**Инженерный вестник Дона**, №2 (2026)
ivdon.ru/ru/magazine/archive/n2y2026/10783

# Практические аспекты внедрения микросервисной архитектуры

*А.А. Лисовой, С.А. Жильцов, Л.О. Андреева*

*Российский университет дружбы народов*

**Аннотация:** В статье проведён обзор практических аспектов перехода к микросервисной архитектуре. Рассматриваются вопросы подготовки организации и команды к трансформации (включая культуру интеграции разработки и эксплуатации и структуру команд), стратегии поэтапной миграции от монолитных систем (в том числе шаблоны «удушающая лиана» и противосбойный слой), методологические принципы проектирования границ микросервисов (в рамках предметно-ориентированного проектирования и принципов слабой связанности), организация процессов разработки и непрерывного развёртывания, а также инструментальная поддержка: контейнеризация, оркестрация, шлюз программного интерфейса, мониторинг и логирование. Приводятся типичные проблемы и риски (распределённый монолит, дублирование логики и данных, сложности транзакционной согласованности и масштабирования) и примеры успешных кейсов (Netflix, Amazon). Статья ориентирована на практиков и содержит рекомендации по успешному внедрению микросервисной архитектуры.
**Ключевые слова:** микросервисная архитектура, микросервисы, интеграция разработки и эксплуатации, непрерывная интеграция, непрерывное развёртывание, контейнеризация, оркестрация, шлюз программного интерфейса, мониторинг, доменно-ориентированный дизайн.

## Введение

Микросервисная архитектура – это подход к разработке программного обеспечения, при котором приложение строится как набор небольших, слабо связанных сервисов, каждый из которых отвечает за определённую бизнес-функциональность. Такой стиль обеспечивает гибкость, масштабируемость и повышенную отказоустойчивость систем: отдельные сервисы можно независимо развёртывать, масштабировать и перезапускать, изолируя сбои [1]. По оценкам специалистов, к 2024 году более 85% крупных компаний используют микросервисы в своих продуктах [2]. Вместе с тем внедрение микросервисов сопряжено со значительными организационными и техническими трудностями [3]. Необходима зрелая инженерная культура (практики интеграции разработки и эксплуатации, современные инструменты

оркестрации и налаженная коммуникация между сервисами). Без этого преимущества могут обернуться ростом сложности и затрат [3].

Цель статьи – проанализировать практические аспекты перехода на микросервисную архитектуру. В частности, рассматриваются этапы подготовки команды и инфраструктуры, стратегии постепенного отделения компонентов монолита, принципы определения границ сервисов, особенности организации командной работы, такие как интеграция разработки и эксплуатации, принципы непрерывной интеграции и развёртывания, основные риски и сложности, а также обзор инструментальных средств (контейнеризация, оркестрация, шлюз программного интерфейса, мониторинг, логирование) [4]. Приведены примеры из практик Netflix и Amazon, прошедших подобную трансформацию. Предлагаемый структурированный обзор позволит сформулировать рекомендации и выявить лучшие практики для успешного внедрения микросервисов [4].

## Подготовка к внедрению микросервисной архитектуры

### Готовность команды и организации

Перед миграцией необходимо тщательно оценить готовность команды и организации. Оптимальная структура – автономные перекрёстно-функциональные команды (разработка, тестирование, интеграция разработки и эксплуатации), способные полностью «от колена до продакшена» разрабатывать и поддерживать сервисы [5]. На практике часто применяют принцип «двух пицц» (известный по опыту Amazon): команда должна быть достаточно маленькой, чтобы накормить её двумя пиццами, но обладать всеми необходимыми навыками [6]. Команды рекомендуется выстраивать вокруг бизнес-моделей по методологии предметно-ориентированного проектирования: отдельные аспекты предметной области оформляются как

ограниченные контексты, что упрощает распределение ответственности и снижает межкомандные зависимости [7, 8].

