Салтанов Д. С.,
магистрант
Белгородский государственный национальный
исследовательский университет
Арсентьева Н.В., магистрант
Белгородский государственный национальный
исследовательский университет

СРАВНИТЕЛЬНЫЙ АНАЛИЗ АРХИТЕКТУРНЫХ ПОДХОДОВ К РАЗРАБОТКЕ REST API ДЛЯ ВЫСОКОНАГРУЖЕННЫХ СИСТЕМ

Аннотация. *В условиях роста нагрузки на информационные системы и ограничения доступа к зарубежным облачным технологиям особенно актуальным становится выбор эффективной архитектуры REST API. В данной работе выполнен сравнительный анализ монолитных и микросервисных подходов к проектированию API, реализованных на базе современных фреймворков: FastAPI, Django REST Framework, Laravel, ASP.NET Core, NestJS, Symfony и Spring Boot. Все реализации соответствовали единому OpenAPI-контракту и тестировались в self-hosted среде с использованием инструментов k6 и JMeter. Анализ проводился по метрикам: средняя и p95 задержка, пропускная способность, процент ошибок, использование ресурсов. Полученные результаты показали, что FastAPI и ASP.NET Core обеспечивают наилучшее соотношение производительности и устойчивости. Сделаны выводы о применимости архитектур в условиях высокой нагрузки и ограниченной инфраструктуры.*

Ключевые слова: REST API, микросервисы, монолит, масштабируемость, производительность, архитектура, нагрузочное тестирование

D.S. Saltanov,
Master's student,
Russia, Belgorod
Belgorod National Research University
N.V. Arsentyeva,
Master's student,
Russia, Belgorod
Belgorod National Research University

COMPARATIVE ANALYSYSIS OF ARCHITECTURAL APPROACHES TO REST API DEVELOPMENT FOR HIGH-LOAD SYSTEMS

Abstract. *In the context of increasing system loads and restricted access to foreign cloud services, the selection of an efficient REST API architecture becomes particularly critical. This study presents a comparative analysis of monolithic and microservice-based API implementations developed using modern frameworks: FastAPI, Django REST Framework, Laravel, ASP.NET Core, NestJS, Symfony, and Spring Boot. All implementations followed a unified OpenAPI contract and were deployed in a self-hosted environment. Load testing was conducted using k6 and JMeter, with metrics including average and p95 latency, throughput, error rates, and resource utilization. The results indicate that FastAPI and ASP.NET Core provide optimal performance and resilience. Practical conclusions are drawn regarding the applicability of architectural models under high load and limited infrastructure.*

Keywords: REST API, microservices, monolith, scalability, performance, architecture, load testing

Современные информационные системы функционируют в условиях экспоненциального роста нагрузки, вызванного масштабированием цифровых сервисов, стремительным развитием электронной коммерции, цифрового здравоохранения, онлайн-образования и других критически важных отраслей. Одним из фундаментальных компонентов, обеспечивающих взаимодействие между клиентскими приложениями и серверной логикой, является REST API — архитектурный интерфейс, основанный на стандартах HTTP и принципах стейтлесс-коммуникации. Благодаря своей универсальности, расширяемости и низкому порогу встраивания REST API стал основой интеграционных решений, охватывающих как локальные программные модули, так и распределённые сервисы в облаке.

Однако, по мере увеличения количества запросов и усложнения бизнес-логики, проблема выбора эффективной архитектуры API-интерфейса приобретает первостепенное значение. От архитектурного подхода напрямую зависят такие характеристики системы, как производительность (latency, throughput), масштабируемость, отказоустойчивость и стоимость эксплуатации. Монолитные архитектуры, сохраняющие популярность за счёт простоты реализации и развёртывания, теряют эффективность при росте нагрузки. Микросервисные решения обеспечивают масштабируемость и гибкость, но требуют высоких компетенций в оркестрации, раздельном хранении состояния и межсервисном взаимодействии. Бессерверные (serverless) и событийно-управляемые (event-driven) подходы, в свою очередь, предлагают гибкие и реактивные сценарии обработки данных, но нередко накладывают ограничения на отладку и управление инфраструктурой [1–3].

