DOI 10.37882/2223-2966.2026.01.20

МЕТОДОЛОГИЯ ТРАНСФОРМАЦИИ LEGACY-МОНОЛИТА В МИКРОСЕРВИСНУЮ АРХИТЕКТУРУ: DDD-ДЕКОМПОЗИЦИЯ, ОШИБКИ И МЕТРИКИ УСПЕХА НА СТЕКЕ JAVA SPRING

METHODOLOGY FOR THE TRANSFORMATION OF LEGACY MONOLITH INTO MICROSERVICE ARCHITECTURE: DDD DECOMPOSITION, ERRORS, AND SUCCESS METRICS ON THE JAVA SPRING STACK

A. Kononov

Summary. The article addresses the problem of using outdated monolithic information systems which, due to high technical debt and tight module coupling, significantly slow down the release of new functionality and hinder the digital transformation of organizations. Moreover, the relevance of the study is driven by the need to introduce a holistic methodology for migrating from legacy monoliths to a microservice architecture based on the Java Spring stack, which helps reduce time-to-product and increase the flexibility of software system development.
Materials and Methods. The research materials included scientific publications, analytical reports, and statistical data from 2020–2025, while the methodological basis comprised a comparative analysis of architectural approaches, the study of decomposition-related errors, and the synthesis of recommendations from leading technology companies.
Results. The author examined the key architectural, infrastructural, and organizational challenges of transitioning from a monolith to microservices, including faulty decomposition and insufficient dependency management. Industry research indicates that these issues are systemic, manifesting as a «distributed monolith», high service coupling, operational complexity, and reduced efficiency of change delivery. The study proposes a comprehensive migration approach incorporating DDD and enhanced observability, enabling a controlled and effective transformation of a legacy monolith.
Conclusion. The application of DDD, DevOps metrics, and staged migration improves the practical effectiveness of transitioning to microservices by reducing risks, accelerating feature delivery, and ensuring sustainable system evolution. This approach makes monolith transformation not only technically justified but also beneficial for business by increasing infrastructure flexibility and manageability.

Keywords: microservice architecture, DDD decomposition, legacy monolith, DevOps metrics (DORA), Java Spring Boot, errors, success metrics, transformation methodology.

Кононов Алексей Николаевич
Ведущий инженер разработки, Центр экспертизы web-разработки дистанционных каналов Банка ГПБ (АО)
aleksey.n.kononov@gazprombank.ru

Аннотация. Статья посвящена проблеме использования устаревших монолитных систем, которые из-за высокого технического долга и жёсткой связанности модулей значительно замедляют выпуск новых функций и тормозят цифровую трансформацию организаций. Кроме того, актуальность исследования связана с необходимостью внедрения целостной методологии перехода от наследуемых монолитов к микросервисной архитектуре на стеке Java Spring, что позволяет сократить time-to-product и повысить гибкость развития программных систем.
Материалы и методы. Материалами исследования послужили научные публикации, аналитические отчёты и статистические данные за 2020–2025 годы, а методологическая база включала сравнительный анализ архитектурных подходов, изучение ошибок декомпозиции и синтез рекомендаций ведущих технологических компаний.
Результаты. Автором рассмотрены основные архитектурные, инфраструктурные и организационные проблемы перехода от монолита к микросервисам, включая ошибочную декомпозицию и недостаточное управление зависимостями. Анализ отраслевых исследований показал, что эти трудности носят системный характер и проявляются в виде «распределённого монолита», высокой связанности сервисов, усложнённой эксплуатации и падения эффективности поставки изменений. В работе предложен комплексный подход к миграции, включающий применение DDD и развитие observability, что обеспечивает управляемую и эффективную трансформацию legacy-монолита.
Заключение. Применение DDD, DevOps-метрик и поэтапной миграции повышает практическую эффективность перехода к микросервисам, снижая риски, ускоряя выпуск функциональности и обеспечивая устойчивое развитие системы. Такой подход делает трансформацию монолита не только технически оправданной, но и выгодной для бизнеса, улучшая гибкость и управляемость инфраструктуры.

Ключевые слова: микросервисная архитектура, DDD-декомпозиция, legacy-монолит, DevOps-метрики (DORA), Java Spring Boot, ошибки, метрики успеха, методология трансформации.