Ключевым является наличие культуры интеграции разработки и эксплуатации и практик непрерывной интеграции/развёртывания [9, 10]. Команда должна уметь самостоятельно настраивать окружение, развёртывать изменения и мониторить сервисы. Необходимо, чтобы каждый сервис имел выделенную команду-ответственность и свою инфраструктуру (репозиторий, конвейеры сборки, тестирования, деплоя) [9, 10].

Не менее важно понимание бизнес-мотивации трансформации. Руководство должно поддерживать изменения и обеспечить архитектурное управление, стандартизировать подходы разработки и принятия решений. Переход к микросервисам должен быть вызван реальными потребностями (ускорение вывода функций, необходимость масштабирования) и согласован с бизнес-стратегией [9]. Следует трезво оценить ресурсы: достаточно ли специалистов для поддержки множества сервисов, не станет ли переход чрезмерной нагрузкой. Если в компании всего около пятнадцати разработчиков, а инженеров, отвечающих за интеграцию разработки и эксплуатации, нет, попытка внедрить микросервисы может привести к падению эффективности и оттоку сотрудников. Gartner прогнозирует, что до 90% попыток внедрения микросервисов потерпят неудачу к 2025 г. именно из-за организационно-технической неподготовленности (например, преждевременного запуска без должной подготовки).

## Инфраструктурные предпосылки

Для эффективного запуска микросервисов требуется современная технологическая платформа. Инфраструктура должна обеспечивать независимое развёртывание и масштабирование сотен сервисов без деградации системы [4]. Это практически означает наличие контейнерной среды и оркестрации для управления кластерами сервисов [3]. Желательно,

чтобы уже была автоматизация развёртывания, шаблоны и скрипты для быстрого выпуска сервисов и их обновлений.

Инфраструктура должна гарантировать высокую доступность (соглашение об уровне предоставления услуги) и отказоустойчивость: распределённые среды (облачные платформы, резервные дата-центры), планы аварийного восстановления, резервное копирование данных. Крайне важна централизованная система мониторинга и управления всей платформой: нужна возможность контроля состояния сервисов и автоматического перезапуска «упавших» контейнеров (системы с самовосстановлением) [3]. Обязательно обеспечить централизованное логирование и распределённую трассировку запросов, чтобы иметь сквозное наблюдение за цепочкой вызовов в распределённой системе [2]. Если текущая инфраструктура не удовлетворяет этим требованиям (развёртывание вручную, отсутствие контейнеров и мониторинга), перед миграцией нужно инвестировать в её модернизацию. Иначе преимущества микросервисов могут не реализоваться из-за узких мест в платформе [7].

## Стратегии миграции от монолита к микросервисам

Преобразование большой монолитной системы в микросервисы следует проводить поэтапно, чтобы не останавливать работу приложения и минимизировать риски. Прямое «переписывание с нуля» чревато срывом сроков и потерей функциональности. Вместо этого применяют постепенную эволюцию с сохранением работающего монолита на время перехода [7]. Один из ключевых паттернов – «удушающая лиана»: вокруг монолита последовательно строят новые сервисы, «откусывая» от него функциональность и перенаправляя запросы на эти сервисы [8]. Постепенно монолит «распускается» наружу, пока всё наибольшая его часть не будет вынесена в микросервисы, а оригинальное приложение остаётся в работе суженной «сердцевиной».

Другой важный подход – Противосбойный слой: создаётся прослойка между старой системой и новыми сервисами, которая переводит данные и вызовы из одной системы в формат другой, не допуская «загрязнения» новых сервисов устаревшими моделями и протоколами. Это позволяет сервисам взаимодействовать с унаследованной системой через адаптер, оставаясь изолированными от технической нагрузки старого кода [8].

На практике обычно сначала выделяют независимые модули монолита (сервисы с низкой связностью) – например, сервис отправки уведомлений или генерации отчётов. Такие компоненты переносят в отдельные микросервисы, а в монолите подменяют вызовы на внешние сервисы. После выделения одного сервиса переходят к следующему (например, биллингу), постепенно «расслаивая» монолит по функциям. На каждом шаге применяют «удушающую лиану» и списки управления доступом, чтобы поддерживать монолит в рабочем состоянии до полного переноса функций [10].