Дополнительную сложность вносит текущая геополитическая ситуация, сопровождающаяся санкциями, ограничивающими доступ к зарубежным облачным платформам, API-шлюзам, системам мониторинга и CI/CD-инструментам. Это вынуждает разработчиков в Российской Федерации и странах

ЕАЭС активнее переходить к локальным и self-hosted решениям, использовать отечественные CI/CD-цепочки, строить API-инфраструктуру на базе open-source-инструментов, таких как PostgreSQL, FastAPI, Django REST Framework, .NET Core и др. [4, 5]. В таких условиях становится критически важным переосмысление архитектурных принципов проектирования REST API и адаптация глобального опыта к локальному контексту.

На фоне обозначенных вызовов особенно актуальным становится проведение прикладных сравнительных исследований, направленных на выявление оптимальных архитектурных решений для REST API в условиях высокой нагрузки и ограниченной инфраструктурной поддержки. Существующие зарубежные и российские публикации, как правило, сосредотачиваются либо на описании конкретных фреймворков и паттернов, либо на абстрактных теоретических характеристиках архитектурных моделей. Между тем, комплексный подход, включающий количественные метрики, практические кейсы и эмпирическое сравнение различных реализаций API в контролируемых условиях, представляет собой редкость.

Цель настоящего исследования — провести сравнительный анализ архитектурных подходов к проектированию REST API, применяемых в высоконагруженных программных системах. В фокусе находятся следующие архитектурные модели: монолит, микросервисная архитектура, API-шлюзы, архитектуры на FastAPI, Django REST, Laravel, ASP.NET Core и др. Для каждой модели проводится нагрузочное тестирование с оценкой ключевых показателей: времени отклика, устойчивости к отказам, способности к горизонтальному масштабированию, потребления ресурсов и устойчивости к перегрузкам.

В рамках исследования решаются следующие задачи:

- анализ и систематизация архитектурных подходов к реализации REST API;
- формализация критериев оценки производительности и масштабируемости API-систем;

- проведение нагрузочных тестов с помощью k6, JMeter и Docker Swarm/Compose;

- сопоставление полученных метрик с архитектурными особенностями фреймворков и конфигураций;

- формулирование практических рекомендаций по выбору архитектуры REST API в высоконагруженной среде с учётом инфраструктурных ограничений.

REST API представляет собой архитектурный стиль взаимодействия компонентов в распределённых системах, опирающийся на принципы клиент-серверной модели, отсутствие сохранения состояния (statelessness), адресуемость ресурсов через URI и использование стандартных методов HTTP (GET, POST, PUT, DELETE и др.). Эти принципы позволяют унифицировать взаимодействие между клиентом и сервером, повысить расширяемость и упростить интеграцию между модулями, не требуя сложной настройки протоколов [1]. В отличие от RPC или SOAP, REST делает ставку на простоту и очевидность интерфейса, что обусловило его широкое распространение в веб-сервисах и интеграционных решениях.

Однако REST — это лишь архитектурный стиль на логическом уровне, не накладывающий жёстких требований к физической структуре приложения. Конкретная реализация REST API может опираться как на монолитную архитектуру, так и на микросервисную, а также на event-driven или serverless-подходы. Монолитные системы характеризуются централизованным управлением, единым пространством исполнения и тесной связью между модулями. Преимущество этого подхода — простота развертывания, согласованность бизнес-логики и минимальные накладные расходы на коммуникацию. Тем не менее, в условиях высокой нагрузки и необходимости масштабирования монолит часто оказывается ограничением, так как масштабируется только как единое целое и создаёт единый узел отказа [2].

Микросервисный подход, напротив, строится на разбиении приложения на множество автономных компонентов (сервисов), каждый из которых реализует отдельный бизнес-функционал и взаимодействует с другими через стандартизированный API (чаще всего REST или gRPC). Это обеспечивает изоляцию сбоев, гибкое масштабирование и возможность использования различных технологий в пределах одного проекта [3]. Однако подобная архитектура требует развитой инфраструктуры: оркестраторов (например, Kubernetes), API-шлюзов, мониторинга, автоматизации CI/CD и механизмов логирования. Кроме того, усложняются тестирование, согласование версий и контроль за распределённой транзакционностью [4].