Введение

Во многих организациях продолжают функционировать крупные монолитные информационные системы, созданные десятилетия назад и обеспечивающие выполнение ключевых бизнес-процессов, несмотря на их технологическую и архитектурную устаревание. Такие системы, как показывают исследования, характеризуются высоким техническим долгом, низкой структурированностью программного кода, фрагментарной документацией и высокой степенью связанности модулей, что делает даже незначительные изменения трудоёмкими и рискованными [1]. Согласно научным данным [2], монолит как единое исполняемое приложение с общей базой данных требует масштабного регрессионного тестирования при каждом обновлении, а применение устаревшего стека технологий в сочетании с отсутствием современных DevOps-практик дополнительно замедляет процессы разработки и развёртывания. В результате релизы становятся редкими и ресурсозатратными, а интеграция с внешними сервисами или цифровыми платформами требует глубокой переработки ядра системы [3]. Совокупность этих факторов приводит к увеличению time-to-product и снижению гибкости бизнеса, что делает переход от legacy-монолита к микросервисной архитектуре на современном стеке Java Spring объективно необходимым.

Анализ работ различных авторов подтверждает, что успешная трансформация устаревших монолитных систем в микросервисную архитектуру возможна лишь при соблюдении системного и методологически обоснованного подхода. Зарипова В.М. и Петрова И.Ю. подчёркивают, что значительный технический долг и архитектурная изношенность подобных систем требуют предварительного устранения глубинных структурных проблем, иначе модернизация становится крайне затруднительной [1]. В свою очередь, Шумилов М.И. указывает, что микросервисная архитектура действительно обеспечивает повышение производительности и масштабируемости высоконагруженных приложений, однако подобный эффект достигается исключительно при корректной декомпозиции и тщательном проектировании сервисов [3]. Развивая данную позицию, Радостев Д.К. и Никитина Е.Ю. подчёркивают необходимость поэтапной миграции, поскольку резкий Big-Bang-переход значительно увеличивает риски и способствует возникновению отказов [4]. Кроме того, Ольхов Д.А. обращает внимание на важность стандартизированного управления API, отсутствие которого приводит к нарушению согласованности распределённой системы и усложняет её сопровождение [5]. Завершая обобщение исследовательских подходов, Черников А.О. и Заводчикова М.Г. указывают, что результативность миграции должна оцениваться на основе формализованных метрик, позволяющих объективно определить степень достижения архитектурных и бизнес-целей.

Несмотря на указанные рекомендации, современные организации по-прежнему широко используют унаследованные монолитные системы, которые со временем накапливают архитектурную сложность, технический долг и растущие затраты на сопровождение. Ужесточение требований к скорости обновления функциональности, интеграции и масштабируемости приводит к тому, что подобные архитектуры становятся всё менее соответствующими текущим потребностям бизнеса, ограничивая развитие и замедляя вывод новых функций [6-9]. В этих условиях микросервисный подход, основанный на принципах Domain-Driven Design (DDD) и реализуемый на стеке Java Spring, выступает эффективным инструментом поэтапной модернизации, позволяя уменьшить размер монолита, повысить гибкость разработки и улучшить эксплуатационные характеристики системы. Следовательно, исследование методологии перехода от legacy-монолита к микросервисной архитектуре, анализ типичных ошибок декомпозиции и определение метрик успешности приобретают особую *актуальность* в контексте цифровой трансформации и возрастающих требований к эффективности программных систем.

Таким образом, *целью* исследования является обоснование методологии поэтапной трансформации legacy-монолита в микросервисную архитектуру с опорой на принципы DDD, анализ типовых ошибок декомпозиции и определение системы метрик успешности при использовании стека Java Spring.

Материалы и методы

Материалами исследования послужили научные публикации российских и зарубежных авторов, отраслевые аналитические отчёты и статистические данные за 2020–2025 годы, посвящённые микросервисной архитектуре, декомпозиции систем и практике миграции от монолитов. Методологическая база включала сравнительный анализ архитектурных подходов, изучение эмпирических данных о распространённых ошибках декомпозиции и синтез рекомендаций, представленных в документации ведущих технологических компаний, таких как Microsoft и Red Hat. Кроме того, в работе использовались количественные данные международных исследований (O'Reilly Media, OpsLevel, Camunda/DZone), позволившие выявить ключевые проблемные зоны при переходе к микросервисной архитектуре и обосновать необходимость системного применения DDD, CI/CD, архитектурной изоляции и метрик DORA.

