, Том 2026, № ?, 2026, стр. 1–6.

ПРОГРАММНАЯ ИНЖЕНЕРИЯ

Экспериментальное исследование адаптивной настройки паттернов устойчивости *timeout* и *retry* в микросервисной архитектуре на базе Spring Cloud

А. С. Коровко, В. А. Пархоменко,

Санкт-Петербургский политехнический университет Петра Великого, Санкт-Петербург, Россия

Аннотация—В работе исследуется влияние адаптивной настройки параметров паттернов устойчивости *timeout* и *retry* на качество обслуживания микросервисного приложения в условиях контролируемых деградаций. Рассматривается подход, в котором поверх статической конфигурации Resilience4j работает внешний контур Adaptive Resilience Manager (ARM), изменяющий параметры по телеметрии Prometheus на уровне API Gateway. Основное внимание уделено экспериментальному сравнению статической и адаптивной конфигураций на стенде Spring Petclinic Microservices в сценариях с инъекцией задержек и транзиентных ошибок. Показано, что адаптивная настройка снижает долю ошибок HTTP 504 и HTTP 502/503 в диапазоне применимости метода, но сопровождается ростом хвостовой задержки успешных ответов и дополнительной ресурсной стоимостью повторных попыток. Также выявлены границы подхода в стресс-сценариях высокой нагрузки и тяжелой деградации.

КЛЮЧЕВЫЕ СЛОВА: микросервисная архитектура, устойчивость, Spring Cloud, Resilience4j, adaptive control, timeout, retry.

1. ВВЕДЕНИЕ

Микросервисный подход обеспечивает независимое масштабирование и быстрые циклы поставки, но повышает чувствительность системы к частичным отказам и сетевым деградациям [1, 2]. В инженерной практике устойчивость обычно обеспечивается паттернами *timeout*, *retry*, *circuit breaker*, которые конфигурируются статически. Такой подход плохо переносит нестационарные условия нагрузки: параметры, достаточные для «легкой» фазы, становятся неэффективными при росте латентности или транзиентных ошибок. Цель работы — экспериментально оценить, как адаптивная подстройка параметров *timeout* и *retry* влияет на наблюдаемые показатели качества обслуживания на уровне API Gateway по сравнению со статической конфигурацией Resilience4j.

2. ОБЗОР РАБОТ И РЕШЕНИЙ

В исследованиях микросервисной устойчивости показано, что выбор параметров паттернов напрямую влияет на компромисс между доступностью, задержкой и ресурсной стоимостью [3, 5]. Для *retry* подтвержден эффект снижения транзиентных ошибок при ограниченном числе повторов и контролируемой стратегии backoff; агрессивные повторы увеличивают нагрузку и ухудшают задержку [5]. Для *timeout* индустриальные рекомендации подчеркивают риск чрезмерного удержания ресурсов при завышенных значениях и риск ранних отказов при заниженных [13–15]. Сравнение подходов в экосистеме Spring Cloud показывает, что библиотечные решения наподобие Resilience4j удобны для внедрения, но в базовом виде ориентированы на статическую конфигурацию параметров [7].

Экспериментальная гипотеза настоящей работы заключается в том, что внешний контур управления, использующий телеметрию и применяющий изменения конфигурации в рантайме, позволяет уменьшить долю целевых ошибок (HTTP 504 для *timeout*, HTTP 502/503 для *retry*) без перехода к сервисной сетке и без модификации бизнес-логики доменных сервисов.

3. АРХИТЕКТУРА ПОДХОДА

Архитектура включает пять компонентов: тестовое приложение Spring Petclinic Microservices, API Gateway, Prometheus, Grafana и внешний управляющий контур ARM [8–10]. ARM периодически считывает агрегированные метрики по маршрутам, сравнивает их с целевыми диапазонами и изменяет параметры Resilience4j на стороне шлюза через административный API. Управление выполняется независимо по `routeId` и ограничивается эксплуатационными границами параметров (минимум/максимум, шаг изменения, интервалы стабилизации).