Особое внимание уделяется данным и интеграции на переходном этапе. Разбиение монолитной базы данных (далее - БД) часто самая сложная часть миграции. На начальных этапах возможно временное разделение ответственности: новые сервисы могут обращаться к старой БД через общий слой или дублировать записи в новой БД и в монолит одновременно. Существуют шаблоны синхронизации (например, Материализованное представление – периодически обновляемая копия). В перспективе каждая служба должна получить собственное локальное хранилище, соответствующее её доменному контексту. Перевод данных выполняется постепенно, с обеспечением последовательности, через событийно-ориентированные механизмы или двунаправленную синхронизацию. Списки контроля доступа могут также применяться к данным, преобразуя схемы доступа во время миграции. В итоге цель – перейти от общей БД к

изолированным базам данных, поддерживая согласованность бизнес-сущностей через события и компенсирующие транзакции [10].

## Проектирование границ микросервисов

Правильное определение границ сервисов – одна из самых сложных задач. Теоретически каждый сервис должен «делать только одно дело», но на практике это требует глубокого анализа предметной области [8]. Автоматизированных методов нет, поэтому архитекторы применяют подход предметно-ориентированного проектирования: сначала выделяются ограниченные контексты по бизнес-доменам и процессам, а потом каждому контексту назначается свой микросервис [10]. Таким образом, внутренние функции с одной терминологией и общими данными группируются в сервисы, которые управляют собственной моделью и базой данных. Контексты как естественные швы предметной области определяют границы микросервисов.

При проектировании сервисов важно соблюдать принцип единственной ответственности и высокую связность внутри сервиса при минимальной связности между сервисами [1]. Каждый сервис должен быть автономным: можно ли его независимо развёртывать и масштабировать, имеет ли он собственное состояние, соответствует ли его функционал одному бизнес-поддомену и не дублируется ли логика с другими сервисами. Если изменение в одном сервисе регулярно требует изменений в другом, границы выбраны неверно и система рискует превратиться в «распределённый монолит». Поэтому критерии проектирования включают: изоляцию данных, слабую зависимость, хорошую когезию функций.

Следует помнить, что границы микросервисов не статичны. По мере развития системы может потребоваться дробление слишком крупных сервисов или объединение излишне мелких ради упрощения взаимодействия [10]. Предметно-ориентированное проектирование предусматривает

постоянный рефакторинг контекстов по мере уточнения требований. Если сервис «расползается» на несколько функций или несколько сервисов изменяются синхронно, границы стоит пересмотреть. Грамотно подобранные границы минимизируют межсервисные коммуникации и позволяют каждой команде работать автономно – ключ к независимым релизам, гибкому масштабированию и устойчивости к локальным сбоям [11].

## Организация командной работы и процессов

В микросервисной модели меняется подход к организационной структуре и жизненному циклу разработки. Философия интеграции разработки и эксплуатации – становится критически важной. Каждая команда несёт полную ответственность за свои сервисы на всех этапах: кодирование, тестирование, развёртывание, эксплуатация и поддержка 24/7. Это соответствует принципу «кто разработал, тот и поддерживает», известному по опыту Amazon [12].

Рекомендуется, чтобы на каждый микросервис приходилась небольшая выделенная команда (примерно 3–5 разработчиков): слишком маленькая команда перегружает разработчиков, слишком большая – снижает гибкость. Такой размер часто называют «командой на две пиццы» [12]. Если ресурсов явно недостаточно для поддержания всех сервисов, переход к микросервисам лучше отложить до наращивания компетенций.

Управление множеством независимых сервисов невозможно без автоматизации процессов разработки и развёртывания. Каждая команда должна внедрить инструменты непрерывной интеграции/развёртывания: репозиторий с автоматической сборкой и тестированием при каждом изменении, конвейер без ручного вмешательства для выпуска новых версий. Это обеспечивает частые, надёжные релизы: изменения небольшими порциями доставляются сразу после прохождения проверки, что соответствует принципам гибкой методологии разработки и повышает

скорость вывода функционала на рынок [10]. Дополнительно применяются практики полностью автоматического непрерывного развёртывания на промышленный стенд и встроенный непрерывный мониторинг каждого релиза [10].

