













Институт
интеллектуальных кибернетических систем
Кафедра №22 «Кибернетика»

Направление подготовки 09.03.04 Программная инженерия

Пояснительная записка

к учебно-исследовательской работе студента на тему:

**Сравнительный анализ реализации микросервисной
архитектуры с использованием паттерна Circuit Breaker на
основе K3s и Istio.**

Группа	Б22-544	
Студент	 (подпись)	Писарев А. И. (ФИО)
Руководитель	 (подпись)	Ровнягин М.М. (ФИО)
Оценка руководителя	30 (0-30 баллов)	
Итоговая оценка		ECTS
	(0-100 баллов)	
Комиссия		
Председатель	 (подпись)	 (ФИО)
	 (подпись)	 (ФИО)
	 (подпись)	 (ФИО)
	 (подпись)	 (ФИО)

Москва 2025



Институт интеллектуальных кибернетических систем

КАФЕДРА КИБЕРНЕТИКИ

Задание на УИР

Студенту гр. Б 22-
544
(группа)

Писарев Александр Ильич
(фио)

ТЕМА УИР

Сравнительный анализ реализации микросервисной архитектуры с использованием паттерна Circuit Breaker на основе K3s и Istio.

ЗАДАНИЕ

№ п/п	Содержание работы	Форма отчетности	Срок исполнения	Отметка о выполнении Дата, подпись рук.
1.	Аналитическая часть			
1.1	Изучение и сравнительный анализ реализации Kubernetes K3s и сервисной mesh-платформы Istio (преимущества, недостатки, особенности настройки). Изучение паттерна «circuit breaker», логики его работы, особенностей реализации в Istio.	Текстовый сравнительный анализ систем, схема взаимодействия микросервисов.	1 неделя	
1.2	Анализ инструментов для нагрузочного тестирования k6: возможности, интеграция с Kubernetes, изучение возможных типов тестирования. Анализ возможностей системы трассировки Jaeger. Изучение способов интеграции в Kubernetes кластер, изучение процесса формирования метрик и отчетов о задержках.	Текстовый отчет, сценарии тестирования, подбор метрик для анализа.	3 неделя	
1.3	<i>Оформление расширенного содержания пояснительной записки (РСПЗ)</i>	Текст РСПЗ	8 неделя	
2.	Теоретическая часть			
2.1	Создание модели микросервисной архитектуры, в основе которой лежит паттерн circuit breaker как метод обеспечения устойчивости системы к сбоям.	Описанием структуры модели, диаграмма алгоритма работы Circuit Breaker.	5 неделя	

2.2	Интеграция в модель методов нагрузочного тестирования, добавление алгоритмов распределенного трассирования для сбора и анализа задержек.	Текстовый отчет со схемой графика.	6 неделя	
3.	Инженерная часть			
3.1	Проектирование архитектуры на уровне UML: создание диаграммы компонентов, диаграммы развертывания для наглядного представления взаимодействий микросервисов, сетевых соединений и конфигурации контейнеров.	UML диаграммы.	7 неделя	
4.	Технологическая и практическая часть			
4.1	Разработка и контейнеризация Python-клиента (echo-сервис, отвечающий на входящий запрос), подготовка Docker-образов, загрузка на Docker Hub, описание процедур сборки и развертывания с использованием helm чартов.	Исходный Python код, Docker-образы, README, yaml файлы.	8 неделя	
4.2	Реализация прокси-клиента с паттерном «circuit breaker», создание Docker-образа, конфигурация для приема внешних запросов и перенаправления на echo-сервис. Создание Helm чартов для развертывания в среде k3s.	Исходный Python код, Docker-образы, README, yaml файлы.	10 неделя	
4.3	Подготовка и настройка системы нагрузочного тестирования k6 и трассировки Jaeger на отдельной виртуальной машине, интеграция с приложениями для сбора и анализа метрик, логов и задержек.	Скрипты k6 и Jaeger, helm чарты.	11 неделя	
4.4	Реализация аналогичной схемы (echo и проху) микросервисов с использованием Istio (установка Istio в кластер, настройка правил «circuit breaker», маршрутизации, сбора метрик и трассировки).	Манифесты для Istio, конфигурационные файлы.	12 неделя	
4.5	Проведение нагрузочного тестирования обеих реализаций («чистая» реализация на python в K3s и на Istio), сбор метрик задержек и пропускной способности с помощью k6 и Jaeger, последующий анализ полученных данных, формирование отчетов и сравнительных графиков (matplotlib, seaborn).	Config файлы тестовых сценариев, графики задержек.	13 неделя	
5.	Оформление пояснительной записки (ПЗ) и иллюстративного материала для доклада.	Текст ПЗ, презентация.	13 неделя	

ЛИТЕРАТУРА

1.	Kleppmann M. Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems. – O'Reilly Media, 2017.
2.	Hightower K., Burns B., Beda J. Kubernetes: Up and Running: Dive into the Future of Infrastructure. – O'Reilly Media, 2017.
3.	Calcote L., Jory Z. Istio: Up and Running: Using a Service Mesh to Connect, Secure, Control, and Observe. – O'Reilly Media, 2020.
4.	Richardson C. Microservices Patterns: With Examples in Java. – Manning Publications, 2018.
5.	Molyneaux I. The Art of Application Performance Testing. – O'Reilly Media, 2011.
6.	Mastering k6: Performance Testing for Cloud Native Applications. – Leanpub, 2020

Дата выдачи задания:

Руководитель

к.т.н., доцент
Ровнягин
М.М.

(ФИО)

« 11 » 02 2025 г.

Студент

Писарев А.И.

(ФИО)

Реферат

Общий объем основного текста, без учета приложений — 37 страниц, с учетом приложений — 61. Количество использованных источников — 22. Количество приложений — 1. МИКРОСЕРВИСНАЯ АРХИТЕКТУРА, ПАТТЕРНЫ ОТКАЗОУСТОЙЧИВОСТИ, КОНТЕЙНЕРИЗАЦИЯ, CIRCUIT BREAKER ПАТТЕРН, SERVICE MESH АРХИТЕКТУРА, ISTIO

Целью данной работы является проектирование, внедрение и экспериментальное исследование паттерна Circuit Breaker в микросервисной архитектуре на базе Kubernetes (K3s), что заключается в создании собственной библиотеки для реализации данного паттерна, а также в проведении сравнительного анализа задержек между интегрированным решением Istio и самостоятельно разработанной реализации с целью выявления практических преимуществ и ограничений каждого подхода.

В первой главе проводится анализ современных архитектурных паттернов и технологий, используемых для обеспечения отказоустойчивости распределённых микросервисных приложений.

Во второй главе разрабатывается модель микросервисной архитектуры, основанная на применении паттерна Circuit Breaker, с интеграцией методов нагрузочного тестирования и алгоритмов распределённого трассирования для систематического сбора и последующего анализа задержек.

В третьей главе описывается архитектура проекта. Представляется UML диаграмма развертывания, а так же диаграмма последовательности.

В четвертой главе приводится описание программной реализации библиотеки на языке Python, описывается настройка и конфигурация Istio, приводится детальный сравнительный анализ времени работы, задержек и эффективности каждой из реализаций.

Содержание

Введение	5
1 Аналитический обзор современных решений для обеспечения отказоустойчивости в микросервисных архитектурах.	7
1.1 Анализ архитектурных паттернов, обеспечивающих отказоустойчивость. . . .	7
1.2 Анализ возможностей контейнеризации для повышения надежности микросервисной архитектуры	9
1.3 Анализ современных service mesh решений	11
1.4 Анализ инструментов и типов тестирования микросервисных архитектур. . .	13
1.5 Выводы	15
1.6 Цели и задачи УИР	16
2 Моделирование микросервисной архитектуры с применением паттерна Circuit Breaker и алгоритмов распределенного трассирования	18
2.1 Модель микросервисной архитектуры с применением паттерна Circuit Breaker	18
2.2 Интеграция методов нагрузочного тестирования и распределенного трассирования	19
2.3 Сравнительный анализ реализаций Circuit Breaker	19
2.4 Выводы	20
3 Проектирование микросервисной архитектуры с помощью UML диаграмм.	21
3.1 Описание архитектуры системы с помощью диаграммы развертывания.	21
3.2 Описание архитектуры системы с помощью диаграммы последовательности.	24
3.3 Выводы	26
4 Реализация и сравнительное тестирование системы.	27
4.1 Состав и структура реализованного программного обеспечения	27
4.2 Основные сценарии использования реализованных решений	28
4.3 Результаты тестирования	29
4.4 Выводы	36
Заключение	37

Приложения	41
А Приложение А. Листинг программной реализации	41

Введение

Индустрия разработки программного обеспечения в последние годы активно переходит от монолитных решений к микросервисной архитектуре, обеспечивающей масштабируемость и отказоустойчивость систем. Однако увеличение количества взаимодействующих компонентов существенно повышает вероятность сбоев, что требует внедрения эффективных механизмов обеспечения стабильности. Одним из ключевых решений данной проблемы является паттерн Circuit Breaker, предназначенный для изоляции неисправных сервисов и предотвращения каскадных отказов в распределенных системах.

Актуальность выбранной тематики обусловлена высоким интересом разработчиков облачных платформ, высоконагруженных сервисов и крупных интернет-проектов к надежности и производительности микросервисных решений. Исследования в области реализации паттерна Circuit Breaker важны для повышения устойчивости таких систем и минимизации негативных последствий сбоев.

Концепция паттерна Circuit Breaker была впервые детально описана в 2007 году Майклом Найгардом[1]. Значительный вклад в развитие практических решений внесла компания Netflix, выпустив в 2012 году библиотеку Hystrix [2]. Новый этап развития произошёл в 2017 году с появлением Istio[3] – платформы, интегрировавшей функциональность Circuit Breaker на уровне инфраструктуры сервисной сетки. Несмотря на широкое распространение данного подхода, открытым остается вопрос о сравнительной эффективности и влиянии различных реализаций паттерна на производительность микросервисной архитектуры.

Настоящая работа посвящена проведению комплексного сравнительного анализа реализации паттерна Circuit Breaker на основе сервисной сетки Istio и собственной Python-библиотеки, разработанной специально для данного исследования. Оригинальность исследования заключается в прямом сопоставлении двух различных подходов – встроенного инфраструктурного решения и специализированного программного модуля.

Научная значимость данной работы определяется эмпирическим характером анализа, выполненного в кластере K3s с использованием современных инструментов нагрузочного тестирования k6 и системы распределенного трассирования Jaeger. В исследовании делается акцент на сравнение задержек при использовании встроенной в Istio реализации с задержками, возникающими при использовании собственной реализации паттерна на Python, а также на устойчивость к отказам. Полученные результаты имеют практическую ценность,

предоставляя разработчикам распределенных систем рекомендации по выбору оптимального подхода для обеспечения отказоустойчивости в микросервисных архитектурах.

В первой главе представлен аналитический обзор современных архитектурных паттернов и технологий, применяемых для обеспечения отказоустойчивости в распределённых микросервисных приложениях. Рассматриваются возможности системы оркестрации, изучаются готовые решения для интеграции паттернов отказоустойчивости. Проводится анализ доступных методов тестирования и средств трассирования.

Во второй главе предложена модель микросервисной архитектуры, основанная на использовании паттерна Circuit Breaker. Описываются архитектурные особенности каждой из реализаций. Уточняется, что при использовании Istio возникают дополнительные sidecar-контейнеры. Особое внимание уделено процессу передачи трассировок от серверных приложений через data collector в систему распределённого трассирования Jaeger, что обеспечивает детальный анализ задержек.

Третья глава посвящена фактическому анализу архитектуры проекта. Представлены UML-диаграмма развёртывания и диаграмма последовательности, раскрывающие фактическое расположение микросервисов в кластере, способы взаимодействия между частями системы.

Четвёртая глава содержит описание практической реализации программной библиотеки на языке Python. Подробно рассмотрены вопросы настройки и конфигурации сервисной сетки Istio, а также представлен сравнительный анализ производительности, задержек и общей эффективности рассмотренных реализаций.

1. Аналитический обзор современных решений для обеспечения отказоустойчивости в микросервисных архитектурах.

В данном разделе обобщаются современные методы и инструменты, которые позволяют достичь высокой отказоустойчивости микросервисной архитектуры. Рассматриваются характерные проблемы в надёжности распределённых сервисов и анализируются архитектурные паттерны, повышающие их устойчивость (в том числе Circuit Breaker, Bulkhead и др.). Анализируются средства инфраструктуры - контейнеризация (Docker/Kubernetes) и сетевые решения (service mesh) – а также их роли в построении отказоустойчивых микросервисов. Далее, проведён сравнительный обзор популярных реализаций service mesh: Istio, Linkerd, Consul, Kuma. Затем, описываются методы тестирования отказоустойчивости: инструменты для нагрузочного тестирования (k6), способы трассировки распределённых запросов (Jaeger).

1.1 Анализ архитектурных паттернов, обеспечивающих отказоустойчивость.

В микросервисной архитектуре отказ одного сервиса способен вызвать цепочку проблем во всей системе - эффект каскадного сбоя. Чтобы локализовать сбои и предотвратить их распространение, применяются паттерны отказоустойчивости. **Circuit Breaker** является одним из наиболее распространённых решений для повышения надёжности сервисов [4]. Его идея позаимствована из электротехники: подобно автоматическому предохранителю, он разрывает цепь вызовов при обнаружении постоянных ошибок во взаимодействии сервисов. Реализация данного паттерна предусматривает мониторинг удалённых вызовов: если за заданный интервал накопилось определённое число неудач (исключений, таймаутов), Circuit Breaker переводится в состояние «Open» и начинает блокировать дальнейшие запросы к проблемному сервису, сразу возвращая ошибку или резервный ответ. Через некоторое время (период полуоткрытого состояния «Half-Open») пропускается пробный запрос, и при успешном ответе замыкается обратно (состояние «Closed»), возобновляя нормальную работу; если же ошибка сохраняется – цикл повторяется. Таким образом, данный паттерн позволяет избегать дополнительной нагрузки на зависимый сервис, давая ему время восстановиться, и сокра-

щает время ожидания для потоков, обращающихся к недоступному ресурсу [5]. На практике, использование Circuit Breaker снижает риск обрушения всей системы из-за одного сбойного компонента.

Другим важным шаблоном является паттерн **Bulkhead**. Этот шаблон предлагает изолировать ресурсы для отдельных сервисов или групп запросов. Например, можно выделить независимые пулы потоков или подключений к базе данных для каждого микросервиса. В результате ошибка или задержка в одном компоненте приведет к потреблению ресурсов только своего пула и не сможет повлиять на всю систему. Паттерн bulkhead повышает устойчивость за счёт ограничения области воздействия сбоя: сбойный сервис исчерпает лишь свой выделенный лимит потоков/соединений, но остальные сервисы продолжают работать штатно [6]. На практике Bulkhead часто реализуется средствами контейнерной оркестрации или специальных библиотек — например, выделением отдельного пула потоков в контейнере для каждого клиента. В сочетании с Circuit Breaker, Bulkhead образует многослойную защиту: один паттерн быстро отсекает проблемные вызовы, а другой обеспечивает изоляцию ресурсов и предотвращает каскадное распространение отказов в системе, локализуя сбои и минимизируя их влияние на остальные компоненты. По данным исследований, паттерн Circuit Breaker способен снизить долю фатальных ошибок почти на 60%, а Bulkhead повысить общую доступность сервисов примерно на 10% [6].

Помимо указанных двух паттернов, существуют другие способы повышения устойчивости архитектуры микросервисов. Часто применяется паттерн **Retry** (повторный запрос) — автоматическая повторная попытка вызова внешнего сервиса при неудаче, обычно с экспоненциальной задержкой между попытками, чтобы не создать дополнительную нагрузку. Retry полезен при кратковременных сбоях (transient errors), но должен сочетаться с ограничением числа попыток и с Circuit Breaker, чтобы повторные вызовы не усугубляли ситуацию при длительной недоступности сервиса [5]. Ещё один важный механизм — **Timeout** (таймауты на вызовы): если внешний запрос не отвечает за разумное время, он прерывается, что освобождает занятые ресурсы. В связке с таймаутами обычно используются Fallback-методы — альтернативные действия при сбое внешнего сервиса (например, возврат кэшированного значения, дефолтного ответа, сообщения об ошибке). Такие меры позволяют системе завершаться аккуратно, без резкого отказа [7].

