УДК 004.415.2
DOI 10.26118/2782-4586.2025.97.83.026

Мурлин Алексей Георгиевич
Кубанский государственный технологический университет
Анисимова Анастасия Романовна
Кубанский государственный технологический университет
Ведерников Георгий Валерьевич
Кубанский государственный технологический университет

Проблема выбора современного архитектурного решения для создания интерактивных приложений и веб-сайтов

Аннотация. Статья посвящена комплексному исследованию современных архитектурных решений для высоконагруженных интерактивных систем и веб-приложений. В условиях стремительного роста требований к производительности и масштабируемости ИТ-инфраструктуры проблема оптимального выбора архитектуры становится критически важной. В ходе исследований был проведен детальный анализ распространенных ошибок проектирования, характерных даже для крупных технологических компаний, которые впоследствии приводят к существенным финансовым и временным затратам на рефакторинг. В работе представлен подробный сравнительный анализ основных архитектурных парадигм: традиционных монолитных систем, микросервисной архитектуры, serverless-подхода и их гибридных комбинаций. Особый акцент сделан на оценке баланса между ключевыми характеристиками: производительностью, масштабируемостью, отказоустойчивостью и общей стоимостью владения. На основе изучения реальных кейсов и современных тенденций разработки предложена методика выбора архитектурного решения, учитывающая как текущие требования, так и перспективы роста системы. Результаты исследования включают практические рекомендации по проектированию устойчивых и эффективных архитектур, позволяющие избежать дорогостоящих изменений на поздних этапах жизненного цикла продукта. Материалы статьи представляют значительную ценность для архитекторов, технических руководителей и разработчиков, сталкивающихся с задачами построения надежных и масштабируемых решений в условиях быстро меняющихся бизнес-потребностей и технологического ландшафта

Ключевые слова: архитектура, микросервисы, монолит, serverless, масштабируемость, производительность, мобильные приложения, веб-приложения, веб-разработка, облачные технологии, контейнеризация, высоконагруженные системы, отказоустойчивость, бизнес-метрики

Murlin Alexey Georgievich
Kuban State Technological University
Anisimova Anastasia Romanovna
Kuban State Technological University
Vedernikov Georgiy Valerievich
Kuban State Technological University

The problem of choosing a modern architectural solution for creating interactive applications and websites

Annotation. The article is devoted to a comprehensive study of modern architectural solutions for highly loaded interactive systems and web applications. With the rapidly increasing demands on performance and scalability of the IT infrastructure, the problem of optimal

architecture choice is becoming critically important. The research conducted a detailed analysis of common design errors, typical even for large technology companies, which subsequently lead to significant financial and time costs for refactoring. The paper presents a detailed comparative analysis of the main architectural paradigms: traditional monolithic systems, microservice architecture, serverless approach and their hybrid combinations. Special emphasis is placed on assessing the balance between key characteristics: performance, scalability, fault tolerance, and total cost of ownership. Based on the study of real-world cases and current development trends, a methodology for choosing an architectural solution is proposed that takes into account both current requirements and the prospects for system growth. The results of the study include practical recommendations for designing sustainable and efficient architectures to avoid costly changes at late stages of the product lifecycle. The materials of the article are of significant value to architects, technical managers and developers who are faced with the challenges of building reliable and scalable solutions in a rapidly changing business needs and technological landscape

Key words: Architecture, microservices, monolith, serverless, scalability, performance, mobile applications, web applications, web development, cloud technologies, containerization, high-load systems, fault tolerance, business metrics

Введение. Современные тенденции цифровой трансформации предъявляют высокие требования к архитектуре программного обеспечения. Пользователи ожидают от приложений и веб-сайтов не только функциональности, но и высокой скорости работы, интуитивно понятного интерфейса, а также адаптивности под различные устройства. В условиях высокой конкуренции на рынке (более 5,7 млн приложений в Google Play и Apple App Store по данным [1]) качество архитектурных решений становится ключевым фактором успеха.

Актуальность исследования определяется современными вызовами цифровой трансформации. Жесткая конкуренция требует оптимизации производительности - 53% пользователей покидают сайт при загрузке более 3 секунд, что стимулирует внедрение микросервисов [5] и serverless-решений [8].

Изменения в поведении пользователей формируют новые требования к цифровым продуктам. Исследования показывают, что 88% посетителей не возвращаются после негативного опыта, а для 70% скорость работы критически влияет на решение о покупке, что объясняет популярность PWA и Edge Computing [3].

