Федеральное государственное бюджетное образовательное учреждение высшего образования
«Национальный исследовательский университет «МЭИ»

Институт: *ИВТИ* Кафедра: *ВТ*

Направление подготовки: *09.03.01 Информатика и вычислительная техника*

ОТЧЕТ по практике

Наименование практики: Производственная практика: научно-исследовательская работа

СТУДЕНТ

/ Кукушкин А.М. /
(подпись) (*Фамилия и инициалы*)

Группа А-06-22
(номер учебной группы)

ПРОМЕЖУТОЧНАЯ АТТЕСТАЦИЯ ПО ПРАКТИКЕ

(отлично, хорошо, удовлетворительно, неудовлетворительно, зачтено, не зачтено)

/ Ключанский А.А. /

Москва

2025

СОДЕРЖАНИЕ

1. Введение

Актуальность темы. Современные платформы электронной коммерции сталкиваются с экспоненциальным ростом нагрузки, требуя архитектурных решений, способных обеспечить эластичное масштабирование и высокий уровень доступности. Монолитные приложения исчерпывают свой потенциал, ограничивая скорость вывода новых функций на рынок и усложняя технологическое обновление. Переход к микросервисной архитектуре позволяет решить эти проблемы путем распределения нагрузки, изоляции сбоев и использования полиглотного стека технологий.

Цель исследования. Целью данной работы является проектирование и разработка отказоустойчивого, масштабируемого и сопровождаемого бэкенда интернет-магазина на основе микросервисной архитектуры с использованием стека технологий .NET, контейнеризацией и оркестрацией.

Задачи исследования. Для достижения поставленной цели были сформулированы следующие задачи:

1. Провести декомпозицию предметной области (электронная коммерция) и спроектировать микросервисную архитектуру, основанную на DDD.
2. Реализовать базовую инфраструктуру, включающую механизмы наблюдаемости (Observability) и устойчивости к сбоям (Resilience).
3. Разработать ключевые бизнес-микросервисы (Identity, Catalog, Ordering, Cart) с применением специализированных хранилищ данных.
4. Обеспечить согласованность данных между сервисами с помощью шаблона Саги.
5. Подготовить систему к оркестрации (Kubernetes) и разработать пайплайн CI/CD.

Научная новизна и практическая значимость. Практическая значимость заключается в создании эталонной архитектуры (reference architecture) на **ASP.NET Core**, демонстрирующей комплексное применение передовых паттернов: gRPC для внутренних коммуникаций, Polly для отказоустойчивости, OpenTelemetry для трассировки и Polyglot Persistence для управления данными.

2. Основная часть

2.1. Теоретические основы и анализ архитектурных стилей

2.1.1. Сравнение монолитной и микросервисной архитектур

Анализ эволюции архитектурных стилей показывает неизбежность перехода от монолита к распределенным системам по мере роста требований к бизнесу (таблица 1).

Таблица 1. Сравнение типов архитектур

Характеристика	**Монолитная архитектура**	**Микросервисная архитектура**
Масштабирование	Неэффективно. Масштабирование всего приложения, включая ненагруженные модули.	**Эластичное, селективное.** Масштабируются только критически нагруженные сервисы.
Отказоустойчивость	Низкая. Единый процесс, каскадные отказы.	**Высокая.** Изоляция процессов и сбоев.
Развертывание	Медленный цикл. Развертывание всего приложения при любом изменении.	**Быстрый, независимый CI/CD.** Сервисы развертываются независимо.
Технологический стек	Единый, ограниченный.	**Полиглот.** Сервис может использовать наиболее подходящий язык/СУБД.
Сложность	Низкая на старте, растет нелинейно.	**Высокая на старте.** Требует сложной инфраструктуры (Kubernetes, Observability).

В целом, развитие архитектурных стилей можно охарактеризовать как переход от традиционного монолитного к более удобному, подходящему к расширению микросервисному решению**,** где главным преимуществом становится повышенная отказоустойчивость, эластичное масштабирование и удобство развертывания.

2.1.2. Обоснование выбора микросервисной архитектуры

Выбор микросервисной архитектуры для платформы электронной коммерции обоснован необходимостью соответствовать современным требованиям к **устойчивости, масштабируемости и скорости разработки (Time to Market)**, которые не может обеспечить традиционный монолит.

A. Отказоустойчивость (Fault Tolerance) и Изоляция сбоев: В монолитной системе сбой в одном модуле (например, ошибка в коде рекомендаций) может привести к полному падению всего приложения. В микросервисной архитектуре, благодаря изоляции процессов, сбой в Recommendation Service или даже Payment Service не повлияет на возможность пользователя просматривать каталог и добавлять товары в корзину (Catalog Service, Cart Service). Изоляция критически важна для поддержания высокого SLA (Service Level Agreement) основных бизнес-функций.

