Оглавление

Предисловие **18**
О чём эта книга 18
Для кого эта книга 18
Как читать эту книгу 19
Чего в книге нет 19
Благодарности 20
Введение. System Design как дисциплина **21**
Что такое System Design (и чем он не является) 21
Зачем нужен System Design: три взгляда 22
Зачем это бизнесу 22
Зачем это архитектору 22
Зачем это разработчику 23
System Design vs Software Architecture vs Enterprise Architecture 23
Краткая история: от монолита до serverless 24
Обзор книги: структура и логика изложения 24
Как выжать из книги максимум 25
Глава 1. Принципы проектирования систем **26**
1.1 Фундаментальные принципы 26
KISS (Keep It Simple, Stupid) 26
YAGNI (You Aren‘t Gonna Need It) 27
DRY (Don‘t Repeat Yourself) 27
Separation of Concerns (разделение ответственности) 27
Encapsulation 28
Principle of Least Astonishment 28
1.2 SOLID для архитектора 28
S: Single Responsibility 28
O: Open-Closed Principle 29
L: Liskov Substitution 29
I: Interface Segregation 29

D: Dependency Inversion 29
1.3 CAP-теорема 30
1.4 PACELC — расширение CAP 30
1.5 Теорема Фальстада (и когда SQL, когда NoSQL) 31
1.6 Закон Амдала 31
1.7 Fallacies of Distributed Computing 32
Итог главы 1 32
Глава 2. Модели данных 33
2.1 Реляционная модель 33
ACID: почему реляционные базы не сдаются 33
Нормальные формы: теория, которую нужно знать 34
Индексы: B-Tree, Hash, GiST, Partial, Covering 34
Транзакции и уровни изоляции 34
2.2 Документная модель 35
2.3 Колоночная модель 35
2.4 Графовая модель 36
2.5 Векторная модель 36
2.6 Сравнение моделей данных (как выбирать) 37
Таблица критериев 37
Примеры выбора 38
Polyglot Persistence (полиглот персистенции) 38
Итог главы 2 38
Глава 3. Согласованность и репликация 40
Часть 1. Модели согласованности 40
1.1 Линеаризуемость (Strong Consistency / Linearizability) 40
1.2 Последовательная согласованность (Sequential Consistency) 41
1.3 Причинная согласованность (Causal Consistency) 41
1.4 Read-your-writes (чтение своих записей) 42
1.5 Монотонные чтения (Monotonic Reads) 43
1.6 Итоговая согласованность (Eventual Consistency) 43

1.7 Сравнительная таблица моделей согласованности 44
Часть 2. Репликация 45
2.1 Single-Leader (мастер-слейв) 45
2.2 Multi-Leader (многолидерная) 46
2.3 Quorum-based 47
2.4 Топологии репликации 47
Часть 3. Теорема CAP на практике 47
Реальные примеры простыми словами 48
Часть 4. Конфликты и их разрешение 48
Методы разрешения (от простого к сложному) 48
Итог главы 3 48
Глава 4. Масштабирование 50
Часть 1. Два пути масштабирования 50
1.1 Вертикальное масштабирование (scaling up) 50
1.2 Горизонтальное масштабирование (scaling out) 51
1.3 Сравнительная таблица 52
Часть 2. Шардирование (горизонтальное партиционирование) 52
2.1 Алгоритмы шардирования 53
2.2 Проблема горячих точек 53
2.3 Перешардирование (resharding) 53
2.4 Шардирование на практике 54
Часть 3. Авто-масштабирование 54
Итог главы 4 55
Глава 5. Отказоустойчивость 56
Часть 1. Паттерны устойчивости (Resilience Patterns) 56
1.1 Timeout (таймаут) 56
1.2 Retry (повтор) 57
1.3 Circuit Breaker (предохранитель) 57
1.4 Bulkhead (перегородка) 58
1.5 Fallback (запасной план) 58

1.6 Graceful degradation (мягкая деградация) 59
Часть 2. Избыточность (Redundancy) 59
2.1 Модели избыточности 59
2.2 N+1 59
Часть 3. Асинхронность как способ выжить 59
Часть 4. Chaos Engineering 60
Итог главы 5 60
Глава 6. Наблюдаемость (Observability) 62
Часть 1. Логирование (Logging) 62
1.1 Уровни логирования 62
1.2 Структурированное логирование 63
1.3 Correlation ID 64
1.4 Централизованное хранение логов (ELK Stack) 64
Часть 2. Метрики (Metrics) 65
2.1 Виды метрик 65
2.2 RED и USE — два шаблона метрик 65
2.3 Prometheus 65
Часть 3. Трейсинг (Tracing) 66
3.1 OpenTelemetry 66
3.2 Инструментация без вторжения в код 66
Часть 4. Дашборды и алерты 67
4.1 SLI, SLO, SLA 67
4.2 Error budget 67
4.3 Алерты 67
Итог главы 6 68
Глава 7. Интеграционные паттерны 69
Часть 1. Messaging (очереди и брокеры) 69
1.1 Point-to-Point (очередь) 69
1.2 Publish-Subscribe (топик) 69
1.3 Kafka vs RabbitMQ 69

Часть 2. Паттерны брокеров сообщений 70
2.1 Competing Consumers (конкурирующие потребители) 70
2.2 Dead Letter Channel (почта мёртвых писем) 70
2.3 Poison Message (ядовитое сообщение) 71
2.4 Idempotent Receiver (идемпотентный получатель) 71
2.5 Idempotent Producer (идемпотентный отправитель) 71
Часть 3. Транзакционность в сообщениях 71
3.1 2PC (Two-Phase Commit, двухфазный коммит) 71
3.2 Saga (паттерн для распределённых транзакций) 71
3.3 Outbox + Polling 72
Часть 4. Маршрутизация и трансформация 72
4.1 Маршрутизация сообщений 72
4.2 Трансформация сообщений 72
Часть 5. Паттерны для API 72
5.1 API Gateway 72
5.2 BFF (Backend for Frontend) 73
5.3 Gateway Aggregation vs Offloading 73
Часть 6. CDC (Change Data Capture) 73
Итог главы 7 73
Глава 8. Архитектурные стили 75
Часть 1. Монолит 75
1.1 Что такое монолит 75
1.2 Когда монолит — правильный выбор 76
1.3 Плюсы монолита 76
1.4 Минусы монолита 77
1.5 История: монолит, который работал годами 77
Часть 2. Микросервисы 77
2.1 Что такое микросервисы 77
2.2 Когда микросервисы нужны 78
2.3 Плюсы микросервисов 78

2.4 Минусы микросервисов 78
2.5 История: микросервисы, которые спасли проект 78
Часть 3. Стратегия миграции: Strangler Pattern 79
Часть 4. Модульный монолит (золотая середина) 79
4.1 Что это 79
4.2 Плюсы модульного монолита 79
Часть 5. EDA (Event-Driven Architecture) 80
5.1 Что это 80
5.2 Когда выбирать EDA 80
5.3 Гарантии доставки 80
Часть 6. Лямбда-архитектура и каппа-архитектура 81
6.1 Лямбда-архитектура 81
6.2 Каппа-архитектура 81
Часть 7. Событийно-ориентированная архитектура популярно и сравнение стилей 81
Итог главы 8 82
Глава 9. Паттерны масштабирования 84
Часть 1. Кэширование 84
1.1 Виды кэшей 84
1.2 Стратегии кэширования 85
1.3 Вытеснение кэша (Eviction Policy) 86
1.4 Инвалидация кэша (проклятие распределённых систем) 86
1.5 Проблемы кэширования 87
Часть 2. Очереди (Queues) 87
2.1 Зачем ставить очередь 87
2.2 Rate Limiting на потребителе 87
2.3 Dead Letter Queue (DLQ) 87
Часть 3. Балансировка нагрузки (Load Balancing) 88
3.1 Алгоритмы балансировки 88
3.2 Где размещать балансировку 88

3.3 Липкие сессии (Sticky Sessions) ... 88
3.4 Балансировка с учётом загруженности (Backpressure) ... 89
Часть 4. Защита от перегрузок (Rate Limiting и Throttling) ... 89
4.1 Rate Limiting ... 89
4.2 Где размещать Rate Limiting ... 89
4.3 Throttling ... 90
Итог главы 9 ... 90
Глава 10. Паттерны доступности и отказоустойчивости ... 91
Часть 1. Репликация (продолжение) ... 91
1.1 Синхронная репликация ... 91
1.2 Асинхронная репликация ... 91
1.3 Кворумная репликация ... 92
1.4 Цепочная репликация ... 92
1.5 Топологии репликации ... 92
Часть 2. Георазпределение (Geo-Distribution) ... 93
2.1 Зачем распределять систему географически ... 93
2.2 GeoDNS и Anycast ... 93
2.3 Active-Active vs Active-Passive ... 93
2.4 Latency-Based Routing ... 94
2.5 Проблемы георазпределения ... 94
Часть 3. DRP (Disaster Recovery Plan) ... 94
3.1 RPO (Recovery Point Objective) и RTO (Recovery Time Objective) ... 95
3.2 Стратегии бэкапов ... 95
3.3 Место бэкапа (тёплое, горячее, холодное) ... 95
3.4 DRP план: что должно быть в документе ... 95
Итог главы 10 ... 96
Глава 11. Контейнеризация и оркестрация ... 97
Часть 1. Docker ... 97
1.1 Что такое контейнер ... 97
1.2 Dockerfile — как собрать образ ... 98

1.3 Слои образов и наследование 99
1.4 Docker Compose — оркестрация для одного сервера 99
1.5 Сети, volumes, логирование 100
Часть 2. Kubernetes 100
2.1 Архитектура Kubernetes 100
2.2 Основные объекты Kubernetes 100
2.3 Пример манифеста 101
2.4 StatefulSet для баз данных 102
2.5 Операторы для сложных приложений 102
Часть 3. Service Discovery, Конфигурация и CI/CD 102
3.1 Service Discovery 102
3.2 Конфигурация без секретов в образе 102
3.3 CI/CD на стероидах 103
Итог главы 11 103
Глава 12. CI/CD: от коммита до продакшена 104
Часть 1. Основы CI/CD 104
1.1 CI (Continuous Integration) 104
1.2 CD (Continuous Delivery) 104
1.3 CD (Continuous Deployment) 105
Часть 2. CI-пайплайн 105
2.1 Стандартный CI-пайплайн 105
2.2 Параллельный vs последовательный прогон 105
2.3 Кэширование артефактов 106
2.4 Качество кода (SonarQube) 106
Часть 3. CD-пайплайн (доставка и деплой) 106
3.1 Стратегии деплоя 106
3.2 Approval gate (ручное одобрение) 106
3.3 Health check и Smoke test 107
3.4 Откат (Rollback) 107
Часть 4. GitOps 107

4.1 Что это ... 107
4.2 Инструменты GitOps ... 107
4.3 Helm: шаблонизация k8s-манифестов ... 108
4.4 Kustomize ... 108
Итог главы 12 ... 108
Глава 13. Облачные провайдеры ... 109
Часть 1. Основные провайдеры ... 109
1.1 Кто есть кто на рынке ... 109
1.2 Основные типы услуг ... 110
1.3 Ключевые различия AWS, Azure, GCP, Yandex Cloud ... 110
Часть 2. Облачные паттерны ... 111
2.1 Database-as-a-Service vs Self-managed ... 111
2.2 Serverless плюсы и минусы ... 113
2.3 Объектное хранилище (S3) ... 113
Часть 3. Стратегии переноса в облако (6 R) ... 114
3.1 Шесть стратегий модернизации ... 114
3.2 Lift-and-shift: плюсы и минусы ... 114
3.3 Рефакторинг под cloud-native ... 115
Итог главы 13 ... 115
Глава 14. URL Shortener (пример полного цикла) ... 116
Часть 1. Сбор требований ... 116
1.1 Функциональные требования ... 116
1.2 Нефункциональные требования (NFR) ... 117
1.3 Оценка нагрузки (Back of the envelope) ... 118
1.4 Проектирование API ... 119
1.5 Выбор базы данных ... 120
Часть 2. Генерация коротких ключей ... 120
2.1 Base62 инкрементальный ID ... 120
2.2 Хэш (MD5, SHA-1) + Base62 ... 121
2.3 UUID ... 121

2.4 Генерация ключа без коллизий на лету 121
Часть 3. Кэширование 121
3.1 Стратегия кэширования 122
3.2 Размер кэша 122
3.3 Предотвращение проблем 122
Часть 4. Итоговая архитектура 123
Итог главы 14 123
Глава 15. Мессенджер (Telegram / WhatsApp) 125
Часть 1. Сбор требований 125
1.1 Функциональные требования 125
1.2 Нефункциональные требования (NFR) 126
Часть 2. Архитектура данных 127
2.1 Базы данных: профили, переписки, метаданные 127
2.2 Текст сообщений и метаданные 128
Часть 3. Протокол и реальное время 128
3.1 Real-time – синхрон через WebSocket 128
3.2 Асинхронная обработка и очередь 129
Часть 4. Медиафайлы 129
4.1 Отдельное хранилище (Blob Storage) + ленивая загрузка 129
4.2 Кэширование миниатюр 130
Часть 5. Офлайн-сообщения, уведомления и синхронизация устройств 130
5.1 Офлайн-сообщения 130
5.2 Push-уведомления (для мобильных устройств) 130
5.3 Синхронизация нескольких устройств 130
Часть 6. Итоговая архитектура 131
Итог главы 15 131
Глава 16. Новостная лента (Twitter / VK) 133
Часть 1. Требования и нагрузка 133
1.1 Функциональные требования 133
1.2 Оценка нагрузки (Twitter-like) 134

Часть 2. Ядро системы: две модели доставки 135
2.1 Fanout on write (Push) 135
2.2 Fanout on read (Pull) 135
Часть 3. Гибридная модель (реальная жизнь) 136
3.1 Push для «обычных» аккаунтов, Pull для «звёзд» 136
3.2 Кэширование популярных постов 137
Часть 4. Детали реализации 137
4.1 Шардирование по пользователям 137
4.2 Балансировка и очереди 138
4.4 Схема данных 138
4.5 Лента рекомендаций 139
Итог главы 16 139
Глава 17. Платёжная система (Stripe / PayPal) 141
Часть 1. Требования: никакой потери денег 141
1.1 Функциональные требования 141
1.2 Нефункциональные требования (самое важное) 141
Часть 2. Идемпотентность 142
2.1 Проблема 142
2.2 Решение: idempotency_key 142
2.3 Где ещё нужна идемпотентность 142
Часть 3. Распределённые транзакции: паттерн Saga 142
3.1 Проблема: один платеж — много сервисов 142
3.2 Что такое Saga 143
3.3 Saga Orchestration (центральный координатор) 143
3.4 Saga Choreography (хореография, децентрализованная) 143
Часть 4. Двойная бухгалтерия 144
4.1 Принцип 144
4.2 Актуально для переводов 144
4.3 Как хранить в БД 144
4.4 Проверка консистентности 145

Часть 5. Интеграция с банками и эквайрингами 145
5.1 Асинхронные ответы (Webhook) 145
5.2 Вебхуки должны быть идемпотентными 145
5.3 Ретраи и таймауты 146
5.4 Dead Letter Queue 146
Часть 6. Итоговая архитектура 146
Итог главы 17 147
Глава 18. Система дистанционного электронного голосования (ДЭГ) 148
Часть 1. Уникальные требования 148
1.1 Три кита ДЭГ 148
1.2 Дополнительные требования 149
Часть 2. Почему блокчейн? 149
2.1 Блокчейн vs традиционная база 150
2.2 Наше решение: блокчейн для неизменяемости, БД для скорости 150
2.3 Выбор блокчейн-платформы 150
Часть 3. Криптографическая схема: слепая подпись 151
3.1 Аналогия с конвертом 151
3.2 Математика слепой подписи 151
Часть 4. Масштабирование и нагрузка 152
4.1 Оценка нагрузки 152
4.2 Узкие места 152
4.3 Решения 152
Часть 6. Итоговая архитектура 152
Компоненты 153
Поток голоса 153
Итог главы 18 153
Глава 19. Процесс прохождения интервью по System Design 154
Часть 1. Что оценивает интервьюер 154
1.1 Критерии оценки (официальные в FAANG) 154
1.2 Чего не нужно делать 155

Часть 2. Структура интервью (45 минут) 156
2.1 Типичный тайминг 156
2.2 Вопросы, которые нужно задать в начале 157
Часть 3. High-level дизайн 157
3.1 Что рисовать на этом этапе 157
3.2 Открытые вопросы на этом этапе 158
3.3 Пример схемы (для Twitter) 158
Часть 4. Глубокое погружение 158
4.1 Какие темы затрагивает погружение 158
4.2 Как он проверит, плаваете ли вы 158
4.3 Шаблон ответа глубокого уровня 159
Часть 5. Борьба с узкими местами 159
5.1 Типичные проблемные места 159
5.2 Как отвечать на «упал сервис» 159
Итог главы 19 160
Глава 20. Частые вопросы и как на них отвечать 161
Часть 1. Вопросы про архитектуру и компромиссы 161
Вопрос 1. «Почему вы выбрали микросервисы, а не монолит?» 161
Вопрос 2. «Почему вы выбрали базу данных X, а не Y?» 161
Вопрос 3. «Как вы будете масштабировать базу данных, когда данных станет в 100 раз больше?» 162
Вопрос 4. «Почему здесь синхронный вызов, а не асинхронный?» 162
Часть 2. Вопросы про отказы и надёжность 163
Вопрос 5. «Что произойдёт, если этот сервис упадёт?» 163
Вопрос 6. «Как вы будете обеспечивать надежность данных?» 163
Вопрос 7. «Что вы будете делать, если запрос стал тормозить, а пользователи жалуются?» 163
Часть 3. Вопросы про проектирование и выбор технологий 164
Вопрос 8. «Как вы будете выбирать между SQL и NoSQL?» 164
Вопрос 9. «Какие паттерны отказоустойчивости вы знаете?» 164
Вопрос 10. «Как бы вы улучшили эту систему?» 165

Итог главы 20 165
Глава 21. Тренажёр: системы для самостоятельного проектирования..... 166
Часть 1. Система бронирования билетов (Ticketmaster) 166
Контекст 166
Требования 166
Вопросы для самопроверки 167
Короткий разбор (подглядеть после того, как спроектируете сами) 167
Часть 2. Система поиска отелей (Booking / Airbnb) 168
Контекст 168
Требования 168
Вопросы для самопроверки 168
Мой короткий разбор (подглядеть после) 169
Часть 3. Система доставки еды (Delivery Club / Uber Eats) 169
Контекст 169
Требования 169
Вопросы для самопроверки 170
Мой короткий разбор (подглядеть) 170
Часть 4. Система аналитики посещаемости (Google Analytics) 170
Контекст 171
Требования 171
Вопросы для самопроверки 171
Мой короткий разбор 172
Контрольные вопросы после проектирования 172
Итог главы 21 172
Глава 22. Шпаргалки по базам данных, брокерам и кэшированию 174
Часть 1. Выбор базы данных 174
Таблица 1.1. Когда какую БД выбирать 174
Таблица 1.2. Сравнение SQL vs NoSQL 175
Часть 2. Выбор брокера сообщений 176
Таблица 2.1. Kafka vs RabbitMQ vs других 176

Таблица 2.2. Когда что выбирать 177
Часть 3. Стратегии кэширования 178
Таблица 3.1. Когда какую стратегию использовать 178
Таблица 3.2. Стратегии вытеснения (Eviction) 179
Часть 4. Балансировка нагрузки и алгоритмы 179
Таблица 4.1. Алгоритмы балансировки 179
Таблица 4.2. Где размещать балансировку 180
Часть 5. Оценка нагрузки (формулы быстрого доступа) 180
5.1 Оценка хранилища 181
5.2 Оценка пропускной способности сети 181
5.3 Оценка количества серверов 181
5.4 Карманный справочник 181
Бонус: чек-лист перед принятием архитектурного решения 181
Итог главы 22 182
Глава 23. Глоссарий терминов 183
A 183
B 183
C 184
D 185
E 185
F 186
G 186
H 186
I 186
K 187
L 187
M 187
N 188
O 188
P 188

R 189
S 189
T 190
U 191
V 191
W 191
X 191
Итог главы 23 191
Глава 24. Ресурсы для дальнейшего изучения 192
Часть 1. Книги (must read) 192
1.1 Архитектура и System Design 192
1.2 Мышление и процессы 193
Часть 2. YouTube-каналы 194
Часть 3. Telegram-каналы и сообщества 195
Рекомендации по поиску: 195
Критерии хорошего канала: 195
Часть 4. Подкасты 195
Часть 5. Тренажёры и симуляторы 196
Поиск в GitHub: 196
Часть 6. Бесплатные курсы (с отбором) 196
Что я рекомендую: 196
Критерии хорошего курса: 197
Бонус: как не забросить обучение 197
Итог главы 24 198
Глава 25. Заключение: Архитектор — это на всю жизнь (если не сгорите) 199
Часть 1. Пять главных уроков, которые я вынес за 10 лет 199
1. Архитектор — это не должность, а образ мышления 199
2. Идеальных решений не бывает, бывают компромиссы 199
3. Системное мышление важнее любой технологии 200

4. Люди важнее кода в 10 раз 200
5. Вы никогда не будете знать абсолютно всё (и это нормально) 200
Часть 2. Что делать дальше 200
Завтра 200
На месяц 201
На год 201
План быстрого старта для нетерпеливых 201
Часть 3. Мои контакты 201
Часть 4. Приглашение в закрытый клуб и на курс 202
Напутствие 202

Предисловие

Когда меня спрашивают, с чего начать изучение System Design, я обычно отвечаю: «С боли».

Боль — это когда ваша система упала в первый час распродажи, потому что вы не рассчитали нагрузку на базу данных. Боль — это когда вы месяц ищете узкое место, которое можно было предусмотреть на этапе проектирования. Боль — это когда на собеседовании вам задают вопрос «спроектируйте YouTube», а вы не знаете, с какой стороны подойти.

Эта книга родилась из боли. Моей боли, боли моих коллег, боли сотен разработчиков и архитекторов, с которыми я говорил на лекциях, конференциях и в своём Telegram-канале.

Я проектировал системы, которыми пользуется вся страна. «Выборы 2.0» — это не абстрактный кейс из учебника, это система, которая должна работать, когда от неё зависят миллионы людей. В этой книге нет притянутых за уши примеров про «Twitter для котиков». Есть реальные задачи, реальные ограничения и реальные компромиссы, которые мы принимали.

О чём эта книга

Это не сборник готовых ответов для собеседований. Таких книг уже достаточно. Они учат запоминать, а не понимать. Спросите такого кандидата через месяц — и он не вспомнит, почему в URL Shortener нужен именно Base62, а не хэш.

Это не справочник по технологиям. Технологии устаревают. То, что сегодня — Kafka, завтра сменит Pulsar, а послезавтра придумают что-то ещё. Я учу мыслить, а не запоминать.

Это учебник по принятию решений. Каждая глава — это череда компромиссов. Быстро или качественно? Дёшево или масштабируемо? PostgreSQL или ClickHouse? Я не даю единственно правильных ответов — я даю инструменты, чтобы вы находили свои ответы.

Для кого эта книга

Вы разработчик с 2–5 годами опыта. Вы пишете код, разбираетесь в своём стеке, но чувствуете: дальше так продолжаться не может. Вы устали чинить

баги в чужой архитектуре. Вы хотите проектировать системы, которые не падают под нагрузкой.

Вы тимлид или начинающий архитектор. Вам уже приходится принимать технические решения, но вы не всегда уверены, что выбрали правильный путь. Вы хотите понимать, почему одна база данных подходит для аналитики, а другая — для транзакций. Вы хотите аргументированно защищать свои решения перед руководством.

Вы студент IT-специальности. Вы учитесь проектировать системы в теории, но не знаете, как применить это на практике. Эта книга покажет вам real-world кейсы, которые не разбирают в вузах.

Как читать эту книгу

У книги три сценария чтения.

Сценарий 1 — «Быстрый старт для собеседования». Вы готовитесь к интервью, времени в обрез. Читайте части 1, 2, 4 и 5. Теория (CAP, ACID, репликация), паттерны (кэширование, очереди), разборы конкретных систем (URL Shortener, мессенджер) и стратегия прохождения интервью — этого достаточно, чтобы уверенно отвечать на 80% вопросов.

Сценарий 2 — «Системное погружение». Вы хотите не просто пройти собеседование, а стать лучшим архитектором в команде. Читайте всю книгу последовательно. Часть 3 («Платформы и инфраструктура») и Часть 6 («Справочник архитектора») дадут вам глубину, которая отличает эксперта от любителя.

Сценарий 3 — «Справочник на рабочем столе». Вы уже опытный специалист и ищете ответ на конкретный вопрос. Откройте Часть 6 («Cheat Sheets»). Там вы найдёте таблицы сравнения баз данных, брокеров, алгоритмов балансировки. Пользуйтесь как навигатором.

Чего в книге нет

Нет кода на конкретном языке программирования. Потому что System Design — это не про синтаксис, это про паттерны и решения. Если я приведу пример на Java — разработчик на Go скажет «не моё». Я привожу псевдокод и схемы, которые понятны всем.

Нет идеализации микросервисов. Их слишком много рекламируют, слишком мало критикуют. Я расскажу, когда они реально нужны, а когда — дорогое удовольствие без особой выгоды.

Нет воды. Каждая глава — это концентрированный опыт, выжимка из десятков проектов и тысяч часов размышлений.

Благодарности

Спасибо моим студентам из МИИТ, МИСИС и МТИ. Ваши вопросы на лекциях заставляли меня пересматривать, перепроверять и переосмысливать то, что казалось очевидным. Эта книга во многом родилась из диалога с вами.

Спасибо коллегам из Ростелекома, Sportmaster и ФСО. Наши споры у доски с маркерами — лучшее, что случалось с моим профессиональным ростом.

Спасибо подписчикам Telegram-канала «Кухтинов.Архитектура». Ваши реакции, комментарии и критика держат меня в тонусе.

И спасибо тебе, читатель. Ты держишь в руках не просто книгу. Это карта, по которой я прошёл сам. Карта не гарантирует, что ты не споткнёшься, но обещает, что ты не заблудишься.

Поехали.

Артур Кухтинов
Москва, 2026

Введение. System Design как дисциплина

Вы когда-нибудь задумывались, почему один и тот же вопрос «спроектируйте Twitter» на собеседовании может провалить кандидата с 10-летним стажем и вытянуть джуна, который год назад переключился с бухгалтерии?

Потому что System Design — это не про знание технологий. Это про **способность думать**.

Что такое System Design (и чем он не является)

System Design — это процесс принятия архитектурных решений для системы, которая должна работать в условиях реальных ограничений: нагрузка, бюджет, время, команда, технологии.

Звучит сложно. На деле — это череда вопросов:

- Сколько пользователей будет у системы через год?
- Что важнее — чтобы данные всегда были актуальны или чтобы сервис всегда был доступен?
- Как мы будем масштабироваться, когда нагрузка вырастет в 100 раз?
- Сколько это будет стоить?

System Design — это не про «нарисовать красивую схему со стрелочками». Это про **компромиссы**. Вы всегда чем-то жертвуете: скоростью, надёжностью, деньгами, простотой. Хороший архитектор — не тот, кто нашёл идеальное решение (его не существует), а тот, кто осознанно выбрал «достаточно хорошо» и может объяснить, почему.

System Design — это не:

- **Не архитектура программного обеспечения (Software Architecture) в узком смысле.** Software Architect чаще работает внутри одной системы или продукта. System Design охватывает взаимодействие нескольких систем, внешние интеграции, масштабирование на уровне дата-центров.
- **Не Enterprise Architecture.** Enterprise Architect думает о том, как IT-ландшафт компании соответствует бизнес-стратегии на 5 лет вперёд. System Design ближе к коду.
- **Не просто подготовка к собеседованию.** Да, многие читают книги по System Design ради FAANG. Но если вы не умеете проектировать реальные системы, вы не пройдёте и техническое интервью в средний стартап.

System Design — это: умение взять расплывчатое «сделайте как у конкурентов» и превратить в конкретные сервисы, базы данных, очереди и схемы. С обоснованием.

Зачем нужен System Design: три взгляда

Зачем это бизнесу

Бизнесу плевать на вашу Kafka. Бизнесу важно, чтобы система не падала в чёрную пятницу, чтобы новая фича выходила не за полгода, а за две недели, чтобы разработчики не убегали в другую компанию от бесконечного легаси.

System Design помогает бизнесу:

- **Предсказуемость.** Когда вы понимаете узкие места системы, вы можете оценить, выдержит ли она рост до 10 млн пользователей — или пора закладывать 20 млн на переписывание.
- **Скорость.** Правильно разделённая на модули система позволяет командам работать параллельно, не наступая друг другу на пятки.
- **Экономию.** Выбрать базу данных, которая не требует кластера из 50 машин, когда достаточно одной, — это миллионы в год.

Зачем это архитектору

Архитектор без System Design — как врач без диагноза. Он может вылечить кашель, но пропустить пневмонию.

System Design даёт архитектору:

- **Язык для обсуждения.** Вы перестаёте тыкать пальцем в схему и начинаете говорить: «Вот здесь у нас узкое место по I/O, предлагаю добавить кэш».
- **Инструменты для решений.** CAP-теорема, паттерны отказоустойчивости, стратегии репликации — это не теория ради теории. Это готовые ответы на вопросы, которые возникают в каждом втором проекте.
- **Аргументы для защиты.** «Потому что я так решил» — не аргумент. «Мы выбрали Kafka, потому что нужна гарантия порядка сообщений, а RabbitMQ с этим не справляется» — аргумент.

Зачем это разработчику

Вы не обязаны становиться архитектором. Но вы обязаны понимать, как устроена система, в которой вы работаете.

Знание System Design даёт разработчику:

- **Понимание контекста.** Вы видите не только свой сервис, а всю картину целиком. Это помогает принимать правильные локальные решения.
- **Карьерный рост.** Middle без System Design останется Middle. Senior с System Design становится Team Lead или архитектором.
- **Уверенность на собеседованиях.** System Design интервью — обязательный этап в любой серьёзной компании. Без подготовки его не пройти.

System Design vs Software Architecture vs Enterprise Architecture

Путаница понятий — вечная боль.

Роль	Горизонт	Что делает	Пример вопроса
System Design	Месяцы — год	Проектирует решение под конкретную задачу с учётом нагрузки, бюджетов, сроков	«Как спроектировать систему для 10 млн пользователей?»
Software Architecture	1–2 года	Определяет структуру ПО внутри системы: компоненты, модули, интерфейсы	«Как нам разделить монолит на микросервисы?»
Enterprise Architecture	3–5 лет	Связывает IT-ландшафт компании с бизнес-стратегией	«Какие технологии мы будем развивать, а какие выводить?»

System Design — это мост между бизнесом (Enterprise Architecture) и кодом (Software Architecture). Он отвечает на вопрос «как именно мы это сделаем технически» в условиях конкретного проекта.

Краткая история: от монолита до serverless

1970–1990-е. Монолиты. Система — одна программа. База данных — одна. Деплой — раз в месяц. Масштабирование — вертикальное (покупаем сервер побольше). Минусы: любое изменение — страх, команда из 50 человек толкается в одном коде.

2000-е. Клиент-сервер и SOA. Появились веб-приложения. Системы начали общаться по сети. SOA (Service-Oriented Architecture) — попытка навести порядок в зоопарке интеграций, но ESB (Enterprise Service Bus) часто становился узким местом.

2010-е. Микросервисы и облака. Netflix показал, как масштабироваться до миллионов пользователей. Docker и Kubernetes сделали микросервисы доступными. Облака (AWS, Azure, GCP) — инфраструктура как код. Минусы: сложность. Распределённые системы — это больно.

2020-е. Serverless, event-driven, AI. Serverless (AWS Lambda) — вы платите только за вызовы, инфраструктурой не управляете. Event-driven архитектуры — все общаются через события. ИИ встраивается во всё.

