# Архитектурная Устойчивость и Распределенная Надежность: Теоретический Фундамент Enterprise-систем

## 1. Введение: Смена Парадигмы — От Детерминизма к Вероятности

Переход квалифицированного инженера из среды разработки игровых движков (Unity/GameDev) в экосистему корпоративных бэкендов на платформе.NET (Enterprise Backend) — это не просто смена синтаксиса или библиотек. Это фундаментальный сдвиг в восприятии времени, пространства и состояния. В классическом GameDev-цикле разработчик оперирует в мире, который, по своей сути, является детерминированным: состояние игрового мира обновляется в строгой последовательности кадров (Update, FixedUpdate), память часто является общей и локальной, а конкурентность ограничена основным потоком и предсказуемыми корутинами.1 Если в Unity происходит сбой обработки кадра, это визуальный глюк. В банковской системе или сервисе бронирования сбой — это финансовые потери или нарушение целостности данных.

В контексте пятого дня интенсивного обучения ("Resilience, Load Testing, Idempotency") мы сталкиваемся с жестокой реальностью распределенных систем: **ненадежностью**. В отличие от монолитного игрового клиента, где все объекты живут в одном адресном пространстве, энтерпрайз-система состоит из множества узлов (API, База Данных, Кэш, Внешние Сервисы), связанных ненадежной сетью. Здесь действуют законы "Восьми заблуждений распределенных вычислений" (Fallacies of Distributed Computing). Сеть не надежна, задержка не равна нулю, а пропускная способность не бесконечна.

Данный отчет представляет собой исчерпывающее теоретическое руководство, призванное деконструировать механизмы выживания высоконагруженных систем. Мы подробно разберем три столпа надежности: **Идемпотентность** (гарантия корректности при повторах), **Нагрузочное Тестирование** (доказательство устойчивости через симуляцию хаоса) и **Резистентность** (способность системы деградировать грациозно, а не коллапсировать).

### 1.1 Философия "Failure is Inevitable"

Центральной темой современной инженерии надежности (Site Reliability Engineering — SRE) является аксиома: **Сбой неизбежен**.2 В любой достаточно сложной системе компоненты будут выходить из строя. Диски переполнятся, сетевые кабели будут повреждены, сторонние API вернут 503 ошибку, а сборщик мусора (GC) в.NET приостановит выполнение потоков в самый неподходящий момент. Попытка построить систему, которая "никогда не падает", обречена на провал и экспоненциальный рост стоимости. Вместо этого мы строим системы, которые **ожидают сбоя** и умеют с ним работать. Это переход от концепции "MTBF" (Mean Time Between Failures — Среднее время между сбоями) к "MTTR" (Mean Time To Recovery — Среднее время восстановления).4

Для системы бронирования переговорных комнат, которую мы рассматриваем как эталонный проект 1, это означает, что пользователь не должен получить двойное бронирование одной комнаты даже если сервер Redis перезагружается в момент транзакции, а база данных PostgreSQL испытывает пиковую нагрузку.

## 2. Идемпотентность в Распределенных API: Теория и Практика

Идемпотентность — это свойство операции, при котором её многократное выполнение приводит к тому же результату, что и однократное.5 В математике это выражается как $f(f(x)) = f(x)$. В контексте REST API это означает, что если клиент отправит один и тот же запрос на создание бронирования трижды (например, из-за таймаута сети), сервер создаст только одно бронирование и вернет одинаковый успешный ответ для всех трех попыток.

### 2.1 Проблема Двух Генералов и "Призрачные Запросы"

Необходимость идемпотентности продиктована природой сетевого взаимодействия. Когда клиент (например, SPA на React или мобильное приложение) отправляет HTTP POST запрос, возможны три исхода:

1. **Успех (2xx):** Сервер обработал запрос, клиент получил подтверждение.
2. **Явный отказ (4xx/5xx):** Сервер отклонил запрос, клиент знает, что операция не выполнена.
3. **Неопределенность (Timeout/Network Error):** Клиент отправил запрос, но соединение оборвалось до получения ответа.