B. Эластичное и Независимое Масштабирование (Elastic and Independent Scaling): Нагрузка в интернет-магазине является асимметричной:

- Catalog Service и Cart Service испытывают высокую нагрузку чтения/записи 24/7.
- Ordering Service и Payment Service испытывают пиковые нагрузки во время распродаж (Black Friday). Монолит требует масштабирования всего приложения, включая ненагруженные административные модули. Микросервисы позволяют независимо масштабировать только те сервисы, которые нуждаются в ресурсах.
- **Пример:** Cart Service может быть масштабирован до 50 инстансов, работающих с Redis, тогда как Identity Service может работать на 3 инстансах с PostgreSQL, что dramatically снижает расходы на облачную инфраструктуру.

C. Технологическая Свобода (Polyglot Stacks) и Скорость Разработки: Разделение на сервисы позволяет командам:

1. **Выбирать оптимальный стек:** Для Cart Service выбран Redis (NoSQL), а для Ordering Service — PostgreSQL (SQL).
2. **Использовать CI/CD:** Каждая команда может независимо разрабатывать, тестировать и развертывать свой сервис, не дожидаясь релиза других команд. Это сокращает цикл **Time to Market** для новых функций.

2.1.3. Критический анализ технологического стека .NET

Стек **ASP.NET Core** выбран как основа для реализации микросервисов по следующим причинам:

A. Высокая Производительность и Эффективность: ASP.NET Core демонстрирует высокую пропускную способность (throughput) в тестах Techempower, что позволяет обрабатывать большое количество одновременных запросов при минимальных затратах ресурсов (CPU, память). Это критично для контейнеризированных сред (Docker, Kubernetes).

B. Кроссплатформенность и Контейнеризация: платформа .NET является кроссплатформенной и полностью поддерживает Docker. Образы ASP.NET Core являются оптимизированными и легковесными, что ускоряет запуск и развертывание в Kubernetes.

C. Интеграция gRPC и Protocol Buffers: .NET имеет встроенную, высококачественную поддержку **gRPC**. Использование **Protocol Buffers** для контрактов обеспечивает:

1. **Бинарная сериализация:** значительно быстрее и компактнее, чем JSON.
2. **Строгая типизация:** контракты компилируются в C# классы, исключая ошибки, связанные с неверным маппингом данных, что повышает надежность межсервисного взаимодействия.

D. Экосистема Устойчивости (Polly) и Наблюдаемости (OpenTelemetry):

- **Polly:** является стандартом де-факто для реализации устойчивости в .NET. Он легко интегрируется с IHttpClientFactory (который используется и gRPC), позволяя декларативно применять политики.
- **OpenTelemetry:** .NET сообщество активно поддерживает OpenTelemetry, обеспечивая глубокую инструментацию для сбора трасс (Tracing) и метрик (Metrics) из коробки.

2.1.4. Ключевые паттерны проектирования распределенных систем

Для противодействия сложностям, присущим распределенным системам, применяются следующие ключевые шаблоны:

A. API Gateway (Шаблон фасада): API Gateway (**Ocelot/YARP**) выступает единой точкой входа и выполняет:

1. **Маршрутизация:** перенаправление запросов на соответствующие внутренние сервисы.
2. **Аутентификация и авторизация:** проверка JWT-токена, выданного Identity Service, и передача Claims в виде HTTP-заголовков (user_id).
3. **Агрегация (опционально):** объединение ответов от нескольких сервисов в один для снижения задержки на клиенте.

B. Database per Service (Изоляция данных): Каждый сервис имеет свою изолированную БД. Основные преимущества:

- **Автономия:** сервис может быть развернут и обновлен без влияния на схему данных других сервисов.
- **Технологический выбор (Polyglot Persistence):** возможность использовать PostgreSQL, Redis, Elasticsearch и т.д., оптимизируя хранилище под задачи домена.

C. Шаблоны устойчивости (Resilience Patterns): обязательное условие для распределенной системы. Реализуются через библиотеку **Polly** (см. п. 1.3.2):

- **Retry:** повторные вызовы при временных ошибках (сетевые сбои, таймауты).
- **Circuit Breaker:** предотвращение каскадного отказа путем "разрыва цепи" к неисправному сервису, переводя его в состояние **Fail-Fast** (быстрый провал).

D. Saga Pattern (Распределенная согласованность): Используется для поддержания согласованности бизнес-процесса между сервисами, не имеющими общего хранилища, за счет компенсирующих транзакций.

2.2. Выполнение Задачи 1: Декомпозиция предметной области и проектирование архитектуры (DDD)

2.2.1. Методология декомпозиции и определение Bounded Contexts

Декомпозиция сложной предметной области электронной коммерции проведена с использованием принципов **Domain-Driven Design (DDD)**. Этот подход фокусируется на создании моделей, соответствующих реальной бизнес-логике, и установлении четких границ между ними.

I. Определение Bounded Contexts (Ограниченных Контекстов):

Каждый Bounded Context представляет собой отдельную область бизнес-знаний, где определена своя уникальная доменная модель и терминология (таблица 2). Цель — обеспечить, чтобы одна и та же сущность (например, «Продукт») имела разное значение и набор атрибутов в разных контекстах, что устраняет семантическую неоднозначность и предотвращает «шаблонный» код.

