МАТЕМАТИЧЕСКОЕ И ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ ВЫЧИСЛИТЕЛЬНЫХ СИСТЕМ, КОМПЛЕКСОВ И КОМПЬЮТЕРНЫХ СЕТЕЙ/MATHEMATICAL SOFTWARE FOR COMPUTERS, COMPLEXES AND COMPUTER NETWORKS

DOI: https://doi.org/10.60797/IRJ.2025.158.27

МЕТОДЫ БАЛАНСИРОВКИ НАГРУЗКИ МЕЖДУ МИКРОСЕРВИСАМИ В ОБЛАЧНОЙ СРЕДЕ

Научная статья

Дубинин Р.С.[1], Темников Д.О.[2,] *
[1] АО «СОЛАР СЕКЬЮРИТИ», Москва, Российская Федерация
[2] EPAM Systems, Редмонд, Соединенные Штаты Америки

* Корреспондирующий автор (danil.temnikov[at]gmail.com)

Аннотация

Эффективная балансировка нагрузки в микросервисных облачных приложениях — ключевой фактор производительности (latency, throughput) и масштабируемости платформы. Статья систематизирует методы распределения трафика: от классических алгоритмов (Round Robin, Least Connections) до децентрализованной клиентской балансировки (Service Mesh), SDN-ориентированных схем и AI-контроллеров. Показаны преимущества и ограничения каждого подхода при динамичных и гетерогенных нагрузках. Авторский вклад заключается в разработке *типологии «шесть уровней балансировки»* (L4 → AI-контроллер) и *decision-tree-схемы*выбора стратегии под заданные KPI. На основе открытых бенчмарков и практических кейсов (Netflix Zuul 2, AWS ALB, Google Cloud L7 LB) продемонстрировано, что прогнозные модели LSTM + DQN снижают p99-latency до 30%, а SDN-контроллеры сокращают задержку в распределённых кластерах на 15–25%. Сделан вывод о необходимости многоуровневой архитектуры LB, объединяющей глобальное сетевое управление, локальные proxy-решения и автоматическое масштабирование, а также о перспективах green balancing и энерго-/стоимостной оптимизации.

Ключевые слова: микросервисы, облачные вычисления, балансировка нагрузки, производительность, масштабируемость, service mesh, SDN, машинное обучение, edge computing, serverless, security-aware load balancing, green balancing.

METHODS FOR BALANCING LOAD BETWEEN MICROSERVICES IN A CLOUD ENVIRONMENT

Research article

Dubinin R.S.[1], Temnikov D.O.[2,] *
[1] SOLAR SECURITY JSC, Moscow, Russian Federation
[2] EPAM Systems, Redmond, USA

* Corresponding author (danil.temnikov[at]gmail.com)

Abstract

Effective load balancing in microservice cloud applications is a key factor in platform performance (latency, throughput) and scalability. The article systematises traffic distribution methods: from classic algorithms (Round Robin, Least Connections) to decentralised client balancing (Service Mesh), SDN-oriented schemes, and AI controllers. The advantages and limitations of each approach under dynamic and heterogeneous loads are shown. The author's contribution lies in the development of *a typology of "six levels of balancing"* (L4 → AI controller) and *a decision tree scheme* for selecting a strategy for given KPIs. Based on open benchmarks and practical cases (Netflix Zuul 2, AWS ALB, Google Cloud L7 LB), it is demonstrated that LSTM + DQN predictive models reduce p99 latency by up to 30%, while SDN controllers reduce latency in distributed clusters by 15–25%. The conclusion is drawn about the necessity of a multi-level LB architecture that combines global network management, local proxy solutions, and automatic scaling, as well as the prospects for green balancing and energy/cost optimisation.

Keywords: microservices, cloud computing, load balancing, performance, scalability, service mesh, SDN, machine learning, edge computing, serverless, security-aware load balancing, green balancing.

Введение

Микросервисная архитектура превратила внутренние вызовы приложения в распределённый сетевой граф: каждый запрос должен быть направлен на один из множества горизонтально масштабируемых экземпляров. Балансировка нагрузки (LB) напрямую определяет четыре ключевых KPI облачного сервиса — latency, throughput, availability и scalability [1], [2], [3]. Когда распределение выполнено неправильно, задержка (p99-latency) растёт, пропускная способность падает, а инфраструктура используется неэффективно.