В третьем случае клиент попадает в ловушку "Проблемы Двух Генералов". Он не знает, достиг ли запрос сервера. Возможно, сервер даже не получил байты запроса. А возможно, сервер успешно обработал транзакцию, списал деньги или забронировал слот, но пакет с подтверждением ("200 OK") потерялся на обратном пути.7

Если клиент, следуя наивной логике, просто повторит запрос (Retry), он рискует создать дубликат ресурса. В системе бронирования это приведет к двум записям на одно время. В платежной системе — к двойному списанию средств. Идемпотентность решает эту проблему, позволяя клиенту безопасно повторять запросы.

### 2.2 Стандарт Idempotency-Key (RFC Draft)

Хотя методы GET, PUT, DELETE являются идемпотентными по определению HTTP спецификации (удаление удаленного объекта возвращает 404 или 200, но состояние системы не меняется), методы POST и PATCH таковыми не являются. Для решения этой задачи IETF (Internet Engineering Task Force) разработал черновик стандарта для заголовка Idempotency-Key.5

#### 2.2.1 Механизм Работы

Протокол взаимодействия выглядит следующим образом:

1. **Генерация Ключа:** Клиент генерирует уникальный идентификатор (обычно UUID v4) для каждого *намерения* действия. Если пользователь нажимает кнопку "Забронировать", генерируется UUID.
2. **Передача:** Этот UUID передается в заголовке Idempotency-Key вместе с телом запроса.
3. **Обработка на Сервере:**
   * Сервер проверяет наличие этого ключа в своем хранилище идемпотентности (обычно Redis).
   * **Сценарий А (Ключ новый):** Сервер выполняет операцию. По завершении он сохраняет результат (Status Code + Response Body) в хранилище, привязывая его к ключу.
   * **Сценарий Б (Ключ существует и обработан):** Сервер видит, что операция уже была выполнена. Он *не запускает* бизнес-логику повторно. Вместо этого он извлекает сохраненный результат из хранилища и возвращает его клиенту.
   * **Сценарий В (Ключ существует, но в процессе):** Если повторный запрос пришел, пока первый еще обрабатывается (Condition Race), сервер должен вернуть специальный код, например 409 Conflict или 425 Too Early, сигнализируя клиенту подождать.7

### 2.3 Архитектурные Стратегии Реализации

Существует два основных подхода к реализации этого механизма: на уровне базы данных и на уровне кэша/middleware.

#### 2.3.1 Дедупликация на уровне Базы Данных

Этот метод полагается на ACID-транзакции реляционной БД. Мы создаем таблицу IdempotencyLog или добавляем уникальный индекс по (UserId, IdempotencyKey).

* **Преимущества:** Гарантированная консистентность. Если транзакция откатывается, запись о ключе тоже исчезает.
* **Недостатки:** Низкая производительность. Каждый запрос требует чтения/записи на диск. Сложнее кэшировать *тело ответа* (JSON), так как БД не оптимизирована для хранения больших BLOB-ов для каждого запроса. Также это смешивает инфраструктурную логику с доменной моделью.10

#### 2.3.2 Дедупликация на уровне Распределенного Кэша (Redis)

Это предпочтительный подход для высоконагруженных систем (.NET Enterprise). Redis, работающий в памяти, обеспечивает минимальную задержку при проверке ключей.

* **Механизм:** Используется паттерн Cache-Aside или Check-and-Set.
* **Проблема:** Redis не поддерживает транзакции в том же смысле, что и SQL (ACID), но поддерживает атомарность операций через Lua-скрипты.

### 2.4 Атомарность и Состояние Гонки (Race Conditions)

Самая критическая уязвимость в реализации идемпотентности — это состояние гонки между проверкой наличия ключа и его записью. Рассмотрим наивную реализацию на C#:

C#

// ОПАСНЫЙ КОД - НЕ ИСПОЛЬЗОВАТЬ  
if (!await \_cache.ExistsAsync(key)) // Точка входа для гонки  
{  
 await \_cache.SetAsync(key, "Processing"); // Блокировка  
 var result = await ProcessRequest();  
 await \_cache.SetAsync(key, result);  
}  
else  
{  
 return await \_cache.GetAsync(key);  
}

В сценарии "Thundering Herd" (раздел 4), когда 50 запросов приходят одновременно, все 50 могут пройти проверку !ExistsAsync до того, как первый успеет записать "Processing". Это приведет к 50-кратному выполнению операции.11