Комплексное применение данных паттернов доказало свою эффективность на практике. В частности, опыты, проведённые корпорацией Netflix при внедрении библиотеки Hystrix, показали существенное снижение числа инцидентов, связанных с каскадными сбоями. Эти паттерны стали настолько важны, что современные фреймворки (например, Resilience4j для

Java) и сервис-меш технологии (см. подраздел 1.2) поддерживают их «из коробки». Таким образом, архитектурные решения в виде Circuit Breaker, Bulkhead и сопутствующие им шаблоны (Retry, Timeout, Fallback) являются важнейшими компонентами обеспечения отказоустойчивости микросервисных приложений[8].

Паттерн Circuit Breaker является ключевым механизмом обеспечения отказоустойчивости микросервисной архитектуры, так как он своевременно обнаруживает сбои и локализует их, предотвращая цепную реакцию отказов в системе. Выбор этого паттерна для исследования обусловлен его доказанной эффективностью в снижении риска системных сбоев и возможности изолировать неисправные компоненты, что существенно повышает общую стабильность распределённых приложений.

1.2 Анализ возможностей контейнеризации для повышения надежности микросервисной архитектуры

Контейнеризация и оркестрация внесли решающий вклад в повышение отказоустойчивости облачных приложений. Контейнеры предоставляют лёгковесную изоляцию сервисов, позволяя каждому микросервису работать в унифицированной среде с чётко определёнными зависимостями. Это снижает вероятность ошибок окружения и облегчает масштабирование. Более того, запуск сервиса в виде контейнера существенно ускоряет его перезапуск при сбое, позволяет автоматизировать повторный запуск.

Ключевым инструментом управления контейнеризированными микросервисами является Kubernetes – платформа оркестрации. Kubernetes автоматически следит за состоянием приложений и применяет стратегии self-healing (самовосстановления): при падении контейнера оркестратор перезапускает его, при отключении узла – переносит поды на работоспособные узлы, а при повышении нагрузки – может автоматически масштабировать реплики сервисов[9]. Тем самым обеспечивается базовая устойчивость системы к аппаратным и программным сбоям без вмешательства администратора. Таким образом, оркестратор действует как «операционная система» для кластера, гарантируя, что заданное число экземпляров каждого сервиса всегда будет создано, несмотря на возможные локальные сбои. Существуют различные реализации Kubernetes. Для ресурсов с ограниченными вычислительными ресурсами разработаны облегчённые дистрибутивы Kubernetes, такие как **K3s**. Данная реализация представляет собой полностью совместимую с Kubernetes версию оркестратора, упакованную в один исполняемый файл размером менее 100 МБ [10]. Использование K3s позволяет вынести микросервисную инфраструктуру за пределы крупных датацентров, обеспечивая при этом механизмы отказоустойчивости Kubernetes в малых масштабах.

Однако одного механизма перезапуска сервисов недостаточно для комплексной надёжности: необходимо обеспечить устойчивость между сервисами, на уровне их взаимодействий. Для обеспечения такой устойчивости, была разработана концепция **service mesh**. Service mesh – это дополнительный коммуникационный слой поверх сетевого взаимодействия микросервисов. Он решает задачи обнаружения сервисов, балансировки нагрузки, шифрования трафика, а также реализации шаблонов устойчивости (таких как Circuit breaker, Bulkhead см. подраздел 1.1) на уровне сети. Типичная архитектура service mesh включает *data plane* (сетевые прокси рядом с каждым сервисом) и *control plane* (централизованный диспетчер, управляющий правилами маршрутизации и сбора телеметрии). В результате каждый межсервисный вызов проходит через локальный прокси, который может выполнить необходимую логику. К примеру, на этом слое запрос может быть перенаправлен по другому маршруту, если основной сервис недоступен, соединение может быть дополнительно зашифровано, или реализован механизм прекращения связи с сервисом при повторяющихся неудачах (Circuit breaker на уровне service mesh). Таким образом, применение mesh-инфраструктуры позволяет централизованно реализовать множество механизмов обеспечения надёжности, которые в ином случае пришлось бы интегрировать непосредственно в код каждого отдельного сервиса. Исследования отмечают, что service mesh позволяет стандартизировать и упростить коммуникации: все вызовы проходят единообразную обработку, что снижает вероятность непредусмотренных отказов[11]. Примеры реализаций service mesh включают **Istio**, **Linkerd**, **Consul Connect** и **Kuma** (их сравнительный анализ приведён в подразделе 1.3). Общая цель у них одна – надёжная доставка запросов между микросервисами[12], но подходы в реализации различаются.

Service mesh тесно интегрируется с оркестратором: например, в Kubernetes прокси-меш обычно развёртываются как sidecar-контейнеры внутри тех же подов, что и основные сервисы. Такая интеграция позволяет Kubernetes автоматически внедрять mesh-политику при масштабировании или перезапуске сервисов. Таким образом, наблюдается эффективное взаимодействие компонентов системы: Kubernetes обеспечивает физическую устойчивость функционирования сервисов (их запуск, перезапуск и распределение), а mesh-слой — логическую устойчивость коммуникаций посредством маршрутизации, повторных попыток и изоляции сбоев. Современные исследования подтверждают, что сочетание оркестрации и service mesh значительно повышает надёжность сложных распределённых архитектур[11]. Можно заключить, что для построения отказоустойчивых микросервисных систем требуется многоуровневый подход. Необходимо обеспечить устойчивость отдельных сервисов (за счёт контейнеризации и оркестрации), а также реализовать устойчивость их взаимодействия (за счёт

service mesh).

1.3 Анализ современных service mesh решений

Разработчики, использующие Kubernetes для оркестрации, могут выбирать из нескольких open-source реализаций service mesh. Наиболее популярны следующие решения:

1. **Istio** — масштабируемый сервис-меш, изначально разработанный Google, IBM и RedHat (появился в 2017 г.).
2. **Linkerd** — лёгкий mesh, ориентированный на простоту и высокую производительность. Изначально разработан компанией Buoyant, переписан на Rust.
3. **Consul (Connect)** — расширение популярной системы Consul от HashiCorp до полноценного service mesh.
4. **Kuma** — относительно новое решение (разработано Kong Inc., выпущено в 2020 г.), позиционируемое как «универсальный сервис-меш».

Для более наглядного сравнения основных характеристик рассмотренных mesh-решений приведём обобщённую таблицу (табл. 1.1). Здесь сопоставляются функциональные возможности, требования и сферы применения Istio, Linkerd, Consul и Kuma.

Таблица 1.1 – Сравнение популярных service mesh для Kubernetes

Аспект	Istio	Linkerd	Consul Connect	Kuma
Архитектура и прокси	Envoy sidecar; контроллер Istiod. Модульная архитектура (Pilot, Mixer в ранних версиях)	Специальный лёгкий проху (Linkerd2-проху) на Rust; минималистичный контроллер. Монолитная архитектура.	Envoy sidecar (или собственный проху), контроллер интегрирован в Consul server. Требуется Consul-кластер.	Envoy sidecar; control-plane Kuma. Архитектура похожа на Istio, но проще
Функционал и трафик	Широкий: маршрутизация, балансировка, mirroring, fault injection, паттерны отказоустойчивости. Поддержка шлюзов ingress/egress.	Базовые возможности: load balancing, service discovery, mTLS. Ограниченная настройка маршрутов.	Широкие возможности service discovery (KV-хранилище). Traffic management присутствует, но менее гибкий, чем в Istio	Настройка трассировки, балансировки, маршрутизации, политики отказов, поддержка нескольких mesh. Мультизональность.

Продолжение на следующей странице

Таблица 1.1 – Продолжение

Аспект	Istio	Linkerd	Consul Connect	Kuma
Безопасность	Полноценная инфраструктура: автовыдача сертификатов, mTLS, детальные настройки авторизации (RBAC, политики доступа)	mTLS по умолчанию для всех соединений. Нет встроенной сложной авторизации (интеграция с K8s RBAC)	mTLS через собственный CA Consul, гибкие политики «Intentions». Централизованное управление ACL	mTLS встроен (встроенный-/внешний CA), политики доступа. Глобальные политики безопасности между зонами
Производительность	Высокий overhead на Envoy проху (CPU/память в 2–3 раза выше). Задержки под нагрузкой [13], хорошая масштабируемость	Минимальный overhead (быстрый проху, мало ресурсов). Незначительные задержки. Эффективен при большом числе сервисов	Умеренные накладные расходы. Дополнительный ресурс на Consul-сервер. При >1000 сервисов Consul может стать узким местом	Сопоставимо с Istio. Контрольная плоскость лёгкая. Оптимизирован для распределённых сред
Мультиклас-терность	Поддерживается (multi-mesh/federation) — сложная настройка. Ориентирован на K8s; VM требуют доп. компонентов	Экспериментальное расширение Multi-Cluster. За пределами K8s не работает (только соединяет кластеры K8s)	Спроектирован для multi-datacenter. Легко соединяет кластеры и VM. Отличен для гибридных облаков	Концепция Zones: объединяет множество K8s-кластеров и других нод. Гибко работает в гибридных средах

Из таблицы видно, что **Istio** предлагает наибольшие возможности в плане управления трафиком и политик, но в то же время требует больше ресурсов и усилий на сопровождение. **Linkerd** наиболее лёгок и прост, что делает его оптимальным для небольших команд или в ситуациях, где дополнительные возможности mesh не столь востребованы. **Consul** выгодно отличается способностью соединять разные среды (не только Kubernetes), однако добавляет сложность развёртывания отдельного Consul-кластера. **Kuma** стремится сочетать преимущества обоих подходов – универсальность Consul с относительной простотой Linkerd.

В контексте данной работы выбор сделан в пользу **Istio**. Во-первых, как отмечается в исследованиях, Istio остаётся наиболее функционально насыщенной и гибкой платформой [11],

которая особенно эффективна в крупных масштабируемых системах. Во-вторых, нас интересует реализация паттерна Circuit Breaker и других механизмов отказоустойчивости на уровне service mesh – Istio предоставляет развитые средства для этого. Таким образом, несмотря на издержки в сложности, Istio представляется оптимальным выбором для изучения и реализации отказоустойчивой микросервисной архитектуры с использованием Kubernetes.

1.4 Анализ инструментов и типов тестирования микросервисных архитектур.

Нагрузочное тестирование микросервисов можно проводить с помощью различных open-source инструментов, среди которых Gatling, Locust, k6 и др. Каждый из них имеет свои преимущества и недостатки, которые стоит учитывать при выборе для тестирования.

1. Gatling[14] – высокопроизводительный инструмент для тестирования, разработанный на Scala. Он эффективно генерирует нагрузку даже при тысячах виртуальных пользователей, сохраняя низкую нагрузку на саму систему тестирования. Детализированные отчёты и графики помогают проводить глубокий анализ результатов. Недостатком является необходимость знания Scala и функционального программирования.
2. Locust [15] – инструмент на Python, который позволяет описывать сценарии тестирования с использованием асинхронного подхода. Это делает его масштабируемым и гибким: поведение пользователей можно задавать программно, что удобно для сложных сценариев [16]. Интеграция с CI/CD и возможность распределённых тестов – дополнительные плюсы. Однако встроенные средства визуализации менее развиты, и для полноценного анализа результатов могут потребоваться дополнительные инструменты.
3. k6[17] – современный инструмент для тестирования API и микросервисов, реализованный на Go с использованием JavaScript для описания сценариев. Он характеризуется низкими накладными расходами и эффективным использованием ресурсов, что позволяет генерировать значительную нагрузку. Прямая интеграция с системами мониторинга и возможность автоматизированного запуска в CI/CD делают его привлекательным для современных веб-сервисов. Основное ограничение – поддержка в основном HTTP/HTTPS и WebSocket, что может не подойти для тестирования специфических протоколов.

Проанализировав перечисленные решения, в данном исследовании выбор сделан в пользу k6 как основного инструмента нагрузочного тестирования. Это обусловлено несколькими факторами. Во-первых, написание сценариев на JavaScript упрощают разработку тестов. Во-

вторых, низкая нагрузка самого инструмента на систему позволяет проводить более чистые эксперименты. В-третьих, он уже применялся в научных работах для анализа микросервисных архитектур. [18]

Нагрузочное тестирование включает несколько типов тестов, каждое из которых позволяет оценить разные аспекты производительности системы. К основным относятся:

1. Нагрузочный тест (Load Test)[19]: проверяет работу сервиса при постепенном увеличении числа запросов или виртуальных пользователей, моделируя реальный рост трафика.
2. Стресс-тест (Stress Test): определяет пределы устойчивости системы, подвергая её нагрузке, значительно превышающей номинальную.
3. Спайк-тест (Spike Test): моделирует резкие кратковременные всплески нагрузки для анализа реакции системы на внезапные изменения.
4. Тест выносливости (Soak Test): проводит длительное воздействие средних нагрузок, выявляя проблемы, связанные с утечками памяти и ухудшением производительности со временем.

В рамках исследования особый интерес представляют сценарии, в которых паттерн будет иметь смысл например, при возникновении чрезмерных нагрузок на сервис и возникновении его зависания. Выбраны два сценария:

1. Нагрузочный тест: В данном сценарии постепенно растёт число виртуальных пользователей или запросов в секунду, что позволяет смоделировать возрастающую нагрузку, аналогичную пиковым периодам активности. При приближении к пределам пропускной способности системы могут возникать ошибки и увеличиваться время отклика. Circuit Breaker должен реагировать, чтобы предотвратить дальнейшие неудачные обращения к перегруженному микросервису.
2. Тест с увеличением размера запроса: Здесь фиксируется количество одновременных запросов, но постепенно увеличивается размер полезной нагрузки каждого запроса. Такой подход имитирует ситуацию, когда сервисы обрабатывают всё более сложные или объёмные сообщения, что может приводить к увеличению времени обработки, повышенной задержке или таймаутам. Circuit Breaker трактует накопление задержек и таймаутов как сигналы ухудшения качества сервиса и переключается в режим для защиты системы от перегрузки.

В микросервисной архитектуре запрос проходит через цепочку сервисов и функций, что требует детального мониторинга его маршрута и анализа задержек на каждом этапе. Трассиров-

ка позволяет выявить узкие места, определить причины ошибок и оптимизировать производительность системы. Распределённое трассирование собирает данные о каждом микросервисе, участвующем в обработке запроса, и позволяет отследить все стадии прохождения запроса - от времени обработки отдельных операций до момента возникновения таймаутов. Для реализации такой трассировки используются решения, среди которых популярны Jaeger, Zipkin и OpenTelemetry Collector.

1. Jaeger[20] – система end-to-end трассировки, разработанная в Uber и поддерживаемая CNCF. Она собирает и визуализирует трассы, поддерживая различные модели хранения и масштабируясь под большие объёмы данных. Благодаря совместимости со стандартом OpenTelemetry, Jaeger легко интегрируется с разными языками и фреймворками. Модульная архитектура (агенты, коллекторы, веб-интерфейс) обеспечивает детальное отображение задержек, ошибок и зависимостей между сервисами, что делает его незаменимым инструментом для анализа распределённых систем.
2. Zipkin[21], созданный в Twitter, также собирает трейс-спаны и предоставляет удобный веб-интерфейс для анализа. Он отличается простотой развертывания и использует формат V3 для передачи заголовков, что особенно удобно в экосистеме Spring Boot. Однако Zipkin менее оптимизирован для крупных инсталляций, что может ограничивать его применение в больших кластерах[22].
3. OpenTelemetry Collector[23] – универсальный агент, собирающий метрики, логи и трассы в разных форматах. Он не является самостоятельной системой визуализации, а служит посредником, отправляя данные в выбранное хранилище или систему анализа (например, Jaeger или Zipkin). Его главное преимущество – независимость от конкретного вендора и гибкость в настройке, что упрощает интеграцию с разными инструментами мониторинга.

Учитывая поддержку стандарта OpenTelemetry, полноценную визуализацию, лёгкую интеграцию с Kubernetes/Istio и масштабируемость, для исследования выбрана система Jaeger как основное средство сбора трассировок и анализа задержек. Jaeger, обладая удобным веб-интерфейсом, превосходит альтернативные решения, обеспечивая надёжную основу для исследования метрик и задержек в микросервисной архитектуре.