Динамичные всплески трафика — от *flash-sales* до социальных трендов — и гетерогенность контейнеров усложняют задачу: статические схемы перераспределения не успевают реагировать, перегружая одни узлы и оставляя другие без дела [2], [4]. Дополнительное усложнение вносит авто-масштабирование: число экземпляров меняется на лету, и LB-алгоритм обязан мгновенно учитывать новую топологию.

Традиционные подходы, созданные для монолитов и виртуальных машин, показали ограничения в контейнерных кластерах [5]. Поэтому индустрия сместилась к сервис-мешам, SDN-контроллерам и AI-управлению трафиком. Цель настоящего обзора — соотнести существующие методы LB с названными KPI и, на этой основе, предложить decision-tree-руководство и типологию «шесть уровней балансировки» — от L4-алгоритмов до AI-контроллеров.

Представленный материал агрегирует опубликованные бенчмарки и практические кейсы, чтобы показать, какой алгоритм выбирать при конкретных требованиях к производительности и масштабируемости.

Концепции балансировки нагрузки в микросервисной архитектуре

Микросервисы проектируются как автономные службы, взаимодействующие по сетевым API; поэтому вся логика приложения пронизывается внутренними вызовами — в крупном кластере их объём превосходит внешний трафик в десятки раз [6]. Каждая такая передача должна дойти до ресурса, способного обработать её без задержки; отсюда критическая роль балансировки нагрузки (LB). Именно она связывает сетевой слой с бизнес-уровнем, обеспечивая четыре целевых показателя (KPI) облачного приложения: latency, throughput, availability и scalability [1], [2], [3].

Простейшая, исторически сложившаяся модель — централизованный *server-side* балансировщик. Он принимает все входящие запросы, видит систему целиком и может применять сложные алгоритмы маршрутизации. Однако такая точка становится узким местом при пиковом трафике, а её отказ угрожает всей платформе. Распределённая альтернатива — *client-side* выбор адресата: каждый сервис-клиент, прежде чем обратиться к провайдеру, запрашивает у службы discovery актуальный список экземпляров и локально решает, кому передать вызов. Этот шаблон лег в основу сервис-мешей (Istio, Linkerd): рядом с контейнером разворачивается sidecar-прокси Envoy, который перехватывает вызовы, измеряет задержку, применяет retry и circuit-breaker-правила и тем самым перемещает LB-логику ближе к коду приложения [6]. Такая децентрализация снижает p99-latency за счёт исключения одного длинного скачка к централизованному шлюзу и повышает отказоустойчивость, поскольку сбой отдельного прокси не влияет на соседние сервисы.

Важнейшее инженерное решение — на каком уровне модели OSI организовать распределение. L4-балансировка работает лишь с TCP/UDP-метаданными, добавляя минимум задержки и выдерживая сотни тысяч RPS, но игнорирует семантику запросов. L7-методы анализируют HTTP-заголовки, URI и cookies, что позволяет, например, направлять все POST /checkout на отдельный пул машин или удерживать сессию пользователя на одном узле. Плата — рост CPU-потребления прокси и необходимость глубокой инспекции пакетов. Практика показывает, что гибридная схема (L4 → L7) чаще всего оптимальна, так как снаружи трафик расслаивается по транспортным признакам, а внутри кластера применяется контент-ориентированное правило.

Балансировка тесно интегрирована с Horizontal Pod Autoscaler. HPA добавляет или удаляет экземпляры в зависимости от загрузки, а LB мгновенно расширяет или сужает свой пул адресатов. Иначе свежесозданные pod’ы могли бы простаивать (*cold-start*), а старые — захлебываться под исторической привязкой клиентов. Одновременно LB отвечает за высокую доступность, когда при детекции сбоя health-чек автоматически исключает узел из ротации, и запросы плавно перенаправляются, не разрушая SLA.

Наконец, современные кластеры обслуживают разные аренды (multi-tenant). Здесь к LB добавляются задачи изоляции: ограничение rate-limit, mTLS-аутентификация между sidecar-прокси и защита от LB-poisoning — попыток выдать ложный сигнал перегрузки, чтобы вытеснить законный трафик. Всё это превращает балансировщик из сетевого распределителя в полноценный координационный сервис на стыке сети, оркестрации и безопасности.

Таблица 1 демонстрирует связь уровней балансировки с ключевыми KPI приложений.