#### 2.4.1 Решение: Lua-скриптинг в Redis

Для обеспечения атомарности необходимо использовать Lua-скрипты, выполняемые на стороне сервера Redis. Redis гарантирует, что во время выполнения скрипта никакие другие команды не будут обработаны. Это превращает последовательность "Проверить -> Если нет, Записать" в одну неразрывную инструкцию.12

Пример логики Lua-скрипта для атомарной блокировки:

1. Принимает Ключ и Значение ("PENDING").
2. Проверяет GET key. Если значение есть — возвращает его.
3. Если значения нет — выполняет SET key value NX EX ttl (Записать, только если не существует, с таймаутом).
4. Возвращает nil (признак того, что блокировка успешно захвачена).

### 2.5 Технические Вызовы Middleware в ASP.NET Core

Реализация идемпотентности через Middleware требует перехвата потока ответа (HttpResponse.Body). В.NET этот поток по умолчанию является write-only и отправляется клиенту по мере записи (streaming). Middleware не может "прочитать" то, что контроллер записал в ответ, чтобы сохранить это в Redis.14

Для решения этой проблемы используется техника **Response Buffering**:

1. Middleware подменяет context.Response.Body на временный MemoryStream.
2. Передает управление дальше по конвейеру (next()).
3. Контроллер пишет ответ в MemoryStream.
4. После возврата управления Middleware "перематывает" поток в начало (Seek(0)), считывает его содержимое в строку (для сохранения в Redis) и копирует байты в оригинальный поток ответа, чтобы отправить их клиенту.16

**Важно:** Этот подход увеличивает потребление памяти, так как весь ответ хранится в RAM сервера до отправки. Для очень больших ответов (файлы, большие отчеты) идемпотентность на уровне тела ответа может быть неприменима.

### 2.6 Жизненный Цикл Ключа

Ключи идемпотентности не должны храниться вечно. Это приведет к утечке памяти в Redis. Обычно устанавливается TTL (Time-To-Live) от 24 до 48 часов. Это перекрывает разумное время, в течение которого клиент может пытаться повторить запрос.17

## 3. Резистентность и Инженерия Хаоса

Резистентность (Resilience) — это способность системы восстанавливаться после сбоев и продолжать функционировать, возможно, в деградированном режиме. В отличие от надежности (Reliability), которая стремится предотвратить сбои, резистентность фокусируется на минимизации последствий уже произошедшего сбоя.18

### 3.1 Паттерн "Circuit Breaker" (Автоматический Выключатель)

Этот паттерн предотвращает каскадные сбои. Если сервис А зависит от сервиса Б, и сервис Б начинает отвечать с задержкой в 30 секунд (или падает с 500 ошибкой), сервис А не должен продолжать долбиться в него, занимая свои потоки ожиданием.

Circuit Breaker работает как конечный автомат с тремя состояниями:

1. **Closed (Закрыт):** Нормальный режим. Запросы проходят. Счетчик ошибок отслеживается.
2. **Open (Открыт):** Если процент ошибок превышает порог (например, 50% за 1 минуту), цепь размыкается. Все новые запросы мгновенно отклоняются с ошибкой BrokenCircuitException, не доходя до внешнего сервиса. Это дает сервису Б время "остыть" и восстановиться.
3. **Half-Open (Полуоткрыт):** По истечении таймаута "отдыха", Circuit Breaker пропускает один пробный запрос. Если он успешен — цепь закрывается (восстановление). Если нет — снова открывается.20

В экосистеме.NET стандартом для реализации этого паттерна является библиотека **Polly**.

### 3.2 Стратегии Повторных Попыток (Retry) и Проблема Синхронизации

Повторные попытки (Retries) эффективны для кратковременных ("мигающих") сбоев сети. Однако бездумное использование Retry может убить систему.

Рассмотрим ситуацию Thundering Herd (Эффект Грочущего Стада).21 Если база данных перегружена и начинает отклонять запросы, а 1000 клиентов одновременно получают ошибку и решают повторить запрос через ровно 1 секунду, то через секунду база данных получит ударную волну в 1000 запросов, что гарантированно добьет её.