Таблица 2. Определение Bounded Contexts

Bounded Context	Соответствующий Микросервис	Ключевая доменная модель и ответственность
Identity & Access	**Identity Service**	**User, Role, Credentials.** Отвечает только за аутентификацию и авторизацию (выдачу JWT).
Catalog	**Catalog Service**	**Product, Category, StockItem.** Ответственность: отображение товаров, управление мастер-данными и запасами.
Ordering	**Ordering Service**	**Order (Агрегат Root), OrderItem, Address.** Ответственность: обработка, управление статусами заказа, координация распределенной транзакции (Саги).
Shopping Cart	**Cart Service**	**ShoppingCart (временный агрегат), CartItem.** Ответственность: высокоскоростное хранение сессионных данных корзины.
Payments	**Payment Service**	**Transaction, PaymentResult.** Ответственность: имитация или интеграция с внешними платежными системами.

II. Паттерн Агрегат (Aggregate): Обеспечение целостности данных

Внутри каждого контекста выделены **Агрегаты** — группы сущностей и объектов-значений, которые должны быть загружены и сохранены как единое целое (в рамках одной транзакции) для обеспечения инвариантов.

- **Order Aggregate (Ordering Service):** Корень агрегата Order инкапсулирует коллекцию OrderItem (позиции заказа). Это гарантирует, что статус заказа может быть изменен только через корневой метод Order.SetStatusToSubmitted(), а не напрямую через изменение статуса в БД или внешнее воздействие (изображение 1).

```
// Order.cs - Order Aggregate Root (Фрагмент для демонстрации инвариантов)
public class Order : Entity, IAggregateRoot
{
    public OrderStatus Status { get; private set; }
    private readonly List<OrderItem> _orderItems;

    // Добавление позиций только через корень агрегата, с проверкой бизнес-правил
    public void AddItem(Guid productId, string productName, decimal unitPrice, int units)
    {
        if (Status != OrderStatus.Draft)
            throw new InvalidOperationException("Заказ не может быть изменен после подтверждения.

        // ... (логика добавления)
    }

    public void SetStatusToSubmitted()
    {
        if (Status == OrderStatus.Draft)
        {
            Status = OrderStatus.Submitted;
            // Публикация доменного события для запуска Саги
            this.AddDomainEvent(new OrderSubmittedIntegrationEvent(this));
        }
    }
}
```

Изображение 1. Определение класса Order

III. Принцип Database per Service:

Каждый микросервис использует свой собственный, изолированный источник данных.

- **Преимущества:** Полная автономия разработки и развертывания. Возможность **Polyglot Persistence**.
- **Следствие:** Отсутствие общих транзакций (JOIN) между базами данных. Это потребовало внедрения шаблона **Сага** для обеспечения согласованности.

2.2.2. Проектирование гибридной архитектуры взаимодействия

Архитектура использует сочетание синхронных и асинхронных механизмов для достижения оптимального баланса между производительностью, надежностью и слабой связанностью.

I. Синхронное взаимодействие (gRPC):

- **Назначение:** Применяется для высокочастотных, критичных по времени ответа вызовов, которые требуют немедленной обработки, например, проверка запасов, аутентификация через API Gateway.
- **Преимущества gRPC:**
    1. **Производительность:** Использование HTTP/2 и бинарной сериализации Protocol Buffers, что дает значительный выигрыш по сравнению с REST/JSON.
    2. **Контрактность:** Строгая типизация контрактов, исключающая ошибки, связанные с десериализацией.

gRPC Контракт (Protos/catalog.proto) (изображение 2)

```
// ...
service StockChecker {
  // Метод, который будет вызван Ordering Service
  rpc CheckAndReserveStock (StockReservationRequest) returns (StockReservationResponse);
}

message StockReservationRequest {
  string order_id = 1;
  repeated ReserveItem items = 2;
}
// ...
```

Изображение 2. Определение gRPC Контракта

Реализация Клиента (Ordering Service): Клиент gRPC интегрируется с ASP.NET Core HttpClientFactory, что позволяет декларативно подключать политики устойчивости **Polly** (см. п. 1.3.2).

II. Асинхронное взаимодействие (RabbitMQ):

- **Назначение:** Реализация шаблона **Сага** и оповещения, где не требуется немедленный ответ.
- **Технология: RabbitMQ** (брокер сообщений) с библиотекой **MassTransit** для управления очередями и обработчиками событий.
- **Преимущества:**
    1. **Слабая связанность (Decoupling):** Сервис-издатель не знает, кто является подписчиком.
    2. **Надежность:** Если сервис-подписчик недоступен, брокер сохранит сообщение до его восстановления (механизм гарантии доставки).

3. **Масштабирование:** Обработка событий может быть горизонтально масштабирована за счет увеличения количества потребителей из очереди.