Что это значит для вас: задачи, которые раньше решались одним инженером, теперь требуют понимания десятка технологий. Но и преимущества — можно сделать систему, о которой 20 лет назад даже не мечтали.

Обзор книги: структура и логика изложения

Книга разделена на 6 частей.

Часть 1. Теоретические основы.
CAP-теорема, модели данных, согласованность, масштабирование, отказоустойчивость, наблюдаемость. Это фундамент. Без него вы не сможете принимать осознанные решения.

Часть 2. Архитектурные паттерны.
Интеграции, архитектурные стили, паттерны масштабирования и отказоустойчивости. Это инструменты — готовые решения для типовых задач.

Часть 3. Платформы и инфраструктура.
Контейнеризация, CI/CD, облака. Как ваши архитектурные решения живут в реальном мире.

Часть 4. Проектирование реальных систем (сквозные примеры).
URL Shortener, мессенджер, новостная лента, платёжная система, система голосования (мой уникальный кейс). Каждый пример — от требований до итоговой схемы.

Часть 5. System Design интервью.
Как проходит собеседование, типичные вопросы, тренажёр для самостоятельной подготовки.

Часть 6. Справочник архитектора.
Таблицы сравнения, формулы для оценки нагрузки, глоссарий, ресурсы. Держите под рукой.

Как выжать из книги максимум

Не читайте от корки до корки. Возьмите сценарий, который подходит вам сейчас.

- Готовитесь к интервью? Части 1, 2, 4, 5.
- Хотите стать лучшим архитектором? Читайте всё.
- Ищете ответ на конкретный вопрос? Часть 6.

Рисуйте. Каждый раз, когда видите схему — перерисуйте её от руки. Лучше запоминается, когда сам провёл стрелочку.

Спорьте. В книге есть утверждения, с которыми можно не согласиться. Отлично! Найдите контраргументы. Архитектура — это не догма.

Применяйте. После каждой главы спросите себя: «А как это относится к моему проекту?». Если никак — придумайте, где могло бы пригодиться.

Перечитывайте. Хорошую книгу читают дважды: первый раз — для понимания, второй — для осознания. Через год вы увидите её по-новому.

Добро пожаловать в мир, где каждый выбор — компромисс, каждая схема — упрощение, а каждая система — потенциальная точка отказа.

Это весело. Обещаю.

Глава 1. Принципы проектирования систем

«Правила существуют не для того, чтобы их нарушать. Они существуют, чтобы ты понимал, что именно ты нарушаешь»

Прежде чем мы погрузимся в базы данных, микросервисы и высокие нагрузки, нужно договориться об основах.

В этой главе — принципы, которые работают независимо от технологий. KISS, YAGNI, DRY, SOLID, CAP, PACELC, закон Амдала — это не просто абстракции из учебников. Это инструменты, которые спасают проекты от провала.

Я не буду гнаться за полнотой. Есть десятки принципов. Я беру те, без которых не могу прожить и дня.

1.1 Фундаментальные принципы

KISS (Keep It Simple, Stupid)

Самый недооценённый принцип. Все хотят сделать «правильно», «масштабируемо», «на все случаи жизни». В результате получают систему, которую никто не понимает, которую невозможно изменить и которая падает при первой нестандартной нагрузке.

Самое сложное в архитектуре — не придумать изощрённое решение. Самое сложное — сделать просто.

Как это выглядит на практике:

- Вы выбираете PostgreSQL для 90% задач, а не Cassandra «на будущее»
- Вы пишете монолит, пока не станет очевидно, что нужны микросервисы
- Вы используете REST, где хватает REST, а не GraphQL «потому что модно»

Антипример из моей практики:
В одном стартапе с 3 разработчиками решили делать микросервисы на Kubernetes с Kafka. Через 3 месяца у них была крутая инфраструктура и 0 фич для пользователей. Заказчик ушёл.

Когда KISS работает против вас:
Если вы упрощаете до состояния, где совместная работа нескольких команд

превращается в ад. Пример: монолит на 1 млн строк кода, где 10 команд ждут очереди на деплой. Это уже не просто, это примитивно. Правило: упрощайте только до границ, когда это полезно.

YAGNI (You Aren‘t Gonna Need It)

Следствие KISS. Не делайте того, что может не понадобиться.

Я встречал архитекторов, которые закладывали в проект шардирование «на будущее», хотя текущая нагрузка умещалась на одном ноутбуке. Они тратили 30% времени на сложность, которая может никогда не понадобиться.

YAGNI не значит «не думай о будущем». Это значит «не реализуй то, что не нужно сейчас». Вы можете продумать, как будете шардировать, когда придёт время. Но не пишите код шардирования в первый день.

DRY (Don‘t Repeat Yourself)

В контексте архитектуры DRY значит: не дублируйте логику в разных сервисах, не плодите одинаковые компоненты, централизуйте то, что должно быть централизованным.

Пример нарушения DRY: В компании 3 сервиса, каждый реализует свою авторизацию. В четвёртый раз логику переписывают. Где-то забыли про refresh-токены, где-то не те роли.

Как исправить: Всё, что касается сквозной функциональности (авторизация, логирование, мониторинг), лучше выносить в отдельную библиотеку или использовать API Gateway.

Separation of Concerns (разделение ответственности)

Каждый модуль, сервис, компонент должен отвечать за что-то одно. И делать это хорошо.

Плохо: Сервис «Пользователи» хранит не только профили, но и историю заказов, и платёжные данные, и уведомления. Это божественный сервис. Он упадёт — всё упадёт. Менять его страшно.

Хорошо: Сервис пользователей — только профили. Сервис заказов — заказы. Платежи — платежи. Каждый живёт своей жизнью.

Encapsulation

Сокрытие деталей реализации. Сервис не должен знать, как устроен соседний сервис. Он должен знать только его API.

Почему важно: Если вы поменяете базу данных внутри сервиса, соседи не должны об этом узнать. Если они знают — ваша архитектура хрупкая.

Пример нарушения: Сервис А ходит напрямую в базу данных сервиса Б. Через полгода вы решили разделить базы. Половина запросов сломалась.

Principle of Least Astonishment

Пользователь (разработчик, администратор, API-клиент) не должен удивляться тому, как работает система.

Плохо: API удаления заказа называется `/api/v1/delete_order`, но на самом деле он помечает заказ как удалённый и оставляет в базе. Разработчик клиента удивлён.

Хорошо: Название говорит само за себя. Если удаляет — значит удаляет. Если помечает — называй `/api/v1/mark_as_deleted`.

1.2 SOLID для архитектора

SOLID создавался для ООП, но его принципы отлично работают на уровне сервисов и систем.

S: Single Responsibility

У сервиса должна быть одна причина для изменения.

Пример: Сервис, который считает налоги и отправляет отчёты. Если меняется налоговый кодекс — меняем логику расчёта. Если меняется форма отчёта — меняем логику отправки. Две причины → два изменения в одном месте.

Как разделить: Сервис расчёта налогов + Сервис отправки отчётов, работающий по API.

O: Open-Closed Principle

Сервисы должны быть открыты для расширения, но закрыты для изменения.

Пример: Вы принимаете платежи через один эквайринг. Вам нужно добавить второй. Вместо того чтобы переписывать сервис «Платежи», вы используете паттерн «Стратегия» или плагин.

L: Liskov Substitution

Если у вас есть интерфейс (или контракт API), любой наследник (или реализация) должен работать без сюрпризов.

Пример нарушения: Контракт API платежей гарантирует идемпотентность. Вы подключаете эквайринг, который не поддерживает идемпотентность. Клиенты, которые рассчитывали на этот контракт, получают двойные списания.

I: Interface Segregation

Не заставляйте клиентов зависеть от методов, которые они не используют.

Плохо: Огромный API Gateway на все случаи жизни. Сервису уведомлений не нужно знать про платежи, но библиотека всё равно тянет зависимость.

Хорошо: Много маленьких API. BFF (Backend for Frontend) для каждого типа клиента. Сервис уведомлений общается с узким API.

D: Dependency Inversion

Зависимости должны быть направлены в сторону абстракций, а не конкретных реализаций. Высокоуровневые модули не зависят от низкоуровневых.

Пример нарушения: Сервис «Платежи» напрямую вызывает API конкретного банка. Сменить банк — переписывать сервис.

Как исправить: Ввести интерфейс `PaymentProcessor`. Сервис «Платежи» зависит от интерфейса. Конкретные эквайринги реализуют этот интерфейс.

1.3 CAP-теорема

Теперь переходим к железу распределённых систем. CAP — это не абстракция. Это ограничение, с которым вы будете сталкиваться постоянно.

Формулировка: В распределённой системе из трёх свойств — **C**onsistency (согласованность), **A**vailability (доступность), **P**artition Tolerance (устойчивость к разрыву) — одновременно можно гарантировать не больше двух.

Что это значит на практике:

- **CP-система (согласованность + устойчивость к разрыву):** При сетевом разделении система может отклонить запрос, но гарантирует, что все узлы видят одни и те же данные. Пример: банковские транзакции, бронирование билетов. Лучше сказать «недоступно», чем ошибиться.
- **AP-система (доступность + устойчивость к разрыву):** При разделении система принимает запросы, но данные на разных узлах могут расходиться. Со временем разногласия уходят (eventual consistency). Пример: лента новостей Instagram. Лучше показать ленту без последних постов, чем не показать ничего.
- **CA-система:** В распределённых системах невозможна (это теорема), но вы можете делать вид, что сети нет, работая в одном дата-центре без шардирования. Пример: монолит.

Как CAP помогает принимать решения:

1. Определите, что для бизнеса критичнее — данные всегда актуальны или сервис всегда доступен.
2. Выберите CP или AP.
3. Смиритесь с недостатками.

1.4 PACELC — расширение CAP

CAP отвечает на вопрос «что происходит при сетевом разделении?». PACELC добавляет: «а что происходит в нормальном режиме (Else) без разделения?».

Формула PACELC: Если есть Partition (P), то выбираем между Availability и Consistency (A/C). Else (E) в нормальном режиме выбираем между Latency и Consistency (L/C).

Примеры:

- **DynamoDB (AP/LC):** При разделении выбирает доступность. В нормальном режиме жертвует консистентностью ради низкой задержки.
- **HBase (CP/LC):** При разделении выбирает консистентность. В нормальном режиме тоже жертвует консистентностью? Нет, HBase строго консистентен — он жертвует задержкой.

Почему это важно: Вы можете выбрать систему, которая в проде нагружает сеть, но это не противоречит CAP.

1.5 Теорема Фальстада (и когда SQL, когда NoSQL)

Теорема звучит сложно, но смысл прост: реляционные базы данных хороши, когда у вас чёткая схема и сложные запросы. NoSQL — когда схема гибкая или нужно горизонтальное масштабирование.

Мой алгоритм выбора:

1. **Вам нужны транзакции (ACID), сложные JOIN, строгая согласованность?** → SQL (PostgreSQL).
2. **У вас много записей, слабоструктурированные данные, нужен быстрый поиск?** → документная (MongoDB).
3. **Вам нужна аналитика на миллиардах строк (агрегации, GROUP BY)?** → колоночная (ClickHouse).
4. **Нужен супер-быстрый доступ по ключу (кэш, сессии)?** → key-value (Redis).
5. **Вам нужно очень много записей (логи, события) и горизонтальное масштабирование?** → Cassandra / ScyllaDB.

Важнее инструмент, а не выбор «навсегда». Никто не запрещает использовать PostgreSQL для транзакций и ClickHouse для аналитики, копируя данные через CDC.

1.6 Закон Амдала

Закон Амдала описывает предел ускорения при распараллеливании.

Формулировка: Ускорение ограничено последовательной частью программы.

Если 90% задачи можно распараллелить, а 10% — нет, то максимальное ускорение никогда не превысит 10 раз, даже если взять бесконечное число процессоров.

Как это относится к архитектуре:

- Не пытайтесь горизонтально масштабировать то, что можно масштабировать вертикально. Если нагрузка растёт плавно, сначала увеличьте ресурсы.
- Ищите последовательные узкие места Их не ускорить добавлением серверов.
- Иногда «ускорить» — значит перепроектировать.

1.7 Fallacies of Distributed Computing

8 ложных представлений, которые приходят к каждому инженеру, столкнувшемуся с распределёнными системами.

1. **Сеть надёжна.** Нет. Роутеры перегружаются, DNS-сервера падают, кабели перегрызают экскаваторы.
2. **Задержка равна нулю.** Нет. Пакет из Москвы в Нью-Йорк идёт минимум 80 мс.
3. **Пропускная способность бесконечна.** Нет. Очень быстро забивается.
4. **Сеть безопасна.** Нет.
5. **Топология не меняется.** Нет.
6. **Администратор один.** Нет или их слишком много, и они не синхронизируют конфиги.
7. **Транспортные расходы равны нулю.** Нет, миллиард микросервисов что-то да едят.
8. **Сеть однородна.** Нет.

Что с этим делать: Проектируйте системы, устойчивые к сетевым сбоям. Таймауты, ретраи, circuit breaker, bulkhead — обязательно.

Итог главы 1

1. **KISS, YAGNI, DRY, Separation of Concerns, Encapsulation, Least Astonishment** — спасут ваш проект от преждевременной сложности.
2. **SOLID** работает и на уровне сервисов, и на уровне кода.
3. **CAP-теорема** ограничивает вас: либо консистентность, либо доступность при сетевом разделении.
4. **PACELC** расширяет CAP: даже без сетевых проблем приходится выбирать между задержкой и консистентностью.
5. **Теорема Фальстада** не велит пихать везде одну базу данных.
6. **Закон Амдала** напоминает о последовательных узких местах.
7. **Fallacies** — молитва инженера распределённых систем.

Глава 2. Модели данных

«Выбор базы данных — это не техническое решение. Это архитектурное решение, которое будет преследовать вас годами»

Сколько раз я видел, как проекты начинали с MongoDB, потому что «она модная», а через год бежали к PostgreSQL, потому что понадобились транзакции. И наоборот: упёрлись в PostgreSQL, когда нужна была гибкая схема.

Модель данных — это фундамент. Поменять её на ходу можно, но это «переселение народа с вещами». Дорого, больно и с риском всё потерять.

В этой главе мы разберём основные модели данных, их сильные и слабые стороны, и главное — я дам алгоритм выбора. Как понять, что вам нужна колоночная база, а не ключ-значение? Как почувствовать приближение «полиглота персистенции»?

2.1 Реляционная модель

Реляционные базы данных (PostgreSQL, MySQL) — старый, добрый, проверенный фундамент архитектуры.

ACID: почему реляционные базы не сдаются

Кислотная гарантия — их главный козырь.

- **Атомарность (Atomicity):** транзакция либо выполняется целиком, либо откатывается. Нет ситуации «деньги списались, но не пришли».
- **Согласованность (Consistency):** транзакция переводит базу из одного корректного состояния в другое, не нарушая ограничений (внешние ключи, уникальность, проверки).
- **Изоляция (Isolation):** параллельные транзакции не мешают друг другу. Уровни изоляции — отдельная песня.
- **Долговечность (Durability):** если транзакция подтверждена, данные переживут перезагрузку.

Для каких задач критичен ACID:

- Платёжные системы. Нельзя потерять перевод.
- Бронирование билетов. Нельзя продать одно место дважды.

- Бухгалтерия. Двойная запись не терпит ошибок.

Нормальные формы: теория, которую нужно знать

Нормализация — способ организации данных без дублирования.

- **1NF:** атомарные значения, нет повторяющихся групп.
- **2NF:** избавляемся от частичных зависимостей (столбец зависит от части ключа).
- **3NF:** избавляемся от транзитивных зависимостей. Не храним возраст, если есть дата рождения.
- **BCNF (Бойса-Кодда):** строже 3NF — каждая детерминанта — потенциальный ключ.
- **4NF:** многозначные зависимости.
- **5NF:** зависимости соединения.

На практике редко кто доходит до 4NF. Обычно достаточно 3NF.
Денормализация делается осознанно для скорости (аналитика, отчёты).

Индексы: B-Tree, Hash, GiST, Partial, Covering

Индексы ускоряют чтение, замедляя запись. Правильно выбранный индекс — может заменить оптимизацию кода.

- **B-Tree:** универсальный вариант, подходит для `= , > , < , BETWEEN, LIKE 'prefix%'`.
- **Hash:** только для `=` (точного равенства). Быстрее B-Tree, но не поддерживает сортировку и диапазоны.
- **GiST/GIN:** для полнотекстового поиска и геоданных.
- **Partial:** индекс на часть таблицы (`WHERE is_active = true`). Экономит место, ускоряет специфичные запросы.
- **Covering:** включает все столбцы для запроса. Не нужно обращаться к основной таблице. Пример: `CREATE INDEX ... INCLUDE (name)`.

Транзакции и уровни изоляции

Когда два пользователя читают и пишут одни и те же данные, без изоляции не обойтись. SQL определяет 4 уровня:

1. **Read Uncommitted (грязное чтение):** читаете незафиксированные изменения. Почти не используем.

2. **Read Committed:** читаете только зафиксированные данные, но в рамках транзакции данные могут измениться (неповторяемое чтение).
3. **Repeatable Read:** в транзакции всё время видите одни и те же строки, но могут добавиться новые (фантомное чтение).
4. **Serializable:** максимальная изоляция — как будто транзакции выполняются одна за другой. Медленно.

   Deadlock (взаимная блокировка): транзакция А удерживает строку 1 и ждёт строку 2, транзакция Б — наоборот. БД обычно сама обнаруживает и откатывает одну из транзакций.

   Совет: не держите транзакции надолго. Чем быстрее закоммитили, тем меньше блокировок.

2.2 Документная модель

Документные базы (MongoDB, Couchbase) хранят данные в JSON-подобных документах. Схема не жёсткая — можно добавлять поля на лету.

Когда выбирать:

- **Гибкая схема с частыми изменениями.** 10 типов товаров с разными характеристиками. В реляционной БД это набор nullable-столбцов или EAV; в документной — естественно.
- **Сильные связи внутри документа.** Весь заказ со всеми позициями можно хранить в одном документе.
- **Высокая нагрузка на запись, простота масштабирования.** MongoDB легко шардируется.

Минусы: нет транзакций и JOIN (агрегации работают, но не так удобно).

2.3 Колоночная модель

ClickHouse, Vertica, Redshift: данные хранятся не по строкам, а по колонкам.

Когда это выигрывает:

- Сложные агрегации над миллиардами записей.
- Нужно быстро читать несколько колонок из всех строк (агрегации).
- Компрессия на диске иногда в 10 раз лучше строчной.

Плата за удобство:

- Медленные точечные UPDATE/DELETE (лучше пачки вставок с перезаписью).
- Сложная работа с JOIN.

2.4 Графовая модель

Neo4j, ArangoDB: вершины (сущности) и рёбра (связи) с атрибутами.

Когда графовая база выигрывает:

- Системы рекомендаций. «Друзья друзей» на SQL — это рекурсивные JOIN.
- Антифрод и анализ связей. Связанные транзакции, общие устройства, IP-адреса.
- Социальные сети. Лента друзей, группы, подписки.

Не используйте графовую базу, если у вас простые CRUD и запросы без глубокой навигации по связям.

2.5 Векторная модель

Новое, но уже мейнстримное направление.

Что такое векторная БД:

- Хранит embedding (векторное представление текста, изображения, звука).
- Ищет ближайших соседей по косинусному сходству (ANN: approximate nearest neighbor).

Применение:

- Поиск по семантическому сходству. «Как починить кран» → найдёт статьи про сантехнику.
- RAG (Retrieval-Augmented Generation) для LLM. Подаём в промпт релевантные документы из вашей базы.
- Системы рекомендаций.

Инструменты: Qdrant, Milvus, Weaviate, pgvector (расширение для PostgreSQL).

2.6 Сравнение моделей данных (как выбирать)

Таблица критериев

Критерий	Реляционная	Документная	Колоночная	Графовая	Векторная
Схема данных	Строгая	Гибкая	Строгая	Гибкая	Гибкая
Транзакции ACID	☑	⚠ ограниченно	✗	☑	✗
Сложные JOIN	☑	⚠ агрегации	✗	☑ граф	✗
Агрегации/OLAP	⚠	✗	☑	✗	✗
Горизонтальное масштабирование	⚠ сложно	☑	☑ нативно	⚠	☑
Глубокий поиск связей	⚠	✗	✗	☑	✗
Семантический поиск	✗	✗	✗	✗	☑

Примеры выбора

- **Платёжный сервис.** Нужны транзакции, строгая согласованность. → **PostgreSQL** (с репликацией для отказоустойчивости).
- **Каталог товаров.** Товары с разными характеристиками. → **MongoDB** (гибкая схема, простое шардирование).
- **Аналитика по миллиардам строк.** Явка на выборах по регионам за секунду. → **ClickHouse**.
- **Система рекомендаций / антифрод.** Сложные связи. → **Neo4j**.
- **Поиск похожих товаров / RAG для чат-бота. → pgvector** (если уже есть PostgreSQL) или отдельная векторная БД.

Polyglot Persistence (полиглот персистенции)

Никто не заставляет выбирать одну базу на всё. Современная архитектура может включать:

- PostgreSQL для транзакций (пользователи, заказы, платежи)
- ClickHouse для аналитики (поведенческие отчёты, дашборды)
- Redis для кэша (сессии, корзина)
- Neo4j для рекомендаций (друзья, похожие товары)
- Qdrant для поиска похожих изображений

Плата: сложные синхронизация данных и эксплуатация.

Итог главы 2

Мы разобрали 5 моделей данных:

1. **Реляционная:** транзакции, строгая схема — фундамент.
2. **Документная:** гибкая схема, простое шардирование — для каталогов и быстрого прототипирования.
3. **Колоночная:** аналитика на огромных объёмах.
4. **Графовая:** когда связи сложнее данных.
5. **Векторная:** семантический поиск и RAG.

Ни одна из них не заменит другую для всех задач. В современном мире нормально иметь в проекте несколько БД.

Что делать прямо сейчас:

1. Оцените текущий проект: соответствует ли модель данных нагрузке?
2. Если не уверены — проведите POC (proof of concept) с альтернативой. 2 дня тестов стоят дешевле, чем год на поддержке неправильной базы.
3. Зафиксируйте выбор в ADR.

Глава 3. Согласованность и репликация

«В распределённых системах нет выбора между идеальной согласованностью и идеальной доступностью. Есть выбор между тем, как именно вы будете страдать»

Когда я впервые столкнулся с проблемой согласованности в реальном проекте, это выглядело невинно. Пользователь обновил свой профиль, но на другой вкладке увидел старые данные. «Ну и что? — подумал я. — Обновится через секунду».

А потом пришёл заказчик из банка.

«Артур, — сказал он спокойным голосом, от которого у меня побежали мурашки. — Деньги списались, но баланс не изменился. Что значит "через секунду"?».

В тот день я понял: согласованность — это не абстрактная теорема из учебника. Это разница между спокойной ночью и звонком от клиента в 3 часа утра.

В этой главе мы разберём, что такое согласованность, почему она не может быть идеальной в распределённых системах, и как выбирать между «всегда актуально» и «всегда доступно». С примерами, с цифрами, с болью.

Часть 1. Модели согласованности

Согласованность — это про то, когда и в каком порядке разные узлы системы видят одни и те же данные. Если у вас одна база данных на одном сервере — проблем нет. Как только вы добавили вторую реплику или шард — всё, вы в мире компромиссов.

1.1 Линеаризуемость (Strong Consistency / Linearizability)

Самая строгая модель. Система ведёт себя как одна машина. После того как запись подтверждена, любой следующий читатель увидит её и все последующие записи.

Как это работает технически:

1. Клиент отправляет запись на лидер-узел
2. Лидер синхронно реплицирует изменения на кворум (N/2+1) узлов

3. Только после подтверждения от кворума клиент получает ответ «успех»
4. Читатель (даже на другой реплике) дожидается применения всех подтверждённых записей

Цена, которую вы платите:

- Задержки растут (ждём ответа от нескольких узлов)
- Доступность страдает (если лидер недоступен — запись невозможна)
- Пропускная способность снижается

Где это строго необходимо:

- Банковские счета (нельзя списать и не зачислить)
- Бронирование билетов (нельзя продать одно место дважды)
- Инвентаризация (последний товар должен уехать только к одному покупателю)
- Системы голосования (нельзя учесть один голос дважды)

Где можно обойтись без линеаризуемости:
В системах, где цена ошибки низкая (лайки, просмотры, рекомендации).

1.2 Последовательная согласованность (Sequential Consistency)

Более слабая модель. Порядок операций сохраняется в рамках одного процесса, но между разными процессами нет единой глобальной последовательности.

Пример: Пользователь А добавил пост, пользователь Б добавил комментарий к нему. В рамках одного клиента («лента А») комментарий будет после поста. Но другой клиент («лента Б») может увидеть комментарий раньше поста — и это допустимо.

Где применяется: В системах, где важна причинно-следственная связь внутри одного пользователя, но не между разными.

1.3 Причинная согласованность (Causal Consistency)

Сильная, но не самая строгая модель. Если событие А причинно зависит от события Б, то все увидят А после Б. Независимые события могут видеть в разном порядке.

Как это работает на практике:

Представьте, что Алиса комментирует пост Боба. Комментарий причинно зависит от поста. Поэтому никто не увидит комментарий раньше самого поста. Но если Алиса и Боб меняют аватарки одновременно (независимые события), разные пользователи могут видеть изменения в разном порядке.

Где это критически важно:

- Комментарии в соцсетях (ответ не может появиться раньше родительского комментария)
- Чат-системы (сообщение-ответ не может обогнать исходное сообщение)
- Системы совместной работы (документ не может зависеть от правки, которая ещё не сохранена)

Почему это удобнее линеаризуемости:
Независимые события (смены аватарки, лайки разным постам) могут обрабатываться параллельно без ожидания глобальной синхронизации.

Как это реализуется: Каждому событию присваивается вектор времени. Операции проверяют, видел ли узел все причинные предшественники. Если нет — операция ждёт или отклоняется.

1.4 Read-your-writes (чтение своих записей)

Очень полезная модель. Читатель всегда видит свои собственные записи. Чужие — не обязательно.

Объяснение на примере:

Вы обновили свой профиль. Сразу после этого вы видите новое имя. Ваш друг, зашедший на ваш профиль через секунду, может ещё видеть старое имя. Это не страшно. Но если бы вы сами увидели старое имя — это было бы нарушение ожиданий «я же только что поменял!».

Где это критически важно:

- Профиль пользователя (вы должны видеть свои изменения сразу)
- Корзина интернет-магазина (вы должны видеть добавленный товар)
- Настройки приложения (сохранил — увидел)

Как реализовать: Нужно уметь маршрутизировать запросы одного пользователя на одну и ту же реплику (липкие сессии) или хранить версию последнего обновления в сессии.

Компромисс: Липкие сессии нарушают балансировку нагрузки. Если реплика упала, пользователь может потерять доступ к своим недавним изменениям.

1.5 Монотонные чтения (Monotonic Reads)

Защита от путешествий во времени. Если вы прочитали значение, то последующие чтения не увидят «откат в прошлое».

Пример проблемы:

Вы зашли на сайт и увидели баланс 1000 рублей. Обновили страницу — а там 900 рублей. Вы не совершали трат. Что случилось? Первый запрос попал на свежую реплику, второй — на устаревшую. Баланс «уменьшился». Это нарушает принцип наименьшего удивления.

Как гарантировать: Как и для read-your-writes, нужны липкие сессии или версионирование.

Где важно: Везде, где пользователь видит данные, которые могут измениться (баланс, статус заказа, инвентаризация).

1.6 Итоговая согласованность (Eventual Consistency)

Самая слабая модель. Система не гарантирует, что читатель увидит свежую запись сразу, но обещает: если запись прекратить, все читатели сойдутся к одному значению.

Срок сходимости: от миллисекунд до часов. Зависит от загрузки сети, частоты конфликтов, выбранной топологии репликации.

Где применяется:

- Лента новостей (допустимо увидеть пост на пару секунд позже)
- Лайки в Instagram (не страшно, если счётчик лайков у разных пользователей немного различается)
- DNS-записи (кэшируются на минуты/часы)
- Системы аналитики (отчёты могут обновляться раз в час)

Почему этого достаточно: Пользователь не заметит разницы в 1–2 секунды. А если заметит — не расстроится. В отличие от ситуации, когда деньги пропали.

1.7 Сравнительная таблица моделей согласованности

Модель	**Гарантии**	**Задержка записи**	**Доступность при разделении**	**Типичное применение**
Strong (Linearizability)	Максимальные	Высокая (ожидание кворума)	CP (может отказать)	Платежи, инвентаризация, голосование
Sequential	Сильные (локальный порядок)	Средняя	CP	Лента одного пользователя
Causal	Сильные (причинные связи)	Средняя	AP с гарантиями	Комментарии, чаты, документы
Read-your-writes	Средние (только для себя)	Низкая	Высокая	Профили, корзина, настройки
Monotonic Reads	Средние (монотонность)	Низкая	Высокая	Балансы, статусы
Eventual	Минимальные	Низкая	Максимальная	Ленты, лайки, DNS, аналитика

Часть 2. Репликация

Репликация — это хранение копий одних и тех же данных на разных узлах.

Зачем нужна репликация:

1. **Доступность** — если узел упал, другие продолжают работать
2. **Масштабирование чтения** — нагрузку чтения можно распределить между репликами
3. **Близость к пользователю** — можно расположить реплику в регионе пользователя
4. **Изоляция нагрузок** — аналитические запросы можно отправлять на отдельную реплику

Плата за репликацию:

1. **Сложность согласованности** — данные могут расходиться
2. **Затраты на хранение** — N копий требуют N места на дисках
3. **Сетевой трафик** — нужно передавать изменения между узлами
4. **Задержки записи** — даже асинхронная репликация добавляет накладные расходы

2.1 Single-Leader (мастер-слейв)

Классическая модель, с которой всё начиналось.

Как работает:

- Есть один лидер (leader/master), который принимает запросы на запись
- Есть несколько последователей (followers/slaves), которые копируют данные с лидера
- Запросы на чтение можно распределять между лидером и последователями

Синхронная vs асинхронная репликация:

Тип	Задержка	Гарантии	Риски
Синхронная	Клиент ждёт, пока данные запишутся	Минимальная потеря данных	Если последователь

Тип	Задержка	Гарантии	Риски
	на лидера и на всех последователей		недоступен, запись встанет
Асинхронная	Клиент получает ответ сразу после записи на лидера	Высокая доступность	При падении лидера можно потерять непереданные изменения

На практике: Один синхронный последователь (гарантия), остальные — асинхронные (скорость).

2.2 Multi-Leader (многолидерная)

Эволюция: писать можно в несколько узлов. Лидеры синхронизируются между собой.

Когда нужна:

- Работа в нескольких дата-центрах (каждый дата-центр имеет своего лидера)
- Мобильные приложения с офлайн-режимом (каждое устройство — лидер)
- Коллаборативные редакторы (Google Docs)

Плата за удобство: конфликты. Если один и тот же объект изменить в двух лидерах одновременно, возникает конфликт.

Разрешение конфликтов:

- **Last Write Wins (LWW)** — выигрывает последнее по времени. Просто, но можно потерять данные.
- **Merge (слияние)** — для корзины, чатов (как в Git).
- **User-defined resolution** — показать оба варианта и спросить.
- **CRDT** — математически гарантирует сходимость.

2.3 Quorum-based

Нет явного лидера. Клиент пишет в набор реплик и ждёт подтверждения от большинства (N/2+1).

Чтение кворума: Клиент читает из нескольких реплик и берёт актуальную версию (по номерам или меткам времени).

Параметры:

- N — общее число реплик
- W — число реплик для подтверждения записи
- R — число реплик для чтения

Условие консистентности: W + R > N. Читатель увидит хотя бы одну свежую запись.

2.4 Топологии репликации

Цепочная (chain): A → B → C → A. Экономия трафика, перестроение при отказе — сложно.

Кольцевая: каждый узел соединён с соседними. Простая, при отказе одного узла кольцо разрывается.

