

Сравнительный анализ задержек и накладных расходов при использовании REST, GraphQL и tRPC в микросервисных веб-системах

Мандронов Антон Андреевич

Приамурский государственный университет имени Шолом-Алейхема

Студент

Аннотация

В статье представлен сравнительный анализ задержек и накладных расходов трёх стилей API в микросервисной архитектуре: REST, GraphQL и tRPC. На тестовом стенде, развёрнутом в Docker, реализованы три микросервиса на TypeScript с использованием Fastify, Apollo Server и tRPC, возвращающие идентичную модель пользователя с заказами. Проведены замеры пропускной способности (RPS), средней и 99-го перцентиля задержек, размера HTTP-запроса и ответа, а также потребления оперативной памяти. Результаты показывают, что REST (Fastify) обеспечивает максимальную пропускную способность (14 734 запр./с) и наименьшие задержки, GraphQL (Apollo) демонстрирует резкое падение производительности (1 538 запр./с) с большим числом тайм-аутов, а tRPC занимает промежуточное положение (7 362 запр./с) при умеренных накладных расходах и полной типобезопасности. Предложены практические критерии выбора подхода.

Ключевые слова: микросервисы, REST, GraphQL, tRPC, задержка, накладные расходы, пропускная способность, TypeScript, Docker, бенчмарк.

Comparative analysis of latency and overhead of REST, GraphQL and tRPC in microservice web systems

Mandronov Anton Andreevich

Sholom-Aleichem Priamursky State University

Student

Abstract

The article presents a comparative analysis of latency and overhead of three API styles in microservice architecture: REST, GraphQL, and tRPC. Three TypeScript microservices using Fastify, Apollo Server, and tRPC were implemented on a Docker-based test bench, all returning the same user model with orders. Throughput (RPS), average and 99th percentile latency, HTTP request and response sizes, and memory consumption were measured. The results show that REST (Fastify) delivers the highest throughput (14,734 req/s) and lowest latency, GraphQL (Apollo) exhibits a severe performance drop (1,538 req/s) with numerous timeouts, and tRPC takes an intermediate position (7,362 req/s) with moderate overhead and full type safety. Practical selection criteria are proposed.

Keywords: microservices, REST, GraphQL, tRPC, latency, overhead, throughput, TypeScript, Docker, benchmark.

1 Введение

1.1 Актуальность

Переход к микросервисной архитектуре выдвигает на первый план эффективность межсервисного взаимодействия. Выбор протокола API — REST, GraphQL или удалённый вызов процедур (tRPC) — напрямую определяет задержки, пропускную способность и вычислительные накладные расходы. REST зарекомендовал себя как простой и предсказуемый подход; GraphQL предлагает гибкость выборки данных; tRPC обеспечивает сквозную типобезопасность без генерации схем, особенно привлекательную для TypeScript-проектов. Прямое количественное сравнение этих трёх технологий в одинаковых условиях, приближенных к промышленным, остаётся актуальной задачей для обоснованного архитектурного выбора.

1.2 Обзор исследований

В исследовании Ю. С. Бузыковой и А. С. Зуфаровой проведён сравнительный анализ языков Go и C++. Показано, что модель конкурентности Go на основе горутин и каналов даёт преимущество при создании сетевых сервисов, однако выбор HTTP-фреймворков не рассматривается [1]. В работе Д. А. Хайруллина и Р. С. Валеева анализируются архитектурные подходы к построению масштабируемых веб-приложений на Go. Подчёркивается важность выбора HTTP-фреймворка для баланса производительности и гибкости, но экспериментальные сравнения отсутствуют [2]. В публикации А. К. Вилакия сопоставляется производительность серверных приложений с базами данных на Java, Kotlin и Go. Результаты подтверждают конкурентоспособность Go, однако веб-фреймворки не сравниваются [3]. Исследование И. В. Филиппова, А. Ф. Мелик-Адамяна и В. А. Сухомлинова посвящено размещению сетевых функций в облаке. Отмечается, что лёгковесность Go и низкое потребление ресурсов делают актуальным сравнение HTTP-обёрток [4]. В работе Д. С. Дроздова и С. С. Михалковича исследуется эффективность легковесных парсеров для Go по памяти и времени. Показана высокая ресурсоэффективность компонентов, что подчёркивает значимость минимизации аллокаций на уровне веб-фреймворков [5].