2.2.3. Стратегия управления данными (Polyglot Persistence и Saga)

I. Polyglot Persistence (Полиглотное хранение): Детализация выбора СУБД

Стратегия предполагает выбор лучшей СУБД для конкретной задачи домена, а не использование единой БД для всех сервисов (таблица 3).

Таблица 3. Выбор СУБД для сервисов

Микросервис	Технология БД	Основной критерий выбора
Ordering Service	**PostgreSQL**	**ACID-транзакции:** Обеспечение целостности при создании заказа и его фиксации.
Cart Service	**Redis**	**Скорость (in-memory) и низкая задержка (Latency):** Исключительно для сессионных, высокочастотных операций Key-Value.
Catalog Service	**PostgreSQL + Redis Caching**	PostgreSQL для мастер-данных, Redis для высокоскоростного кэширования часто запрашиваемых товаров.

II. Saga Pattern (Хореография): Детализация механизмов координации

Сага — это последовательность локальных транзакций, где каждая транзакция обновляет данные в рамках одного сервиса и публикует интеграционное событие, которое запускает следующую локальную транзакцию в другом сервисе.

Хореография (Choreography): Выбран этот подход, где нет центрального координатора. Сервисы сами реагируют на события, что повышает децентрализацию и устойчивость к сбоям.

Компенсирующие транзакции: Главный механизм обеспечения целостности.

- **Сценарий:** Ordering Service начал Сагу → Catalog Service зарезервировал товар → Payment Service **не смог** списать средства.
- **Действие:** Payment Service публикует событие **PaymentFailed** (изображение 3).
- **Компенсация:** Catalog Service подписывается на PaymentFailed и запускает **компенсирующую транзакцию**, которая отменяет резервирование запасов.

```
// Интеграционное событие, запущенное при неудаче платежа
public class PaymentFailedIntegrationEvent
{
    public Guid OrderId { get; set; }
    public string Reason { get; set; }
    // ...
}

// Обработчик компенсации в Catalog Service
public class PaymentFailedEventHandler : IIntegrationEventHandler<PaymentFailedIntegrationEvent>
{
    public async Task Handle(PaymentFailedIntegrationEvent @event)
    {
        // Логика компенсации: Отменить резервирование в БД
        // await _repository.CancelStockReservation(@event.OrderId);
        Log.Warning("Compensation started: Stock reservation cancelled for Order {OrderId}.", @event.OrderId);
    }
}
```

Изображение 3. Реализация модели событий

2.3. Выполнение Задачи 2: Реализация базовой инфраструктуры и шаблонов взаимодействия (C#)

2.3.1. Реализация Observability: Логирование (Serilog) и Трассировка (OpenTelemetry)

I. Serilog: Связывание логов с трассировкой

Ключевая задача в распределенной системе — видеть все лог-сообщения, относящиеся к одному запросу, даже если они прошли через 3-4 сервиса. Это достигается за счет автоматического обогащения контекста (изображение 4).

```
// ...
builder.Host.UseSerilog((context, services, configuration) => configuration
    // ...
    .Enrich.FromLogContext()         // <- Извлекает TraceId и SpanId из текущего OpenTelemetry контекста
    .Enrich.WithMachineName()
    .Enrich.WithProperty("ApplicationName", builder.Environment.ApplicationName)
    .WriteTo.Seq(context.Configuration["SeqServerUrl"] ?? "http://seq")
    .MinimumLevel.Override("Microsoft", LogEventLevel.Warning)
);
```

Изображение 4. Детализация контекста

II. OpenTelemetry: Автоматическая инструментация

OpenTelemetry интегрируется с .NET фреймворком, автоматически собирая данные:

- **AddAspNetCoreInstrumentation():** Генерирует корневой **Span** для каждого входящего HTTP-запроса, назначая ему **TraceId**.
- **AddHttpClientInstrumentation():** Захватывает исходящий HTTP-вызов, создает дочерний **Span**, и, что наиболее важно, автоматически вставляет заголовок TraceParent в исходящий запрос, обеспечивая сквозную трассировку в следующем сервисе (изображение 5).

```
// ...
builder.Services.AddOpenTelemetry()
    .WithTracing(tracing =>
    {
        tracing
            .AddAspNetCoreInstrumentation(o => o.RecordException = true)
            .AddHttpClientInstrumentation()       // Инструментация для gRPC/REST клиентов
            .AddEntityFrameworkCoreInstrumentation() // Трассировка запросов к БД
            .AddZipkinExporter(options =>
            {
                options.Endpoint = new Uri(builder.Configuration["Zipkin:Endpoint"]);
            });
    })
// ...
```