Таблица 1 - Уровни балансировки и ключевые KPI приложений

DOI: https://doi.org/10.60797/IRJ.2025.158.27.1

Уровень балансировки	Сильные стороны (основной KPI)	Ограничения	Типовые сценарии
L4 (транспортный)	Минимальная задержка, высокий сквозной RPS (latency, throughput)	Нет content-routing; SPOF при центральном узле	Ingress TCP, gRPC-стримы
L7 (прикладной)	Гибкая маршрутизация, canary/blue-green-деплой → выше availability и controllable scalability	Дополнительный CPU-overhead, рост latency при пике	HTTP/REST API, GraphQL
Client-side / Service-mesh	Локальные решения снижают latency; встроенные retry & circuit-breaker повышают availability	Overhead sidecar-прокси; сложнее наблюдаемость	Кластеры > 50 сервисов, zero-trust / security-aware LB

Балансировка нагрузки таким образом становится основой производительности и надёжности микросервисного облака. Далее рассмотрены конкретные алгоритмы и технологии, реализующие описанные принципы.

Методы и алгоритмы балансировки нагрузки

Существует широкий спектр алгоритмов распределения нагрузки, применяемых в современных балансировщиках. Классические подходы опираются на сравнительно простые эвристики и не требуют глобальной информации о системе. Ниже приведены наиболее распространённые методы с указанием их ключевых достоинств и ограничений.

Первым алгоритм, Round Robin (циклический) — запросы последовательно направляются на каждый узел. Алгоритм прост и работает при однородности ресурсов, однако в гетерогенной инфраструктуре или при изменчивой нагрузке он перегружает менее производительные экземпляры [4]. Кроме того, присутствует эффект *cold-start*: новые pod’ы с пустой статистикой сразу получают полную долю трафика, что повышает задержку. В работе [11] показано, что в смешанном кластере производительность слабых узлов падает на 18–24% именно из-за циклического распределения.

Второй алгоритм, Weighted Round Robin (взвешенный циклический) — узлам назначаются статические веса, отражающие их относительные ресурсы. Метод компенсирует дисбаланс аппаратных характеристик, но не реагирует на фактическую загрузку; при длительной аффинности клиента к узлу (*stickiness*) возможна локальная перегрузка.

Следующий, третий, Least Connections (наименьшее число соединений) — балансировщик выбирает сервер с минимальным числом активных соединений. Способ адаптивен, однако подвержен *incast*: несколько параллельных клиентов одновременно видят один свободный узел и перенаправляют к нему трафик, что кратковременно увеличивает p99-latency. Алгоритм реагирует постфактум, выравнивая нагрузку с задержкой [4].

Далее рассмотрим Least Response Time (минимального времени отклика) — решение принимается по усреднённому RTT. Подход ориентирован на пользовательское восприятие, но чувствителен к сетевому джиттеру и заведомо реже выбирает холодные экземпляры, замедляя их прогрев.

Следующим вариантом будет консистентное хеширование (Consistent Hashing) — хеш-функция от идентификатора клиента гарантирует постоянное назначение узла и минимальные перестановки при изменении кластера. Ограничением остаётся возможный дисбаланс при неравномерном распределении хеш-пространства и закрепление нагрузки на подмножестве узлов (*stickiness*).

Последним рассматриваемым алгоритмом будет случайное распределение (Random) – запрос направляется на случайный сервер; вероятность перегрузки снижается лишь при большом числе обращений. Метод практического применения почти не имеет.

Для наглядного сравнения свойств основных алгоритмов приведена таблица 2.

Таблица 2 - Сравнение основных алгоритмов балансировки нагрузки

DOI: https://doi.org/10.60797/IRJ.2025.158.27.2

Алгоритм	Критерий выбора узла	Достоинства	Недостатки
Round Robin	Циклический обход списка узлов	Простой, равномерный при одинаковых узлах	Не учитывает нагрузку; не подходит при гетерогенных серверах
Weighted Round Robin	Цикл с учетом весовых коэффициентов	Учитывает различия в мощности узлов	Статические веса, без учета текущей нагрузки
Least Connections	Наименьшее число активных сессий	Реагирует на текущее состояние узлов	Не предвидит всплески; требуется учет соединений в реальном времени
Least Response Time	Минимальная задержка отклика	Ориентирован на качество (быстроту) обслуживания	Сложнее в реализации; нестабилен при изменчивых сетевых задержках
Consistent Hashing	Хеш от идентификатора клиента	Обеспечивает привязку сессий; минимизирует перераспределение при изменениях	Возможен дисбаланс при неравномерном хешировании; не учитывает текущую нагрузку
Random	Случайный выбор	Очень прост в реализации	Вероятностно может давать перегрузки; игнорирует любое состояние

В таблице суммированы критерии работы каждого алгоритма, их преимущества и ограничения. Простые циклические или случайные схемы уместны в однородных либо низконагруженных средах; при высоко-динамичном трафике они ведут к неравномерному использованию ресурсов [4]. Алгоритмы, учитывающие текущее состояние

(число соединений, RTT), адаптивнее, но требуют постоянного сбора метрик и всё ещё не прогнозируют будущую нагрузку.