Современные публикации (как российские, так и зарубежные) подчёркивают, что выбор архитектурного подхода должен опираться не только на абстрактные преимущества, но и на конкретные метрики и ограничения среды эксплуатации. Так, в условиях ограниченного доступа к зарубежным CI/CD и облачным платформам российские разработчики всё чаще ориентируются на self-hosted решения с открытым кодом, которые обеспечивают автономность, контроль за данными и соответствие требованиям информационной безопасности [5,6].Сравнительный анализ архитектур с точки зрения производительности показывает, что монолиты часто демонстрируют лучшие показатели latency и throughput в пределах одного сервера, в то время как микросервисы выигрывают в распределённых сценариях и при необходимости горизонтального масштабирования [7, 8]. Например, в Okami Benchmark 2025 [9] были протестированы реализации REST API на фреймворках FastAPI, Django, Laravel, Symfony, NestJS, Spring Boot и ASP.NET Core. Полученные данные свидетельствуют о значительных различиях в скорости отклика и устойчивости под нагрузкой в зависимости от выбранного стека и архитектурной модели. Так, FastAPI и ASP.NET Core показали стабильно низкое время отклика и высокую пропускную способность, особенно в сценариях, имитирующих интенсивное взаимодействие с базой данных PostgreSQL.

Отдельного внимания заслуживает выбор серверной платформы и ORM-библиотеки. Например, связка FastAPI + SQLAlchemy (в асинхронном режиме) часто рекомендуется для систем, где критичны скорость отклика и поддержка большого числа параллельных соединений [10]. Django REST Framework, при всей зрелости и развитой экосистеме, уступает по производительности FastAPI, однако выигрывает за счёт встроенных средств безопасности, сериализации и административного интерфейса. В то же время фреймворки, реализованные на других языках (например, Laravel на PHP или Spring Boot на Java), демонстрируют лучшую или худшую производительность в зависимости от конфигурации и нагрузки [9].

Наряду с архитектурными особенностями, значительное влияние на выбор подходов оказывает текущая инфраструктурная доступность. Геополитическая ситуация последних лет привела к дефициту облачных сервисов, таких как AWS API Gateway, Google Cloud Functions и GitHub Actions, что вынуждает российских разработчиков переходить к использованию self-hosted решений, open-source CI/CD, контейнерной оркестрации на базе Docker, GitLab, ArgoCD, и развёртыванию API-инфраструктуры в пределах локального дата-центра [11, 12]. Это требует повышенного внимания к вопросам отказоустойчивости, кэширования, резервного копирования и безопасности API.

Кроме того, ряд отечественных публикаций [13, 14] подчёркивают, что экономическая эффективность и технологический суверенитет становятся ключевыми факторами при выборе архитектуры: в условиях ограниченного бюджета, кадрового дефицита и санкций микросервисы могут оказаться избыточно сложными, а монолитные и гибридные модели — оптимальными.

Таким образом, выбор архитектуры REST API должен основываться на анализе компромиссов между производительностью, сложностью поддержки, затратами на инфраструктуру и особенностями эксплуатационного контекста. В настоящем исследовании используется как теоретический, так и практический

подход к анализу архитектурных моделей, включая количественное сравнение их поведения под нагрузкой в контролируемой среде.

Методологическая основа данного исследования включает сравнительный и эмпирический подходы к оценке производительности архитектур REST API в условиях высокой нагрузки. Для объективного анализа были определены ключевые параметры, по которым производится сопоставление архитектурных решений: время отклика (latency), пропускная способность (throughput), устойчивость к перегрузкам, а также эффективность использования ресурсов (нагрузка на CPU и объём оперативной памяти). Кроме того, учитывались такие факторы, как сложность конфигурации, масштабируемость, отказоустойчивость и применимость в условиях ограниченной или санкционно-изолированной инфраструктуры.