Изображение 5. Детализация инструментации

2.3.2. Реализация шаблонов устойчивости

Применение Polly к gRPC-клиенту в Ordering Service. Политики применяются к HttpClient через IHttpClientFactory (изображение 6).

```
static IAsyncPolicy<HttpResponseMessage> GetResiliencePolicy()
{
    // 1. Политика Retry
    var retryPolicy = HttpPolicyExtensions
        .HandleTransientHttpError()
        .WaitAndRetryAsync(5,
            retryAttempt => TimeSpan.FromSeconds(Math.Pow(2, retryAttempt)), // Экспоненциальная задержка
            (result, timeSpan, retryCount, context) =>
            {
                Log.Warning("Retry {RetryCount} failed. Waiting {TimeSpan} for CatalogService.", retryCount, timeSpan);
            }
        );

    // 2. Политика Circuit Breaker
    var circuitBreakerPolicy = HttpPolicyExtensions
        .HandleTransientHttpError()
        .CircuitBreakerAsync(
            3, // Порог срабатывания: 3 последовательные ошибки
            TimeSpan.FromSeconds(30), // Время, на которое "открывается" цепь
            onBreak: (ex, breakDelay) =>
            {
                Log.Error(ex, "Circuit Breaker opened! All subsequent calls to CatalogService will fail-fast for {Delay}", breakDelay);
            },
            onReset: () => Log.Information("Circuit Breaker closed. CatalogService is recovering.")
        );

    // 3. Комбинирование: Circuit Breaker защищает Retry.
    return Policy.WrapAsync(circuitBreakerPolicy, retryPolicy);
}
```

Изображение 6. Детализация логики объединенной политики

2.3.3. Реализация Identity Service (JWT-аутентификация)

Механизм Claims (изображение 7): После успешной аутентификации Identity Service генерирует токен, содержащий минимальный набор Claims. Этот токен используется API Gateway для аутентификации, а ключевой Claim "user_id" извлекается и передается всем downstream-сервисам в HTTP-заголовке.

```
// ...
[HttpPost("login")]
public async Task<IActionResult> Login([FromBody] LoginModel model)
{
    // ... (Проверка пользователя)

    var authClaims = new List<Claim>
    {
        new Claim(ClaimTypes.Name, user.UserName),
        new Claim("user_id", user.Id.ToString()),
        new Claim(JwtRegisteredClaimNames.Jti, Guid.NewGuid().ToString())
    };
    // ... (Создание и подпись токена)

    return Ok(new
    {
        token = new JwtSecurityTokenHandler().WriteToken(token),
        // ...
    });
}
```

Изображение 7. AccountController.cs

2.3.4. Реализация Catalog Service (gRPC-сервер)

Интерфейс gRPC и его роль в Саге: Метод CheckAndReserveStock (изображение 8) является локальной транзакцией в рамках Саги.

```
public class StockCheckerService : StockChecker.StockCheckerBase
{
    // ...
    public override async Task<StockReservationResponse> CheckAndReserveStock(
        StockReservationRequest request, ServerCallContext context)
    {
        Log.Information("Received stock reservation request for Order {OrderId}", request.OrderId);

        // ... (логика проверки)

        if (success)
        {
            // 1. Атомарное выполнение: уменьшение StockQuantity и запись резерва в БД.
            // await _repository.DecreaseStock(itemsToReserve);

            // 2. Уведомление следующего шага Саги
            // await _eventBus.Publish(new StockReservedIntegrationEvent(request.OrderId));
        }
        else
        {
            // Если нехватка, НЕ публикуем ничего, Сага останавливается.
            Log.Warning("Stock reservation failed for order {OrderId}. Shortage in: {Shortages}",
                        request.OrderId, string.Join(", ", shortageItems));
        }

        return new StockReservationResponse
        {
            IsSuccessful = success,
            Message = success ? "Stock reserved" : $"Shortage: {string.Join(", ", shortageItems)}"
        };
    }
}
```

Изображение 8. StockCheckerService.cs

2.3.5. Реализация Cart Service (Redis-репозиторий)

Ключ хранения: В Redis данные хранятся в формате cart:{userId}, где значением является JSON-сериализованный объект ShoppingCart. Реализуем репозиторий для работы с Cart (изображение 9).

```
using StackExchange.Redis;
using System.Text.Json;

public class RedisCartRepository : ICartRepository
{
    private readonly IDatabase _database;
    private readonly TimeSpan _cartExpiration = TimeSpan.FromDays(7);

    public RedisCartRepository(IConnectionMultiplexer redis)
    {
        _database = redis.GetDatabase();
    }
    private string GetKey(string userId) => $"cart:{userId}";

    public async Task<ShoppingCart> UpdateCartAsync(ShoppingCart cart)
    {
        var serializedCart = JsonSerializer.Serialize(cart);

        // StringSetAsync выполняет запись данных с установкой TTL
        bool updated = await _database.StringSetAsync(
            GetKey(cart.UserId),
            serializedCart,
            _cartExpiration // TTL = 7 дней
        );

        if (!updated)
        {
            throw new RedisException($"Failed to save cart for user {cart.UserId}.");
        }

        return cart;
    }
}
```

Изображение 9. RedisCartRepository.cs

2.4. Выполнение Задачи 3: Разработка ключевых бизнес-микросервисов и координация (Saga)

2.4.1. Реализация Ordering Service

Ordering Service является координатором сложного процесса покупки. Его основная ответственность — управление агрегатом Order и запуск распределенной транзакции (Саги).