#### 3.2.1 Exponential Backoff with Jitter

Для решения этой проблемы используется алгоритм экспоненциальной отсрочки с джиттером (случайным разбросом).23

Формула задержки: $Delay = 2^{attempt} \times base\\_delay + Random(0, jitter\\_max)$.

Это "размазывает" нагрузку во времени, предотвращая синхронизированные атаки клиентов на восстанавливающийся сервис.

### 3.3 Грациозная Деградация (Graceful Degradation)

Когда компонент системы недоступен, приложение должно не падать с "Белым Экраном Смерти", а предоставлять урезанный функционал.24

* Если упал сервис рекомендаций — показываем просто список популярных товаров.
* Если упал Redis — идем в базу данных напрямую (с риском снижения производительности) или возвращаем ошибку только для операций записи, оставляя чтение доступным.

#### 3.3.1 Fail-Safe vs. Fail-Secure

Выбор стратегии деградации зависит от типа операции. Аналогия из физической безопасности 26:

* **Fail-Safe (Отказобезопасность):** При отключении электричества электронный замок открывается (чтобы люди могли эвакуироваться). В ПО это аналог возврата дефолтного значения или кэшированных данных при ошибке. Применимо для операций чтения (GetSchedule).
* **Fail-Secure (Отказоустойчивость):** При отключении электричества замок остается закрытым (чтобы защитить имущество). В ПО это аналог возврата ошибки. Применимо для операций, требующих строгой консистентности (CreateBooking). Мы не можем разрешить бронирование, если не работает механизм блокировок, иначе получим дабл-букинг.28

## 4. Распределенная Конкурентность и Блокировки

В монолитном приложении на одном сервере для синхронизации доступа к общим ресурсам используется конструкция lock (в C#) или Mutex. Они работают на уровне потоков операционной системы и общей памяти. В распределенной системе, где приложение запущено в 10 экземплярах (репликах) в Kubernetes, lock бесполезен: он блокирует поток только на *одном* сервере, в то время как другие 9 серверов продолжают свободно писать в базу.18

### 4.1 Распределенные Блокировки (Distributed Locks)

Для синхронизации процессов на разных машинах требуется внешний координатор. Redis идеально подходит для этой роли благодаря своей скорости.

Простейший алгоритм блокировки:

SET resource\_name unique\_token NX PX 30000

* **NX (Not Exists):** Записать, только если ключа нет. Это атомарный примитив захвата лока.
* **PX 30000:** Установить срок жизни 30 секунд. Это механизм защиты от "мёртвых блокировок" (Deadlocks). Если сервис упадет, не освободив лок, Redis сам удалит его через 30 секунд, позволив другим процессам продолжить работу.29

### 4.2 Алгоритм Redlock

Для систем, требующих высокой гарантии (где падение одного узла Redis недопустимо), используется алгоритм Redlock. Он подразумевает наличие N (обычно 5) независимых мастеров Redis. Клиент пытается захватить лок на всех N узлах. Лок считается захваченным, если он получен на большинстве (N/2 + 1) узлов за время, меньшее чем валидность лока.30

Однако Мартин Клеппман (автор "Designing Data-Intensive Applications") критикует Redlock за зависимость от системных часов. В рамках нашего учебного проекта (Week 14) мы используем упрощенную модель с одним инстансом Redis, но с пониманием ограничений.

### 4.3 Оптимистичная Конкурентность (Optimistic Concurrency)

Альтернативой пессимистичным блокировкам (Redlock) является оптимистичная конкурентность на уровне БД.

В EF Core это реализуется через токен конкурентности (обычно поле xmin в PostgreSQL или RowVersion в SQL Server). При обновлении записи генерируется SQL:

UPDATE Bookings SET User = 'A', Version = 2 WHERE Id = 10 AND Version = 1

Если другой пользователь уже обновил запись (и Version стала 2), этот запрос затронет 0 строк, и EF Core выбросит DbUpdateConcurrencyException. Это эффективнее блокировок при низком уровне конфликтов (Low Contention), но при высоком ("Thundering Herd") приводит к большому количеству ошибок.10

## 5. Нагрузочное Тестирование: Валидация Гипотез

Без нагрузочного тестирования все архитектурные решения являются лишь гипотезами. Инструмент **NBomber** позволяет нам перевести теоретические рассуждения в плоскость эмпирических данных.32

### 5.1 Модели Нагрузки: Open vs. Closed

Существует два фундаментальных способа моделирования нагрузки, и выбор неправильного может привести к ложным результатам.33

#### 5.1.1 Closed Model (Закрытая Система)

В этой модели фиксированное число пользователей (вокеров) циклично выполняют действия. Новый запрос отправляется только *после* завершения предыдущего.