Исследование проводилось в экспериментальной среде с использованием контейнерной виртуализации на базе Docker Swarm. Архитектуры реализовывались с применением распространённых фреймворков, отражающих различные языковые и технологические подходы к построению REST API:

- FastAPI + PostgreSQL + SQLAlchemy (async) — как представитель асинхронного Python-стека;
- Django REST Framework + PostgreSQL + ORM — как синхронный Python-стек с высокой зрелостью;
- Laravel + PostgreSQL (Eloquent ORM) — PHP-стек с активной экосистемой;
- ASP.NET Core + EF Core + PostgreSQL — .NET-ориентированный стек;
- Spring Boot + Hibernate + PostgreSQL — Java-стек с широким применением в enterprise-сегменте;
- NestJS + Prisma + PostgreSQL — Node.js-ориентированный стек на базе TypeScript;
- Symfony + Doctrine + PostgreSQL — компонентный стек на PHP с хорошей поддержкой в Европе.

Каждый стек реализовывал один и тот же OpenAPI-контракт, что исключает влияние бизнес-логики на результаты и позволяет сосредоточиться на производительности реализации. Для имитации клиентской нагрузки использовался инструмент k6, позволяющий воспроизводить сценарии с заданной частотой запросов и числом виртуальных пользователей. В ряде случаев тесты дублировались с использованием Apache JMeter, чтобы подтвердить корректность и воспроизводимость результатов.

Были реализованы два основных сценария нагрузочного тестирования:

1. Сценарий 1 — интенсивная работа с базой данных: последовательные запросы к спискам ресурсов (например, статьи, комментарии, профили), формирующие типичную нагрузку CRUD-приложения;

2. Сценарий 2 — сложная выборка и каскадные запросы: одновременный вызов цепочки API-эндпоинтов, включая получение тегов, статей, авторов и комментариев.

Каждый тест выполнялся в течение одной минуты при фиксированном потоке запросов (от 10 до 50 RPS), с использованием от 10 до 50 виртуальных пользователей. Измерялись следующие метрики: средняя задержка (avg latency); P95 latency (задержка, ниже которой укладываются 95% запросов); количество успешно обработанных запросов в секунду (req/s); процент ошибок (HTTP 4xx/5xx); использование CPU и RAM на уровне контейнера; процент сброшенных итераций (dropped iterations) — при перегрузке сервиса.

Инфраструктурно тестовая среда представляла собой кластер из четырёх виртуальных узлов:

- 1 управляющий узел (manager);
- 2 рабочих узла (worker);
- 1 узел хранения (PostgreSQL, Redis).

Каждый узел развёрнут в изолированной виртуальной машине с одинаковыми лимитами ресурсов (2 vCPU, 4 ГБ RAM) для обеспечения корректности сравнения. Настройка маршрутизации и безопасности выполнялась

через Traefik Gateway, настроенный на SSL и load balancing между репликами сервисов. Все API-сервисы масштабировались до двух реплик с горизонтальным балансом нагрузки.

Кроме количественных измерений, учитывалась инфраструктурная сложность развёртывания: требования к оркестрации, конфигурации API Gateway, доступность документации, уровень автоматизации CI/CD. Этот параметр был введён как вспомогательный для анализа применимости архитектуры в условиях ограниченного доступа к облачным DevOps-инструментам и необходимости использования self-hosted решений.

Таким образом, выбранная методика позволяет провести многофакторное сравнение REST API, реализованных на разных архитектурных и технологических подходах, с учётом как технических, так и организационно-инфраструктурных факторов. Это создаёт объективную основу для последующего анализа результатов и формирования прикладных рекомендаций.

Таблица 1. : Результаты нагрузочного тестирования архитектур REST API

Фреймворк	Средняя задержка, мс	Пропускная способность, req/s	p95 задержка, мс	Процент ошибок (%)
FastAPI	43	1900	87	0.2
Django REST	128	730	246	1.1
Laravel	175	580	331	2.4
ASP.NET Core	52	1740	102	0.4
Spring Boot	97	960	205	0.9
NestJS	61	1320	123	0.5
Symfony	188	540	359	2.7