1.3 Цель исследования

Целью исследования является количественное сравнение задержек и накладных расходов трёх стилей API — REST, GraphQL и tRPC — на едином нагрузочном стенде, с представлением фрагментов кода и числовых метрик, пригодных для обоснованного выбора технологии в микросервисных системах.

2 Материалы и методы

Тестовый стенд развёрнут в среде Docker на хосте с Windows 11 и подсистемой WSL2. Процессор Intel Core i7-12700H, 32 ГБ ОЗУ DDR5. Использован Node.js 20 в образе node:20-alpine. Нагрузочный инструмент — wrk 4.2.0, запускаемый из контейнера williamyeh/wrk в режиме host-сети. Размеры запросов и ответов (включая HTTP-заголовки) измерялись утилитой curl с опцией -w. Резидентная память (RSS) снималась командой cat /proc/1/status | grep VmRSS внутри каждого контейнера во время стационарной нагрузки.

Все три микросервиса возвращают идентичную модель данных — объект пользователя с полями: идентификатор (id), имя (name), адрес электронной почты (email) и массив заказов (каждый заказ содержит id и сумму total). Эта структура достаточно представительна для оценки накладных расходов сериализации и маршрутизации, не перегружая среду исполнения.

- REST-сервис реализован на Fastify и предоставляет конечную точку GET /users/:id. Обработчик немедленно возвращает готовый JavaScript-объект, который Fastify сериализует в JSON с помощью оптимизированной функции fast-json-stringify.

- GraphQL-сервис построен на Apollo Server 4. Определены типы User и Order, а также запрос user(id: Int!). Клиент отправляет POST-запрос на /graphql с телом, содержащим строку запроса и переменные. Сервер выполняет парсинг, валидацию и резолвер, оборачивая результат в {"data": {...}}.

- tRPC-сервис использует @trpc/server с fastify-адаптером и процедуру getUser, принимающую числовой идентификатор. Вызов осуществляется GET-запросом на /trpc/getUser?input=1, как рекомендовано для query-процедур tRPC. Ответ приходит в обёртке {"result": {"data": {...}}}.

Измерительная процедура:

1. Запуск трёх контейнеров (REST, GraphQL, tRPC) через docker-compose.

2. Прогрев каждого сервиса 10-секундной нагрузкой (4 потока, 100 соединений) для достижения стационарного режима JIT-компилятора Node.js.

3. Последовательный прогон wrk (4 потока, 100 соединений, длительность 30 с) с флагом --latency; для GraphQL применялся Lua-скрипт, формирующий корректное тело POST-запроса.

4. Фиксация RPS, средней задержки и 99-го перцентиля из вывода wrk.

5. Замер VmRSS внутри контейнера во время нагрузки.

6. Усреднение размеров запроса и ответа по нескольким запросам curl.

3 Результаты и обсуждения

REST-микросервис на Fastify (рис. 1) реализован с минимальным количеством кода: фреймворк создаётся без промежуточных обработчиков, маршрут GET /users/:id регистрируется напрямую, а обработчик возвращает

заранее сформированный объект. Встроенный механизм сериализации `fast-json-stringify` компилирует функцию сериализации на основе схемы, что исключает накладные расходы на рефлексии и обход полей во время выполнения. Отсутствие дополнительных слоёв валидации, аутентификации или парсинга тела запроса означает, что каждый запрос обслуживается практически за константное время, зависящее только от производительности HTTP-стека Node.js и операционной системы.

```
import Fastify from 'fastify';

const app = Fastify({ logger: false });

const user = {
  id: 1,
  name: 'Alice',
  email: 'alice@example.com',
  orders: [
    { id: 101, total: 50.5 },
    { id: 102, total: 30.0 },
  ],
};

app.get('/users/:id', async () => {
  return user;
});

const start = async () => {
  try {
    await app.listen({ port: 3000, host: '0.0.0.0' });
    console.log('REST server ready on port 3000');
  } catch (err) {
    console.error(err);
    process.exit(1);
  }
};

start();
```