Бизнес-реалии требуют экономически эффективных решений. Облачные технологии сокращают затраты на инфраструктуру на 20-30% [4], а прогнозируемый переход 85% компаний на контейнеризацию к 2025 году подтверждает эту тенденцию.

Глобализация усиливает роль распределенных архитектур. CDN ускоряют загрузку на 50% [3], что критически важно, так как 100 мс задержки снижают конверсию на 7% по данным Akamai.

Архитектура программного обеспечения играет ключевую роль в обеспечении производительности и качества пользовательского опыта. Современные подходы, такие как микросервисная архитектура и serverless-вычисления, становятся все более востребованными - они позволяют эффективно распределять нагрузку и значительно сокращают время обработки запросов.

В условиях стремительного роста пользовательской аудитории особое значение приобретает масштабируемость архитектурных решений. Современные подходы позволяют не только горизонтально масштабировать отдельные сервисы без необходимости перестройки всей системы, но и оптимизировать использование вычислительных ресурсов за счет динамического распределения мощностей. Согласно данным Gartner [4], компании, внедрившие технологии контейнеризации (Kubernetes, Docker), смогли сократить время развертывания новых функций на 40% по сравнению с традиционными монолитными решениями.

Не менее важным аспектом является упрощение процессов разработки и поддержки. Современные архитектурные паттерны, включая компонентный подход во фронтенд-разработке (React, Vue.js, Angular) и API-ориентированную архитектуру (REST, GraphQL), способствуют повторному использованию кода и ускорению разработки. Как отмечают эксперты GitHub [9], проекты с модульной архитектурой требуют на 30% меньше усилий для поддержки по сравнению с монолитными системами.

Эти принципы модульности и гибкости находят свое развитие в микросервисной архитектуре. В данном подходе приложение разделяется на множество независимых сервисов, каждый из которых отвечает за определенную бизнес-функцию. Основное преимущество такого подхода - возможность независимой разработки и развертывания отдельных компонентов, что подтверждается успешным опытом Uber [7]. Однако важно учитывать и сложности, связанные с необходимостью использования специализированных инструментов оркестрации (Kubernetes, Istio) и дополнительными затратами на межсервисное взаимодействие.

Монолитная архитектура, несмотря на свою кажущуюся простоту, продолжает оставаться актуальной для определенных сценариев. Ее основные преимущества - простота развертывания и минимальные накладные расходы - делают этот подход идеальным выбором для стартапов и MVP-проектов, где критически важна скорость вывода продукта на рынок (см. рисунок 1).

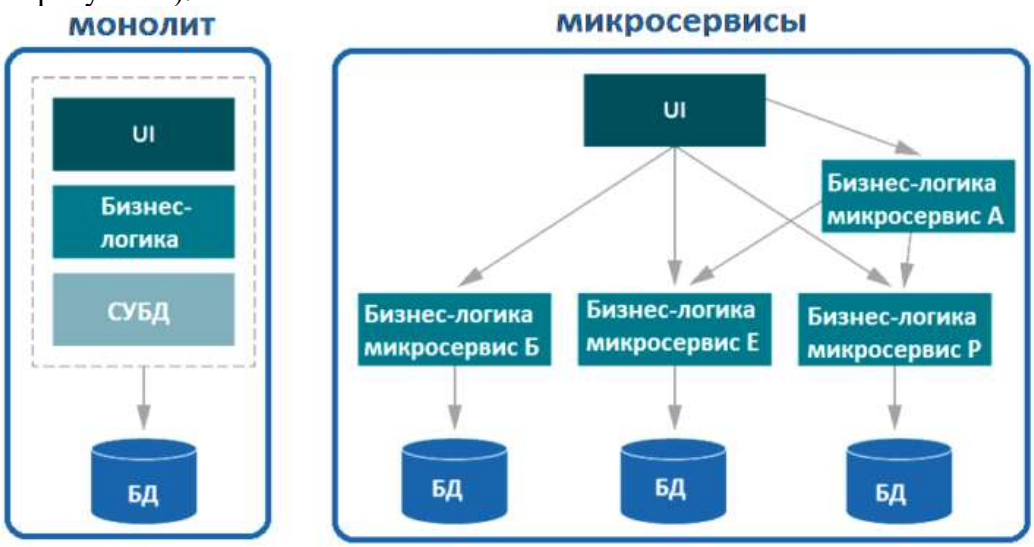

Рисунок 1 – Представление монолита и микросервисной архитектуры

Для более четкого понимания монолитной архитектуры рассмотрим сравнительный анализ двух популярных архитектурных систем, представленный в таблице 1.