I. Логика создания заказа (Агрегат Order):

Сервис получает команду на создание заказа, выполняет локальную транзакцию и публикует интеграционное событие, инициирующее Сагу (изображение 10).

```
// Ordering Service: OrderService.cs
public async Task<Guid> CreateOrderAsync(string userId, IEnumerable<OrderItemDto> items)
{
    // 1. Локальная транзакция: Создание объекта Order в состоянии Draft
    var order = new Order(userId, OrderStatus.Draft);

    foreach (var item in items)
    {
        order.AddItem(item.ProductId, item.ProductName, item.UnitPrice, item.Units);
    }

    await _orderRepository.AddAsync(order);
    await _unitOfWork.SaveChangesAsync();

    // 2. Публикация интеграционного события OrderStarted
    // Событие содержит всю информацию, необходимую для следующего шага Саги (Catalog Service)
    await _eventBus.Publish(new OrderStartedIntegrationEvent(
        order.Id, order.UserId, order.OrderItems.Select(i => new { i.ProductId, i.Units }))
    );

    // 3. Возврат ID заказа клиенту
    return order.Id;
}
```

Изображение 10. Логика создания заказа (Агрегат Order)

II. Синхронный вызов gRPC (Резервирование запасов):

В Ordering Service реализован клиент для Catalog Service (см. п. 1.3.2), который используется для синхронной проверки запасов или для отката резервирования. **Важно:** Запрос gRPC в реальной Саге может быть заменен асинхронным взаимодействием, но для быстрой проверки наличия синхронный вызов, защищенный **Polly**, предпочтителен (изображение 11).

```
// Ordering Service: CatalogGrpcClient.cs
public async Task<StockReservationResponse> ReserveStockAsync(Guid orderId, List<ReserveItem> items)
{
    // Благодаря AddPolicyHandler (Polly) этот вызов автоматически защищен Retry/Circuit Breaker
    var request = new StockReservationRequest { OrderId = orderId.ToString() };
    request.Items.AddRange(items);

    try
    {
        var response = await _stockCheckerClient.CheckAndReserveStockAsync(request);
        return response;
    }
    catch (BrokenCircuitException ex)
    {
        // Обработка случая, когда Circuit Breaker разомкнулся (Catalog Service недоступен)
        Log.Error(ex, "Catalog Service circuit is open. Cannot reserve stock.");
        return new StockReservationResponse { IsSuccessful = false, Message = "Catalog Service unavailable." };
    }
}
```

Изображение 11. Синхронный вызов gRPC

2.4.2. Реализация координации Саги

Для реализации хореографии Саги используется библиотека **MassTransit**, которая упрощает настройку роутинга и обработчиков событий (Consumer).

I. Настройка MassTransit (C#):

Конфигурация MassTransit в Program.cs для Ordering Service (изображение 12).

```
// Ordering Service: Program.cs
builder.Services.AddMassTransit(x =>
{
    // Добавление Consumer для обработки ответов Саги
    x.AddConsumer<StockReservedIntegrationEventHandler>();
    x.AddConsumer<PaymentSucceededIntegrationEventHandler>();
    x.AddConsumer<PaymentFailedIntegrationEventHandler>(); // Для компенсации

    x.UsingRabbitMq((context, cfg) =>
    {
        // Подключение к брокеру
        cfg.Host(builder.Configuration["RabbitMQ:Host"], "/", h =>
        {
            h.Username(builder.Configuration["RabbitMQ:User"]);
            h.Password(builder.Configuration["RabbitMQ:Pass"]);
        });

        // Настройка очередей для Consumer
        cfg.ReceiveEndpoint("order-stock-reserved", e =>
        {
            e.ConfigureConsumer<StockReservedIntegrationEventHandler>(context);
        });

        // ... (другие ReceiveEndpoints)
    });
});
```

Изображение 12. Использование MassTransit

II. Обработчик события StockReservedIntegrationEvent:

Этот обработчик в Ordering Service принимает результат резервирования и решает, что делать дальше: продолжить Сагу (перейти к оплате) или отменить заказ (если товара нет).(изображение 13)

```
// Ordering Service: StockReservedIntegrationEventHandler.cs
public class StockReservedIntegrationEventHandler : IConsumer<StockReservedIntegrationEvent>
{
    private readonly IOrderRepository _orderRepository;
    private readonly IPublishEndpoint _publishEndpoint;

    public async Task Consume(ConsumeContext<StockReservedIntegrationEvent> context)
    {
        var eventData = context.Message;
        var order = await _orderRepository.GetOrderById(eventData.OrderId);

        if (order == null) return;

        if (eventData.IsSuccessful)
        {
            // УСПЕХ: Товар зарезервирован. Переход к следующему шагу: Оплата.
            Log.Information("Stock confirmed for Order {OrderId}. Proceeding to payment.", order.Id);

            // 1. Обновление статуса заказа в локальной БД
            order.SetStatusToAwaitingPayment();
            await _orderRepository.UpdateAsync(order);

            // 2. Публикация нового события для Payment Service
            await _publishEndpoint.Publish(new OrderAwaitingPaymentIntegrationEvent(order.Id, order.TotalCost));
        }
        else
        {
            // НЕУДАЧА: Товара нет. Завершение Саги компенсацией (отменой).
            Log.Warning("Stock failed for Order {OrderId}. Cancelling order.", order.Id);

            order.SetStatusToCancelled("Insufficient stock.");
            await _orderRepository.UpdateAsync(order);

            // Публикация события OrderCancelled для оповещения пользователя и других сервисов
            await _publishEndpoint.Publish(new OrderCancelledIntegrationEvent(order.Id));
        }
    }
}
```