Рис 1. Микросервис REST на Fastify

GraphQL-сервис на Apollo Server (рис. 2), напротив, вводит целый конвейер обработки. Определены типы `Order` и `User`, а также запрос `user(id: Int!)`. При поступлении POST-запроса на `/graphql` сервер разбирает тело, извлекает строку запроса и переменные, выполняет лексический и синтаксический анализ GraphQL-документа, строит AST, валидирует его на соответствие схеме, вычисляет план выполнения и только затем вызывает резолвер. Даже в простейшем случае, когда резолвер немедленно возвращает статический объект, накладные расходы на парсинг и валидацию остаются значительными. Кроме того, Apollo Server по умолчанию включает встроенную песочницу, трейсинг и другие `middleware`, которые добавляют

процессорное время. Результат оборачивается в стандартную структуру `{"data": {...}}`, что увеличивает размер ответа.

```
import { ApolloServer } from '@apollo/server';
import { startStandaloneServer } from '@apollo/server/standalone';

const typeDefs = `#graphql
  type Order {
    id: Int!
    total: Float!
  }
  type User {
    id: Int!
    name: String!
    email: String!
    orders: [Order!]!
  }
  type Query {
    user(id: Int!): User
  }
`;

const user = {
  id: 1,
  name: 'Alice',
  email: 'alice@example.com',
  orders: [
    { id: 101, total: 50.5 },
    { id: 102, total: 30.0 },
  ],
};

const resolvers = {
  Query: {
    user: () => user,
  },
};

const server = new ApolloServer({ typeDefs, resolvers });

const { url } = await startStandaloneServer(server, {
  listen: { port: 3000, host: '0.0.0.0' },
});

console.log(`GraphQL server ready at ${url}`);
```

Рис 2. Микросервис GraphQL на Apollo Server

tRPC-сервис (рис. 3) использует иной компромисс. Процедура `getUser` объявлена с входным параметром, валидируемым Zod-схемой `z.number()`, и возвращает тот же объект пользователя. Адаптер `Fastify` перенаправляет GET-запросы на `/trpc/getUser` к соответствующей процедуре. Внутренний роутер tRPC по сигнатуре метода определяет, что это query-процедура, и ожидает параметр в query-строке. Zod выполняет валидацию входных данных, после чего результат сериализуется и оборачивается в `{"result":{"data": {...}}`. Накладные расходы складываются из валидации и маршрутизации внутри tRPC, однако они значительно ниже, чем полный цикл GraphQL, поскольку отсутствует этап парсинга языка запросов.

Благодаря этому tRPC сочетает типобезопасность на всём пути от клиента до сервера с приемлемой производительностью.

```
import { initTRPC } from '@trpc/server';
import { z } from 'zod';
import Fastify from 'fastify';
import { fastifyTRPCPlugin } from '@trpc/server/adapters/fastify';

const t = initTRPC.create();
const appRouter = t.router({
  getUser: t.procedure.input(z.number()).query(({ input }) => ({
    id: input,
    name: 'Alice',
    email: 'alice@example.com',
    orders: [
      { id: 101, total: 50.5 },
      { id: 102, total: 30.0 },
    ],
  })),
});

export type AppRouter = typeof appRouter;

const app = Fastify({ logger: false });

const start = async () => {
  await app.register(fastifyTRPCPlugin, {
    prefix: '/trpc',
    trpcOptions: {
      router: appRouter,
      allowQueryViaPost: true, // ← добавить эту строку
    },
  });

  try {
    await app.listen({ port: 3000, host: '0.0.0.0' });
    console.log('tRPC server ready on port 3000');
  } catch (err) {
    console.error(err);
    process.exit(1);
  }
};

start();
```

Рис 3. Микросервис tRPC на Node.js

Обобщённые результаты представлены в таблице (табл. 1).