Результаты и обсуждение

Переход от монолитной информационной системы к микросервисной архитектуре нередко осложняется ошибками декомпозиции, главным из которых является некорректное определение границ сервисов. На прак-

тике функциональность часто разделяется по техническим слоям, а не по предметным областям, что приводит не к модульности, а к формированию «распределённого монолита» — системы с высокой связанностью компонентов, множеством зависимостей и отсутствием возможности автономного развёртывания отдельных сервисов [4]. При таком подходе микросервисы теряют свои ключевые преимущества: независимую эволюцию, гибкое масштабирование и сокращение времени вывода изменений. В связи с этим как отечественные исследования, так и официальные методические рекомендации Microsoft подчёркивают необходимость строгого следования принципам Domain-Driven Design, обеспечивающим корректное выделение bounded context и построение архитектуры, ориентированной на реальные бизнес-домены, а не на техническое устройство исходного монолита [10].

Вместе с тем, несмотря на наличие таких рекомендаций, неверное определение границ микросервисов остаётся одной из наиболее распространённых проблем миграции, поскольку функциональность нередко дробится по техническим признакам, а не по доменным смысловым областям. Это приводит к образованию «распределённого монолита» — архитектуры с высокой связанностью, большим числом синхронных коммуникаций и отсутствием автономности сервисов. Масштаб данной проблемы подтверждается отраслевой статистикой: в исследовании O'Reilly Microservices Adoption (1502 респондента) сложности декомпозиции входят в число ведущих препятствий внедрения микросервисов, уступая лишь культурным барьерам; более половины опрошенных отмечают трудности выделения доменов и нарастающую архитектурную сложность [9]. Аналогично, рекомендации Microsoft и Red Hat подчёркивают, что корректное определение bounded context должно предшествовать созданию сервисов, поскольку нарушение данного принципа неизбежно приводит к усложнению интеграций, снижению автономности и росту хрупкости архитектуры [10].

В то же время, даже при следовании данным рекомендациям, проблемы миграции усугубляются ошибками проектирования: чрезмерная фрагментация увеличивает сетевые накладные расходы, тогда как крупные, недостаточно разделённые сервисы сохраняют ограничения монолита и препятствуют изоляции изменений. Дополнительные трудности возникают при отказе от промежуточных архитектурных этапов, таких как модульный монолит, что делает переход резким и трудно контролируемым (табл. 1). На инфраструктурном уровне значительными барьерами выступают сохранение общей базы данных, несогласованность API и недостаточная автоматизация CI/CD, что способствует формированию «распределённого монолита» и повышает риск нестабильных релизов.

Таблица 1.

Сравнение монолита, модульного монолита и микросервисной архитектуры

Характеристика	Монолитная архитектура	Модульный монолит	Микросервисная архитектура
Связанность компонентов	Высокая, жёсткие внутренние зависимости	Средняя, зависимости ограничены границами модулей	Низкая, слабосвязанные сервисы
Границы ответственности	Отсутствуют или размыты	Чётко выделенные модули внутри единого приложения	Строгие bounded context, автономные домены
Автономность развёртывания	Нет — развёртывается весь монолит	Частичная: единый артефакт, но независимые модули	Полная: каждый сервис развёртывается независимо
Хрупкость изменений	Высокая: любое изменение затрагивает весь код	Средняя: ошибки локализуются на уровне модулей	Низкая: проблема ограничена отдельным сервисом
Изоляция данных	Общая база данных	Частичная изоляция схем	Полная изоляция: database-per-service
Сложность интеграции	Низкая, всё в одном процессе	Средняя	Высокая: требуется API gateway, сервисная сеть
Сетевые накладные расходы	Минимальные	Низкие	Высокие из-за межсервисных коммуникаций

Источник: Составлено автором

Кроме того, указанные инфраструктурные ограничения усугубляются недостаточной наблюдаемостью и слабо проработанными механизмами безопасности, что дополнительно снижает прозрачность и устойчивость системы. Даже при наличии технических решений трансформация остаётся неэффективной, если организационная структура компании по-прежнему ориентирована на функциональные, а не доменные команды. Следовательно, успешная миграция возможна лишь при сочетании архитектурной дисциплины, развития инфраструктурных практик и адаптации процессов разработки к требованиям распределённой среды [4, 5].