Звезда: центральный узел рассылает изменения всем. Простота, но центральный узел — узкое место.

Полносвязная: все со всеми. Максимальная надёжность, квадратичный рост соединений.

Часть 3. Теорема CAP на практике

Формулировка: Из трёх свойств — Consistency, Availability, Partition Tolerance — в распределённой системе можно гарантировать не более двух.

CP-система: выбирает согласованность. При разделении сеть может сказать «не знаю» или отказать.

AP-система: выбирает доступность. При разделении сеть продолжает работу, но узлы могут расходиться.

Реальные примеры простыми словами

Платежи (CP): Если связь между серверами прервана, система откажется обрабатывать транзакцию. Лучше сказать «сервис недоступен», чем допустить ошибку.

DNS (AP): Записи кэшируются, обновляются с задержками. Важнее, чтобы сайт открылся, даже если IP чуть устарел.

Часть 4. Конфликты и их разрешение

Конфликты неизбежны в AP-системах и многолидерной репликации.

Методы разрешения (от простого к сложному)

1. **Last Write Wins (LWW):** выигрывает последнее по времени обновление. Просто, но теряет данные.
2. **Merge (слияние):** для корзины покупок или чатов — объединяем изменения.
3. **User-defined resolution:** показать оба варианта и спросить.
4. **CRDT (Conflict-free Replicated Data Type):** структуры данных, которые математически сходятся. Git — граф коммитов, Redis Enterprise — счётчики.

Итог главы 3

Модели согласованности:

- Strong — для денег и критичных операций
- Causal — для комментариев и чатов
- Eventual — для лайков и лент

Виды репликации:

- Single-leader — самый простой
- Multi-leader — гибкий, но конфликтный
- Quorum — надёжный, но сложный в настройке

CAP-теорема: CP vs AP, третьего не дано. Выбирайте осознанно.

Что делать прямо сейчас:

1. Перечислите все типы данных в вашем проекте.

2. Для каждого типа выберите модель согласованности (strong, causal, eventual).
3. Выберите топологию репликации.
4. Зафиксируйте выбор в ADR (Architecture Decision Record).

Не пытайтесь сделать всё сильно согласованным — это ударит по доступности и задержке. И не делайте всё итогово согласованным — там, где важны деньги, клиенты этого не простят. Ищите золотую середину под каждый тип данных.

Причина, по которой мы так подробно разобрали согласованность: выбор не той модели стоит компаниям миллионов рублей. Я видел проекты, где на всё ставили eventual consistency и теряли деньги. И проекты, где на всё ставили strong consistency и теряли пользователей, потому что система была недоступна. Не повторяйте этих ошибок.

Следующая глава — «Масштабирование» — добавит серверы, партиции и немного здорового цинизма.

Глава 4. Масштабирование

«Добавить ещё один сервер — это не магия. Это способ отложить неизбежное»

Знаете, что общего между масштабированием и диетой? И то, и другое откладывают на потом, пока проблема не станет критической. «Потом добавим серверов», «потом оптимизируем запросы», «потом перепишем архитектуру». А потом наступает чёрная пятница, и всё падает.

В этой главе я расскажу, как масштабировать системы так, чтобы они не падали в самый ответственный момент. Когда добавлять серверы, когда переписывать код, а когда — бежать в облака.

Часть 1. Два пути масштабирования

1.1 Вертикальное масштабирование (scaling up)

Вы просто берёте сервер помощнее. Больше CPU, больше RAM, быстрее диски.

Плюсы:

- Простота. Никакой магии — купил железо, установил, всё работает. Команда не учится новым технологиям, код не переписывается.
- Транзакции и согласованность остаются простыми — всё на одной машине.
- Латентность минимальная — данные не ходят по сети.

Минусы:

- Потолок. Даже у суперкомпьютера есть предел. В какой-то момент вы не сможете купить сервер мощнее (или он будет стоить как самолёт).
- Стоимость растёт нелинейно. Сервер в 2 раза мощнее часто стоит в 4 раза дороже.
- Единая точка отказа. Упадёт этот сервер — упадёт всё.

Мой опыт: В ФСО почти всё было вертикальным. Бюджеты позволяли купить один мощный сервер вместо трёх слабых. Администраторов было мало, а поддерживать кластер — это отдельная боль.

Когда выбирать: На старте проекта (быстро, дёшево, сердито), когда нагрузка предсказуема и растёт плавно, когда команда не готова к горизонтальному масштабированию.

1.2 Горизонтальное масштабирование (scaling out)

Добавляете не один мощный сервер, а много маленьких. Нагрузка распределяется между ними.

Плюсы:

- Нет теоретического потолка. Хоть 10 000 серверов — добавляй по мере роста.
- Дешевле на больших масштабах. 10 дешёвых серверов часто выгоднее 1 монстра.
- Отказоустойчивость. 1 сервер из 10 упал — остальные работают.
- Технологический вектор развития. Все крупные компании (Google, Amazon, Яндекс) масштабируются горизонтально.

Минусы:

- Сложность разработки. Распределённые системы — это боль (CAP-теорема, сети, согласованность).
- Сложность эксплуатации. Кластер из 100 серверов нужно мониторить, балансировать, чинить.
- Сетевые задержки. Данные ходят между серверами, это медленнее, чем локальная память.
- Согласованность требует дополнительных усилий.

Мой опыт: В Ростелекоме мы строили кластер из 50+ серверов для «Выборов 2.0». Система выдержала нагрузку, но мы потратили месяцы на настройку сети, отказоустойчивости и мониторинга.

Когда выбирать: Нагрузка растёт быстро и непредсказуемо, важна отказоустойчивость (система не должна падать), команда готова учиться.

1.3 Сравнительная таблица

Критерий	Вертикальное	Горизонтальное
Простота	Высокая	Низкая
Максимальная мощность	Есть потолок	Нет потолка
Стоимость на малых масштабах	Низкая	Высокая
Стоимость на больших масштабах	Очень высокая	Низкая
Отказоустойчивость	Нет (SPOF)	Есть
Согласованность данных	Простая	Сложная

Золотое правило: Начинайте с вертикального масштабирования, пока оно дешево и просто. Переходите к горизонтальному, когда упрётесь в потолок или когда не справляется вертикальное.

Часть 2. Шардирование

(горизонтальное партиционирование)

Шардирование — это разбиение данных на части (шарды) и распределение их по разным серверам. Каждый сервер отвечает за свою часть.

Ключевое отличие от репликации: Репликация — копии одних и тех же данных на разных серверах. Шардирование — разные данные на разных серверах.

2.1 Алгоритмы шардирования

Range-based (по диапазону значений): данные разбиваются на диапазоны (А–Д, Е–О, П–Я).

Плюсы: простота, легко добавлять новые шарды (просто создать новый диапазон).

Минусы: горячие точки. Пользователи на букву «А» могут быть активнее, чем на «Я».

Пример: ключи от 0 до 10 000 → шард 1, 10 001–20 000 → шард 2.

Hash-based (по хэшу ключа): ключ прогоняется через хэш-функцию, результат определяет шард.

Плюсы: равномерное распределение.

Минусы: сложно добавлять новые шарды — при изменении числа шардов почти все ключи переезжают.

Consistent hashing (кольцевой хэш): хэш-пространство представляется в виде кольца. Ключ попадает на шард, чей хэш следующий по кольцу. Добавление нового узла затрагивает только его соседей.

Где используется: Amazon DynamoDB, Cassandra, Akamai CDN, Redis Cluster.

2.2 Проблема горячих точек

Некоторые ключи могут быть популярнее других. Например, профиль президента читают чаще, чем профиль обычного пользователя.

Решения:

- Разбить популярный ключ на несколько (user_1, user_2, user_3).
- Добавить кэш перед шардированным слоем.
- Выделить популярные сущности в отдельный шард.

2.3 Перешардирование (resharding)

Наступает момент, когда старые шарды переполнились или нагрузка выросла. Нужно добавлять шарды.

Проблема: При hash-based шардировании изменение числа шардов требует перекладки почти всех данных. Это может занять дни.

Решение: Использовать согласованное хэширование (при добавлении узла переезжает только 1/N данных). Или двухуровневое шардирование (ключ → шард, шард → сервер).

2.4 Шардирование на практике

В базах данных: MongoDB делает шардирование прозрачным (указываете ключ шардирования, драйвер сам маршрутизирует запросы).

В прикладном коде: Вы сами решаете, в какой шард писать. Например, по user_id % number_of_shards.

API Gateway + шардирование: Gateway может направлять запросы на нужный шард, анализируя путь или заголовки.

Часть 3. Авто-масштабирование

Облака позволяют масштабироваться автоматически.

Cloud Auto-scaling:

- Горизонтальное: добавляем новые инстансы при росте нагрузки
- Вертикальное: заменяем инстанс на более мощный (с перезагрузкой)
- Predictive: используем ML, предсказывающий нагрузку

Kubernetes HPA (Horizontal Pod Autoscaler):

- Добавляет новые поды при росте метрики (CPU, память, кастомные метрики)
- Убирает поды при падении нагрузки

Пример конфигурации HPA:
minReplicas: 2
maxReplicas: 10
targetCPUUtilizationPercentage: 70

Правильный баланс: Авто-масштабирование не спасёт от плохой архитектуры. Если ваш код содержит узкое место (одна общая база данных), добавление сотен подов её добьёт.

Итог главы 4

1. **Вертикальное масштабирование** — просто, понятно, не требует переписывания кода. Начинайте с него.
2. **Горизонтальное масштабирование** — единственный путь для больших нагрузок. Платите сложностью.
3. **Шардирование** — разбиение данных на части. Главная боль: перешардирование и горячие точки.
4. **Авто-масштабирование** — автоматизация добавления/удаления ресурсов, но не панацея.

Что делать прямо сейчас:

1. Проверьте метрики вашей системы. Где узкое место? CPU, память, диск, сеть, БД?
2. Оцените рост нагрузки. Если за год она вырастет в 5+ раз — пора думать о горизонтальном масштабировании.
3. Если выбрали шардирование — решите, как будете перешардироваться через год.
4. В облаке настройте авто-масштабирование, но с паникой не переусердствуйте.

Следующая глава — «Отказоустойчивость» — научит вас строить системы, которые не падают даже когда всё идёт не по плану.

Глава 5. Отказоустойчивость

«Системы не падают из-за того, что что-то пошло не так. Они падают из-за того, что не было спланировано, что что-то пойдёт не так»

В 2019 году я дежурил в чёрную пятницу. Нагрузка была в 10 раз выше обычной. Я сидел с чашкой кофе и смотрел на дашборды, которые медленно, но верно краснели. А потом график ушёл в ноль.

«Всё, — подумал я. — Мы упали».

Но через 20 секунд график вернулся. Система не умерла. Она замедлилась, но выжила. Потому что мы заложили **отказоустойчивость**.

В этой главе я расскажу, как строить системы, которые выживают в аду. Паттерны, инструменты, архитектурные решения — и немного историй о том, как я не спал ночами.

Часть 1. Паттерны устойчивости (Resilience Patterns)

1.1 Timeout (таймаут)

Проблема: Вы вызываете внешний API, который завис. Ваш поток ждёт ответа. Пользователь ждёт. Память заканчивается. Всё падает.

Решение: Всегда устанавливайте таймауты. Соединение, чтение, запись — на каждый вызов.

Как это выглядит в коде:

python

```
# Плохо: requests.get(url)  # может висеть вечно
# Хорошо:
requests.get(url, timeout=(3.0, 5.0))  # 3 сек на соединение, 5 сек на чтение
```

Совет: Не ставьте один таймаут для всего. Для локальных вызовов время меньше, для внешних API — больше, для файловых операций — ещё больше. Адаптируйтесь к среде.

1.2 Retry (повтор)

Проблема: Случился временный сбой: сеть «потеряла» пару пакетов, база перегружена, внешний API на секунду отвалился.

Решение: Повторить запрос автоматически.

Правила хорошего Retry:

- **Ограничьте число попыток.** 3–5 повторов — обычно достаточно.
- **Экспоненциальная задержка.** С каждой попыткой ждём в 2 раза дольше.
- **Jitter (случайность).** Не повторяйте точно в 1, 2, 4 секунды — иначе все клиенты по расписанию ударят одновременно.
- **Знайте идемпотентность.** Retry безопасен только если операцию можно повторить без побочных эффектов.

Где работает Retry, а где не работает: Запрос на чтение можно повторять сколько угодно. Запрос на создание платежа — только если вы реализовали идемпотентность (один и тот же запрос не создаст два платежа).

1.3 Circuit Breaker (предохранитель)

Проблема: Сервис Б умер. Сервис А продолжает долбить его запросами, тратя ресурсы. Пользователи ждут. Всё падает.

Решение: Circuit Breaker — три состояния:

1. **Closed (замкнут):** Запросы идут. Считаем ошибки. Если ошибок слишком много — переходим в Open.
2. **Open (разомкнут):** Запросы не идут, сразу возвращаем ошибку. Ждём таймаут (обычно 10–60 сек).
3. **Half-Open (полуоткрыт):** Пропускаем ограниченное число запросов для проверки. Если успешны — переходим в Closed. Если нет — возвращаемся в Open.

Как выбрать порог ошибок: 50% ошибок за 10 секунд для высоконагруженной системы, 5 ошибок подряд для критичной.

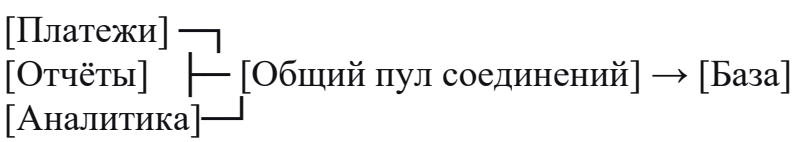
Где взять Circuit Breaker:

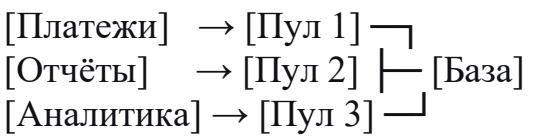
- Библиотеки: Hystrix (Java), Resilience4j (Java), Polyglot — реализуйте сами.
- API Gateway (Kong, NGINX).
- Service Mesh (Istio, Linkerd).

1.4 Bulkhead (перегородка)

Проблема: Сервис «Платежи» умер, но он использует общий пул соединений с сервисом «Пользователи». Сервис «Пользователи» тоже умер. И так по цепочке — всё падает.

Решение: Разделите ресурсы на изолированные пулы.





У каждого сервиса свой пул соединений. Если «Платежи» исчерпали свой пул — «Отчёты» и «Аналитика» продолжают работать.

Где использовать: Пул соединений с БД, HTTP-клиенты к разным внешним сервисам, потоки в приложении для разных типов задач.

1.5 Fallback (запасной план)

Проблема: Основной сервис недоступен. Что делать?

Решение: Запасной вариант:

- **Статический fallback:** Вернуть кэшированную версию данных.
- **Упрощённый fallback:** Вернуть подмножество данных (например, список товаров без цен).
- **Асинхронный fallback:** Встать в очередь и обработать позже.

Пример из жизни: При недоступности сервиса погоды показываем «по данным на вчера». При недоступности сервиса платежей кладём заказ в корзину, а не теряем его.

Важно: Fallback не должен быть сложнее основного решения. Цель — не упасть, а не идеально отработать.

1.6 Graceful degradation (мягкая деградация)

Идея: Система может работать не на 100%, но продолжает работать. Отключаем неважные функции, чтобы спасти важные.

Пример: Интернет-магазин в чёрную пятницу: отключаем рекомендации, но оставляем добавление в корзину и оформление заказа.

Как реализовать:

- Функции должны быть **ранжированы** по важности.
- Система должна **чувствовать** нагрузку (метрики, CPU, память).
- При превышении порога автоматически отключаем неважные функции.

Часть 2. Избыточность (Redundancy)

2.1 Модели избыточности

Active-Passive: Работает один узел, второй ждёт в режиме сна. Если активный упал, пассивный просыпается и подхватывает нагрузку. Просто, но второй сервер простаивает.

Active-Active: Работают оба узла, нагрузка балансируется. Выше доступность и эффективность, но сложнее обеспечить консистентность.

2.2 N+1

«Запасной узел сверх минимума». Если системе нужно 3 сервера для работы — поставьте 4.

Где применимо: Серверы приложений, балансировщики, диски в RAID. Для БД недостаточно (нужна репликация с синхронизацией).

Часть 3. Асинхронность как способ выжить

Синхронный вызов: клиент ждёт ответа. Если сервис упал — клиент упал вместе с ним.

Асинхронный вызов: клиент отправил запрос и забыл. Ответ придёт потом через очередь.

Плюсы асинхронности:

- Отказ сервиса-обработчика не убивает отправителя.
- Можно сглаживать пики нагрузки (очередь накопится, обработчик разгребёт).

Минусы:

- Сложность отладки.
- Нет немедленного ответа (не подходит для UI).

Что выбрать: Если пользователь ждёт ответа (платеж, регистрация) — нужен синхрон (но с таймаутами). Если ответ не нужен немедленно (отправка email, генерация отчёта) — асинхрон.

Часть 4. Chaos Engineering

Идея: Не ждите, пока система упадёт сама. Ломайте её специально, чтобы найти слабые места.

Принципы:

1. Определите нормальное состояние системы (метрики).
2. Выдвиньте гипотезу: система выживет, если упадёт один сервис.
3. Поставьте эксперимент (отключите сервис или замедлите сеть).
4. Откатите изменения.
5. Если гипотеза не подтвердилась — чините.

Инструменты:

- Chaos Monkey (Netflix) — отключает сервисы случайно.
- Gremlin — коммерческая платформа.
- Litmus — open-source для Kubernetes.

Мой опыт: В Ростелекоме мы каждую неделю запускали Chaos Monkey на тестовом кластере. Нашли 5 критических багов до того, как в прод упало.

Итог главы 5

1. **Timeout** — не ждите вечно.
2. **Retry** — повторяйте, но разумно (без DoS-атаки на самих себя).
3. **Circuit Breaker** — не долбите в закрытую дверь.
4. **Bulkhead** — не пускайте соседей в чужой карман.

5. **Fallback** — всегда имейте план Б.
6. **Graceful degradation** — лучше что-то, чем ничего.
7. **Redundancy** — один — хорошо, два — надёжнее.
8. **Асинхронность** — отказ не цепной.
9. **Chaos Engineering** — проверяйте устойчивость в бою.

Что делать прямо сейчас:

1. Пройдите по каждому паттерну и спросите: «А что у нас с таймаутами? А retry? А circuit breaker?».
2. Напишите список всех зависимостей вашей системы. Классифицируйте их по важности.
3. Добавьте monitoring для каждого паттерна.
4. Запустите Chaos Monkey в тестовой среде. Завтра же.

Следующая глава — «Наблюдаемость (Observability)» — покажет, как понять, что происходит в системе, даже когда вы спите.

Глава 6. Наблюдаемость (Observability)

«Если вы не можете измерить систему, вы не можете ею управлять. Если вы не можете её наблюдать, вы не можете узнать, что она сломалась»

В 2018 году наша система упала в 3 часа ночи. Я проснулся от звонка дежурного администратора: «Артур, всё красное, ничего не работает, а логи молчат».

Мы не знали, что пошло не так. У нас были логи — горы логов. Но найти в них причину было невозможно. У нас были метрики — но они показывали только «что», а не «почему». У нас не было трейсинга, поэтому мы не могли понять, какой из 10 сервисов завис.

Система простояла 2 часа. Потому что мы не могли её наблюдать.

После этого случая я пересмотрел подход к наблюдаемости. Теперь это не «дополнительная опция», а обязательное требование к архитектуре.

В этой главе я расскажу, как сделать систему прозрачной. Логи, метрики, трейсинг — три столпа наблюдаемости. И как не утонуть в их данных.

Часть 1. Логирование (Logging)

Логи — это записи о том, что произошло в системе. События, ошибки, предупреждения, информационные сообщения.

1.1 Уровни логирования

Уровень	Когда использовать	Пример
ERROR	Система не может выполнить операцию. Требуется вмешательство	«Не удалось сохранить заказ в БД»

Уровень	Когда использовать	Пример
WARN	Что-то пошло не так, но система самостоятельно справилась	«Таймаут при вызове внешнего API, повтор через 1 сек»
INFO	Нормальные бизнес-события	«Пользователь 123 создал заказ 456»
DEBUG	Для отладки. Включать только по необходимости	«Запрос к БД: SELECT * FROM users»

Золотое правило: В production должны быть INFO и выше. DEBUG — только для локальной разработки или при расследовании инцидента.

1.2 Структурированное логирование

Плохой лог: `"User 123 created order 456"` — невозможно парсить.

Хороший лог (JSON):

json

```
{
  "timestamp": "2025-12-22T14:30:00Z",
  "level": "INFO",
  "service": "order-service",
  "user_id": 123,
  "order_id": 456,
  "message": "Order created successfully"
}
```

Почему JSON? Потому что его легко парсить программами. Вы можете фильтровать, группировать, искать, строить дашборды.

1.3 Correlation ID

Самая частая проблема в микросервисах: запрос проходит через 10 сервисов, а понять, какой лог к какому запросу относится, невозможно.

Решение: **Correlation ID** — уникальный идентификатор запроса, который прокидывается через все сервисы.

Поток Correlation ID:

Клиент → API Gateway (генерирует X-Request-Id)
API Gateway → Сервис А (передаёт X-Request-Id)
Сервис А → Сервис Б (передаёт X-Request-Id)
Сервис А → БД (пишет X-Request-Id в комментарий)

Как реализовать:

- На входе: генерируем UUID.
- На выходе во все логи: включаем correlation_id.
- При вызове другого сервиса: передаём в заголовке (например, X-Request-Id).

Результат: Все логи одного запроса можно найти по correlation_id.

1.4 Централизованное хранение логов (ELK Stack)

Когда у вас 50 сервисов, логи на каждом сервере — это ад. Нужно централизованное хранилище.

ELK Stack:

- **E**lasticsearch — поисковый движок, хранит логи
- **L**ogstash — собирает, парсит, обогащает логи
- **K**ibana — визуализация, дашборды, поиск

Как работает:

1. Приложение пишет логи в stdout (консоль)
2. Filebeat (агент на сервере) отправляет логи в Logstash
3. Logstash парсит, обогащает и отправляет в Elasticsearch
4. Kibana позволяет искать и анализировать

Затраты: Elasticsearch требует ресурсов. На старте можно использовать облачные решения (ELK as a Service).

Часть 2. Метрики (Metrics)

Логи отвечают на вопрос «что произошло?». Метрики — на вопрос «как часто это происходит и как долго?».

2.1 Виды метрик

Счётчик (Counter): Только увеличивается. *Пример:* количество обработанных запросов, количество ошибок.

Таймер (Timer): Измеряет длительность и количество операций. *Пример:* время ответа API, время выполнения SQL-запроса.

Гистограмма (Histogram): Распределение значений по корзинам. *Пример:* сколько запросов уложилось в 10 мс, 50 мс, 100 мс, 500 мс, 1 с.

2.2 RED и USE — два шаблона метрик

RED (Rate, Errors, Duration): для мониторинга сервисов с точки зрения пользователя.

- **Rate** — количество запросов в секунду
- **Errors** — доля ошибок (5xx, таймауты)
- **Duration** — время ответа (p50, p90, p99)

USE (Utilization, Saturation, Errors): для мониторинга ресурсов.

- **Utilization** — загрузка ресурса (CPU, память, диск)
- **Saturation** — очередь запросов (длина очереди, количество ожидающих)
- **Errors** — ошибки на уровне ресурса

2.3 Prometheus

Стандарт де-факто для сбора метрик.

Как работает:

1. Приложение (или специальный exporter) открывает HTTP-эндпоинт `/metrics`
2. Prometheus «вытягивает» метрики с заданным интервалом
3. Метрики хранятся в TSDB (Time Series Database)

Exporter'ы для популярных систем: Node Exporter (системные метрики сервера), Blackbox Exporter (проверка доступности HTTP/ICMP), MySQL/PostgreSQL Exporter.

Часть 3. Трейсинг (Tracing)

Логи и метрики не позволяют увидеть полную картину запроса, проходящего через несколько сервисов. Трейсинг решает эту проблему.

Терминология:

- **Trace** — полный путь запроса от начала до конца.
- **Span** — одна операция в трейсе (HTTP-вызов, запрос к БД, обработка).
- **Parent-child relationship** — связь между спанами.

3.1 OpenTelemetry

Стандарт для сбора трейсов, метрик и логов (единый API).

Аннотация кода:

python

```
from opentelemetry import trace

tracer = trace.get_tracer(__name__)

@tracer.start_as_current_span("process_payment")
def process_payment(user_id, amount):
    with tracer.start_as_current_span("validate_payment"):
        # валидация
    with tracer.start_as_current_span("call_payment_gateway"):
        # вызов внешнего API
```

3.2 Инструментация без вторжения в код

- **Jaeger, Zipkin** — визуализация трейсов.
- **Service Mesh** (Istio, Linkerd) — автоматически добавляет трейсинг для всех вызовов между сервисами.
- **Агенты** (Dynatrace, Datadog) — подключаются к JVM/Python и автоматически инструментируют код.

Часть 4. Дашборды и алерты

Собирать логи, метрики и трейсы мало. Нужно на них реагировать.

4.1 SLI, SLO, SLA

- **SLI (Service Level Indicator)** — метрика, отражающая качество сервиса (время ответа, доступность)
- **SLO (Service Level Objective)** — целевое значение SLI (99.9% запросов за 200 мс)
- **SLA (Service Level Agreement)** — юридически закреплённое обещание клиенту (с компенсациями)

Пример SLO:

- Доступность: 99.9% (простых не более 8.76 часов в год)
- Задержка: 95% запросов за 500 мс, 99% за 2 секунды
- Ошибки: не более 0.1% ответов с 5xx

4.2 Error budget

Формула: Error Budget = 100% — SLO

Если SLO = 99.9%, Error Budget = 0.1% времени.

Как использовать: Пока бюджет не потрачен — можно рисковать новыми релизами и экспериментами. Как только бюджет исчерпан — останавливаем изменения и чиним стабильность.

4.3 Алерты

Правила хороших алертов:

- **Алерт должен быть действием.** Не «метрика X превысила порог», а «позвони инженеру».
- **Алертов должно быть мало.** Если их 50 в день, их никто не читает.
- **Важно не только оповещение, но и контекст.** В алерте должна быть инструкция: что делать, куда смотреть.

Уровни алертов:

- **P1 (Critical):** всё сломалось → звоним всем (SMS, звонок).
- **P2 (High):** что-то важное не работает → пишем в чат, но не будим ночью.
- **P3 (Medium):** странно, но работает → в лог, разберёмся утром.

Итог главы 6

1. **Логи** — запишите всё, что происходит. Обязательно со структурированием (JSON) и correlation ID.
2. **Метрики** — измеряйте производительность и надёжность. Используйте RED для сервисов, USE для ресурсов.
3. **Трейсинг** — свяжите запросы между сервисами, чтобы находить узкие места.
4. **Дашборды** — показывайте ключевые показатели.
5. **Алерты** — реагируйте только на критическое. Остальное — в утренний разбор.

Что делать прямо сейчас:

1. Внедрите correlation ID во все сервисы.
2. Настройте централизованное логирование (ELK или облачное решение).
3. Соберите метрики через Prometheus и постройте дашборды в Grafana.
4. Определите SLO для ключевых сценариев.
5. Настройте алерты, которые не будут шуметь ночью по пустякам.

Следующая глава — «Интеграционные паттерны» — покажет, как соединять сервисы, чтобы они не падали друг от друга.

Глава 7. Интеграционные паттерны

«Соединить два сервиса легко. Соединить двадцать сервисов так, чтобы они не убили друг друга, — это искусство»

В 2021 году я пришёл в проект, где было 15 микросервисов. Они общались друг с другом синхронно через REST. Запрос пользователя запускал цепочку из 7 вызовов. Время ответа было 2 секунды. При падении одного сервиса всё падало.

Мы перешли на асинхронные интеграции и паттерны устойчивости. Время ответа упало до 300 мс. Система переживала падения отдельных сервисов без остановки.

В этой главе — как соединять сервисы, чтобы они не убивали друг друга. Очереди, маршрутизация, трансформация, CDC, API Gateway, BFF. И когда что выбирать.

Часть 1. Messaging (очереди и брокеры)

1.1 Point-to-Point (очередь)

Каждое сообщение из очереди забирает **один** потребитель. **Пример:** отправка email — не нужно дублировать.

1.2 Publish-Subscribe (топик)

Сообщение получают **все** подписчики топика. **Пример:** событие «пользователь зарегистрирован» → один сервис отправляет приветственное письмо, другой — начисляет бонусы, третий — обновляет CRM.

1.3 Kafka vs RabbitMQ

Единого ответа «что лучше» нет, нужен анализ сценария.

Критерий	Kafka	RabbitMQ
Модель	Pub-Sub (логирование, события)	Очереди (задачи, RPC)
Хранение сообщений	Да (дни/недели)	Нет (удалил — прочитал)
Порядок сообщений	В партиции	Не гарантирован
Пропускная способность	Очень высокая (100k+ msg/sec)	Высокая (20-50k msg/sec)
Сложность настройки	Высокая	Средняя
Случаи использования	Аудит, CDC, аналитика, стриминг	Задачи фона, RPC, распределение работы

Часть 2. Паттерны брокеров сообщений

2.1 Competing Consumers (конкурирующие потребители)

Несколько потребителей читают из одной очереди. Сообщение достаётся одному. Помогает параллельной обработке. **Важно:** обеспечивать идемпотентность.

2.2 Dead Letter Channel (почта мёртвых писем)

Сообщение не удалось обработать через N попыток. Такие сообщения не удаляются, а складываются в отдельную очередь (Dead Letter Queue). Раз в день разбираем — чиним баги, дообрабатываем вручную.

2.3 Poison Message (ядовитое сообщение)

Сообщение, которое ломает обработку всегда (например, null вместо JSON). Dead Letter Channel не помогает — оно будет возвращаться в основную очередь и ломать обработчики снова и снова. **Как лечить:** валидировать сообщение при входе и при ошибке перемещать в отдельную очередь для ручного анализа, не возвращая обратно.

2.4 Idempotent Receiver (идемпотентный получатель)

При повторной обработке одного и того же сообщения результат не меняется. **Реализация:** сохраняем ID обработанных сообщений в БД; перед обработкой проверяем, не было ли уже такого.

2.5 Idempotent Producer (идемпотентный отправитель)

В Kafka: при повторной отправке сообщения (из-за сетевого сбоя) оно не дублируется.

Часть 3. Транзакционность в сообщениях

3.1 2PC (Two-Phase Commit, двухфазный коммит)

Классический протокол распределённых транзакций. **Фаза 1 (prepare):** все участники подтверждают готовность. **Фаза 2 (commit/rollback):** если все готовы — commit, иначе — rollback.

Минусы: медленно, coordinator — единая точка отказа.

3.2 Saga (паттерн для распределённых транзакций)

Разбиваем транзакцию на шаги. Каждый шаг имеет компенсирующее действие. Если шаг не удался — запускаем компенсации в обратном порядке.

Orchestration (дирижёр): центральный координатор диктует участникам, что делать.

Choreography (хореография): участники общаются через события, без центра.

3.3 Outbox + Polling

Надёжная публикация событий из сервиса. Сервис пишет событие в свою БД (в рамках транзакции). Отдельный процесс читает из таблицы outbox и публикует в очередь. После публикации помечает как отправленное.

Где использовать: Непрерываемая публикация событий (платежи, заказы). Ни одно событие не должно потеряться.

Часть 4. Маршрутизация и трансформация

4.1 Маршрутизация сообщений

Content-Based Router: по содержимому сообщения определяет, в какой канал отправить.

Splitter: разбивает сообщение на несколько (одна пачка — в последовательные задачи).

Aggregator: ждёт несколько сообщений и объединяет в одно.

4.2 Трансформация сообщений

Message Translator: преобразует формат (XML → JSON).