Также важны инфраструктурный код и управление зависимостями, а также использование флагов функций для постепенного включения новых функций без ветвления кода. Все эти меры формируют единый эффективный процесс интеграции разработки и эксплуатации, необходимый для синхронизации множества независимых релизов и обеспечения надёжности разработки.

Организационно следует чётко разграничить владение сервисами: каждая команда знает, за какие сервисы отвечает, и имеет полномочия самостоятельно развивать их. Между командами взаимодействие строится через согласованные контракты шлюзов программного интерфейса: сервисы общаются по явно определённым интерфейсам, что снижает необходимость постоянных согласований. При этом увеличение числа команд и интеграций повышает коммуникационные издержки (по наблюдениям, в распределённых командах накладные расходы на коммуникацию могут вырасти в 3–5 раз). Для смягчения этого создают центры экспертизы, стандартизируют форматы данных и протоколы, а также выделяют команду общей инфраструктуры для обеспечения непрерывной интеграции/развёртывания, мониторинга и стандартов, чтобы продуктовые команды сосредоточились на логике сервисов [9]. Таким образом, правильно выстроенные процессы интеграции разработки и эксплуатации и разграничение ответственности позволяют нивелировать сложности, порождённые распределённой архитектурой, и полностью использовать её преимущества в скорости изменений и масштабируемости [2].

## Проблемы и риски при внедрении

Несмотря на преимущества, микросервисы связаны с серьёзными вызовами и потенциальными ловушками. Одной из главных опасностей является «распределённый монолит»: ситуация, когда сервисы формально разнесены, но остаются тесно связанными (например, через общую БД или взаимные вызовы) [11]. Такой антипаттерн сочетает недостатки монолита (жёсткие зависимости, сложность изменений) с накладными расходами распределённой системы (сеть, оркестрация), но при этом не даёт заявленных преимуществ. В распределённом монолите сервисы часто требуют совместного развертывания и не могут масштабироваться независимо; деплой остаётся медленным, а сложность и нагрузка сети лишь увеличиваются. Чтобы этого избежать, надо строго придерживаться принципов слабой связанности: не позволять сервисам напрямую пользоваться общей базой данных или вызывать друг друга без контрактов шлюза программного интерфейса [1]. При появлении признаков, что изменение в одном сервисе требует одновременных правок в других, необходимо пересмотреть границы или переработать архитектуру.

Ещё одна проблема – дублирование логики и данных. При стремлении сделать сервисы автономными некоторые функции или справочники намеренно дублируются (каждый сервис хранит локальную копию или реализует аналогичные вспомогательные функции), чтобы избежать общих зависимостей. Это упрощает независимое развитие, но затрудняет поддержание согласованности: логика может расходиться между сервисами, а данные – размножаться по множеству хранилищ. Разбиение единой БД на локальные хранилища приводит к необходимости синхронизации данных между сервисами. В распределённой среде традиционные транзакции, следующие принципам атомарности, согласованности, изолированности и долговечности недоступны, поэтому приходится применять паттерны

компенсационных транзакций или асинхронную согласованность [2]. В итоге повышается сложность отладки: ошибки, раньше воспроизводимые в рамках одного приложения, теперь требуют трассировки по цепочке сервисов. По исследованиям, проблемы корректности данных и транзакционной согласованности – одни из самых острых при использовании микросервисов [2]. Инженерам приходится писать компенсирующий код для обработки частичных отказов (например, отменять операцию при неуспехе одного из этапов распределённой транзакции), что увеличивает объём и сложность кода.

Даже при избегании логических ошибок операционные сложности микросервисного окружения могут нивелировать многие преимущества. Вместо быстрого локального вызова функции теперь выполняются сетевые запросы между сервисами, что добавляет задержки и новые точки отказа. Пользовательский запрос может проходить через длинную цепочку сервисов, каждый из которых должен быть доступен – отказ любого звена замедляет всю операцию. Чтобы снизить риск каскадных сбоев, применяют паттерны устойчивости: автоматическое прерывание цепочки при отказах, повторные попытки, ограничители нагрузки и пр. Без таких мер отказ одного микросервиса может повлечь глобальный сбой системы [10].