В связи с этим объективная оценка успешности миграции может быть обеспечена с использованием метрик DORA, которые отражают скорость и надёжность процессов поставки изменений. Увеличение частоты развёртываний, сокращение Lead Time, снижение доли неудачных релизов и уменьшение MTTR демонстрируют,

что сервисная изоляция, автоматизация CI/CD и применяемые архитектурные решения дают практический эффект, повышая предсказуемость изменений и уменьшая time-to-product [6]. Значимую роль в оценке глубины проведённой декомпозиции играют архитектурно-технические метрики: доля функциональности, перенесённой в автономные сервисы, сокращение остаточного монолита, корректность выделения bounded context, снижение числа межсервисных вызовов, отсутствие циклических зависимостей и наличие изолированных хранилищ данных. Эти показатели отражают зрелость архитектуры и напрямую влияют на производительность, масштабируемость и отказоустойчивость системы [7].

Дополняя архитектурный анализ, бизнес-метрики позволяют оценить стратегическую ценность миграции. Сокращение сроков вывода новых функций, повышение скорости реакции на изменение требований, снижение затрат на сопровождение и рост удовлетворённости пользователей подтверждают, что микросервисная архитектура обеспечивает не только технический, но и значимый бизнес-эффект [8]. Однако, несмотря на указанные преимущества, ошибки декомпозиции, архитектурные ограничения и организационно-инфраструктурные проблемы имеют системный характер и существенно влияют на успешность миграции от монолита к микросервисной архитектуре. Для структурирования выявленных трудностей и обобщения наиболее распространённых причин сбоев в проектах трансформации ниже приводится систематизированный перечень типичных ошибок, характеризующих основные архитектурные, технические и организационные риски перехода (табл. 2).

Согласно данным отраслевых исследований, проблемы некорректной декомпозиции и неуправляемого роста связей между компонентами имеют системный характер и широко распространены в практике внедрения микросервисной архитектуры. Так, результаты исследования OpsLevel показывают, что 62 % организаций рассматривают управление зависимостями как значительную проблему, что свидетельствует о сложности поддержки взаимодействующих сервисов в распределённой среде [11]. Аналогичные тенденции выявлены в опросе O'Reilly Media: 37 % респондентов отмечают декомпозицию как одну из ключевых задач, а 56 % указывают на трудности координации множества сервисов и общую архитектурную усложнённость [9]. Исследование Camunda / DZone дополняет эту картину, фиксируя, что 59 % компаний испытывают недостаточную видимость сквозных (end-to-end) бизнес-процессов, проходящих через цепочку микросервисов [12]. Эти данные подтверждают, что ошибки выделения сервисов и управления их зависимостями являются не единичными, а устойчивыми и статистически подтверждёнными проблемами современной микросервисной экосистемы. Визуализация основных результатов указанных исследований представлена на рисунках 1 и 2, демонстрирующих распространённость ключевых проблем и их влияние на архитектурную сложность распределённых систем (рис. 1 и 2).

Таблица 2.
Типовые ошибки при трансформации legacy-монолита в микросервисную архитектуру

