
10. В чём разница между аутентификацией и авторизацией?
11. Какие преимущества даёт централизованное логирование на уровне Gateway?
12. Как API Gateway помогает при переходе от монолита к микросервисам?
13. Какие недостатки имеет подход с единым API Gateway?
14. Как обеспечить высокую доступность самого API Gateway?
15. Как API Gateway поддерживает сценарии A/B-тестирования и canary-релизов?

5. ТРЕБОВАНИЯ К ОТЧЕТУ

Отчёт должен содержать:

1. Титульный лист (ФИО, группа, вариант)
2. Цель работы
3. Теоретическая часть:
 - Что такое API Gateway, его роли
 - Функции маршрутизации, агрегации, rate limiting, аутентификации
4. Практическая часть:
 - Листинг кода всех микросервисов (с комментариями к ключевым частям)
 - Листинг конфигурации API Gateway
 - Листинг агрегирующего эндпоинта
 - Команды запуска
5. Результаты выполнения:

- Скриншоты запущенных сервисов (docker ps или логи)
 - Скриншоты прохождения аутентификации (получение JWT)
 - Скриншоты запросов через Gateway (curl или Postman)
 - Скриншот логов Rate Limiting (превышение лимита)
 - Скриншот агрегированного ответа
6. Анализ ошибок (если возникали)
 7. Вывод (что освоено)
 8. Ответы на контрольные вопросы (3-5 по выбору)

6. КРИТЕРИИ ОЦЕНИВАНИЯ

Шкала оценивания по уровням сложности

Критерий	Базовый (10)	Средний (14)	Продвинутый (18)
Микросервисы работают	2	1	1
Маршрутизация настроена	3	2	1
JWT аутентификация работает	3	2	1
Rate Limiting настроен	2	2	1
Агрегация данных (API Gateway)	—	3	2
Prometheus + метрики	—	—	2
Canary deployment	—	—	2
WebSocket/gRPC поддержка	—	—	2

Критерий	Базовый (10)	Средний (14)	Продвинутый (18)
Качество отчёта	—	4	4
Максимум	10	14	18

Шкала перевода баллов в оценку

Оценка	Базовый	Средний	Продвинутый
5	9-10	13-14	16-18
4	7-8	10-12	12-15
3	5-6	7-9	8-11
2	<5	<7	<8

Штрафы

Нарушение	Штраф
Нет проверки аутентификации через Gateway	-3
Rate Limiting не настроен	-2
Отсутствует скриншот агрегированного ответа (средний/продвинутый)	-2

Нарушение	Штраф
JWT валидируется в микросервисах, а не в Gateway (неправильная архитектура)	-2
Работа не запускается	не засчитывается

7. ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ

Программа	Назначение	Доступность в РФ
NGINX (алпайн образ)	API Gateway с маршрутизацией и rate limiting	+
Kong (опционально)	Специализированный API Gateway	+
Spring Cloud Gateway (опционально)	API Gateway для Java-стека	+
Traefik (опционально)	API Gateway для Kubernetes	+
Python + FastAPI	Разработка микросервисов	+
JWT-библиотека (PyJWT)	Генерация и валидация JWT	+
Docker + Docker Compose	Контейнеризация и	+

Программа	Назначение	Доступность в РФ
	оркестрация	
Postman / curl	Тестирование API Gateway	+
Prometheus (продвинутый)	Сбор метрик	+
Grafana (продвинутый)	Визуализация метрик	+

Лабораторная работа №6

Сравнительный анализ монолитной и микросервисной архитектур

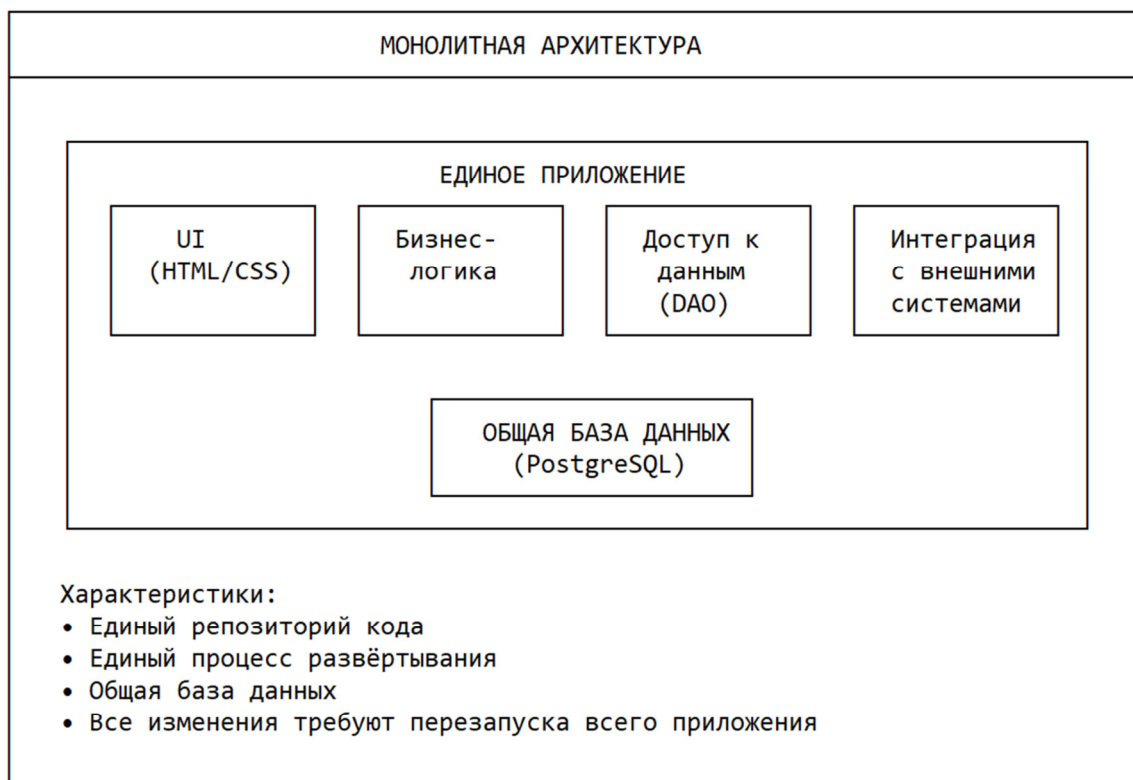
Цель работы: Провести практическое сравнение монолитной и микросервисной архитектур по критериям производительности, сложности разработки, отказоустойчивости и масштабируемости на примере одного и того же приложения, реализованного в двух вариантах.

Формируемые компетенции: ПК-5

1. ТЕОРЕТИЧЕСКОЕ ОБОСНОВАНИЕ

1.1. Монолитная архитектура: определение и характеристики

Монолитная архитектура - это подход, при котором все компоненты приложения (пользовательский интерфейс, бизнес-логика, доступ к данным) объединены в единый исполняемый модуль.



Преимущества монолита:

Преимущество	Описание
Простота разработки	Одна кодовая база, общие инструменты
Простота развёртывания	Один артефакт (JAR/WAR)
Простота тестирования	Сквозные тесты проще
Высокая производительность	Нет сетевых вызовов между компонентами
Простая отладка	Весь код в одном процессе
ACID-транзакции	Единая БД, строгая согласованность

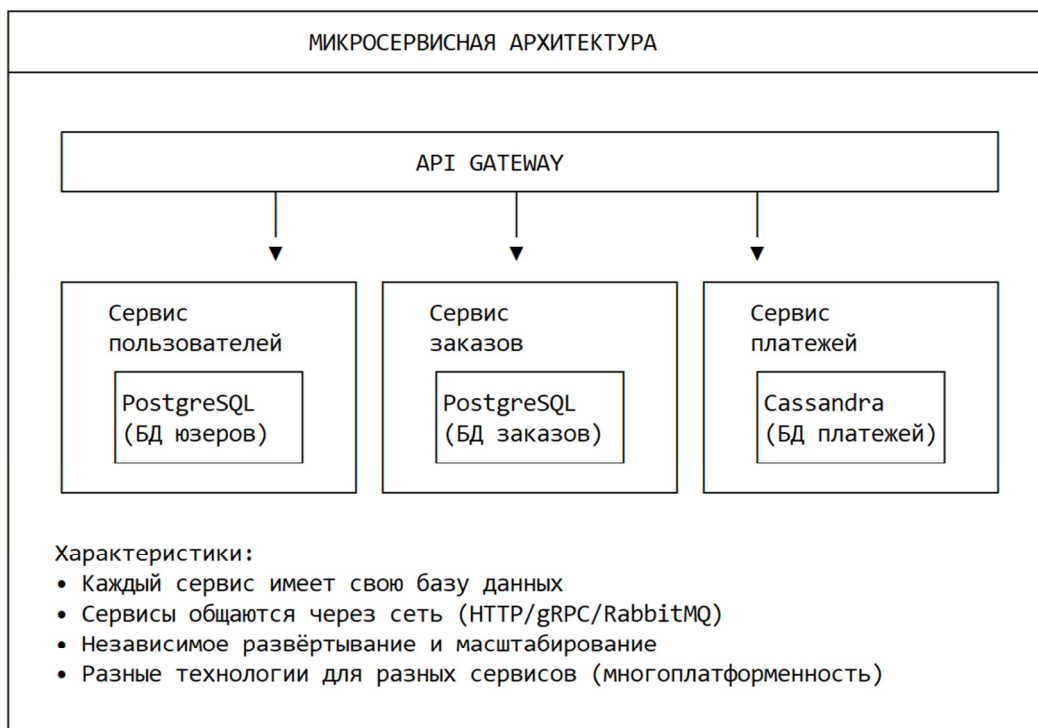
Недостатки монолита:

Недостаток	Описание
------------	----------

Недостаток	Описание
Сложность масштабирования	Приходится масштабировать всё приложение целиком
Высокая связанность	Изменение в одном модуле может сломать другие
Медленный цикл разработки	Даже малое изменение требует пересборки и перезапуска
Технологическая стагнация	Трудно внедрить новую технологию
Риск монополии базы	Одна БД может стать узким местом
Сложность онбординга	Кодовая база может быть огромной и запутанной

1.2. Микросервисная архитектура: определение и характеристики

Микросервисная архитектура - это подход, при котором приложение строится как набор небольших, независимо развёртываемых сервисов, каждый из которых реализует определённую бизнес-функцию.



Преимущества микросервисов:

Преимущество	Описание
Независимое масштабирование	Масштабируется только нагруженный сервис
Независимое развёртывание	Изменение в одном сервисе не требует перезапуска других
Технологическая свобода	Каждый сервис можно писать на своём языке/стеке
Изоляция отказов	Отказ одного сервиса не «роняет» всю систему
Независимые команды	Каждая команда может разрабатывать свой сервис

Преимущество	Описание
Адаптация под нагрузку	Легко масштабировать только те части системы, которые это требуют

Недостатки микросервисов:

Недостаток	Описание
Сложность распределённых транзакций	ACID заменяется Saga, eventual consistency
Сетевые задержки	Сетевые вызовы медленнее внутрипроцессных
Сложность отладки	Запрос проходит через несколько сервисов
Сложность тестирования	Нужны интеграционные тесты
Управление конфигурацией	Много сервисов → много конфигураций
Сложность мониторинга	Нужна распределённая трассировка

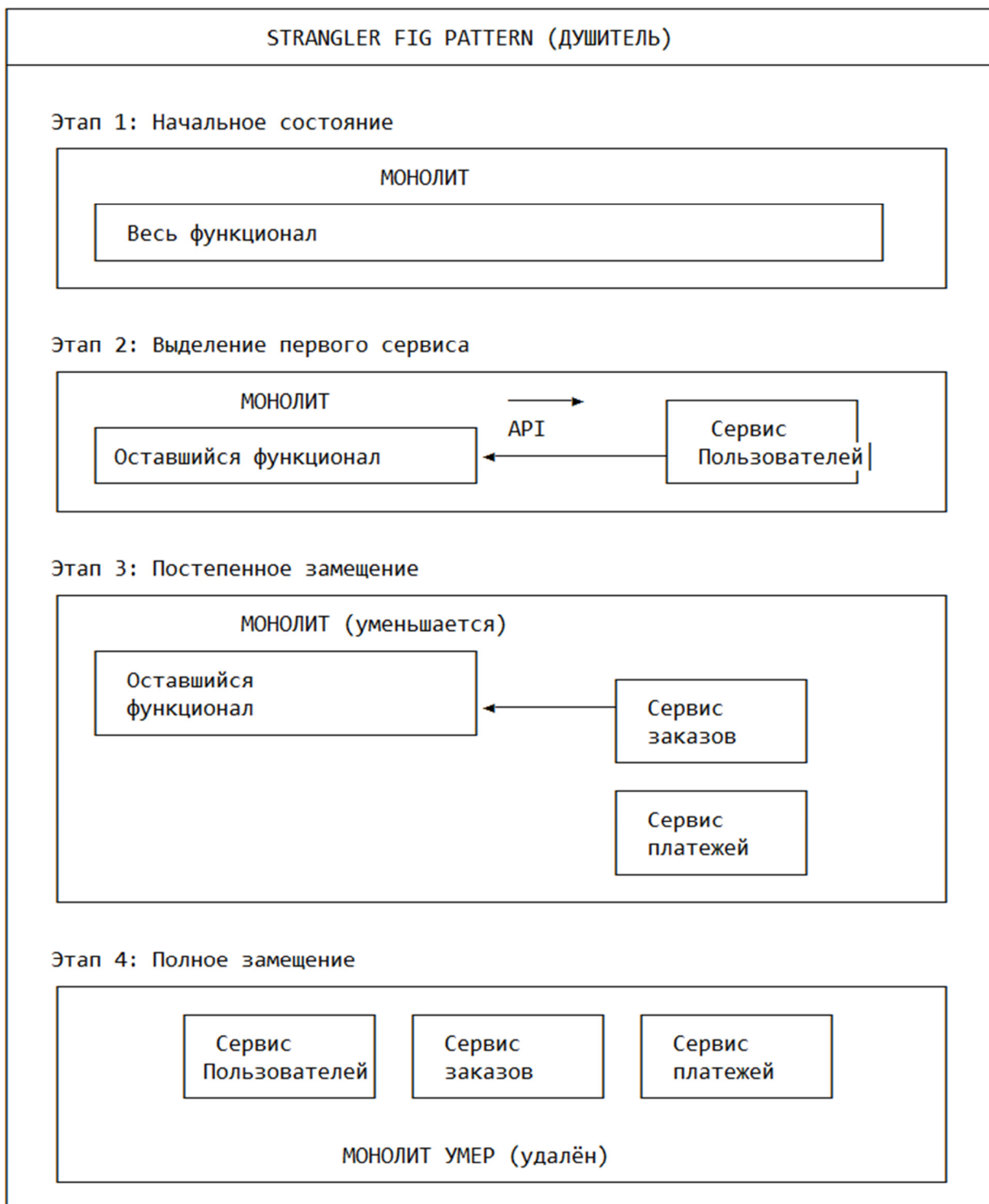
1.3. Сравнительная таблица: монолит vs микросервисы