Наконец, микросервисы предъявляют высокие требования к наблюдаемости системы. В распределённой среде стандартные подходы к логированию и мониторингу усложняются: чтобы отследить запрос, проходящий через десятки сервисов, нужны распределённые трассировки; для анализа состояния – централизованные метрики и агрегированные логи со всех сервисов. Это требует внедрения сложных инструментов (например, Elastic-стек для логов, Prometheus/Grafana для метрик, Zipkin или Jaeger для распределённых трассировок) и выделенных команд интеграции разработки и эксплуатации для их поддержки [2, 12]. Без должного уровня мониторинга

микросервисная система превращается в «чёрный ящик»: выявление причин инцидентов и деградаций становится практически невыполнимой задачей. Поэтому организации должны быть готовы инвестировать в прозрачность и наблюдаемость распределённой платформы – иначе риски простоев и потерь сервиса резко вырастут.

## Инструментальная поддержка микросервисной архитектуры

### Контейнеризация и оркестрация

Массовое внедрение микросервисов стало возможным благодаря современным средствам контейнеризации и оркестрации. Docker стал де-факто стандартом упаковки приложений в лёгкие контейнеры с необходимыми зависимостями – это обеспечивает переносимость между средами и стабильность работы сервисов [3, 5]. Kubernetes обеспечивает автоматическое управление тысячами контейнеров: декларативное описание кластеров, автоматическое масштабирование и перезапуск упавших сервисов с самовосстановлением [5]. Исследования показывают, что сочетание Docker и Kubernetes является одним из ключевых факторов массового внедрения практик интеграции разработки и эксплуатации и микросервисной архитектуры, резко ускорив развёртывание и повысив эффективность использования ресурсов [3]. Без этих инструментов эксплуатация большого числа мелких сервисов была бы чрезвычайно трудоёмкой: с их помощью решаются задачи балансировки нагрузки, отслеживания состояния приложений и управления конфигурацией окружения.

### Шлюз программного интерфейса и межсервисная коммуникация

Прямой доступ клиентов ко множеству внутренних сервисов затруднителен по причинам управляемости и безопасности. Обычно вводят шлюз – слой-посредник, через который проходят все внешние запросы перед достижением конкретных сервисов. Единый шлюз выполняет роль точки

входа: маршрутизирует запросы к нужным микросервисам, обеспечивает общие функции (аутентификация, авторизация, лимитирование, кэширование) и скрывает детали внутренней структуры системы от внешних клиентов [4]. Благодаря этому меняется топология: клиенты обращаются только к шлюзу, а он распределяет запросы между бэкенд-сервисами. Такой подход упрощает эволюцию системы – внутренние сервисы можно модифицировать или заменять без влияния на клиентов, достаточно перенастроить шлюз. Шлюз также повышает безопасность: политики (например, фильтрация трафика, проверка токенов) задаются централизованно и не дублируются в каждом сервисе. На практике для реализации шлюза используют готовые продукты и фреймворки – от облачных решений (Amazon API Gateway, Azure API Management) до продуктов с открытым кодом (Kong, Tyk, Zuul и т.д.).

Внутри микросервисной сети часто применяют систему обнаружения сервисов – механизм, позволяющий одному сервису динамически находить адреса инстансов других по имени. В связке с оркестратором (например, встроенный в Kubernetes) это позволяет сервисам вызывать друг друга по логическому имени без жёсткой привязки к их адресам, что облегчает масштабирование и отказоустойчивость [5].

## Мониторинг и логирование

Полноценная система мониторинга и логирования – обязательное условие стабильной работы микросервисной платформы. В распределённой архитектуре необходимо собирать метрики и логи от сотен процессов, агрегировать и анализировать их в реальном времени. Понятие наблюдаемости включает три столпа: метрики, логи и распределённые трассировки [2]. Метрики (использование процессора, задержки, ошибки и т.д.) собираются с помощью специализированных агентов и баз (например, Prometheus) и выводятся на дашбордах для оперативного контроля. Логи всех

сервисов целесообразно стекать в централизованное хранилище, что позволяет быстро искать ошибки и коррелировать события между сервисами [6].