Изображение 13. Реализация StockReservedIntegrationEvent

2.5. Выполнение Задачи 4: Внедрение Observability, безопасности и подготовка к продакшену

2.5.1. Настройка Observability

После внедрения Serilog (логи), OpenTelemetry (трассы) и Prometheus (метрики), эти данные должны быть агрегированы и визуализированы для обеспечения наблюдаемости системы (таблица 4).

Таблица 4. Сравнения для Observability

Источник данных	**Инструмент сбора/хранения**	**Инструмент визуализации/анализа**	**Основная задача**
Метрики	Prometheus (Сбор данных с экспортеров)	Grafana	Мониторинг RPS, latency, CPU/Memory каждого сервиса. Создание алертов.
Логи	Seq / ELK Stack	Seq / Kibana	Анализ структурированных логов, поиск по TraceId для отладки.
Трассы	Zipkin / Jaeger	Zipkin / Jaeger UI	Сквозной анализ выполнения запроса, выявление узких мест и ошибок (например, где произошел timeout).

Алертинг (Prometheus + Alertmanager): Ключевые алерты:

- HighLatency: Среднее время ответа (p95) для Ordering Service превышает 500мс в течение 5 минут.

- CircuitBreakerOpen: Circuit Breaker открылся в любом сервисе (сигнал о недоступности критического зависимого сервиса).
- HighErrorRate: Уровень HTTP 5xx ошибок превышает 5% за 1 минуту.

2.5.2. Безопасность (mTLS и защита API Gateway)

I. Защита API Gateway (Ocelot/YARP):

API Gateway (единственная публичная точка входа) обеспечивает первую линию защиты, валидируя JWT-токены, выпущенные Identity Service.

- **Настройка JWT-схемы:** Gateway настраивается на проверку подписи токена с использованием публичного ключа Identity Service.
- **Маршрутизация с авторизацией (изображение 14)**

```
// Ocelot Gateway configuration (ocelot.json)
{
  "Routes": [
    {
      "DownstreamPathTemplate": "/api/{everything}",
      "DownstreamScheme": "http",
      "DownstreamHostAndPorts": [ { "Host": "catalogservice", "Port": 80 } ],
      "UpstreamPathTemplate": "/catalog/{everything}",
      "UpstreamHttpMethod": [ "Get" ],
      "AuthenticationOptions": {
        "AuthenticationProviderKey": "Bearer", // Проверка JWT
        "AllowedScopes": []
      }
    },
    {
      "DownstreamPathTemplate": "/api/order",
      // ...
      "ClientWhitelist": [ "MobileApp", "WebApp" ], // Ограничение доступа по ID клиента
      "AddClaimsToRequest": [ // Извлечение Claims и передача в сервис
        { "Type": "user_id", "Value": "{http://schemas.xmlsoap.org/ws/2005/05/identity/claims/nameidentifier}" }
      ],
      "RouteClaimsRequirement": { "Role": "Customer" } // Требование роли
    }
  ]
}
```

Изображение 14. Реализация маршрутизации с авторизацией

II. Взаимная Аутентификация (mTLS) для внутренней сети:

Для обеспечения безопасности внутренней коммуникации (между сервисами, а не между Gateway и сервисом) используется **mTLS (mutual TLS)**.

- **Механизм:** Каждый микросервис имеет свой TLS-сертификат и настроен на то, чтобы **требовать** сертификат от вызывающего сервиса.
- **Преимущество:** Гарантирует, что только доверенные, аутентифицированные сервисы могут взаимодействовать друг с другом внутри кластера Kubernetes, даже если злоумышленник получит доступ к внутренней сети.