Критерий	Монолит	Микросервисы
Производительность (latency)	Низкая (нет сетевых вызовов)	Выше из-за сетевых вызовов
Пропускная способность (RPS)	Высокая (минимум оверхеда)	Ниже (сетевой оверхед + сериализация)

Критерий	Монолит	Микросервисы
Горизонтальное масштабирование	Сложное (масштабируется целиком)	Простое (масштабируется каждый сервис)
Время развёртывания	Медленное (всё приложение целиком)	Быстрое (только изменённый сервис)
Время сборки	Растёт с размером кодовой базы	Линейное (зависит от размера сервиса)
Отказоустойчивость	Низкая (отказ одного модуля валит всё)	Высокая (изоляция отказов)
Сложность разработки (начальная)	Низкая	Высокая
Сложность разработки (масштаб 50+ разработчиков)	Очень высокая	Умеренная
Технологическая гибкость	Нет (все связаны одним стеком)	Да (каждый сервис может быть на своём стеке)
ACID-транзакции	Есть (легко)	Нет (требуется Saga/2PC/eventual consistency)

1.4. Стратегии миграции от монолита к микросервисам

Strangler Fig Pattern (паттерн «душитель»):



Основные паттерны миграции:

Паттерн	Описание	Когда применять
---------	----------	-----------------

Паттерн	Описание	Когда применять
Strangler Fig	Постепенное замещение монолита микросервисами	Безопасный, низкорисковый подход
Branch by Abstraction	Внедрение абстракции перед реализацией микросервиса	Когда нужна обратная совместимость
Anti-Corruption Layer	Защита нового кода от legacy-структур	При интеграции со старыми системами
Event Interception	Перехват событий для миграции	Асинхронная обработка

1.5. Пример приложения для сравнения

В данной лабораторной работе рассматривается **система управления задачами (Task Manager)** со следующей функциональностью:

- Регистрация/аутентификация пользователей
- Управление задачами (CRUD)
- Назначение задач пользователям
- Комментарии к задачам
- Уведомления о статусе задач

Монолитная реализация: одно приложение на FastAPI/Flask с общей БД PostgreSQL.

Микросервисная реализация:

- Сервис аутентификации (JWT)
- Сервис пользователей (профили)
- Сервис задач (CRUD задач)
- Сервис комментариев

- Сервис уведомлений (RabbitMQ)
- API Gateway (NGINX/Kong)

2. ПРИМЕРЫ ПРАКТИЧЕСКОГО ВЫПОЛНЕНИЯ

2.1. Пример монолитной реализации (Python/FastAPI)

2.1.1. Структура монолита

```
monolith/
├── app/
│   ├── __init__.py
│   ├── main.py           # Точка входа FastAPI
│   └── models.py         # SQLAlchemy модели (User, Task,
Comment)
│   ├── schemas.py       # Pydantic схемы
│   ├── database.py      # Подключение к БД
│   ├── auth.py          # Аутентификация и JWT
│   ├── users.py         # Роуты пользователей
│   ├── tasks.py         # Роуты задач
│   ├── comments.py      # Роуты комментариев
│   └── notifications.py  # Сервис уведомлений (внутренний)
├── requirements.txt
├── Dockerfile
└── docker-compose.yml
```

2.1.2. Монолит: основное приложение (main.py)

```
from fastapi import FastAPI, Depends, HTTPException
from fastapi.security import HTTPBearer, HTTPAuthorizationCredentials
from sqlalchemy.orm import Session
from . import models, schemas, auth, users, tasks, comments
from .database import engine, get_db

# Создание таблиц
models.Base.metadata.create_all(bind=engine)

app = FastAPI(title="Task Manager (Monolith)")
```

```

security = HTTPBearer()

# Вспомогательная зависимость для получения текущего пользователя
async def get_current_user(
    credentials: HTTPAuthorizationCredentials = Depends(security),
    db: Session = Depends(get_db)
) -> models.User:
    token = credentials.credentials
    payload = auth.verify_token(token)
    if not payload:
        raise HTTPException(status_code=401, detail="Invalid token")

    user = db.query(models.User).filter(models.User.id ==
payload.get("user_id")).first()
    if not user:
        raise HTTPException(status_code=401, detail="User not found")
    return user

# Подключение роутеров
app.include_router(auth.router, prefix="/api/auth", tags=["auth"])
app.include_router(users.router, prefix="/api/users", tags=["users"],
dependencies=[Depends(get_current_user)])
app.include_router(tasks.router, prefix="/api/tasks", tags=["tasks"],
dependencies=[Depends(get_current_user)])
app.include_router(comments.router, prefix="/api/comments", tags=["comments"],
dependencies=[Depends(get_current_user)])

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

```

2.1.3. Монолит: модель данных (models.py)

```

from sqlalchemy import Column, Integer, String, Boolean, DateTime, ForeignKey,
Text
from sqlalchemy.orm import relationship
from datetime import datetime
from .database import Base

class User(Base):
    __tablename__ = "users"
    id = Column(Integer, primary_key=True, index=True)
    username = Column(String(50), unique=True, index=True)
    email = Column(String(100), unique=True, index=True)
    hashed_password = Column(String(255))
    is_active = Column(Boolean, default=True)
    created_at = Column(DateTime, default=datetime.utcnow)

    tasks = relationship("Task", back_populates="owner")

```

```

        comments = relationship("Comment", back_populates="author")

class Task(Base):
    __tablename__ = "tasks"
    id = Column(Integer, primary_key=True, index=True)
    title = Column(String(200))
    description = Column(Text)
    status = Column(String(20), default="pending") # pending, in_progress, done
    owner_id = Column(Integer, ForeignKey("users.id"))
    created_at = Column(DateTime, default=datetime.utcnow)

    owner = relationship("User", back_populates="tasks")
    comments = relationship("Comment", back_populates="task")

class Comment(Base):
    __tablename__ = "comments"
    id = Column(Integer, primary_key=True, index=True)
    content = Column(Text)
    task_id = Column(Integer, ForeignKey("tasks.id"))
    author_id = Column(Integer, ForeignKey("users.id"))
    created_at = Column(DateTime, default=datetime.utcnow)

    task = relationship("Task", back_populates="comments")
    author = relationship("User", back_populates="comments")

```

2.1.4. Монолит: сервис задач (tasks.py)

```

from fastapi import APIRouter, Depends, HTTPException, status
from sqlalchemy.orm import Session
from .. import models, schemas
from ..database import get_db

router = APIRouter()

@router.get("/")
async def get_tasks(
    skip: int = 0,
    limit: int = 100,
    current_user: models.User = Depends(get_current_user),
    db: Session = Depends(get_db)
):
    """Получение всех задач текущего пользователя"""
    tasks = db.query(models.Task).filter(
        models.Task.owner_id == current_user.id
    ).offset(skip).limit(limit).all()
    return tasks

@router.post("/", status_code=status.HTTP_201_CREATED)
async def create_task(
    task: schemas.TaskCreate,

```

```

    current_user: models.User = Depends(get_current_user),
    db: Session = Depends(get_db)
):
    """Создание новой задачи"""
    db_task = models.Task(
        title=task.title,
        description=task.description,
        owner_id=current_user.id
    )
    db.add(db_task)
    db.commit()
    db.refresh(db_task)

    # Отправка уведомления (синхронно, в монолите)
    await send_notification(current_user.id, f"Создана задача: {task.title}")

    return db_task

@router.get("/{task_id}")
async def get_task(
    task_id: int,
    current_user: models.User = Depends(get_current_user),
    db: Session = Depends(get_db)
):
    task = db.query(models.Task).filter(
        models.Task.id == task_id,
        models.Task.owner_id == current_user.id
    ).first()
    if not task:
        raise HTTPException(status_code=404, detail="Task not found")
    return task

```

2.2. Пример микросервисной реализации

2.2.1. Структура микросервисов

```

microservices/
├── api-gateway/
│   └── nginx.conf
├── auth-service/
│   ├── app.py
│   └── Dockerfile
├── user-service/
│   └── app.py

```

```

|   |—— Dockerfile
|   |—— users.db (SQLite или PostgreSQL)
|—— task-service/
|   |—— app.py
|   |—— Dockerfile
|   |—— tasks.db
|—— notification-service/
|   |—— app.py
|   |—— Dockerfile
|   |—— consumer.py
|—— docker-compose.yml
|—— test_load.py

```

2.2.2. Task Service (микросервис задач)

```

# task-service/app.py
from fastapi import FastAPI, HTTPException, Depends
from sqlalchemy import create_engine, Column, Integer, String, JSON, DateTime
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.orm import sessionmaker, Session
import httpx
import os
from datetime import datetime

app = FastAPI(title="Task Service")

# База данных
DATABASE_URL = os.getenv("DATABASE_URL", "sqlite:///./tasks.db")
engine = create_engine(DATABASE_URL)
SessionLocal = sessionmaker(autocommit=False, autoflush=False, bind=engine)
Base = declarative_base()

class TaskModel(Base):
    __tablename__ = "tasks"
    id = Column(Integer, primary_key=True, index=True)
    title = Column(String(200))
    description = Column(String(500))
    status = Column(String(20), default="pending")
    owner_id = Column(Integer)
    created_at = Column(DateTime, default=datetime.utcnow)

Base.metadata.create_all(bind=engine)

```

```

def get_db():
    db = SessionLocal()
    try:
        yield db
    finally:
        db.close()

# Зависимость для проверки JWT через Auth Service
async def get_current_user(authorization: str = None):
    if not authorization:
        raise HTTPException(status_code=401, detail="Missing token")

    async with httpx.AsyncClient() as client:
        response = await client.get(
            "http://auth-service:8001/verify",
            headers={"Authorization": authorization}
        )
        if response.status_code != 200:
            raise HTTPException(status_code=401, detail="Invalid token")
    return response.json()

@app.get("/tasks")
async def get_tasks(
    user: dict = Depends(get_current_user),
    db: Session = Depends(get_db)
):
    """Получение задач пользователя"""
    tasks = db.query(TaskModel).filter(TaskModel.owner_id ==
user["user_id"]).all()
    return tasks

@app.post("/tasks")
async def create_task(
    title: str,
    description: str,
    user: dict = Depends(get_current_user),
    db: Session = Depends(get_db)
):
    """Создание задачи с отправкой уведомления через RabbitMQ"""
    task = TaskModel(
        title=title,
        description=description,
        owner_id=user["user_id"]
    )
    db.add(task)
    db.commit()
    db.refresh(task)

# Асинхронная отправка уведомления через RabbitMQ
async with httpx.AsyncClient() as client:

```

```

    await client.post(
        "http://notification-service:8002/notify",
        json={
            "user_id": user["user_id"],
            "message": f"Task created: {title}"
        }
    )

return task

```

2.2.3. Docker Compose для микросервисов

```

version: '3.8'

services:
  api-gateway:
    image: nginx:alpine
    ports:
      - "80:80"
    volumes:
      - ./api-gateway/nginx.conf:/etc/nginx/conf.d/default.conf
    depends_on:
      - auth-service
      - user-service
      - task-service
      - notification-service

  auth-service:
    build: ./auth-service
    environment:
      - SECRET_KEY=your-secret-key
    ports:
      - "8001:8001"

  user-service:
    build: ./user-service
    environment:
      - DATABASE_URL=postgresql://user:pass@user-db:5432/users
    depends_on:
      - user-db

  user-db:
    image: postgres:15
    environment:
      - POSTGRES_USER=user
      - POSTGRES_PASSWORD=pass
      - POSTGRES_DB=users

  task-service:

```

```

build: ./task-service
environment:
  - DATABASE_URL=postgresql://user:pass@task-db:5432/tasks
depends_on:
  - task-db

task-db:
  image: postgres:15
  environment:
    - POSTGRES_USER=user
    - POSTGRES_PASSWORD=pass
    - POSTGRES_DB=tasks

notification-service:
  build: ./notification-service
  environment:
    - RABBITMQ_URL=amqp://guest:guest@rabbitmq:5672
  depends_on:
    - rabbitmq

rabbitmq:
  image: rabbitmq:3-management
  ports:
    - "15672:15672"

```

МЕТОДИЧЕСКИЕ УКАЗАНИЯ ДЛЯ ВЫПОЛНЕНИЯ ПРАКТИЧЕСКОЙ ЧАСТИ РАБОТЫ

Задачи работы:

1. **Реализация монолитного приложения:**
 - Разработать монолитное приложение с функциональностью Task Manager
 - Реализовать аутентификацию, CRUD задач, комментарии
 - Запустить монолит и проверить работу
2. **Реализация микросервисного приложения:**
 - Разделить монолит на 3-5 микросервисов
 - Реализовать API Gateway (NGINX/Kong)
 - Настроить обмен между сервисами через HTTP/gRPC/RabbitMQ
3. **Проведение сравнительных тестов:**
 - Измерить RPS (запросов в секунду) для разных эндпоинтов
 - Измерить latency (p50, p95, p99) для каждого запроса

- Провести нагрузочное тестирование (wrk/k6) для обоих вариантов

4. Тестирование отказоустойчивости:

- Остановить один сервис в микросервисной архитектуре (например, task-service)
- Проверить, осталась ли доступна остальная функциональность
- Сравнить с монолитом (остановка приложения = полная недоступность)

5. Сравнение сложности разработки:

- Оценить время разработки монолита vs микросервисов (фактически или субъективно)
- Сравнить количество кода и сложность конфигураций

6. Оформление отчёта:

- Результаты нагрузочного тестирования (графики)
- Таблица сравнения характеристик
- Анализ Trade-Offs (компромиссов)

3. ВАРИАНТЫ ЗАДАНИЙ

3.1. Базовый уровень (15 вариантов)

Требования ко всем вариантам базового уровня:

1. Реализовать простое монолитное приложение (FastAPI / Flask / Node.js)
2. Реализовать микросервисную версию того же приложения (2-3 микросервиса)
3. Провести базовое нагрузочное тестирование

№	Тип приложения	Функциональность (монолит)	Количество микросервисов
1	Task Manager	Регистрация, задачи,	3 (auth, tasks,