1.5 Выводы

В результате проведенного анализа можно подвести следующие итоги:

1. Анализ показал, что такие шаблоны, как Circuit Breaker и Bulkhead эффективно предотвращают распространение сбоев в распределённой системе. Circuit Breaker ограничи-

вает каскадные ошибки, а Bulkhead не даёт одному сбойному компоненту исчерпать общие ресурсы системы. Circuit Breaker выбран для исследования из-за его распространённости, а так же проверенной способности снижать вероятность системных сбоев.

2. Оркестратор обеспечивает автоматическое поддержание работы сервисов, создавая устойчивую основу исполнения. Service mesh добавляет уровень защиты на сетевом уровне, управляя межсервисными вызовами: от балансировки нагрузки до шифрования и механизмов отказоустойчивости. Совместное использование этих технологий позволяет достичь высокого уровня отказоустойчивости.
3. Существующие реализации service mesh отличаются балансом функциональности и сложности. Сравнение сервисов (Istio, Linkerd, Consul, Kuma) продемонстрировало, что нет универсального решения: выбор mesh зависит от потребностей конкретного проекта. Для целей нашего исследования выбрано Istio как наиболее функционально насыщенная и популярная в индустрии.
4. Нагрузочные тесты с помощью k6 выявляют поведение системы на предельных режимах работы и эффективны для проведения анализа времени работы системы. Трассировка с помощью Jaeger даёт понимание, где возникают задержки или сбои в цепочке сервисов. Промежуточный сборщик логов и трейсов позволяет асинхронно работать с данными.

1.6 Цели и задачи УИР

Для достижения цели необходимо выполнить следующие задачи:

1. Создание экспериментальной инфраструктуры

- 1.1. Разработка тестового микросервисного приложения с возможностью искусственного введения сбоев и задержек для моделирования различных сценариев отказов.
- 1.2. Развёртывание кластера Kubernetes (K3s) и интеграция в него сервисной платформы Istio.
- 1.3. Реализация конфигураций Istio, обеспечивающих применение паттерна Circuit Breaker (через правила DestinationRule).
- 1.4. Развёртывание другого кластера Kubernetes (K3s) без использования service mesh.
- 1.5. Написание Python библиотеки, реализовывающей Circuit Breaker паттерн и интеграция ее в код микросервисов.
- 1.6. Внедрение системы распределённого трейсинга Jaeger.

1.7. Настройка data collector для оценки состояния инфраструктуры и поведения системы при сбоях.

2. Экспериментальное исследование и тестирование

2.1. Разработка и проведение нагрузочных тестов с использованием инструмента k6 для анализа производительности и отказоустойчивости системы при различных уровнях нагрузки и при активации механизмов Circuit Breaker.

2.2. Эмуляция отказов компонентов приложения и сетевых задержек с помощью средств Istio Fault Injection с оценкой эффективности реагирования сервисов.

2.3. Сбор и анализ трассировок Jaeger и метрик производительности системы (latency, throughput, доля ошибок, потребление ресурсов), для объективной оценки работы механизмов отказоустойчивости.

2.4. Сравнение результатов работы системы в нескольких конфигурациях (без Istio, с Istio без Circuit Breaker, с Istio и Circuit Breaker) для количественного подтверждения влияния исследуемых подходов.

3. Анализ результатов и формулирование рекомендаций

3.1. Интерпретация экспериментальных данных и оценка влияния service mesh (Istio) и паттерна Circuit Breaker на показатели отказоустойчивости микросервисного приложения.

3.2. Сравнение экспериментальных показателей задержек между самостоятельно реализованной библиотекой и Envoy proxy.

3.3. Разработка практических рекомендаций по оптимальному использованию механизмов Circuit Breaker и сервисных сетей (service mesh), с выделением сценариев и условий, при которых их применение наиболее эффективно.

2. Моделирование микросервисной архитектуры с применением паттерна Circuit Breaker и алгоритмов распределенного трассирования

В данном разделе описываются модели микросервисной архитектуры, ориентированной на повышение устойчивости системы к отказам за счёт применения паттерна Circuit Breaker. Рассматриваются подходы к реализации паттерна в конкретном программном окружении и способы интеграции с существующими сервисами. Также затронуты вопросы организации инфраструктуры на базе k3s с применением инструментов управления конфигурацией (Helm), а также интеграции компонентов мониторинга и трассирования с использованием Fluent Bit и Jaeger.

2.1 Модель микросервисной архитектуры с применением паттерна Circuit Breaker

Рассматриваемая микросервисная архитектура реализована в рамках k3s кластера, где осуществляется маршрутизация трафика между различными узлами посредством внутреннего ClusterIP и TCP соединений. Клиент направляет HTTP-запросы на прокси-компонент, который затем взаимодействует с сервером. Сервер представлен в виде Flask-приложения, обрабатывающего запросы и формирующего ответы, а прокси-компонент выступает в качестве промежуточного звена между клиентом и сервером.

В версии без использования Istio прокси-компонент реализован с применением собственной Python-библиотеки, включающей паттерн Circuit Breaker. Данный паттерн обеспечивает контроль состояния взаимодействия с сервером: при обнаружении ошибок или превышении заданных пороговых значений прокси может немедленно отказать в обслуживании запроса, предотвращая дальнейшие попытки обращения к серверу. При этом веб-приложения сервера и прокси развернуты на отдельных нодах, что повышает отказоустойчивость системы за счёт изоляции компонентов.

При использовании service mesh решения Istio архитектурное устройство сохраняется, однако в поды компонентов прокси и веб-сервера добавляются sidecar-контейнеры, предоставляющие функции проксирования и сетевого взаимодействия. В этом варианте реализация паттерна Circuit Breaker осуществляется средствами Istio, что позволяет исключить

наличие данного паттерна в коде прокси-приложения.

2.2 Интеграция методов нагрузочного тестирования и распределенного трассирования

В предлагаемой архитектуре интеграция методов нагрузочного тестирования и распределённого трассирования осуществляется посредством специализированных компонентов, развернутых на выделенной k3s ноде. Здесь разворачиваются инструмент k6 для имитации нагрузки, промежуточный сборщик логов fluent bit и система централизованного трассирования Jaeger. Сервер и прокси-компоненты выводят свои логи и трейсы в stdout, после чего fluent bit асинхронно подхватывает, группирует и отправляет их в Jaeger для дальнейшего анализа. Такой подход обеспечивает оперативный сбор данных, позволяя визуализировать распределённые транзакции и выявлять узкие места в работе микросервисной системы.

При использовании Istio принцип работы системы сохраняется: компоненты веб-сервер и прокси, дополнительно снабжённые sidecar-контейнерами, продолжают генерировать логи и трейсы, которые обрабатываются fluent bit и передаются в Jaeger. Нагрузочное тестирование посредством k6 также остаётся неизменным, что позволяет оценить устойчивость системы при различных сценариях нагрузки. Такой унифицированный подход к сбору и анализу данных обеспечивает прозрачность потоков информации и способствует более эффективной диагностике и оптимизации работы микросервисов.

2.3 Сравнительный анализ реализаций Circuit Breaker

Реализация Circuit Breaker в анализируемых системах осуществляется посредством Helm-чартов, что обеспечивает стандартизированный процесс развертывания и конфигурации. В инфраструктурном подходе с использованием Istio применяется дополнительный YAML-файл – destination rule, в котором описаны параметры Circuit Breaker. В свою очередь, программная реализация на базе Python требует внесения изменений непосредственно в код прокси-компонента, что обуславливает более тесную интеграцию логики обработки сбоев в приложение.

Механизмы определения сбоев в обоих подходах базируются на анализе логов и трассировочных данных. Логи, генерируемые Python кодом, асинхронно собираются Fluent Bit, а трассировочные данные визуализируются в Jaeger UI, что позволяет оперативно выявлять аномалии в работе системы. Такой подход обеспечивает прозрачное наблюдение за потоками данных и помогает быстро диагностировать проблемы, независимо от способа реализации Circuit Breaker.

Быстрое восстановление работоспособности системы достигается за счет возможностей платформы k3s: рестарты контейнеров и перезапуски релизов с помощью Helm позволяют минимизировать время простоя.

Обе реализации Circuit Breaker демонстрируют высокую отказоустойчивость системы, обеспечивая оперативное обнаружение и восстановление сбоев за счет интеграции в k3s кластер с использованием Helm-чартов. Решение на базе Istio, с применением YAML-конфигурации `destination rule`, упрощает централизованное управление и мониторинг, исключая необходимость внесения изменений в исходный код приложения. В то время как программная реализация на Python обеспечивает более гибкую настройку логики непосредственно в коде, что может быть предпочтительно для специфичных бизнес-задач.

2.4 Выводы

В данном разделе сформулированы ключевые выводы по теоретической разработке модели микросервисной архитектуры, основанной на сравнительном анализе двух подходов к реализации паттерна Circuit Breaker. Исследование показало, что применение `service mesh`, в частности Istio, позволяет централизовать управление сетевыми взаимодействиями за счет использования YAML-конфигураций (`destination rule`), что упрощает настройку отказоустойчивости и обеспечивает прозрачное внедрение политик обработки сбоев. Альтернативно, собственная реализация на базе Python предусматривает внесение изменений непосредственно в код прокси-компонента, что обеспечивает более гибкую настройку логики Circuit Breaker для специфичных бизнес-сценариев.

Интеграция методов распределённого трассирования и нагрузочного тестирования посредством Jaeger, Fluent Bit и k6 продемонстрировала высокую эффективность в обнаружении аномалий и оперативном восстановлении работоспособности системы. Сбор логов и трейсов, осуществляемый через `stdout` компонентов и асинхронно обрабатываемый Fluent Bit, позволяет проводить детальный визуальный анализ в Jaeger UI, что существенно ускоряет диагностику и минимизирует время простоя. Автоматизированное восстановление через рестарты контейнеров в k3s и быстрый рестарт релизов посредством Helm-чартов дополнительно повышают устойчивость архитектуры.

Выбор между Istio и собственным решением на Python сводится к компромиссу между удобством централизованного администрирования и гибкостью настройки, а так же временем работы. Дальнейшие практические исследования будут направлены на детальное измерение времени работы и задержек, что позволит оценить влияние выбранного подхода на общую производительность и надежность архитектуры.

3. Проектирование микросервисной архитектуры с помощью UML диаграмм.

В данной главе представлено подробное описание спроектированной микросервисной архитектуры, отражающее как физическое распределение компонентов, так и их логическую взаимосвязь. Используя UML диаграммы развертывания, продемонстрировано разделение системы на микросервисы с акцентом на стандартизированные интерфейсы для обмена данными между ними, а также описаны внешние интерфейсы, обеспечивающие связь с клиентами и другими системами.

Также приведены диаграммы последовательности, иллюстрирующие все стадии маршрутизации запросов от клиента до сервера и обратно. Такой комплексный подход, включающий использование `sidecar`-компонентов для управления трафиком и интеграцию систем мониторинга, позволяет получить достоверную модель архитектуры, готовую к реализации и дальнейшему тестированию.

3.1 Описание архитектуры системы с помощью диаграммы развертывания.

На диаграмме развертывания (см. рис. 3.1) система представлена с использованием Istio `sidecar` в каждом поде: `Python-server`, прокси-сервер дополнены `sidecar` контейнерами, обеспечивающими перехват и маршрутизацию трафика. Такое решение упрощает контроль сетевых потоков и повышает наблюдаемость благодаря прямой интеграции с `fluentbit`, который собирает логи и метрики. Вариант без Istio (см. рис. 3.2) показывает, как сервисы могут взаимодействовать напрямую, сохраняя более простую структуру развертывания, но при этом теряя автоматизированные возможности балансировки и политики безопасности. На обоих вариантах диаграммы отражено, что каждая виртуальная машина в кластере `k3s` содержит отдельные поды: один с основным приложением и `sidecar` (или без него), и дополнительный `pod` с `fluentbit`. Helm-чарт используется для автоматизации развертывания, что позволяет централизованно управлять конфигурациями.