Таблица 1 – СРАВНЕНИЕ МОНОЛИТНОЙ И МИКРОСЕРВИСНОЙ АРХИТЕКТУР

Характеристика	Монолит	Микросервисы
Сложность разработки	Простота разработки и отладки.	Сложнее из-за распределённой природы.
Масштабируемость	Сложно масштабировать	Легко масштабировать отдельные сервисы.
Отказоустойчивость	Низкая: сбой в одном модуле влияет на всё.	Высокая: сбой в одном сервисе не влияет на другие.
Технологический стек	Один стек для всего приложения.	Разные стеки для разных сервисов.
Стоимость	Низкие начальные затраты.	Высокие начальные затраты на инфраструктуру.
Поддержка	Проще поддерживать на ранних этапах.	Сложнее из-за распределённой природы.

Serverless-архитектура открывает новые возможности для разработчиков, позволяя сосредоточиться на бизнес-логике, а не на управлении инфраструктурой. Как демонстрирует опыт Slack [8], такой подход особенно эффективен для обработки событий в реальном времени. Однако следует учитывать ограничения, связанные с "холодным стартом" функций и максимальным временем их выполнения.

Использование CDN (Content Delivery Network) стало стандартом для обеспечения быстрой доставки контента пользователям по всему миру. Опыт Spotify [10] показывает, что применение CDN позволяет не только сократить время загрузки медиафайлов на 50%, но и значительно повысить отказоустойчивость системы. Однако важно понимать, что для динамического контента эффективность CDN может быть ограничена.

Таким образом, рассмотренные архитектурные решения позволяют сделать выводы: монолит — просто, но не масштабируется; микросервисы — гибко, но сложно; serverless — быстро и экономично, но с ограничениями; CDN — ускоряет доставку контента, но требует затрат.

В целях умения выбрать правильное решение, рассмотрим пример №1 современного архитектурного подхода для создания интерактивных приложений и веб-сайтов на рисунке 2.

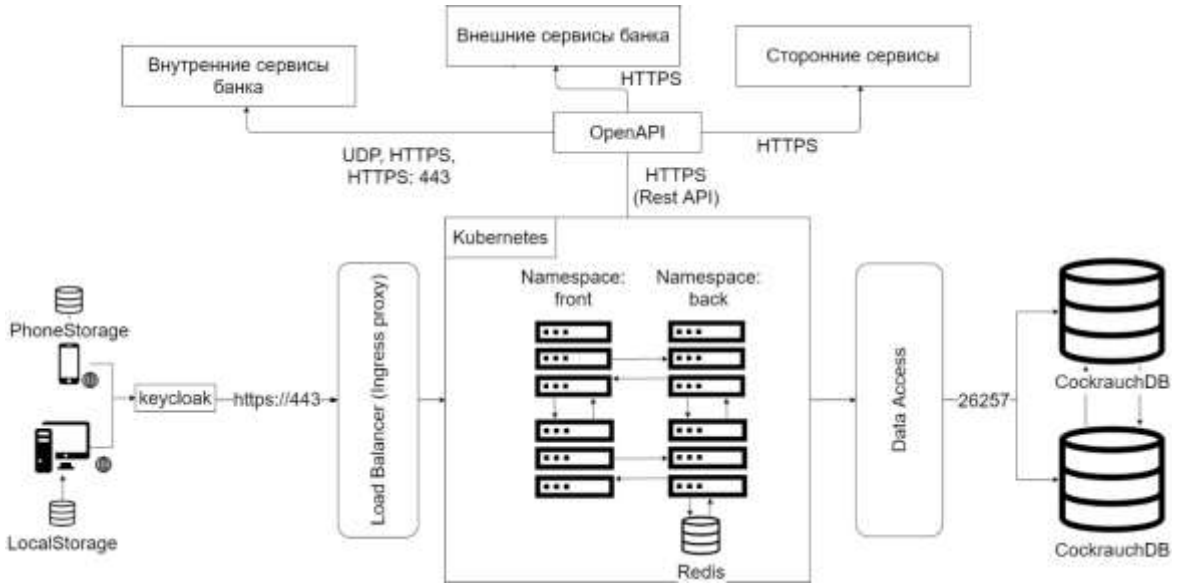

Рисунок 2 – Пример №1 актуальной архитектурной схемы

В современной разработке программного обеспечения архитектура системы играет ключевую роль в обеспечении производительности, масштабируемости и надежности. Можно выделить несколько ключевых компонентов и технологий, которые формируют основу распределенной системы.