№	Тип приложения	Функциональность (монолит)	Количество микросервисов
		комментарии	comments)
2	Book Store	Каталог, корзина, заказы	3 (catalog, cart, orders)
3	Blog	Посты, комментарии, лайки	3 (posts, comments, likes)
4	E-commerce	Товары, корзина, оплата	3 (products, cart, payment)
5	Social Media	Профиль, посты, подписки	3 (users, posts, follows)
6	Delivery	Рестораны, меню, заказы	3 (restaurants, orders, delivery)
7	Hotel Booking	Номера, бронирования, отзывы	3 (rooms, bookings, reviews)
8	Event Manager	События, регистрация, билеты	3 (events, registration, tickets)
9	Questionnaire	Анкеты, ответы, статистика	3 (forms, responses, stats)
10	Inventory	Товары, склад, поставки	3 (products, stock, supply)
11	Subscription	Планы, подписки,	3 (plans,

№	Тип приложения	Функциональность (монолит)	Количество микросервисов
		платежи	subscriptions, billing)
12	Job Board	Вакансии, резюме, отклики	3 (jobs, resumes, applications)
13	Course Platform	Курсы, уроки, прогресс	3 (courses, lessons, progress)
14	CRM	Контакты, сделки, задачи	3 (contacts, deals, tasks)
15	Chat App	Комнаты, сообщения, пользователи	3 (rooms, messages, users)

3.2. Средний уровень (15 вариантов)

Требования ко всем вариантам среднего уровня:

1. Всё из базового + дополнительно:
2. Использовать отдельные БД для разных микросервисов
3. Асинхронное взаимодействие через очереди (RabbitMQ/Kafka)
4. API Gateway с аутентификацией и rate limiting

№	Асинхронное взаимодействие	Отдельные БД	API Gateway
1	RabbitMQ (task → notif)	PostgreSQL ×3	NGINX
2	RabbitMQ (order → payment)	PostgreSQL + Redis ×2	NGINX
3	Kafka (post → analytics)	MongoDB +	Kong

№	Асинхронное взаимодействие	Отдельные БД	API Gateway
		PostgreSQL	
4	RabbitMQ (booking → email)	PostgreSQL ×2 + Redis	Kong
5	Kafka (event → logger)	MongoDB ×2	Traefik
6	RabbitMQ (payment → invoice)	PostgreSQL ×2 + Redis	Traefik
7	Redis Streams (user → stats)	PostgreSQL + MongoDB	NGINX
8	RabbitMQ (comment → moderation)	PostgreSQL ×2	NGINX
9	Kafka (order → warehouse)	PostgreSQL ×2 + Cassandra	Kong
10	RabbitMQ (upload → process)	MongoDB + S3-minio	Kong
11	Redis Pub/Sub (chat → presence)	Redis + PostgreSQL	Traefik
12	RabbitMQ (task → deadline)	PostgreSQL ×2	NGINX
13	Kafka (click → analytics)	ClickHouse + PostgreSQL	Kong

№	Асинхронное взаимодействие	Отдельные БД	API Gateway
4	1 RabbitMQ (quote → price)	PostgreSQL + Redis	NGINX
5	1 Redis Streams (device → telemetry)	TimescaleDB + PostgreSQL	Traefik

3.3. Продвинутый уровень (15 вариантов)

Требования ко всем вариантам продвинутого уровня:

1. Всё из среднего + дополнительно:
2. Разные технологии для разных сервисов (полиглотность)
3. Разные БД для разных сервисов (PostgreSQL, MongoDB, Redis, Cassandra)
4. Реализация Saga для распределённых транзакций
5. Service Mesh (Istio/Linkerd) для observability

№	Saga тип	Service Mesh	Полиглотность сервисов
1	Choreography (события)	Istio	Python (auth), Go (tasks), Node (notif)
2	Orchestration (оркестратор)	Istio	Java (orders), Python (payment), Go (stock)
3	Choreography (Kafka)	Linkerd	Python (users), Node (posts), Go (analytics)
4	Orchestration (Temporal)	Istio	Go (workflow), Java (booking), Python (payment)
5	Choreography	Linkerd	Python (cart), Node (checkout), Go

№	Saga тип	Service Mesh	Полиглотность сервисов
	(RabbitMQ)		(delivery)
6	Orchestration (Camunda)	Istio	Java (order), Python (inventory), Node (shipping)
7	Choreography (Redis Streams)	Linkerd	Go (game), Python (match), Node (leaderboard)
8	Orchestration (AWS Step Functions)	Istio	Python (process), Go (transcode), Node (notify)
9	Choreography + DA (dead letter)	Linkerd	Node (upload), Python (convert), Go (store)
10	Orchestration (Zeebe)	Istio	Go (order), Java (payment), Python (fulfill)
11	Choreography (NATS)	Linkerd	Python (sensor), Go (ingest), Node (alert)
12	Orchestration (Conductor)	Istio	Java (orchestrator), Go (task1), Python (task2)
13	Choreography (Pulsar)	Linkerd	Python (producer), Go (processor), Node (sink)
14	2-phase commit	Istio	Java (coordinator), Go (participant1), Python (participant2)

№	Saga тип	Service Mesh	Полиглотность сервисов
	(эмуляция)		
15	Hybrid (choreography + orchestration)	Linkerd	Node (front), Go (saga worker), Python (compensation)

4. КОНТРОЛЬНЫЕ ВОПРОСЫ

1. В чём основные различия между монолитной и микросервисной архитектурами?
2. Какие преимущества даёт микросервисная архитектура при масштабировании?
3. Почему микросервисы обычно имеют более высокую latency, чем монолит?
4. Какую архитектуру вы выберете для высоконагруженной платёжной системы и почему?
5. Что такое Strangler Fig Pattern и зачем он используется?
6. Какие проблемы возникают при распределённых транзакциях в микросервисах?
7. Как изоляция отказов работает в микросервисной архитектуре?
8. В каком случае монолит всё ещё может быть предпочтительнее микросервисов?
9. Как вы тестируете производительность монолита vs микросервисов?
10. Что важнее для high-load системы: низкая latency или высокая пропускная способность?
11. Какие метрики (RPS, p95 latency) оказались лучше в монолите, а какие — в микросервисах в вашем эксперименте?

12. Почему микросервисы требуют API Gateway?
13. Что такое «распределённый монолит» и чем он хуже классического монолита?
14. Как выбрать границы микросервисов (Domain-Driven Design)?
15. Каковы минимальные требования к команде для успешного перехода на микросервисы?

5. ТРЕБОВАНИЯ К ОТЧЕТУ

Отчёт должен содержать:

1. Титульный лист
2. Цель работы
3. Теоретическая часть:
 - Сравнение двух архитектур (таблица)
 - Паттерны миграции (Strangler Fig)
4. Практическая часть:
 - Листинг кода монолита
 - Листинг кода микросервисов (основные компоненты)
 - Docker Compose файлы
5. Результаты нагрузочного тестирования:
 - Таблица RPS для разных эндпоинтов
 - Таблица latency (p50, p95, p99)
 - Графики сравнения
6. Результаты тестирования отказоустойчивости:
 - Описание сценария отказа (сервис остановлен)
 - Сравнение поведения монолита и микросервисов
7. Анализ Trade-Offs:
 - Таблица плюсов и минусов для конкретного приложения
 - Рекомендации по выбору архитектуры
8. Вывод
9. Ответы на контрольные вопросы

6. КРИТЕРИИ ОЦЕНИВАНИЯ

Шкала оценивания

Критерий	Базовый (10)	Средний (14)	Продвинутый (18)
Монолит реализован и работает	3	2	1
Микросервисы реализованы и работают	3	2	1
Нагрузочное тестирование проведено	2	2	2
Сравнительная таблица заполнена	2	2	2
Разные БД для разных сервисов	—	2	1
Асинхронное взаимодействие через RabbitMQ	—	2	1
Saga распределённых транзакций	—	—	3
Service Mesh (observability)	—	—	3
Качество отчёта и выводы	—	2	4

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

Шкала перевода в оценку

О ценка	Ба зовый	Ср едний	Продви нутый
5	9- 10	13- 14	16-18
4	7-8	10- 12	12-15
3	5-6	7-9	8-11
2	<5	<7	<8

Штрафы

Нарушение	Штра ф
Нет нагрузочного тестирования	-3
Нет сравнительного анализа (таблицы с метриками)	-3
Микросервисы не работают независимо (одна общая БД)	-3
Отсутствие графиков latency/RPS	-2
Работа не запускается	не засчитывается

7. ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ

Программа	Назначение	Доступность в РФ
Python + FastAPI	Разработка приложений	+
Node.js + Express	Альтернативный стек	+
Docker + Compose	Контейнеризация	+
PostgreSQL / SQLite	Базы данных	+
MongoDB	NoSQL БД	+
Redis	Кэш и брокер	+
RabbitMQ / Kafka	Очереди сообщений	+
NGINX / Kong / Traefik	API Gateway	+
wrk / k6	Нагрузочное тестирование	+
Istio / Linkerd	Service Mesh (продвинутый)	+
Prometheus + Grafana	Мониторинг (опционально)	+

Лабораторная работа №7

Репликация баз данных: синхронная и асинхронная

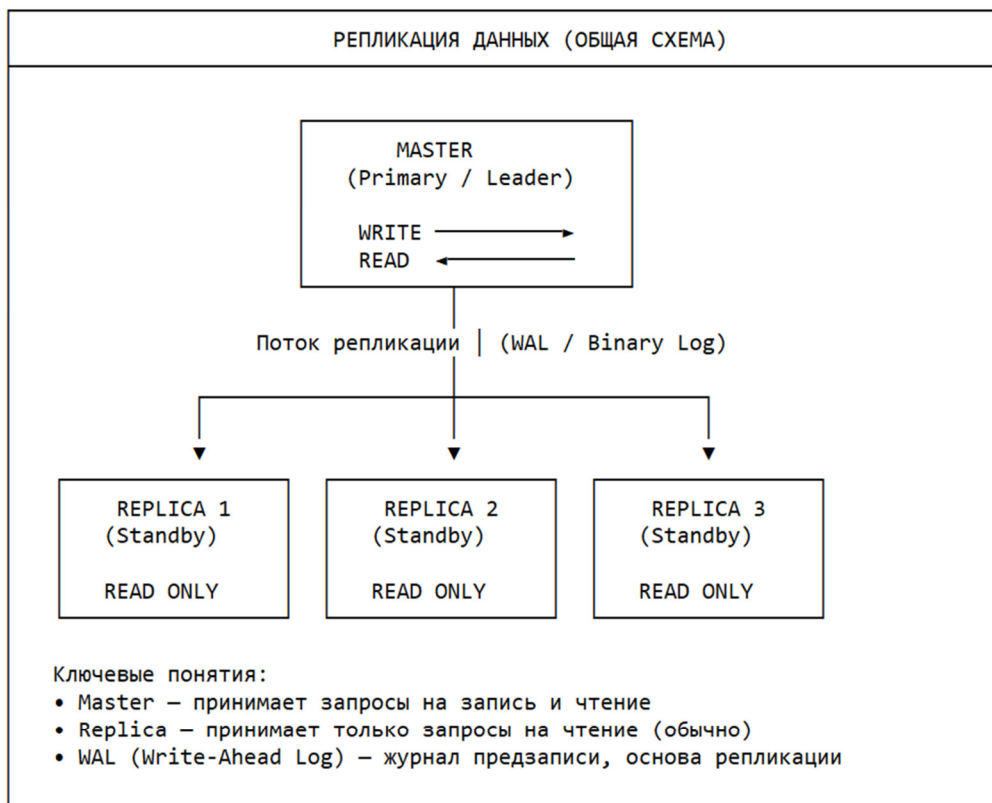
Цель работы: Сформировать практические навыки настройки и анализа различных режимов репликации баз данных (синхронной и асинхронной) на примере PostgreSQL, оценить влияние режимов репликации на производительность (задержка записи) и согласованность данных, а также измерить время переключения при отказе master-узла (failover).

Формируемые компетенции: ПК-5

1. ТЕОРЕТИЧЕСКОЕ ОБОСНОВАНИЕ

1.1. Понятие репликации данных

Репликация данных - это процесс копирования данных с одного сервера базы данных (master/primary) на один или несколько других серверов (replicas/secondaries). Репликация является фундаментальным механизмом для обеспечения отказоустойчивости, повышения доступности и распределения нагрузки чтения.



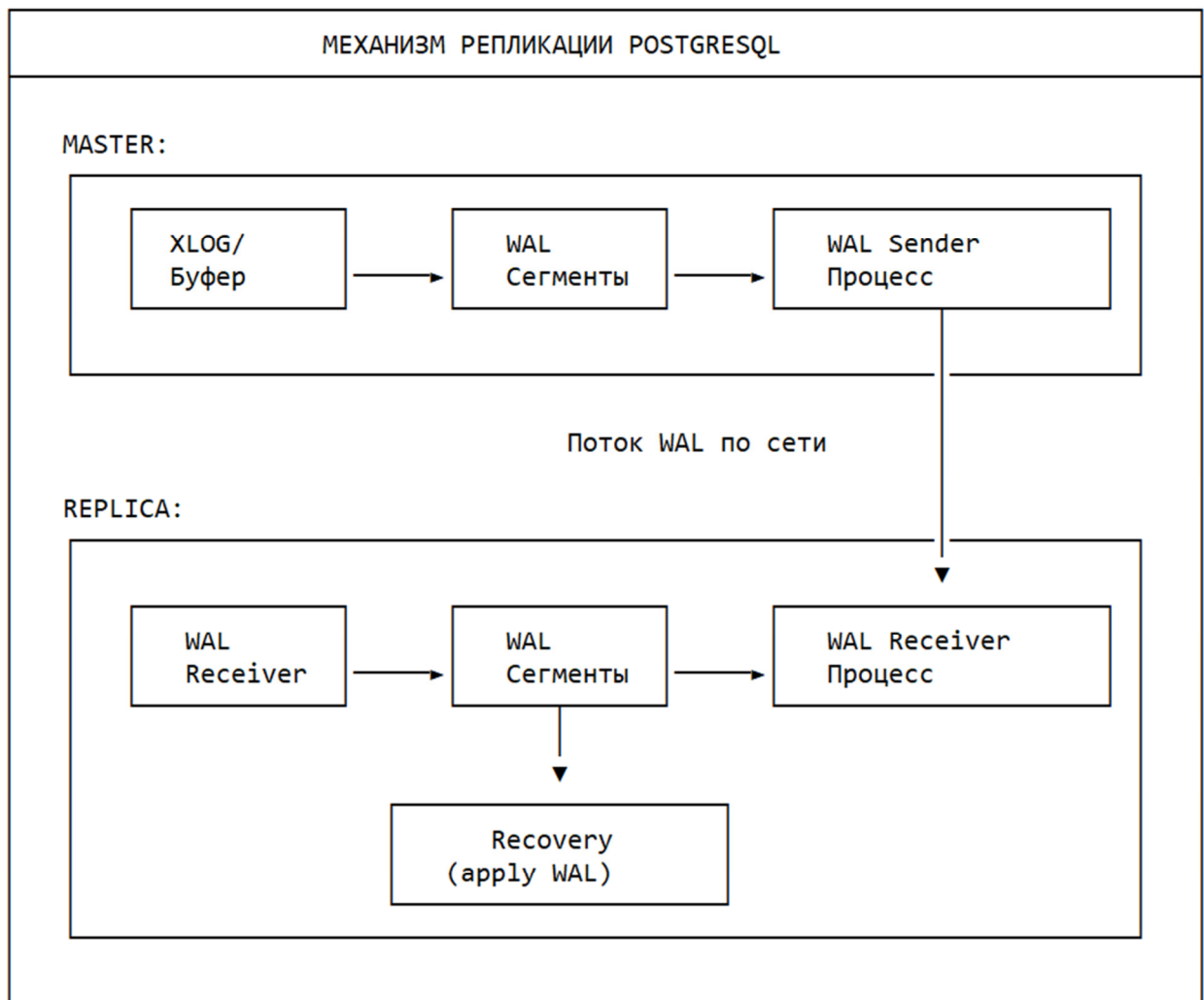
1.2. Синхронная vs асинхронная репликация