Envelope Wrapper: добавляет заголовки.

Content Enricher: обогащает недостающими данными.

Claim Check: тело большого сообщения сохраняем в хранилище, а в очередь кладём ссылку.

Часть 5. Паттерны для API

5.1 API Gateway

Единая точка входа для всех клиентов. **Умеет:**

- Маршрутизация запросов к нужному сервису
- Проверка аутентификации, авторизации (один раз на всех!)
- Rate limiting (защита от DDoS), агрегирование ответов от нескольких сервисов
- Логирование, метрики, кэширование

5.2 BFF (Backend for Frontend)

Отдельный API Gateway для каждого клиента (веб, мобильное приложение, партнёр). У каждого клиента свои требования.

Плюсы: каждый BFF независим, не нужно согласовывать общий API.

Минусы: дублирование логики, больше кода.

5.3 Gateway Aggregation vs Offloading

Агрегация: Gateway собирает ответы от нескольких сервисов (один запрос клиента → много запросов к бэкендам). Offloading: проверка аутентификации, логирование, rate limiting выносятся на Gateway → уменьшение нагрузки.

Часть 6. CDC (Change Data Capture)

Отслеживание изменений в БД и превращение их в события.

Зачем: синхронизация БД с внешними системами и аналитическими хранилищами.

Методы CDC:

- **Trigger-based:** триггеры на таблицах (для нечастых изменений). Риск замедления запросов на вставку.
- **Log-based (лучший):** реплицируем WAL (Write-Ahead Log). Не влияет на производительность, ловит даже удаления.

Инструменты: Debezium, Kafka Connect, AWS DMS. Пример: Debezium слушает WAL PostgreSQL, изменения попадают в Kafka → analytics в ClickHouse.

Итог главы 7

1. **Типы очередей:** point-to-point (одна задача — одному) и pub-sub (читают все).
2. **Kafka или RabbitMQ?** Kafka — для потоков, аналитики, истории; RabbitMQ — для задач, RPC.
3. **Dead Letter, Poison Message** — как не потерять и не зациклить сообщения.
4. **Saga** — вместо 2PC, если нужна распределённая транзакция, но можно с компенсациями.
5. **API Gateway / BFF** — единая точка входа, разгрузка аутентификации, rate limiting.
6. **CDC (Change Data Capture)** — репликация изменений БД в реальном времени для аналитики.

Что делать прямо сейчас:

1. Составьте карту всех интеграций в проекте.
2. Для каждой определите: синхронная или асинхронная, нужны ли гарантии доставки, идемпотентность.
3. Выберите брокер (скорее всего Kafka для событий и RabbitMQ для задач).
4. Настройте CDC, если синхронизируете аналитику.

Избегайте синхронных цепочек — это путь к падениям. Внедряйте очереди как можно раньше, пока система маленькая и отказоустойчивость легко добавить.

Следующая глава — «Архитектурные стили» — покажет, когда монолит лучше микросервисов, а когда микросервисы всё-таки нужны.

Глава 8. Архитектурные стили

«Выбрать архитектурный стиль — это как выбрать инструмент для операции. Скальпель хорош для одного, топор — для другого. Но если вы возьмёте топор для удаления аппендицита, пациент не выживет»

Самый частый спор в IT: монолит или микросервисы?

Я участвовал в этом споре десятки раз. И каждый раз убеждаюсь: правильный ответ — «зависит от контекста». Но за этими словами скрывается конкретный анализ: размер команды, темпы роста, требования к отказоустойчивости, бюджеты.

В этой главе — подробный разбор архитектурных стилей. Когда они хороши, когда плохи, и как не ошибиться с выбором.

Часть 1. Монолит

Самый древний, самый понятный, самый недооценённый стиль.

1.1 Что такое монолит

Одно приложение, которое делает всё. Один код, одна база данных, один деплой.

Виды монолитов:

Вид	Описание	Плюсы	Минусы
Однослойный (Big Ball of Mud)	Вся логика в одном слое. Контроллеры говорят с БД напрямую	Быстро начать	Ад поддерживать

Вид	Описание	Плюсы	Минусы
Слоеный (Layered)	Разделение на слои: контроллеры → сервисы → репозитории	Чёткая ответственность	Может быть избыточно для малого проекта
Модульный (Modular)	Чётко выделены модули, каждый модуль имеет свои границы	Уже почти микросервисы, но без сетевых вызовов	Требует дисциплины

1.2 Когда монолит — правильный выбор

1. **Маленькая команда (до 10 человек).** Микросервисы — это инфраструктурная сложность на 3+ команды. 3 разработчика не потянут кластер Kafka.
2. **Простой продукт (CRUD).** Интернет-магазин с 10 000 товаров и корзиной — отличный монолит.
3. **Быстрый старт.** В стартапе скорость выхода на рынок важнее масштабируемости.
4. **Нет потребности масштабировать части по отдельности.** Если вся система грузится одинаково — незачем делить.
5. **Транзакции (ACID).** В монолите транзакции простые и быстрые. В микросервисах — Saga и компенсации.

1.3 Плюсы монолита

- **Простая разработка:** один репозиторий, нет сетевых вызовов между модулями
- **Простое тестирование:** интеграционные тесты запускаются локально
- **Простой деплой:** один артефакт, один сервер
- **Простая отладка:** один процесс, один стек

1.4 Минусы монолита

- **Ограничение масштабирования:** всё масштабируется целиком, даже если нужна только одна функция
- **Связанность команд:** 10 команд не могут работать над одним монолитом
- **Деплой страха:** даже маленькое изменение требует деплоя всего приложения
- **Технологическая статичность:** сложно внедрить новый стек, язык

1.5 История: монолит, который работал годами

Мы как-то делали монолит для системы опросов. Код был страшный. Тесты — никакие. Документация — в голове у одного разработчика.

Но система работала 3 года без единого падения.

Почему? Потому что у нас была:

- **Одна команда (5 человек)** — все знали весь код
- **Низкая нагрузка (100 RPS)** — масштабирование не требовалось
- **Стабильные требования** — ничего не менялось годами

Если у вас похожие условия — не усложняйте.

Часть 2. Микросервисы

Мечта, которая часто превращается в кошмар.

2.1 Что такое микросервисы

Система, состоящая из маленьких, независимых сервисов. Каждый сервис имеет свою базу данных и своё бизнес-логику. Сервисы общаются через сеть (по HTTP, gRPC или через очереди).

Признаки хороших микросервисов:

- **Независимый деплой:** можно обновить один сервис, не трогая другие
- **Децентрализация данных:** у каждого сервиса своя база
- **Автономность команды:** одна команда — один сервис
- **Отказоустойчивость:** упал один сервис — остальные живут

2.2 Когда микросервисы нужны

1. **Большая команда (50+ человек).** Микросервисы позволяют делить команды и не наступать друг другу на пятки.
2. **Разные циклы релизов.** Каталог обновляется каждый день, платёжный модуль — раз в месяц.
3. **Независимое масштабирование.** Платежи грузятся в 10 раз сильнее каталога.
4. **Разные технологии.** Каталог на Go (быстрый), отчёты на Python (удобный).

2.3 Плюсы микросервисов

- **Независимое масштабирование:** только нужные компоненты
- **Технологическая гибкость:** каждый сервис может использовать свой стек
- **Изоляция отказов:** один сервис упал — остальные работают
- **Маленький код:** легче понять и тестировать
- **Независимые релизы:** можно обновлять часть системы без остановки всего

2.4 Минусы микросервисов

- **Сложность разработки:** сети, версионирование, согласованность
- **Сложность тестирования:** интеграционные тесты надо поднимать всё окружение
- **Сложность эксплуатации:** кластер из 50 сервисов нужно мониторить
- **Сетевые задержки:** вместо вызова памяти — вызов по сети
- **Распределённые транзакции:** больше нельзя положиться на ACID
- **Бог сервисов:** один сервис, от которого всё зависит

2.5 История: микросервисы, которые спасли проект

В Ростелекоме мы делали «Выборы 2.0». 300+ микросервисов, 7 команд, 3 подрядчика. Нагрузка — в чёрную пятницу чилийскую.

Микросервисы были необходимы:

- Команда подрядчиков могла независимо обновлять свой сервис
- При падении сервиса уведомлений, голосование продолжалось
- Узкие места масштабировали отдельно

Да, было сложно. Но мы бы не справились с монолитом.

Часть 3. Стратегия миграции: Strangler Pattern

Переезд с монолита на микросервисы нельзя сделать за день. Нужна стратегия.

Шаги:

1. **Выделяем первый сервис.** Выбираем самый независимый модуль (например, уведомления).
2. **API Gateway.** Весь внешний трафик идёт через шлюз, который отправляет одни запросы в монолит, другие — в новые сервисы.
3. **Постепенная замена.** По одному сервису вырезаем из монолита.
4. **Старый монолит умирает.** Когда все функции вынесены, монолит отключается.

Сроки: Миграция занимает годы. И это нормально.

Часть 4. Модульный монолит (золотая середина)

4.1 Что это

Вы делаете монолит, но с чёткими границами между модулями. Модули не зависят от внутренностей друг друга.

Как это выглядит в коде:

python

```python
# Вместо этого:
def create_order(user_id, items):
    user = User.objects.get(id=user_id)  # модуль пользователей
    # ... логика заказа

# Делаем так:
from user_module import get_user
def create_order(user_id, items):
    user = get_user(user_id)  # вызов через интерфейс
```

4.2 Плюсы модульного монолита

- Просто начинать: вся разработка в одном репозитории

- Масштабируемость: по командам делить сложно
- Прозрачные границы: «модульный монолит» заставляет явно определить интерфейсы
- Простота эксплуатации: один сервер, один деплой

Когда использовать: 90% проектов. Если вы читаете эту книгу и не уверены — начните с модульного монолита.

Часть 5. EDA (Event-Driven Architecture)

5.1 Что это

Сервисы не вызывают друг друга напрямую, а публикуют события. Другие сервисы подписываются на интересующие их события.

Синхрон (request-response):

text

Клиент → API Gateway → Сервис А → Сервис Б → База

Асинхрон (event-driven):

Клиент → API Gateway → Сервис А
↓ (событие)
Kafka
↓ (подписка)
Сервис Б

5.2 Когда выбирать EDA

- Нетребовательная задержка (можно 1–2 секунды)
- Слабые связи между сервисами
- Высокая нагрузка, которую нужно сглаживать очередями
- Требуется аудит всех изменений

5.3 Гарантии доставки

- **At-most-once**: сообщение может потеряться, не больше одной доставки. Применять для некритичных данных.

- **At-least-once**: сообщение не потеряется, но может дублироваться (Kafka, RabbitMQ). Применять для критичных данных в сочетании с идемпотентностью.
- **Exactly-once**: теоретический идеал. На практике at-least-once + идемпотентность.

Совет: Никогда не полагайтесь на exactly-once. Всегда проектируйте идемпотентность.

Часть 6. Лямбда-архитектура и каппа-архитектура

6.1 Лямбда-архитектура

Два слоя обработки данных: пакетный (batch) и скоростной (speed). Слой сервировки (serving) объединяет результаты.

Плюсы: лучшее из двух миров, точность пакетной обработки и скорость потоковой.

Минусы: сложность (две параллельные кодовые базы), дублирование данных, дорого поддерживать.

6.2 Каппа-архитектура

Упрощение лямбды. Убираем пакетный слой, оставляем только потоковую обработку.

Плюсы: одна кодовая база, дешевле поддерживать.

Минусы: возможны задержки при пересмотре истории.

Часть 7. Событийно-ориентированная архитектура популярно и сравнение стилей

Стиль	Сложность	Масштабируемость	Изменяемость	Когда выбирать
Монолит	Низкая	Низкая	Низкая	Малая команда, простой продукт, быстрый старт
Модульный монолит	Средняя	Средняя	Средняя	90% проектов — это ваш выбор по умолчанию
Микросервисы	Высокая	Высокая	Высокая	Большая команда, разные циклы релизов, независимое масштабирование
EDA	Высокая	Высокая	Высокая	Асинхронная связь, слабые зависимости, высокая нагрузка

Итог главы 8

1. **Монолит** — не стыдно. Начинайте с него, если не уверены.
2. **Модульный монолит** — идеальная золотая середина для большинства проектов.

3. **Микросервисы** — только если вы готовы платить сложностью.
4. **EDA** — для асинхронных систем.
5. **Strangler Pattern** — стратегия миграции с монолита.

Что делать прямо сейчас:

1. Определите размер вашей команды (сейчас и через год).
2. Определите темп изменений в разных частях системы.
3. Если есть сомнения — начните с модульного монолита.
4. Чётко определите границы модулей (bounded contexts).
5. Если через год станет ясно, что нужны микросервисы — начните по одному вырезать (Strangler Pattern).

Следующая глава — «Паттерны масштабирования» — покажет, как не уронить систему, когда пользователей станет в 100 раз больше.

Глава 9. Паттерны масштабирования

«Добавить ещё один сервер — это не магия. Это способ отложить неизбежное. Настоящее масштабирование начинается там, где серверы перестают помогать»

В 2022 году мы готовились к пиковой нагрузке. Я сидел с архитекторами и считал: сколько серверов нужно, чтобы выдержать 1 миллион транзакций в час.

«Добавим ещё 10 серверов», — предложил один. «Нужно 50», — возразил другой.

Мы добавили 30. На следующий день я смотрел на графики: серверы были загружены на 40%. Мы перестраховались. Но если бы нагрузка выросла в 3 раза, 30 серверов бы не спасли — упала бы база данных.

В этой главе — паттерны масштабирования, которые работают. Кэширование, очереди, балансировка, rate limiting. И главное — как понять, что пора их применять.

Часть 1. Кэширование

Самый эффективный способ ускорить систему. Данные, которые часто читаются и редко меняются, хранятся в быстром хранилище (обычно в оперативной памяти).

1.1 Виды кэшей

Вид кэша	Где хранится	Скорость	Когда использовать
Клиентский (браузер)	На устройстве пользователя	Мгновенно	Статические ресурсы (CSS, JS, изображения)

Вид кэша	Где хранится	Скорость	Когда использовать
CDN (Content Delivery Network)	Географически распределённые серверы	Очень быстро	Контент, общий для всех пользователей
API Gateway	В шлюзе	Очень быстро	Ответы API, которые можно закэшировать для многих пользователей
Сервисный (In-Memory)	В памяти сервера приложения	Наносекунды	Результаты запросов к БД, сессии
Распределённый (Redis, Memcached)	Отдельный клавер кэша	Микросекунды	Данные, которые должны быть доступны с любого сервера

1.2 Стратегии кэширования

Cache-Aside (Lazy Loading): приложение само проверяет кэш, при промахе загружает данные в БД и кладёт в кэш.

Плюсы: простота, контроль. *Минусы:* первый запрос всегда медленный.

Read-Through: кэш сам умеет загружать данные из БД.

Плюсы: код проще. *Минусы:* меньше контроля.

Write-Through: запись сначала идёт в кэш, затем синхронно в БД.

Плюсы: кэш всегда свежий. *Минусы:* задержка записи увеличивается.

Write-Back (Write-Behind): запись только в кэш, в БД — асинхронно.

Плюсы: очень быстрая запись. *Минусы:* риск потери данных при падении кэша.

Write-Around: запись сразу в БД, кэш обновляется при чтении.

Плюсы: экономия кэша для редко читаемых данных. *Минусы:* чтение свежих данных будет медленным.

Когда что выбрать:

- **Cache-Aside** — универсальный вариант по умолчанию.
- **Write-Through** — когда важна согласованность кэша и БД.
- **Write-Back** — когда скорость записи критична, а потеря данных допустима (логи, аналитика).

1.3 Вытеснение кэша (Eviction Policy)

Когда кэш заполнен, нужно решать, что удалить.

- **LRU (Least Recently Used):** удаляем давно неиспользуемые
- **LFU (Least Frequently Used):** удаляем редко используемые
- **FIFO (First In First Out):** удаляем старые
- **TTL (Time To Live):** удаляем через фиксированное время (проще всего)

1.4 Инвалидация кэша (проклятие распределённых систем)

Данные в БД изменились — кэш устарел. Нужно его обновить или удалить.

Проблема: трудно понять, какие ключи в кэше зависят от изменённых данных.

Стратегии:

- **TTL:** через N секунд кэш устаревает сам (самая простая).
- **Write-Through:** обновляем кэш при записи в БД.
- **Pub/Sub:** сервис, меняющий данные, публикует событие «обнови кэш».
- **Теги в кэше:** храним теги (например, `user:123`), при изменении удаляем всё с тегом.

1.5 Проблемы кэширования

Thundering Herd (эффект стада): кэш истек, и тысячи запросов одновременно летят в БД.

Решение: блокировка при обновлении кэша (один запрос обновляет, остальные ждут).

Stale cache (устаревший кэш): данные изменились, но в кэше старое.

Решение: TTL или активная инвалидация.

Cache stampede: то же, что Thundering Herd + stale.

Решение: probabilistic early expiration.

Часть 2. Очереди (Queues)

Очереди сглаживают пики нагрузки. Запросы не обрабатываются мгновенно, а складываются в очередь и разбираются в темпе обработчика.

2.1 Зачем ставить очередь

- **Сглаживание пиков:** если запросов слишком много, очередь растёт, но система не падает.
- **Разделение производителя и потребителя:** производитель не ждёт потребителя. Потребитель может упасть и подняться — запросы не потеряются.
- **Балансировка нагрузки между потребителями (Competing Consumers).**

2.2 Rate Limiting на потребителе

Обработчик не должен работать быстрее, чем очередь успевает отдавать. Можно ограничить скорость через лимитеры, чтобы не генерировать излишнюю нагрузку на БД.

2.3 Dead Letter Queue (DLQ)

Сообщения, которые не удалось обработать, перемещаются в отдельную очередь для разбора. Не теряем сообщения, не зацикливаемся на заведомо битых.

Часть 3. Балансировка нагрузки (Load Balancing)

Распределение запросов между несколькими серверами.

3.1 Алгоритмы балансировки

Алгоритм	Описание	Когда использовать
Round Robin	По очереди	Серверы одинаковой мощности
Least Connections	Тому, у кого меньше активных соединений	Запросы разной длительности
Least Time	Тому, у кого меньше время ответа	Задержки критичны
IP Hash	Один IP всегда на один сервер	Нужны липкие сессии
Random	Случайный выбор	Простота

3.2 Где размещать балансировку

- **DNS-балансировка:** несколько IP на домен. *Плюс:* просто. *Минус:* нет учёта нагрузки, кэширование DNS.
- **L3/L4 балансировка (по IP и портам):** дешево, но не видит содержимого запроса.
- **L7 балансировка (по HTTP):** может маршрутизировать по URL, заголовкам, кукам.

3.3 Липкие сессии (Sticky Sessions)

Пользователь всегда попадает на один и тот же сервер. Нужно для хранения сессии в памяти сервера.

Минусы: если сервер упал, пользователь теряет сессию. Лучше хранить сессии в Redis.

3.4 Балансировка с учётом загруженности (Backpressure)

Балансировщик получает метрики с серверов (CPU, память, очередь) и направляет запросы на менее загруженные.

Часть 4. Защита от перегрузок (Rate Limiting и Throttling)

4.1 Rate Limiting

Ограничение количества запросов от одного клиента за единицу времени.

Зачем: защита от DDoS, от соседа по базе данных, тарификация (бесплатный тариф — 100 запросов/день).

Алгоритмы:

- **Token Bucket:** в ведре токены. Каждый запрос забирает токен. Токены добавляются с фиксированной скоростью. Позволяет кратковременные всплески.
- **Leaky Bucket:** очередь фиксированного размера. Запросы обрабатываются с постоянной скоростью. Сглаживает идеально.
- **Fixed Window:** в окне (например, 1 минута) не более N запросов. На границах окон возможны всплески (2N запросов за 2 окна).
- **Sliding Log:** храним лог времени запросов и отсекаем старые. Точный, но затратный по памяти.
- **Sliding Window (рекомендуемый):** комбинация Fixed Window + Sliding Log.

4.2 Где размещать Rate Limiting

- **На API Gateway:** для всех клиентов — простейшее централизованное решение.
- **В сервисе:** для тонкой настройки по ключам (API key + user_id).
- **На уровне базы данных:** лимит соединений, активных транзакций.

4.3 Throttling

Уменьшение скорости обработки без жёстких лимитов. Например, при превышении нагрузки замедляемся, но не отказываем.

Итог главы 9

1. **Кэширование** — самый эффективный способ ускорить систему. Начинайте с Cache-Aside и TTL.
2. **Очереди** — сглаживают пики и развязывают производителей и потребителей.
3. **Балансировка** — распределяет нагрузку между серверами.
4. **Rate Limiting** — защищает систему от перегрузок.

Что делать прямо сейчас:

1. Составьте список «тяжёлых» запросов. Внедрите кэш для тех, у кого read-heavy.
2. Внедрите TTL на все кэши (даже если кажется, что данные меняются редко).
3. Настройте rate limiting на API Gateway.
4. Раз в квартал проводите нагрузочное тестирование.

Следующая глава — «Платформы и инфраструктура» — расскажет о контейнеризации, оркестрации и облаках.

Глава 10. Паттерны доступности и отказоустойчивости

«Системы не падают из-за того, что что-то пошло не так. Они падают из-за того, что не было спланировано, что что-то пойдёт не так»

В 2019 году я дежурил в чёрную пятницу. Нагрузка была в 10 раз выше обычной. Я смотрел на дашборды, которые медленно краснели. А потом график ушёл в ноль.

«Всё, — подумал я. — Мы упали».

Но через 20 секунд график вернулся. Система не умерла. Она замедлилась, но выжила. Потому что мы заложили отказоустойчивость. Мы проектировали не «если упадёт», а «когда упадёт».

В этой главе — паттерны, которые спасают системы от смерти. Репликация, георазпределение, DRP (Disaster Recovery Plan). И немного историй о том, как я не спал ночами, восстанавливая данные из бэкапов после падения дискового массива.

Часть 1. Репликация (продолжение)

1.1 Синхронная репликация

Клиент ждёт, пока данные запишутся на лидера и на N реплик. Высокая надёжность, но задержка растёт, а доступность падает (если реплика недоступна, запись не пройдёт).

Где используется: в системах, где потеря данных недопустима (банки, платёжные системы).

1.2 Асинхронная репликация

Клиент получает ответ после записи на лидера. Реплики обновляются в фоне. Высокая доступность и скорость, но возможна потеря непереданных изменений при падении лидера.

Где используется: везде, где небольшая потеря данных допустима (логи, лайки, комментарии).

1.3 Кворумная репликация

Прогрессивная штука. При N репликах:

- Запись успешна при подтверждении от W реплик
- Чтение успешно при ответе от R реплик

Условие согласованности: W + R > N. Тогда читатель всегда увидит хотя бы одну свежую запись.

Пример для N=3 (реплик):

- W=2, R=2 → 2+2=4 > 3 — согласованность обеспечена
- W=1, R=1 → 1+1=2 < 3 — возможны расхождения

1.4 Цепочная репликация

Реплики выстроены в цепочку: A → B → C. Запись идёт в A, A реплицирует на B, B — на C.

Плюсы: мало сетевого трафика (N-1 связей вместо N^2).

Минусы: при отказе любого звена цепочка перестраивается.

Где используется: распределённые БД (Cassandra, ScyllaDB при определённых настройках).

1.5 Топологии репликации

- **Звезда (star):** центральный узел рассылает изменения всем остальным. Простота, центральный узел — узкое место и точка отказа.
- **Кольцо (ring):** каждый узел соединён с соседями. Простота, при отказе одного узла кольцо разрывается.
- **Полносвязная (fully connected):** каждый с каждым (N(N-1)/2 связей). Максимальная надёжность, квадратичный рост соединений.

Часть 2. Геораспределение (Geo-Distribution)

2.1 Зачем распределять систему географически

Если ваши пользователи по всему миру, нужно, чтобы серверы были рядом с ними (снижение задержки). Дата-центр в Москве не спасёт пользователя во Владивостоке.

Причины:

- **Задержка:** свет в оптоволокне — 200 000 км/с. Москва – Владивосток → 70 мс RTT (туда-обратно). Это заметно.
- **Доступность:** при землетрясении в Калифорнии сервера во Франкфурте могут остаться в живых.
- **Юридические требования:** данные российских пользователей должны храниться в РФ.

2.2 GeoDNS и Anycast

GeoDNS возвращает IP-адрес сервера, ближайшего к клиенту (по геолокации).

Anycast: один IP-адрес объявляется из нескольких точек мира. Маршрутизаторы направляют клиента в ближайшую точку.

2.3 Active-Active vs Active-Passive

Стратегия	Как работает	Задержка	Доступность	Сложность
Active-Passive	Один дата-центр работает, второй спит	Низкая (в активном)	Средняя (переключение при падении)	Низкая
Active-Active	Оба работают, нагрузка	Низкая (везде)	Высокая	Высокая

Стратегия	Как работает	Задержка	Доступность	Сложность
	распределяется			

Active-Passive проще: вы всегда знаете, где данные. Но второй дата-центр простаивает. При падении первого нужно переключить трафик на пассивный (от минут до часов). Данные нужно синхронизировать асинхронно.

Active-Active сложнее: синхронизация данных между дата-центрами — это боль. Конфликты, задержки, сложный мониторинг. Но оба ДЦ не простаивают.

2.4 Latency-Based Routing

Маршрутизация запроса в дата-центр, который даст наименьшую задержку. Это динамический вариант GeoDNS, учитывающий реальные задержки сети (не только географию).

Пример: Пользователь в Индонезии может быстрее достучаться до Австралии, чем до Индии, даже если Индия ближе по карте (качество кабелей, загруженность).

2.5 Проблемы георазпределения

- **Задержки синхронизации.** Чем дальше дата-центры, тем больше latency RTT. Синхронная репликация становится невозможной.
- **Разрешение конфликтов.** Если в двух ДЦ изменили одну запись, нужна стратегия слияния (Last Write Wins, Merge, CRDT).
- **Стоимость.** Два дата-центра, два кластера, два трафика.
- **Сложность отладки.** Запрос пошёл в Европу, подзапрос — в Азию, ответ пришёл через 2 секунды.

Часть 3. DRP (Disaster Recovery Plan)

План действий при катастрофе, когда отказал не один сервер, а весь дата-центр (наводнение, отключение электричества, сбой сети).

3.1 RPO (Recovery Point Objective) и RTO (Recovery Time Objective)

- **RPO** — сколько данных вы готовы потерять. «Мы бэкапимся каждые 5 минут → можем потерять максимум 5 минут данных». Чем меньше RPO, тем дороже (нужны непрерывные бэкапы).
- **RTO** — за сколько времени система должна восстановиться. «Мы должны запуститься за 4 часа». Серверы под рукой, бэкапы отреставрированы.

Пример для вашего проекта:

- RPO = 1 час. Бэкапы БД каждые 60 минут.
- RTO = 4 часа. На восстановление из бэкапа уходит до 4 часов.

3.2 Стратегии бэкапов

- **Полный бэкап (Full Backup):** вся база данных целиком. Долго делать, долго восстанавливать, много места.
- **Инкрементальный бэкап (Incremental):** только изменения с последнего бэкапа. Быстро, мало места. Восстановление: последний полный + цепочка инкрементальных.
- **Дифференциальный бэкап (Differential):** изменения с последнего полного бэкапа. Быстрее восстановления инкрементального.

Рекомендация: полный бэкап раз в неделю + инкрементальные раз в день.

3.3 Место бэкапа (тёплое, горячее, холодное)

- **Горячее место (Hot site):** резервный ДЦ с работающими серверами, данными (Live-синхронизация). Но дороже всего. RTO — минуты.
- **Тёплое место (Warm site):** серверы есть, данные подтягиваются из бэкапов, ПО устанавливается. RTO — часы.
- **Холодное место (Cold site):** есть помещение, стойки, но нет оборудования. Приносим своё, восстанавливаем из лент. RTO — дни.

3.4 DRP план: что должно быть в документе

1. Кто отвечает за восстановление (контакты, роли).
2. RPO / RTO.
3. Конфигурация оборудования (IP-адреса, версии ПО).
4. Схема бэкапов, расположение бэкапов.

5. Пошаговая инструкция по восстановлению (альтернативные ДЦ).
6. Как тестировать план (раз в год устраивать «пожарную тревогу»).

Важно: DRP-план нужно тестировать. Самый частый провал — план есть, а никто его не тестировал, и в реальной катастрофе инструкции не работают, пароли устарели, а сотрудник, который всё помнил, уволился.

Итог главы 10

1. **Виды репликации:** синхронная (надёжно, но медленно), асинхронная (быстро, риск потери данных), кворумная (баланс).
2. **Геораспределение:** GeoDNS для ближайшего сервера, Active-Active vs Active-Passive.
3. **DRP:** RPO (сколько данных потеряем) и RTO (как долго будем восстанавливаться).
4. **Бэкапы:** полные раз в неделю, инкрементальные — каждый день.
5. **Тестирование DRP:** раз в год устраивай «пожарную тревогу».

Что делать прямо сейчас:

1. Определите RPO и RTO для ключевых сервисов.
2. Настройте автоматические бэкапы БД и файлов.
3. Храните бэкапы в другом ДЦ или облаке.
4. Разработайте DRP-план (хотя бы одну страницу с инструкциями).
5. Проверьте DRP-план в тестовом режиме в следующую пятницу.

Следующая глава — «Контейнеризация и оркестрация» — про Docker, Kubernetes и мир, где инфраструктура описывается кодом.

Глава 11. Контейнеризация и оркестрация

«Docker решил проблему “на моей машине работает”. Kubernetes решил проблему “на моих ста серверах работает”. Оба создали новые проблемы, но старые хотя бы исчезли»

В 2015 году я настраивал окружение для нового проекта. Установка Java, Postgres, Redis, очереди, настройка конфигов — заняла три дня. А потом пришёл новый разработчик, и мы повторили всё заново.

Когда мы внедрили Docker, настройка окружения стала занимать 15 минут. Когда внедрили Kubernetes — перестала занимать вообще. Разработчик запускал команду, и через минуту у него работала точная копия продакшена.

В этой главе — про контейнеризацию и оркестрацию. Как упаковать приложение так, чтобы оно работало везде. Как управлять сотнями контейнеров, чтобы они не убивали друг друга.

Часть 1. Docker

1.1 Что такое контейнер

Контейнер — это изолированное окружение для приложения. Оно включает код, runtime, системные инструменты, библиотеки — всё, что нужно для работы.

Чем контейнер отличается от виртуальной машины:

Характеристика	**Контейнер**	**Виртуальная машина**
Изоляция	На уровне ОС (процессы)	Полная (гипервизор)
Размер	Мегабайты (только приложение + зависимости)	Гигабайты (целая ОС)

Характеристика	Контейнер	Виртуальная машина
Запуск	Секунды	Минуты
Плотность	Тысячи на сервер	Десятки на сервер
Совместимость	Требует ту же версию ядра ОС	Любая ОС

Контейнеры легче, быстрее, экономичнее. Но виртуальные машины дают полную изоляцию.

1.2 Dockerfile — как собрать образ

dockerfile

```
# Базовый образ (легковесная ОС с Python)
FROM python:3.11-slim

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

# Копируем код приложения
COPY ./app /app
WORKDIR /app

# Запускаем приложение
CMD ["python", "main.py"]
```

Лучшие практики:

- Используйте официальные образы (python:3.11-slim, а не python:latest — последний может измениться)
- Старайтесь уменьшить количество слоёв (объединяйте RUN-команды)
- Не запускайте процессы от root (создайте непривилегированного пользователя)
- Используйте .dockerignore, чтобы не копировать лишнее (**pycache**, .git, локальные конфиги)

1.3 Слои образов и наследование

Каждая инструкция в Dockerfile создаёт новый слой. Docker кэширует слои, поэтому при повторной сборке пересобираются только изменённые.

