

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

Национальный исследовательский ядерный университет «МИФИ»



**Институт интеллектуальных кибернетических систем
КАФЕДРА КИБЕРНЕТИКИ (№ 22)**

Направление подготовки 09.03.04 Программная инженерия

Расширенное содержание пояснительной записки

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

Писарев А. И.

Руководитель

Ровнягин М. М.

Научный консультант

Климов В. В.

Оценка руководителя

5 (отлично)

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 отчёт, 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 трафика.	6 неделя	
3.	Инженерная часть			
3.1	Проектирование архитектуры на уровне UML: создание диаграммы компонентов, диаграммы развёртывания 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 — 36 страниц, с учетом приложений — 36. Количество использованных источников — 22. Количество приложений — 0. МИКРОСЕРВИСНАЯ АРХИТЕКТУРА, ПАТТЕРНЫ ОТКАЗОУСТОЙЧИВОСТИ, КОНТЕЙ-
НЕРИЗАЦИЯ, 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	4
1 Аналитический обзор современных решений для обеспечения отказоустойчивости в микросервисных архитектурах.	6
1.1 Анализ архитектурных паттернов, обеспечивающих отказоустойчивость. . . .	6
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	16
1.6 Цели и задачи УИР	17
2 Моделирование микросервисной архитектуры с применением паттерна Circuit Breaker и алгоритмов распределенного трассирования	19
2.1 Модель микросервисной архитектуры с применением паттерна Circuit Breaker	19
2.2 Интеграция методов нагрузочного тестирования и распределенного трассирования	20
2.3 Сравнительный анализ реализаций Circuit Breaker	21
2.4 Выводы	22
3 Проектирование микросервисной архитектуры с помощью UML диаграмм.	23
3.1 Описание архитектуры системы с помощью диаграммы развертывания. . . .	23
3.2 Описание архитектуры системы с помощью диаграммы последовательности.	24
3.3 Выводы	25
4 Реализация и сравнительное тестирование системы.	30
4.1 Состав и структура реализованного программного обеспечения	30
4.2 Основные сценарии использования реализованной библиотеки	31
4.3 Результаты тестирования	31
4.4 Выводы	31
Заключение	36

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.

***Аннотация.** В разделе обобщаются современные подходы к обеспечению отказоустойчивости микросервисных архитектур с учётом их специфики и типичных рисков. Анализируются характерные проблемы надёжности распределённых сервисов и архитектурные паттерны (Circuit Breaker, Bulkhead), локализирующие влияние отказов. Рассматриваются инфраструктурные решения, включая контейнеризацию и платформы оркестрации (Docker, Kubernetes), а также сетевые решения уровня service mesh. Приводится сравнительный анализ популярных реализаций service mesh (Istio, Linkerd, Consul, Kuma) с обоснованием выбора Istio для практического применения. Рассматриваются методы оценки эффективности реализации: нагрузочное тестирование, трассировка. Сделан вывод о необходимости комплексного подхода, объединяющего паттерны, инфраструктуру и инструменты тестирования для достижения заданного уровня надёжности.*

Микросервисные системы предъявляют особые требования к обеспечению надёжности и устойчивости к отказам. В данном разделе обобщаются современные методы и инструменты, которые позволяют достичь высокой отказоустойчивости микросервисной архитектуры. Рассматриваются характерные проблемы надёжности распределённых сервисов и анализируются архитектурные паттерны, повышающие устойчивость (в том числе сравнение паттернов Circuit Breaker, Bulkhead и др.). Анализируются средства инфраструктуры – контейнеризации (Docker/Kubernetes) и сетевые решения (service mesh) – и 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 такие как k6, трассировки распределённых запросов (Jaeger).

1.1 Анализ архитектурных паттернов, обеспечивающих отказоустойчивость.

***Аннотация.** В отказоустойчивых системах сбои отдельных компонентов не должны приводить к недоступности всего приложения. Для достижения этого в микросервисной*

архитектуре используются специальные шаблоны проектирования (паттерны) надёжности. Два ключевых паттерна – это «Circuit Breaker» и «Bulkhead». Circuit Breaker отслеживает выполнение удалённых запросов и при накоплении ошибок автоматически «размыкает цепь», предотвращая дальнейшие вызовы проблемного сервиса на время его восстановления. Bulkhead изолирует ресурсы (пулы потоков, подключения) для разных частей системы, не позволяя отказу или перегрузке в одном сервисе использовать все общие ресурсы и нарушить работу других. В данном подразделе анализируется механизм действия этих паттернов и их применение для предотвращения каскадных отказов. Также кратко рассматриваются дополнительные приёмы повышения надёжности – повторные попытки (Retry), таймауты и резервные ответы (Fallback). В итоге подчёркивается, что сочетание грамотной архитектуры (паттерны отказоустойчивости) с правильной реализацией позволяет существенно повысить устойчивость сложных распределённых систем.