Характеристика	Синхронная репликация	Асинхронная репликация
Подтверждение записи	Ждёт подтверждения от реплик	Не ждёт подтверждения
Согласованность	Строгая (данные на репликах всегда актуальны)	Конечная (eventual consistency)
Скорость записи	Медленнее (зависит от сети и скорости реплик)	Быстрее (только запись на master)
Риск потери данных	Минимальный (при отказе master данные на репликах есть)	Есть (неотреплицированные данные могут потеряться)
Доступность записи	Может снижаться при недоступности реплик	Всегда доступна (если master жив)
Задержка (latency)	Выше (RTT до реплик + запись на диск)	Ниже (только local commit)
Когда выбирать	Финансовые	Логи, аналитика,

Характеристика	Синхронная репликация	Асинхронная репликация
	транзакции, критичные данные	некритичные данные

1.3. Архитектура репликации в PostgreSQL

PostgreSQL использует **физическую** репликацию на основе **WAL** (Write-Ahead Log):



Ключевые параметры репликации в PostgreSQL:

Параметр	Назначение	Значение для синхронной	Значение для асинхронной
synchronous_commit	Режим синхронности	on / remote_apply	off / local
synchronous_standby_names	Список синхронных реплик	'replica1,replica2'	" (пустой)
wal_level	Уровень детализации WAL	replica или logical	replica
max_wal_senders	Макс. количество отправителей WAL	10	10
hot_standby	Чтение на реплике	on	on

Уровни synchronous_commit:

Уровень	Описание
off	Асинхронная репликация, не ждёт подтверждения
local	Ждёт только записи в локальный WAL (не ждёт реплик)
on	Ждёт подтверждения записи на диск реплики (fsync)
remote_apply	Ждёт, пока реплика применит изменения (самый

Уровень	Описание
	строгий)

1.4. Поток репликации

Последовательность при синхронной репликации:

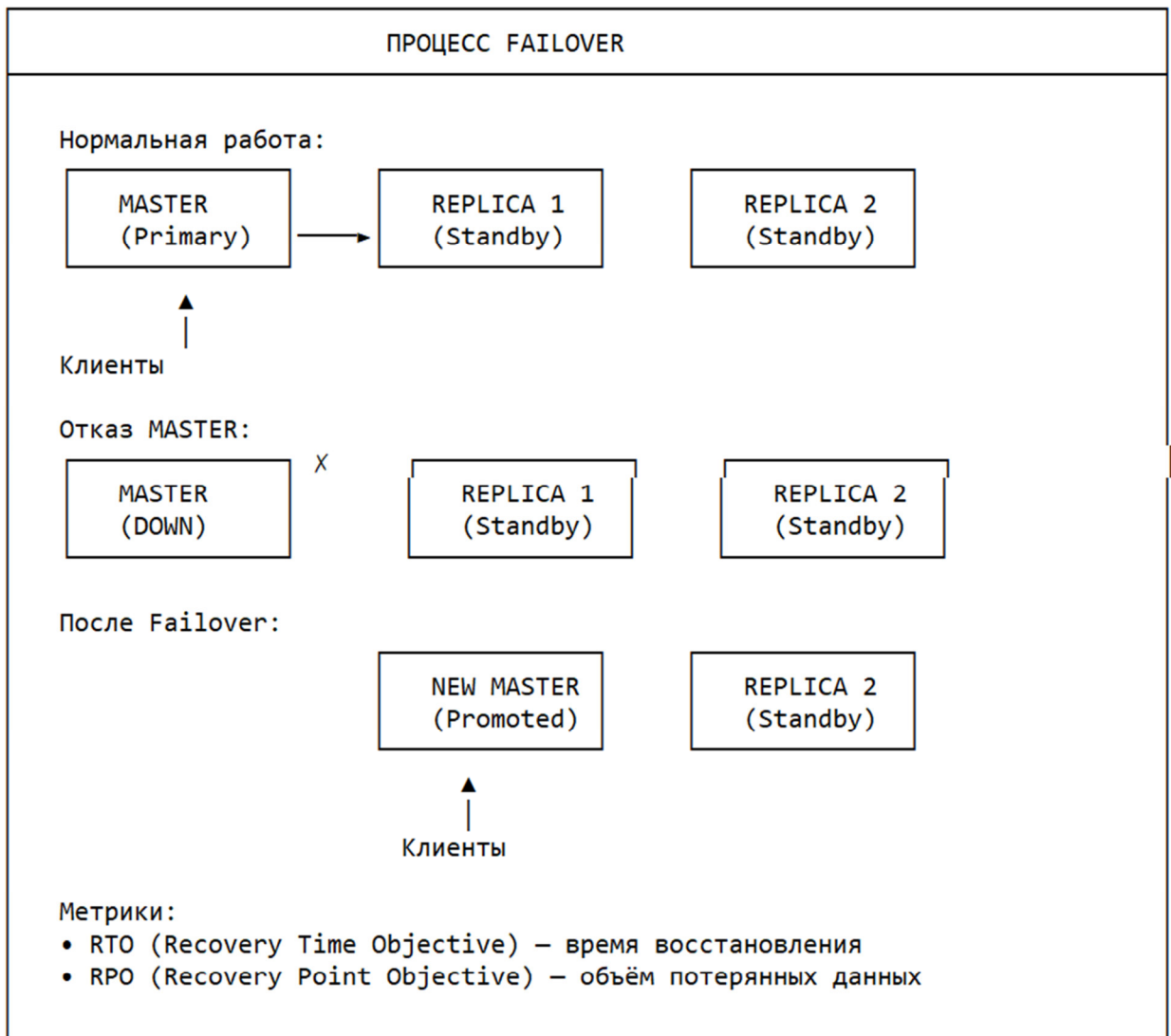
1. Клиент → Запись INSERT в master
2. Master → Запись в WAL (локально)
3. Master → Отправка WAL на синхронные реплики
4. Реплика → Получение WAL и запись на диск
5. Реплика → Подтверждение (ACK) на master
6. Master → Подтверждение клиенту (успех)

Последовательность при асинхронной репликации:

1. Клиент → Запись INSERT в master
2. Master → Запись в WAL (локально)
3. Master → Подтверждение клиенту (успех)
4. Master → Отправка WAL на реплики в фоновом режиме

1.5. Failover и переключение при отказе

Failover - процесс переключения с отказавшего master-узла на одну из реплик.



1.6. Инструменты для управления репликацией

Инструмент	Назначение	Особенности
PostgreSQL Streaming Replication	Встроенная репликация	Простая настройка, физическая репликация
Patroni	Автоматический failover + управление	Использует etcd/Consul/ZooKeeper
pgpool-II	Балансировка + репликация	Поддерживает не только PostgreSQL

Инструмент	Назначение	Особенности
repmgr	Управление репликацией	Простой failover
PgBouncer	Пул соединений	Оптимизация подключений

2. ПРИМЕРЫ ПРАКТИЧЕСКОГО ВЫПОЛНЕНИЯ

2.1. Настройка асинхронной репликации PostgreSQL

2.1.1. Структура проекта

```

postgres-replication/
├── docker-compose.yml
├── master/
│   ├── postgresql.conf
│   ├── pg_hba.conf
│   └── init.sql
├── replica1/
│   ├── postgresql.conf
│   └── pg_hba.conf
├── replica2/
│   ├── postgresql.conf
│   └── pg_hba.conf
├── test_async.sh
└── test_sync.sh

```

2.1.2. Docker Compose (docker-compose.yml)

```

version: '3.8'

services:
  master:
    image: postgres:15
    container_name: pg-master
    environment:

```

```

    POSTGRES_USER: testuser
    POSTGRES_PASSWORD: testpass
    POSTGRES_DB: testdb
  ports:
    - "5432:5432"
  volumes:
    - ./master/postgresql.conf:/etc/postgresql/postgresql.conf
    - ./master/pg_hba.conf:/etc/postgresql/pg_hba.conf
    - ./master/init.sql:/docker-entrypoint-initdb.d/init.sql
  command: postgres -c config_file=/etc/postgresql/postgresql.conf

replica1:
  image: postgres:15
  container_name: pg-replica1
  environment:
    POSTGRES_USER: testuser
    POSTGRES_PASSWORD: testpass
    POSTGRES_DB: testdb
  ports:
    - "5433:5432"
  volumes:
    - ./replica1/postgresql.conf:/etc/postgresql/postgresql.conf
    - ./replica1/pg_hba.conf:/etc/postgresql/pg_hba.conf
  depends_on:
    - master
  command: |
    bash -c "
    rm -rf /var/lib/postgresql/data/*
    until pg_basebackup -h master -U replicator -D /var/lib/postgresql/data -R
-P; do
      echo 'Waiting for master...'
      sleep 2
    done
    postgres -c config_file=/etc/postgresql/postgresql.conf
    "

replica2:
  image: postgres:15
  container_name: pg-replica2
  environment:
    POSTGRES_USER: testuser
    POSTGRES_PASSWORD: testpass
    POSTGRES_DB: testdb
  ports:
    - "5434:5432"
  volumes:
    - ./replica2/postgresql.conf:/etc/postgresql/postgresql.conf
    - ./replica2/pg_hba.conf:/etc/postgresql/pg_hba.conf
  depends_on:
    - master
  command: |

```

```

bash -c "
rm -rf /var/lib/postgresql/data/*
until pg_basebackup -h master -U replicator -D /var/lib/postgresql/data -R
-P; do
    echo 'Waiting for master...'
    sleep 2
done
postgres -c config_file=/etc/postgresql/postgresql.conf
"

```

2.1.3. Конфигурация master (postgresql.conf)

```

# Репликация
wal_level = replica
max_wal_senders = 10
wal_keep_size = 64MB
hot_standby = on

# Аутентификация
listen_addresses = '*'

# Синхронность (асинхронная – пустой список)
synchronous_commit = off
synchronous_standby_names = ''

```

2.1.4. pg_hba.conf master

#	TYPE	DATABASE	USER	ADDRESS	METHOD
host	replication		replicator	172.16.0.0/12	md5
host	all		all	0.0.0.0/0	md5

2.1.5. Конфигурация replica (postgresql.conf)

```

# Режим реплики
hot_standby = on
primary_conninfo = 'host=master port=5432 user=replicator password=replpass'

```

2.1.6. Тестирование асинхронной репликации (test_async.sh)

```

#!/bin/bash

echo "=== Тестирование АСИНХРОННОЙ репликации ==="

# Запись на master

```

```

echo "Запись данных на master..."
docker exec pg-master psql -U testuser -d testdb -c "
CREATE TABLE IF NOT EXISTS test_repl (
    id SERIAL PRIMARY KEY,
    value TEXT,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
INSERT INTO test_repl (value) VALUES ('Async test 1');
INSERT INTO test_repl (value) VALUES ('Async test 2');
"

# Мгновенная проверка на реплике (может не увидеть данные)
echo "НЕМЕДЛЕННОЕ чтение с replica1 (может не увидеть данные)..."
docker exec pg-replica1 psql -U testuser -d testdb -c "SELECT * FROM test_repl;"

echo ""
echo "Ожидание 2 секунды (для асинхронной репликации)..."
sleep 2

echo "ПОВТОРНОЕ чтение с replica1 (должно увидеть данные)..."
docker exec pg-replica1 psql -U testuser -d testdb -c "SELECT * FROM test_repl;"

# Измерение задержки записи
echo ""
echo "Измерение задержки записи (асинхронная)..."
time docker exec pg-master psql -U testuser -d testdb -c "INSERT INTO test_repl
(value) VALUES ('Performance test');"

```

2.2. Настройка синхронной репликации PostgreSQL

2.2.1. Изменение конфигурации master для синхронной репликации

```

# Синхронная репликация
synchronous_commit = on
synchronous_standby_names = 'FIRST 1 (replica1, replica2)'
# или для всех: synchronous_standby_names = '*'

```

2.2.2. Тестирование синхронной репликации (test_sync.sh)

```

#!/bin/bash

echo "=== Тестирование СИНХРОННОЙ репликации ==="

# Проверка статуса репликации
echo "Статус репликации:"
docker exec pg-master psql -U testuser -d testdb -c "
SELECT application_name, state, sync_state, sync_priority
FROM pg_stat_replication;
"

```

```

# Измерение задержки записи
echo ""
echo "Измерение задержки записи (синхронная)..."
time docker exec pg-master psql -U testuser -d testdb -c "
INSERT INTO test_repl (value) VALUES ('Sync performance test');
"

# Эмуляция отказа реплики
echo ""
echo "Эмуляция отказа replica1 (остановка контейнера)..."
docker stop pg-replica1

echo "Попытка записи при недоступной реплике..."
timeout 5 docker exec pg-master psql -U testuser -d testdb -c "
INSERT INTO test_repl (value) VALUES ('Test during fail');
"

if [ $? -eq 124 ]; then
    echo "ЗАПИСЬ ЗАБЛОКИРОВАНА (ожидаемое поведение синхронной репликации)"
fi

echo ""
echo "Восстановление replica1..."
docker start pg-replica1
sleep 5

echo "Проверка, что данные синхронизировались..."
docker exec pg-replica1 psql -U testuser -d testdb -c "SELECT COUNT(*) FROM
test_repl;"