На практике балансировщики комбинируют несколько методов. Так, в *Kubernetes* по-умолчанию применяется круговое разрешение через *iptables* (L4) совместно с периодическими *health-checks*: экземпляры, не прошедшие проверку, временно исключаются из ротации. Гибкие решения (*NGINX*, *HAProxy*, *Envoy*) позволяют администратору выбирать алгоритм — Round Robin, Least Connections, Least Response Time и т.д. — а также задавать дополнительные политики: IP-аффинность, геораспределение, ограничение RPS для «шумных» клиентов.

У классических алгоритмов есть и системные ограничения. Round Robin предполагает равенство узлов [4], что редко соблюдается в микросервисной инфраструктуре [11]. Least Connections подвержен эффекту *incast*: ряд фронтендов одновременно выбирает самый свободный узел и мгновенно его перегружает [6], [7]. Каждый локальный балансировщик (например, sidecar-прокси) видит лишь часть топологии, поэтому без координации возможна ситуация, когда трафик концентрируется на ограниченном подмножестве ресурсов, хотя в кластере есть свободные мощности.

Кроме того, реактивные алгоритмы реагируют с опозданием: узел, признанный наименее загруженным, быстро становится перегруженным, тогда как информация о его новом состоянии расходится по системе не мгновенно. Эти факты подчёркивают потребность в более интеллектуальных, проактивных методах, способных прогнозировать трафик и предотвращать дисбаланс — прежде всего в облачной среде, где нагрузка меняется резко и нелинейно [2], [4]. Современные подходы, решающие указанные проблемы, рассмотрены в следующем разделе.

Современные подходы к балансировке микросервисов

Эволюция микро- и облачных архитектур породила методы распределения нагрузки, которые учитывают динамику трафика, гетерогенность ресурсов и требования к задержкам. В этом разделе последовательно рассмотрены пять направлений: service-mesh, SDN-интеграция, AI/ML-подходы, edge и serverless-балансировка, а также security-aware LB.

Как упоминалось, один из трендов – отказ от единой точки входа в пользу децентрализованной балансировки. В современных облачных платформах (например, Kubernetes) нередко применяется т.н. service mesh — инфраструктурный слой, встроенный в кластер, который прозрачно перехватывает сетевые вызовы между микросервисами. Компоненты service mesh (например, Envoy в Istio) действуют как распределенные балансировщики, выполняя алгоритмы выбора узла прямо на стороне клиента. Такой подход улучшает масштабирование — каждая служба принимает решение локально — и повышает устойчивость, поскольку отказ одного прокси не выводит из строя всю систему. Для корректной работы необходима координация: прокси должны обмениваться сведениями о перегрузке и состоянии узлов.

Практическую реализацию показала Netflix при переходе на связку Zuul 2 + Eureka: после включения клиентской балансировки 99-й перцентиль задержек на внешнем шлюзе снизился почти втрое по сравнению с прежним централизованным циклическим алгоритмом [12]. Академический пример — система BLOC (Balancing Load with Overload Control), встроенная в Istio [6], [7]. BLOC использует обратную связь: если бэкенд сигнализирует кодом 503/429 или меткой overload, прокси снижает скорость отправки запросов (back-off) и перераспределяет поток. В испытаниях авторов размах задержек (p90 – p10) уменьшился в 2–4 раза, а p99-latency сократилась примерно вдвое относительно обычного Round Robin.

Помимо обратной связи BLOC включает динамический rate-limiting, что уменьшает вероятность эффекта incast — лавинообразной перегрузки одного узла. Похожие идеи реализуются в гибридных API-шлюзах: грубое региональное распределение выполняет входной шлюз, а fine-grained равновесие — sidecar-прокси каждого сервиса.