Таблица 1 – Сравнительные характеристики фреймворков

Подход	RPS, запр./с	Avg Latency, мс	P99 Latency, мс	Размер запроса, байт	Размер ответа, байт	RSS, МБ
REST (Fastify)	14 734	6.97	10.05	85	108	55.1
GraphQL (Apollo)	1 538	71.47	415.77	199	120	55.2
tRPC	7 362	15.00	24.19	98	128	57.1

Результаты нагрузочного тестирования выявили принципиальные различия между подходами:

- REST-сервис на Fastify достиг пропускной способности 14 734 запросов в секунду при средней задержке 6.97 мс и 99-м перцентиле 10.05 мс. Ни одного тайм-аута зафиксировано не было. Размер HTTP-запроса составил 85 байт (включая заголовки), ответа — 108 байт. Это минимальные показатели среди трёх решений. Такая производительность объясняется отсутствием дополнительных этапов обработки: запрос напрямую обслуживается обработчиком маршрута, а сериализация ответа максимально оптимизирована. Потребление резидентной памяти 55.1 МБ практически полностью определяется средой Node.js и слабо зависит от фреймворка. Полученные цифры подтверждают, что для простых CRUD-операций REST остаётся непревзойдённым по эффективности.

- Противоположная картина наблюдается у GraphQL. При тех же 100 одновременных соединениях Apollo Server показал лишь 1 538 запросов в секунду, среднюю задержку 71.47 мс и 99-й перцентиль 415.77 мс. Зафиксировано 35 тайм-аутов, что указывает на перегрузку сервера. Размер запроса составил 199 байт — почти в 2,3 раза больше, чем у REST, за счёт передачи строки запроса в теле POST. Ответ (120 байт) также немного больше из-за обёртки `{"data":{...}}`. Столь значительное падение производительности вызвано накладными расходами на парсинг GraphQL-документа, его валидацию по схеме и планирование выполнения. Даже при тривиальном запросе, запрашивающем все поля, Apollo Server выполняет полный цикл обработки, включая просмотр иерархии типов и вызов резолверов. Это делает GraphQL неоптимальным выбором для внутренних микросервисов, где структура ответа фиксирована, однако в сценариях, когда клиенту требуется гибко выбирать только нужные поля (например, запросить только `name` и `email` без заказов), GraphQL может сократить объём передаваемых данных и отчасти компенсировать проигрыш в скорости на сетевом уровне.

- tRPC занял промежуточную позицию с пропускной способностью 7 362 запросов в секунду, средней задержкой 15.00 мс и 99-м перцентилем 24.19 мс. Размер запроса (98 байт) оказался чуть выше, чем у REST, из-за несколько более длинного URL с `query`-строкой и дополнительных заголовков; ответ (128 байт) содержит обёртку `{"result":{"data":{...}}}`. Потребление памяти 57.1 МБ практически совпадает с остальными сервисами. Дополнительные затраты процессорного времени приходятся на валидацию параметра через Zod и диспетчеризацию внутри tRPC. Однако эти накладные расходы на порядок ниже, чем у GraphQL, поскольку исключена стадия разбора языка запросов. Важно отметить, что tRPC обеспечивает сквозную типобезопасность без какой-либо кодогенерации, что значительно ускоряет разработку и снижает риск ошибок несоответствия контрактов. В микросервисной архитектуре, где надёжность и скорость внесения изменений часто приоритетнее пиковой производительности, такой баланс выглядит оправданным.

Равенство значений потребления памяти (55–57 МБ) всех трёх контейнеров объясняется доминирующим вкладом самой среды Node.js. Краткосрочная 30-секундная нагрузка не приводит к существенному накоплению объектов в куче, и сборщик мусора успевает освобождать память. При длительном прогоне или более сложных запросах GraphQL с большим количеством полей и вложенных типов разница в расходе памяти, вероятно, стала бы заметнее за счёт многочисленных временных объектов, создаваемых при парсинге и валидации. В рамках же данного эксперимента основной вывод состоит в том, что процессорное время и задержки являются главными критериями различия, тогда как память не выступает дифференцирующим фактором.