```

2.2.3. Измерение RPO и RTO при failover

```

#!/bin/bash
# measure_failover.sh

echo "=== Измерение RPO и RTO при failover ==="

# Запись данных перед failover
docker exec pg-master psql -U testuser -d testdb -c "
INSERT INTO test_repl (value, created_at) VALUES ('Before failover', NOW());
"

# Остановка master
echo "Эмуляция отказа master..."
docker stop pg-master

# Засекаем время начала failover
START_TIME=$(date +%s%N)

# Промоутируем replica1 в master

```

```

echo "Повышение replica1 до master..."
docker exec pg-replica1 bash -c "
rm -f /var/lib/postgresql/data/standby.signal
"

# Перезапуск replica1 как master
docker restart pg-replica1
sleep 5

# Засекаем время окончания failover
END_TIME=$(date +%s%N)
RTO_MS=$(( (END_TIME - START_TIME) / 1000000 ))

echo "RTO (время восстановления): ${RTO_MS} мс"

# Проверка потерянных данных (RPO)
docker exec pg-replica1 psql -U testuser -d testdb -c "
SELECT COUNT(*) as lost_records
FROM test_repl
WHERE value LIKE '%Before failover%';
"

echo "RPO = количество записей, потерянных при failover"

```

2.3. Нагрузочное тестирование

```

#!/bin/bash
# benchmark.sh

echo "=== Сравнение производительности синхронной vs асинхронной репликации ==="

# Тест асинхронной (5 параллельных потоков, 1000 записей каждый)
echo "Асинхронная репликация:"
time for i in {1..5}; do
    (for j in {1..1000}; do
        docker exec pg-master psql -U testuser -d testdb -c "INSERT INTO
test_repl (value) VALUES ('async_${i}_${j}');" > /dev/null 2>&1
    done) &
done
wait

# Тест синхронной (переключение конфигурации)
echo "Переключение на синхронный режим..."
docker exec pg-master psql -U testuser -d testdb -c "ALTER SYSTEM SET
synchronous_commit = on;"
docker exec pg-master psql -U testuser -d testdb -c "SELECT pg_reload_conf();"

echo "Синхронная репликация:"
time for i in {1..5}; do
    (for j in {1..1000}; do

```

```
docker exec pg-master psql -U testuser -d testdb -c "INSERT INTO
test_repl (value) VALUES ('sync_${i}_${j}');" > /dev/null 2>&1
done) &
done
wait
```

МЕТОДИЧЕСКИЕ УКАЗАНИЯ ДЛЯ ВЫПОЛНЕНИЯ ПРАКТИЧЕСКОЙ ЧАСТИ РАБОТЫ

Задачи работы:

1. **Развёртывание кластера PostgreSQL:**
 - Настроить master и 2 реплики с помощью Docker Compose
 - Настроить асинхронную репликацию
2. **Эксперимент 1: Асинхронная репликация:**
 - Выполнить запись на master
 - Измерить задержку до появления данных на реплике (inconsistency window)
 - Оценить скорость записи (INSERT в секунду)
3. **Эксперимент 2: Синхронная репликация:**
 - Переключить синхронный режим (synchronous_commit = on)
 - Сравнить скорость записи с асинхронным режимом
 - Пронаблюдать блокировку записи при отказе реплики
4. **Эксперимент 3: Failover:**
 - Остановить master
 - Повысить одну из реплик до master
 - Измерить RTO (время восстановления) и RPO (потерянные данные)
5. **Оформление отчёта:**
 - Таблица сравнения синхронной и асинхронной репликации
 - Графики зависимости времени записи от режима
 - Выводы о выборе режима для различных сценариев

3. ВАРИАНТЫ ЗАДАНИЙ

3.1. Базовый уровень (15 вариантов)

Требования ко всем вариантам базового уровня:

- Развернуть PostgreSQL master + 1 реплика
- Настроить асинхронную репликацию
- Измерить время записи и задержку появления данных на реплике

№	Тип нагрузки	Кол-во записей	Измеряемая метрика
1	Последовательная	100	Время записи, задержка репликации
2	Последовательная	1000	Время записи, задержка репликации
3	Последовательная	5000	Время записи, задержка репликации
4	Параллельная (2 потока)	100	Время записи, задержка репликации
5	Параллельная (2 потока)	1000	Время записи, задержка репликации
6	Параллельная (5 потоков)	100	Время записи, задержка репликации
7	Параллельная (5 потоков)	1000	Время записи, задержка репликации
8	Batch INSERT (10 записей/пакет)	1000	Время записи, задержка репликации
9	Batch	1000	Время записи,

№	Тип нагрузки	Кол-во записей	Измеряемая метрика
	INSERT (50 записей/пакет)		задержка репликации
10	Batch INSERT (100 записей/пакет)	1000	Время записи, задержка репликации
11	Крупные записи (TEXT 1KB)	500	Время записи, задержка репликации
12	Крупные записи (TEXT 10KB)	500	Время записи, задержка репликации
13	Крупные записи (TEXT 100KB)	500	Время записи, задержка репликации
14	UPDATE (изменение существующих)	500	Время записи, задержка репликации
15	Mixed (INSERT + UPDATE)	500	Время записи, задержка репликации

3.2. Средний уровень (15 вариантов)

Требования ко всем вариантам среднего уровня:

1. Всё из базового + дополнительно:

- а. Настроить синхронную репликацию
(параметр `synchronous_commit = on`)
- б. Сравнить производительность синхронной и асинхронной репликации
- с. Измерить RTO и RPO при failover

	Количество реплик	Уровень <code>synchronous_commit</code>	Сценарий failover
	1	on	Ручной (promote replica)
	1	remote_apply	Ручной (promote replica)
	2	on (FIRST 1)	Ручной (promote replica1)
	2	on (FIRST 2)	Ручной (promote replica1)
	2	remote_apply (FIRST 1)	Ручной (promote replica2)
	3	on (QUORUM 2)	Ручной (promote replica1)
	3	remote_apply (QUORUM 2)	Ручной (promote replica1)
	1	on	Автоматический (Patroni)
	2	on (FIRST 1)	Автоматический (Patroni)

	Количество реплик	Уровень synchronous_commit	Сценарий failover
0	2	remote_apply (FIRST 1)	Автоматический (Patroni)
1	3	on (QUORUM 2)	Автоматический (Patroni)
2	1	on	С восстановлением старого master
3	2	on (FIRST 1)	С восстановлением старого master
4	2	remote_apply	С восстановлением старого master
5	3	on (QUORUM 2)	С восстановлением старого master

3.3. Продвинутый уровень (15 вариантов)

Требования ко всем вариантам продвинутого уровня:

- Всё из среднего + дополнительно:
 - Использовать Patroni для автоматического failover
 - Интегрировать etcd/Consul для хранения состояния кластера
 - Измерить RTO и RPO в разных сценариях отказов
 - Сравнить с ручным failover

№	Инструмент	Хранилище состояния	Сценарий отказа	Доп. метрики
1	Patroni	etcd	Отказ master	RTO, RPO,

№	Инструмент	Хранилище состояния	Сценарий отказа	Доп. метрики
				время переключения
2	Patroni	etcd (3 узла)	Отказ master + сбой сети	RTO, RPO, влияние split- brain
3	Patroni	Consul	Отказ master	RTO, RPO, нагрузка на etcd
4	Patroni	ZooKeeper	Отказ master	RTO, RPO, потребление ресурсов
5	repmgr	БД	Отказ master	RTO, RPO, сравнение с Patroni
6	pgpool-II	—	Отказ master + балансировка	RTO, RPO, сравнение
7	Patroni (Geo)	etcd (multi- DC)	Отказ целого дата-центра	RTO, RPO, гео- распределение
8	Patroni + HAProxy	etcd	Отказ master + сбой реплики	RTO, RPO, нагрузка HAProxy
9	Patroni +	etcd	Отказ master +	RTO, RPO,

№	Инструмент	Хранилище состояния	Сценарий отказа	Доп. метрики
	Keepalived		VIP переключение	время переключения VIP
10	Patroni + DNS	etcd	Отказ master + DNS переключение	RTO, RPO, TTL DNS
11	Stolon	etcd	Отказ master	RTO, RPO, сравнение с Patroni
12	Patroni + синхронная	etcd	Отказ master (синхронный режим)	RTO, RPO, потеря данных
13	Patroni + асинхронная	etcd	Отказ master (асинхронный режим)	RTO, RPO, потеря данных
14	Patroni + каскадная репликация	etcd	Отказ master с каскадной репликой	RTO, RPO, задержка каскада
15	Patroni + бинарная репликация	etcd	Отказ master + бинарная репликация	RTO, RPO, сравнение с физической

4. КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Чем отличается синхронная репликация от асинхронной?
2. Какой параметр в PostgreSQL отвечает за синхронную репликацию?
3. Что такое WAL (Write-Ahead Log) и как он используется в репликации?
4. Какой режим репликации обеспечивает нулевую потерю данных (RPO = 0)?
5. Что происходит с записью в синхронном режиме, если реплика недоступна?
6. Какие метрики RTO и RPO характеризуют?
7. Как измерить задержку репликации (replication lag)?
8. Какие преимущества у асинхронной репликации перед синхронной?
9. Для каких сценариев подходит синхронная репликация?
10. Как повысить реплику до master (ручной failover)?
11. Какие инструменты автоматического failover для PostgreSQL существуют?
12. Что такое «каскадная репликация»?
13. Как влияет количество синхронных реплик на производительность записи?
14. Чем физическая репликация отличается от логической?
15. Как определить, отстаёт ли реплика в PostgreSQL?

5. ТРЕБОВАНИЯ К ОТЧЕТУ

1. Отчёт должен содержать:
2. Титульный лист
3. Цель работы
4. Теоретическая часть: синхронная vs асинхронная репликация, WAL, RTO, RPO
5. Практическая часть:

6. Листинг docker-compose.yml
7. Листинг конфигураций PostgreSQL (master, replica)
8. Команды для настройки репликации
9. Результаты экспериментов:
10. Таблица сравнения времени записи (синхронная vs асинхронная)
11. График зависимости времени записи от количества записей
12. Значения RTO и RPO при failover
13. Анализ:
14. Объяснение, почему синхронная репликация медленнее
15. Объяснение, почему асинхронная может потерять данные
16. Вывод: в каких случаях выбирать синхронную, в каких — асинхронную
17. Ответы на контрольные вопросы

6. КРИТЕРИИ ОЦЕНИВАНИЯ

Критерий	Базовый (10)	Средний (14)	Продвинутый (18)
Кластер PostgreSQL развёрнут	3	2	1
Асинхронная репликация работает	3	2	1
Синхронная репликация работает	—	3	2
Замеры времени записи проведены	2	2	1
Измерена задержка репликации	2	1	1

Критерий	Базовый (10)	Средний (14)	Продвинут ый (18)
RTO/RPO измерены	—	2	2
Patroni автоматический failover	—	—	3
Сравнение ручного vs автоматического	—	—	3
Качество отчёта	—	2	4
Максимум	10	14	18

Шкала перевода в оценку

О ценка	Ба зовый	Ср едний	Продви нутый
5	9- 10	13 -14	16-18
4	7- 8	10 -12	12-15
3	5- 6	7- 9	8-11
2	< 5	<7	<8

Штрафы

Нарушение	Штра ф
-----------	-----------

Нарушение	Штраф
Нет сравнения времени записи (синхр vs асинхр)	-3
Нет измерения RTO/RPO	-3
Репликация не работает	работа не засчитывается

7. ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ

Программа	Назначение	Доступность в РФ
Docker + Docker Compose	Контейнеризация	+
PostgreSQL (официальный образ)	База данных	+
psql	Клиент PostgreSQL	+
Patroni (продвинутый)	Автоматический failover	+
etcd / Consul (продвинутый)	Хранилище состояния	+
Python (опционально)	Автоматизация экспериментов	+

Программа	Назначение	Доступность в РФ
pgbench	Нагрузочное тестирование	+

Лабораторная работа №8

Шардирование данных (горизонтальное партиционирование)

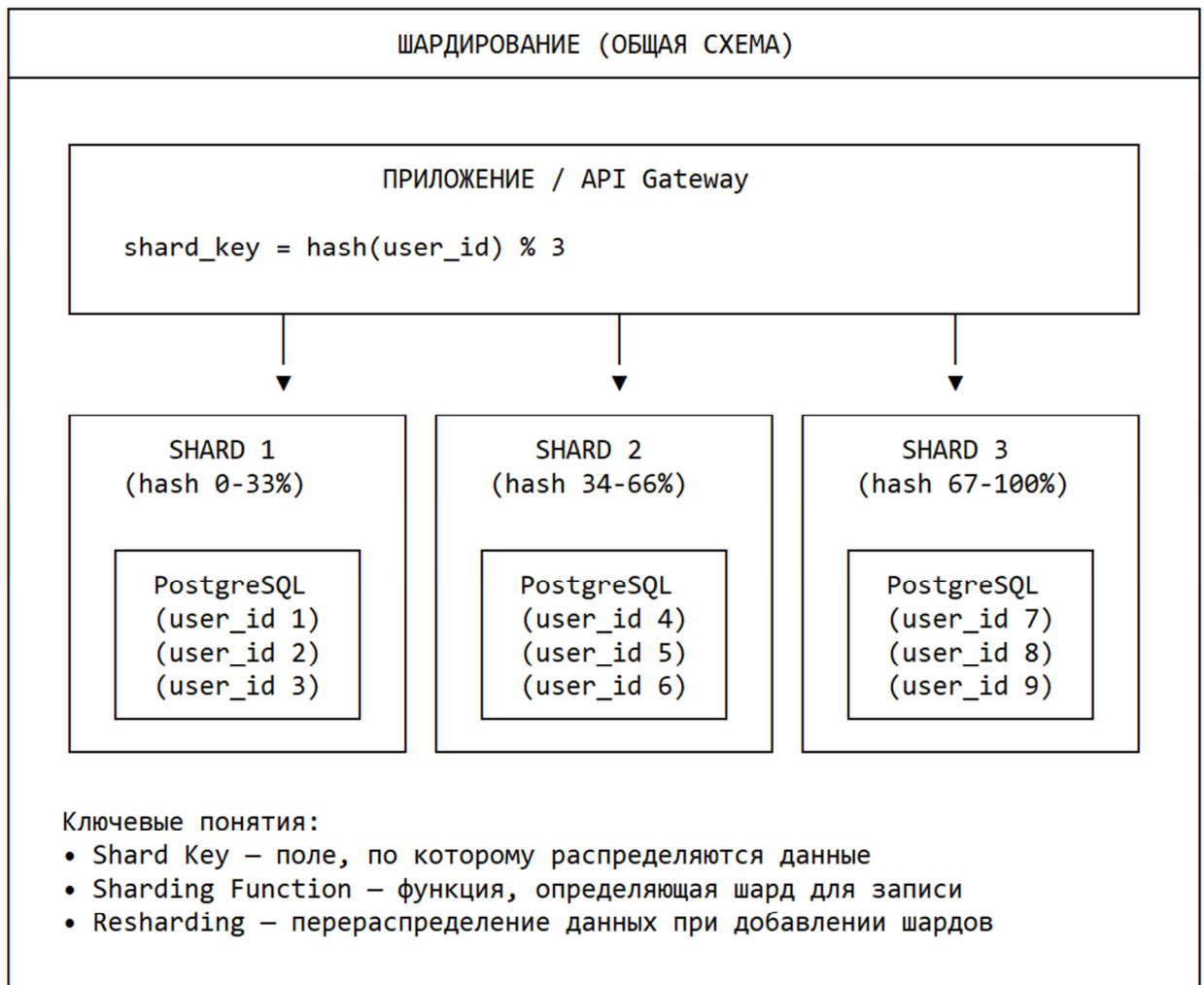
Цель работы: Сформировать практические навыки проектирования и реализации шардирования данных в высоконагруженных системах, включая выбор стратегии шардирования (хеш-шардирование, диапазонное шардирование, consistent hashing), настройку прикладного уровня шардирования и анализ влияния шардирования на производительность запросов.