Порядок инструкций имеет значение — часто меняющееся в конец:

1. COPY requirements.txt (меняется редко)
2. RUN pip install (долго, но кэшируется)
3. COPY app (меняется часто)

1.4 Docker Compose — оркестрация для одного сервера

Запуск нескольких контейнеров вручную — это боль: помнить команды, указывать сети, пробрасывать порты.

yaml

```yaml
version: '3.8'
services:
  app:
    build: .
    ports:
      - "8000:8000"
    depends_on:
      - postgres
      - redis

  postgres:
    image: postgres:15
    environment:
      POSTGRES_DB: mydb
      POSTGRES_USER: user
      POSTGRES_PASSWORD: pass
    volumes:
      - postgres_data:/var/lib/postgresql/data

  redis:
    image: redis:7-alpine

volumes:
```

postgres_data:

1.5 Сети, volumes, логирование

- **Сети (networks):** контейнеры общаются по именам (в Compose — имя сервиса). Создавайте отдельные сети для изоляции.
- **Volumes:** данные переживают перезапуск контейнера (проброс папки с хоста или именованный volume).
- **Логирование:** docker logs container_name по умолчанию. В production — драйвер для отправки логов в централизованную систему (ELK).

Часть 2. Kubernetes

Docker Compose хорош для одного сервера. Когда серверов 10+ и контейнеров 100+, нужен Kubernetes (k8s).

2.1 Архитектура Kubernetes

Control Plane (Master):

- **API Server:** интерфейс управления кластером (kubectl, UI).
- **etcd:** распределённое хранилище конфигурации.
- **Scheduler:** решает, на какой ноде запустить под.
- **Controller Manager:** следит за состоянием (деплоймент, реплики).

Nodes (Worker):

- **kubelet:** агент на ноде, запускает поды.
- **kube-proxy:** сетевые правила, балансировка.
- **Container Runtime:** Docker, containerd (выполняет контейнеры).

2.2 Основные объекты Kubernetes

Pod: один или несколько контейнеров, которые должны быть вместе (делят сеть, volume). Атомарная единица в K8s.

Deployment: декларативное описание подов (3 реплики, образ и версия). Поддерживает версионирование, откат, rolling update.

Service: стабильный IP и DNS для доступа к подам, балансировка нагрузки (Round Robin; поддерживает сессии с `sessionAffinity: ClientIP`).

Ingress: HTTP-маршрутизация с поддержкой TLS. Внешний трафик → Ingress → Service → Pod.

ConfigMap: хранение конфигурации (переменные окружения, файлы).

Secret: как ConfigMap, но для паролей, токенов, ключей (хранится в зашифрованном виде в etcd).

StatefulSet: для приложений с состоянием (базы данных, Kafka). Даёт стабильные имена подам, стабильное хранилище на диске.

2.3 Пример манифеста

yaml

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: myapp-deployment
spec:
  replicas: 3
  selector:
    matchLabels:
      app: myapp
  template:
    metadata:
      labels:
        app: myapp
    spec:
      containers:
      - name: myapp
        image: myapp:latest
        ports:
        - containerPort: 8000
        env:
        - name: DB_HOST
          value: "postgres-service"
```

2.4 StatefulSet для баз данных

Базы данных не переносят смены IP при перезапуске. StatefulSet даёт:

- Стабильное имя: `postgres-0`, `postgres-1`, `postgres-2`.
- Стабильный сетевой идентификатор: `postgres-0.postgres-headless.default.svc.cluster.local`.
- PersistentVolume, привязанный к поду.

2.5 Операторы для сложных приложений

StatefulSet не хватает для автоматического восстановления отказавшей базы данных. Нужен оператор — приложение внутри K8s, которое управляет другим приложением.

Примеры операторов:

- **Strimzi** — управляет Kafka (создание топиков, ребалансировка партиций).
- **Postgres Operator** — автоматическое переключение при падении мастер-ноды.
- **Prometheus Operator** — управление мониторингом.

Часть 3. Service Discovery, Конфигурация и CI/CD

3.1 Service Discovery

Как поду найти другой под? Без K8s пришлось бы хранить список IP в Consul или ZooKeeper.

В Kubernetes: Service создаёт DNS-запись. Под может обратиться к сервису по имени. При добавлении новых реплик DNS обновляется.

3.2 Конфигурация без секретов в образе

Никогда не встраивайте пароли и токены в Docker-образ — они попадут в репозиторий и registry.

ConfigMap и Secret:

- Монтируются как переменные окружения или файлы в под
- Обновляются без пересборки образа (нужен перезапуск пода)

Внешние секрет-менеджеры:

- **HashiCorp Vault:** динамические секреты, аудит доступа
- **AWS Secrets Manager / GCP Secret Manager:** интеграция с облаком

3.3 CI/CD на стероидах

Без Kubernetes: собрали артефакт, остановили приложение, заменили файлы, запустили.

С Kubernetes: собрали образ, запушили в registry, обновили манифест, `kubectl apply`. K8s сам сделает rolling update и откатится, если проверки не пройдут.

Что даёт CI/CD с K8s:

- **Rolling update:** новый под поднимается, старый умирает — даунтайма нет.
- **Canary:** 10% трафика на новую версию, 90% — на старую. Вытесняем по метрикам.
- **GitOps:** вся конфигурация в Git. При изменении Git → ArgoCD синхронизирует кластер.

Итог главы 11

1. **Docker** — упаковка приложения в контейнер. `Dockerfile`, `docker-compose` для локальной разработки.
2. **Kubernetes** — оркестрация контейнеров. Поды, деплойменты, сервисы, ингрессы.
3. **StatefulSet** — для баз данных и других приложений с состоянием.
4. **Операторы** — управление сложными приложениями внутри K8s.
5. **CI/CD с K8s** — rolling update, канарейки, автоматический откат.

Что делать прямо сейчас:

1. Сконтейнеризуйте приложение — добавьте Dockerfile.
2. Запустите локально docker-compose с зависимостями (БД, кэш).
3. Почитайте про Kubernetes (хотя бы поды и деплойменты).
4. Если работаете в облаке — протестируйте managed Kubernetes (Managed K8s, EKS, GKE, Yandex Cloud).

Следующая глава — «CI/CD: от коммита до продакшена» — автоматизируйте всё, что можно автоматизировать, без страха, что сломается в 3 часа ночи.

Глава 12. CI/CD: от коммита до продакшена

«Если вы деплоите вручную, вы уже делаете это неправильно. Вопрос не в том, упадёт ли у вас прод, а в том, когда это случится и как быстро вы откатитесь»

В 2017 году мы деплоили раз в месяц. В пятницу вечером, после того как все ушли домой, чтобы «никого не беспокоить». Деплой занимал 4 часа. Если что-то шло не так, откат занимал ещё 2 часа. А в понедельник пользователи писали в поддержку: «Почему висит кнопка оплаты?».

В 2020 году мы внедрили CI/CD. Деплой стал занимать 8 минут. Откат — 2 минуты. Ошибки находили до того, как они попадали в прод. Пользователи перестали жаловаться.

В этой главе — о том, как построить конвейер, который сделает ваши релизы скучными и предсказуемыми.

Часть 1. Основы CI/CD

1.1 CI (Continuous Integration)

Разработчики часто мержат код в основную ветку. Каждый мерж вызывает автоматическую сборку и тестирование.

Что даёт CI:

- Ошибки находятся сразу, а не через месяц
- Нет «адских мержей» перед релизом
- Команда всегда работает с актуальной версией кода

1.2 CD (Continuous Delivery)

Код всегда готов к релизу. Последний шаг — нажатие кнопки «деплоить в прод» — выполняется человеком.

Что даёт CD:

- Релиз в любой момент (даже в 3 часа ночи, если горит)
- Бизнес решает, когда выпускать, а не разработка

- Страх перед релизом исчезает

1.3 CD (Continuous Deployment)

Автоматический деплой в прод после прохождения всех проверок. Без кнопки «деплоить».

Что даёт CD:

- Функции попадают к пользователям через минуты после написания кода
- Максимальная скорость доставки

Где нельзя использовать: В авиации, медицине, финансах, космосе — нужна кнопка и подпись ответственного. Там, где ошибка стоит жизней или миллионов.

Часть 2. CI-пайплайн

2.1 Стандартный CI-пайплайн

1. **Линтер** — проверка стиля кода, потенциальных ошибок (ruff, flake8, eslint).
2. **Компиляция** — синтаксические ошибки.
3. **Юнит-тесты** — проверка изолированных модулей и функций.
4. **Интеграционные тесты** — проверка взаимодействия компонентов.
5. **Сборка образа** — создание Docker-образа.
6. **Пуш в registry** — отправка образа в Docker registry.

2.2 Параллельный vs последовательный прогон

Последовательный: каждый шаг ждёт предыдущий. Просто, но медленно.

Параллельный: линтер + юнит-тесты одновременно. Быстро, но сложнее.

Комбинируйте: последовательно на критических этапах, параллельно — на независимых.

2.3 Кэширование артефактов

Между запусками CI можно кэшировать зависимости (например, `pip cache`, `node_modules`). Это ускоряет сборку в 10 раз.

2.4 Качество кода (SonarQube)

SonarQube анализирует:

- **Code Smells:** подозрительные места (дублирование, слишком сложные методы)
- **Уязвимости:** потенциальные дыры (SQL injection, XSS)
- **Покрытие тестами:** сколько строк кода выполняется тестами

Правило: если покрытие упало — пайплайн красный.

Часть 3. CD-пайплайн (доставка и деплой)

3.1 Стратегии деплоя

Rolling (катящийся): по одному заменяем старые поды на новые.

Самый простой, рекомендованный по умолчанию. Минус: старые и новые версии живут рядом.

Blue-Green: два окружения: Blue (старая версия), Green (новая). Тестируем Green, переключаем трафик.

Минус: нужно вдвое больше ресурсов. Зато мгновенный откат: переключили обратно на Blue.

Canary: сначала трафик на новую версию — 1%, 5%, 10%... Если ошибок нет, увеличиваем долю.

Минус: сложность настройки. *Плюс:* минимальные потери при ошибках.

3.2 Approval gate (ручное одобрение)

Human in the loop. Пайплайн доходит до этапа деплоя и останавливается, ждёт нажатия кнопки «approve».

Где нужно: в финансах, на sensitive данных. **Где не нужно:** в стартапах, которые хотят быть быстрыми.

3.3 Health check и Smoke test

После деплоя нужно убедиться, что система жива.

Health check: эндпоинт `/health`, который возвращает 200 OK, если всё хорошо. Kubernetes использует его для readiness/liveness.

Smoke test: после деплоя выполняем критический сценарий (регистрация, оплата, поиск). Если smoke test не прошёл — откатываемся.

3.4 Откат (Rollback)

Rolling: при ошибке Kubernetes сам откатит до предыдущей версии.

Blue-Green: переключили трафик обратно на Blue (мгновенно).

Часть 4. GitOps

4.1 Что это

Вся конфигурация системы (деплойменты, сервисы, ингрессы) хранится в Git. При изменении Git-репозитория автоматически синхронизируется кластер.

GitOps-цикл:

1. Разработчик меняет код
2. CI собирает образ и пушит в registry
3. Разработчик меняет тег в Git-репозитории манифестов
4. GitOps-оператор видит изменение и обновляет кластер

4.2 Инструменты GitOps

ArgoCD: оператор для Kubernetes. Следит за репозиторием и синхронизирует кластер. Показывает diff между Git и кластером.

Flux: альтернатива ArgoCD, проще, но меньше функций.

4.3 Helm: шаблонизация k8s-манифестов

Helm — пакетный менеджер для Kubernetes. Позволяет параметризовать манифесты, переиспользовать компоненты.

Преимущества: единый формат, возможность переиспользования [Helm charts], управление зависимостями.

4.4 Kustomize

Встроенный в kubectl инструмент. Наследует манифесты и переопределяет отдельные поля.

Проще Helm, не требует шаблонов, только наложение (патчи). Лучше для небольших проектов.

Итог главы 12

1. **CI** — автоматические сборка и тесты (быстро находим ошибки).
2. **CD** — автоматическая доставка до прода (релиз в любой момент).
3. **Continuous Deployment** — полностью автоматический деплой (максимальная скорость).
4. **CI-пайплайн:** линтер → компиляция → юнит-тесты → интеграционные тесты → сборка образа.
5. **Стратегии деплоя:** Rolling (просто), Blue-Green (без даунтайма), Canary (безопасно).
6. **GitOps:** вся конфигурация в Git (ArgoCD, Flux).

Что делать прямо сейчас:

1. Опишите ваш текущий процесс деплоя (ручной/автоматический).
2. Добавьте в CI: линтер, юнит-тесты.
3. Настройте автоматическую сборку Docker-образа в CI.
4. Внедрите Rolling update через Kubernetes.
5. Если есть возможность — переходите на GitOps (ArgoCD).

Следующая глава — «Платформы и инфраструктура: Облачные провайдеры» — расскажет о выборе между AWS, Azure, GCP и Yandex Cloud, а также стратегиях переноса в облако.

Глава 13. Облачные провайдеры

«Облако — это не место. Это способ думать об инфраструктуре»

В 2018 году я сидел в дата-центре и ждал, когда доставят новый сервер. Заказ был оформлен неделю назад. Сервер должен был прийти ещё вчера. Нагрузка росла. Я смотрел на пустую стойку и думал: «Почему мы не в облаке?».

Сейчас я смотрю на это иначе. Облако — не панацея. Оно решает одни проблемы и создаёт другие. В этой главе — как выбирать между облаками, как мигрировать и как не прогореть на счетах.

Часть 1. Основные провайдеры

1.1 Кто есть кто на рынке

По данным QYResearch на 2025 год, глобальный рынок облачных услуг достиг 2,259млрд. акаунтов. В 2032году ожидается рост до5,002 млрд (CAGR 12.0%) . Рынок консолидирован: пять крупнейших игроков контролируют более 50% .

Amazon Web Services (AWS) — старейший игрок с 2006 года, самый широкий спектр услуг, лидер по рыночной доле. Особенности: «Инновации как сервис» (сотни новых фич в год), сложное ценообразование, самая большая экосистема партнёров.

Microsoft Azure — второй по величине, сильная интеграция с Windows и .NET, популярность в корпоративном сегменте.

Google Cloud Platform (GCP) — лидер в AI/ML и аналитике данных (BigQuery), сильная сеть, уступает AWS/Azure по доле.

Yandex Cloud — российский лидер, соответствие 152-ФЗ, простота и низкий порог входа для локальных разработчиков.

VK Cloud — альтернатива в РФ, также соответствует регуляторным требованиям.

Выбор провайдера в России ограничен. Если ваши пользователи в РФ и данные должны храниться на территории страны — выбирайте между Yandex

Cloud, VK Cloud или SberCloud. AWS, Azure, GCP не гарантируют compliance и могут быть недоступны.

1.2 Основные типы услуг

- **IaaS (Infrastructure as a Service)** — аренда виртуальных машин, самая гибкая опция. Вы управляете ОС и приложениями, провайдер — железом и виртуализацией.
- **PaaS (Platform as a Service)** — среда исполнения приложений (Heroku, Google App Engine, Cloud Foundry). Не думаете об ОС и масштабировании, но ограничены в настройках.
- **SaaS (Software as a Service)** — готовое ПО (Google Workspace, Salesforce). Не думаете ни о чём, просто пользуетесь.
- **FaaS (Functions as a Service) / Serverless** — запуск кода по запросу (AWS Lambda, Google Cloud Functions, Yandex Cloud Functions). Платите только за время выполнения — идеально для событийно-ориентированных API.
- **DBaaS (Database as a Service)** — управляемые БД (Amazon RDS, Google Cloud SQL, Yandex Managed Service for PostgreSQL). Провайдер сам настраивает репликацию, бэкапы и отказоустойчивость.

1.3 Ключевые различия AWS, Azure, GCP, Yandex Cloud

Характеристика	**AWS**	**Azure**	**GCP**	**Yandex Cloud**
Вычислительные мощности	EC2 (гибкий)	Virtual Machines	Compute Engine	Compute Cloud
Serverless	Lambda	Functions	Cloud Functions	Cloud Functions
Объектное хранилище	S3	Blob Storage	Cloud Storage	Object Storage

Характеристика	AWS	Azure	GCP	Yandex Cloud
БД (SQL)	RDS	SQL Database	Cloud SQL	Managed Service for PostgreSQL/MySQL
БД (NoSQL)	DynamoDB	Cosmos DB	Firestore	Managed Service for MongoDB (ограниченно)
Аналитика	Redshift, Athena, EMR	Synapse Analytics	BigQuery	Data Proc, Yandex Query
AI/ML	SageMaker	Machine Learning Studio	Vertex AI, TPU	Yandex DataSphere, SpeechKit, Vision
Контейнеры	EKS (K8s), ECS	AKS	GKE	Managed Service for Kubernetes

Часть 2. Облачные паттерны

2.1 Database-as-a-Service vs Self-managed

Это один из главных споров при переходе в облако.

Управляемая БД (DBaaS) : провайдер сам настраивает репликацию, бэкапы, пачтинг. Вы получаете готовый инстанс через API за 5-15 минут . Провайдер VK Cloud фиксирует SLA 99.95% (не более 21 минуты даунтайма в месяц) . По данным Grand View Research, глобальный рынок DBaaS в 2024 году оценивался в $19.95 млрд .

Self-managed на облачной ВМ: вы арендуете виртуальную машину и сами устанавливаете БД. Полный контроль над конфигурацией, но и вся ответственность на вас.

Ключевые отличия :

Аспект	DBaaS	Self-managed
Развёртывание	5-15 минут	Часы-дни (установка, настройка)
Автомасштабирование дисков	Есть (без остановки)	Нет (требует ручного вмешательства)
Резервное копирование	Автоматическое (до секунды PITR)	Ручное (ваш скрипт)
Отказоустойчивость	Встроенная (Multi-AZ)	Проектируете сами
Доступ к параметрам ядра	Ограничен	Полный
Совокупная стоимость	Ниже (в среднем на 40-60%)	Выше (лицензии + администрирование)
Восстановление после сбоя	15-60 минут	4-8 часов (ручное)

Когда брать DBaaS: стандартные нагрузки, где не нужен экстремальный тюнинг; когда в команде нет сильных администраторов БД; при быстром запуске MVP.

Когда self-managed: специфические настройки под кастомную нагрузку; требования регуляторов к полному контролю над инфраструктурой; legacy-приложения с жёсткими зависимостями от версии СУБД.

2.2 Serverless плюсы и минусы

Serverless — запуск кода по запросу. Нет серверов, нет круглосуточных затрат.

Плюсы:

- **Масштабируется автоматически** от нуля до бесконечности
- **Платите только за вызовы**: нет простаивающих ресурсов
- **Не думаете об инфраструктуре**: ни ОС, ни патчей, ни кластеров
- Быстрый старт для небольших сервисов

Минусы:

- **Холодный старт** (100-500 мс) — приложение "засыпает" и просыпается на запрос
- **Ограничение по времени выполнения**: обычно до 900 секунд (15 минут)
- **Ограничения памяти и CPU**: до 10 ГБ, не для тяжёлых вычислений
- **Вендор-лок**: перенести код между провайдерами сложно

Когда использовать serverless:

- API с низкой или непредсказуемой нагрузкой
- Обработка файлов (ресайз картинок, транскодинг)
- ETL-задачи (конвертация, обогащение)
- Вебхуки и обработчики событий

Cloud Run от Google — хороший компромисс: запускает контейнеры как serverless, работает с любым языком, масштабируется до нуля и холодный старт — пара секунд .

2.3 Объектное хранилище (S3)

Объектное хранилище — не файловая система и не блочное устройство. Это хранилище ключ-значение с гигантской ёмкостью.

Плюсы: неограниченное масштабирование, дешевизна (в 5-10 раз дешевле дисков виртуальных машин), надёжность, встроенная репликация и версионирование.

Когда использовать: файлы пользователей (загрузки, аватары), логи, бэкапы, статический контент для сайтов, модели машинного обучения.

Часть 3. Стратегии переноса в облако (6 R)

3.1 Шесть стратегий модернизации

Эти шесть подходов (иногда расширяют до 7 R) — стандарт индустрии .

Rehost (Lift & Shift) — перенос приложения как есть. ✔ Быстро (недели), надежно (меньше рисков). ✗ Не используете преимущества облака. Используйте для legacy без времени на рефакторинг или при срочной миграции из ЦОД.

Replatform — переход на managed-сервисы (БД в RDS, перенос веб-приложения в Kubernetes). Умеренные усилия, уже получаете часть преимуществ облака.

Refactor / Re-architect — переписывание на микросервисы, serverless. Сложно, дорого, но максимальный эффект. Когда старый монолит совсем не тянет нагрузку.

Repurchase (Replace) — переход на SaaS (Jira вместо багтрекера). Быстро, но привязанность к вендору. Замена неконкурентных систем.

Retire — выключение ненужных сервисов. Экономит деньги и упрощает ландшафт.

Retain — оставить как есть. Иногда проще не трогать то, что работает.

3.2 Lift-and-shift: плюсы и минусы

Когда компания впервые идёт в облако, 80% проектов делают через Rehost.

Плюсы: минимальные риски, быстрая победа (миграция за недели), команда не переучивается.

Минусы: платите за облако, но не пользуетесь его преимуществами. Может оказаться дороже, чем on-premise, если оставить неоптимизированные запросы и лишние ВМ.

3.3 Рефакторинг под cloud-native

Cloud-native — приложения, рождённые в облаке, используют все его плюсы.

Принципы:

- **Микросервисы**: система из маленьких сервисов вместо одного монолита
- **Контейнеры и оркестрация** (Kubernetes)
- **Декларативность** (описываете желаемое состояние, Kubernetes приводит к нему)
- **Масштабирование** по событиям (CPU, очередь)
- **Устойчивость** к отказам (программная, на уровне приложения)
- **Наблюдаемость** (логи, метрики, трейсы — обязательны)

Cloud Native требует больше ресурсов на старте, но окупается гибкостью и скоростью.

Итог главы 13

1. **Основные провайдеры:** AWS, Azure, GCP, Yandex Cloud, VK Cloud. В РФ выбор ограничен локальными игроками.
2. **IaaS, PaaS, FaaS, DBaaS** — выбирайте уровень абстракции, исходя из своей экспертизы.
3. **Serverless (FaaS):** идеально для редких событий, но холодный старт и лимиты.
4. **Managed Databases (DBaaS):** экономят до 60% времени администрирования, выбор по умолчанию для 90% проектов .
5. **Стратегии миграции:** 6 R от Lift & Shift до Retire. Начинайте с Rehost, целитесь в Re-architect.

Что делать прямо сейчас:

1. Оцените требования: compliance, экспертиза, бюджет, пользователи.
2. При миграции начинайте с Rehost (lift & shift) для быстрой победы.
3. Постепенно переходите к Replatform (RDS вместо самописной БД).
4. Для новых проектов думайте Cloud-native с самого начала.
5. В России учитывайте 152-ФЗ и ограничения по трансграничной передаче данных.

Следующая глава — "Проектирование реальных систем: URL Shortener" — первый полноценный разбор конкретного приложения, от метрик до кода.

Глава 14. URL Shortener (пример полного цикла)

«Самый простой сервис часто даёт самые глубокие уроки. Если вы не можете спроектировать короткие ссылки, вы не сможете спроектировать YouTube».

После долгого и подробного погружения в теорию (CAP, ACID, масштабирование) пришло время для практики.

В этой главе мы пройдём **полный цикл проектирования системы** — от пустого листа до готовой архитектуры.

Почему URL Shortener? Потому что это классика System Design интервью. Сервис кажется тривиальным, но за ним стоят ключевые проблемы: **генерация уникальных ключей, шардирование, кэширование, аналитика, нагрузка, срок жизни данных**.

К концу главы у вас будет не только понимание, как работает сокращалка ссылок, но и шаблон, по которому вы сможете проектировать любые другие системы. Поехали.

Часть 1. Сбор требований

Прежде чем рисовать сервера и базы данных, нужно понять, **что именно мы строим**.

1.1 Функциональные требования

Что система должна делать? Распишем подробно.

Сокращение ссылки (write):

- Пользователь отправляет длинную URL (например, https://www.ozon.ru/product/krasnyy-chaynik-9876543210/).
- Система генерирует короткий код (например, abc123), сохраняет связь в БД.
- Возвращает короткую ссылку в формате https://short.xyz/abc123.

Редирект (read):

- Пользователь переходит по короткой ссылке https://short.xyz/abc123.
- Система ищет оригинальный URL по коду abc123.
- Возвращает HTTP-редирект (301 или 302) на оригинальный URL.
- Считает статистику перехода (опционально).

Опционально — расширенные функции: авторизация, кастомные алиасы (пользователь сам задаёт мой-чайник вместо abc123), TTL ссылки (автоматическое удаление через N дней), аналитика (сколько переходов, откуда, когда).

1.2 Нефункциональные требования (NFR)

Самое важное. Технические характеристики.

Хранилище (Storage):

- Где и как хранить пары (короткий код → длинная ссылка)? В БД, в Redis, в файлах?
- Сколько данных? Если мы генерируем **1 млн ссылок в день**, за год получится 365 млн записей. Каждая запись: ключ 6-10 байт + URL 100-200 байт + метаданные = ~250 байт. Умножаем на 365 млн ≈ 90 ГБ в год. Плюс индексы ×2 = 180 ГБ. Плюс аналитика: каждый переход пишется в таблицу или лог (ещё +90 ГБ). Итог: ~300–400 ГБ в год — спокойно влезает на один сервер с SSD на 1 ТБ. Можно не шардировать первые пару лет.

Масштабируемость (Scalability):

- **Чтение:** редиректов может быть в 100 раз больше, чем создания ссылок. Каждая короткая ссылка может быть популярной (вирусный пост, рассылка).
- **Запись:** 1 млн новых ссылок в день = 12 ссылок в секунду (пик может быть в 5-10 раз выше) — это очень мало для современной БД, даже на одном сервере.
- **Чтение:** 100 млн переходов в день = 1 160 переходов в секунду (пик может быть в 10-20 раз выше — 10-20 тысяч RPS). Это уже серьёзная нагрузка, нужен кэш.

Задержка (Latency): редирект должен быть максимально быстрым (<10 мс на уровне бэкенда). Всё, что дольше, пользователь заметит.

Доступность (Availability) и надёжность (Durability): ссылки не должны теряться, система должна быть доступна 99.99% (простой не более 52 минут в год). Редирект не должен падать, иначе пользователи не попадут на целевой сайт.

Аналитика (Analytics): нужно знать, сколько переходов по каждой ссылке (возможно, с разбивкой по времени, географии, реферерам).

1.3 Оценка нагрузки (Back of the envelope)

Посчитаем для среднего сервиса: 1 млн новых ссылок в день, 100 млн редиректов в день.

Показатель	**Формула**	**Результат**
Новых ссылок в день (запись)	задано	1 млн
Ссылок в секунду (пик ×10)	1 млн / 86400 ×10	≈115 RPS
Редиректов в день (чтение)	задано	100 млн
Редиректов в секунду (пик ×10)	100 млн / 86400 ×10	≈ 11 574 RPS
Объём данных за год (ключ: 10 байт, URL: 200 байт, мета: 50 байт)	260 байт × 365 млн	≈95 ГБ
Индексы (×1.5)	95 × 1.5	≈142 ГБ
Всего хранилище + запас 2-3 года		300-500 ГБ

Выводы:

- **Write – низкая нагрузка:** 115 RPS пика — выдержит даже один сервер PostgreSQL без реплик.
- **Read – высокая нагрузка:** 11 574 RPS — это много даже для кластера БД. Нужен кэш (Redis), который может отдавать до 1 млн RPS на одном узле.
- **Хранилище – комфортно на одном сервере:** 300–500 ГБ помещается на любой SSD-диск. Шардирование не требуется в первые годы. Но учитывайте, что индексы занимают память и размер; таблица записей с миллиардом строк уже выглядит не очень опрятно. Можно партицировать по времени (интервалам создания ссылки) для удобства обслуживания.

1.4 Проектирование API

API — контракт между клиентами и сервером.

Создание короткой ссылки:

http

```
POST /api/v1/shorten
Content-Type: application/json

{
  "url": "https://www.ozon.ru/product/...",
  "alias": "my-link",      // опционально
  "ttl_days": 30           // опционально, срок жизни
}
```

Пример ответа:

json

```
{
  "short_code": "abc123",
  "short_url": "https://short.xyz/abc123",
  "original_url": "https://www.ozon.ru/product/...",
  "created_at": "2025-12-22T14:30:00Z",
  "ttl_days": 30
}
```

Получение информации о ссылке:

http