В микросервисной архитектуре отказ одного сервиса способен вызвать цепочку проблем во всей системе – эффект каскадного сбоя. Чтобы локализовать сбои и предотвратить их распространение, применяются шаблоны 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) размыкатель пропускает пробный запрос, и при успешном ответе замыкается обратно (состояние «Closed»), возобновляя нормальную работу; если же ошибка сохраняется – цикл повторяется. Таким образом, данный паттерн позволяет избежать бесполезной нагрузки на зависимый сервис, давая ему время восстановиться, и предотвращает занятия потоков ожиданием недоступного ресурса [5]. Практически, 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 БД для каждого микросервиса. В результате отказ или задержка в одном компоненте потребляет ресурсы только своего пула и не сможет повлиять на всю систему. Bulkhead-паттерн 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: размыкатель быстро отсекает проблемные вызовы, а Bulkhead 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 ошибок почти на 60%, а Bulkhead повысить общую доступность сервисов примерно на 10% [6].

Помимо указанных двух паттернов, в арсенале архитектуры микросервисов имеются и другие приёмы повышения устойчивости. 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 (транзистентных ошибках), 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 или деградация функциональности). Такие меры позволяют системе деградировать gracefully, без резкого отказа [7].

Комплексное применение данных паттернов доказало свою эффективность на практике. В частности, опыты, проведённые 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 сопутствующих им шаблонов (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

Аннотация. Рассмотрено влияние контейнеризации и оркестрации на отказоустойчивость облачных приложений. Проведен анализ использования *Docker* для создания лёгкой изоляции микросервисов, благодаря чему будет снижена вероятность ошибок окружения. Проведено исследование автоматизированного управления состоянием сервисов с помощью *Kubernetes*. Проанализированы механизмы оркестраторов, такие как перезапуск контейнеров и перенос подов. Подчеркнута роль *Kubernetes* как «операционной системы» для кластера и обеспечения надежности. Изучено применение *service mesh* как дополнительного слоя для управления межсервисными взаимодействиями. Упоминается о различных решениях *service-mesh*: *Istio*, *Linkerd*, *Consul Connect* и *Kuma*. Сделан акцент на многоуровневом подходе к созданию отказоустойчивых микросервисных систем

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*: при падении контейнера оркестратор перезапускает его, при отключении узла – переносит поды на работоспособные узлы, а при повышении нагрузки – может автоматически масштабировать реплики сервисов[**kuber**]. 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 работает, 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 [9]. 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 breaking, retries) 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: например, перенаправить запрос по другому маршруту, если основной сервис недоступен, зашифровать соединение, или прервать попытки связи 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[10]. 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[11], но подходы 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 систем[10]. Можно заключить, что для построения отказоустойчивых микросервисов требуется многоуровневый подход: устойчивость отдельных сервисов (за счёт контейнеризации и оркестрации) плюс устойчивость их взаимодействия (за счёт service mesh).

1.3 Анализ современных service mesh решений

Аннотация. Рассмотрены основные open-source реализации service mesh для Kubernetes, а именно Istio, Linkerd, Consul Connect и Kuma. Проведен подробный анализ архитектурных особенностей каждой системы, включая использование прокси и контроллеров. Особое внимание уделено функциональному сравнению возможностей, таких как маршрутизация, балансировка нагрузки и fault injection. Описываются также механизмы обеспечения безопасности, обеспечивающие автоматическую выдачу сертификатов шифрования и настройку политик доступа. Анализируются накладные расходы и производительность каждого решения, учитывая специфику распределенных систем. Обосновывается рассмотрение Istio, несмотря на повышенные требования к ресурсам.