Формируемые компетенции: ПК-5 (ИД-1, ИД-4, ИД-5, ИД-6, ИД-7)

1. ТЕОРЕТИЧЕСКОЕ ОБОСНОВАНИЕ

1.1. Понятие шардирования

Шардирование (Sharding) - это метод горизонтального масштабирования баз данных, при котором данные распределяются по нескольким независимым серверам (шардам) на основе некоторого ключа (shard key). Каждый шард содержит только часть данных, и все шарды вместе образуют полную логическую базу данных.



1.2. Вертикальное vs горизонтальное масштабирование

Характеристика	Вертикальное масштабирование	Горизонтальное масштабирование (шардирование)
Суть	Увеличение мощности одного сервера	Добавление новых серверов
Пределы	Ограничено физическим железом (max 64-128 core)	Практически не ограничено
Сложность	Простое (аренда bigger instance)	Сложное (логика распределения

Характеристика	Вертикальное масштабирование	Горизонтальное масштабирование (шардирование)
		запросов)
Стоимость	Суперлинейный рост	Линейный рост
Отказоустойчивость	Низкая (SPOF)	Средняя (отказ одного шарда = потеря части данных)
Когда применять	Начальный этап, некритичные системы	Терабайты данных, миллионы RPS

1.3. Стратегии шардирования

1.3.1. Хеш-шардирование (Hash Sharding)

Данные распределяются на основе хеш-значения ключа шардирования.

$\text{shard_id} = \text{hash}(\text{shard_key}) \% \text{number_of_shards}$

Преимущества	Недостатки
Равномерное распределение данных	Сложный решардинг (при изменении количества шардов)
Простая реализация	Невозможны диапазонные запросы
Хорошо подходит для point-запросов	Неравномерность хеш-функции (коллизии)

Пример Python:

```
def get_shard(user_id, num_shards=4):  
    return hash(user_id) % num_shards  
  
# user_id=123 → shard 3  
# user_id=456 → shard 1
```

1.3.2. Диапазонное шардирование (Range Sharding)

Данные распределяются по диапазонам значений ключа шардирования.

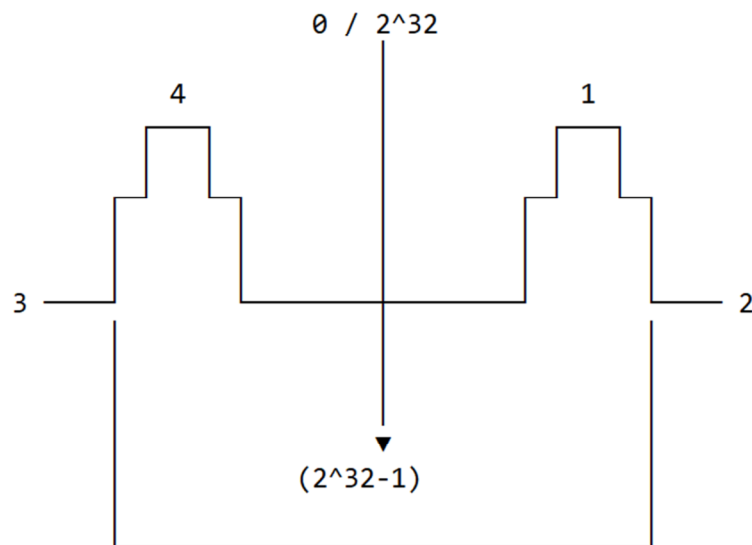
```
Шард 1: user_id 1-1000  
Шард 2: user_id 1001-2000  
Шард 3: user_id 2001-3000
```

Преимущества	Недостатки
Эффективные диапазонные запросы	Возможен «горячий» шард (дисбаланс)
Простой решардинг (сплит шардов)	Требует знания распределения ключей
Естественное упорядочивание	Сложность равномерного распределения

1.3.3. Consistent Hashing (кольцевое хеширование)

Метод, минимизирующий объем перемещаемых данных при изменении количества шардов.

CONSISTENT HASHING (КОЛЬЦЕВОЕ ХЕШИРОВАНИЕ)



- Узлы (шарды) размещаются на кольце
- Данные попадают на первый узел по часовой стрелке
- При добавлении узла перемещается только $1/N$ данных

Пример consistent hashing на Python:

```
import hashlib
import bisect

class ConsistentHash:
    def __init__(self, nodes=None, replicas=150):
        self.replicas = replicas
        self.ring = {}
        self.sorted_keys = []
        if nodes:
            for node in nodes:
                self.add_node(node)

    def _hash(self, key):
        return int(hashlib.md5(key.encode()).hexdigest(), 16)

    def add_node(self, node):
        for i in range(self.replicas):
            virtual_key = f"{node}:{i}"
            hash_val = self._hash(virtual_key)
            self.ring[hash_val] = node
            bisect.insort(self.sorted_keys, hash_val)

    def remove_node(self, node):
        for i in range(self.replicas):
            virtual_key = f"{node}:{i}"
```

```

        hash_val = self._hash(virtual_key)
        del self.ring[hash_val]
        self.sorted_keys.remove(hash_val)

    def get_node(self, key):
        if not self.ring:
            return None
        hash_val = self._hash(key)
        idx = bisect.bisect_right(self.sorted_keys, hash_val) %
len(self.sorted_keys)
        return self.ring[self.sorted_keys[idx]]

# Использование
nodes = ["shard1", "shard2", "shard3"]
ch = ConsistentHash(nodes)
print(ch.get_node("user_123")) # shard2
print(ch.get_node("user_456")) # shard1

```

1.4. Проблемы шардирования

Проблема	Описание	Решение
Неравномерная нагрузка	Один шард получает больше запросов	Consistent hashing, мониторинг, ручной баланс
Перешардирование (resharding)	Миграция данных при добавлении шардов	Consistent hashing (минимизирует перемещения)
Глобальные транзакции	Транзакции, затрагивающие несколько шардов	Saga, eventual consistency, distributed transactions (2PC)
JOIN между шардами	Объединение данных из разных БД	Денормализация, прикладные JOIN, дублирование данных
Глобальные	Автоинкремент не	UUID, Snowflake ID,

Проблема	Описание	Решение
уникальные ID	работает глобально	Redis генератор
Горячие ключи	Один ключ получает непропорционально много запросов	Sharding on secondary key, read replicas

1.5. Архитектуры шардирования

Архитектура	Описание	Инструменты
Application-level sharding	Логика шардирования в приложении	Любое приложение + несколько БД
Database-native sharding	Встроенная поддержка шардирования	MongoDB sharding, Citus (PostgreSQL)
Proxy-based sharding	Прокси-сервер выполняет маршрутизацию	Vitess (MySQL), PgSharding
Middleware sharding	Отдельный слой маршрутизации	ShardingSphere, Jedis (Redis Cluster)

1.6. Сравнение подходов к шардированию

Критерий	Хеш-шардирование	Диапазонное	Consistent Hashing
Равномерность	Высокая	Низкая	Высокая

Критерий	Хеш- шардирование	Диапазонное	Consistent Hashing
		(дисбаланс)	
Диапазонные запросы	+	-	-
Сложность решардинга	Высокая	Средняя	Низкая
Простота реализации	Низкая	Средняя	Высокая
Горячие ключи	+(хеш выравнивает)	-(есть риск)	+(хеш выравнивает)
Популярно сть	Высокая	Средняя	Высокая (Redis Cluster, Cassandra)

2. ПРИМЕРЫ ПРАКТИЧЕСКОГО ВЫПОЛНЕНИЯ

2.1. Application-level шардирование на Python

2.1.1. Структура проекта

app-sharding/

└─ shard_manager.py # Логика шардирования

- |— app.py # FastAPI приложение
- |— config.py # Конфигурация шардов
- |— docker-compose.yml # 3 шарда PostgreSQL
- |— docker-compose.yml
- |— test_sharding.py # Тесты шардирования

2.1.2. Конфигурация шардов (config.py)

```
# Конфигурация подключений к шардам
SHARD_CONFIG = {
    0: {
        "host": "localhost",
        "port": 5432,
        "database": "shard0",
        "user": "testuser",
        "password": "testpass"
    },
    1: {
        "host": "localhost",
        "port": 5433,
        "database": "shard1",
        "user": "testuser",
        "password": "testpass"
    },
    2: {
        "host": "localhost",
        "port": 5434,
        "database": "shard2",
        "user": "testuser",
        "password": "testpass"
    }
}

NUM_SHARDS = 3
```

NUM_SHARDS = 3

2.1.3. Менеджер шардирования (shard_manager.py)

```
import hashlib
from typing import Any, Dict, List, Optional
import psycopg2
from psycopg2.extras import RealDictCursor

class ShardManager:
    def __init__(self, shard_config: Dict[int, Dict], num_shards: int):
        self.shard_config = shard_config
```

```

self.num_shards = num_shards
self.connections = {}

def _get_shard_id(self, shard_key: Any) -> int:
    """Определение шарда на основе хеша ключа"""
    return hash(shard_key) % self.num_shards

def _get_connection(self, shard_id: int):
    """Получение соединения с шардом (с кэшированием)"""
    if shard_id not in self.connections:
        config = self.shard_config[shard_id]
        self.connections[shard_id] = psycopg2.connect(
            host=config["host"],
            port=config["port"],
            database=config["database"],
            user=config["user"],
            password=config["password"],
            cursor_factory=RealDictCursor
        )
    return self.connections[shard_id]

def insert_user(self, user_id: int, name: str, email: str) -> Dict:
    """Вставка пользователя в соответствующий шард"""
    shard_id = self._get_shard_id(user_id)
    conn = self._get_connection(shard_id)

    with conn.cursor() as cur:
        cur.execute("""
            INSERT INTO users (id, name, email, created_at)
            VALUES (%s, %s, %s, NOW())
            RETURNING *
        """, (user_id, name, email))
        conn.commit()
        result = cur.fetchone()

    return {"shard_id": shard_id, **result}

def get_user(self, user_id: int) -> Optional[Dict]:
    """Получение пользователя из шарда"""
    shard_id = self._get_shard_id(user_id)
    conn = self._get_connection(shard_id)

    with conn.cursor() as cur:
        cur.execute("SELECT * FROM users WHERE id = %s", (user_id,))
        result = cur.fetchone()

    if result:
        return {"shard_id": shard_id, **result}
    return None

def get_all_users(self) -> List[Dict]:

```

```

        """Сканирование всех шардов (медленная операция!)"""
        all_users = []
        for shard_id in range(self.num_shards):
            conn = self._get_connection(shard_id)
            with conn.cursor() as cur:
                cur.execute("SELECT * FROM users ORDER BY id")
                users = cur.fetchall()
                for user in users:
                    all_users.append({"shard_id": shard_id, **user})
        return all_users

    def insert_order(self, order_id: int, user_id: int, product: str, amount:
float) -> Dict:
        """Вставка заказа в шард пользователя (co-located)"""
        shard_id = self._get_shard_id(user_id)
        conn = self._get_connection(shard_id)

        with conn.cursor() as cur:
            cur.execute("""
                INSERT INTO orders (id, user_id, product, amount, created_at)
                VALUES (%s, %s, %s, %s, NOW())
                RETURNING *
            """, (order_id, user_id, product, amount))
            conn.commit()
            result = cur.fetchone()

        return {"shard_id": shard_id, **result}

    def get_user_orders(self, user_id: int) -> List[Dict]:
        """Получение заказов пользователя (в одном шарде)"""
        shard_id = self._get_shard_id(user_id)
        conn = self._get_connection(shard_id)

        with conn.cursor() as cur:
            cur.execute("SELECT * FROM orders WHERE user_id = %s", (user_id,))
            return [{"shard_id": shard_id, **row} for row in cur.fetchall()]

```

2.1.4. Docker Compose (docker-compose.yml)

```

version: '3.8'

services:
  shard0:
    image: postgres:15
    container_name: shard0
    environment:
      POSTGRES_USER: testuser
      POSTGRES_PASSWORD: testpass
      POSTGRES_DB: shard0
    ports:

```

```

    - "5432:5432"
  volumes:
    - ./init_shard.sql:/docker-entrypoint-initdb.d/init.sql

shard1:
  image: postgres:15
  container_name: shard1
  environment:
    POSTGRES_USER: testuser
    POSTGRES_PASSWORD: testpass
    POSTGRES_DB: shard1
  ports:
    - "5433:5432"
  volumes:
    - ./init_shard.sql:/docker-entrypoint-initdb.d/init.sql

shard2:
  image: postgres:15
  container_name: shard2
  environment:
    POSTGRES_USER: testuser
    POSTGRES_PASSWORD: testpass
    POSTGRES_DB: shard2
  ports:
    - "5434:5432"
  volumes:
    - ./init_shard.sql:/docker-entrypoint-initdb.d/init.sql