```
GET /api/v1/info/abc123
```

Статистика переходов:

http

```
GET /api/v1/stats/abc123
```

Редирект (для пользователей):

http

```
GET /{short_code}
```

→ Location: original_url

1.5 Выбор базы данных

Нужно решить: SQL или NoSQL?

- **SQL (PostgreSQL):** Транзакции (ACID), дешево, простота, удобная аналитика (SQL). Write 115 RPS — это легко. Read 11 574 RPS — нужно кэшировать, но БД тоже даст пару тысяч RPS без проблем. Индексы, партиционирование по времени создания — всё это есть в PostgreSQL. Выбор по умолчанию.
- **NoSQL (MongoDB):** Гибкая схема, простое шардирование (но нам пока не нужно), высокий RPS на запись (а у нас его нет). Нет транзакций, сложнее аналитика.
- **Redis как база данных:** Да, в теории можно хранить всё в Redis, если данные умещаются в оперативную память. Запись до 1 млн RPS, чтение — ещё быстрее. Нет постоянного хранилища (данные могут потеряться при перезагрузке).

Мой выбор: PostgreSQL + Redis.

- PostgreSQL для постоянного хранилища (источник истины).
- Redis для кэша горячих ссылок (ускорение редиректа).
- Если нагрузка вырастет, добавим реплики PostgreSQL для чтения.

Часть 2. Генерация коротких ключей

Самый интересный вопрос: как из длинной ссылки получить короткий код `abc123`?

2.1 Base62 инкрементальный ID

Идея: глобальный счётчик (1, 2, 3...). Переводим число в Base62 (цифры + строчные + прописные буквы). 1 → `1`, 10 → `a`, 61 → `Z`, 62 → `10`.

Плюсы: простота, короткие коды, монотонность.

Минусы: централизованный счётчик (единая точка отказа, но можно вынести в Redis и шардировать). Кто-то может перебирать ID и собирать все ссылки сервиса.

Решение проблемы перебора: добавляем проверку прав доступа (авторизация для информации о ссылке) или генерируем рандомные ключи с проверкой на

коллизию (если уже существует — генерируем новый). Для рандома возьмите криптостойкий генератор (secrets в Python).

2.2 Хэш (MD5, SHA-1) + Base62

Берём MD5 от URL, берём первые 6-8 символов Base62.

Плюсы: нет центрального счётчика, распределённая генерация.

Минусы: коллизии (разные URL могут дать одинаковый хэш). Нужна проверка и повторная генерация (с добавлением соли). Ключи не монотонные.

2.3 UUID

Универсальный идентификатор (128 бит). Кодируем в Base62, получаем длинную строку (20-22 символа). Слишком длинный для короткой ссылки, но можно.

Плюсы: глобально уникальный, генерируется без проверки.

Минусы: длинный, неудобный для пользователя.

2.4 Генерация ключа без коллизий на лету

Алгоритм:

1. Генерим 6-значный случайный код (Base62, 62^6 ≈ 56 млрд комбинаций).
2. Проверяем в БД, существует ли уже.
3. Если да — генерим заново (+1 к длине или новая соль).
4. Если нет — сохраняем.

Для нагрузки 115 RPS вероятность коллизии очень низкая (56 млрд комбинаций). Можно даже держать в Redis Bloom filter для быстрой проверки.

Что выбрать? Base62 инкрементальный ID (простота) или рандомная генерация (скрытность). Для публичного сервиса с возможностью перебора лучше рандомная.

Часть 3. Кэширование

Редиректов в 100 раз больше, чем созданий ссылок. Кэшировать нужно обязательно.

3.1 Стратегия кэширования

Cache-Aside (Lazy Loading):

- Запрос на редирект приходит → ищем в Redis.
- Если есть в Redis → возвращаем URL (быстро, за микросекунды).
- Если нет в Redis → ищем в PostgreSQL, сохраняем в Redis на TTL (например, 24 часа), возвращаем URL.

Почему именно Cache-Aside:

- Нет лишних обновлений кэша.
- Самые популярные ссылки быстро кэшируются.
- При падении Redis всё работает (медленнее, но работает).

Инвалидация: по TTL; при удалении ссылки — напрямую.

3.2 Размер кэша

Храним всё? 365 млн ссылок × 250 байт ≈ 90 ГБ. Влезает в оперативную память (например, Dedicated Redis на 128 ГБ). Но зачем? Достаточно хранить топ популярных ссылок (по правилу 80/20: 20% ссылок дают 80% трафика).

Кэшируем горячие ссылки, холодные — из БД.

3.3 Предотвращение проблем

Thundering Herd (одновременное обновление): при истечении TTL тысячи запросов бросаются в БД.

Решение: блокировка на уровне ключа при обновлении кэша (один клиент обновляет, остальные ждут).

Cache penetration (несуществующий ключ): злоумышленник долбит несуществующий код.

Решение: кэшируем отрицательный ответ на короткое время (5-10 минут).

Часть 4. Итоговая архитектура

Собираем всё вместе.

Компоненты:

- **API Gateway:** единая точка входа, rate limiting, авторизация (если есть)
- **Сервис «Создание ссылок» (write):** принимает запрос, генерирует код, пишет в БД, шлёт событие в Kafka (для аналитики)
- **Сервис «Редирект» (read):** читает из кэша или БД, возвращает редирект
- **PostgreSQL (основное хранилище):** таблицы `links` (код → URL, дата создания, TTL, автор) и `users` (если есть авторизация)
- **Redis (кэш):** HOT-ссылки (TTL 24 часа)
- **Kafka (очередь):** события о переходах (для асинхронной аналитики)
- **Аналитический сервис:** читает из Kafka, аггрегирует, складывает в ClickHouse

Поток создания ссылки:

1. Клиент → API Gateway → Сервис создания
2. Генерация ключа (случайный код + проверка существования)
3. Запись в PostgreSQL (`code`, `original_url`, `created_at`)
4. Запись в Redis (кэш, чтобы при первом же редиректе не идти в БД)
5. Ответ клиенту

Поток редиректа:

1. Клиент → API Gateway → Сервис редиректа
2. Поиск в Redis (ключ `short:abc123`)
3. Если найден → HTTP 301 (permanent) или 302 (temporary)
4. Если нет → поиск в PostgreSQL, сохранение в Redis, HTTP 301/302
5. Асинхронная отправка события в Kafka (лог перехода)
6. Пользователь перенаправлен на оригинальный URL.

Итог главы 14

Мы спроектировали полноценный высоконагруженный сервис сокращения ссылок:

- **Требования:** до 1 млн новых ссылок/день, 100 млн редиректов/день.
- **Хранилище:** PostgreSQL (источник истины) + Redis (кэш).
- **Генерация ключей:** Base62 случайными кодами с проверкой коллизии.

- **Масштабирование:** при росте нагрузки добавим реплики PostgreSQL, шардинг по диапазонам кодов.
- **Аналитика:** асинхронная через Kafka + ClickHouse.

  Мы использовали лишь малую часть возможностей каждого компонента. Система готова к нагрузке в миллионы редиректов в сутки и лёгкому масштабированию.

  Следующая глава — «Мессенджер (Telegram / WhatsApp)» — где мы добавим real-time, веб-сокеты и синхронизацию устройств.

Глава 15. Мессенджер (Telegram / WhatsApp)

«Мессенджер — это не просто чат. Это распределённая система с real-time, синхронизацией устройств, медиафайлами и миллиардами сообщений. И всё это должно работать мгновенно»

Когда я начинал проектировать мессенджер для одного государственного проекта, я думал: «Ну что там сложного? Отправил сообщение — получил ответ». Наивный.

Реальность оказалась жестокой. Как гарантировать доставку, если сеть нестабильна? Как синхронизировать сообщения на телефоне, планшете и ноутбуке? Что делать, если пользователь офлайн? Как передать фото весом 20 МБ, не убив память сервера?

В этой главе мы пройдём полный цикл проектирования мессенджера. На примере Telegram/WhatsApp, но не копируя их слепо — у нас нет сотен миллионов пользователей, но есть здравый смысл и трезвая оценка требований.

Часть 1. Сбор требований

1.1 Функциональные требования

Основной сценарий:

Личные чаты:

- Пользователи отправляют текстовые сообщения друг другу.
- Сообщения доставляются в реальном времени (задержка < 100 мс).
- Сообщения сохраняются на сервере и на всех устройствах пользователя.

Групповые чаты:

- Пользователи создают группы, приглашают участников.
- Сообщение в группе доставляется всем участникам.
- Поддержка до 10 000 участников в группе (как в Telegram).

Медиафайлы:

- Отправка изображений, видео, аудио, документов.
- Оптимизация (миниатюры, сжатие) на стороне сервера.

Офлайн-сообщения:

- Если получатель не в сети, сообщение сохраняется и доставляется при подключении.

Уведомления:

- Push-уведомления для мобильных устройств (через FCM/APNS).

Синхронизация устройств:

- Пользователь залогинен на телефоне, планшете, ноутбуке.
- Сообщения, прочитанные на одном устройстве, помечаются как прочитанные на всех.

End-to-end шифрование (опционально, но желательно):

- Секретные чаты.
- Ключи шифрования хранятся только на устройствах, не на сервере.

1.2 Нефункциональные требования (NFR)

Хранилище (Storage):

Сообщения хранятся вечно (или до удаления пользователем). Оценим объём.

Дано:

- 10 млн активных пользователей.
- Каждый отправляет 10 сообщений в день (цифра для оценки: WhatsApp — 30+ сообщений).
- Среднее сообщение: текст 100 байт (это примерно 50-100 символов + метаданные).

Расчёт за день:

10 млн × 10 × 100 байт ≈ 10 ГБ новых сообщений в день → 3,6 ТБ в год. Добавляем индексы (×1,5) и медиафайлы (фото 1 МБ, видео 5-10 МБ). Если предполагаются файлы, объём взлетит до петабайтов.

Масштабируемость (Scalability):

Пиковая нагрузка может быть в 5-10 раз выше средней (вечером, в выходные, после рассылки). С сообщениями и чатами нужна горизонтальная шардизация по shard_id (например, по user_id).

Доступность (Availability) и надёжность (Durability):

Сообщения не должны теряться — это важно. Доставка сообщений должна быть 99.99% (система доступна всегда). Если один сервер падает, другие подхватывают нагрузку.

Задержка (Latency):

Отправка сообщения (round-trip) < 100 мс для 99% пользователей. Real-time должно ощущаться как «мгновенно».

Часть 2. Архитектура данных

2.1 Базы данных: профили, переписки, метаданные

PostgreSQL:

- Профили пользователей (user_id, name, avatar_url, last_seen).
- Группы (group_id, name, avatar_url, created_at).
- Участники групп (group_id, user_id, role).
- Сообщения: в PostgreSQL храним только метаданные, а текст — в NoSQL или отдельной таблице с партиционированием.

На самом деле, для миллиардов сообщений реляционная база становится тяжёлой. Даже партиционирование по месяцам не спасает при 10 млн RPS на чтение истории.

Кассандра / ScyllaDB (распределённая NoSQL для сообщений):

Идеально подходит для сообщений, если мы заранее знаем запросы:

- SELECT * FROM messages WHERE chat_id = ? ORDER BY created_at DESC LIMIT 50.
- Partition key: chat_id, Clustering key: created_at.

Кассандра даёт высокую доступность и отказоустойчивость без единой точки отказа.

2.2 Текст сообщений и метаданные

Можно хранить всё в одной NoSQL-таблице:

json

```
{
  "chat_id": "user123:user456",
  "message_id": 987654321,
  "sender_id": "user123",
  "text": "Привет!",
  "created_at": "2025-12-22T14:30:00Z",
  "delivered": false,
  "read": false
}
```

delivered и read можно обновлять (для Кассандры это дорого; можно перекладывать в Redis).

Часть 3. Протокол и реальное время

3.1 Real-time – синхрон через WebSocket

Чтобы пользователь видел новые сообщения без задержки, нужно постоянное соединение с сервером.

WebSocket: двунаправленный канал. Сервер может отправлять данные клиенту без запроса. Идеально для чатов.

При подключении:

1. Клиент открывает WebSocket к API Gateway.
2. Передаёт user_id и токен авторизации.
3. Сервер запоминает соединение в map: user_id → список подключений (устройств).

Отправка сообщения:

1. Клиент А отправляет сообщение через WebSocket.
2. Сервер получает, сохраняет в БД, проверяет асинхронно.
3. Сервер находит WebSocket клиента Б (и другие устройства).
4. Отправляет сообщение через WebSocket.

5. Клиент Б получает и отображает сообщение.

Время жизни WebSocket: пока открыт. Если закрыт (прервано соединение), сообщение идёт через офлайн-очередь.

3.2 Асинхронная обработка и очередь

Сохранение сообщения в БД может быть асинхронным:

1. Сообщение попадает в очередь (Kafka).
2. Клиент получает подтверждение, что сообщение принято.
3. Консьюмер читает из очереди и сохраняет в БД.

Почему асинхронно: чтобы не задерживать отправку. Клиент не ждёт, пока сообщение запишется на диск.

Но есть риск: если консьюмер упадёт, сообщение может потеряться. Для надёжности нужна репликация и подтверждения.

Часть 4. Медиафайлы

4.1 Отдельное хранилище (Blob Storage) + ленивая загрузка

Медиафайлы (фото, видео) нужно хранить отдельно от сообщений.

Поток отправки:

1. Клиент загружает файл в Blob Storage (S3, объектное хранилище).
2. Получает media_id.
3. Отправляет сообщение с media_id.
4. Сервер сохраняет media_id в БД, не храня сам файл.

Поток получения:

1. Клиент получает сообщение с media_id.
2. Загружает файл по media_id из Blob Storage.

4.2 Кэширование миниатюр

Для изображений и видео нужны миниатюры (превью). Храним их отдельно, в том же Blob Storage, и кэшируем в Redis или CDN.

Часть 5. Офлайн-сообщения, уведомления и синхронизация устройств

5.1 Офлайн-сообщения

Пользователь отключился от WebSocket (закрыл приложение, нет интернета). Сообщения не должны теряться.

Решение: хранить неотправленные сообщения в отдельной очереди для пользователя (например, в Redis List или Kafka). При подключении клиент запрашивает все новые сообщения (`GET /api/v1/messages/after/{last_message_id}`) и доставляет их.

5.2 Push-уведомления (для мобильных устройств)

Если приложение закрыто, WebSocket не работает, но уведомление показать нужно.

Решение:

- Сообщение попадает в очередь User → Push Notification Service.
- Сервис отправляет push через FCM (Google) или APNS (Apple).
- Пользователь видит уведомление «Новое сообщение от ...».

Ограничения: в push нельзя передавать текст сообщения в E2EE-чатах. Отправляем только уведомление «у вас новое сообщение».

5.3 Синхронизация нескольких устройств

Пользователь залогинен на телефоне, планшете, ноутбуке. Сообщение, прочитанное на телефоне, должно перестать быть «непрочитанным» на других устройствах.

Решение: Команда `mark_read` (Chat ID, Message ID). Сервер обновляет статус `read` в БД. Все устройства получают событие через общий WebSocket или при следующей синхронизации.

Часть 6. Итоговая архитектура

Собираем все компоненты в единую схему:

Компоненты:

- **API Gateway (WebSocket + REST):** маршрутизация, подключение WebSocket, rate limiting.
- **Сервис сообщений (chat service):** сохранение в очередь, управление офлайн-очередью, отправка pusher.
- **Redis:** хранение активных WebSocket-сессий (`user_id → список соединений`), очередь офлайн-сообщений, статусы `delivered`/`read`.
- **Kafka:** очередь сообщений для асинхронной обработки.
- **PostgreSQL:** профили пользователей, чаты (метаданные).
- **Cassandra / ScyllaDB:** все сообщения с партиционированием по `chat_id`.
- **Blob Storage (S3):** медиафайлы с миниатюрами.
- **ClickHouse:** аналитика использования (сообщения в день, активность пользователей).

Поток отправки сообщения:

1. Клиент A отправляет сообщение через WebSocket в API Gateway.
2. Gateway валидирует, определяет `chat_id`.
3. Сообщение попадает в **Kafka**.
4. Консьюмер из Kafka сохраняет сообщение в **Cassandra** (асинхронно).
5. Сервер смотрит в Redis, у кого из получателей открыт WebSocket.
6. Если открыт — отправляет сообщение через WebSocket.
7. Если нет — кладёт сообщение в офлайн-очередь (Redis или Kafka для пользователя), отправляет push-уведомление через FCM/APNS.

Итог главы 15

Мы спроектировали высоконагруженный мессенджер:

- **Real-time доставка через WebSocket** с сохранением сообщений в БД (асинхронно через очередь).
- **Офлайн-очередь** для временно недоступных пользователей и **push-уведомления** для мобильных устройств.
- **Синхронизация устройств** через общую модель хранения и статусы прочтения.
- **Медиафайлы** вынесены в Blob Storage, миниатюры — в CDN.

- **Горизонтальное масштабирование** за счёт партиционирования сообщений по `chat_id` в Cassandra и шардирования пользователей.

  Бонус: мы почти не использовали блокчейн (потому что он не нужен для доставки сообщений). End-to-end шифрование добавить легко, если нужно.

  Следующая глава — «Новостная лента (Twitter / VK)» — переключит фокус с сообщений на публичную активность миллионов пользователей.

Глава 16. Новостная лента (Twitter / VK)

«Проектировать новостную ленту — это как строить скоростную автомагистраль, где каждый водитель хочет ехать первым, но все должны видеть одни и те же дорожные знаки в реальном времени».

После мессенджера, где мы работали с приватными чатами 1-1 и группами до 10 000 человек, новостная лента ставит совершенно иную задачу. Здесь один автор пишет пост, а его видят миллионы подписчиков. И инженерная дилемма звучит так:

Как показывать миллионам подписчиков миллиарды постов за миллисекунды?

В этой главе мы разберём ключевой паттерн — fanout (расходимость) — и выясним, что выгоднее: раздавать посты сразу при публикации (push) или собирать их на лету при чтении (pull). А главное — почему правильный ответ часто лежит где-то посередине.

Часть 1. Требования и нагрузка

1.1 Функциональные требования

- **Публикация поста.** Пользователь пишет текст (до 280–1000 символов), прикрепляет медиа (фото/видео), указывает хештеги.
- **Лента подписок.** Пользователь видит посты всех людей и пабликов, на которые подписан, в хронологическом или «умном» порядке.
- **Лайки и репосты.** Возможность оценивать и распространять чужие записи.
- **Поиск по хештегам и ключевым словам** (необязательно, но чаще присутствует).
- **Лента рекомендаций.** Блок «Вам может понравиться» — на основе интересов (для MVP опустим).

1.2 Оценка нагрузки (Twitter-like)

Показатель	Значение
Пользователей	500 млн (Twitter), упростим до 50 млн
Активных в день (DAU)	200 млн (Twitter), 10 млн (наш масштаб)
Постов в день	500 млн (Twitter), 20 млн (наш масштаб)
Подписок на пользователя	в среднем 100–200
Просмотров ленты в день	15–20 млрд (Twitter), 500 млн – 1 млрд (наш масштаб)

Нагрузка на запись (публикация):

- 20 млн постов / 86 400 секунд ≈ 230 постов/сек.
- Пик (×5) ≈ 1 150 постов/сек — решается вертикально или простым реплицированием.

Нагрузка на чтение (генерация ленты):

- 10 млн DAU × 100 просмотров ленты/день (примерно каждые 15 минут) ≈ 1 млрд просмотров в день.
- Пик (×5) ≈ 58 000 RPS (запросов в секунду). Это серьёзно.

Объём данных:

- Пост: 1 КБ (текст) + метаданные (ID автора, время, доп. поля) ≈ 1,5 КБ.
- 20 млн постов/день × 365 × 1,5 КБ ≈ 11 ТБ в год.

Часть 2. Ядро системы: две модели доставки

Здесь начинается самое интересное.

2.1 Fanout on write (Push)

При публикации поста система раскладывает его в ленту каждому подписчику.

Как работает:

1. Алиса публикует пост.
2. Сервис ленты смотрит в граф подписок: кто подписан на Алису (5 млн фанатов).
3. Для каждого подписчика добавляет ID этого поста в их персональную ленту (хранилище `timeline:{user_id}`), обычно в Redis Sorted Set (ключ — timestamp, значение — post_id).

Плюсы:

- Чтение невероятно быстрое: просто достать список `post_id` из Redis.
- Ленты легко кэшируются и пагинируются.

Минусы:

- Тяжело на запись: при публикации поста у звезды (100 млн фанатов) запись занимает минуты, а иногда и часы.
- Хранилище «раздувается»: один пост, даже не очень популярный, дублируется на количество подписчиков (10 млн подписчиков → 10 млн записей в Redis).

2.2 Fanout on read (Pull)

При запросе ленты система идёт в граф подписок и собирает посты «на лету».

Как работает:

1. Боб открывает ленту.

2. Сервис ленты смотрит, на кого подписан Боб (условно 200 аккаунтов).
3. Для каждого из 200 аккаунтов идёт в индекс постов (по автору) и достаёт N последних.
4. Мерджит, сортирует по времени, обрезает до 20-50 постов и отдаёт.

Плюсы:

- Нет аммортизации фолловеров-звезд. Запись остаётся лёгкой всегда.
- Данные не дублируются (экономия памяти).

Минусы:

- Тяжело на чтение: при каждом запросе Боба нужно опрашивать 200 авторов, что при 58 000 RPS убивает базу данных.

Часть 3. Гибридная модель (реальная жизнь)

Никто не использует чистый push или pull. Инженеры Twitter, VK и Facebook пришли к гибридному подходу.

3.1 Push для «обычных» аккаунтов, Pull для «звёзд»

Звезды (блогеры, политики, паблики-миллионники) генерируют огромные fanout, но их мало (менее 0.1% от всех аккаунтов).

Алгоритм:

1. **Список звёзд** хранится отдельно (User.is_big = True).
2. При публикации:
   - Если автор НЕ звезда → fanout on write (push) для всех фолловеров (их немного, <5000).
   - Если автор звезда → не делаем fanout, просто кладём пост в глобальную таблицу.
3. При чтении ленты:

- Сначала достаём push-ленту (из Redis).
- Потом смотрим, на каких звёзд подписан пользователь, и делаем pull (до 5-10 самых свежих постов от них).
- Мерджим два потока, сортируем, отдаём.

Так мы получаем скорость push-чтения (99% контента уже лежит в Redis) и масштабируемость pull-записи (звёзды не дублируют посты на миллионы подписчиков).

3.2 Кэширование популярных постов

Некоторые посты становятся вирусными (репостнули, залайкали миллионы раз). Их нужно кэшировать максимально агрессивно.

Решение: хранить посты звёзд не в Redis «для каждого», а в общем кэше — «популярные посты» (latest popular). При pull у звёзды хватаем 5-10 постов из этого общего кэша, а не из персонального.

Timeline сборка (примерно за 50 мс):

1. Берём из Redis персональную ленту (push). (30 мс)
2. Идём в общий кэш «популярные посты». (10 мс)
3. Мерджим и сортируем. (10 мс)

Часть 4. Детали реализации

4.1 Шардирование по пользователям

Пользователей миллионы, хранить всех в одном узле нельзя. Шарды по `user_id`, например, 1024 шарда (`hash(user_id) % 1024`).

Push-лента: Redis-кластер, шардирование по `user_id`.

Pull-лента (индекс постов): PostgreSQL или Cassandra с партиционированием по автору `author_id`.

4.2 Балансировка и очереди

Публикация от звезды (20 млн подписчиков) не должна вешать сервис. Добавляем **очередь задач (Kafka)**.

Поток публикации от звезды:

1. Пост попадает в очередь `fanout_big`.
2. Несколько воркеров читают, шардируют работу (fanout по частям подписчиков).
3. Для каждого пользователя (пакетами по 1000) отправляем задачу в Redis (запись в ленту).
4. Пользователь видит пост в лучшем случае через 30-60 секунд, но звезда и не ждёт моментальной доставки всем подписчикам.

Для обычных аккаунтов fanout делается синхронно (за 1-2 секунды).

4.4 Схема данных

PostgreSQL (основная БД):

- `users` (`user_id` PK, `name`, ...).
- `follows` (`follower_id`, `followee_id`, `created_at`).
- `posts` (`post_id` PK, `author_id`, `content`, `created_at`).

Redis (Push-лента):

- `timeline:{user_id}` — Sorted Set: `score = created_at_unix`, `member = post_id`.

Cassandra (Pull-индекс для звёзд):

- `posts_by_author` (`author_id`, `created_at`, `post_id`, `content`).

Общий кэш популярных постов (Redis): popular_posts — Sorted Set, содержит post_id популярных постов (например, по количеству лайков за последние N часов).

4.5 Лента рекомендаций

Рекомендательная система отличается от ленты подписок («Вам может понравиться»). Работает на основе коллаборативной фильтрации («другие пользователи с похожими интересами тоже читают этот паблик») или content-based («похожий текст/хештеги»). Полноценный ML-пайплайн сложен.

MVP упрощение: можно делать random из «популярных постов за час».

Итог главы 16

Мы спроектировали новостную ленту с нагрузкой до 58 000 RPS чтения и 1 100 RPS записи.

Ключевые решения:

- **Гибридный fanout:** push для обычных (быстрое чтение), pull для звёзд (экономия памяти и записи).
- **Очередь задач (Kafka)** для публикации от звёзд.
- **Шардирование по** user_id для Redis и Cassandra.
- **Общий кэш популярных постов** для быстрого pull.
- **Pull-индекс (Cassandra)** для запросов по автору.

Что не вошло в главу (но важно знать):

- **Real-time обновление ленты (WebSocket).** Мы предполагаем, что пользователь обновляет ленту вручную (pull-to-refresh). В реальных соцсетях новые посты подгружаются автоматически с помощью long polling или WebSocket.
- **Ранжирование ленты** (замена хронологического порядка на «умный»). Это требует отдельной сложной системы скоринга.

- **Медиафайлы** (Blob Storage, CDN) — принцип тот же, что в мессенджере, но с более агрессивной миниатюризацией.

  Следующая глава — «Платёжная система» — переключит фокус с нагрузок на абсолютную надёжность. Там нет места eventual consistency и потерянным сообщениям.

Глава 17. Платёжная система (Stripe / PayPal)

«В обычных системах losing сообщение — это баг. В платёжной системе losing транзакцию — это катастрофа. Здесь нет понятия почти правильно. Есть только правильно и неправильно. И за неправильно отвечаешь ты»

Когда я проектировал платёжную систему для государственного сервиса, мне сказали: «Потеря одного рубля — это репутационная катастрофа. Потеря одного голоса — это политическая катастрофа. У тебя и то, и другое».

Я переписал архитектуру четыре раза. Четвёртый вариант работает до сих пор.

В этой главе — всё, что я узнал о платёжных системах. Идемпотентность, саги, двойная бухгалтерия, работа с банками. И главное — как не потерять деньги клиентов.

Часть 1. Требования: никакой потери денег

1.1 Функциональные требования

- **Приём платежей:** пользователь платит картой, через SBP, Apple/Google Pay.
- **Возвраты:** полные или частичные.
- **Подписки (рекуррентные платежи):** ежемесячное списание.
- **Вывод средств:** на счёт продавца.
- **Отчёты и выписки:** для бухгалтерии.

1.2 Нефункциональные требования (самое важное)

- **Exactly-once:** ни одного лишнего списания, ни одной потери платежа.
- **Долговечность:** платёж не теряется даже при падении всех серверов.
- **Согласованность:** балансы всегда сходятся.
- **Аудит:** любое изменение можно отследить.
- **Доступность:** 99.99% (но если выбирать между доступностью и согласованностью — выбираем согласованность).

Золотое правило: лучше сказать «сервис недоступен, попробуйте позже», чем списать деньги дважды или потерять их.

Часть 2. Идемпотентность

2.1 Проблема

Пользователь нажал «оплатить». Запрос ушёл на сервер. Сервер начал обрабатывать. В этот момент у пользователя упал интернет, он нажал «оплатить» ещё раз.

Что произойдёт? Без идемпотентности деньги спишутся дважды.

2.2 Решение: idempotency_key

Клиент генерирует уникальный ключ для каждого платежа (например, UUID) и передаёт его в заголовке `Idempotency-Key`.

Сервер:

1. Проверяет в БД, был ли уже такой ключ.
2. Если был — возвращает сохранённый результат (не списывает повторно).
3. Если нет — обрабатывает платеж, сохраняет ключ и результат.

Хранение ключей: таблица `idempotency_keys` (ключ, результат, created_at). Очищаем через N дней (обычно 30).

2.3 Где ещё нужна идемпотентность

- **Retry при сетевых сбоях.** Клиент автоматически повторяет запрос. Идемпотентность спасает от дублей.
- **Webhook'и от банка.** Банк может прислать одно и то же уведомление дважды.

Главное правило: любой запрос, который меняет состояние (списание, возврат, отмена подписки), должен быть идемпотентным.

Часть 3. Распределённые транзакции: паттерн Saga

3.1 Проблема: один платеж — много сервисов

Платеж в современной системе задействует несколько сервисов:

1. **Сервис платежей** (создаёт запись).
2. **Сервис балансов** (обновляет счёт пользователя).
3. **Сервис уведомлений** (отправляет email/ push).
4. **Эквайринг** (банк, который списывает деньги с карты).

Если всё в одной транзакции, глобальный ACID невозможен — это распределённая система. Нужен паттерн Saga.

3.2 Что такое Saga

Saga — последовательность локальных транзакций. У каждой транзакции есть компенсирующее действие (rollback).

Пример:

1. **Сервис платежей:** создать запись `PENDING`.
   - Компенсация: удалить запись.
2. **Эквайринг:** зарезервировать деньги.
   - Компенсация: отменить резервирование.
3. **Сервис балансов:** увеличить баланс продавца.
   - Компенсация: уменьшить баланс.
4. **Сервис уведомлений:** отправить письмо.
   - Компенсация: отправить письмо об ошибке.

Если шаг 3 упал, запускаются компенсации для шагов 2 и 1. Система откатывается до исходного состояния.

3.3 Saga Orchestration (центральный координатор)

Оркестратор (один сервис) говорит, что делать каждому участнику.

Плюсы: проще отслеживать состояние, централизованная логика.
Минусы: координатор — единая точка отказа и возможное узкое место.

3.4 Saga Choreography (хореография, децентрализованная)

Каждый сервис после выполнения публикует событие. Другие сервисы подписываются и реагируют.

Плюсы: нет центральной точки, слабая связанность.
Минусы: сложнее отлаживать, цикличные зависимости.

Мой выбор: для платёжной системы — оркестрация. Слишком дорого ошибиться.

Часть 4. Двойная бухгалтерия

Бухгалтеры придумали двойную запись 500 лет назад. В IT она работает до сих пор.

4.1 Принцип

Каждая операция записывается дважды:

- По дебету одного счёта
- По кредиту другого

Сумма дебетов всегда равна сумме кредитов. Баланс всегда сходится.

4.2 Актуально для переводов

Пример перевода 100 руб со счёта А на счёт Б:

Дебет: счёт А (уменьшение) 100 руб
Кредит: счёт Б (увеличение) 100 руб

Если записать только одну сторону, баланс нарушится.

4.3 Как хранить в БД

Таблица `ledger_entries`:

entry_id	account_id	amount	type	transfer_id
1	A	100	DEBIT	TXN001

entry_id	account_id	amount	type	transfer_id
2	Б	100	CREDIT	TXN001

Выборка по transfer_id даёт обе стороны. Сумма дебетов всегда = сумме кредитов.

4.4 Проверка консистентности

Периодически запускать аудит: сумма всех дебетов = сумме всех кредитов. Если нет — ищем ошибку.

Часть 5. Интеграция с банками и эквайрингами

Платежи не заканчиваются в вашей системе — вы отправляете запросы во внешние банки и платёжные шлюзы.

5.1 Асинхронные ответы (Webhook)

Банк не может ответить мгновенно (фриз, проверка). Поэтому ответ приходит через 2-10 секунд на ваш Webhook.

Поток:

1. Вы отправляете запрос на /charge в эквайринг.
2. Эквайринг возвращает status: pending.
3. Через несколько секунд эквайринг стучится в ваш /webhook/payment с status: success или failed.

5.2 Вебхуки должны быть идемпотентными

Эквайринги могут дублировать уведомления. Idempotency-Key спасает.

5.3 Ретраи и таймауты

При недоступности банка нужно повторять запрос (экспоненциальная задержка с джиттером). Но в какой-то момент признавать отказ и переключаться на другой эквайринг.

5.4 Dead Letter Queue

Запросы, которые не удалось обработать даже после 5 повторов, уходят в DLQ для ручного разбора (баг в интеграции или проблемы на стороне банка).

Часть 6. Итоговая архитектура

Собираем компоненты:

Компоненты:

- **Payment Gateway:** приём запросов от клиентов, проверка идемпотентности.
- **Orchestrator (Saga Coordinator):** управляет распределённой транзакцией.
- **Ledger Service:** двойная запись, хранение балансов.
- **Внешние эквайринги:** банки (через API).
- **Dead Letter Queue (Kafka):** сбойные транзакции.

Поток успешного платежа:

1. Клиент → Gateway (`idempotency_key`).
2. Gateway → Orchestrator: начать сагу.
3. Orchestrator → Ledger: заблокировать деньги (дебет).
4. Orchestrator → Эквайринг: списать.
5. Эквайринг отвечает через Webhook → Orchestrator.
6. Orchestrator → Ledger: подтвердить кредит.
7. Orchestrator → Notification: отправить квитанцию.
8. Клиент получает `success`.

Поток ошибки (эквайринг отказал):

1. Orchestrator → Ledger: компенсация (дебет — отмена).
2. Orchestrator → Notification: уведомление об ошибке.
3. Клиент получает `failed`.

Итог главы 17

Мы спроектировали платёжную систему, которая не теряет деньги и не списывает дважды.

Ключевые решения:

- **Идемпотентность (Idempotency Key)** — защита от дублей.
- **Saga** — управление распределёнными транзакциями.
- **Двойная бухгалтерия (Double-entry)** — проверяемость баланса.
- **Асинхронная интеграция с банками** через Webhook.
- **Dead Letter Queue** — для сбойных платежей.

Что делать прямо сейчас

1. Внедрите идемпотентность во все запросы, меняющие состояние.
2. Разработайте схему компенсаций для каждого сервиса.
3. Храните все движения в ledger с двойной записью.
4. Никогда не обновляйте баланс «на прямую» — только через события.
5. Периодически запускайте аудит сверки балансов.

Следующая глава — «Система голосования (Выборы 2.0)» — ваш уникальный кейс. Там мы объединим требования надёжности платёжной системы с нагрузкой новостной ленты и добавим долю безумия в виде блокчейна.

Глава 18. Система дистанционного электронного голосования (ДЭГ)

«Обычная система падает — теряются деньги. Система голосования падает — теряется доверие. И восстановить его почти невозможно. ДЭГ сочетает требования банковской надёжности и соцсети с добавлением криптографии. Добро пожаловать в ад архитектора».

Система ДЭГ (дистанционного электронного голосования) — это отдельный зверь. Её требования противоречат друг другу:

- **Тайна голосования:** никто не должен знать, за кого вы проголосовали.
- **Проверяемость:** вы должны убедиться, что ваш голос учтён правильно.
- **Невозможность фальсификации:** нельзя добавить или удалить голоса.
- **Надёжность:** система не должна падать в день выборов.
- **Аудируемость:** после выборов можно проверить всё результат.

В обычной жизни тайна и проверяемость конфликтуют. Если я могу проверить свой голос, значит кто-то может связать меня с ним. Математика даёт выход — криптография.

В этой главе — уникальный кейс, основанный на опыте создания реальной системы ДЭГ. Блокчейн, слепые подписи, пиковые нагрузки и бюрократия. И главное — как мы не упали в день голосования.

Часть 1. Уникальные требования

1.1 Три кита ДЭГ

В отличие от обычного веб-приложения, у ДЭГ три противоречивых требования:

Тайна голосования:

- Никто (включая администраторов базы данных) не должен знать, за кого проголосовал избиратель.
- Даже сам избиратель не сможет доказать, за кого он проголосовал (защита от покупки голосов).

Проверяемость:

- Избиратель может убедиться, что его голос учтён и не потерялся.

- Наблюдатели могут проверить, что все голоса посчитаны честно.

Невозможность фальсификации:

- Нельзя добавить «левые» голоса.
- Нельзя удалить голоса.
- Нельзя изменить уже учтённый голос.

1.2 Дополнительные требования

Надёжность:

- Доступность 99.99% в день голосования.
- Падение системы на час недопустимо — выборы не перенесут.

Производительность:

- Пиковая нагрузка: 1 млн голосов в час (последний час голосования).
- Время обработки голоса: не более 5 секунд.

Безопасность:

- Защита от DDoS.
- Защита от MITM.
- Аудит всех действий.

Легальность:

- Соответствие ФЗ-152 (персональные данные).
- Хранение данных на территории РФ.
- Сертификация ФСТЭК (если требуется).

Часть 2. Почему блокчейн?

Когда заказчик слышит слово «ДЭГ», он сразу говорит: «А давайте на блокчейне!».

Блокчейн решает проблему доверия, но создаёт проблемы производительности.

2.1 Блокчейн vs традиционная база

Аспект	Традиционная БД	Блокчейн
Доверие	Доверяем администратору БД	Доверяем математике
Производительность	10 000+ RPS	10–100 RPS (public) / 1 000+ RPS (private)
Задержка	Миллисекунды	Секунды (для консенсуса)
Удаление	Можно (грустно)	Нельзя
Аудит	По журналам	Встроен

2.2 Наше решение: блокчейн для неизменяемости, БД для скорости

Мы не стали хранить все голоса в блокчейне — это было бы безумием. Вместо этого:

1. **Сами голоса** — в обычной БД.
2. **Хэши голосов** — в блокчейне.
3. Наблюдатели могут проверить, что ни один голос не был изменён.

Поток фиксации:

1. Голос сохранён в БД.
2. Вычисляем хэш.
3. Отправляем хэш в блокчейн.
4. Если кто-то изменит голос в БД, хэш не сойдётся.

2.3 Выбор блокчейн-платформы

Hyperledger Fabric — разрешённый блокчейн от Linux Foundation.

Почему:

- **Приватный:** не нужен майнинг, высокая производительность (до 5 000 транзакций/сек).
- **Смарт-контракты (chaincode):** запись хэшей, проверка целостности.
- **Консенсус Raft:** быстрее Proof of Work.
- **Узлы:** ЦИК, Общественная палата, партии, наблюдатели.

Часть 3. Криптографическая схема: слепая подпись

Как обеспечить тайну и проверяемость одновременно? Классическое решение — **слепая подпись**.

3.1 Аналогия с конвертом

Представьте, что вы пишете бюллетень, кладёте его в конверт и заклеиваете. Затем кладёте этот конверт в специальный прибор (машина слепой подписи), который ставит печать на внешний конверт, не вскрывая его. Комиссия видит только печать, но не содержимое.

Вы забираете конверт с печатью, идёте домой, вскрываете его и опускаете бюллетень в урну. Урна принимает только бюллетени с печатью комиссии. Комиссия не знает, за кого вы проголосовали, но может проверить, что печать подлинная.

3.2 Математика слепой подписи

1. **Избиратель** создаёт бюллетень, шифрует его и «ослепляет» (применяет математическое преобразование).
2. **Система** проверяет право голоса и подписывает ослеплённый бюллетень, не видя содержимое.
3. **Избиратель** снимает «ослепление» и получает подписанный бюллетень.
4. **Избиратель** отправляет бюллетень в блокчейн.

Результат: система знает, что бюллетень подписан ею, но не знает его содержимого. Избиратель может проверить подпись, но наблюдатель не свяжет его с голосом.

Часть 4. Масштабирование и нагрузка

4.1 Оценка нагрузки

- **Пользователей:** 10 млн зарегистрированных.
- **Пиковая нагрузка (последний час):** 1 млн голосов в час ≈ 280 RPS.
- **Средняя нагрузка:** 100–200 RPS.
- **Время обработки одного голоса:** < 3 секунд (криптография).

4.2 Узкие места

1. **Блокчейн-узел:** даже Fabric не даст 10 000 транзакций/сек при сложных смарт-контрактах.
2. **Криптография:** проверка подписей требует CPU.
3. **Синхронизация между узлами:** задержки из-за сети и консенсуса.

4.3 Решения

Шардирование избирателей по регионам:

- Каждый регион — свой канал в Hyperledger Fabric.
- Узлы-наблюдатели имеют доступ к каналам всех регионов.

Предварительная проверка подписей до блокчейна:

- Отдельный сервис валидирует подпись.
- В блокчейн уходит только хэш (меньше данных).

Балансировка криптографических операций:

- Кэширование результатов проверки публичных ключей.
- Вынос криптографии на отдельные воркеры.

Часть 6. Итоговая архитектура

Компоненты

- **Портал голосования:** веб-интерфейс, генерация бюллетеней, криптография на клиенте.
- **Сервис регистрации:** проверка права голоса (интеграция с епархией).
- **Сервис голосования:** приём бюллетеней, проверка подписей, отправка хэшей в блокчейн.
- **Блокчейн-кластер (Hyperledger Fabric):** фиксация хэшей, проверка целостности.
- **Сервис подсчёта:** чтение хэшей из блокчейна, агрегация результатов.
- **ClickHouse:** аналитика (явка, распределение по регионам).

Поток голоса

1. Избиратель авторизуется → Портал.
2. Сервер регистрации проверяет право голоса.
3. Клиент генерирует бюллетень, «слепит», отправляет на подпись.
4. Сервер голосования подписывает «слепой» бюллетень.
5. Клиент «снимает слепоту», получает подписанный бюллетень.
6. Клиент отправляет бюллетень в сервис голосования → запись в БД + хэш в блокчейн.
7. После голосования — сервис подсчёта: агрегация + публикация результатов.

Итог главы 18

Мы спроектировали ДЭГ — систему, сочетающую требования банковской надёжности, нагрузку соцсети и криптографическую сложность.

Ключевые решения:

- **Блокчейн (Hyperledger Fabric)** — фиксация хэшей, а не самих голосов.
- **Слепая подпись** — тайна + проверяемость.
- **Шардирование по регионам** — масштабирование блокчейна.
- **Асинхронная обработка** — очереди для непопулярных бюллетеней.

В следующей главе — «System Design интервью: как проходить и побеждать» — стратегия, ловушки и шаблоны ответов, которые помогут пройти собеседование в любую компанию уровня FAANG.

Глава 19. Процесс прохождения интервью по System Design

«Собеседование по System Design — это не экзамен на знание технологий. Это проверка того, как вы думаете, когда нет правильного ответа»

Вы знаете CAP-теорему, ACID, паттерны проектирования и можете нарисовать архитектуру YouTube за час. Вы готовы к интервью? Возможно. Но знаете ли вы, что оценивает интервьюер, когда смотрит на вашу схему?

За годы проведения собеседований (и прохождения их сам) я заметил: кандидаты часто проваливаются не из-за недостатка знаний. Они проваливаются, потому что не понимают структуру интервью, не слышат интервьюера или не показывают свой образ мыслей.

В этой главе — пошаговая стратегия прохождения собеседования по System Design. От первых вопросов до финального «у нас есть ещё вопросы?».

Часть 1. Что оценивает интервьюер

Прежде чем готовиться к ответам, поймите, что будет оценивать ваш собеседник.

1.1 Критерии оценки (официальные в FAANG)

Аспект	Что проверяется	Вес (примерный)
Работа с требованиями	Способность структурировать неопределённость, задавать правильные вопросы	25%
Архитектурное мышление	Умение видеть систему целиком, trade-offs, компромиссы	30%

Аспект	Что проверяется	Вес (примерный)
Знание паттернов и технологий	Брокеры, БД, кэши, балансировка — обоснованный выбор	20%
Коммуникация	Ясность, умение объяснять, слышать собеседника	15%
Количественная оценка	Оценка нагрузки, ёмкости, задержек	10%

Важный вывод: знание технологий — только 20% успеха. 80% — это то, как вы мыслите, общаетесь и работаете с неопределённостью.

1.2 Чего не нужно делать

Этот список основан на моём опыте интервью, где кандидаты прощались с оффером.

Не молчите. У вас будет 45–50 минут. Интервьюер не может заглянуть в вашу голову. Если вы молчите, он не знает, что вы думаете. Проговаривайте свои мысли вслух даже перед тем, как рисовать схему.

Не уходите в детали на первых минутах. «А ES-модули в nginx можно настроить так, что...» — через 5 минут после старта. Вы утонули в имплементации, забыв о требованиях.

Не выбирайте технологию без обоснования. «Возьмём Kafka». Почему? «Это модно и все берут». Аргумент не катит.

Не говорите «это зависит от контекста» и останавливайтесь. Дальше должно следовать: «Вот при каких условиях я выбрал бы X, а при каких — Y».

Не защищайте решение, которое интервьюер раскритиковал. Он может давить специально, проверяя реакцию. Спокойно объясните: «Я понимаю, что у

этого решения есть минусы, они такие-то. Альтернативно можно сделать так, но тогда придётся заплатить тем-то».

Часть 2. Структура интервью (45 минут)

2.1 Типичный тайминг

Этап	Длительность	Что происходит
Введение и уточнение требований	5–10 мин	Вы задаёте вопросы, интервьюер отвечает
High-level дизайн	5–10 мин	Рисуете основные компоненты и связи
Глубокое погружение	15–25 мин	Детали ключевых компонентов, выбор технологий
Борьба с узкими местами	5–10 мин	«А что будет, если упадёт X?», «А как мы будем масштабировать Y?»
Итоги и вопросы к интервьюеру	2–3 мин	Вы подводите итог, задаёте вопросы

Важно:

- Вы управляете временем, а не интервьюер.
- Если чувствуете, что застряли на деталях, скажите: «Давайте вернёмся к этому позже, сейчас я хочу закончить схему».
- Если времени мало, явно скажите: «У нас осталось 10 минут, поэтому я пропущу детали кэширования и сосредоточусь на БД».

2.2 Вопросы, которые нужно задать в начале

Не начинайте рисовать сразу. Вы должны понять, что именно от вас хотят.

Вопросы про функции:

- «Какие основные операции должна поддерживать система?»
- «Кто пользователи — физические лица, компании, внутренние сервисы?»

Вопросы про нагрузку:

- «Сколько пользователей (DAU/MAU)?»
- «Какая ожидаемая нагрузка в пик (RPS, чтение/запись)?»
- «Сколько мы ожидаем хранить данных? За какой период?»

Вопросы про нефункциональные требования:

- «Что важнее — доступность или согласованность?»
- «Какая задержка допустима для пользователя?»
- «Какие требования к безопасности?»

Вопросы про ограничения:

- «Какой бюджет (hardware, облако)?»
- «Нужно ли поддерживать legacy-системы?»

Часть 3. High-level дизайн

Этот этап покажет, что вы видите лес, а не только деревья.

3.1 Что рисовать на этом этапе

- **Компоненты уровня системы:** клиенты, API Gateway, сервисы (Auth, Core, Notification), хранилища (БД, кэш, очереди).
- **Связи между ними:** стрелки с подписями (какой протокол, синхронный/асинхронный).
- **Ключевых внешних систем:** CDN, эквайринги.

НЕ нужно рисовать:

- Детали реализации отдельных сервисов (внутренние классы, таблицы БД).
- Конфигурацию окружения (K8s pods, реплики).

3.2 Открытые вопросы на этом этапе

Интервьюер может спросить:

- «Почему вы выбрали микросервисы, а не монолит?»
- «Зачем здесь нужен кэш?»
- «Какие проблемы может создать такой дизайн?»

Пример ответа на вопрос о микросервисах:
«Я выбрал микросервисы, потому что... (аргумент). Но я понимаю, что это приносит сложность распределённых транзакций и сетевых задержек. Если бы команда была меньше, я бы предпочёл монолит».

3.3 Пример схемы (для Twitter)

На уровне схема включает:

text