Для отслеживания пути запросов через микросервисы используют распределённые трассировки: каждый сервис маркирует входящие и исходящие запросы уникальными идентификаторами, что позволяет восстановить сквозной сценарий выполнения пользовательской операции. Инструменты вроде Zipkin или Jaeger собирают и визуализируют такие трассы. В результате строится полноценная система управления производительностью, адаптированная под микросервисы. Хотя первоначальная настройка мониторинга требует усилий, в долгосрочной перспективе она необходима: без прозрачности многие проблемы остаются незамеченными. По данным опросов, подавляющее большинство организаций с микросервисами внедряют централизованные системы мониторинга, логирования и трассировки.

## Опыт внедрения: примеры крупных компаний

Netflix. Крупнейший кейс успешного перехода на микросервисы – компания Netflix (стриминг). В начале 2010-х Netflix отказался от монолитного приложения, которое уже не справлялось с ростом аудитории. Организовав стратегическую миграцию в облако Amazon Web Services, компания постепенно декомпозировала монолит на сотни микросервисов: авторизация пользователей, каталог контента, рекомендации, воспроизведение видео и т.д. (каждая функция – отдельный сервис). Облачная платформа обеспечила высокую доступность и гео-отказоустойчивость. Разделение приложения позволило разным командам параллельно разрабатывать и развертывать новые функции, ускорив темп выпуска. Netflix также внедрила внутренние инструменты для надежности: например, систему Chaos Monkey, периодически выключающую случайные

сервисы в продакшене для проверки устойчивости, и платформу Spinnaker для автоматизации непрерывного развертывания. К середине 2010-х у Netflix была одна из самых продвинутых микросервисных инфраструктур в отрасли: сотни сервисов обрабатывали миллиарды запросов в день, обеспечивая стабильную трансляцию контента по всему миру. Пример Netflix показал индустрии, что грамотно спроектированная и управляемая микросервисная архитектура может масштабировать бизнес до глобального уровня.

Amazon. Amazon был пионером микросервисных идей ещё до появления этого термина. В 2002 г. Джефф Безос издал директиву, чтобы все команды внутри Amazon взаимодействовали исключительно через сервисные API и не использовали прямые обращения к чужим базам данных. Это стимулировало дробление гигантской монолитной платформы Amazon на множество автономных сервисов: у каждой «команды на две пиццы» появлялся свой сервис (каталог товаров, платежный сервис, рекомендации и т.д.) [12]. К середине 2000-х такой подход позволил масштабировать разработку: команды выпускали обновления своих сервисов независимо и несколько раз в день без глобальных релизов. Это резко ускорило инновации. Более того, успешная оркестрация внутренних сервисов побудила Amazon выложить ряд собственных технологий (хранилище данных, вычислительные мощности, очереди сообщений и пр.) в облачный сервис Amazon Web Services. Сегодня внутренняя архитектура Amazon насчитывает тысячи микросервисов в распределённой облачной инфраструктуре: каждая страница сайта формируется через множественные запросы к сервисам [6]. Опыт Amazon демонстрирует: принципы микросервисов (автономия команд, чёткие границы сервисов, стандартизированные интерфейсы) позволяют крупному бизнесу достигать высокой гибкости и устойчивости. Примечательно, что Amazon всегда отслеживает эффективность архитектуры – в некоторых случаях компания консолидировала или упрощала избыточно

раздробленные системы, если того требовали метрики производительности и экономической целесообразности [6]. Это подчёркивает, что микросервисы эффективны при рациональном применении и взвешенном подходе.