Ошибка	Описание	Как проявляется в проекте и эксплуатации
1. Неверная DDD-декомпозиция	Некорректное выделение предметных областей и контекстов	Сервисные границы не совпадают с бизнес-функциями; растёт число междоменных изменений
2. Декомпозиция по техническим слоям	Деление функциональности по архитектурным слоям, а не по доменным границам	Образуется сервисная сетка, полностью повторяющая монолит; бизнес-процессы «разорваны»
3. Ошибочная гранулярность сервисов	Слишком крупные или чрезмерно мелкие сервисы	Неэффективная маршрутизация запросов; усложнение трассировки; высокий overhead сервисной сети
4. Общая БД вместо доменной изоляции	Реальное отсутствие data ownership	Миграция зависает из-за блокировки схемы; изменения данных становятся дорогостоящими
5. Несформированная интеграционная стратегия	API появляются стихийно, отсутствует единая политика интеграции	Усложняется версионирование; растёт количество точек отказа между сервисами
6. Недостаточная автоматизация DevOps	Частичное внедрение CI/CD, ручные процедуры	Возрастает время поставки; релизы нестабильны; труднее применять Canary и Blue-Green
7. Игнорирование observability	Отсутствуют сквозная трассировка, метрики, логирование	Ошибки в распределённой системе диагностируются медленно; MTTR высокое
8. Технический долг монолита	Старый код, слабая модульность, отсутствие документации	Миграция идёт медленно; трансформация обходится дороже, чем планировалось
9. Big-Bang-миграция	Переход без модульного монолита, адаптеров, фасадов	Срывы сроков, частые откаты, непредсказуемое состояние архитектуры
10. Несоответствие оргструктуры микросервисам	Команды остаются функциональными вместо доменных	Зависимости между командами не уменьшаются; поток изменений замедлен

Источник: Составлено автором

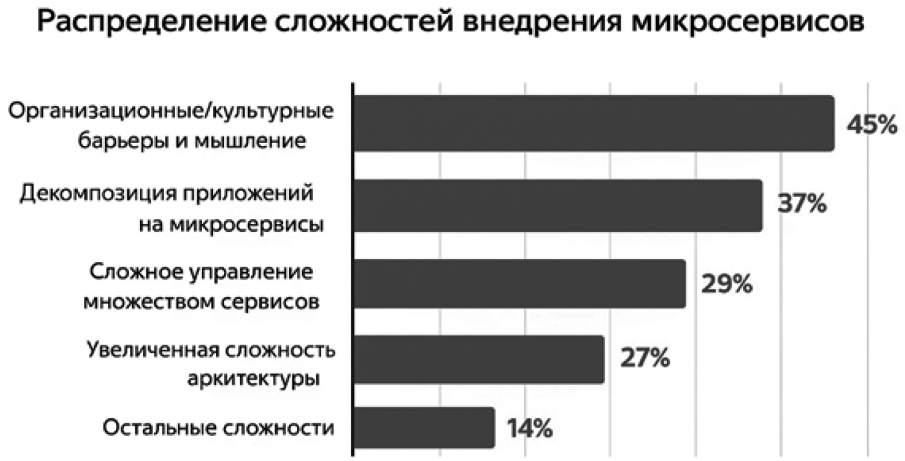

Рис. 1. Распределение сложностей внедрения микросервисов

Источник: Составлено автором на основании [9–12]