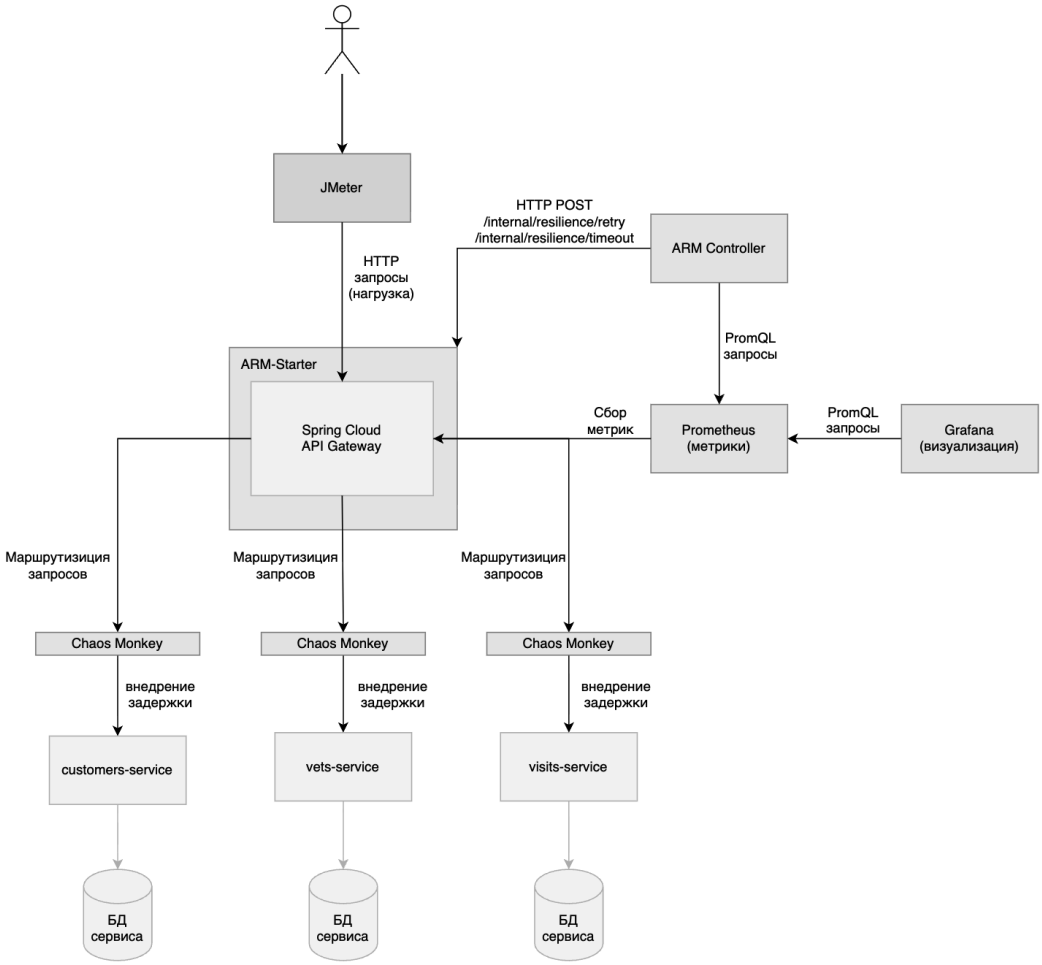

Рис. 1. Экспериментальный стенд на базе spring-petclinic-microservices (из ВКР, в оттенках серого)

4. ТЕСТИРОВАНИЕ

Корректность контура управления подтверждена модульными тестами правил принятия решений, BDD-сценариями ключевых пользовательских случаев и генеративными проверками устойчивости к некорректным данным мониторинга: за 10 минут обработано более 4 млн

случайно сгенерированных входов без сбоев. Покрытие критичных слоев (service/client) по JaCoCo превышает 95%, что снижает риск регрессий в логике изменения *timeout* и *retry*.