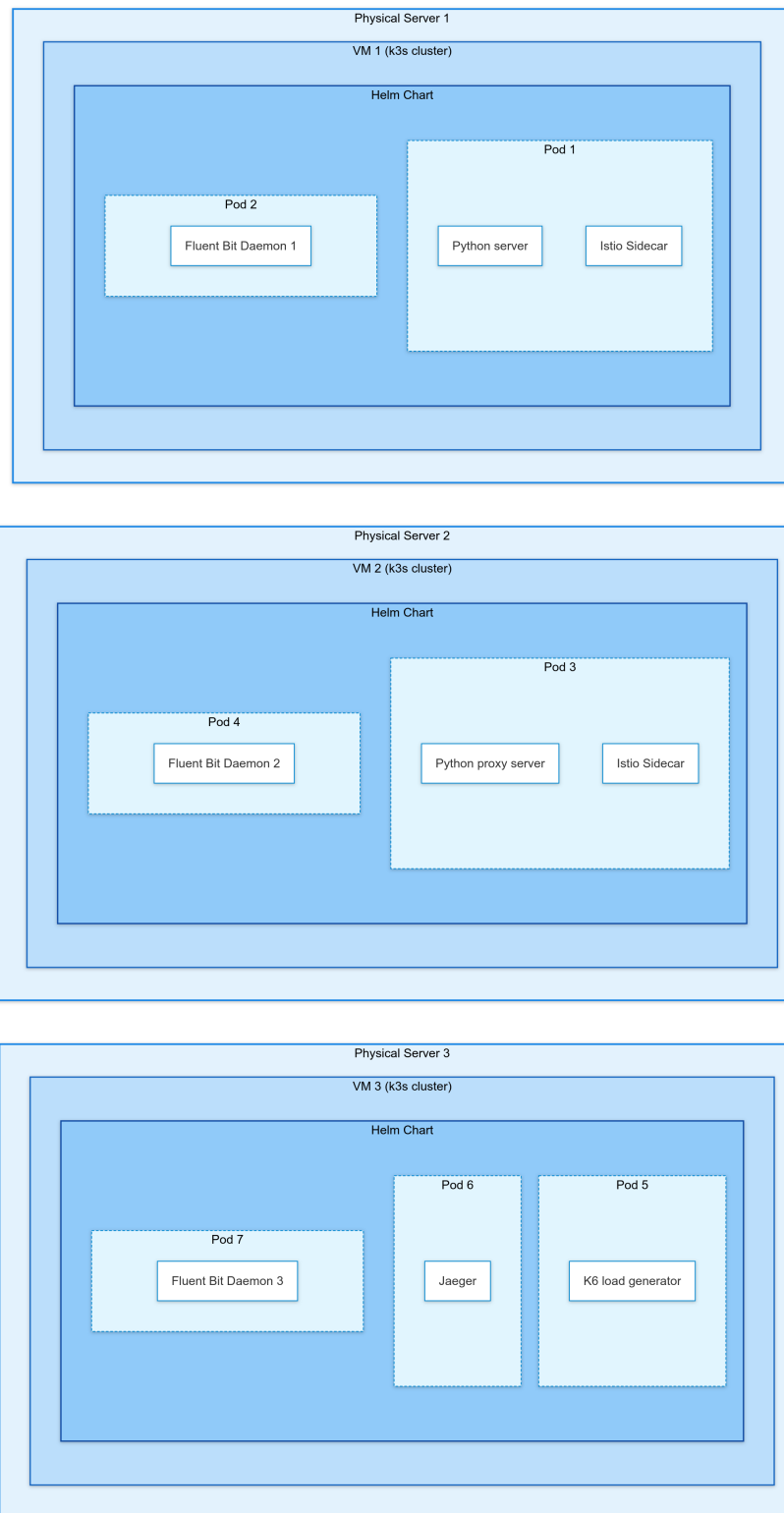


Рисунок 3.1 – Диаграмма развертывания версии с Isito

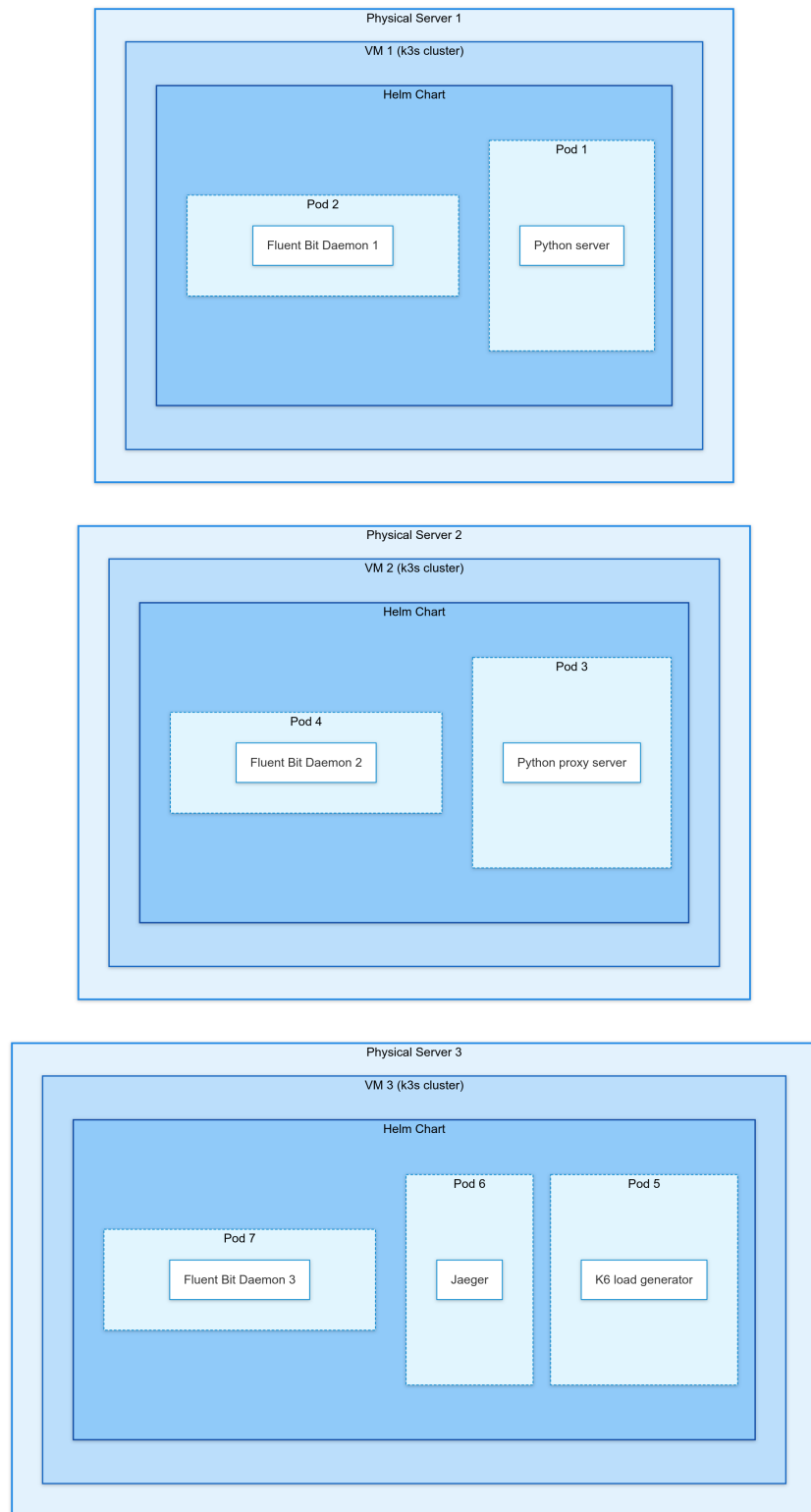


Рисунок 3.2 – Диаграмма развертывания версии без Isito

3.2 Описание архитектуры системы с помощью диаграммы последовательности.

Диаграмма последовательности (см. рис. 3.3), демонстрирует прохождение трафика через Istio sidecar: клиент формирует запрос, прокси перенаправляет его в Istio, который дополнительно проверяет доступность основного сервиса и контролирует возможные сбои. Это даёт дополнительный уровень управления и безопасности, но требует промежуточной обработки.

В варианте без service mesh (см. рис. 3.4) реализована более простая схема обмена сообщениями: прокси общается с сервером напрямую, без дополнительного слоя Istio. Такая реализация может быть проще в настройке, но лишена встроенных возможностей мониторинга и контроля, которые предоставляет Istio.

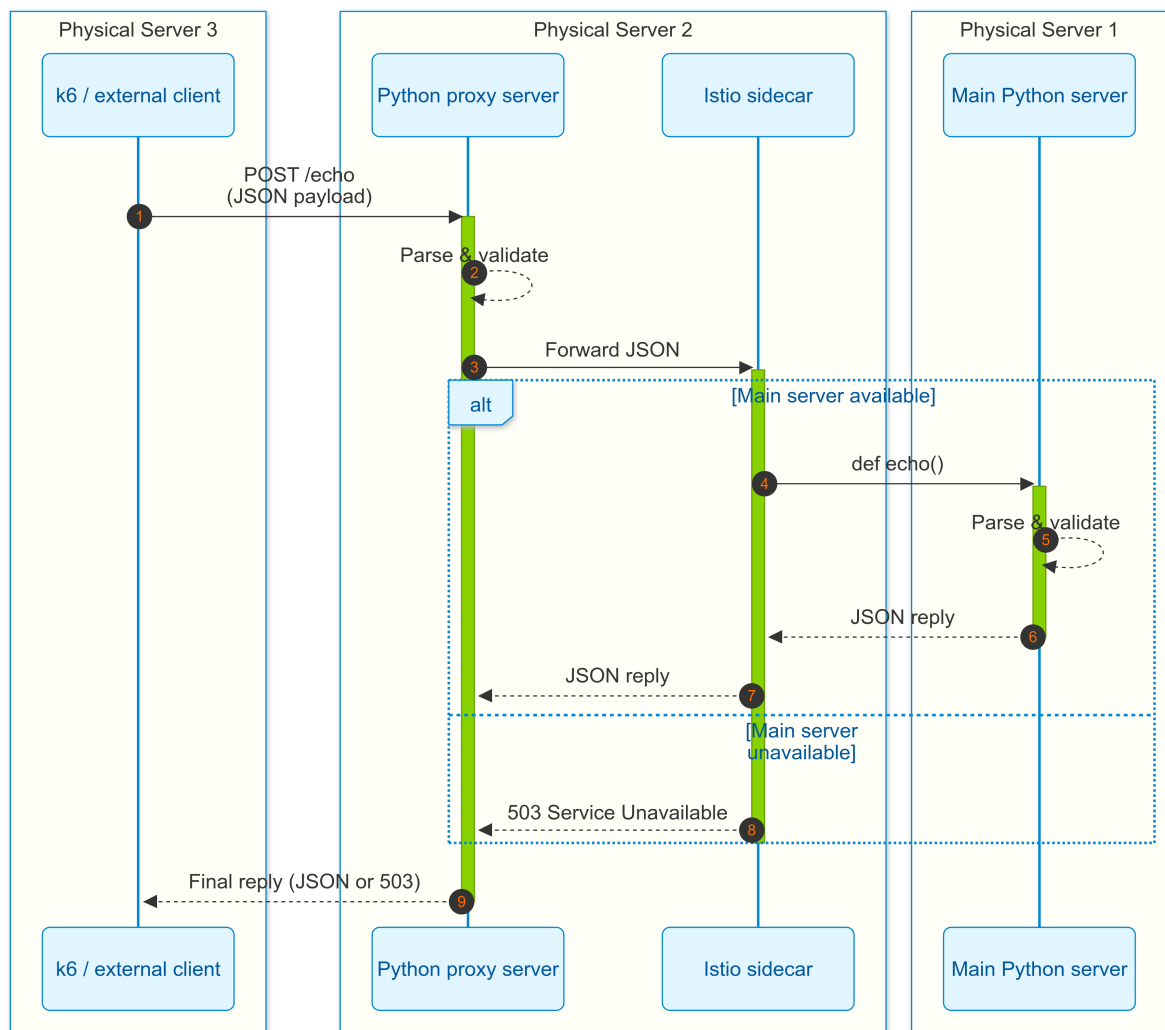


Рисунок 3.3 – Диаграмма последовательности версии с Istio

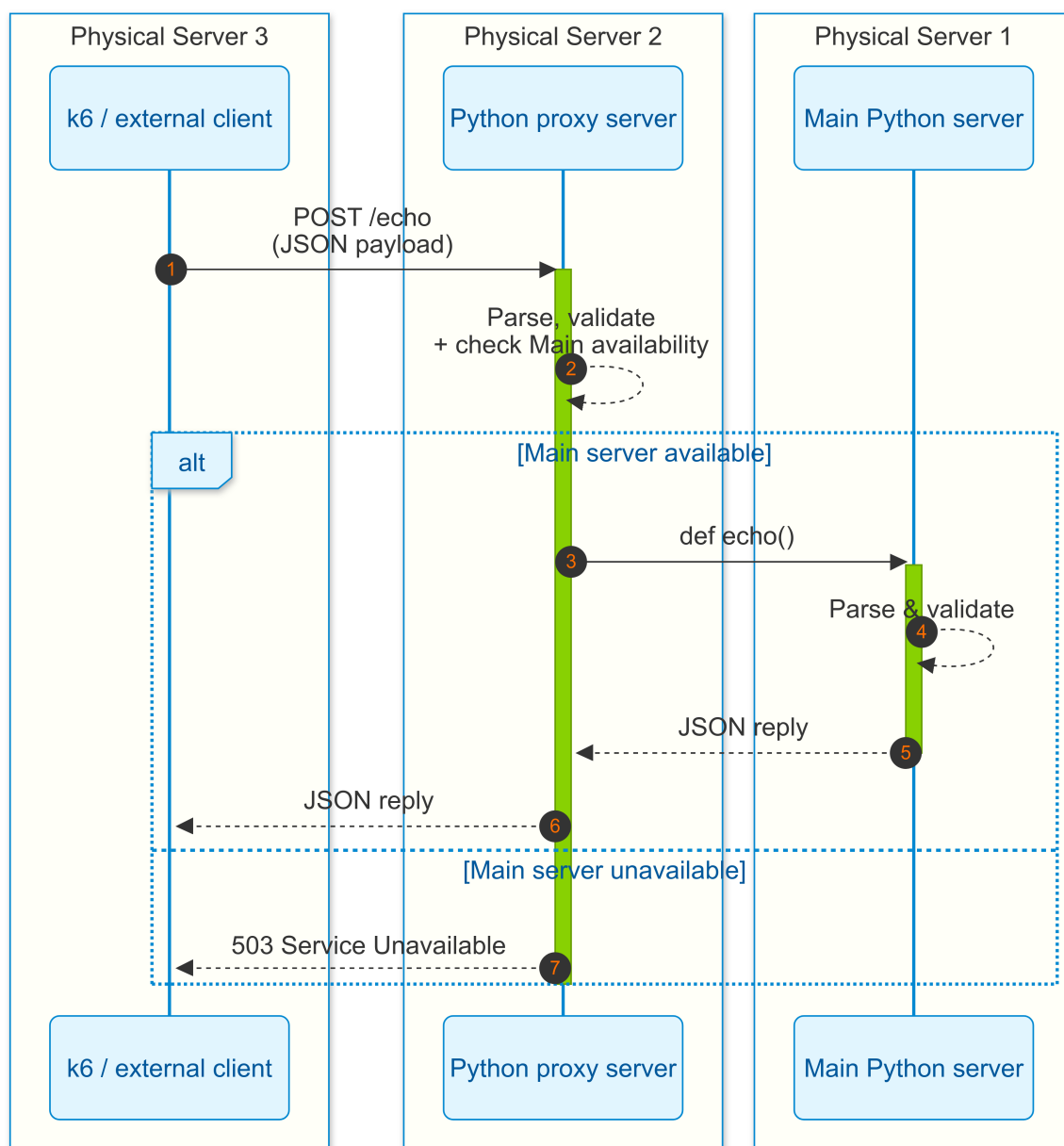


Рисунок 3.4 – Диаграмма последовательности версии без Istio

3.3 Выводы

В итоге проведённого проектирования и анализа можно заключить, что созданная модель микросервисной архитектуры учитывает все ключевые аспекты: от физической инфраструктуры и виртуальных машин в кластере k3s до логики взаимодействия сервисов в вариантах с Istio sidecar и без него. Подробно отражены потоки данных, маршрутизация и методы автоматизации развертывания, что формирует целостное представление о системе.

Разработанные диаграммы развертывания и последовательности подтвердили, что заложенные решения по управлению трафиком и сбору метрик (fluentbit, Jaeger) легко масштабируются и обеспечивают высокий уровень наблюдаемости. Таким образом, полученная модель является надёжной основой для дальнейшей реализации и проведения полноценных нагрузочных тестов.

4. Реализация и сравнительное тестирование системы.

В данном разделе представлено комплексное описание реализации и тестирования системы. Рассмотрен процесс программирования основных компонентов и их интеграции в общую архитектуру. Рассмотрены детали реализаций микросервисов, на которые будут приходить запросы. Объяснено как разворачиваются компоненты в кластере. Особое внимание уделено способу сбора трассировки, настройке сборщика данных и конфигурации Jaeger. Разъясняется конфигурация service mesh в Istio, в сравнении с самостоятельно реализованной альтернативой. Описаны методики тестирования, позволяющие оценить производительность и отказоустойчивость системы. Приводятся результаты экспериментов нагрузочного тестирования. Строятся, сравниваются и объясняются графики задержек в каждой из реализаций. Рассмотрены перспективы и возможности дальнейшего развития исследования.

4.1 Состав и структура реализованного программного обеспечения

Реализовано программное обеспечение, базирующееся на микросервисной архитектуре, развернутой в отдельном кластере виртуальных машин с использованием облегченной платформы Kubernetes - k3s. Для каждой из рассматриваемых версий системы (с применением Istio и без него) был создан собственный кластер k3s. Каждый кластер развернут на отдельных физических серверах. Каждая виртуальная машина в составе кластера конфигурировалась с выделением 2 процессорных ядер и 4 ГБ оперативной памяти. При этом кластер построен по схеме «один сервер — два агента»: серверная роль выполнялась первой виртуальной машиной, а две другие были агентами. Архитектура системы с Istio включает следующие компоненты:

- Главный сервер, реализованный на языке Python с использованием фреймворка Flask и запускаемый через Gunicorn для эффективного параллельного обслуживания запросов;
- Прокси сервер, также основанный на Python и Flask, на который запросы изначально поступают;
- Сервис трассировки Jaeger v2, включающий компоненты пользовательского интерфейса (Jaeger UI) и HTTP-сервиса для сбора и отображения трассировок;
- Компоненты Fluent Bit, развернутые в режиме DaemonSet на каждом узле для сбора и агрегации логов;

- Istio Sidecar, управляющий межсервисными коммуникациями и реализующий функции шаблона Circuit Breaker;
- средство нагрузочного тестирования K6, запускаемое в виде задания (Job) в кластере Kubernetes.