* **Сценарий:** Если сервер начинает тормозить, вокеры тоже замедляются. Частота запросов (RPS) падает.
* **Применение:** Тестирование внутренних систем, где количество сотрудников ограничено. Симуляция поведения "очереди".
* **NBomber:** Simulation.KeepConstant(copies: 50) — поддерживает 50 активных виртуальных пользователей.

#### 5.1.2 Open Model (Открытая Система)

В этой модели новые пользователи приходят с фиксированной частотой, независимо от состояния системы.

* **Сценарий:** Если сервер тормозит, очередь запросов растет бесконечно, пока не закончится память или сокеты. Это симуляция реального DDoS или наплыва пользователей ("Хабраэффект", старт продаж билетов).
* **Применение:** Публичные API, e-commerce.
* **NBomber:** Simulation.Inject(rate: 100, interval: 1s) — впрыскивает 100 пользователей в секунду, даже если предыдущие 100 еще не получили ответ.33 Это наиболее жестокий и честный тест для системы бронирования.

### 5.2 Статистика Латенси: Почему P50 бесполезен

Среднее арифметическое (Mean) и медиана (P50) скрывают проблемы. Если 99 запросов выполняются за 10мс, а 1 запрос за 10 секунд (из-за GC или блокировки), среднее будет ~110мс, что выглядит приемлемо. Но этот 1% пользователей страдает.

В распределенных системах важен "Длинный Хвост" (Long Tail) распределения задержек. Мы ориентируемся на перцентили:

* **P95:** 95% пользователей получают ответ быстрее этого времени.
* **P99:** Самые медленные 1% запросов. Именно здесь проявляются проблемы конкурентности, блокировок и таймаутов.34
* **Max:** Абсолютный худший случай.

### 5.3 Сценарий "Race Condition Attack"

Для проверки идемпотентности и блокировок мы создаем специальный сценарий в NBomber.

Цель: Симулировать "Состояние Гонки", отправив 50 идентичных запросов на создание брони (один RoomId, одно время, один IdempotencyKey) максимально одновременно.

Код сценария на C# 37:

C#

var scenario = Scenario.Create("race\_attack", async context =>  
{  
 var request = Http.CreateRequest("POST", "https://api/bookings")  
 .WithHeader("Idempotency-Key", "fixed-uuid-123") // Один ключ для всех!  
 .WithBody(new { roomId = 1, time = "10:00" });  
   
 var response = await Http.Send(httpClient, request);  
 return response;  
})  
.WithLoadSimulations(  
 // Запускаем 50 копий одновременно  
 Simulation.KeepConstant(copies: 50, during: TimeSpan.FromSeconds(10))   
);

**Критерий Успеха:**

1. **БД:** Создана ровно 1 запись.
2. **Ответы:** NBomber показывает 50 успешных ответов (если идемпотентность работает идеально и возвращает кэшированный успех) ИЛИ 1 успех и 49 409 Conflict.
3. **Провал:** Если в БД 2+ записи или если возникают 500 Internal Server Error (ошибки дедлоков в БД).

### 5.4 Проблема "Coordinated Omission" (Скрытая Ошибка)

При нагрузочном тестировании важно учитывать, что если инструмент тестирования сам не справляется с генерацией нагрузки (например, CPU тестера загружен на 100%), он может неверно измерять время ответа, "пропуская" моменты, когда сервер наиболее загружен. NBomber архитектурно минимизирует это, но мониторинг ресурсов самой тестовой машины обязателен.32

## 6. Детальная Архитектура Реализации (День 5)

Синтезируя вышесказанное, мы формируем архитектурный план для 5-го дня обучения.