Разработчики Kubernetes-среды 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, переписана на Rust.
3. **Consul (Connect)** — расширение популярной системы Consul от HashiCorp до полноценного service mesh.
4. **Kuma** — относительно новое решение (разработано Kong Inc., открыто 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-proxy) на Rust; минималистичный контроллер. Монолитная архитектура	Envoy sidecar (или собственный проху), контроллер интегрирован в Consul server. Требуется Consul-кластера	Envoy sidecar; control-plane Kuma (кластеризуемый). 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	Основной набор: трассировка, балансировка, маршрутизация, политики отказов, поддержка нескольких mesh. Мультизональность.
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 Envoy (CPU/память в 2–3 раза выше). Задержки под нагрузкой[12], 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

Продолжение на следующей странице

Таблица 1.1 – Продолжение

Аспект	Istio	Linkerd	Consul Connect	Kuma
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 привлекательным 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 [10], 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 в рамках Kubernetes.

1.4 Анализ инструментов и типов тестирования микросервисных архитектур.

Аннотация. Приведено сравнительное исследование open-source инструментов нагрузочного тестирования, включая Gatling, Locust и k6, с анализом их сильных и слабых сторон. Разработана методика оценки производительности микросервисов посредством анализа и выбора тестов. Отражена архитектура модуля нагрузочного тестирования, реализованного с использованием k6, что позволяет генерировать значительную нагрузку при низких накладных расходах. Разработан модуль распределенного трассирования с использованием Jaeger, обеспечивающий сбор и визуализацию метрик, задержек и ошибок. Проведен анализ альтернативных систем трассировки, включая Zipkin и OpenTelemetry Collector, с акцен-

том на модульную и масштабируемую архитектуру Jaeger.

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[13] – 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 [14] – 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 [15]. 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[16] – 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, сценарии 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. [17]

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

1. Нагрузочный тест (Load Test)[18]: проверяет работу сервиса при постепенном увели-

чении 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. Трассировка позволяет выявить узкие места, определить причины ошибок и оптимизировать производительность системы. Распределённое трассирование собирает данные о каждом микросервисе, участвующем в обработке запроса, и позволяет восстановить полную картину: 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[19] – 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[20], 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[21].
3. OpenTelemetry Collector[22] – 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

Аннотация. *Определены основные направления разработки и применения архитектурных паттернов для повышения отказоустойчивости микросервисов. Выполненный анализ показал эффективность паттернов Circuit Breaker и Bulkhead для предотвращения распространения сбоев. Сформулированы основные требования к использованию контейнерной оркестрации (Kubernetes) и service mesh (Istio, Linkerd, Consul Connect, Kuma), подчеркнута важность их совместного применения. Выполнено сравнение существующих решений service mesh, выявлены преимущества платформы Istio для сложных проектов. Показана необходимость практического тестирования отказоустойчивости, включая нагрузочные испытания, трассировку. Подчёркнута важность комплексного подхода, охватывающе-*

го проектирование, технологический стек и регулярную валидацию системы. 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 ограничивает 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

Аннотация На основе проведенного анализа сформулирована цель учебно-исследовательской работы: разработать и экспериментально сравнить решения для обеспечения отказоустойчивости микросервисных приложений, реализованных на платформе Kubernetes, на основе паттерна Circuit Breaker.

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

Аннотация. В данном разделе представлена теоретическая разработка модели микросервисной архитектуры, использующей паттерн Circuit Breaker для обеспечения устойчивости к сбоям. Анализируются два подхода реализации данного паттерна – программная реализация с использованием собственной библиотеки на базе Python и интегрирование паттерна в код и инфраструктурное решение с использованием Istio, конфигурируемое дополнительно, что позволяет выбрать между гибкой настройкой и централизованным управлением отказами. Особое внимание уделяется методам распределённого трассирования с применением промежуточного сборщика данных Fluent Bit и системой трассирования Jaeger, а также нагрузочному тестированию посредством k6, что обеспечивает детальный анализ производительности системы и оперативное обнаружение сбоев. Проведённый анализ выявляет ключевые отличия, влияющие на время работы и задержки, что изучается в последующих главах.

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

Аннотация. В подразделе описывается структура кластерной микросервисной системы, состоящей из серверных компонентов и прокси с реализацией паттерна Circuit Breaker. Рассматриваются два варианта реализации архитектуры: с использованием готового решения Istio Proxy и с применением Python-библиотеки. Описаны принципы взаимодействия компонентов и маршрутизации трафика между узлами внутри k3s кластера.