Версия системы без Istio сохраняет аналогичную структуру и состав компонентов, включая наличие sidecar контейнера Istio Sidecar и соответствующих механизмов ingress и egress, обеспечиваемых этим сервисом. Все компоненты системы упакованы в Docker контейнеры и размещены в открытом доступе на Docker Hub. Листинги Docker файлов для основного и прокси сервера приведены в приложении (см. А.3 и А.4). Для автоматизации процесса развертывания, конфигурирования и обновления компонентов используются Helm-чарты, параметры которых централизованно управляются посредством файла values.yaml (см. А.6). Такой подход позволяет гибко и эффективно управлять параметрами среды исполнения и автоматизировать операции развертывания системы

4.2 Основные сценарии использования реализованных решений

Библиотека Circuit Breaker была создана с целью повышения отказоустойчивости распределённых приложений, особенно в случаях, когда применение полнофункционального решения, такого как Istio, избыточно. Предлагаемая библиотека может легко интегрироваться в различные микросервисные архитектуры, позволяя эффективно управлять состоянием запросов и обрабатывать ошибки взаимодействия между компонентами системы. Особенностью библиотеки является её простота и минимальное влияние на производительность приложений, что подтверждается проведёнными испытаниями (см. главу 4.3). Применение данной библиотеки позволяет существенно ускорить работу микросервисных архитектур за счёт устранения излишней нагрузки на промежуточные прокси-сервисы. Помимо реализации Circuit Breaker была разработана удобная система трассировки, представленная JSON-экспортером, который пишет данные trace в стандартный поток вывода (stdout). Данный подход позволяет избежать лишних затрат процессорных ресурсов на передачу трассировочных данных напрямую и интегрироваться с такими инструментами, как Fluent Bit, для последующей обработки и анализа. Экспортер полностью совместим с протоколами OpenTelemetry и может быть легко добавлен в уже существующую инфраструктуру, использующую Jaeger, с минимальными изменениями в коде приложения:

```
from otlp_json_console_exporter import OTLPJsonConsoleExporter
json_exporter = OTLPJsonConsoleExporter()
provider.add_span_processor(BatchSpanProcessor(json_exporter))
```

Таким образом, предложенные решения не только обеспечивают простоту интеграции и использования паттерна Circuit Breaker, но и предоставляют эффективные инструменты для оценки и анализа задержек в существующих архитектурах. Полный листинг кода JSON-экспортера приведён в приложении (см. А.15).

4.3 Результаты тестирования

В рамках тестирования были проведены испытания системы согласно сценариям, описанным в разделе 1.4. В частности, выполнены следующие тесты:

- **Тест работоспособности библиотеки Circuit Breaker.** Проведено испытание с помощью инструмента k6 и визуализации в Lens, подтвердившее корректное срабатывание блокировки обращений после трёх неудачных попыток запросов к несуществующему адресу (HTTP 404) (см. 4.1). После достижения заданного порога система блокирует доступ к целевому серверу на 30 секунд, обеспечивая устойчивость к повторяющимся сбоям.

```
time="2025-06-10T11:36:27Z" level=info msg="SUCCESS [30]: Normal response received" source=console
time="2025-06-10T11:36:28Z" level=info msg="SUCCESS [31]: Normal response received" source=console

running (0m45.0s), 2/5 VUs, 130 complete and 0 interrupted iterations
normal_operation ✓ [ 100% ] 1 VUs 30s
trigger_failures [ 33% ] 2 VUs 15.0s/45s
test_circuit_open • [ 0% ] waiting 0m30.0s
test_recovery • [ 0% ] waiting 1m30.0s
time="2025-06-10T11:36:28Z" level=info msg="503 ERROR [32]: operation error: Connection refused by backend server (CB state: CLOSED)" source=console
time="2025-06-10T11:36:28Z" level=info msg="503 ERROR [33]: operation error: HTTP 500: Internal server error from backend (CB state: CLOSED)" source=console
time="2025-06-10T11:36:28Z" level=info msg="503 ERROR [34]: operation error: Connection timeout to backend server (CB state: OPEN)" source=console
time="2025-06-10T11:36:28Z" level=info msg="CIRCUIT BREAKER OPEN [32]: circuit breaker: open state blocking requests" source=console

running (0m46.0s), 2/5 VUs, 134 complete and 0 interrupted iterations
normal_operation ✓ [ 100% ] 1 VUs 30s
trigger_failures [ 36% ] 2 VUs 16.0s/45s
test_circuit_open • [ 0% ] waiting 0m29.0s
test_recovery • [ 0% ] waiting 1m29.0s
time="2025-06-10T11:36:29Z" level=info msg="CIRCUIT BREAKER OPEN [33]: circuit breaker: open state blocking requests" source=console
time="2025-06-10T11:36:29Z" level=info msg="CIRCUIT BREAKER OPEN [35]: circuit breaker: open state blocking requests" source=console
time="2025-06-10T11:36:29Z" level=info msg="CIRCUIT BREAKER OPEN [34]: circuit breaker: open state blocking requests" source=console
```

Рисунок 4.1 – Лог k6 об успешном применении Circuit Breaker

- **Нагрузочное тестирование с различным числом виртуальных пользователей (vus).** Проведены испытания с постепенным увеличением нагрузки, при котором количество пользователей варьировалось в пределах 5, 10, 25 и 50 vus. Каждый тест длился три минуты, состоял из трех этапов: постепенного нарастания нагрузки в течение первых 30 секунд, поддержание пикового уровня нагрузки в течение последующих двух минут и завершающий 30-секундный период постепенного снижения нагрузки.
- **Тестирование с увеличением размера передаваемого запроса.** Также проведено тестирование производительности при постепенном увеличении размера передаваемых запросов до 1, 5, 100 и 500 КБ при фиксированном количестве виртуальных пользователей (10 vus). Каждый тест также состоял из трёх этапов: начального нарастания нагрузки в течение 30 секунд, поддержания нагрузки на максимальном уровне в течение

ние двух минут и завершающего 30-секундного этапа плавного снижения нагрузки.

Для анализа задержек и оценки времени отклика использовалась система трассировки Jaeger, один trace которой состоит из 11 spans, представленных на диаграмме 4.2.

Trace состоит из следующих спанов:

- **root span (POST /echo)** – корневой span, обозначающий начальную точку обработки запроса.
- **echo_proxy_client** – обработка запроса на прокси-сервере.
- **circuit_breaker_call** – span, отражающий логику работы Circuit Breaker.
- **parse_request_client** – span, обозначающий разбор запроса на прокси.
- **redirect_to_server_client (POST /echo)** – перенаправление запроса на сервер.
- **construct_response_client** – формирование ответа клиенту на прокси.
- **echo_request_server** – принятие запроса сервером.
- **parse_request_server** – разбор запроса на сервере.
- **process_message_server** – непосредственная обработка сообщения на сервере.
- **construct_response_server** – формирование ответа сервером.

На диаграмме (см. 4.3) представлено формирование спанов в Python-коде и последующая их агрегация в базе данных Jaeger v2.

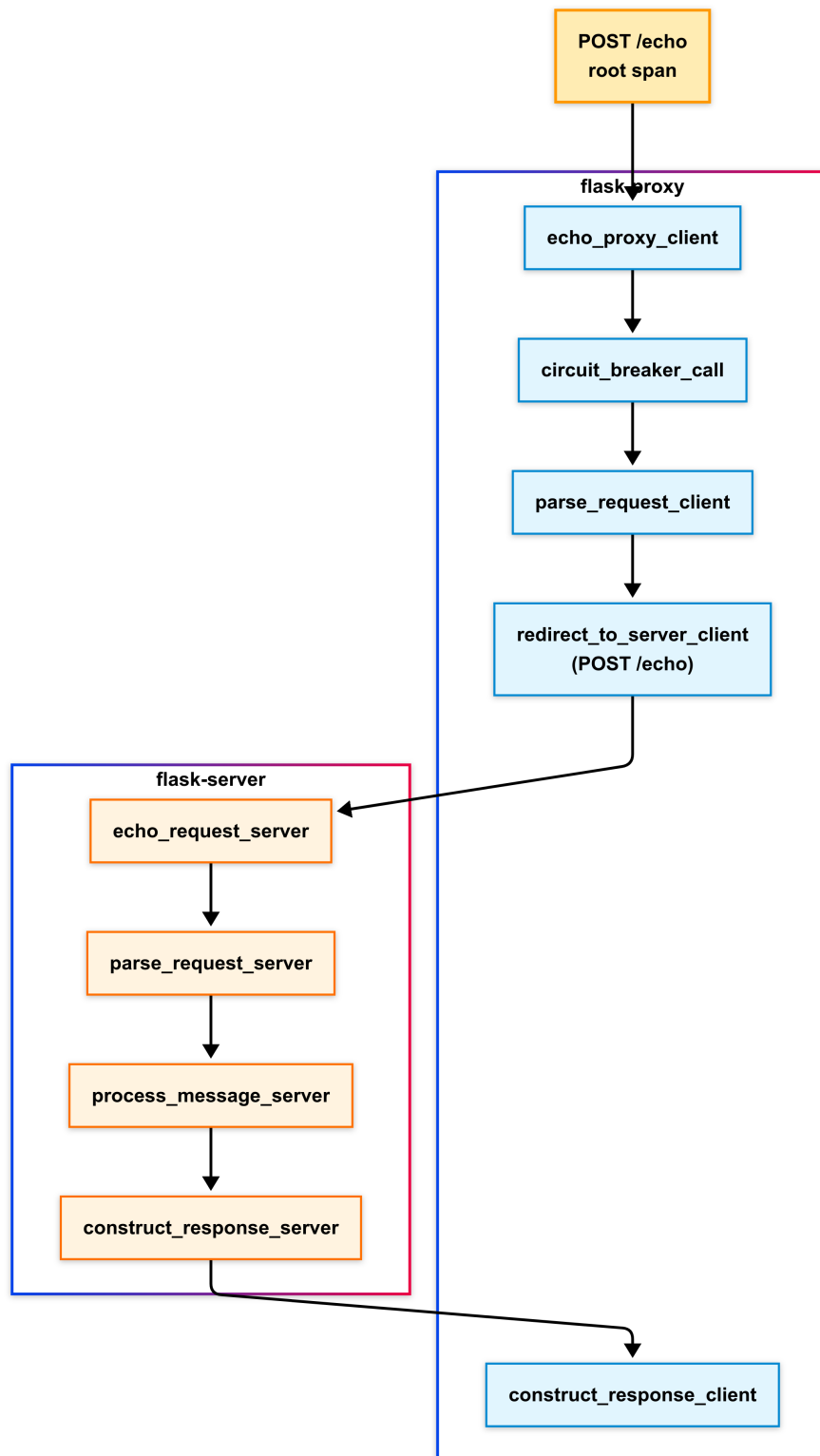


Рисунок 4.2 – Описание структуры trace.

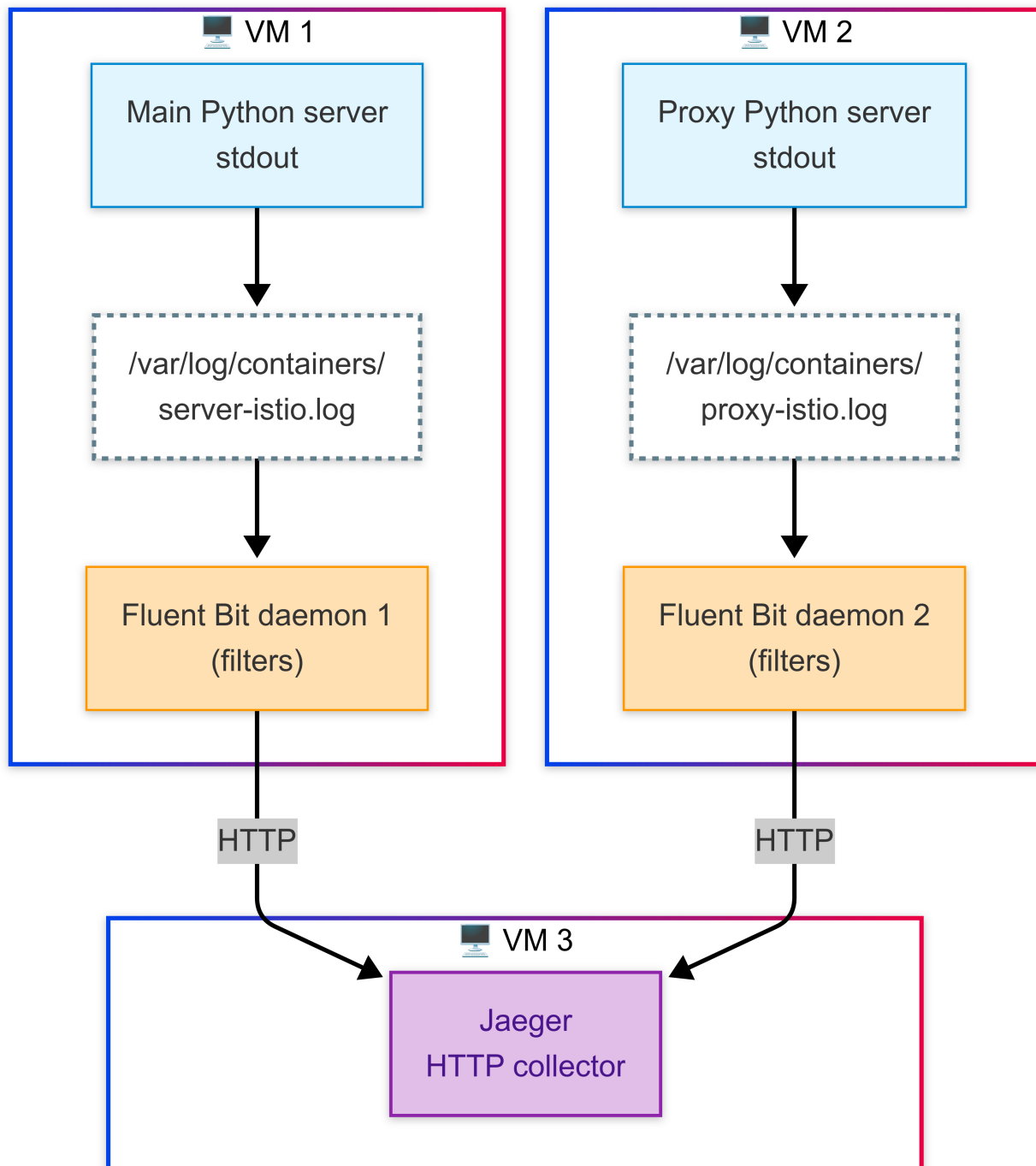


Рисунок 4.3 – Описание формирования trace.

По результатам нагрузочного тестирования построена диаграмма распределения задержек в зависимости от количества виртуальных пользователей (см. рис. 4.4). Данная диаграмма отражает сравнительную производительность реализации с использованием Istio и без него. На примере обработки 50 виртуальных пользователей видно, что сервер испытывает значительную нагрузку, и паттерн Circuit Breaker в этих условиях позволяет быстрее отвечать на запросы, в то время как собственная реализация демонстрирует значительно меньшие задержки. Более наглядное сравнение показывает столбчатая диаграмма (см. рис. 4.5), где реализация с Istio характеризуется примерно на 20% большей задержкой независимо от количества виртуальных пользователей.

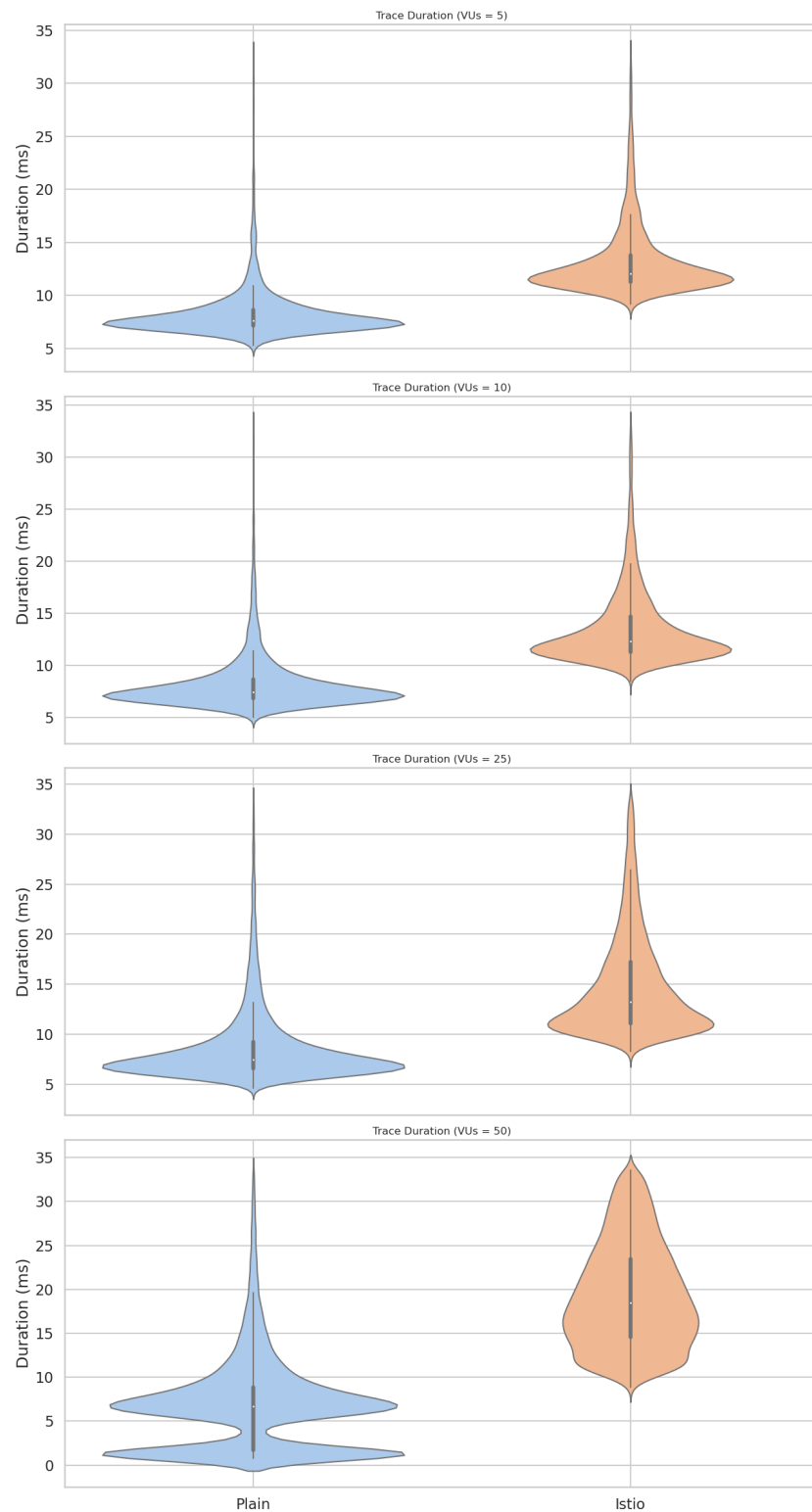


Рисунок 4.4 – Диаграмма распределения задержек в зависимости от количества виртуальных пользователей.