Таблица 3 - Характеристики современных стратегий балансировки нагрузки в микросервисах

DOI: https://doi.org/10.60797/IRJ.2025.158.27.3

Подход	Уровень внедрения	Механизм принятия решения	Требуемые данные/канал обратной связи	Ключевые преимущества	Ограничения	Типичные технологии
Service Mesh + BLOC	L7 (sidecar/Envoy)	Клиентский выбор узла с сигналами overload	Латентность узла, флаг 503/429, метрики очереди	Исключение «единой точки отказа», быстрая адаптивная разгрузка	Дополнительная задержка на прокси; сложная отладка политик	Istio + Envoy, Linkerd 2
SDN-контроллер	L3/L4 (центральный)	Правила маршрутизации в сетевых коммутаторах	Загрузка каналов, топология, требования QoS	Видит сеть целиком, оптимизирует межкластерный трафик	Требует SDN-инфраструктуры; риск SPOF контроллера	ONOS, OpenDaylight

Подход	Уровень внедрения	Механизм принятия решения	Требуемые данные/канал обратной связи	Ключевые преимущества	Ограничения	Типичные технологии
AI-driven LB	Многоуровневый (L7 + оркестратор)	Прогноз + динамическое изменение весов	Исторический трафик, телеметрия Pod/Node, метрики SLA	Проактивное распределение, снижение хвостовых задержек	Необходимость обучения/переподготовки моделей; накладные расходы	Prophecy (Meta), K8s + Prometheus + LSTM

Другая современная стратегия — использовать возможности SDN для централизованного управления трафиком микросервисов. В многоуровневых облачных и 5G-сетях SDN-контроллер отслеживает не только загрузку серверов, но и пропускную способность каналов, задержки между узлами и текущую топологию. Благодаря единой картине он динамически переписывает правила маршрутизации, направляя поток туда, где совокупная стоимость (latency + link load) минимальна.

Kalafatidis и Mamatas описали SDN-решение, в котором контроллер получает профили нагрузок микросервисов и, учитывая ёмкость каналов и близость данных, перераспределяет потоки так, чтобы выравнивать использование ЦП и сети; время отклика при этом снижалось на 18 – 25% [8], [9]. В промышленности сопоставимую функцию выполняют контроллеры ONOS и OVN: для 5G-сценариев они формируют набор slice-правил и удерживают сквозную задержку ниже 20 мс даже при 30% перегрузке радиоканала.

Проблема глобального распределения обостряется в мультикластерных развёртываниях. Простые DNS-решения (GeoDNS) направляют клиента в ближайший регион, но игнорируют внутреннюю нагрузку. Продвинутые механизмы, такие как Latency-aware Load Balancing для межкластерного service-mesh, в реальном времени измеряют RTT между регионами и перераспределяют запросы при изменении сетевых условий, снижая совокупную задержку до 15% относительно GeoDNS [10].

Однако одно из наиболее перспективных направлений — применение методов искусственного интеллекта для прогнозирования нагрузки и проактивного (упреждающего) распределения запросов. Традиционные алгоритмы, как отмечалось, работают реактивно. Если же балансировщик сможет предсказывать, как изменится поток запросов в ближайшие секунды или минуты, он сможет заранее перераспределить ресурсы или трафик, избегая перегрузок.

В последние 2–3 года появились работы, где используются методы machine learning и deep learning для этих целей [4]. Например, Bandarupalli применили рекуррентные нейронные сети (LSTM) для анализа паттернов входящего трафика микросервисов и динамического управления балансировкой нагрузки [4]. Модель обучается на исторических данных (трассах запросов в Kubernetes-кластере, метриках Prometheus) и умеет прогнозировать всплески или спады нагрузки с учетом сезонности и других факторов. На основе этого прогноза система адаптирует веса распределения или инициирует масштабирование сервисов.

В экспериментальном сравнении с классическими алгоритмами (Round Robin, Least Connections) такой подход показал существенный выигрыш: задержки сокращены до 25%, пропускная способность увеличена на 30% по сравнению с базовыми методами [4]. Это связано с тем, что нейросетевая модель учитывает скрытые зависимости в последовательности запросов и способна заранее перераспределить нагрузку до наступления пика. Кроме того, AI-алгоритм может оптимизировать балансировку комплексно, учитывая сразу несколько метрик (latency, CPU, и пр.) и выполняя более осмысленное решение, чем жёсткие правила. Ниже приведен упрощенный псевдокод подобного интеллектуального балансировщика:

Ниже приведён упрощённый фрагмент логики прогнозирующего балансировщика (основная часть цикла; подробный код опущен для компактности):

```
state ← collect_metrics()
traffic_forecast ← LSTM(state)
weights ← adjust(traffic_forecast)
chosen ← select_server(weights)
route(request, chosen)
reward ← –tail_latency
agent.learn(state, chosen, reward)   # DQN/DDPG
```

В этой схеме LSTM выполняет краткосрочный прогноз, а агент глубокого Q-обучения (DQN) или DDPG корректирует веса, получая награду за уменьшение хвостовых задержек. Эксперименты показывают, что такой агент обеспечивает до 97% своевременного выполнения запросов до истечения их дедлайна, превосходя классические эвристики.