2.6. Выполнение Задачи 5: Оркестрация, развертывание и тестирование

2.6.1. Контейнеризация

Каждый микросервис контейнеризируется в отдельный образ Docker (изображение 15).

```
# 1. BUILD Stage: Сборка и компиляция
FROM mcr.microsoft.com/dotnet/sdk:8.0 AS build
WORKDIR /src
COPY ["Ordering/Ordering.csproj", "Ordering/"]
RUN dotnet restore "Ordering/Ordering.csproj"
COPY . .
WORKDIR "/src/Ordering"
RUN dotnet build "Ordering.csproj" -c Release -o /app/build

# 2. PUBLISH Stage: Публикация в виде готового приложения
FROM build AS publish
RUN dotnet publish "Ordering.csproj" -c Release -o /app/publish /p:UseAppHost=false

# 3. FINAL Stage: Окончательный образ (минимальный runtime)
FROM mcr.microsoft.com/dotnet/aspnet:8.0 AS final
WORKDIR /app
COPY --from=publish /app/publish .
ENTRYPOINT ["dotnet", "Ordering.dll"]
# Образ final использует минимальный ASP.NET runtime, что снижает размер и уязвимости.
```

Изображение 15. Dockerfile (Пример для Ordering Service)

2.6.2. Оркестрация

Kubernetes (K8s) используется для автоматического управления жизненным циклом (развертывание, масштабирование, самовосстановление).

- **Deployment:** Определяет желаемое состояние (например, 5 реплик Catalog Service) (изображение 16).
- **Service:** Определяет, как получить доступ к набору подов (например, catalog-service доступен по внутреннему DNS-имени http://catalogservice).
- **ConfigMap / Secret:** Хранение конфигурации (строки подключения, JWT-секреты).

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: cart-service
spec:
  replicas: 5 # 5 инстансов для высокой нагрузки
  selector:
    matchLabels:
      app: cart-service
  template:
    metadata:
      labels:
        app: cart-service
    spec:
      containers:
      - name: cart-service
        image: your-registry/cart-service:latest
        ports:
        - containerPort: 80
        env: # Переменные среды для Redis
        - name: Redis__Host
          valueFrom:
            configMapKeyRef:
              name: infrastructure-config
              key: redis.host
        resources:
          limits: # Ограничение ресурсов для изоляции сбоев
            memory: "128Mi"
            cpu: "250m"
        readinessProbe: # Проверка готовности к приему трафика
          httpGet:
            path: /health/ready
            port: 80
```

Изображение 16. Kubernetes Deployment (Пример для Cart Service)

2.6.3. CI/CD Пайплайн

Пайплайн, реализованный, например, через **GitHub Actions** или **Azure DevOps**, автоматизирует весь процесс.

1. **Сборка (Build):** Запуск dotnet restore, dotnet build для всех сервисов.
2. **Тестирование (Test):** Запуск Unit- и Интеграционных тестов.

3. **Контейнеризация:** Сборка Docker-образов и пометка (tagging) их SHA-хешем.
4. **Публикация (Push):** Отправка Docker-образов в частный Container Registry.
5. **Развертывание (Deploy):** Обновление манифестов Kubernetes (например, через Helm или Kustomize) для использования нового образа и применение изменений в кластер.

Заключение

В рамках данной научно-исследовательской работы была успешно спроектирована и реализована отказоустойчивая микросервисная архитектура для бэкенда интернет-магазина. Все поставленные задачи были выполнены:

- **Задача 1 (Архитектура):** Проведена декомпозиция на 5 Bounded Contexts с использованием DDD и принципа Database per Service, реализована гибридная коммуникация (gRPC, RabbitMQ).
- **Задача 2 (Инфраструктура):** Внедрены механизмы Observability (Serilog, OpenTelemetry) и Устойчивости (Polly: Circuit Breaker, Retry) для защиты межсервисных вызовов.
- **Задача 3 (Бизнес-логика):** Реализована ключевая логика Ordering Service, включая координацию распределенной транзакции через шаблон **Хореографии Саги**, что обеспечивает согласованность данных между изолированными сервисами.
- **Задача 4 (Prod Readiness):** Обеспечена безопасность на уровне API Gateway (JWT) и на внутреннем уровне (планирование mTLS), настроена система мониторинга (Prometheus, Grafana).
- **Задача 5 (Развертывание):** Все сервисы контейнеризированы (Docker) и готовы к оркестрации в Kubernetes, а также спроектирован автоматизированный CI/CD пайплайн.

Список использованных источников

1. Fowler, Martin. *Patterns of Enterprise Application Architecture*. Addison-Wesley Professional, 2002.
2. Newman, Sam. *Building Microservices: Designing Fine-Grained Systems*. O'Reilly Media, 2015.
3. Evans, Eric. *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Addison-Wesley Professional, 2003.
4. Microsoft. *Architecting Modern Web Applications with ASP.NET Core and Azure*. [Электронный ресурс]. -URL: https://learn.microsoft.com/en-us/dotnet/architecture/modern-web-apps-azure/ (дата обращения: 1.10.2025)
5. Polly Documentation. *Resilience and Transient-Fault-Handling Library for .NET*. [Электронный ресурс]. -URL: https://www.pollydocs.org/ (дата обращения: 9.10.2025)
6. OpenTelemetry Documentation. *Observability Framework*. [Электронный ресурс].-URL: https://opentelemetry.io/docs (дата обращения: 15.10.2025)
7. Kubernetes Documentation. *Container Orchestration System*. [Электронный ресурс].-URL: https://kubernetes.io/docs/home (дата обращения: 19.11.2025)