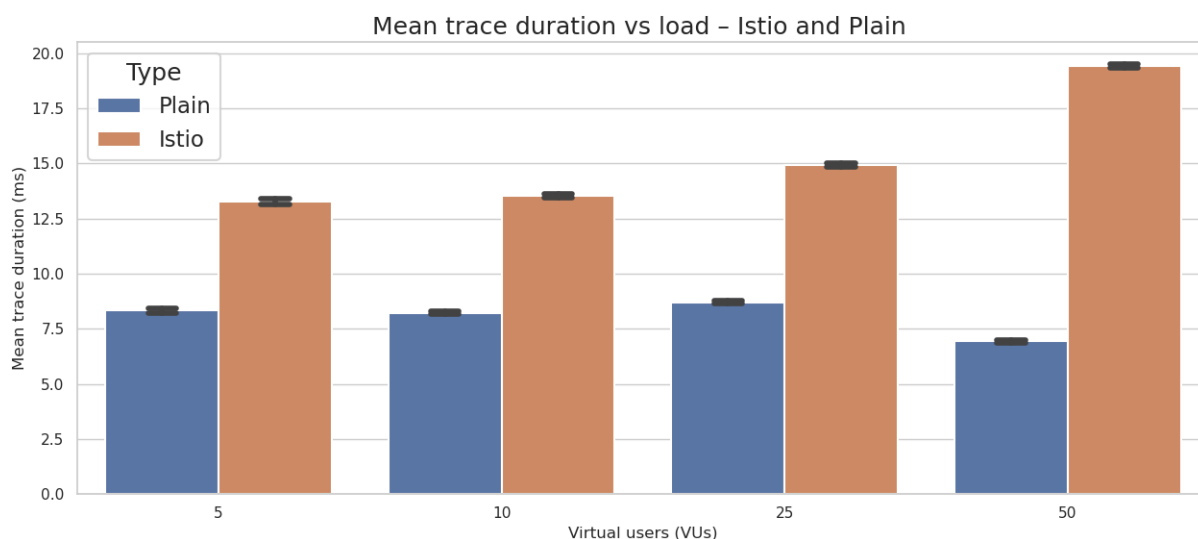


Рисунок 4.5 – Среднее время выполнения trace в зависимости от числа виртуальных пользователей.

Дополнительно проведены тесты с различным размером полезной нагрузки (payload). Эти тесты подтвердили работоспособность предложенного решения при любых объемах данных, а также показали незначительность влияния размера полезной нагрузки на относительную производительность обеих реализаций (см. рис. 4.6). На рисунке 4.7 представлена столбчатая диаграмма среднего времени выполнения отдельных спанов в тесте с payload размером 100 КБ. Заметно, что спаны, требующие сетевого взаимодействия, демонстрируют разницу в скорости между двумя реализациями, в то время как операции, выполняемые исключительно процессором, имеют одинаковую производительность в обоих случаях.

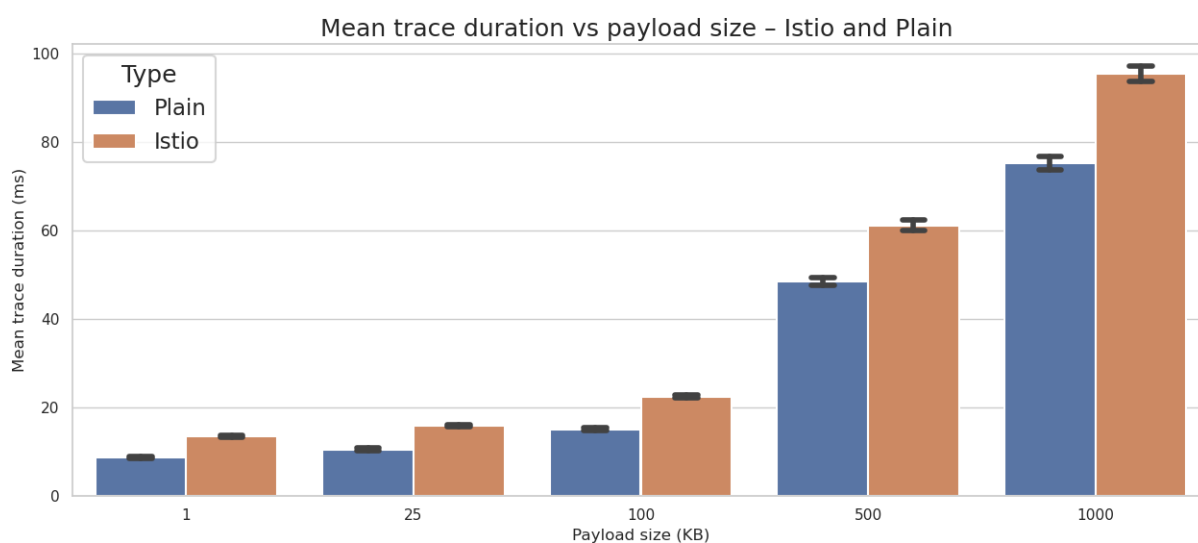


Рисунок 4.6 – Столбчатая диаграмма среднего времени спана в зависимости от размера запроса.

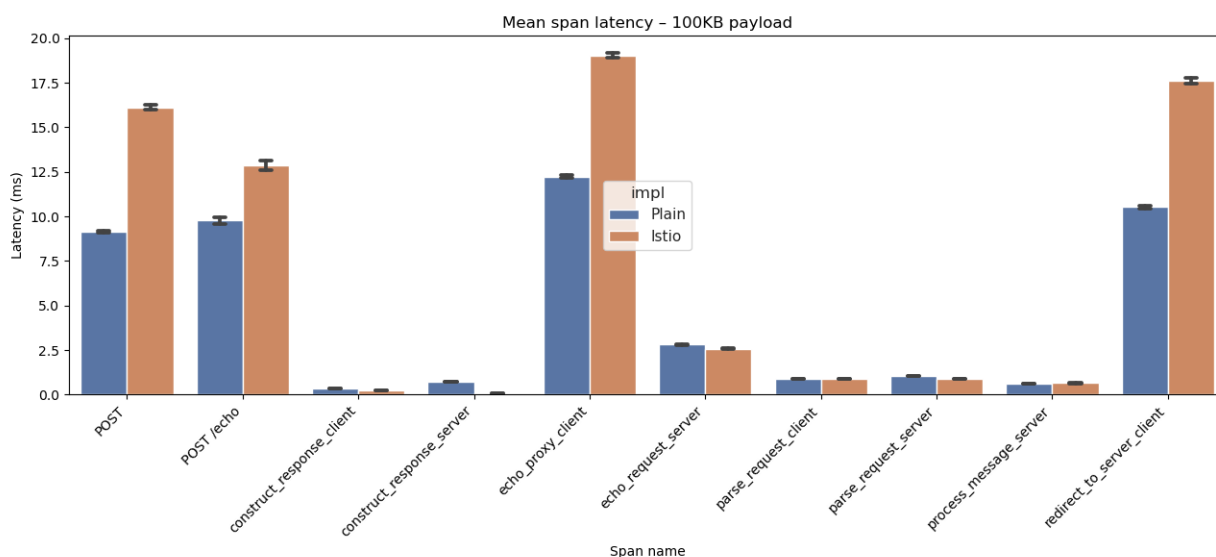


Рисунок 4.7 – Диаграмма среднего времени выполнения отдельных спанов для запросов с полезной нагрузкой 100 КБ.

4.4 Выводы

Проведённые испытания подтвердили преимущество кастомной реализации паттерна Circuit Breaker над реализацией на базе Istio с точки зрения сетевых задержек. Снижение задержек примерно на 20% делает пользовательское решение предпочтительным для приложений, чувствительных к производительности и сетевым взаимодействиям. При этом затраты времени на выполнение внутренних вычислений остаются одинаковыми, что свидетельствует о стабильности и предсказуемости обеих реализаций. Полученные данные позволяют рекомендовать разработанную библиотеку для использования в ситуациях, где важна минимизация задержек и требуется упрощённая инфраструктура без дополнительных накладных расходов, характерных для Service Mesh решений.

Предполагается дальнейшее развитие библиотеки за счёт интеграции других паттернов отказоустойчивости, а также её последующая интеграция в проект KubeSmart — интеллектуальную систему оркестрации Docker-контейнеров для платформы PaaS, разрабатываемую лабораторией искусственного интеллекта и больших данных НИЯУ МИФИ под руководством Ровнягина М. М.

Заключение

В рамках данной работы произведен анализ, разработка и реализация модели микросервисной архитектуры, с паттерном Circuit Breaker, распределённым трассированием и методами нагрузочного тестирования. Основной акцент исследования сделан на измерении задержек возникающих в такой системе, на сравнении готового решения (Istio) с самостоятельно реализованной библиотекой.

Результатами исследования являются:

- UML диаграммы, описывающие микросервисную архитектуру кластеров с service mesh и без нее.
- Разработка двух вариантов реализации паттерна Circuit Breaker. Первый вариант основан на использовании инфраструктурного решения (Istio), а второй – на программном подходе с использованием Python-библиотеки.
- Интеграция методов нагрузочного тестирования и распределённого трассирования. Применение инструмента k6 для моделирования нагрузки в сочетании с алгоритмами сбора и анализа трассировочных данных. Количественные оценки производительности микросервисов в различных режимах работы.
- Анализ эффективности и времени работы предложенных решений. Проведение сравнительного анализа разработанных подходов по критически важным параметрам позволит сформировать основу для дальнейшей оптимизации и практической реализации микросервисных архитектур.

Предполагаемая область применения результатов исследования охватывает распределённые вычислительные среды, в которых важными характеристиками являются высокая отказоустойчивость и производительность. Практическая значимость работы заключается в возможности проведения обоснованного анализа и выбора оптимального способа реализации паттерна – либо посредством использования service mesh, либо через разработку самописного решения для микросервисов. Данное исследование позволяет оценить рентабельность и эффективность различных методов до реальной эксплуатации системы.

В перспективе планируется расширение исследования за счёт изучения других паттернов и оценки альтернативных решений в области service mesh. Полученные результаты могут стать базой для создания нового поколения высокопроизводительных и отказоустойчивых микросервисных архитектур.

Список литературы

1. *Nygard M. T.* Release It! — 1st. — Pragmatic Bookshelf, 2007.
2. *Netflix I.* Hystrix Wiki. — URL: <https://github.com/Netflix/Hystrix/wiki/> (дата обр. 25.03.2025) ; Accessed on 25 March 2025.
3. *Shriram Rajagopalan L. R.* Introducing Istio. — 2017. — URL: <https://istio.io/latest/news/releases/0.x/introducing-istio/> (дата обр. 25.03.2025) ; Presentation at GlueCon 2017.
4. *Newman S.* Building Microservices: Designing Fine-Grained Systems. — 2nd. — O'Reilly Media, 2021.
5. *Punithavathy E., Priya N.* Auto retry circuit breaker for enhanced performance in microservice applications // Int. J. of Electrical and Computer Engineering. — 2024. — Vol. 14, no. 2. — P. 2274–2281.
6. *Shekhar G.* Microservices Design Patterns for Cloud Architecture // SSRG International Journal of Computer Science and Engineering. — 2024. — URL: <https://doi.org/10.14445/23488387/IJCSE-V11I9P101> ; [Электрон. ресурс]. Дата обращения: 24.03.2025.
7. *Nygard M. T.* Release It! Design and Deploy Production-Ready Software. — 2nd. — Pragmatic Bookshelf, 2018.
8. *Pethuru Raj Anupama Raman H. S.* Architectural Patterns. — Packt Publishing, 2017. — ISBN 9781787287495.
9. *Brendan Burns Joe Beda K. H., Evenson L.* Kubernetes: Up and Running, 3rd Edition. — O'Reilly Media, Inc., 2022. — ISBN 9781098110208.
10. *Méndez S.* Edge Computing Systems with Kubernetes. — Packt Publishing, 2022.
11. *Palavesam K. V., Krishnamoorthy M. V., S M A.* A Comparative Study of Service Mesh Implementations in Kubernetes for Multi-cluster Management // Journal of Advances in Mathematics and Computer Science. — 2025. — Янв. — Т. 40, № 1. — С. 1—16. — DOI: 10.9734/jamcs/2025/v40i11958. — URL: <https://journaljamcs.com/index.php/JAMCS/article/view/1958>.

12. *Farkiani B.* Service Mesh: Architectures, Applications, and Implementations. — 2022. — URL: https://www.cse.wustl.edu/~jain/cse574-22/ftp/svc_mesh/index.html; [Электрон. ресурс]. Дата обращения: 24.03.2025.
13. Performance Comparison of Service Mesh Frameworks: the mTLS Test Case / A. Bremler-Barr [et al.] // arXiv. — 2024. — arXiv: 2411.02267. — URL: <https://arxiv.org/abs/2411.02267>.
14. Gatling: Load testing designed for DevOps and CI/CD. — URL: <https://gatling.io/> (дата обр. 25.03.2025); Accessed on 25 March 2025.
15. Locust: Locust Load Testing Tool. — URL: <https://locust.io/> (дата обр. 25.03.2025); Accessed on 25 March 2025.
16. Synthetic Time Series for Anomaly Detection in Cloud Microservices / M. Allam [и др.]. — 2024. — arXiv: 2408.00006 [cs.DC]. — URL: <https://arxiv.org/abs/2408.00006>.
17. Grafana k6: Load testing for engineering teams. — URL: <https://k6.io/> (дата обр. 25.03.2025); Accessed on 25 March 2025.
18. *Aqasizade H., Ataie E., Bastam M.* Kubernetes in Action: Exploring the Performance of Kubernetes Distributions in the Cloud. — 2024. — arXiv: 2403.01429 [cs.DC]. — URL: <https://arxiv.org/abs/2403.01429>.
19. Design, monitoring, and testing of microservices systems: The practitioners' perspective / M. Waseem [и др.] // Journal of Systems and Software. — 2021. — Дек. — Т. 182. — С. 111061. — ISSN 0164-1212. — DOI: 10.1016/j.jss.2021.111061. — URL: <http://dx.doi.org/10.1016/j.jss.2021.111061>.
20. Jaeger: open source, distributed tracing platform. — URL: <https://www.jaegertracing.io/> (дата обр. 26.03.2025); Accessed on 26 March 2025.
21. Zipkin. — URL: <https://zipkin.io/> (дата обр. 26.03.2025); Accessed on 25 March 2026.
22. *Borges M. C., Werner S., Kilic A.* FaaS Troubleshooting - Evaluating Distributed Tracing Approaches for Serverless Applications // 2021 IEEE International Conference on Cloud Engineering (IC2E). — IEEE, 10.2021. — С. 83—90. — DOI: 10.1109/ic2e52221.2021.00022. — URL: <http://dx.doi.org/10.1109/IC2E52221.2021.00022>.