1. Фронтенд (Frontend) системы отвечает за взаимодействие с пользователем и построена с использованием современных технологий: Vue.js.

JavaScript-фреймворк, используемый для создания динамических и отзывчивых пользовательских интерфейсов. Vue.js известен своей простотой интеграции и высокой производительностью.

Admin-panel JS Vue - это административная панель, также разработанная на Vue.js. Он может использоваться для управления контентом, настройками системы или мониторинга.

NGINX - веб-серверы, которые используются для обслуживания фронтенд-приложений. Nginx обеспечивает высокую производительность и надежность при обработке HTTP-запросов.

2. Бэкенд (Backend) отвечает за обработку данных, бизнес-логику и взаимодействие с базами данных.

PHP/Laravel - популярный фреймворк для веб-разработки на PHP. Laravel предоставляет удобные инструменты для создания сложных веб-приложений, включая маршрутизацию, ORM (Object-Relational Mapping) и поддержку API.

Nginx-backend - веб-сервер, используемый для проксирования запросов к бэкенд-сервисам. Это позволяет эффективно распределять нагрузку и обеспечивать безопасность.

Calc-backend PHP - специализированный сервис, отвечающий за выполнение расчетов или обработку данных. Он является частью микросервисной архитектуры, где каждый сервис выполняет определенную функцию.

Для управления данными и повышения производительности системы используются следующие технологии

REDIS - система управления базами данных в памяти, которая используется для кэширования и временного хранения данных. REDIS значительно ускоряет доступ к часто используемым данным.

Сервер БД - упоминание сервера базы данных указывает на использование реляционной CockrauchDB

3. Сетевая инфраструктура (сетевые компоненты) играет важную роль в обеспечении взаимодействия между различными компонентами системы:

UDP и Https - упоминание этих протоколов указывает на использование различных сетевых технологий для передачи данных. UDP (User Datagram Protocol) часто используется для быстрой передачи данных, где потеря пакетов не критична.

Https 443* - этот протокол указывает на использование защищенного соединения через HTTPS (порт 443), что обеспечивает шифрование данных и безопасность при передаче.

4. Использование пространств имен (namespaces) указывает на организацию системы в виде микросервисов (контейнеризованных приложений):

Front и Back - эти пространства имен связаны с различными сервисами, изолированными друг от друга. Это характерно для систем, использующих Kubernetes или Docker, где каждый сервис работает в своем окружении.

5. Дополнительные компоненты

В системе также присутствуют дополнительные компоненты, которые могут выполнять специализированные функции:

Leasing-nginx gateway - это шлюз, который обрабатывает или маршрутизирует запросы между различными частями системы.

S3, Graylog - эти компоненты являются внутренними инструментами или сервисами, используемыми хранения файлов и логирования

Vault - используется для безопасного хранения и управления такими данными, как API-ключи, пароли, сертификаты и другие конфиденциальные

Webor - используется для записи метрик, который в последующем используются для построения воронок менеджерами

Представленная архитектурная схема описывает современную распределенную систему, построенную на основе микросервисной архитектуры. Использование таких технологий, как Vue.js, PHP/Laravel, REDIS и nginx, позволяет создавать высокопроизводительные и масштабируемые приложения. Организация системы с использованием пространств имен и контейнеризации обеспечивает гибкость и изоляцию сервисов, что особенно важно для крупных проектов. В целом, такая архитектура подходит для сложных веб/МП-приложений, требующих высокой надежности и производительности.

В целях закрепления понимания микросервисной архитектуры рассмотрим еще один пример проектирования Информационной системы управления парковкой, архитектурная схема которого представлена на рисунке 3.