```
[Клиент] → [API Gateway] → [Fanout Service] → [Timeline Cache (Redis)]
                                ↓
                          [Post Storage (Cassandra)]
```

Часть 4. Глубокое погружение

Вы выбрали главные компоненты, теперь интервьюер хочет понять, действительно ли вы их знаете.

4.1 Какие темы затрагивает погружение

- **Базы данных:** выбор между SQL/NoSQL, схема шардирования, индексы.
- **Кэширование:** стратегия (cache-aside, write-through), инвалидация, TTL.
- **Очереди:** зачем нужна, какой брокер, гарантии доставки.
- **Регуляция:** rate limiting, circuit breaker, retry.

4.2 Как он проверит, плаваете ли вы

Примерные вопросы:

- «Как вы будете шардировать данные, чтобы избежать горячих ключей?»

- «Что произойдёт, если кэш внезапно потеряет все данные?»
- «Как вы обработаете случай, когда сообщение в очереди не может быть доставлено 3 раза?»

4.3 Шаблон ответа глубокого уровня

1. **Объясните выбор.**
2. **Назовите минусы.** *«У этого решения есть недостаток...»*
3. **Предложите альтернативу.** *«Другой подход... но он дороже в поддержке».*

Часть 5. Борьба с узкими местами

Интервьюер: «А что будет, если этот сервис упадёт?».

Он проверяет, думали ли вы об отказоустойчивости.

5.1 Типичные проблемные места

- **Single point of failure (SPOF):** единая точка отказа. «Что, если API Gateway упадёт?».
- **Database overload:** перегрузка БД. «А если у нас 1 млн RPS, а ваша БД держит только 10k?».
- **Hot partitions:** горячие партиции. «Все запросы идут на один шард, как быть?».
- **Network partition:** сбой в сети. «Возможна ли запись при разделении дата-центров?».

5.2 Как отвечать на «упал сервис»

Золотой принцип: **Называйте компромиссы.**

«Если сервис А упадёт, мы можем...
(1) отключить зависимую функцию и показывать fallback (например, закешированные данные),
(2) переключить трафик на реплику в другой зоне (active-passive),
(3) вернуть пользователю ошибку, когда отказоустойчивость обходится дороже.
Выбор зависит от того, насколько критична функция».

Итог главы 19

Мы разобрали структуру System Design интервью:

- 50% успеха — **вопросы в начале** (получите требования и ограничения).
- 20% — **high-level дизайн** (компоненты и связи).
- 20% — **глубокое погружение** (выбор БД, кэшей, очередей).
- 10% — **устойчивость к сбоям** (что делать при отказе).

Что делать завтра: Потренируйтесь в мысленном интервью. Возьмите систему (например, YouTube) и проговорите вслух все этапы, даже если рядом никого нет.

Следующая глава — «Частые вопросы и как на них отвечать» — где мы разберём самые трудные вопросы и превратим их в ваши козыри.

Глава 20. Частые вопросы и как на них отвечать

«Интервьюер не хочет услышать "правильный" ответ. Его не существует. Он хочет услышать, как вы думаете»

После того как вы освоили структуру интервью (глава 19), пришло время поговорить о конкретных вопросах. В этой главе — 10 вопросов, которые мне задавали на собеседованиях, и 10 вопросов, которые я задавал кандидатам.

Каждый разбор включает:

- **Что проверяет интервьюер** (зачем он вообще это спросил)
- **Как НЕ надо отвечать** (типичные ошибки)
- **Как надо отвечать** (стратегия и пример)

Запомните главное: интервьюер не ищет «единственно верный ответ». Он хочет понять, как вы мыслите, когда нет готового решения.

Часть 1. Вопросы про архитектуру и компромиссы

Вопрос 1. «Почему вы выбрали микросервисы, а не монолит?»

Что проверяет: понимаете ли вы trade-offs. Умеете ли обосновывать архитектурный стиль под конкретные ограничения.

Как НЕ надо отвечать: «Микросервисы — это современно, все так делают».

Как надо отвечать: шаблон ответа — перечислите условия, при которых микросервисы выигрывают, а затем когда монолит лучше.

Пример:
«Я выбрал микросервисы, потому что у нас разные команды, разные графики релизов и требования к масштабированию. Но я понимаю цену: распределённые транзакции, сетевая задержка, сложность эксплуатации. Если бы команда была меньше 10 человек, а нагрузка предсказуема, я бы выбрал монолит».

Вопрос 2. «Почему вы выбрали базу данных X, а не Y?»

Что проверяет: знаете ли сильные и слабые стороны разных типов БД.

Как НЕ надо отвечать: «PostgreSQL быстрее» (без цифр), «MongoDB масштабируется».

Как надо отвечать: сравните требования вашей задачи с характеристиками БД.

Пример для выбора PostgreSQL vs Cassandra:
«Мне нужны сложные запросы (JOIN, агрегации) и строгая согласованность — это PostgreSQL. Cassandra дала бы лучшую запись и масштабируемость, но пришлось бы отказаться от транзакций и сложных запросов. Для этого проекта консистентность важнее».

Вопрос 3. «Как вы будете масштабировать базу данных, когда данных станет в 100 раз больше?»

Что проверяет: понимаете ли шардирование, репликацию и их ограничения.

Как НЕ надо отвечать: «Добавим ещё серверов».

Как надо отвечать: сначала оцените текущую нагрузку, предложите вертикальное масштабирование, а затем — горизонтальное.

Пример:
«Сначала попробую вертикальное масштабирование (если мы не упёрлись в потолок). Если не поможет — шардирование. Для нашей модели ключ шардирования — user_id. Проблема: горячие пользователи. Для них добавлю отдельный кэш. При шардировании понадобится перешардирование, поэтому использую consistent hashing».

Вопрос 4. «Почему здесь синхронный вызов, а не асинхронный?»

Что проверяет: когда вы выбираете синхронность, когда асинхронность и какую цену платите.

Как НЕ надо отвечать: «Асинхронность сложнее».

Как надо отвечать: опишите компромисс.

Пример:
«Здесь нужно подтверждение платежа до того, как показать пользователю результат — синхронность. Если сервис платежей упадёт, пользователь получит ошибку, но это лучше, чем неизвестность. Если бы мы отправляли

email, я бы выбрал асинхронность — пользователь не ждёт, очередь сгладит пики».

Часть 2. Вопросы про отказы и надёжность

Вопрос 5. «Что произойдёт, если этот сервис упадёт?»

Что проверяет: думали ли вы об отказоустойчивости.

Как НЕ надо отвечать: «Этого не случится, у нас хорошие инженеры».

Как надо отвечать: опишите fallback сценарии, retry, circuit breaker.

Пример:
«Если сервис упадёт, запросы пойдут в retry с экспоненциальной задержкой. Если не помогло — circuit breaker откроется, и пользователь увидит fallback: закэшированные данные (для чтения) или сообщение об ошибке (для записи). Администратор получит алерт, и сервис починят».

Вопрос 6. «Как вы будете обеспечивать надежность данных?»

Что проверяет: знание репликации, бэкапов, RPO/RTO.

Как НЕ надо отвечать: «Будем бэкапиться раз в день».

Как надо отвечать: опишите стратегию с RPO/RTO.

Пример:
«Для этого сервиса RPO = 5 минут (теряем не более 5 минут данных). RTO = 1 час. Использую синхронную репликацию на 2 узла, асинхронную — в другой дата-центр. Бэкапы — инкрементальные каждые 5 минут, полные — раз в неделю. Хранение бэкапов — 30 дней».

Вопрос 7. «Что вы будете делать, если запрос стал тормозить, а пользователи жалуются?»

Что проверяет: умеете ли вы диагностировать проблемы.

Как НЕ надо отвечать: «Добавим серверов».

Как надо отвечать: перечислите шаги диагностики и типовые узкие места.

Пример:
«Сначала смотрю на метрики: CPU, память, диск, сеть. Где узкое место? Проверяю медленные запросы в БД (индексы?). Смотрю на кэш (кеш-промахи?). Анализирую логи на наличие ошибок. После нахождения причины принимаю решение: оптимизация запроса, добавление индекса, увеличение кэша, горизонтальное масштабирование узкого компонента».

Часть 3. Вопросы про проектирование и выбор технологий

Вопрос 8. «Как вы будете выбирать между SQL и NoSQL?»

Что проверяет: знаете требования ACID, нормализацию, масштабирование.

Как НЕ надо отвечать: «SQL для всего, потому что я его знаю».

Как надо отвечать: таблица решений на основе требований.

Пример:
«Если нужны транзакции и сложные запросы (JOIN) — SQL. Если схема гибкая и нужно горизонтальное масштабирование — NoSQL. В этом проекте у нас оба кейса: пользователи (SQL) и лента событий (NoSQL). Использую polyglot persistence».

Вопрос 9. «Какие паттерны отказоустойчивости вы знаете?»

Что проверяет: знание инструментов, а не только слов.

Как НЕ надо отвечать: «Таймауты, ретраи, circuit breaker».

Как надо отвечать: кратко опишите, как и когда работает каждый паттерн.

Пример:
«Для защиты от зависших внешних вызовов — таймаут. Для временных сбоев — retry (экспоненциальная задержка + jitter). Чтобы не долбить упавший сервис — circuit breaker с состояниями closed, open, half-open. Чтобы изолировать ресурсы — bulkhead (раздельные пулы)».

Вопрос 10. «Как бы вы улучшили эту систему?»

Что проверяет: критическое мышление.

Как НЕ надо отвечать: «Всё отлично, ничего менять не нужно».

Как надо отвечать: назовите узкие места, предложите улучшения и компромиссы.

Пример:
«Сейчас единая точка отказа — база данных. Я бы добавил реплику с возможностью переключения при падении. Также кэш для горячих запросов снизит нагрузку на БД в 2-3 раза. Если бюджет позволит, внедрю шардирование для масштабирования на следующий год».

Итог главы 20

Мы разобрали ключевые вопросы, их скрытый смысл и правильную структуру ответа.

Главное правило: не давайте ответ «да/нет». Всегда объясняйте «почему» и «какой ценой».

Что делать прямо сейчас: Потренируйтесь отвечать на эти вопросы вслух перед зеркалом или с другом. Засекайте время (на каждый — не более 2–3 минут).

Следующая глава — «Тренажёр: системы для самостоятельного проектирования» — где вы примените все знания на практике и получите обратную связь (хотя бы от самих себя).

Глава 21. Тренажёр: системы для самостоятельного проектирования

«Архитектура не изучается по книгам. Она изучается на практике. Теория — это карта. Реальные системы — это компас. Только проектируя, вы научитесь выбирать»

Вы прочитали книгу. Вы знаете CAP-теорему, ACID, паттерны, микросервисы. Но готовы ли вы к реальному собеседованию или работе над проектом? Единственный способ проверить — взять чистый лист и начать проектировать.

Эта глава — тренажёр. Четыре системы разного уровня сложности. Никаких подсказок в тексте — только требования и ограничения. Ваша задача — пройти полный цикл: от сбора требований до итоговой схемы.

После каждой системы — контрольные вопросы для самопроверки.

Часть 1. Система бронирования билетов (Ticketmaster)

Сложность: средняя. Время на проектирование: 45–60 минут.

Контекст

Вы делаете систему для продажи билетов на концерты и спортивные события. Билеты на популярные мероприятия раскупаются за секунды. Два пользователя не должны купить одно и то же место.

Требования

Функциональные:

- Поиск событий (по городу, дате, исполнителю).
- Выбор мест в зале (интерактивная карта) и их бронирование.
- Оплата билетов (через внешний платёжный шлюз).
- Генерация QR-кода после оплаты.

- Возврат билетов (с комиссией).

Нефункциональные:

- Доступность: 99.9%.
- В пик продаж (>1 млн пользователей одновременно) система не должна падать.
- Время отклика при выборе мест: < 200 мс.
- Обработка платежа: < 5 сек.

Ограничения:

- Бюджет средний, управляемые БД (облачные) предпочтительнее.
- Команда из 10 разработчиков.

❓ Вопросы для самопроверки

1. Как вы обеспечите, что одно место не будет продано дважды в момент штурма?
2. Какую базу данных выберете для хранения мест и бронирований? Почему?
3. Нужен ли здесь кэш? Если да, то какой стратегии придерживаться и когда его инвалидировать?
4. Как спроектируете взаимодействие с платёжным шлюзом? Синхронно или асинхронно?
5. Как будете обрабатывать таймаут при оплате? Как не потерять бронирование?
6. Как будете генерировать QR-коды и защищать их от подделки?

Короткий разбор (подглядеть после того, как спроектируете сами)

Главная проблема — race condition на последний билет.
Решение: использовать базу данных с транзакциями, которые проверяют доступность места перед бронированием. Бронирование создаётся на 5-10 минут, затем падает, если не оплачено. Для пиковых нагрузок — добавить очередь (Kafka). Для чтения свободных мест — кэш в Redis (инвалидация при бронировании).

Часть 2. Система поиска отелей (Booking / Airbnb)

Сложность: выше средней. Время: 50–60 минут.

Контекст

Система бронирования отелей. Основное — быстрый поиск доступных вариантов по тысячам фильтров (даты, цена, рейтинг, удобства, отзывы).

Требования

Функциональные:

- Поиск по датам, городу, цене, типу номера, удобствам.
- Сортировка по цене, рейтингу, расстоянию до центра.
- Детальная страница отеля с отзывами и фото.
- Бронирование и оплата (интеграция с платёжным шлюзом).

Нефункциональные:

- Время поиска: < 500 мс для 99% запросов.
- 100 000 RPS в пик (новогодние праздники).
- Доступность: 99.99%.

Ограничения:

- Отели добавляют новые цены и свободные номера ежедневно.
- Данные об отелях не меняются часто (раз в день — достаточно).

❓ Вопросы для самопроверки

1. Какую базу данных выберете для хранения отелей и поиска (Elasticsearch? Postgres? ClickHouse?) и почему?
2. Как обеспечите актуальность информации о свободных номерах при бронировании?

3. Как спроектируете фильтрацию и сортировку? Какие индексы помогут?
4. Нужна ли денормализация данных для ускорения поиска?
5. Как будете обрабатывать нагрузку в 100 000 RPS? Шардирование? Кэширование?

Мой короткий разбор (подглядеть после)

Главная сложность — быстрый поиск по многим полям.
Решение: денормализованные данные в Elasticsearch. Отели обновляются пачками раз в час. Свободные номера и цены — отдельный сервис (Redis + Cassandra). API поиска ходит в ES (основные данные), свободные номера — отдельный запрос в Redis.

Часть 3. Система доставки еды (Delivery Club / Uber Eats)

Сложность: высокая. Время: 60–75 минут.

Контекст

Платформа для заказа еды: пользователи выбирают блюда из меню ресторана, платят, курьер доставляет. Главная сложность — трекинг курьеров в реальном времени и оптимальное назначение заказов.

Требования

Функциональные:

- Просмотр списка ресторанов, меню, отзывов.
- Оформление заказа с выбором адреса и способа оплаты.
- Трекинг курьера в реальном времени (позиция на карте).
- Уведомления о статусе заказа.
- Оценка доставки.

Нефункциональные:

- 1 млн заказов в день.
- Доступность: 99.99%.
- Трекинг курьеров: обновление координат каждые 5–10 секунд.

Ограничения:

- Георасределено: рестораны и курьеры по всему городу.

❓ Вопросы для самопроверки

1. Какую базу данных выберете для хранения заказов? Почему?
2. Как будете назначать курьера на заказ (matching)?
3. Как оптимизируете гео-запросы (координаты рядом)?
4. Как обеспечите согласованность статуса заказа между сервисами?
5. Как спроектируете трекинг (WebSocket/polling)?

Мой короткий разбор (подглядеть)

Главная сложность — геопространственные запросы и matching курьера. Решение: PostgreSQL с PostGIS для хранения текущих координат курьеров + Redis Geo для быстрого поиска ближайших. Matching — отдельный сервис с очередью заказов (Kafka) и воркерами, которые подбирают курьера по алгоритму (приоритет близость + текущая загрузка). Трекинг через WebSocket.

Часть 4. Система аналитики посещаемости (Google Analytics)

Сложность: очень высокая. Время: 75–90 минут.

Контекст

Система сбора и анализа событий с сайтов и мобильных приложений: просмотры страниц, клики, конверсии. Данные поступают десятками миллиардов в день.

Требования

Функциональные:

- Сбор событий в реальном времени.
- Хранение и агрегация.
- Выборка по периодам, странам, устройствам.
- Когортный анализ (удержание пользователей).

Нефункциональные:

- 100 млрд событий в день.
- Задержка сбора < 1 сек.
- Отчёты < 5 сек на выборках за год.

Ограничения:

- Open source предпочтительнее (бюджет ограничен).
- Команда из 20 инженеров.

❓ Вопросы для самопроверки

1. Какой тип БД выберете для сырых событий (колоночная/аналитическая)?
2. Как будете агрегировать данные для быстрых отчётов?
3. Какую схему партиционирования и шардирования примените?
4. Как обработаете 100 млрд событий/день? Примерно сколько строчек в секунду?

Мой короткий разбор

Главная сложность — объём данных.
Решение: сырые события в Kafka → потоковое агрегирование в ClickHouse. Секундные пачки (окна). ClickHouse — колоночная, партиционирование по дате, шарды по site_id. Для быстрой отчётности — материализованные вьюхи (агрегаты по дням, странам). Приём данных через HTTP API с балансировкой.

Контрольные вопросы после проектирования

Архитектура:

- Учтены ли все функциональные требования?
- Где единые точки отказа? Как они исключены или чем компенсируются?
- Какие trade-offs вы приняли (согалсованность vs доступность, стоимость vs задержка)?

Надёжность:

- Что будет, если база данных упадёт на час?
- А если упадёт очередь (Kafka)? Есть ли реплика?
- Как обеспечивается целостность: транзакции, идемпотентность?

Масштабирование:

- Какая сущность станет узким местом при росте в 10–100 раз?
- Есть ли ограничения по ежедневному объёму данных?

Эксплуатация:

- Как тестировать систему под нагрузкой?
- Как будете мониторить (метрики, алерты)?

Итог главы 21

Вы прошли 4 системы разной сложности.

- Если с лёгкостью справились с 1–2 системами (бронирование билетов, отели) — вы готовы к позиции Middle архитектора или кандидата на интервью.
- Если доехали до систем доставки еды или аналитики, не запнувшись — добро пожаловать в Senior/Lead.

Дальше — справочник (приложения). Теперь дело за практикой. Берите новую систему (например, YouTube) и проектируйте за час. С каждым разом будет легче.

Следующая глава — «Шпаргалки по базам данных, брокерам и кэшированию». Библия архитектора на каждый день.

Глава 22. Шпаргалки по базам данных, брокерам и кэшированию

«Архитектор не может помнить всё. Но он должен знать, где быстро найти ответ. Эти шпаргалки — моя настольная книга. Теперь они станут и вашей».

Вы спроектировали не одну систему. Вы знаете, как выбирать между разными технологиями. Но в реальной работе часто нужно быстрое решение — без долгих исследований и POC.

Эта глава — набор шпаргалок для принятия решений. Когда вы сомневаетесь между PostgreSQL и MongoDB, между Kafka и RabbitMQ, между Cache-Aside и Write-Through, просто откройте нужную таблицу. Здесь нет «единственно верного» ответа, но есть **направление мысли** и **типичные компромиссы**.

Часть 1. Выбор базы данных

Таблица 1.1. Когда какую БД выбирать

Сценарий	Рекомендация	Почему
Транзакции, сложные запросы, строгая согласованность	PostgreSQL / MySQL	ACID, поддержка JOIN, индексы,成熟
Огромные объёмы данных, горизонтальное масштабирование, высокая запись	Cassandra / ScyllaDB	Нет единой точки отказа, отличная запись, децентрализация
Гибкая схема, быстрое прототипирование	MongoDB	Быстрая разработка, легко менять структуру

Сценарий	Рекомендация	Почему
Аналитика, агрегации на миллиардах строк	ClickHouse	Колоночное хранение, сжатие, быстрые агрегации (в 100-1000 раз быстрее PostgreSQL)
Кэш, сессии, лидерборды	Redis	Хранение в оперативной памяти, микросекундные задержки
Графы связей (соцсети, рекомендации)	Neo4j / ArangoDB	Нативный обход графа (SQL с рекурсивными JOIN'ами будет тормозить)
Полнотекстовый поиск	Elasticsearch	Инвертированные индексы, relevance scoring
Временные ряды (IoT, метрики)	InfluxDB / TimescaleDB	Специализированное сжатие, retention policies

Таблица 1.2. Сравнение SQL vs NoSQL

Критерий	SQL	NoSQL
Схема	Фиксированная (миграции)	Гибкая
Транзакции	ACID (full)	Ограниченные (часто только на уровне записи)

Критерий	SQL	NoSQL
Масштабирование	Вертикальное (сложно горизонтальное)	Горизонтальное (из коробки)
Запросы	Сложные (JOIN, GROUP BY)	Простые (по ключу / диапазону)
Согласованность	Строгая	Часто итоговая
Зрелость	Высокая (десятки лет)	Средняя (5-15 лет)

Правило выбора: начинайте с PostgreSQL. Переходите на NoSQL, когда упрётесь в масштабирование, или схема станет слишком гибкой для миграций.

Часть 2. Выбор брокера сообщений

Таблица 2.1. Kafka vs RabbitMQ vs других

Критерий	Kafka	RabbitMQ	Redis Pub/Sub	SQS (AWS)
Модель	Pub-sub (лог)	Очередь / обменник	Pub-sub	Очередь / FIFO
Хранение сообщений	Да (дни-недели, replay)	Нет (удалил — прочитал)	Нет	Да (до 14 дней)
Порядок сообщений	В партиции	Нет (не гарантирован)	Нет	Да (FIFO очередь)

Критерий	Kafka	RabbitMQ	Redis Pub/Sub	SQS (AWS)
Пропускная способность	Очень высокая (100k+ msg/сек)	Высокая (20-50k)	Очень высокая (зависит от сети)	Высокая (ограничена AWS)
Сложность	Высокая	Средняя	Низкая	Низкая (managed)
Гарантии доставки	At-least-once, exactly-once (транзакции)	At-most-once / at-least-once	At-most-once	At-least-once (FIFO — exactly-once)

Таблица 2.2. Когда что выбирать

Сценарий	Рекомендация	Почему
Логи, аудит, события, потоковая обработка	Kafka	Хранение сообщений, replay, высокая пропускная способность
Задачи фона (отправка email, генерация отчётов)	RabbitMQ	Простота, очереди и обменники
Быстрый обмен событиями в рамках одного узла (без сохранения)	Redis Pub/Sub	Микросекундные задержки

Сценарий	Рекомендация	Почему
Простая очередь без администрирования (в облаке)	SQS	Managed, не думаете об инфраструктуре

Часть 3. Стратегии кэширования

Таблица 3.1. Когда какую стратегию использовать

Стратегия	Описание	Когда использовать
Cache-Aside (Lazy Loading)	Приложение само проверяет кэш, при промахе — грузит из БД	Большинство случаев, read-heavy
Read-Through	Кэш сам умеет загружать данные из БД	Хотите скрыть сложность от приложения
Write-Through	Запись в кэш и БД синхронно	Нужна актуальность кэша после записи
Write-Back (Write-Behind)	Запись в кэш, затем асинхронно в БД	Очень высокая интенсивность записи (логи, события)
Write-Around	Запись в БД, кэш пропускается	Редко читаемые данные (экономия кэша)

Таблица 3.2. Стратегии вытеснения (Eviction)

Алгоритм	Описание	Когда использовать
LRU (Least Recently Used)	Вытеснение давно неиспользуемых	Универсальный выбор
LFU (Least Frequently Used)	Вытеснение редко используемых	Есть «горячий» набор ключей
FIFO (First In First Out)	Вытеснение самых старых	Временные данные (сессии, логи)
TTL (Time To Live)	Вытеснение по истечении времени	Самый простой, не требует сложной логики

Часть 4. Балансировка нагрузки и алгоритмы

Таблица 4.1. Алгоритмы балансировки

Алгоритм	Описание	Плюсы	Минусы
Round Robin	По очереди	Простота, равномерно под одинаковую нагрузку	Не учитывает разную мощность серверов
Least Connections	Туда, где меньше соединений	Учитывает разную длительность запросов	Нужно отслеживать соединения

Алгоритм	Описание	Плюсы	Минусы
Least Time	Туда, где меньше времени ответа	Самый адаптивный	Сложность, возможны скачки
IP Hash	hash(IP) % N	Липкие сессии	При изменении N — всё перехешируется
Random	Случайный выбор	Простота, отсутствие состояния	Непредсказуемость

Таблица 4.2. Где размещать балансировку

Место	Плюсы	Минусы
DNS	Простота, географическая маршрутизация	Кэширование DNS, нет учёта загрузки
L4 (транспортный уровень)	Быстро, низкоуровнево, работает для любых протоколов	Не видит содержимого запроса
L7 (прикладной уровень, HTTP)	Маршрутизация по URL, заголовкам, cookie	Медленнее, сложнее

Часть 5. Оценка нагрузки (формулы быстрого доступа)

Эти формулы пригодятся на этапе сбора требований, чтобы примерно прикинуть, сколько серверов нужно, каков будет трафик и объём данных.

5.1 Оценка хранилища

Общий объём = (размер одной записи в байтах) × (количество записей в день) × (дней хранения) × (K сжатия).

K сжатия:

- Текст (JSON/CSV) — 0.2–0.5 (хорошо сжимается).
- Изображения — 0.9–1 (почти не сжимаются, уже сжаты).
- Видео — 1.0 (не сжимаются).

5.2 Оценка пропускной способности сети

Мегабиты в секунду (Mbps) = (размер ответа в Мбитах) × RPS.

5.3 Оценка количества серверов

Серверов = (максимальная нагрузка в RPS на один сервер) ÷ (RPS, который мы хотим выдержать).

Пример: хотим выдержать 10 000 RPS. Один сервер (n1-highcpu) держит 2 000 RPS. Нужно 5 серверов + запас 20–30%.

5.4 Карманный справочник

- **1 день** = 86 400 секунд.
- **1 месяц (30 дней)** = 2 592 000 секунд.
- **1 год** = 31 536 000 секунд.

1 Гбит/с = 125 МБ/с.
1 ТБ = 1 000 ГБ.

Бонус: чек-лист перед принятием архитектурного решения

- **Согласованность** — нужна ли строгая или допустима итоговая?
- **Транзакции** — нужны ли распределённые (через компенсации) или локальные?

- **Объём данных** — сколько записей в день, год? Впишется ли в диск SSD одной машины?
- **Нагрузка чтения / записи** — какой RPS (чтение/запись) в пик? Во сколько раз чтение больше записи?