23. Vendor-agnostic way to receive, process and export telemetry data. — URL: <https://opentelemetry.io/docs/collector/> (дата обр. 27.03.2025) ; Accessed on 25 March 2027.

А. Приложение А. Листинг программной реализации

Реализация микросервисной архитектуры с Istio.

Листинг А.1 – Python код основного сервера

```
# server.py
import os
import time
import sys
import json
import threading
import collections

from flask import Flask, request, jsonify
from opentelemetry import trace
from opentelemetry.sdk.resources import Resource
from opentelemetry.sdk.trace import ReadableSpan, TracerProvider
from opentelemetry.sdk.trace.export import (
    SpanExporter,
    SpanExportResult,
    BatchSpanProcessor,
)
from opentelemetry.instrumentation.flask import FlaskInstrumentor
from opentelemetry.instrumentation.requests import RequestsInstrumentor
# Custom exporter
from otel_json_console_exporter import OTLPJsonConsoleExporter

# 2) Tracing setup

resource = Resource(attributes={"service.name": "flask-server"})
provider = TracerProvider(resource=resource)
trace.set_tracer_provider(provider)

json_exporter = OTLPJsonConsoleExporter()
# Use BatchSpanProcessor so that as soon as "echo_request_server" ends,
# 'exporters _flush()' writes the JSON of the entire trace to stdout.
provider.add_span_processor(BatchSpanProcessor(json_exporter))

# Previous gRPC to Jaeger attempt
# from opentelemetry.exporter.otlp.proto.grpc.trace_exporter import OTLPSpanEx
# jaeger_uri = os.getenv("JAEGER_ENDPOINT", "jaeger-otlp-service:4317")
# otlp_exporter = OTLPSpanExporter(endpoint=jaeger_uri, insecure=True)
# provider.add_span_processor(BatchSpanProcessor(otlp_exporter))

# 3) Flask + instrumentation
app = Flask(__name__)
FlaskInstrumentor().instrument_app(app)
RequestsInstrumentor().instrument()
tracer = trace.get_tracer(__name__)
```

```

@app.route("/echo", methods=["POST"])
def echo():
    with tracer.start_as_current_span("echo_request_server") as span:
        start_time = time.time()
        try:
            with tracer.start_as_current_span("parse_request_server"):
                data = request.get_json()
                message = data.get("message", "") if data else ""

            with tracer.start_as_current_span("process_message_server"):
                # Example application logs, written to stdout:
                print(f"[Server] Received message: {message}")

            with tracer.start_as_current_span("construct_response_server"):
                response_data = {
                    "message": f"Echo: {message}",
                    "timestamp": int(time.time())
                }

                duration = time.time() - start_time
                span.set_attribute("total.duration", duration)
                return jsonify(response_data)

        except Exception as e:
            print(f"[Server] ERROR: {e}", file=sys.stderr)
            span.record_exception(e)
            return jsonify({"error": str(e)}), 500

if __name__ == "__main__":

    print("[Server] Starting server on 0.0.0.0:5001 (stdout for logs + spans)")
    app.run(host="0.0.0.0", port=5001, debug=False)

```

Листинг A.2 – Python код прокси сервера

```

# client.py
import os
import time
import requests
import logging
import json
from flask import Flask, request, jsonify

from opentelemetry import trace
from opentelemetry.sdk.resources import Resource
from opentelemetry.sdk.trace import TracerProvider, ReadableSpan
from opentelemetry.sdk.trace.export import BatchSpanProcessor
# from opentelemetry.sdk.trace.export import SimpleSpanProcessor
from opentelemetry.instrumentation.flask import FlaskInstrumentor
from opentelemetry.instrumentation.requests import RequestsInstrumentor

# — custom console exporter (exactly the same file you use on the server) —
from otlp_json_console_exporter import OTLPJsonConsoleExporter

```

```

logging.basicConfig(level=logging.INFO)
logger = logging.getLogger("flask-proxy")

# -----
# Tracing setup: print every finished process-root trace as one JSON line
# -----
resource = Resource(attributes={"service.name": "flask-proxy"})
provider = TracerProvider(resource=resource)
trace.set_tracer_provider(provider)

# □ send every span to the pod log (Fluent Bit reads it later)
json_exporter = OTLPJsonConsoleExporter()

# ↓ change here: use BatchSpanProcessor so that each "root" span flushes instantly
provider.add_span_processor(BatchSpanProcessor(json_exporter))
# provider.add_span_processor(SimpleSpanProcessor(json_exporter))

app = Flask(__name__)
FlaskInstrumentor().instrument_app(app)
RequestsInstrumentor().instrument()
tracer = trace.get_tracer(__name__)

@app.route("/echo", methods=["POST"])
def proxy_echo():
    with tracer.start_as_current_span("echo_proxy_client") as span:
        start_time = time.time()
        try:
            with tracer.start_as_current_span("parse_request_client"):

                real_server_url = f'http://{os.getenv("SERVER_SERVICE")}:5001/echo'
                json_data = request.get_json()

                with tracer.start_as_current_span("redirect_to_server_client"):

                    response = requests.post(real_server_url, json=json_data, timeout=3)
                    response.raise_for_status()

                with tracer.start_as_current_span("construct_response_client"):
                    response_json = response.json()

            span.set_attribute("response_time", time.time() - start_time)
            return response_json

        except Exception as exc:
            span.record_exception(exc)
            span.add_event("Error during request forwarding")
            logger.error("Error forwarding request: %s", exc)
            return jsonify({"error": str(exc)}), 503

if __name__ == "__main__":

    # Make Flask listen on 0.0.0.0:8001 so that K3s' Service (port 8001) can re

```

```
app.run(host="0.0.0.0", port=8001)
```

Листинг A.3 – Полный Dockerfile сборки сервера

```
FROM python:3.12
WORKDIR /home/hpc/aleksandr
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt
COPY server.py .
COPY otlp_json_console_exporter.py .
EXPOSE 5001
# CMD ["python", "server.py"]
CMD ["gunicorn", "-w", "4", "-k", "gthread", "-t", "30", "-b", "0.0.0.0:5001",
```

Листинг A.4 – Полный Dockerfile сборки прокси сервера

```
FROM python:3.11
WORKDIR /home/hpc/aleksandr

COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt

COPY client.py .
COPY otlp_json_console_exporter.py .

EXPOSE 8001
# CMD ["python", "client.py"]
CMD ["gunicorn", "-w", "4", "-k", "gthread", "-t", "30", "-b", "0.0.0.0:8001",
```

Листинг A.5 – Описание Helm chart

```
apiVersion: v2
name: flask-server
description: A Helm chart for deploying a Python Flask server with Istio sidecar
type: application
version: 0.1.0
appVersion: "1.0"
```

Листинг A.6 – Описание переменных в helm

```
server:
  replicaCount: 1
  image:
    repository: "robocatt/flask-server-istio"
    tag: "v1.0.50"
    pullPolicy: IfNotPresent

  service:
    type: ClusterIP
    port: 5001

  affinity:
    nodeAffinity:
```

```

        requiredDuringSchedulingIgnoredDuringExecution:
          nodeSelectorTerms:
            - matchExpressions:
                - key: server
                  operator: In
                  values:
                    - "true"

env:
  jaegerEndpoint: "jaeger-otlp-service:4317"
resources: {}
nodeSelector: {}
tolerations: []

client:
  replicaCount: 1
  image:
    repository: "robocatt/flask-client-istio"
    tag: "v1.0.20"
    pullPolicy: IfNotPresent
  service:
    type: ClusterIP
    port: 8001

affinity:
  nodeAffinity:
    requiredDuringSchedulingIgnoredDuringExecution:
      nodeSelectorTerms:
        - matchExpressions:
            - key: worker
              operator: In
              values:
                - "true"

resources: {}
nodeSelector: {}
tolerations: []

jaeger:
  name: jaeger
  image:
    repository: jaegertracing/jaeger
    tag: latest
    pullPolicy: IfNotPresent

uiService:
  name: jaeger-ui
  type: NodePort
  port: 16686
  targetPort: 16686
  nodePort: 31687

internalService:
  name: jaeger-otlp
  type: ClusterIP

```

```

    port: 4317
    targetPort: 4317

    httpService:
      name: jaeger-http
      type: ClusterIP
      port: 4318
      targetPort: 4318

```

Листинг A.7 – Описание сервиса серверов

```

{{- $releaseName := .Release.Name -}}
{{- $services := dict "server" .Values.server "client" .Values.client }}

{{- range $key, $val := $services }}
---
apiVersion: v1
kind: Service
metadata:
  name: {{ $releaseName }}-flask-{{ $key }}-istio
  labels:
    app: flask-{{ $key }}-istio
spec:
  type: {{ $val.service.type }}
  ports:
    - name: http
      port: {{ $val.service.port }}
      targetPort: {{ $val.service.port }}
      protocol: TCP
  selector:
    app: flask-{{ $key }}-istio
{{- end }}

```

Листинг A.8 – Описание сервиса Jaeger

```

{{- $jaegerServices := dict "ui" .Values.jaeger.uiService "internal" .Values.j
{{- range $key, $svc := $jaegerServices }}
---
apiVersion: v1
kind: Service
metadata:
  name: {{ $svc.name }}-service
  labels:
    app: jaeger
spec:
  type: {{ $svc.type }}
  ports:
    - name: {{ $key }}
      port: {{ $svc.port }}
      targetPort: {{ $svc.targetPort }}
      {{- if eq $key "ui" }}
      nodePort: {{ $svc.nodePort }}
      {{- end }}
  selector:
    app: jaeger

```



```
{{- end }}
```

Листинг А.9 – Описание пространства имен

```
apiVersion: v1
kind: Namespace
metadata:
  name: logging
```

Листинг А.10 – Описание FluentBit DaemonSet

```
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: fluent-bit
  namespace: logging
  labels:
    app: fluent-bit
spec:
  selector:
    matchLabels:
      app: fluent-bit
  template:
    metadata:
      labels:
        app: fluent-bit
    spec:
      serviceAccountName: default
      volumes:
        - name: config-volume
          configMap:
            name: fluent-bit-config
        - name: varlog
          hostPath:
            path: /var/log
        - name: pos
          emptyDir: {}
      containers:
        - name: fluent-bit
          image: fluent/fluent-bit:latest
          args: ["-c", "/fluent-bit/etc/fluent-bit.conf"]
          resources:
            requests:
              cpu: 50m
              memory: 100Mi
            limits:
              cpu: 200m
              memory: 200Mi
          volumeMounts:
            - name: config-volume
              mountPath: /fluent-bit/etc
            - name: varlog
              mountPath: /var/log
              readOnly: true
            - name: pos
```

```
mountPath: /fluent-bit/pos
```

Листинг А.11 – Описание FluentBit конфига

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: fluent-bit-config
  namespace: logging
data:

  fluent-bit.conf: |-
    [SERVICE]
      Flush          1
      Log_Level      info
      Parsers_File   parsers.conf

    #-----
    # 1) INPUT – tail only flask-server Istio side-car logs
    #-----
    [INPUT]
      Name            tail
      Tag             otlp.span

    Path              /var/log/containers/*_default_flask-*_istio-*.log
      Parser          cri
      DB              /fluent-bit/pos/flask-server-istio.db
      Read_from_Head  True
      Mem_Buf_Limit   5MB
      Skip_Long_Lines On

    #-----
    # 2) FILTER – keep stdout lines whose JSON starts with "{"
    #-----
    [FILTER]
      Name            record_modifier
      Match           otlp.span
      Allowlist_key   log

    # FILTER – move JSON into 'body' + build headers
    [FILTER]
      Name            lua
      Match           otlp.span
      Script           j2h.lua
      Call            add_body_and_headers

    # OUTPUT – POST one span per request to Jaeger OTLP/HTTP
    [OUTPUT]
      Name            http
      Match           otlp.span
      Host            jaeger-http-service.default.svc.cluster.local
      Port            4318
      URI             /v1/traces
      Body_key        $body
      Headers_key     $headers
```

```

    Log_Response_Payload  True
    allow_duplicated_headers  False
    workers                2

#-----
# 4-b) OUTPUT – copy the same record to stdout for debugging
#-----
[OUTPUT]
    Name      stdout
    Match     otlp.span

parsers.conf: |-
    [PARSER]
        Name      cri
        Format     regex

    Regex      ^(?<time>[^\ ]+) (?<stream>stdout|stderr) (?<logtag>[^\ ]*) (?<logspan>[^\ ]*)
    Time_Key    time
    Time_Format %Y-%m-%dT%H:%M:%S.%L%z
    Time_Keep   On

j2h.lua: |-
    function add_body_and_headers(tag, ts, record)
        local msg = record["log"]
        if msg == nil then
            return 0, ts, record
        end

        -- If the first character is "{", treat as JSON span. Otherwise drop.
        if string.sub(msg, 1, 1) == "{" then
            -- Copy the raw JSON into "body"
            record["body"] = msg

            -- Build the headers map

            record["headers"] = { ["Content-Type"] = "application/json" }

            -- Remove "log" so that only "body" + "headers" remain
            record["log"] = nil
            return 1, ts, record
        end

        -- Otherwise, not JSON → drop
        return 0, ts, record
    end
end

```

Листинг A.12 – Описание Circuit Breaker в Istio

```

apiVersion: networking.istio.io/v1beta1
kind: DestinationRule
metadata:
  name: {{ .Release.Name }}-flask-server-cb
  labels:
    app: flask-server-istio
spec:

```

```

host: {{ .Release.Name }}-flask-server-istio
trafficPolicy:
  connectionPool:
    tcp:

maxConnections: 1 # Limits Envoy to open a single connection to the upstream
http:

http1MaxPendingRequests: 0 # Disallow queuing of concurrent requests

maxRequestsPerConnection: 1 # Each connection only handles one request
outlierDetection:
  consecutiveGatewayErrors: 3
  consecutive5xxErrors: 3
  interval: 10s
  baseEjectionTime: 30s
  maxEjectionPercent: 100

```

Листинг A.13 – Описание deployment серверов

```

{{- $releaseName := .Release.Name -}}

{{- $deployments := dict "server" .Values.server "client" .Values.client }}

{{- range $key, $val := $deployments }}
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: {{ $releaseName }}-flask-{{ $key }}-istio
  labels:
    app: flask-{{ $key }}-istio
spec:
  replicas: {{ $val.replicaCount }}
  selector:
    matchLabels:
      app: flask-{{ $key }}-istio
  template:
    metadata:
      labels:
        app: flask-{{ $key }}-istio
      annotations:
        sidecar.istio.io/inject: "true"
    spec:
      containers:
        - name: flask-{{ $key }}-istio
          image: "{{ $val.image.repository }}:{{ $val.image.tag }}"
          imagePullPolicy: {{ $val.image.pullPolicy }}
          ports:
            - containerPort: {{ $val.service.port }}
          env:
            {{- if eq $key "client" }}

# Only include the SERVER_SERVICE env variable for the client deployment.
            - name: SERVER_SERVICE
              value: "{{ $releaseName }}-flask-server-istio"

```

```

        {{- end }}
    resources:
{{ toYaml $val.resources | indent 10 }}
    nodeSelector:
{{ toYaml $val.nodeSelector | indent 8 }}
    tolerations:
{{ toYaml $val.tolerations | indent 8 }}
    affinity:
{{ toYaml $val.affinity | indent 8 }}
{{- end }}

```

Листинг A.14 – Описание deployment Jaeger