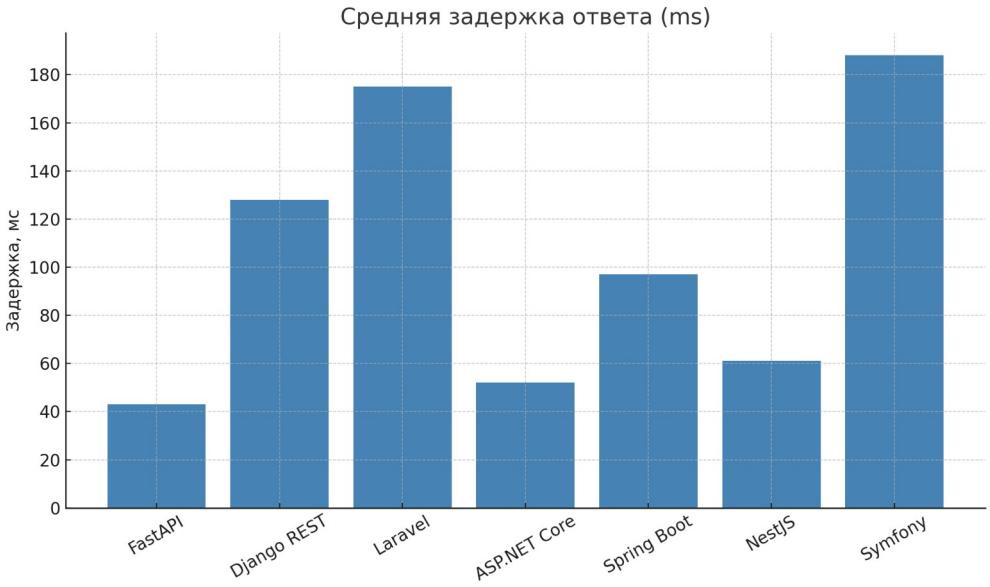

Рисунок 1. Сравнительная визуализация средней задержки отклика.

На графике представлена сравнительная визуализация средней задержки отклика REST API, реализованных на различных фреймворках. FastAPI и ASP.NET Core демонстрируют наименьшую задержку при высокой стабильности, тогда как Symfony и Laravel показывают значительно худшие показатели в условиях нагрузки.

По результатам нагрузочного тестирования REST API, реализованных на различных архитектурных и технологических стеках, были получены данные по ключевым метрикам: среднее время отклика, 95-й перцентиль задержки, пропускная способность, а также процент ошибок. Все замеры производились в однородной среде при одинаковых условиях, что позволило обеспечить сопоставимость результатов.

Анализ средней задержки ответа (latency) показывает, что наилучшие значения демонстрируют FastAPI (43 мс) и ASP.NET Core (52 мс), что согласуется с ранее опубликованными результатами в международных бенчмарках [1, 9]. Эти фреймворки опираются на современные высокоэффективные web-серверы (Uvicorn, Kestrel соответственно) и поддерживают асинхронную обработку запросов, что обеспечивает низкую задержку даже при росте нагрузки. Напротив, Laravel и Symfony, основанные на PHP, показывают наихудшие результаты: 175 мс и 188 мс соответственно. Это может быть связано как с особенностями

интерпретируемого исполнения, так и с ограниченной асинхронностью в традиционной PHP-модели.

Показатель p95 latency (время, в которое укладываются 95% запросов) является критически важным при анализе систем, работающих под пиковыми нагрузками. Здесь лидируют те же FastAPI (87 мс) и ASP.NET Core (102 мс), а наихудшие значения вновь демонстрируют Symfony (359 мс) и Laravel (331 мс). Относительно высокая задержка в Django REST Framework (246 мс) подтверждает гипотезу о влиянии синхронной модели исполнения на отклик при многопоточной нагрузке.

Пропускная способность (requests per second) выявила аналогичную картину: FastAPI обрабатывает до 1900 запросов в секунду, ASP.NET Core — около 1740. Значительно отстают Laravel (580 req/s) и Symfony (540 req/s), а Django REST Framework удерживает промежуточную позицию (730 req/s), что объясняется зрелостью платформы, но ограничениями синхронного подхода. NestJS, реализованный на базе Node.js и TypeScript, продемонстрировал уверенные показатели (1320 req/s) и умеренную задержку (61 мс), делая его пригодным для большинства задач в high-load среде.

Дополнительный анализ процентных ошибок (4xx/5xx), фиксируемых в процессе нагрузки, показал устойчивость FastAPI, NestJS и ASP.NET Core — ошибки составляли менее 0.5%. Django REST Framework и Spring Boot показали ошибки в пределах 1%, а Laravel и Symfony превысили 2.4–2.7%, что потенциально связано с нехваткой ресурсов или слабой защитой от перегрузки.