### 6.1 Пайплайн Обработки Запроса (Request Pipeline)

Следующая таблица описывает прохождение запроса через слои защиты:

| **Этап** | **Компонент** | **Действие** | **Технология** | **Механизм Отказа (Fallback)** |
| --- | --- | --- | --- | --- |
| 1 | **Load Balancer** | Rate Limiting (ограничение частоты). | Nginx / YARP | 429 Too Many Requests |
| 2 | **Middleware** | Проверка Idempotency-Key. | ASP.NET Core + Redis Lua | Пропуск (если Redis лежит - Fail Secure или Fail Safe зависит от конфига) |
| 3 | **Policy** | Circuit Breaker & Retry. | Polly | Возврат дефолтного ответа или Cached Data |
| 4 | **Service** | Распределенная Блокировка (Redlock). | Redis | 503 Service Unavailable (Fail Secure) |
| 5 | **Database** | Транзакция и Unique Constraint. | PostgreSQL | Rollback + Exception |

### 6.2 Интеграция с Observability (OpenTelemetry)

Для того чтобы видеть, как работают эти механизмы в реальном времени под нагрузкой NBomber, мы используем метрики OpenTelemetry (OTEL).

* **Счетчики (Counters):** idempotency.hits (сколько раз вернули кэшированный ответ), idempotency.misses.
* **Гистограммы (Histograms):** lock.acquisition\_time (как долго ждем захвата Redlock). Если P99 этого времени растет, значит, конкуренция за комнаты слишком высока, и нужно шардировать данные или оптимизировать UX.39
* **Трассировка (Tracing):** В Jaeger мы должны видеть спаны: Middleware -> Redis:EVAL -> Service -> Redis:SETNX -> DB:INSERT. Это позволяет визуально подтвердить, что при повторном идемпотентном запросе спан DB:INSERT отсутствует.

### 6.3 Чек-лист Реализации

Для успешного завершения дня 5, студент должен реализовать:

1. **Idempotency Middleware:** С поддержкой буферизации MemoryStream для кэширования ответов.
2. **Lua Scripts:** lock.lua и release.lua для атомарных операций в Redis.
3. **NBomber Test Suite:** Консольное приложение, запускающее сценарий атаки "Race Condition".
4. **Resilience Policies:** Настройка Polly в Program.cs для HTTP-клиентов и вызовов БД.

## 7. Заключение

Инженерная зрелость Enterprise-разработчика определяется не знанием синтаксиса C#, а пониманием того, как система ведет себя в пограничных состояниях. Идемпотентность, резистентность и нагрузочное тестирование — это не "бонусные фичи", а фундаментальные требования к любой распределенной системе, работающей с деньгами или ограниченными ресурсами. Переход от Unity к.NET Backend требует замены ментальной модели "игрового цикла" на модель "асинхронного потока событий в ненадежной среде". Представленные в этом отчете механизмы (Redis Lua, Polly, NBomber) являются стандартным инструментарием для обуздания хаоса распределенных систем.

### Таблица Сравнения: Монолит (Unity) vs Распределенная Система (.NET)

| **Характеристика** | **Unity / Monolith** | **Enterprise / Distributed** | **Решение** |
| --- | --- | --- | --- |
| **Память** | Общая (Shared Memory), быстрая. | Разделенная (No Shared Memory), медленная. | Redis (Distributed Cache). |
| **Синхронизация** | lock, Monitor (внутри процесса). | Redlock, SETNX (между процессами). | Распределенные блокировки. |
| **Сбои** | Фатальные (Crash to Desktop). | Частичные (Partial Failure). | Circuit Breaker, Graceful Degradation. |
| **Повторы** | Редко нужны (локально). | Необходимы (сеть ненадежна). | Idempotency Keys. |
| **Время** | Линейное (Time.time). | Относительное (NTP skew, Latency). | Vector Clocks, NTP, Idempotency. |