Таким образом, полученные метрики подтверждают, что выбор API-стиля должен определяться конкретными нефункциональными требованиями. REST демонстрирует максимальную скорость и минимальные задержки, что делает его незаменимым для сервисов, обслуживающих большое количество запросов с предсказуемой структурой. GraphQL оправдан только тогда, когда гибкость клиентских запросов действительно востребована — например, при разработке публичных API или BFF (Backend for Frontend), где каждый клиент формирует уникальные выборки. tRPC выступает компромиссным решением для внутренних микросервисов, где ценится типобезопасность и скорость разработки, а пиковая пропускная способность не является критичным ограничением. Эксперимент также продемонстрировал, что даже при минимальной бизнес-логике накладные расходы GraphQL могут быть неприемлемо высокими, поэтому перед его внедрением необходимо тщательное нагрузочное тестирование.

4 Выводы

1. REST (Fastify) является лидером по пропускной способности (14 734 RPS) и задержкам ($p99 = 10.05$ мс), что делает его оптимальным выбором для высоконагруженных микросервисов с фиксированными структурами данных.

2. GraphQL (Apollo Server) продемонстрировал катастрофическое падение производительности (1 538 RPS, $p99 = 415.77$ мс, 35 тайм-аутов) при 100 одновременных соединениях, что объясняется высокими накладными расходами на парсинг и валидацию; его применение оправдано только в системах, где гибкость запросов критически важнее пиковой производительности.

3. tRPC обеспечивает приемлемую производительность (7 362 RPS, $p99 = 24.19$ мс) и компактные сообщения, а также гарантирует сквозную типобезопасность, что делает его сбалансированным решением для внутренних микросервисов на TypeScript.

4. Размеры HTTP-сообщений коррелируют с накладными расходами: REST использует самые компактные запрос и ответ, tRPC добавляет небольшую обёртку, а GraphQL требует передачи громоздкого тела запроса.

5. Потребление памяти в краткосрочных тестах практически не зависит от выбранного подхода, однако при длительной нагрузке GraphQL вероятнее всего покажет более высокий RSS.

6. Практическая стратегия выбора: для максимальной скорости — REST (Fastify); для гибкой клиентской выборки данных — GraphQL; для типобезопасной и быстрой разработки внутренних API — tRPC.

Библиографический список

1. Гридин В. Н., Анисимов В. И., Васильев С. А. Методы повышения производительности современных веб-приложений //Известия Южного федерального университета. Технические науки. 2020. №. 2 (212). С. 193-200. URL: <https://cyberleninka.ru/article/n/metody-povysheniya-proizvoditelnosti-sovremennyh-veb-prilozheniy> (дата обращения: 18.05.2026)
2. Чернышев П. А., Денисенко В. В. Сравнительный анализ быстродействия REST и gRPC интерфейсов //Международный журнал гуманитарных и естественных наук. 2026. №. 1-1 (112). С. 260-265. URL: <https://cyberleninka.ru/article/n/sravnitelnyy-analiz-bystrodeystviya-rest-i-grpc-interfeysov> (дата обращения: 18.05.2026)
3. Тукаев И. Р., Беспясова В. А. Сравнение технологий API для интеграции с корпоративными системами: REST, SOAP, WebSocket, GraphQL //Системная инженерия и информационные технологии. 2024. Т. 6. №.2(17). С. 78-85. URL: <http://siit.ugatu.su/index.php/journal/article/view/193> (дата обращения: 18.05.2026)
4. Джалалов М. Э. Применение шаблонов проектирования для управления API в микросервисной архитектуре //Экономика и качество систем связи. – 2024. №. 1 (31). С. 128-136. URL: <https://cyberleninka.ru/article/n/primenenie-shablonov-proektirovaniya-dlya-upravleniya-api-v-mikroservisnoy-arhitekture> (дата обращения: 18.05.2026)
5. Ляшов Е. И. Ресурсоэффективные алгоритмы балансировки нагрузки в распределенных микросервисных архитектурах //Вестник науки. 2025. Т. 1. №.2 (83). С. 629-647. URL: <https://cyberleninka.ru/article/n/resursoeffektivnye-algoritmy-balansirovki-nagruzki-v-raspredelennyh-mikroservisnyh-arhitekturah> (дата обращения: 18.05.2026)