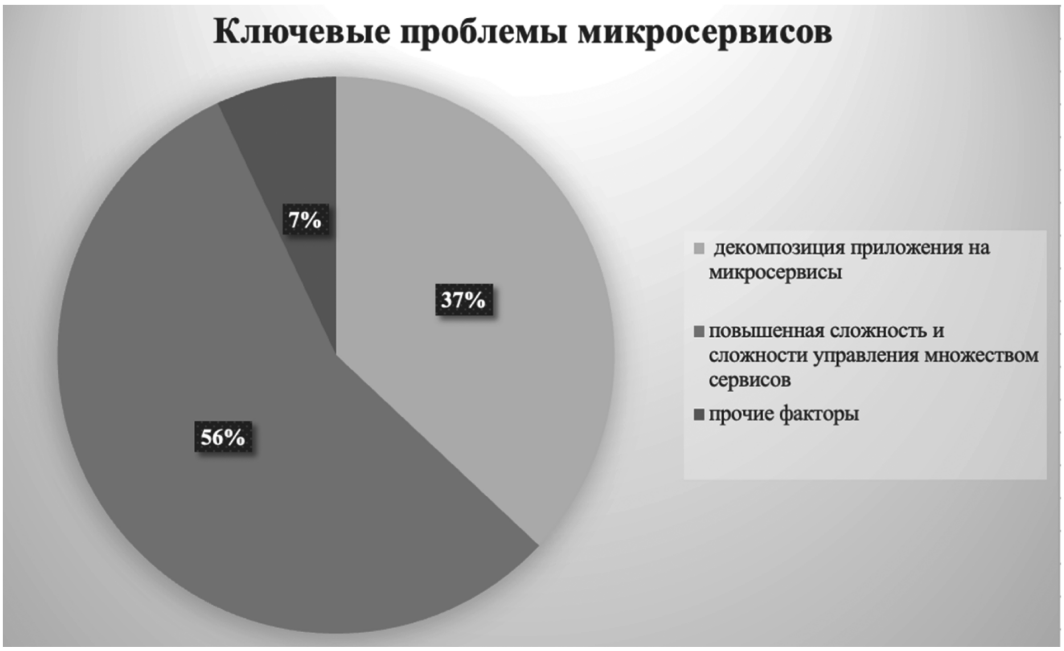

Рис. 2. Ключевые проблемы микросервисов

Источник: Составлено автором на основании [9–12]

Переходя от анализа проблем к рассмотрению целевой архитектуры, следует отметить, что при миграции от монолита к микросервисной модели на стеке Java Spring Boot структура системы формируется вокруг нескольких типов специализированных сервисов. Центральное место занимают доменные микросервисы, соответствующие отдельным bounded context и обеспечивающие автономное развитие бизнес-функций. Инфраструктурный слой включает сервис внешней конфигурации (Spring Cloud Config Server), сервис обнаружения (Eureka/Discovery Client) и API-шлюз (Spring Cloud Gateway), выполняющий функции маршрутизации, контроля доступа и консолидации интеграций. Дополнительными компонентами выступают сервисы аутентификации и авторизации на основе Spring Security, а также средства наблюдаемости, реализуемые с применением Spring Boot Actuator и механизмов распределённого трейсинга. В совокупности эти сервисы формируют рекомендуемую в официальной документации Spring целевую архитектуру, обеспечивающую изоляцию компонентов, управляемость и масштабируемость системы [13, 14].

Для устранения выявленных проблем необходима комплексная архитектурная и организационная трансформация системы. Ключевую роль играет полноценный DDD-анализ с корректным выделением bounded context, позволяющий перейти от технически ориентированного разбиения к доменной декомпозиции и сформировать устойчивые, слабо связанные сервисы. Оптимальная степень гранулярности достигается через анализ бизнес-процессов, нагрузочных профилей и при-

менение методик, таких как event-storming, что позволяет избежать как чрезмерного раздробления, так и сохранения укрупнённых доменов. Существенное значение имеют промежуточные архитектурные решения — модульный монолит, фасады, адаптеры и ACL-слой, обеспечивающие поэтапный характер миграции и снижение технологических рисков.

Кроме того, для обеспечения архитектурной целостности и устойчивости распределённой системы требуется изоляция данных по принципу database-per-service и стандартизированное управление интеграциями посредством API-gateway, политики версионирования API и контрактного тестирования. Повышение надёжности и скорости поставки обеспечивает внедрение CI/CD, контейнеризации, автоматизированных проверок безопасности и систем оркестрации. Завершает архитектурную модель развитая платформа наблюдаемости, включающая централизованное логирование, метрики, трейсинг и алертинг, а также планомерное сокращение технического долга. Переход к кросс-функциональным доменным командам обеспечивает согласованность организационной структуры с архитектурными принципами и устраняет разрывы ответственности.

Таким образом, проведённый анализ демонстрирует, что успешная трансформация legacy-монолита возможна только при применении целостного, согласованного подхода, включающего глубокий доменный анализ, корректную архитектурную декомпозицию, развитие инфраструктурных практик и адаптацию организационной структуры. Лишь при соблюдении этих условий микросервисная архитектура реализует свои ключевые преимущества — предсказуемость изменений, управляемость, масштабируемость и снижение зависимости от устаревших технологических решений, что подтверждает целесообразность её внедрения.

Заключение

Успешная трансформация legacy-монолита в микросервисную архитектуру требует не разовой модернизации, а комплексного подхода, включающего корректную DDD-декомпозицию, зрелые DevOps-практики и архитектурную изоляцию сервисов. Анализ научных и отраслевых источников показывает, что основные барьеры связаны с ошибками декомпозиции, слабым управлением зависимостями и недостаточной автоматизацией, а метрики DORA и архитектурные показатели обеспечивают объективную оценку успешности перехода. Поэтапная миграция, использование промежуточных архитектур и развитие наблюдаемости позволяют безопасно выносить функциональность, снижать технический долг и ускорять вывод изменений, делая микросервисную архитектуру на Java Spring эффективным инструментом повышения гибкости и устойчивости систем.