*Конец отчета.*

#### Источники

1. Неделя 14
2. The Anatomy of a Site Reliability Engineer: Mastering the Art of Keeping Systems Running, дата последнего обращения: декабря 10, 2025, <https://algocademy.com/blog/the-anatomy-of-a-site-reliability-engineer-mastering-the-art-of-keeping-systems-running/>
3. Netflix Chaos Monkey: Ensuring Reliability through Controlled Chaos | by Mayank Jain, дата последнего обращения: декабря 10, 2025, <https://medium.com/@mayank2000jain/netflix-chaos-monkey-ensuring-reliability-through-controlled-chaos-b630403864eb>
4. Resilience Engineering: Learning to Embrace Failure - ACM Queue, дата последнего обращения: декабря 10, 2025, <https://queue.acm.org/detail.cfm?id=2371297>
5. draft-ietf-httpapi-idempotency-key-header-07 - IETF Datatracker, дата последнего обращения: декабря 10, 2025, <https://datatracker.ietf.org/doc/html/draft-ietf-httpapi-idempotency-key-header-07>
6. Idempotency - What is an Idempotent REST API? - REST API Tutorial, дата последнего обращения: декабря 10, 2025, <https://restfulapi.net/idempotent-rest-apis/>
7. `Idempotency-Key` IETF standards draft — brandur.org, дата последнего обращения: декабря 10, 2025, <https://www.brandur.org/fragments/idempotency-key-draft>
8. The Idempotency-Key HTTP Header Field - IETF Datatracker, дата последнего обращения: декабря 10, 2025, <https://datatracker.ietf.org/doc/draft-ietf-httpapi-idempotency-key-header/>
9. Working with the new Idempotency Keys RFC - HTTP Toolkit, дата последнего обращения: декабря 10, 2025, <https://httptoolkit.com/blog/idempotency-keys/>
10. Idempotency in Distributed Systems: When and Why It Matters - DZone, дата последнего обращения: декабря 10, 2025, <https://dzone.com/articles/importance-of-idempotency-in-distributed-systems>
11. Writes done Right : Atomicity and Idempotency with Redis, Lua, and Go - DEV Community, дата последнего обращения: декабря 10, 2025, <https://dev.to/pixperk/writes-done-right-atomicity-and-idempotency-with-redis-lua-and-go-5ebd>
12. Scripting with Lua | Docs - Redis, дата последнего обращения: декабря 10, 2025, <https://redis.io/docs/latest/develop/programmability/eval-intro/>
13. Writes done Right : Atomicity and Idempotency with Redis, Lua, and Go - Medium, дата последнего обращения: декабря 10, 2025, <https://medium.com/@pixperk/writes-done-right-atomicity-and-idempotency-with-redis-lua-and-go-9d37204e5a3d>
14. Idempotency in REST APIs in ASP.NET Core | by Daniel Alabuja | Oct, 2025 - Medium, дата последнего обращения: декабря 10, 2025, <https://medium.com/@Alabuja/idempotency-in-rest-apis-in-asp-net-core-4a3f30cb8e94>
15. How to read ASP.NET Core Response.Body? - Stack Overflow, дата последнего обращения: декабря 10, 2025, <https://stackoverflow.com/questions/43403941/how-to-read-asp-net-core-response-body>
16. ResponseBuffering for Asp.Net core - GitHub Gist, дата последнего обращения: декабря 10, 2025, <https://gist.github.com/doggy8088/3219b0ff47d4d89d11b83304b6c804bc>
17. Implementing Idempotency Keys in REST APIs | Zuplo Learning Center, дата последнего обращения: декабря 10, 2025, <https://zuplo.com/learning-center/implementing-idempotency-keys-in-rest-apis-a-complete-guide>
18. Microservices Resilience Patterns - GeeksforGeeks, дата последнего обращения: декабря 10, 2025, <https://www.geeksforgeeks.org/system-design/microservices-resilience-patterns/>
19. Resilience Engineering and Indicators of Resiliencei - International Risk Governance Council, дата последнего обращения: декабря 10, 2025, <https://irgc.org/wp-content/uploads/2018/09/Herrera-Resilience-Engineering-and-Indicators-of-Resilience.pdf>
20. Implement Circuit Breaker using Polly in .Net Core 8 - DEV Community, дата последнего обращения: декабря 10, 2025, <https://dev.to/neers/implement-circuit-breaker-using-polly-in-net-core-8-5gm1>
21. Distributed Systems Horror Stories: The Thundering Herd Problem ..., дата последнего обращения: декабря 10, 2025, <https://encore.dev/blog/thundering-herd-problem>