arm-controller

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
com.korovko.arm_controller.service		95%		91%	9	62	11	173	3	26	0	2
com.korovko.arm_controller.client		100%		n/a	0	9	0	32	0	9	0	4
Total	45 of 1,022	95%	6 of 72	91%	9	71	11	205	3	35	0	6

Рис. 2. Отчет покрытия JaCoCo (из ВКР, в оттенках серого)

5. МЕТОДИКА ЭКСПЕРИМЕНТОВ

Эксперименты выполнялись на стенде Spring Petclinic Microservices с формированием нагрузки в Apache JMeter и контролируемой инъекцией деградаций [11, 12]. Рассмотрены три группы сценариев: A1–A3 (адаптация *timeout*), B1–B3 (адаптация *retry*), C1–C2 (совместное управление). В сравнительных сценариях сопоставлялись Resilience4j static и ARM adaptive при одинаковых входных условиях; в валидационных и стресс-сценариях анализировалась корректность реакции ARM и границы применимости подхода. Основные метрики: доли ответов 2xx, 502, 503, 504, p99 (2xx) на gateway, а также ресурсные показатели.

6. РЕЗУЛЬТАТЫ ЭКСПЕРИМЕНТОВ

6.1. Сценарии A1–A3: адаптация timeout

В A1 (стационарная latency-деградация 700–1300 мс) адаптивная настройка снизила E_{504} с 33–35% (static) до 0–1.1% (adaptive), при этом p99(2xx) вырос с уровня около 1.1 с до 1.42 с. Результат подтверждает ожидаемый компромисс «меньше ранних отказов — выше хвостовая задержка успешных ответов».

Рис. 3. Эксперимент A2: static (слева) и adaptive (справа), в оттенках серого

В A2 (нестационарная латентность) показано, что фиксированный низкий *timeout* дает приемлемую задержку в «легкой» фазе, но приводит к 80–85% 504 в «тяжелой» фазе. Фиксированный высокий *timeout* почти устраняет 504, но повышает p99 до 1.25–1.30 с. ARM в «тяжелой» фазе снижает 504 с пика 28–30% до 0–1%, адаптируя параметр по ситуации.

В A3 (неоднородность по маршрутам) ARM независимо по `routeId` оставляет *timeout* без роста на «легком» `customers_route` и увеличивает его до верхней границы на `vets_route`/`visits_route`. При этом для тяжелых маршрутов сохраняется 55–60% 504 из-за выхода фактической латентности за установленный предел $\tau_{\max} = 1500$ мс.

6.2. Сценарии B1–B3: адаптация retry

В B1 (транзиентные 502/503 около 10%) адаптивный *retry* снизил $E_{502/503}$ с 9–11% до 0.3–0.8% и повысил долю 2xx с 89–91% до 99–100%. Цена улучшения — рост p99(2xx) с 5–10 мс до 50–70 мс из-за повторных попыток.

Рис. 4. Эксперимент B1: static без повторов (слева) и adaptive retry (справа), в оттенках серого

В B2 (низкая транзиентность 1%) ARM не «разгоняет» *maxAttempts* без устойчивого сигнала ухудшения: после переходного участка метрики стабилизируются вблизи цели. В B3 (нетранзиентные ошибки 404/500) показана граница применимости *retry*: при отсутствии ухудшения $E_{502/503}$ контур корректно не увеличивает число попыток.

6.3. Сценарии C1–C2: совместное управление

В C1 (одновременные latency и fault деградации) ARM уменьшает оба типа целевых ошибок: E_{504} снижается с 25–35% до 0–1%, $E_{502/503}$ — с 8–12% до 0–1%, а доля 2xx растет до 99–100%. При этом p99(2xx) стабилизируется около 1.42 с, отражая фактический хвост латентности зависимостей.

В C2 (стресс: 800 RPS, concurrency 1000, высокая латентность и транзиентность на части маршрутов) выявлены ограничения метода: на проблемных маршрутах доминируют 504 и растет p99, несмотря на достижение верхних границ параметров. Это подтверждает необходимость эксплуатационных ограничений и защитных правил для режима насыщения.