ЛИТЕРАТУРА

1. Зарипова В.М., Петрова И.Ю. Унаследованные информационные системы: проблемы и решения / В.М. Зарипова, И.Ю. Петрова // Инженерно-строительный вестник Прикаспия. — 2022. — № 2(40). — URL: https://cyberleninka.ru/article/n/unasledovannye-informatsionnye-sistemy-problemy-i-resheniya (дата обращения: 21.11.2025).
2. Михайлов А.А. Дипломная работа / А.А. Михайлов. — URL: https://elib.bsu.by/bitstream/123456789/272273/1/Дипломная_работа_МихайловАА_ПИ.pdf (дата обращения: 21.11.2025).
3. Шумилов М.И. Оптимизация высоконагруженных веб-проектов с использованием микросервисной архитектуры / М.И. Шумилов // Universum. Технические науки. — 2024. — № 11(128). — URL: https://7universum.com/ru/tech/archive/item/18560 (дата обращения: 21.11.2025).
4. Радостев Д.К., Никитина Е.Ю. Стратегия миграции программного кода из монолитной архитектуры в микросервисы / Д.К. Радостев, Е.Ю. Никитина // Вестник Пермского университета. Серия: Математика. Механика. Информатика. — 2021. — № 2(53). — URL: https://cyberleninka.ru/article/n/strategiya-migratsii-programmnogo-koda-iz-monolitnoy-arhitektury-v-mikroservisy (дата обращения: 21.11.2025).
5. Ольхов Д.А. Анализ подходов к управлению прикладным программным интерфейсом микросервисов в облачной среде / Д.А. Ольхов // Символ науки. — 2021. — № 3. — С. 26–32. — EDN ZWDSTR.
6. DORA Metrics. Four Keys Guide / DORA Metrics. — URL: https://dora.dev/guides/dora-metrics-four-keys (дата обращения: 21.11.2025).
7. Черников А.О., Заводчикова М.Г. Разработка метода оценки эффективности перехода от монолитной к сервис-ориентированной архитектуре для некоммерческих организаций / А.О. Черников, М.Г. Заводчикова // Экономика и бизнес: теория и практика. — 2025. — № 9(127). — URL: https://cyberleninka.ru/article/n/razrabotka-metoda-otsenki-effektivnosti-перехода-от-monolitnoy-k-servis-orientirovannoy-arhitekture-dlya-nekommercheskih (дата обращения: 21.11.2025).
8. Кабарухин А.П. Выгоды перехода от монолитной к микросервисной архитектуре приложения / А.П. Кабарухин // Проблемы современной науки и образования. — 2022. — № 1(170). — С. 18–23. — EDN FQHKDC.
9. O'Reilly Media. Microservices Adoption in 2020 / O'Reilly Media. — URL: https://www.oreilly.com/radar/microservices-adoption-in-2020 (дата обращения: 21.11.2025).
10. Microsoft. Azure Architecture Center: Domain Analysis / Microsoft. — URL: https://learn.microsoft.com/en-us/azure/architecture/microservices/model/domain-analysis (дата обращения: 21.11.2025).
11. OpsLevel. Challenges of Implementing Microservice Architecture / OpsLevel. — URL: https://www.opslevel.com/resources/challenges-of-implementing-microservice-architecture (дата обращения: 21.11.2025).
12. Camunda. DZone Microservices Research / Camunda. — URL: https://dzone.com/articles/new-research-shows-63-percent-of-enterprises-are-a (дата обращения: 21.11.2025).
13. Spring. Microservices [Электронный ресурс] / Spring. — URL: https://spring.io/microservices (дата обращения: 21.11.2025).
14. Spring Cloud Config. Reference Documentation [Электронный ресурс] / Spring. — URL: https://docs.spring.io/spring-cloud-config/docs/current/reference/html/ (дата обращения: 21.11.2025).

© Кононов Алексей Николаевич (aleksey.n.kononov@gazprombank.ru)
Журнал «Современная наука: актуальные проблемы теории и практики»