```

2.1.5. Инициализация шардов (init_shard.sql)

```

-- Создание таблицы пользователей
CREATE TABLE IF NOT EXISTS users (
  id INTEGER PRIMARY KEY,
  name VARCHAR(100) NOT NULL,
  email VARCHAR(100) UNIQUE NOT NULL,
  created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

-- Создание таблицы заказов
CREATE TABLE IF NOT EXISTS orders (
  id INTEGER PRIMARY KEY,
  user_id INTEGER NOT NULL,
  product VARCHAR(200) NOT NULL,
  amount DECIMAL(10, 2) NOT NULL,
  created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

-- Индекс для быстрого поиска заказов пользователя
CREATE INDEX idx_orders_user_id ON orders(user_id);

```

```
-- Вставка тестовых данных
INSERT INTO users (id, name, email) VALUES
    (1, 'Alice', 'alice@example.com'),
    (2, 'Bob', 'bob@example.com'),
    (3, 'Charlie', 'charlie@example.com')
ON CONFLICT (id) DO NOTHING;
```

2.1.6. Тестирование шардирования (test_sharding.py)

```
#!/usr/bin/env python3
import time
from shard_manager import ShardManager
from config import SHARD_CONFIG, NUM_SHARDS

def test_sharding():
    print("=== Тестирование шардирования ===\n")

    sm = ShardManager(SHARD_CONFIG, NUM_SHARDS)

    # 1. Вставка пользователей
    print("1. Вставка 100 пользователей...")
    start = time.time()
    for i in range(1, 101):
        sm.insert_user(i, f"User{i}", f"user{i}@example.com")
    insert_time = time.time() - start
    print(f"    Время вставки: {insert_time:.2f} сек\n")

    # 2. Получение пользователя
    print("2. Получение пользователя по ID...")
    user = sm.get_user(42)
    print(f"    Пользователь 42 в шарде {user['shard_id']}: {user['name']}\n")

    # 3. Распределение данных по шардам
    print("3. Распределение данных по шардам:")
    shard_counts = {0: 0, 1: 0, 2: 0}

    for i in range(1, 101):
        shard_id = sm._get_shard_id(i)
        shard_counts[shard_id] += 1

    for shard_id, count in shard_counts.items():
        print(f"    Шард {shard_id}: {count} пользователей")

    # 4. Вставка заказов
    print("\n4. Вставка заказов:")
    start = time.time()
    for user_id in range(1, 101):
        for order_num in range(3):
            sm.insert_order(
                order_id=user_id * 1000 + order_num,
```

```

        user_id=user_id,
        product=f"Product_{order_num}",
        amount=100.0 * order_num
    )
    order_time = time.time() - start
    print(f"    Время вставки 300 заказов: {order_time:.2f} сек")

# 5. Получение заказов пользователя
print("\n5. Заказы пользователя 42:")
orders = sm.get_user_orders(42)
for order in orders:
    print(f"    Шард {order['shard_id']}: Заказ {order['id']} -
{order['product']}")

# 6. Сканирование всех пользователей (медленно)
print("\n6. Сканирование всех пользователей (full scan)...")
start = time.time()
all_users = sm.get_all_users()
scan_time = time.time() - start
print(f"    Найдено {len(all_users)} пользователей за {scan_time:.2f} сек")

print("\n=== Тест завершён ===")

if __name__ == "__main__":
    test_sharding()

```

2.2. MongoDB Native Sharding

2.2.1. Docker Compose для MongoDB Sharding

```

version: '3.8'

services:
  config-server:
    image: mongo:6.0
    container_name: config-server
    command: mongod --configsvr --replSet configReplSet --port 27019
    ports:
      - "27019:27019"

  shard1:
    image: mongo:6.0
    container_name: shard1
    command: mongod --shardsvr --replSet shard1ReplSet --port 27018
    ports:
      - "27018:27018"

  shard2:
    image: mongo:6.0
    container_name: shard2
    command: mongod --shardsvr --replSet shard2ReplSet --port 27018

```

```

ports:
  - "27028:27018"

shard3:
  image: mongo:6.0
  container_name: shard3
  command: mongod --shardsvr --replSet shard3ReplSet --port 27018
  ports:
    - "27038:27018"

mongos:
  image: mongo:6.0
  container_name: mongos
  command: mongos --configdb configReplSet/config-server:27019
  ports:
    - "27017:27017"
  depends_on:
    - config-server
    - shard1
    - shard2
    - shard3

```

2.2.2. Настройка шардирования в MongoDB

```

// Инициализация Config Server Replica Set
rs.initiate({
  _id: "configReplSet",
  members: [{ _id: 0, host: "config-server:27019" }]
})

// Инициализация шардов
rs.initiate({
  _id: "shard1ReplSet",
  members: [{ _id: 0, host: "shard1:27018" }]
})

rs.initiate({
  _id: "shard2ReplSet",
  members: [{ _id: 0, host: "shard2:27018" }]
})

rs.initiate({
  _id: "shard3ReplSet",
  members: [{ _id: 0, host: "shard3:27018" }]
})

// Добавление шардов в кластер
sh.addShard("shard1ReplSet/shard1:27018")
sh.addShard("shard2ReplSet/shard2:27018")
sh.addShard("shard3ReplSet/shard3:27018")

```

```
// Включение шардирования для базы данных
sh.enableSharding("mydb")

// Шардирование коллекции users по хешу _id
sh.shardCollection("mydb.users", { "_id": "hashed" })

// Шардирование коллекции orders по user_id (диапазонное)
sh.shardCollection("mydb.orders", { "user_id": 1 })
```

МЕТОДИЧЕСКИЕ УКАЗАНИЯ ДЛЯ ВЫПОЛНЕНИЯ ПРАКТИЧЕСКОЙ ЧАСТИ РАБОТЫ

Задачи работы:

1. **Проектирование схемы шардирования:**
 - Выбрать ключ шардирования (shard key)
 - Выбрать стратегию (хеш, диапазонное, consistent hashing)
 - Определить количество шардов
2. **Реализация шардирования:**
 - Развернуть несколько экземпляров БД (Docker Compose)
 - Реализовать функцию определения шарда
 - Настроить connection pool для каждого шарда
3. **Эксперимент 1: Равномерность распределения:**
 - Вставить большое количество записей (1000-10000)
 - Проверить равномерность распределения по шардам
 - Визуализировать результаты
4. **Эксперимент 2: Производительность запросов:**
 - Измерить время point-запроса (по шард-ключу)
 - Измерить время запроса без шард-ключа (full scan)
 - Сравнить производительность
5. **Эксперимент 3: Решардинг:**
 - Добавить новый шард
 - Перераспределить данные (миграция)
 - Измерить объём перемещённых данных
6. **Оформление отчёта:** результаты, графики, выводы

3. ВАРИАНТЫ ЗАДАНИЙ

3.1. Базовый уровень (15 вариантов)

Требования: application-level шардирование, 3 шарда, хеш-стратегия

	Сущность	Ключ шардирования	Кол-во записей
	Пользователи	user_id (число)	1000
	Пользователи	email (строка)	1000
	Заказы	order_id	1000
	Заказы	user_id	1000
	Товары	product_id	1000
	Товары	category_id	1000
	Логи	log_id (UUID)	10000
	События	event_timestamp	10000
	Сессии	session_id (строка)	5000
0	Сообщения	chat_id	5000
1	Транзакции	transaction_id	5000

	Сущность	Ключ шардирования	Кол-во записей
2	Комментарии	post_id	5000
3	Лайки	user_id	10000
4	Аналитика	device_id	10000
5	Телеметрия	sensor_id	10000

3.2. Средний уровень (15 вариантов)

Требования: как базовый + consistent hashing + измерение равномерности

№	Решардинг (добавление шарда)	Метрики для анализа
1	3 → 4 шарда	Процент перемещённых данных
2	3 → 5 шардов	Процент перемещённых данных
3	3 → 6 шардов	Процент перемещённых данных
4	3 → 4 шарда	Время решардинга
5	3 → 5 шардов	Время решардинга
6	3 → 6 шардов	Время решардинга

№	Решардинг (добавление шарда)	Метрики для анализа
7	Хеш vs Consistent Hashing	Сравнение перемещений
8	Виртуальные узлы (100)	Равномерность распределения
9	Виртуальные узлы (200)	Равномерность распределения
10	Виртуальные узлы (500)	Равномерность распределения
11	Диапазонное vs Consistent	Равномерность + запросы
12	Хеш vs Consistent (решардинг 3→5)	Объём миграций
13	Consistent (replicas=50) vs (150)	Распределение и память
14	Consistent + веса узлов	Распределение с учётом мощности
15	Consistent + динамические веса	Адаптивное распределение

3.3. Продвинутый уровень (15 вариантов)

Требования: средний уровень + MongoDB Native Sharding или Citus +
кросс-шардовые запросы

	Платформ а	Топо логия	Кросс-шардовые запросы
	MongoDB	3 шарда	JOIN двух коллекций
	MongoDB	4 шарда	Агрегация по всем шардам
	MongoDB	5 шарда	Диапазонный запрос
	Citus (PostgreSQL)	3 шарда	JOIN с reference table
	Citus (PostgreSQL)	4 шарда	DISTINCT по всем шардам
	Citus (PostgreSQL)	5 шарда	GROUP BY + агрегация
	Vitess (MySQL)	3 шарда	JOIN между шардами
	Vitess (MySQL)	4 шарда	Транзакция на нескольких шардах
	Vitess (MySQL)	5 шарда	Решардинг онлайн
	ClickHouse	3	Распределённая агрегация

	Платформ а	Топо логия	Кросс-шардовые запросы
0		шарда	
1	ClickHouse	4 шарда	Аналитика over все шарды
2	Cassandra	3 узла	QUORUM запросы
3	Cassandra	4 узла	Диапазонный запрос (secondary index)
4	Redis Cluster	3 мастера	Кросс-слотовые запросы (ошибки)
5	Redis Cluster	5 мастеров	Миграция слотов

4. КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Что такое шардирование и чем оно отличается от репликации?
2. Какие стратегии шардирования существуют?
3. В чём разница между хеш-шардированием и диапазонным шардированием?
4. Что такое consistent hashing и в чём его преимущество?
5. Как выбрать ключ шардирования?
6. Какие проблемы возникают при шардировании?
7. Что такое рещардинг и как его минимизировать?
8. Как выполнить JOIN между шардами?
9. Как обеспечить уникальные ID при шардировании?

10. Что такое «горячий ключ» и как с ним бороться?
11. Как влияет количество шардов на производительность?
12. Встроенное шардирование vs application-level: что лучше?
13. Как протестировать равномерность распределения данных?
14. Что такое "secondary sharding key"?
15. Когда шардирование не нужно (альтернативы)?

5. ТРЕБОВАНИЯ К ОТЧЕТУ

1. **Титульный лист**
2. **Цель работы**
3. **Теоретическая часть:** стратегии шардирования, consistent hashing, проблемы
4. **Практическая часть:**
 - Листинг кода шард-менеджера
 - Листинг docker-compose.yml
 - Команды для работы с шардами
5. **Результаты экспериментов:**
 - Гистограмма распределения данных по шардам
 - Сравнение времени point-запроса и full scan
 - Процент перемещённых данных при решардинге (consistent hashing)
6. **Анализ:** почему равномерность важна, как шардирование влияет на запросы
7. **Вывод**
8. **Ответы на контрольные вопросы**

6. КРИТЕРИИ ОЦЕНИВАНИЯ

Критерий	Базовый (10)	Средний (14)	Продвинутый (18)
----------	-----------------	-----------------	---------------------

Критерий	Базовый (10)	Средний (14)	Продвинутый (18)
Шарды развёрнуты (3 шт)	3	2	1
Шард-менеджер реализован	3	2	1
Равномерность распределения	2	2	1
Point-запрос по ключу работает	2	2	1
Consistent hashing реализован	—	3	2
Измерен процент перемещённых данных	—	3	2
MongoDB/Citus кластер	—	—	3
Кросс-шардовые запросы	—	—	3
Качество отчёта	—	—	4
Максимум	10	14	18

Шкала оценки: 5 -отлично : 90%+, 4 – хорошо : 75-89%, 3-удовлетворительно: 60-74%, 2 – хорошо : <60%

7. ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ

Программа	Назначение	Доступность
Docker + Compose	Контейнеризация шардов	+
PostgreSQL	Реляционная БД	+
MongoDB	Документная БД (native sharding)	+
psycopg2	Драйвер PostgreSQL	+
Python 3	Реализация шард-менеджера	+

Лабораторная работа №9

Асинхронная обработка с очередями сообщений (RabbitMQ)

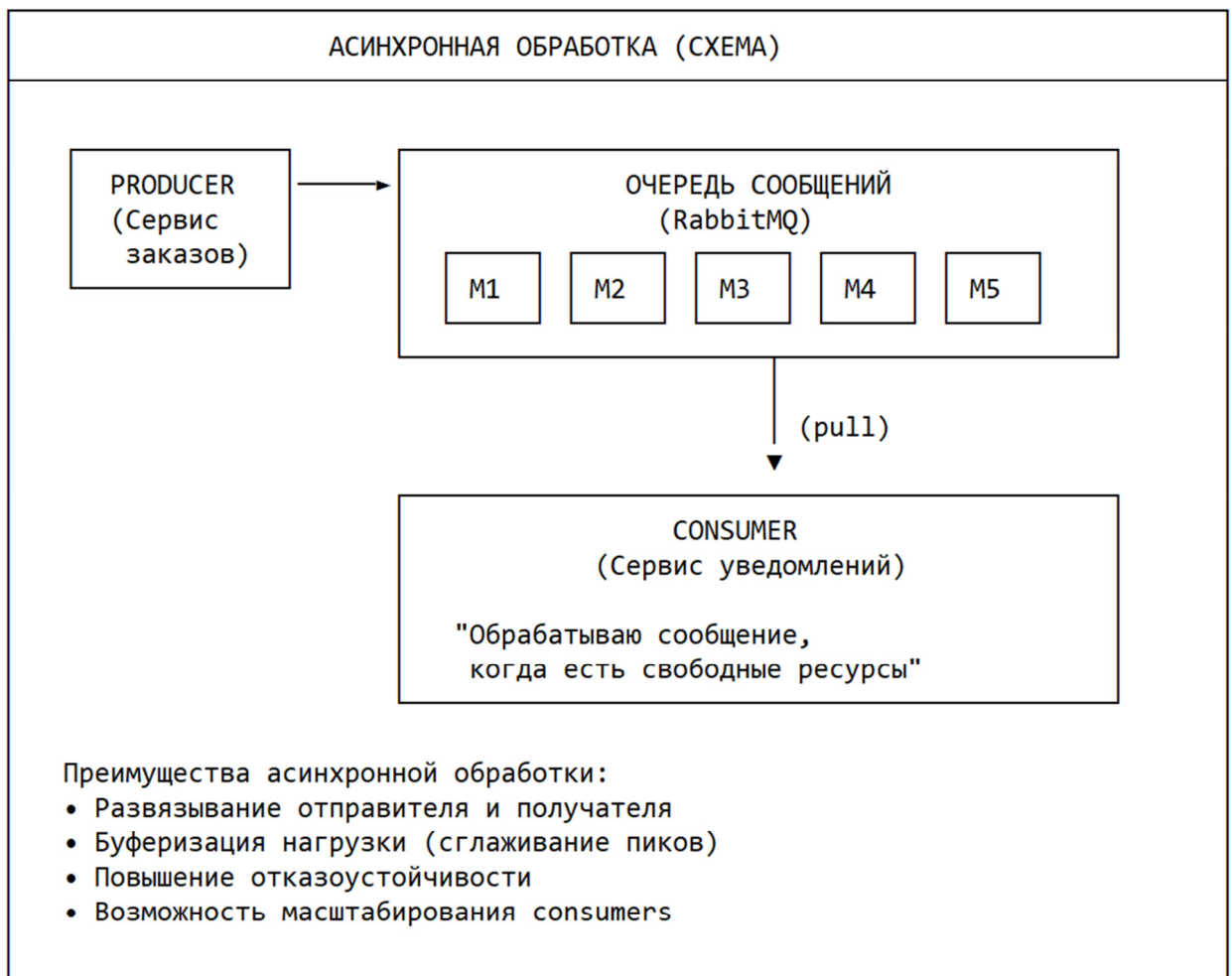
Цель работы: Сформировать практические навыки построения асинхронных систем обмена сообщениями на основе RabbitMQ для развязывания компонентов, повышения отказоустойчивости и масштабируемости высоконагруженных систем, а также освоить паттерны работы с очередями (competing consumers, dead letter queue, publish/subscribe).