```

apiVersion: apps/v1
kind: Deployment
metadata:
  name: {{ .Values.jaeger.name }}-deployment
  labels:
    app: jaeger
spec:
  replicas: 1
  selector:
    matchLabels:
      app: jaeger
  template:
    metadata:
      labels:
        app: jaeger
    spec:
      affinity:
        nodeAffinity:
          requiredDuringSchedulingIgnoredDuringExecution:
            nodeSelectorTerms:
              - matchExpressions:
                  - key: load
                    operator: In
                    values:
                      - "true"
      containers:
        - name: jaeger

image: "{{ .Values.jaeger.image.repository }}:{{ .Values.jaeger.image.tag }}"
imagePullPolicy: {{ .Values.jaeger.image.pullPolicy }}
volumeMounts:
  - name: config-volume
    mountPath: /jaeger/etc
args:
  - --config=/jaeger/etc/jaeger-conf.yaml
# args :
# - --collector.otlp.enabled=true
# - --collector.otlp.grpc.host-port=:4317 # grpc var
# - --collector.http-server.host-port=:4318 # old http?
# - --collector.otlp.http.host-port=:4318

# https://www.jaegertracing.io/docs/1.67/cli/#jaeger-collector

```

```

# env:
#   - name: SPAN_STORAGE_TYPE
#     value: "memory"
#   - name: SAMPLING_TYPE
#     value: "const"
#   - name: SAMPLING_PARAM
#     value: "1"
ports:
- name: ui
  containerPort: {{ .Values.jaeger.uiService.targetPort }}
- name: otlp

containerPort: {{ .Values.jaeger.internalService.targetPort }}
- name: http
  containerPort: {{ .Values.jaeger.httpService.targetPort }}
volumes:
- name: config-volume
  configMap:
    name: jaeger-config

```

Листинг A.15 – Python код экспортера OTLP trace в stdout

```

import threading
import collections
import sys
import json

from opentelemetry.sdk.trace import ReadableSpan
from opentelemetry.sdk.trace.export import SpanExporter, SpanExportResult
def _enc(v):
    if isinstance(v, bool):
        return {"boolValue": v}
    if isinstance(v, int):
        return {"intValue": str(v)}
    if isinstance(v, float):
        return {"doubleValue": v}

    return {"stringValue": str(v)}

def _span_json(s: ReadableSpan):
    parent_id = ""
    if s.parent and s.parent.is_valid and s.parent.span_id != 0:
        parent_id = f"{s.parent.span_id:016x}"
    return {
        "traceId": f"{s.context.trace_id:032x}",
        "spanId": f"{s.context.span_id:016x}",
        "parentSpanId": parent_id,
        "name": s.name,
        "kind": int(s.kind.value),
        "startTimeUnixNano": str(s.start_time),
        "endTimeUnixNano": str(s.end_time),
        "status": {
            "code": int(s.status.status_code.value),
            "message": s.status.description or "",
        },
    },

```

```

    "attributes": [{"key": k, "value": _enc(v)} for k, v in s.attributes.items()]
}

```

```

class OTLPJsonConsoleExporter(SpanExporter):
    """
    Emits one OTLP/JSON line per process-root trace.
    Works with SimpleSpanProcessor or BatchSpanProcessor.
    """
    def __init__(self):
        self._lock = threading.Lock()

    self._buffer = collections.defaultdict(list)  # trace_id -> list[ReadableSpan]

    def export(self, spans):
        with self._lock:
            for span in spans:
                tid = span.context.trace_id
                self._buffer[tid].append(span)

                parent = span.parent

            if parent is None or not parent.is_valid or parent.is_remote:
                self._flush(tid)

            return SpanExportResult.SUCCESS

    def shutdown(self):
        with self._lock:
            for tid in list(self._buffer.keys()):
                self._flush(tid)

    def _flush(self, tid: int):
        batch = self._buffer.pop(tid, [])
        if not batch:
            return

        buckets = collections.defaultdict(lambda: collections.defaultdict(list))
        for s in batch:
            r_key = tuple(sorted(s.resource.attributes.items()))

            s_key = (s.instrumentation_scope.name, s.instrumentation_scope.version)
            buckets[r_key][s_key].append(s)

        payload = {"resourceSpans": []}
        for r_key, scopes in buckets.items():
            payload["resourceSpans"].append({
                "resource": {
                    "attributes": [

{"key": k, "value": _enc(v)} for (k, v) in r_key
                    ]
                },
                "scopeSpans": [
                    {

"scope": {"name": name, "version": version or ""},

```

```

        "spans": [_span_json(sp) for sp in span_list],
    }
    for (name, version), span_list in scopes.items()
],
}))

# Write exactly one JSON object (no extra newlines, no prefix)
sys.stdout.write(json.dumps(payload, separators=(",", ":")) + "\n")
sys.stdout.flush()

```

Реализация микросервисной архитектуры без Istio во многом похожа. Приведены лишь отличающиеся файлы.

Листинг А.16 – Python код основного сервера

```

import time
import os
from flask import Flask, request, jsonify

from opentelemetry import trace
from opentelemetry.sdk.resources import Resource
from opentelemetry.sdk.trace import TracerProvider
from opentelemetry.sdk.trace.export import BatchSpanProcessor
from opentelemetry.instrumentation.flask import FlaskInstrumentor
from opentelemetry.instrumentation.requests import RequestsInstrumentor
# from opentelemetry.exporter.otlp.proto.grpc.trace_exporter import OTLPSpanEx
from otlp_json_console_exporter import OTLPJsonConsoleExporter

# otlp_endpoint = os.getenv("OTLP_COLLECTOR_ENDPOINT", "collector-endpoint:431
resource = Resource(attributes={"service.name": "flask-server"})
provider = TracerProvider(resource=resource)
trace.set_tracer_provider(provider)

# otlp_exporter = OTLPSpanExporter(endpoint=otlp_endpoint, insecure=True)
# provider.add_span_processor(BatchSpanProcessor(otlp_exporter))
json_exporter = OTLPJsonConsoleExporter()
provider.add_span_processor(BatchSpanProcessor(json_exporter))

app = Flask(__name__)
FlaskInstrumentor().instrument_app(app)
RequestsInstrumentor().instrument()

# tracer instance to manually create spans
tracer = trace.get_tracer(__name__)

@app.route('/echo', methods=['POST'])
def echo():
    # Root span for the entire echo request.
    with tracer.start_as_current_span("echo_request_server") as span:
        start_time = time.time()

        # Span for parsing the incoming JSON payload.

        with tracer.start_as_current_span("parse_request_server") as parse_span:
            data = request.get_json()

```



```

        message = data.get("message", "") if data else ""
        parse_span.set_attribute("message.length", len(message))

    # Span for processing the message.

    with tracer.start_as_current_span("process_message_server") as process_span:
        print(f"[Server] Received message from client: {message}")
        # time.sleep(0.1)
        process_span.set_attribute("message.logged", True)

    # if random.random() < 0.25:
    #     print("[Server] Simulated failure!")
    #     return "Internal Server Error", 500

    # Span for constructing the response.

    with tracer.start_as_current_span("construct_response_server") as construct_span:
        response_data = {
            "message": f"Echo: {message}",
            "timestamp": int(time.time()),
        }

    construct_span.set_attribute("response.size", len(str(response_data)))
    response = jsonify(response_data)

    total_duration = time.time() - start_time
    span.set_attribute("total.duration", total_duration)
    return response

if __name__ == "__main__":
    print("[Server] Starting with OTel tracing on port 5001 ...")
    app.run(host="0.0.0.0", port=5001, debug=True)

```

Листинг A.17 – Python код прокси сервера

```

import os
import time
import requests
from flask import Flask, request, jsonify
from opentelemetry import trace
from opentelemetry.sdk.resources import Resource
from opentelemetry.sdk.trace import TracerProvider
from opentelemetry.sdk.trace.export import BatchSpanProcessor
from opentelemetry.instrumentation.flask import FlaskInstrumentor
from opentelemetry.instrumentation.requests import RequestsInstrumentor
from otel_json_console_exporter import OTLPJsonConsoleExporter

from circuit_breaker import CircuitBreaker

# ----- OpenTelemetry plumbing
resource = Resource.create(attributes={"service.name": "circuit-breaker-proxy"})
trace.set_tracer_provider(TracerProvider(resource=resource))
trace.get_tracer_provider().add_span_processor(
    BatchSpanProcessor(OTLPJsonConsoleExporter())
)

```

```

# ----- Flask app
app = Flask(__name__)
FlaskInstrumentor().instrument_app(app)
RequestsInstrumentor().instrument()

breaker = CircuitBreaker(failure_threshold=3, open_timeout=30)
tracer = trace.get_tracer(__name__)

# ----- Routes
@app.route("/echo", methods=["POST"])
def proxy_echo():
    with tracer.start_as_current_span("echo_proxy_client") as span:
        start = time.time()

        def forward():

server_service = os.getenv("SERVER_SERVICE", "flask-server")
        real_url = f"http://{server_service}:5001/echo"

        resp = requests.post(real_url, json=request.get_json(), timeout=3)
        if resp.status_code >= 400:

            raise RuntimeError(f"HTTP {resp.status_code}: {resp.text}")
            return resp

        try:
            resp = breaker.call(forward)
            span.set_attribute("proxy_echo.status", "ok")
            return resp.json()
        except Exception as exc:
            span.record_exception(exc)
            span.set_attribute("proxy_echo.status", "error")
            return jsonify({"error": str(exc)}), 503

@app.route("/test-failure", methods=["POST"])
def test_failure_endpoint():
    with tracer.start_as_current_span("test_failure"):

        def always_fail():
            raise RuntimeError("simulated backend failure")

        breaker.call(always_fail) # always raises & counts as failure
        # 'Well never reach here
        return jsonify({"msg": "unexpected"})

@app.errorhandler(404)
def catch_all(_):
    return jsonify({"error": "endpoint not found"}), 404

if __name__ == "__main__":
    print("[Proxy] starting on :8001")
    app.run(host="0.0.0.0", port=8001, debug=True)

```

Листинг A.18 – Python код Circuit Breaker библиотеки

```
"""
Reusable circuit-breaker implementation with OpenTelemetry tracing.

Install locally (pip install -e .) or add the folder to PYTHONPATH and
import with:

    from circuit_breaker import CircuitBreaker
"""

import time
from opentelemetry import trace

__all__ = ["CircuitBreakerState", "CircuitBreaker"]

class CircuitBreakerState:
    CLOSED = "CLOSED"
    OPEN = "OPEN"
    HALF_OPEN = "HALF_OPEN"

class CircuitBreaker:
    """
    A simple, thread-unsafe circuit breaker.

    Parameters
    -----
    failure_threshold : int
        Number of consecutive failures before opening the circuit.
    open_timeout : int | float
        Seconds to wait before moving from OPEN -> HALF_OPEN.
    """

    def __init__(self, *, failure_threshold: int = 3, open_timeout: int = 10):
        self.state = CircuitBreakerState.CLOSED
        self.failure_count = 0
        self.failure_threshold = failure_threshold
        self.open_timeout = open_timeout
        self.next_attempt_time: float | None = None
        self.tracer = trace.get_tracer(__name__)

    # ----- helper
    def _handle_success(self, span):
        if self.state == CircuitBreakerState.HALF_OPEN:
            span.set_attribute("cb.state_transition", "HALF_OPEN → CLOSED")
            self.state, self.failure_count = CircuitBreakerState.CLOSED, 0
            span.set_attribute("cb.new_state", self.state)

    def _handle_failure(self, span, exc: Exception):
        self.failure_count += 1
        span.set_attribute("cb.failure_count", self.failure_count)

        if self.state == CircuitBreakerState.HALF_OPEN:
```

```

        # probe failed -> reopen
        self.state = CircuitBreakerState.OPEN
        self.next_attempt_time = time.time() + self.open_timeout

span.set_attribute("cb.state_transition", "HALF_OPEN → OPEN")
    elif (
        self.state == CircuitBreakerState.CLOSED
        and self.failure_count >= self.failure_threshold
    ):
        self.state = CircuitBreakerState.OPEN
        self.next_attempt_time = time.time() + self.open_timeout
        span.set_attribute("cb.state_transition", "CLOSED → OPEN")

# ----- API
def call(self, func, *args, **kwargs):
    """
    Invoke *func* under circuit-breaker protection.

    Any exception raised by *func* is wrapped and re-raised
    so callers can handle it uniformly.
    """

    with self.tracer.start_as_current_span("circuit_breaker_call") as span:
        span.set_attribute("cb.current_state", self.state)

        # Guard: circuit is OPEN?
        if self.state == CircuitBreakerState.OPEN:
            if time.time() < (self.next_attempt_time or 0):

span.set_attribute("cb.request_status", "blocked (OPEN)")
                raise RuntimeError("Circuit breaker is OPEN")
                # timeout expired □ probe in HALF-OPEN
                self.state = CircuitBreakerState.HALF_OPEN
                self.failure_count = 0

span.set_attribute("cb.state_transition", "OPEN -> HALF_OPEN")

        # Try the protected call
        try:
            result = func(*args, **kwargs)
        except Exception as exc:
            self._handle_failure(span, exc)
            span.record_exception(exc)
            raise RuntimeError(
                f"operation error ({self.state}): {exc}"
            ) from exc
        else:
            self._handle_success(span)
            return result

```

Листинг A.19 – Python код инициализации Circuit Breaker библиотеки

```

from .cb_lib_core import CircuitBreaker, CircuitBreakerState

__all__ = ["CircuitBreaker", "CircuitBreakerState"]

```

Листинг A.20 – Конфигурация k6 для нагрузочного тестирования

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: k6-script
data:
  loadtest.js: |
    import http from 'k6/http';
    import { check, sleep } from 'k6';

    export const options = {
      stages: [
        { duration: '0.5m', target: 25 }, // ramp-up
        { duration: '2m', target: 25 }, // sustain
        { duration: '0.5m', target: 0 }, // ramp-down
      ],
      thresholds: {
        http_req_failed: ['rate<0.01'],
        http_req_duration: ['p(95)<500'],
      },
    };

    const BASE_URL = 'http://release2-flask-client-istio.default.svc.cluster.local';

    const PARAMS = { headers: { 'Content-Type': 'application/json' }, timeout: 1000 };

    export default function () {
      const res = http.post(
        BASE_URL,
        JSON.stringify({ message: 'Latency test' }),
        PARAMS,
      );

      check(res, { 'status 200': (r) => r.status === 200 });
      sleep(Math.random() * 0.5);
    }
```

Листинг A.21 – Конфигурация k6 для тестирования payload

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: k6-script
data:
  loadtest.js: |
    import http from 'k6/http';
    import { check, sleep } from 'k6';
    import { Trend } from 'k6/metrics';

    const PAYLOAD_KB = 1000; // 1, 5, 100 500 1000
```

```

export const options = {
  stages: [
    { duration: '0.5m', target: 5 }, // ramp-up
    { duration: '2m', target: 5 }, // steady
    { duration: '0.5m', target: 0 }, // ramp-down
  ],
  tags: { payload: `${PAYLOAD_KB}kb` }, // propagates to results
  thresholds: {
    http_req_failed: ['rate<0.05'],
    http_req_duration: ['p(95)<5000'],
  },
};

function makeBody(kb) {

const padding = 'X'.repeat(kb * 1024 - 20); // 20 bytes of JSON overhead
  return JSON.stringify({ message: padding });
}

const latencyMs = new Trend('latency_ms');

const URL = 'http://release2-flask-client-istio.default.svc.cluster.local:8

const PARAMS = { headers: { 'Content-Type': 'application/json' }, timeout:

export default function () {
  const body = makeBody(PAYLOAD_KB);

  const start = Date.now();
  const res = http.post(URL, body, PARAMS);
  latencyMs.add(Date.now() - start);

  check(res, { 'status 200': (r) => r.status === 200 });
  sleep(Math.random() * 0.5);
}

```

Листинг A.22 – Общий скрипт настройки k6

```

apiVersion: batch/v1
kind: Job
metadata:
  name: k6-loadtest
spec:
  template:
    spec:
      affinity:
        nodeAffinity:
          requiredDuringSchedulingIgnoredDuringExecution:
            nodeSelectorTerms:
              - matchExpressions:
                  - key: load
                    operator: In
                    values:

```

```
        - "true"
restartPolicy: Never
containers:
- name: k6
  image: loadimpact/k6:latest
  command: ["k6", "run", "/scripts/loadtest.js"]
  volumeMounts:
    - name: k6-script-volume
      mountPath: /scripts
volumes:
- name: k6-script-volume
  configMap:
    name: k6-script
backoffLimit: 1
```