Эти SDN- и AI-подходы формируют базу для многоуровневой системы: центральный контроллер грубо перераспределяет нагрузку между кластерами, а локальные сервис-меши с RL-агентами обеспечивают тонкую балансировку внутри регионов, учитывая как сетевые, так и вычислительные факторы.

Еще одна актуальная тенденция — тесная интеграция балансировщиков с менеджерами ресурсов. Например, в Docker Swarm встроенный круговой алгоритм первоначально не учитывал загрузку CPU и памяти хостов, что приводило к перегрузке отдельных узлов и простаиванию других [1]. Улучшенная версия анализирует метрики Prometheus и перераспределяет контейнеры с учётом текущего состояния; при испытаниях для Big Data-потока разброс загрузки процессора между узлами сократился вдвое [1].

Аналогично, в экосистеме Kubernetes контроллеры типа *Vertical Pod Autoscaler* или *Node-Problem Detector* могут временно приостановить входящий трафик к поду, близкому к исчерпанию ресурсов, что даёт горизонтальному автоскейлеру время запустить новые экземпляры. Комбинация «авто-масштабирование + балансировка» поддерживает QoS даже при резких пиках: трафик постепенно перевешивается на «тёплые» узлы, избегая как перегрузки старых, так и холодного старта новых [2].

В serverless- и edge-средах управление «холодным стартом» также переносится в балансировщик. Компонент *Activator* в Knative держит небольшой пул «тёплых» подов и через Istio-LB дозирует первую волну запросов; публикация команды Knative отмечает сокращение p95-latency с ≈600 мс до <180 мс knative.dev. В AWS Lambda глобальный балансировщик поддерживает механизм «burst concurrency»: функция может мгновенно получить до 500 параллельных вызовов (или 5 000 RPS) каждые 10 с, после чего нагрузка нарезается равномерно docs.aws.amazon.com.

Для многоарендных облаков к балансировке добавляются задачи изоляции и безопасности. Sidecar-прокси внедряют mTLS, а политики *rate-limiting* и *circuit-breaker* на уровне LB блокируют L7-DoS; Istio позволяет ограничить список сервисов, к которым может обращаться прокси, через объект Sidecar istio.io.

В совокупности перечисленные методы расширяют возможности балансировки. На внешнем уровне SDN-контроллер или latency-aware DNS распределяет пользователей по регионам, внутри региона client-side прокси реагирует на локальные метрики, а серверless-компоненты управляют холодным стартом функций. Глобальный контроллер обучается на телеметрии всех уровней и с помощью моделей LSTM + DQN прогнозирует нагрузку на минуты вперёд, меняя веса до появления пика. Подобные интеллектуальные менеджеры уже анонсированы Amazon и Google; они автоматически переключают политику с Round Robin на latency-aware при росте задержек.

Следует помнить о затратах: сбор телеметрии и переобучение моделей увеличивают накладные расходы, а неправильно настроенные петли обратной связи могут вызвать колебания нагрузки. Тем не менее тенденция такова, что балансировщик превращается в координационный сервис, тесно связанный с мониторингом, авто-масштабированием и политиками безопасности.

Заключение

Балансировка нагрузки между микросервисами в облачной среде — многогранная инженерная задача, определяющая *latency*, *throughput*, *availability* и *scalability* современного приложения. В статье прослежена эволюция решений: от классических алгоритмов (Round Robin, Least Connections) до децентрализованных service-mesh, SDN-ориентированных схем и AI-контроллеров. Показано, что универсального алгоритма не существует: каждый метод эффективен лишь при определённых предпосылках топологии и профиле трафика.

Практический анализ позволяет сформулировать пять кратких рекомендаций для инженеров-эксплуатантов:

1. Фиксируйте целевой KPI перед выбором алгоритма: для p99-latency ≤ 100 мс применяйте L7-балансировку с client-side retry, для потоков > 50 000 RPS — L4/ eBPF-решения.

2. Снимайте телеметрию в реальном масштабе времени (Prometheus ≥ 2.55, k6): без нее невозможно корректно настраивать веса и пороги.

3. Комбинируйте уровни: SDN-контроллер выполняет грубое межрегиональное распределение, а service-mesh обеспечивает тонкую балансировку внутри кластера.