Формируемые компетенции: ПК-5

1. ТЕОРЕТИЧЕСКОЕ ОБОСНОВАНИЕ

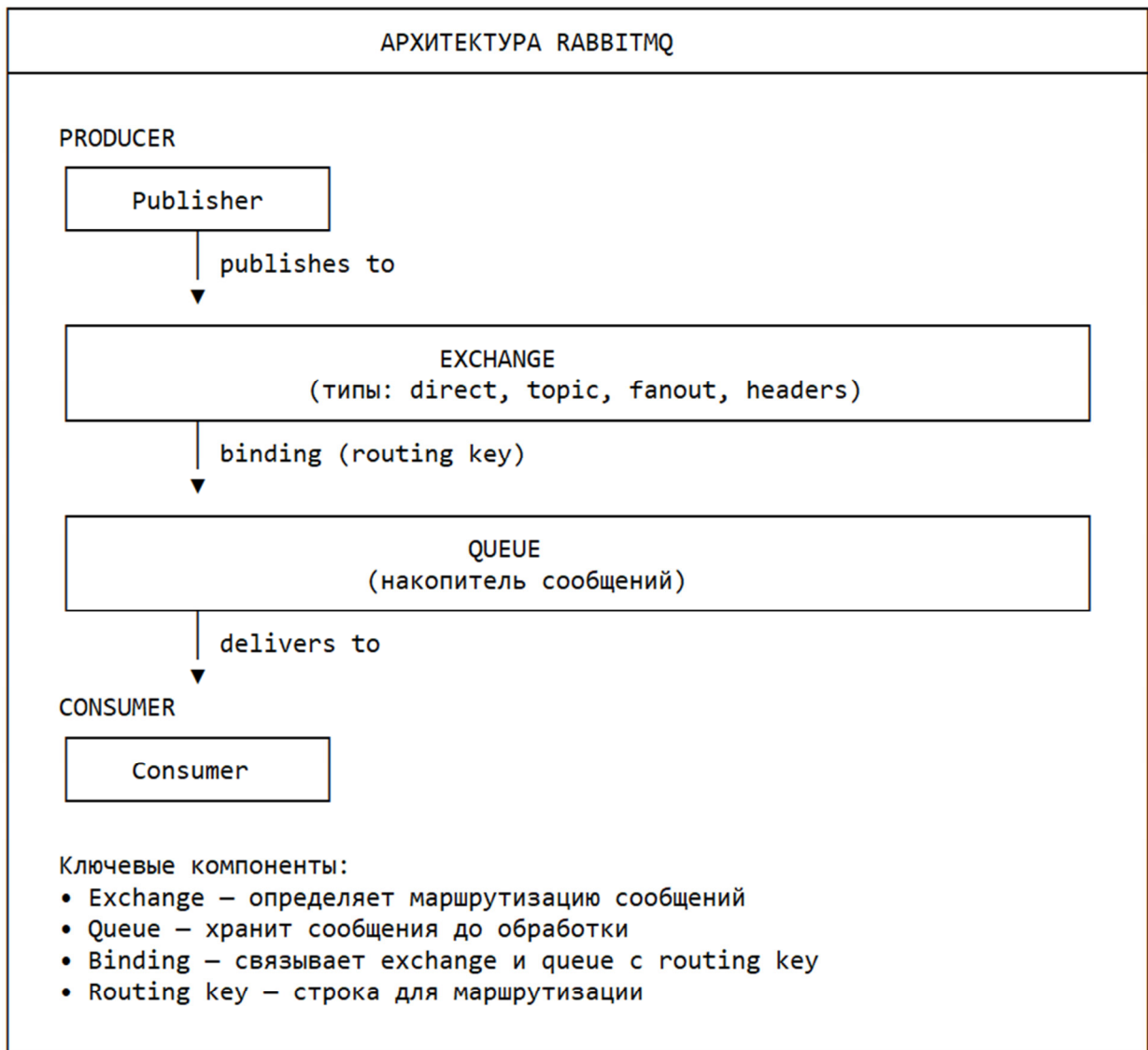
1.1. Понятие асинхронной обработки и очередей сообщений

Асинхронная обработка - это подход, при котором отправитель и получатель сообщения не взаимодействуют напрямую в реальном времени. Сообщение помещается в очередь, а получатель обрабатывает его, когда будет готов. Это позволяет развязать компоненты системы и повысить её отказоустойчивость.



1.2. RabbitMQ: архитектура и ключевые компоненты

RabbitMQ - брокер сообщений, реализующий протокол AMQP 0-9-1. Он выступает посредником между producer и consumer, обеспечивая надёжную доставку сообщений.



1.3. Типы exchanges (обменников)

Тип exchange	Описание	Пример routing key	Куда попадает сообщение
Direct	Точное совпадение routing key	"error"	В очереди, связанные с ключом "error"
Topic	Шаблонное совпадение (wildcards)	"log.error.db"	*.error.*, log.#

Тип exchange	Описание	Пример routing key	Куда попадает сообщение
Fanout	Широковещательная рассылка	игнорируется	Во все связанные очереди
Headers	По заголовкам сообщения (не по routing key)	не используется	По совпадению headers

Примеры маршрутизации:

```

Direct exchange:
    routing_key = "order.created" → очередь с binding "order.created"

Topic exchange:
    routing_key = "europe.orders.created" → binding "*.orders.*" (сработает)
    routing_key = "europe.orders.created" → binding "europe.#" (сработает)

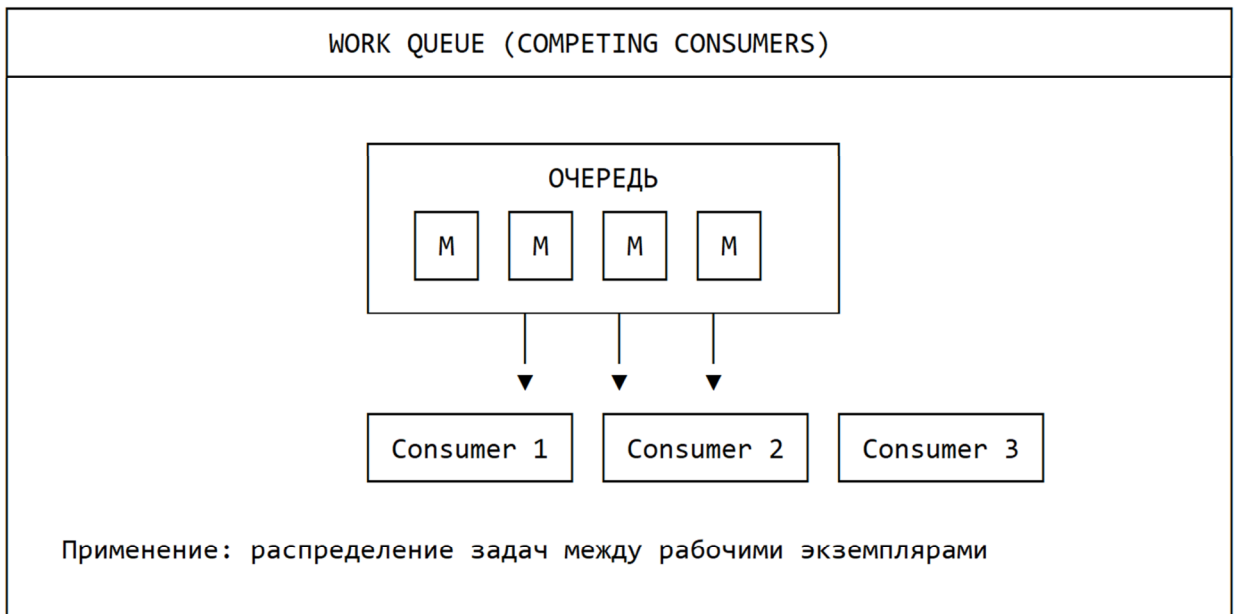
Fanout exchange:
    routing_key любое → → во все связанные очереди

```

1.4. Паттерны работы с очередями

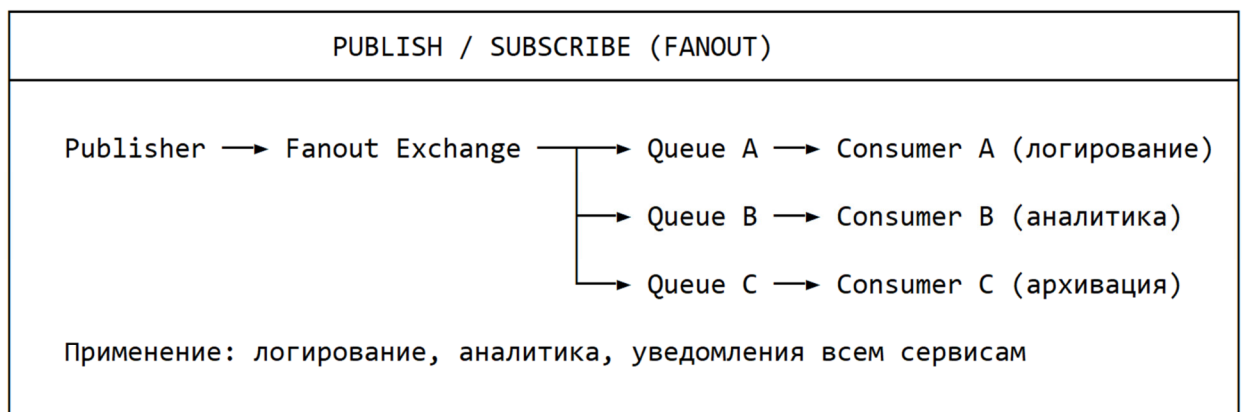
1.4.1. Work Queue (Competing Consumers)

Несколько consumers слушают одну очередь, сообщения распределяются между ними.



1.4.2. Publish/Subscribe (Fanout)

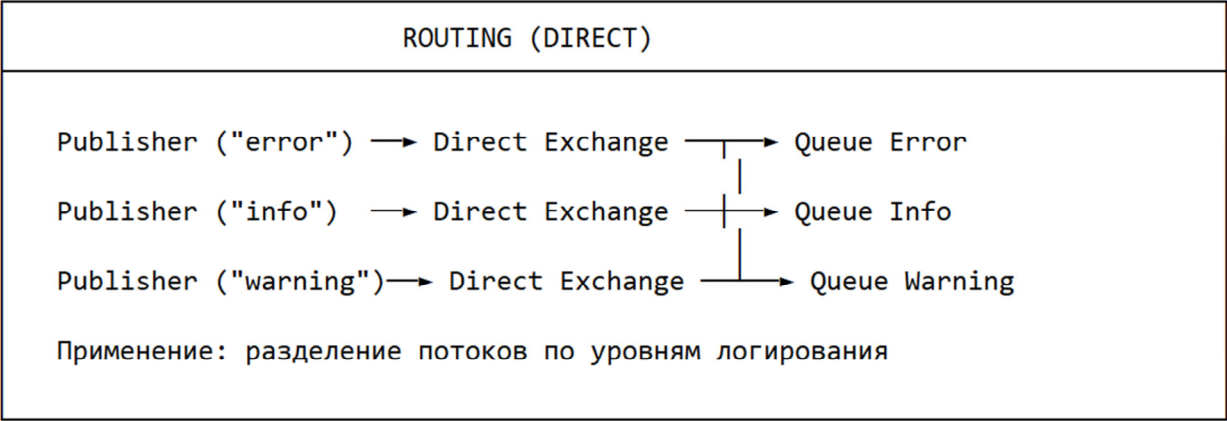
Одно сообщение доставляется всем подписчикам.



1.4.3. Routing (Direct)

Сообщения направляются в очереди на основе точного совпадения routing key.

1.5. Механизмы надёжности RabbitMQ



Механизм	Описание	Пример использования
Publisher Confirms	Publisher ждёт подтверждения от брокера	Важные транзакции (платежи)
Consumer Acks	Consumer подтверждает успешную обработку	После записи в БД
Durable queues	Очередь сохраняется после перезапуска	Все production очереди
Persistent messages	Сообщение сохраняется на диск	Критичные данные
Dead Letter Exchange	Необработанные сообщения → отдельная очередь	Ошибки, требующие анализа
TTL (Time To Live)	Автоматическое удаление старых сообщений	Устаревшие уведомления
Queue TTL	Автоудаление очереди при	Временные

Механизм	Описание	Пример использования
	неактивности	очереди

1.6. Dead Letter Queue (DLQ)

Сообщения попадают в DLQ, если:

- Сообщение отвергнуто consumer'ом (reject/requeue=false)
- Истёк TTL сообщения
- Достигнут лимит длины очереди
- Сообщение не может быть доставлено