22. The Thundering Herd Problem. Introduction | by Rohit Kumar - Medium, дата последнего обращения: декабря 10, 2025, <https://stenzr.medium.com/the-thundering-herd-problem-2d19e9492cbc>
23. Thundering herd problem - Wikipedia, дата последнего обращения: декабря 10, 2025, <https://en.wikipedia.org/wiki/Thundering_herd_problem>
24. REL05-BP01 Implement graceful degradation to transform applicable hard dependencies into soft dependencies - Reliability Pillar - AWS Documentation, дата последнего обращения: декабря 10, 2025, <https://docs.aws.amazon.com/wellarchitected/latest/reliability-pillar/rel_mitigate_interaction_failure_graceful_degradation.html>
25. Four Considerations When Designing Systems For Graceful Degradation - New Relic, дата последнего обращения: декабря 10, 2025, <https://newrelic.com/blog/best-practices/design-software-for-graceful-degradation>
26. Fail Safe vs. Fail Secure in Access Control - Verkada, дата последнего обращения: декабря 10, 2025, <https://info.verkada.com/compare/fail-safe-vs-fail-secure/>
27. Fail Safe vs Fail Secure: Which Lock is Right for Your Needs - ProdataKey, дата последнего обращения: декабря 10, 2025, <https://www.prodatakey.com/single-post/fail-safe-vs-fail-secure>
28. Fail Safe vs. Fail Secure Electronic Locksets - Office of Research Facilities, дата последнего обращения: декабря 10, 2025, <https://orf.od.nih.gov/TechnicalResources/Documents/Technical%20Bulletins/20TB/Fail%20Safe%20vs.%20Fail%20Secure%20Electronic%20Locksets%20June%202020%20-%20Technical%20Bulletin%20UPDATED_508.pdf>
29. Atomic Redis transaction for idempotency required for database inserts using Redis Springboot 3 - Stack Overflow, дата последнего обращения: декабря 10, 2025, <https://stackoverflow.com/questions/78820801/atomic-redis-transaction-for-idempotency-required-for-database-inserts-using-red>
30. Idempotency in a Distributed System | by Sameer Ahmed - Medium, дата последнего обращения: декабря 10, 2025, <https://sameerahmed56.medium.com/idempotency-in-a-distributed-system-df67fbd93b49>
31. Resiliency and high availability in microservices - .NET - Microsoft Learn, дата последнего обращения: декабря 10, 2025, <https://learn.microsoft.com/en-us/dotnet/architecture/microservices/architect-microservice-container-applications/resilient-high-availability-microservices>
32. Overview - NBomber, дата последнего обращения: декабря 10, 2025, <https://nbomber.com/docs/getting-started/overview/>
33. Load Simulation | NBomber, дата последнего обращения: декабря 10, 2025, <https://nbomber.com/docs/nbomber/load-simulation/>
34. Understanding the P95/P99 Latency Principle: Why the Slowest Requests Matter Most, дата последнего обращения: декабря 10, 2025, <https://medium.com/@rajesh.sgr/understanding-the-p95-p99-latency-principle-why-the-slowest-requests-matter-most-1bcabf3bf5e5>
35. P50 vs P95 vs P99 Latency: What These Percentiles Actually Mean (And How to Use Them), дата последнего обращения: декабря 10, 2025, <https://oneuptime.com/blog/post/2025-09-15-p50-vs-p95-vs-p99-latency-percentiles/view>
36. Why is tail latency (p95/p99) often more important than average latency for evaluating the performance of a vector search in user-facing applications? - Zilliz Vector Database, дата последнего обращения: декабря 10, 2025, <https://zilliz.com/ai-faq/why-is-tail-latency-p95p99-often-more-important-than-average-latency-for-evaluating-the-performance-of-a-vector-search-in-userfacing-applications>
37. Scenario | NBomber, дата последнего обращения: декабря 10, 2025, <https://nbomber.com/docs/nbomber/scenario/>
38. Hello World Tutorial - NBomber, дата последнего обращения: декабря 10, 2025, <https://nbomber.com/docs/getting-started/hello_world_tutorial/>
39. Idempotency-Key header - HTTP - MDN Web Docs, дата последнего обращения: декабря 10, 2025, <https://developer.mozilla.org/en-US/docs/Web/HTTP/Reference/Headers/Idempotency-Key>
40. Mastering Latency Metrics: P90, P95, P99 | System Design - YouTube, дата последнего обращения: декабря 10, 2025, <https://www.youtube.com/watch?v=lJ4NEMNBeS4>