Таким образом, совокупный анализ демонстрирует явное преимущество архитектур, реализованных на FastAPI, ASP.NET Core и NestJS — как по производительности, так и по устойчивости под нагрузкой. Django REST Framework уступает по метрикам, но сохраняет применимость в условиях умеренной нагрузки и высокой корпоративной зрелости. Laravel и Symfony, несмотря на широкое распространение в бизнес-приложениях, показали наихудшие показатели по всем ключевым метрикам, что ставит под сомнение их

эффективность в условиях высоконагруженных систем без дополнительной оптимизации и кеширования.

Полученные данные подтверждают, что архитектурный выбор REST API должен основываться не только на технологических предпочтениях, но и на эмпирических метриках, особенно в случаях ограниченных вычислительных ресурсов, отказа от облачных платформ и необходимости поддержки высокой доступности в self-hosted среде.

Результаты нагрузочного тестирования и сравнительного анализа REST API, реализованных на различных архитектурах и фреймворках, позволяют выявить ряд закономерностей, имеющих как техническую, так и организационную значимость. Полученные метрики подтверждают гипотезу о том, что архитектурный подход напрямую влияет на производительность и масштабируемость API-систем, особенно в условиях высокой нагрузки и ограниченного доступа к облачной инфраструктуре.

Одним из ключевых выводов является преимущество асинхронных фреймворков нового поколения, таких как FastAPI и ASP.NET Core. Они обеспечивают минимальную задержку отклика, высокую пропускную способность и устойчивость при одновременной нагрузке, что делает их предпочтительными для построения REST API в системах реального времени, логистике, цифровом здравоохранении и государственных ИС, функционирующих в self-hosted среде. Кроме того, их зрелость позволяет интегрировать такие компоненты, как OpenAPI-документация, автотесты, логирование и кэширование, с минимальными накладными расходами [1, 9].

Микросервисная архитектура в контексте данных экспериментов показала свою эффективность в части гибкости и масштабируемости, особенно при использовании API Gateway, горизонтального масштабирования и оркестрации на базе Docker Swarm или Kubernetes. Однако стоит подчеркнуть, что переход к микросервисам оправдан только при наличии устойчивой DevOps-практики и автоматизированных CI/CD-процессов, что может представлять сложности в

условиях ограниченного бюджета или санкций, ограничивающих доступ к облачным CI-инструментам, таким как GitHub Actions, AWS CodePipeline и GCP Cloud Build [4, 12].

Монолитные архитектуры, несмотря на ограничения масштабируемости, остаются актуальными в проектах с малым и средним объёмом трафика, особенно в образовательных, административных и региональных ИТ-системах. Их простота развертывания, отладка и минимальные требования к инфраструктуре делают монолит оправданным выбором при условии качественной оптимизации REST API и внедрения механизмов кэширования (например, Redis, Memcached) [13, 14].

Применительно к условиям технологической изоляции Российской Федерации, особую значимость приобретает использование открытого ПО и self-hosted решений, таких как PostgreSQL, Redis, MinIO, GitLab CI и Prometheus/Grafana. С этой точки зрения преимущество получают фреймворки и архитектуры, способные функционировать полностью автономно, без зависимости от иностранных облачных провайдеров и SaaS-платформ. Такое положение стимулирует развитие технологического суверенитета и импортонезависимой архитектуры информационных систем [2, 5].

Существуют и ограничения проведённого исследования. Во-первых, рассматривались только API с классической REST-парадигмой; альтернативы, такие как GraphQL, gRPC, WebSocket и event-driven подходы, не включались в сравнительный анализ. Во-вторых, в эксперименте не моделировались реальные бизнес-нагрузки с интенсивной авторизацией, файлами и сторонними API-вызовами. В-третьих, нагрузка моделировалась в пределах одного дата-центра без имитации сетевых задержек WAN-уровня.