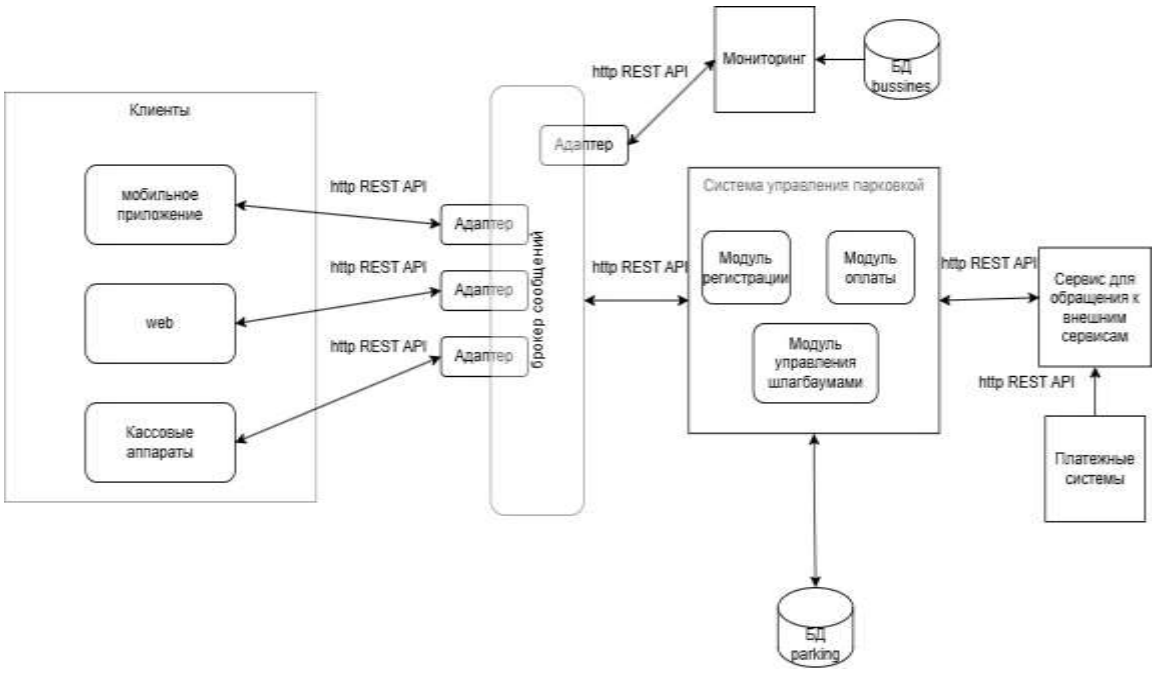

Рисунок 3 – Информационная система управления парковокой

Взаимодействие компонентов клиент-серверного приложения.

1. Сторона Frontend - популярные технологии, используемые frontend-разработчиками: JavaScript, React, Vue.js или Angular.

Взаимодействие с пользователем: Фронтэнд отвечает за интерфейс пользователя, обрабатывает действия пользователя. Например, во время регистрации пользователя в систему он вводит свои персональные данные в соответствующие поля (inputs), после чего нажимает на кнопку «Зарегистрироваться» и заполненные данные при помощи специальных контрактов REST API (HTTP-запросы) передаются к бэкенду.

2. Сторона Backend - популярные технологии: Node.js, Python (Django, Flask), C# и др.

Обработка запросов: Бэкенд принимает запросы от фронтэнда, обрабатывает их и выполняет необходимые действия. Например, получив данные пользователя, происходит процесс валидации: проверка на наличие уже зарегистрированного пользователя с полученными ФИО или логином пользователя, проверка на наличие данного пользователя в черном списке. При успешных проверках последующая отправка изменений/создание пользователя в смежные системы или базы данных.

Обычно используется REST API для взаимодействия между фронтэндом и бэкендом. API предоставляет конечные точки для получения и отправки данных.

Формат данных: Обычно используется JSON для передачи данных между клиентом и сервером.

Процесс взаимодействия компонентов.

1. Запуск приложения: Пользователь открывает веб-приложение в браузере или в мобильном приложении.

2. Фронтэнд загружает и отображает пользовательский интерфейс.

3. Взаимодействие пользователя: Пользователь взаимодействует с интерфейсом, например, заполняет форму и нажимает кнопку.

4. Отправка запроса: Фронтэнд формирует и отправляет запрос к API бэкенда. Брокер сообщений направляет запрос в нужный сервис.

5. Обработка запроса на бэкенде: Бэкенд принимает запрос, выполняет логику (например, валидацию, обработку данных), возможно, обращается к базе данных, сторонним сервисам.

6. Ответ API: Бэкенд формирует ответ и отправляет его обратно на фронтэнд.

7. Обновление интерфейса: Фронтэнд получает ответ, обновляет интерфейс в соответствии с полученными данными (например, показывает сообщение об успехе или ошибке).

При проектировании информационных систем крайне важно тщательно подбирать архитектурный стиль с учетом специфики проекта. Ключевыми факторами выбора являются сложность и масштаб проекта, изменчивость требований к функционалу и нагрузке, бюджетные ограничения, доступные ресурсы для разработки и поддержки, а также уровень квалификации команды и наличие необходимого инструментария.