- **Задержка (latency)** — допустимо ли 500 мс? Или нужно <50 мс (тогда кэш и SSD)?
- **Доступность** — допустим ли простой на 1 час в год (99.99%)? Или нужно 99.999% (5 минут в год)?
- **Опыт команды** — знают ли эту технологию? Готовы ли учиться?
- **Бюджет** — open source / managed service / платная лицензия?
- **Время выхода на рынок** — сколько дней/недель можно потратить на инфраструктуру?

Итог главы 22

Мы собрали шпаргалки по ключевым архитектурным решениям.

- **Базы данных:** от PostgreSQL до Cassandra.
- **Брокеры:** от Kafka до SQS.
- **Кэширование:** стратегии и алгоритмы вытеснения.
- **Балансировка:** алгоритмы и уровни.
- **Оценка нагрузки:** формулы для быстрого старта.

Теперь у вас есть карманный справочник для собеседований и реальных проектов. Когда вы сомневаетесь между технологией А и Б, откройте эту главу и пройдите по чек-листу.

Следующая глава — «Глоссарий терминов». Потому что архитекторы тоже иногда забывают разницу между RPO и RTO.

Глава 23. Глоссарий терминов

«Архитекторы говорят на языке, который должен понимать не только разработчик, но и бизнес. Этот глоссарий — ваш переводчик»

В этой главе — все ключевые термины, которые встречались в книге. От ACID до CRDT, от CAP до TTL. Если вы забыли разницу между RPO и RTO или что такое backpressure — открывайте и читайте.

A

ACID (Atomicity, Consistency, Isolation, Durability) — набор свойств транзакций в реляционных БД. Атомарность (всё или ничего), согласованность (переход из одного корректного состояния в другое), изоляция (параллельные транзакции не мешают друг другу), долговечность (данные переживают перезагрузку).

ADR (Architecture Decision Record) — документ, фиксирующий архитектурное решение. Контекст → варианты → выбор → последствия.

API Gateway — единая точка входа для всех клиентов. Занимается маршрутизацией, аутентификацией, rate limiting, агрегацией ответов, логированием.

AP (Availability + Partition Tolerance) — в CAP-теореме: система выбирает доступность и устойчивость к разрыву, жертвуя согласованностью.

At-least-once delivery — гарантия доставки сообщения (брокер доставит хотя бы один раз). Возможны дубли.

At-most-once delivery — гарантия, что сообщение будет доставлено не более одного раза (может потеряться).

Active-Active — оба узла (дата-центра) активны, трафик распределяется между ними.

Active-Passive — активен один узел, второй находится в режиме ожидания.

B

Backpressure — механизм, при котором получатель сообщает отправителю, что он перегружен и нужно замедлиться. Используется в очередях и реактивных системах.

Base62 — система счисления (цифры + строчные + прописные буквы). Используется для генерации коротких идентификаторов.

Bounded context — границы, в которых определённая модель данных имеет смысл. Термин из DDD.

Bulkhead (перегородка) — паттерн устойчивости. Изоляция ресурсов, чтобы падение одного компонента не влияло на другие.

Blue-Green deployment — стратегия деплоя. Две идентичные среды (Blue — текущая, Green — новая). После тестирования трафик переключается на Green.

C

C4 model — способ визуализации архитектуры на 4 уровнях: Context → Containers → Components → Code.

CAP theorem — в распределённой системе из трёх свойств: Consistency (согласованность), Availability (доступность), Partition Tolerance (устойчивость к разрыву сети) — можно гарантировать не более двух одновременно.

Cache-Aside (Lazy Loading) — стратегия кэширования. Приложение само проверяет кэш, при промахе загружает данные из БД.

Canary deployment — постепенное переключение трафика на новую версию (1% → 5% → 20% → 100%).

CDC (Change Data Capture) — отслеживание изменений в БД и превращение их в события (обычно через Kafka для синхронизации с аналитикой).

CDN (Content Delivery Network) — географически распределённая сеть серверов для быстрой доставки статического контента.

Circuit Breaker (предохранитель) — паттерн устойчивости. При большом числе ошибок перестаёт отправлять запросы к проблемному сервису.

Consistent Hashing (согласованное хэширование) — алгоритм шардирования, при котором добавление/удаление узлов затрагивает только K/N данных.

Correlation ID — уникальный идентификатор запроса, который прокидывается через все сервисы для связывания логов и трейсов.

CP (Consistency + Partition Tolerance) — в CAP-теореме: система выбирает согласованность и устойчивость к разрыву, жертвуя доступностью.

CQRS (Command Query Responsibility Segregation) — разделение модели записи (команды) и чтения (запросы). Позволяет оптимизировать их независимо.

CRDT (Conflict-Free Replicated Data Type) — структура данных, которая гарантирует сходимость при репликации без явного разрешения конфликтов (используется в распределённых системах и базах данных).

D

Database per service — паттерн микросервисной архитектуры: каждый сервис имеет свою БД.

DDD (Domain-Driven Design) — подход к проектированию, фокусирующийся на предметной области, а не на технологиях.

Dead Letter Queue (DLQ) — очередь для сообщений, которые не удалось обработать. Позволяет не терять данные и анализировать проблемы.

Durability — см. ACID.

E

EDA (Event-Driven Architecture) — архитектурный стиль, в котором компоненты общаются через события. Сервисы не вызывают друг друга напрямую, а публикуют события.

Eventually consistent (итоговая согласованность) — модель согласованности: система не гарантирует, что читатель увидит свежую запись сразу, но если новых записей нет, все читатели придут к одному значению.

Exactly-once delivery — гарантия доставки сообщения без потерь и дублирования. Сложна в реальных распределённых системах (часто приближается к at-least-once + идемпотентность).

Error budget — допустимый процент ошибок или недоступности системы (100% – SLO).

Erasure coding — метод хранения данных, при котором исходные данные разбиваются на фрагменты, и система может восстановить данные, имея лишь их часть (используется в distributed storage вместо 3-way replication для экономии места).

F

Fallback — запасной план при отказе основного сервиса: кэшированные данные, упрощённая версия, сообщение об ошибке.

Fanout — распространение одного сообщения многим подписчикам (в push-модели ленты новостей).

G

Gossip protocol — протокол распространения информации в распределённых системах (каждый узел обменивается данными со случайными соседями). Используется в Cassandra, DynamoDB.

Graceful degradation — свойство системы: при отказе части компонентов продолжать работу, пусть и с ограниченной функциональностью.

H

High availability (HA) — способность системы оставаться доступной (обычно 99.9% или выше).

Horizontal scaling (масштабирование горизонтальное) — добавление узлов (серверов) для увеличения ёмкости.

I

Idempotency (идемпотентность) — свойство операции: многократное выполнение даёт тот же результат, что и однократное. Критично для платёжных систем.

Idempotency key — уникальный ключ, передаваемый клиентом, чтобы сервер не обработал запрос дважды.

Isolation — см. ACID.

K

Kappa architecture — упрощённый вариант лямбда-архитектуры: один слой потоковой обработки вместо двух (batch + stream).

L

Lambda architecture — обработка данных через два слоя: пакетный (batch) и скоростной (speed) + слой сервировки (serving).

Leader election (выбор лидера) — механизм выбора одного узла из кластера для координации.

Leaky bucket (алгоритм) — rate limiting через фиксированную ширину отверстия и постоянную скорость утекания.

Linearizability (линеаризуемость) — сильнейшая модель согласованности: операция становится видна всем атомарно между её началом и концом.

Liveness probe (Kubernetes) — проверка, что процесс жив (иначе pod перезапустится).

LRU (Least Recently Used) — стратегия вытеснения кэша, удаляющая давно неиспользуемые ключи.

M

Materialized view (материализованное представление) — предвычисленный запрос, хранящийся как физическая таблица (для ускорения чтения).

Microservices — архитектурный стиль, при котором система состоит из маленьких, независимо развёртываемых сервисов.

Monolith (монолит) — приложение, в котором все компоненты тесно связаны и развёртываются вместе.

MTTR (Mean Time To Recovery) — среднее время восстановления после отказа.

Multi-leader replication (многолидерная репликация) — несколько узлов принимают запись и синхронизируются между собой.

N

N+1 problem — проблема дублирования запросов при работе с ORM. Пример: 10 пользователей → каждый запрос на его заказы → 11 запросов вместо 2.

N+1 redundancy — N серверов + 1 запасной для отказа.

O

Observability (наблюдаемость) — способность понять состояние системы по её внешним проявлениям (логи, метрики, трейсы). Три столпа: Logging, Metrics, Tracing.

Orchestration (оркестрация) — централизованное управление рабочими процессами: один координатор говорит участникам, что делать.

Outbox pattern — извлечение сообщений из БД для последующей отправки в очередь (чтобы не потерять сообщение при сбое).

P

PACELC theorem — расширение CAP. Если работает Partition (P), выбираем между A и C. Else (E, нормальный режим) выбираем между L (latency) и C (consistency).

Partition tolerance — свойство системы продолжать работу при потере связи между узлами.

Parquet — колоночный формат хранения данных (Hadoop-экосистема). Используется для аналитики.

Polyglot persistence (полиглот персистенции) — использование нескольких типов БД в одной системе (например, PostgreSQL + MongoDB + ClickHouse).

ProRata (модель дохода) — автор получает доход от подписки пропорционально прочитанному времени (Литрес).

Pub-sub (Publish-Subscribe) — модель обмена сообщениями: отправитель публикует в топик, все подписчики получают.

R

Raft — алгоритм консенсуса для распределённых систем (альтернатива Paxos). Используется в etcd, Consul, MongoDB.

Rate limiting (ограничение скорости) — ограничение количества запросов от клиента за единицу времени.

Read-your-writes — модель согласованности: читатель видит свои собственные записи.

Read-Through (кэш) — кэш сам умеет загружать данные при промахе.

Replica (реплика) — копия данных на другом узле.

Resharding (перешардирование) — процесс перераспределения данных при изменении количества шардов.

Retry — повтор запроса после временной ошибки.

Rollback (откат) — возврат системы к предыдущей версии (code + data).

Rolling update (катящийся деплой) — постепенная замена старых подов на новые без даунтайма.

RPO (Recovery Point Objective) — сколько данных допустимо потерять при аварии. RPO = 5 минут: можно потерять 5 минут данных.

RTO (Recovery Time Objective) — за сколько система должна восстановиться после аварии. RTO = 4 часа: восстановление займёт до 4 часов.

S

Saga — паттерн для распределённых транзакций: последовательность локальных транзакций с компенсирующими операциями.

Scalability (масштабируемость) — способность системы увеличивать производительность при добавлении ресурсов.

Serverless (бессерверные вычисления) — модель, при которой разработчик не управляет серверами (платит только за время выполнения).

Service discovery (обнаружение сервисов) — нахождение адресов сервисов в динамическом окружении.

SLA (Service Level Agreement) — юридический договор с клиентом о качестве сервиса (с компенсациями).

SLI (Service Level Indicator) — метрика качества (например, время ответа). Числовая характеристика.

SLO (Service Level Objective) — целевое значение SLI (например, 99.9% запросов за 500 мс).

Strong consistency (строгая согласованность) — после записи все читатели видят её (атомарно). См. линеаризуемость.

STAR (Situation, Task, Action, Result) — модель для ответов на поведенческие вопросы на собеседовании.

T

Throttling (троттлинг) — замедление обработки без полного отказа (в отличие от rate limiting). Используется для защиты от перегрузок.

Throughput (пропускная способность) — количество операций в секунду (RPS, TPS).

Timeout (таймаут) — ограничение времени ожидания ответа.

Token bucket (алгоритм) — rate limiting через пополняющуюся ёмкость токенов.

TOGAF (The Open Group Architecture Framework) — фреймворк для корпоративной архитектуры.

TTL (Time To Live) — время жизни записи в кэше (или очереди).

Trade-off — компромисс между двумя противоречивыми требованиями.

U

Upsert — операция: обновить запись, если существует, или вставить, если нет.

Use Case — сценарий использования системы.

V

Vertical scaling (вертикальное масштабирование) — увеличение ресурсов одного узла (CPU, RAM, диск).

W

WAL (Write-Ahead Log) — журнал упреждающей записи (PostgreSQL, Cassandra). Все изменения сначала пишутся в лог, затем в таблицу.

Webhook — HTTP-вызов от системы к системе (обычно для асинхронного уведомления). Банки присылают статус платежа на ваш /webhook/payment.

Write-Through, Write-Back, Write-Around — стратегии кэширования: синхронная / асинхронная запись в БД / обход кэша.

X

X-axis scaling — шардирование (горизонтальное масштабирование по копиям данных).

Y-axis scaling — разделение по функциям (микросервисы). **Z-axis scaling** — шардирование по клиентам.

Итог главы 23

Теперь у вас есть не только знания, но и язык, на котором вы сможете обсуждать архитектуру с коллегами и заказчиками. Глоссарий пригодится при подготовке к интервью, чтении технических статей и в реальных проектах.

Следующая глава — «Ресурсы для дальнейшего изучения» — список книг, курсов, каналов и сообществ, которые помогут вам оставаться в тонусе после прочтения этой книги.

Глава 24. Ресурсы для дальнейшего изучения

«Архитектура — это не пункт назначения. Это бесконечное путешествие. Технологии меняются, паттерны эволюционируют, а вы остаётесь студентом на всю жизнь»

Вы дочитали эту книгу до конца. Поздравляю! Но это не финиш, а только старт.

В этой главе — лучшие ресурсы, которые помогут вам оставаться в тонусе. Книги, которые должен прочитать каждый архитектор. YouTube-каналы, где теория оживает. Telegram-сообщества, где можно спросить совета. И инструменты, чтобы тренироваться.

Часть 1. Книги (must read)

Эти книги спасут вас от выгорания, провалов и скуки.

1.1 Архитектура и System Design

Название	Автор	Почему читать	Сложность
Designing Data-Intensive Applications	Martin Kleppmann	Библия современной архитектуры. Базы данных, очереди, репликация, шардирование — без привязки к технологиям	Высокая
Software Architecture: The Hard Parts	Neal Ford, Mark Richards	Про компромиссы. Как принимать решения, когда нет правильных ответов	Высокая

Название	Автор	Почему читать	Сложность
Building Microservices	Sam Newman	Когда нужны микросервисы, а когда — нет. Честно о сложностях	Средняя
Microservices Patterns	Chris Richardson	Практические рецепты: Saga, CQRS, API Gateway, Circuit Breaker	Средняя
Release It!	Michael Nygard	Как системы падают в проде и как сделать их устойчивыми	Средняя
Clean Architecture	Robert Martin	Отделение бизнес-логики от инфраструктуры	Средняя

1.2 Мышление и процессы

Название	Автор	Почему читать
The Phoenix Project	Gene Kim, Kevin Behr, George Spafford	Роман о DevOps и IT-менеджменте. Читается как детектив, меняет мышление
The Unicorn Project	Gene Kim	Продолжение The Phoenix Project. Про разработку и архитектуру
Team Topologies	Matthew Skelton, Manuel Pais	Как организовывать команды для эффективной разработки

Название	Автор	Почему читать
Accelerate	Nicole Forsgren, Jez Humble, Gene Kim	Что реально влияет на скорость и качество (научные исследования)

Часть 2. YouTube-каналы

Канал	Описание	Кому полезно
System Design Interview	Разборы проектирования Twitter, YouTube, Uber	Готовящимся к интервью
Tech Dummies	Объяснения сложных тем простыми словами с картинками	Начинающим
Gaurav Sen	System Design, распределённые системы	Средний уровень
Hussein Nasser	Бэкенд, базы данных, архитектура	Практикующим
CodeOpinion	CQRS, Event Sourcing, DDD	Продвинутым
HighLoad – Александр Безумнов	Русскоязычный канал про высокие нагрузки	Русскоязычным

Часть 3. Telegram-каналы и сообщества

Ищите сообщества по ключевым словам в поиске Telegram.

Рекомендации по поиску:

- Введите в поиске: «архитектура ПО», «highload», «system design», «backend», «devops»
- Обратите внимание на количество участников и активность (посты должны выходить хотя бы раз в 2–3 дня)
- Для профессионального общения ищите чаты с пометкой «чат» или «discussion»

Критерии хорошего канала:

- Регулярные посты (не реже 2–3 раз в неделю)
- Контент по делу (кейсы, разборы, новости) без «воды»
- Активные комментарии и обсуждения
- Отсутствие агрессивной рекламы

Часть 4. Подкасты

Подкаст	Описание	Где слушать
Software Engineering Radio	Глубокие интервью с экспертами	Apple, Spotify, сайт
The InfoQ Podcast	Новости архитектуры и разработки	Apple, Spotify
Ardan Labs Podcast	Go, архитектура, cloud	Apple, Spotify

Подкаст	Описание	Где слушать
Разбор полётов	История падений крутых систем (русский)	Apple, YouTube
Архитектура и не только	Русскоязычный подкаст от практиков	Apple, YouTube

Часть 5. Тренажёры и симуляторы

- **System Design Interview (GitHub)** — поищите репозитории с шаблонами ответов на System Design вопросы.
- **LeetCode (раздел System Design)** — задачи с проектированием.
- **Pramp** — бесплатные mock-интервью с реальными людьми.
- **Interviewing.io** — анонимные технические интервью.
- **AWS Architecture Center** — реальные кейсы и референсные архитектуры от Amazon.

Поиск в GitHub:

- Введите: «system design interview», «awesome architecture», «system-design-primer»
- Изучайте репозитории с высоким количеством звёзд (> 5000)
- Читайте чужие ADR (Architecture Decision Records) в открытых проектах

Часть 6. Бесплатные курсы (с отбором)

Не гонитесь за сертификатами. Ищите курсы, где есть практика и обратная связь.

Что я рекомендую:

- **Курс «Принятие архитектурных решений»** (мой) — 15 недель, практика, разбор ваших проектов.

- **Техностим (Mail.ru Group)** — бесплатно, но с конкурсным отбором. Сильная программа по highload и архитектуре.
- **Coursera: Software Design and Architecture** (University of Alberta) — академично, глубоко, на английском.
- **YouTube-лекции MIT** по распределённым системам.

Критерии хорошего курса:

- Есть обратная связь от преподавателя или ассистентов
- Не только видео, но и задания с кодом/проектированием
- Сообщество (чат, форум) для обсуждения
- Реальные кейсы, а не абстрактные примеры

Бонус: как не забросить обучение

План на послепрочтения:

Неделя 1:

- Выберите 1 книгу из списка, которую не читали.
- Прочитайте первую главу. Сделайте конспект 3–5 тезисов.

Неделя 2–4:

- Прочитайте книгу полностью. Делайте заметки в Notion/Obsidian, выписывайте идеи, которые примените в работе.

Месяц 2:

- Найдите Telegram-канал по архитектуре. Читайте 10 минут в день.
- Напишите комментарий или задайте вопрос.

Месяц 3:

- Пройдите один бесплатный курс или посмотрите 3 видео с YouTube.
- Попробуйте спроектировать небольшую систему (например, URL shortener) с нуля.

Месяц 6:

- Начните вести блог или писать посты в Telegram о том, что узнали.
- Выступите с докладом на внутреннем митапе в компании.

Итог главы 24

Мы собрали лучшие ресурсы для дальнейшего изучения:

- **Книги** от Клеппмана до Team Topologies.
- **YouTube** для визуалов.
- **Telegram** для живого общения.
- **Подкасты** для дороги и спортзала.
- **Тренажёры** для подготовки к интервью.
- **Бесплатные курсы** (мой в том числе).

Но главный ресурс — это вы.

Следующая глава — «Заключение: архитектор — это на всю жизнь (если не сгорите)». Подводим итоги и смотрим в будущее. Но без слёз (почти).

Глава 25. Заключение: Архитектор — это на всю жизнь (если не сгорите)

«Архитектура — это не должность. Это способ смотреть на мир. Вы не перестаёте быть архитектором, когда уходите с работы. Вы просто меняете систему, которую проектируете»

Мы прошли долгий путь. От фундаментальных принципов до проектирования реальных систем, от интервью до справочника архитектора. Вы прочитали 24 главы. Вы разобрали URL Shortener, мессенджер, новостную ленту, платёжную систему и даже ДЭГ.

Но книга заканчивается. Ваш путь — нет.

В этой главе — главные мысли, которые я хочу, чтобы вы вынесли. И напутствие, которое я повторяю себе в трудные дни.

Часть 1. Пять главных уроков, которые я вынес за 10 лет

1. Архитектор — это не должность, а образ мышления

Вы можете называться архитектором, но мыслить как разработчик — видеть только свой модуль, не замечать связей, не думать о последствиях. А можно не иметь официальной должности, но мыслить системно — и быть архитектором по сути.

Должность могут дать или отнять. Образ мышления остаётся с вами навсегда.

2. Идеальных решений не бывает, бывают компромиссы

В архитектуре нет правильных ответов. Есть trade-offs.

Каждый раз, когда вы выбираете технологию, вы чем-то жертвуете: скоростью, надёжностью, деньгами, временем команды. И это нормально.

Плохой архитектор не знает, чем жертвует. Хороший — выбирает осознанно. Великий — умеет объяснить свой выбор бизнесу.

3. Системное мышление важнее любой технологии

Технологии устаревают. PostgreSQL, Kafka, Kubernetes — всё это когда-нибудь заменят чем-то другим.

Умение видеть систему целиком, понимать связи, предвидеть последствия — остаётся с вами навсегда. Учитесь мыслить системно. Это инвестиция, которая окупается всегда.

4. Люди важнее кода в 10 раз

Самый красивый код не спасёт проект, если команда не понимает, зачем она это делает. Самая совершенная архитектура развалится, если заказчик не слышит архитектора, а архитектор — заказчика.

Инвестируйте в коммуникацию. Учитесь слушать, объяснять, убеждать, признавать ошибки и прощать чужие.

5. Вы никогда не будете знать абсолютно всё (и это нормально)

Мир IT меняется слишком быстро. Невозможно уследить за всеми технологиями, паттернами, подходами.

И не нужно.

Важно знать, где искать то, чего вы не знаете. Важно не бояться признавать пробелы. Важно уметь учиться быстро.

Синдром самозванца не проходит. Он просто становится тише, когда вы перестаёте требовать от себя невозможного.

Часть 2. Что делать дальше

Завтра

- Выберите один пет-проект (URL shortener, простой мессенджер) или одну главу книги для перечитывания.
- Спроектируйте архитектуру. Не пишите код, только схемы и ADR.

На месяц

- Пройдите чек-лист самодиагностики (глава 3.4). Честно.
- Выпишите 3 сильные стороны и 3 слабые. На каждую слабую — конкретное действие по улучшению.

На год

- Найдите ментора. В своей компании, на конференциях, в профессиональных чатах.
- Смените проект, компанию или домен — если чувствуете, что застряли.
- Пройдите мой курс «Принятие архитектурных решений». 15 недель, практика, ваши проекты.

План быстрого старта для нетерпеливых

Месяц 1: главы 1–4 (фундамент + дорожная карта). Возьмите ответственность за небольшой модуль на работе. Начните писать ADR.

Месяц 2: главы 5–8 (опыт + принятие решений + коммуникация). Найдите пет-проект и спроектируйте его от начала до конца.

Месяц 3: главы 9–12 (ошибки + собеседования + рост). Обновите резюме. Начните ходить на собеседования (не чтобы уволиться, а чтобы тренироваться).

Часть 3. Мои контакты

Я не исчезаю после этой книги. Вы можете найти меня в:

Telegram-канал «Кухтинов.Архитектура» — t.me/kukhtinov_arch
Свежие кейсы, разборы, анонсы, живое общение.

Личный Telegram — @artur_kukhtinov
По делу. Предупреждаю, вопросов много, я один. Отвечаю выборочно.

Почта — arthur.kukhtinov@gmail.com
Для долгих писем, предложений, сотрудничества.

Часть 4. Приглашение в закрытый клуб и на курс

Если вы хотите большего, чем книга:

- Разбор вашего проекта с моей обратной связью.
- Регулярные живые встречи с разбором кейсов.
- Сообщество таких же начинающих и практикующих архитекторов.

Добро пожаловать в закрытый клуб «Архитектор будущего».
Ссылка — в Telegram-канале.

А для тех, кто хочет пройти системный курс из 15 недель с живыми вебинарами, домашними заданиями и сертификатом — есть курс «Принятие архитектурных решений». Анонсы в канале.

Напутствие

Если вы прочитали только эту страницу — запомните:

Архитектор — это не должность. Это образ мышления.

Идеальных решений не бывает. Бывают компромиссы.

Системное мышление важнее любой технологии.

Люди важнее кода в 10 раз.

Вы никогда не будете знать абсолютно всё. И это нормально.

Действуйте. Ошибайтесь. Исправляйтесь. Растите.

Я в вас верю.

Артур Кухтинов
Москва, 2026