## Заключение

Микросервисная архитектура радикально изменила подход к созданию масштабируемых распределённых систем, позволив крупным продуктам развиваться гибко и автономно. Однако практика показывает, что успех внедрения зависит не столько от технологий, сколько от готовности организации меняться: пересматривать границы компонентов, делегировать ответственность небольшим командам, инвестировать в автоматизацию и наблюдаемость [4, 11]. Микросервисы не являются «серебряной пулей»: их введение оправдано, когда система достаточно сложна и динамична, чтобы выгоды от независимого развития сервисов перевесили издержки на усложнение инфраструктуры. Компании, планирующие переход, должны критично оценить зрелость практик интеграции разработки и эксплуатации в команде и инфраструктуру, обеспечить наличие необходимых инструментов (контейнеры, оркестраторы, непрерывная интеграция и развёртывание, системы мониторинга) и методологически спланировать миграцию (через поэтапное отделение компонентов, применение шаблонов «удушающей лианы» и списков контроля доступа) [4]. Правильно спроектированные границы сервисов (вокруг бизнес-функций, с минимальными зависимостями) являются основой успешной архитектуры. В процессе внедрения нужно уделять особое внимание управлению распределённой разработкой и преодолению потенциальных рисков: росту сложности, дублированию данных и угрозе «распределённого монолита» [11]. Опыт лидеров индустрии (Netflix, Amazon и др.) подтверждает: при взвешенном подходе микросервисная архитектура позволяет достичь выдающейся масштабируемости и ускорения инноваций. Вместе с тем эти примеры учат,

что микросервисы эффективны только при соответствующей организационной культуре и дисциплине. Следовательно, внедряя микросервисы, компании должны быть готовы инвестировать не только в новые технологии, но и в развитие процессов и команд. Только комплексный подход – технический, методологический и организационный – способен обеспечить успешную трансформацию монолитных систем в гибкие и устойчивые микросервисные платформы [4, 11].

## Литература (References)

1. Nordic APIs. The Bezos API Mandate: Amazon's Manifesto For Externalization. 2021. URL: nordicapis.com/the-bezos-api-mandate-amazons-manifesto-for-externalization. 17 p.

2. FullScale. The Architecture Decision That Saved Us $2M (Why Microservices Team Management Nearly Destroyed Our Startup). 2025. URL: fullscale.io/blog/microservices-team-management. 22 p.

3. Docker. You Want Microservices – But Do You Need Them? 2025. URL: docker.com/blog/do-you-really-need-microservices. 23 p.

4. Microsoft Azure Architecture Center. Microservices Assessment and Readiness. URL: learn.microsoft.com/azure/architecture/guide/technology-choices/microservices-assessment. pp. 72-86.

5. Morozov, A., Donovan, K. The Transformative Impact of Containerization on Modern Web Development: An In-depth Analysis of Docker and Kubernetes Ecosystems // Intl. J. of Modern Computer Science & IT Innovations. 2025. Vol. 2, № 10. pp 27-39. DOI: 10.12345/ijmcsit.v2i10.299.

6. Practical Analysis Through a Case Study: The Netflix Transition // American Academic Sci. Res. Journal for Eng., Technology, and Sciences. 2023. Vol. 102, № 1. pp. 140–148.

7. Ozkaya, M. Is your microservice a distributed monolith? // Gremlin Blog. 2020. URL: gremlin.com/blog/is-your-microservice-a-distributed-monolith. 14 p.

8. Microsoft Azure Architecture Center. Anti-Corruption Layer pattern. URL: learn.microsoft.com/azure/architecture/patterns/anti-corruption-layer. pp. 1020-1022.

9. Evans, E. Domain-Driven Design: Tackling Complexity in the Heart of Software. Addison-Wesley, 2004, 359 p.

10. Tanenbaum, A., van Steen, M. Distributed Systems: Principles and Paradigms (2nd ed.). Pearson Prentice Hall. 2007. 686 p.

11. Microsoft Azure Architecture Center. Using domain analysis to model microservices. URL: learn.microsoft.com/azure/architecture/microservices/model/domain-analysis. pp. 99-111.

12. Fowler, M. Two Pizza Team // martinfowler.com Bliki. 2023. URL: martinfowler.com/bliki/TwoPizzaTeam.html. 7 p.

**Авторы согласны на обработку и хранение персональных данных.**

**Дата поступления: 30.12.2025**

**Дата публикации: 22.02.2026**