Монолитная архитектура наиболее подходит для проектов малой и средней сложности со стабильными требованиями и ограниченными ресурсами. Все компоненты встраиваются в единое приложение, что обеспечивает простоту разработки и развертывания, однако создает трудности при масштабировании и обновлении системы. Как показывает практика использования в бухгалтерских приложениях, монолит проще в разработке и тестировании, поскольку все функции сосредоточены в одном месте без сетевых взаимодействий. Однако при росте нагрузки приходится масштабировать всё приложение целиком, а изменения в одной части могут повлиять на другие компоненты.

Для крупных сложных систем с динамично меняющимися требованиями оптимальным выбором становится микросервисная архитектура. Пример приложения "ЯндексGo" демонстрирует, как такая архитектура позволяет добавлять новые функции без перестройки всей системы. Каждый сервис является самостоятельным и может развертываться независимо, обеспечивая гибкость в выборе технологий и устойчивость к отказам отдельных компонентов [9]. Однако при этом значительно возрастает сложность управления и координации взаимодействия между сервисами, а сетевые задержки могут снижать общую производительность.

Serverless-архитектура особенно эффективна для задач с непостоянной нагрузкой, таких как обработка изображений через AWS Lambda. Этот подход позволяет оплачивать только фактически используемые ресурсы и избавляет разработчиков от управления серверной инфраструктурой. Вместе с тем возможны задержки при запуске функций после периодов простоя, сложности в отладке распределенных систем и ограничения на выполнение со стороны платформы.

Для высоконагруженных веб-приложений с глобальной аудиторией, подобных Netflix, критически важным становится применение CDN (Content Delivery Network). Эта технология распределяет статический контент по географически разнесенным узлам, ускоряя доступ к ресурсам и снижая нагрузку на основной сервер. Однако производительность в данном случае напрямую зависит от провайдера CDN, эксплуатационные расходы могут быть существенными, а механизм кэширования делает обновление контента не мгновенным.

Заключение

Принимая решение о выборе архитектурного стиля, следует учитывать, что микросервисная архитектура лучше подходит для больших распределенных команд и сложных приложений, тогда как монолит может быть более целесообразен для малых команд и проектов. Serverless-подход способен ускорить разработку для стартапов и эпизодических задач, а CDN остается оптимальным решением для глобальной доставки контента.

Список источников

1. Statista. Количество приложений в ведущих магазинах приложений [Электронный ресурс]. 2023. URL: https://www.statista.com (дата обращения: 07.04.2025).

2. Google. Core Web Vitals и исследования пользовательского опыта [Электронный ресурс]. 2023. URL: https://web.dev/vitals/ (дата обращения: 07.04.2025).

3. Akamai. Влияние задержек на электронную коммерцию [Электронный ресурс]. 2023. URL: https://www.akamai.com/resources (дата обращения: 07.04.2025).

4. Gartner. Будущее облачных и контейнерных технологий [Электронный ресурс]. 2023. URL: https://www.gartner.com (дата обращения: 07.04.2025).

5. Netflix Tech Blog. Масштабирование микросервисов [Электронный ресурс]. 2023. URL: https://netflixtechblog.com/ (дата обращения: 07.04.2025).

6. Gartner. Будущее облачно-нативных архитектур [Электронный ресурс]. 2023. URL: https://www.gartner.com (дата обращения: 07.04.2025).

7. Uber Engineering. Микросервисы в Uber [Электронный ресурс]. 2023. URL: https://eng.uber.com/ (дата обращения: 07.04.2025).

8. AWS. Примеры использования Serverless-технологий [Электронный ресурс]. 2023. URL: https://aws.amazon.com/solutions/case-studies/ (дата обращения: 07.04.2025).

9. Skillfactory. Микросервисная архитектура - что это такое простыми словами [Электронный ресурс]. 2024. URL: https://blog.skillfactory.ru/glossary/mikroservisnaya-arhitektura/ (дата обращения: 08.04.2025)

Сведения об авторах

Мурлин Алексей Георгиевич, кандидат технических наук, доцент кафедры информационных систем и программирования ФГБОУ ВО “Кубанский государственный технологический университет”, г. Краснодар, Россия

Анисимова Анастасия Романовна, студент кафедры информационных систем и программирования, ФГБОУ ВО “Кубанский государственный технологический университет”, г. Краснодар, Россия

Ведерников Георгий Валерьевич, студент кафедры информационных систем и программирования, ФГБОУ ВО “Кубанский государственный технологический университет”, г. Краснодар, Россия