4. Плавно вводите новые экземпляры (warm-up, gradual weight): это снижает эффект cold-start и выбросы хвостовых задержек.

5. Укрепляйте безопасность LB-контура: mTLS между прокси, rate-limiting и проверка подлинности сигналов overload предотвращают L7-DoS и LB-poisoning.

Интеграция с SDN расширяет оптимизационное пространство, позволяя учитывать состояние сети — важный фактор для георасределённых систем и предстоящих сетей 5G/6G. Использование прогнозных моделей LSTM и агентов DQN/DDPG доказывает, что *проактивное* управление трафиком способно снизить p99-latency на десятки процентов по сравнению с реактивными схемами; однако промышленное внедрение ИИ-контроллеров требует учёта накладных расходов на телеметрию, периодическое переобучение и формальные гарантии устойчивости.

Эффективная балансировка должна рассматриваться как иерархическая функция:

- инфраструктурный слой — проверенные алгоритмы распределения;
- слой оркестрации — сбор метрик и динамическая настройка параметров;
- глобальный слой управления — AI-контроллер, оптимизирующий картину целиком.

Будущие исследования, согласно выявленным тенденциям, сфокусируются на: адаптивной балансировке для гибридных и мультиоблачных сред; учёте энергокритерия (green balancing) при распределении трафика; безопасном выводе RL-агентов в production с возможностью быстрого отката.

Грамотное применение рассмотренных методов позволяет облачным платформам достигать высокой эластичности — обслуживать растущие пользовательские потоки без деградации SLA и при этом рационально расходовать ресурсы. Балансировка нагрузки остаётся одним из краеугольных элементов архитектуры масштабируемых и отказоустойчивых микросервисных систем.

Конфликт интересов

Не указан.

Рецензия

Нуриев М.Г., Казанский национальный исследовательский технический университет им. А. Н. Туполева – КАИ, Казань Российская Федерация
DOI: https://doi.org/10.60797/IRJ.2025.158.27.4

Conflict of Interest

None declared.

Review

Nuriev M.G., Kazan National Research Technical University named after A. N. Tupolev – KAI, Kazan Russian Federation
DOI: https://doi.org/10.60797/IRJ.2025.158.27.4

Список литературы / References

1. Singh N. Load balancing and service discovery using Docker Swarm for microservice based big data applications / N. Singh [et al.] // Journal of Cloud Computing. — 2023. — Vol. 12. — № 1. — P. 4.
2. Rabiu S. A cloud-based container microservices: A review on load-balancing and auto-scaling issues / S. Rabiu, C.H. Yong, S.M.S. Mohamad // International Journal of Data Science. — 2022. — Vol. 3. — № 2. — P. 80–92.
3. Dhiman G. Federated learning approach to protect healthcare data over big data scenario / G. Dhiman [et al.] // Sustainability. — 2022. — Vol. 14. — № 5. — P. 1–14.
4. Bandarupalli G. Enhancing Microservices Performance with AI-Based Load Balancing: A Deep Learning Perspective / G. Bandarupalli. — 2025. — P. 1–8.
5. Saxena D. Analysis of selected load balancing algorithms in containerized cloud environment for microservices / D. Saxena, B. Bhowmik // 2024 IEEE 4th International Conference on VLSI Systems, Architecture, Technology and Applications (VLSI SATA). — IEEE, 2024. — P. 1–6.
6. Bhattacharya R. Bloc: Balancing load with overload control in the microservices architecture / R. Bhattacharya, T. Wood // 2022 IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS). — IEEE, 2022. — P. 91–100.
7. Bhattacharya R. Load balancing for microservice service meshes / R. Bhattacharya // 2022 IEEE International Conference on Autonomic Computing and Self-Organizing Systems Companion (ACSOS-C). — IEEE, 2022. — P. 63–65.
8. Niu Y. Load balancing across microservices / Y. Niu, F. Liu, Z. Li // IEEE INFOCOM 2018-IEEE Conference on Computer Communications. — IEEE, 2018. — P. 198–206.
9. Kalafatidis S. Microservices-adaptive software-defined load balancing for 5G and beyond ecosystems / S. Kalafatidis, L. Mamatas // IEEE network. — 2022. — Vol. 36. — № 6. — P. 46–53.
10. Michaelis O. L3: Latency-aware Load Balancing in Multi-Cluster Service Mesh / O. Michaelis, S. Schmid, H. Mostafaei // Proceedings of the 25th International Middleware Conference. — 2024. — P. 49–61.
11. Елагин В.С. Функциональное назначение балансировщика нагрузки в облачных микросервисхных архитектурах / В.С. Елагин, В.Е. Николаев // Информационные технологии и телекоммуникации. — 2020. — Vol. 8. — № 1. — P. 67–75.
12. Zuul 2 : The Netflix Journey to Asynchronous, Non-Blocking Systems. — URL: https://netflixtechblog.com/zuul-2-the-netflix-journey-to-asynchronous-non-blocking-systems-45947377fb5c (accessed: 23.04.2025).
13. Announcing Knative v0.4 Release // Knative. — URL: https://knative.dev/blog/releases/announcing-knative-v0-4-release/ (accessed: 23.04.2025).
14. Understanding Lambda function scaling // AWS. — URL: https://docs.aws.amazon.com/lambda/latest/dg/lambda-concurrency.html (accessed: 23.04.2025).
15. Traffic Management // Istio. — URL: https://istio.io/latest/docs/concepts/traffic-management/ (accessed: 23.04.2025).