Рис. 5. Эксперимент C2: стресс-сценарий и границы применимости (в оттенках серого)

Таблица 1. Сводка ключевых количественных результатов

Сценарий	Условия	Эффект ARM (по сравнению со static, где применимо)	Компромисс / ограничение
A1	latency 700–1300 мс	E_{504}: 33–35% → 0–1.1%	p99(2xx): 1.05–1.15 с → 1.38–1.46 с
A2	нестационарная латентность	пик E_{504} в «тяжелой» фазе: 28–30% → 0–1% после адаптации	запаздывание обратного снижения *timeout* из-за стабилизации
A3	неоднородные маршруты	независимая подстройка по `routeId`	при $\tau_{\max} = 1500$ мс на тяжелых маршрутах остается 55–60% 504
B1	транзиентные 502/503 ≈10%	$E_{502/503}$: 9–11% → 0.3–0.8%; 2xx: 89–91% → 99–100%	p99(2xx): 5–10 мс → 50–70 мс
B2/B3	низкая транзиентность / нетранзиентные ошибки	отсутствие необоснованного роста *maxAttempts*	*retry* эффективно только для целевых транзиентных кодов
C1	совместная деградация latency+fault	E_{504} и $E_{502/503}$ снижаются до 0–1%; 2xx до 99–100%	p99(2xx): 1.38–1.46 с
C2	стресс: 800 RPS, conc. 1000	корректная работа контура до верхних границ параметров	в насыщении доминируют 504, p99 растет

ЗАКЛЮЧЕНИЕ

Экспериментально подтверждено, что адаптивная настройка *timeout* и *retry* на уровне API Gateway может существенно уменьшать долю целевых ошибок по сравнению со статической конфигурацией Resilience4j в контролируемых сценариях деградаций. При этом улучшение качества обслуживания достигается не бесплатно: рост успешности сопровождается увеличением хвостовой задержки и дополнительной ресурсной стоимостью. В стресс-режимах показаны границы применимости подхода, что требует явных эксплуатационных ограничений, стабилизации и правил безопасной работы в насыщении.

СПИСОК ЛИТЕРАТУРЫ

1. Microservices, Spring. https://spring.io/microservices
2. Podduturi S.M. Security and Performance Optimization in Microservices for Real-Time Data Systems, International Journal for Multidisciplinary Research, 2024.
3. Mendonca N.C., Aderaldo C.M., Camara J., Garlan D. Model-Based Analysis of Microservice Resiliency Patterns, 2020. https://ieeexplore.ieee.org/document/9101301
4. Gesvindr D., Davidek J., Buhnova B. Design of Scalable and Resilient Applications using Microservice Architecture in PaaS Cloud, 2019. https://dl.acm.org/doi/10.5220/0007842906190630
5. Aderaldo C.M., Mendonca N.C. How The Retry Pattern Impacts Application Performance: A Controlled Experiment, 2023. https://dl.acm.org/doi/10.1145/3613372.3613409
6. Qualitative and quantitative comparison of Spring Cloud and Kubernetes in migrating from a monolithic to a microservice architecture. https://link.springer.com/article/10.1007/s11761-023-00364-w
7. Resilience4j. https://resilience4j.readme.io/docs/getting-started
8. spring-petclinic-microservices. https://github.com/spring-petclinic/spring-petclinic-microservices
9. Prometheus. https://prometheus.io/
10. Grafana. https://grafana.com/
11. Chaos Monkey. https://github.com/Netflix/chaosmonkey
12. Apache JMeter. https://jmeter.apache.org/
13. Addressing Cascading Failures, Google SRE Book. https://sre.google/sre-book/addressing-cascading-failures/
14. Timeouts, retries, and backoff with jitter, AWS Builder's Library. https://aws.amazon.com/builders-library/timeouts-retries-and-backoff-with-jitter
15. AWS Well-Architected: Design interactions in distributed systems to withstand failures. https://wa.aws.amazon.com/wellarchitected/2020-07-02T19-33-23/wat.question.REL_5.en.html