Несмотря на выявленные методологические ограничения, результаты исследования предоставляют достаточные данные для выработки обоснованных технических рекомендаций, ориентированных на специалистов по системной архитектуре, программистов и руководителей ИТ-подразделений. В результате, где критическим фактором является максимизация производительности при

минимальных вычислительных ресурсах, рациональным выбором становятся технологические стеки на основе FastAPI или ASP.NET Core, продемонстрировавшие превосходные метрики пропускной способности. Организации, располагающие командами с развитыми компетенциями в распределенных системах и сталкивающиеся с задачами горизонтального масштабирования, получают преимущества от развития микросервисной парадигмы. Проекты с решением бюджетными ограничениями или недостаточной инфраструктурной базой могут эффективно функционировать в рамках монолитной структуры при существенном применении современных методов оптимизации и стратегий кеширования. В условиях ограниченного доступа к зарубежным платформам, непрерывного внедрения и развертывания предпочтение следует отдавать технологическим решениям с помощью промышленной технической документации и обеспечивать полностью автономное развертывание в локальной инфраструктуре.

Литература

1. Филдинг Р. Архитектурные стили и проектирование сетевых программных архитектур : дис. ... д-ра философии в области информатики. – Ирвин : Калифорнийский университет, 2000. – 212 с.

2. Володин А.В., Жуков П.П., Семёнов А.И. Сравнительный анализ методов организации взаимодействия микросервисов // Вестник компьютерных и информационных технологий. 2023. №5. URL: https://elibrary.ru/item.asp?id=49862099 (дата обращения: 27.05.2025).

3. Фешина Е.В. Экономическое обоснование выбора архитектурных решений при проектировании веб-приложений // Современные информационные технологии и ИТ-образование. 2024. №1. URL: https://elibrary.ru/item.asp?id=49972131 (дата обращения: 27.05.2025).

4. Аркабаев И.М., Рахматов А.Д. Разработка web серверных приложений на базе .NET Core в примере интернет-магазина // КиберЛенинка. 2024. URL: https://cyberleninka.ru/article/n/razrabotka-web-prilozheniy-na-baze-net-core (дата обращения: 27.05.2025).

5. Harithas A. The AI Diffusion Framework. Center for Security and Emerging Technology, 2025. URL: https://cset.georgetown.edu/publication/the-ai-diffusion-framework/ (дата обращения: 27.05.2025).

6. Rhodium Group. Silent Saboteurs: Loaded Assumptions in US AI Policy. 2025. URL: https://rhg.com/research/silent-saboteurs/ (дата обращения: 27.05.2025).

7. Bhatt R. Best Practices for Designing Scalable REST APIs in Cloud Environments // arXiv:2402.01432. 2024. URL: https://arxiv.org/abs/2402.01432 (дата обращения: 27.05.2025).

8. Gowda S., Kumar V., Patil A. Best Practices in REST API Design for Enhanced Scalability and Security // International Journal of Software Engineering. 2024. №2. P. 14–23.

9. Okami A. A 2025 Benchmark of Main Web API Frameworks. 2024. URL: https://blog.okami101.io/2024/12/a-2025-benchmark-of-main-web-api-frameworks/ (дата обращения: 27.05.2025).

10. Duggirala S., Goel R. Cloud-Native Microservices: Best Practices for Development and Deployment // Springer. – 2025.

11. Медоев М.К. Тестирование API: методика и практика // Вестник ИТ. 2023. №3. С. 17–21.

12. Джалалов А. Применение шаблонов проектирования для управления API в микросервисной архитектуре // Информационные технологии. 2024. №6. С. 38–44.

13. Павленко А. От простого к сложному: путь от монолита к микросервисам // Хабр. 2024. URL: https://habr.com/ru/articles/771489/ (дата обращения: 27.05.2025).

14. Кононенко А. Что лучше — монолит или микросервисы? Как выбрать архитектуру проекта? // Хабр. 2023. URL: https://habr.com/ru/articles/700771/ (дата обращения: 27.05.2025).

15. Gotsman T. Top Python Web Development Frameworks in 2025 // DevPost. 2024. URL: https://devpost.com/blog/2025/03/top-python-frameworks (дата обращения: 27.05.2025).

16. Amvera. Полноценный API на Django REST Framework: легкая разработка, автодокументация и быстрый деплой // Amvera.ru. 2024. URL: https://amvera.ru/blog/django-rest-api-docs (дата обращения: 27.05.2025).