Список литературы на английском языке / References in English

1. Singh N. Load balancing and service discovery using Docker Swarm for microservice based big data applications / N. Singh [et al.] // Journal of Cloud Computing. — 2023. — Vol. 12. — № 1. — P. 4.
2. Rabiu S. A cloud-based container microservices: A review on load-balancing and auto-scaling issues / S. Rabiu, C.H. Yong, S.M.S. Mohamad // International Journal of Data Science. — 2022. — Vol. 3. — № 2. — P. 80–92.
3. Dhiman G. Federated learning approach to protect healthcare data over big data scenario / G. Dhiman [et al.] // Sustainability. — 2022. — Vol. 14. — № 5. — P. 1–14.
4. Bandarupalli G. Enhancing Microservices Performance with AI-Based Load Balancing: A Deep Learning Perspective / G. Bandarupalli. — 2025. — P. 1–8.
5. Saxena D. Analysis of selected load balancing algorithms in containerized cloud environment for microservices / D. Saxena, B. Bhowmik // 2024 IEEE 4th International Conference on VLSI Systems, Architecture, Technology and Applications (VLSI SATA). — IEEE, 2024. — P. 1–6.
6. Bhattacharya R. Bloc: Balancing load with overload control in the microservices architecture / R. Bhattacharya, T. Wood // 2022 IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS). — IEEE, 2022. — P. 91–100.
7. Bhattacharya R. Load balancing for microservice service meshes / R. Bhattacharya // 2022 IEEE International Conference on Autonomic Computing and Self-Organizing Systems Companion (ACSOS-C). — IEEE, 2022. — P. 63–65.
8. Niu Y. Load balancing across microservices / Y. Niu, F. Liu, Z. Li // IEEE INFOCOM 2018-IEEE Conference on Computer Communications. — IEEE, 2018. — P. 198–206.

9. Kalafatidis S. Microservices-adaptive software-defined load balancing for 5G and beyond ecosystems / S. Kalafatidis, L. Mamatas // IEEE network. — 2022. — Vol. 36. — № 6. — P. 46–53.

10. Michaelis O. L3: Latency-aware Load Balancing in Multi-Cluster Service Mesh / O. Michaelis, S. Schmid, H. Mostafaei // Proceedings of the 25th International Middleware Conference. — 2024. — P. 49–61.

11. Elagin V.S. Funkcional'noe naznachenie balansirovshhika nagruzki v oblachnyh mikroservishnyh arhitekturah [The functional purpose of load balancers in cloud microservice architectures] / V.S. Elagin, V.E. Nikolaev // Informacionnye tehnologii i telekommunikacii [Information Technology and Telecommunications]. — 2020. — Vol. 8. — № 1. — P. 67–75. [in Russian]

12. Zuul 2 : The Netflix Journey to Asynchronous, Non-Blocking Systems. — URL: https://netflixtechblog.com/zuul-2-the-netflix-journey-to-asynchronous-non-blocking-systems-45947377fb5c (accessed: 23.04.2025).

13. Announcing Knative v0.4 Release // Knative. — URL: https://knative.dev/blog/releases/announcing-knative-v0-4-release/ (accessed: 23.04.2025).

14. Understanding Lambda function scaling // AWS. — URL: https://docs.aws.amazon.com/lambda/latest/dg/lambda-concurrency.html (accessed: 23.04.2025).

15. Traffic Management // Istio. — URL: https://istio.io/latest/docs/concepts/traffic-management/ (accessed: 23.04.2025).