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

Аннотация. В подразделе представлена методология применения инструмента k6 для симуляции нагрузки на микросервисную систему. Описан процесс сбора и анализа трассировочных данных с использованием промежуточного сборщика и Jaeger как системы централизованного трассирования. Рассмотрен поток данных от микросервисов через компоненты трассирования для последующего визуального анализа в веб-интерфейсе Jaeger.

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

Аннотация. Подраздел содержит сравнительный анализ двух подходов к реализации паттерна Circuit Breaker в микросервисной архитектуре: инфраструктурного с использованием Istio и программного на базе Python. Описываются отличия полученной архитектуры и реализации. Рассмотрены особенности настройки, механизмы определения сбоев и алгоритмы восстановления для обоих вариантов в контексте устойчивости системы.

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 необходи-

мость 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

Аннотация. В подразделе представлены заключительные выводы по результатам теоретической разработки модели микросервисной архитектуры. Обозначены отличия реализации с использованием *service mesh*, от реализации с самостоятельно написанным паттернов. Учитывается эффективность методов распределенного трассирования и нагрузочного тестирования для обеспечения отказоустойчивости системы. Определены ключевые факторы, влияющие на производительность и надежность разработанной архитектуры.

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 k8s 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.

Аннотация. В данном разделе приведено подробное описание спроектированной микросервисной архитектуры с использованием UML диаграмм, демонстрирующих как физическое, так и логическое распределение компонентов. Отражено разделение системы на микросервисы с акцентом на их взаимное взаимодействие через стандартизированные интерфейсы. Приведено описание внешних интерфейсов, обеспечивающих связь с внешними клиентами и другими системами. Отражена модель внутренних интерфейсов, определяющих обмен данными между микросервисами. Приведены UML диаграммы развертывания, иллюстрирующие физическую инфраструктуру, включая использование *sidecar*-компонентов в Istio. Приведены диаграммы последовательности, на которых отражены все стадии прохождения запроса от клиента до сервера и получение ответных сообщений.

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.

Аннотация. Отражены ключевые аспекты взаимодействия компонентов на основе стандартной UML-нотации. Приведено подробное описание архитектуры системы с разделением на физические и виртуальные узлы. Разработана методика группировки контейнеров в поды для отображения ролей приложений и их *sidecar*-компонентов. Отражено распределение программных артефактов по физическим серверам и виртуальным машинам в рам-

ках кластера k3s. Приведены зависимости между компонентами, иллюстрирующие потоки данных и сетевые связи. Разработаны схемы, демонстрирующие интеграцию системного мониторинга с помощью fluentbit и Jaeger. Отражены принципы управления трафиком через Istio sidecar для обеспечения безопасности и надежности. Приведено использование Helm-чарта как средства автоматизированного развертывания сервисов.

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.

3.2 Описание архитектуры системы с помощью диаграммы последовательности.

Аннотация. Разработана модель коммуникаций, иллюстрирующая маршрутизацию запросов от иницилирующего сервиса к конечному получателю. Приведено подробное описание последовательности обмена сообщениями между сервисами, демонстрирующее логику взаимодействия. Отражено, как в версии с использованием Istio трафик проходит через sidecar-компоненты, обеспечивая дополнительный контроль и безопасность. Приведено сравнение с самописной реализацией, где обмен сообщениями осуществляется напрямую без промежуточного проху-слоя. Разработана последовательность вызовов, охватывающая все ключевые этапы обработки запросов в системе. Отражены различия в производительности и надежности между обеими реализациями коммуникации сервисов.

Диаграмма последовательности, представленная на (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.

Вариант без service mesh (см. рис. 3.4) показывает более прямую схему обмена сообще-

ниями: 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.

3.3 Выводы

Аннотация. *Получена достоверная модель системы, полностью отражающая архитектуру, которая остается лишь реализовать. Спроектированы основные программные модули, включая сервер, прокси-сервер и модуль генерации нагрузки k6. Учтены особенности физической инфраструктуры с разделением на физические серверы и виртуальные машины в кластере k3s. Отражена интеграция data collector для сбора и централизованной передачи логов и трейсов. Приведена схема автоматизированного развертывания с использованием Helm-чарта для управления микросервисами. Итоговый результат – надежная и полная инженерная модель, готовая к непосредственной реализации и проведению тестов.*

В итоге проведенного 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.

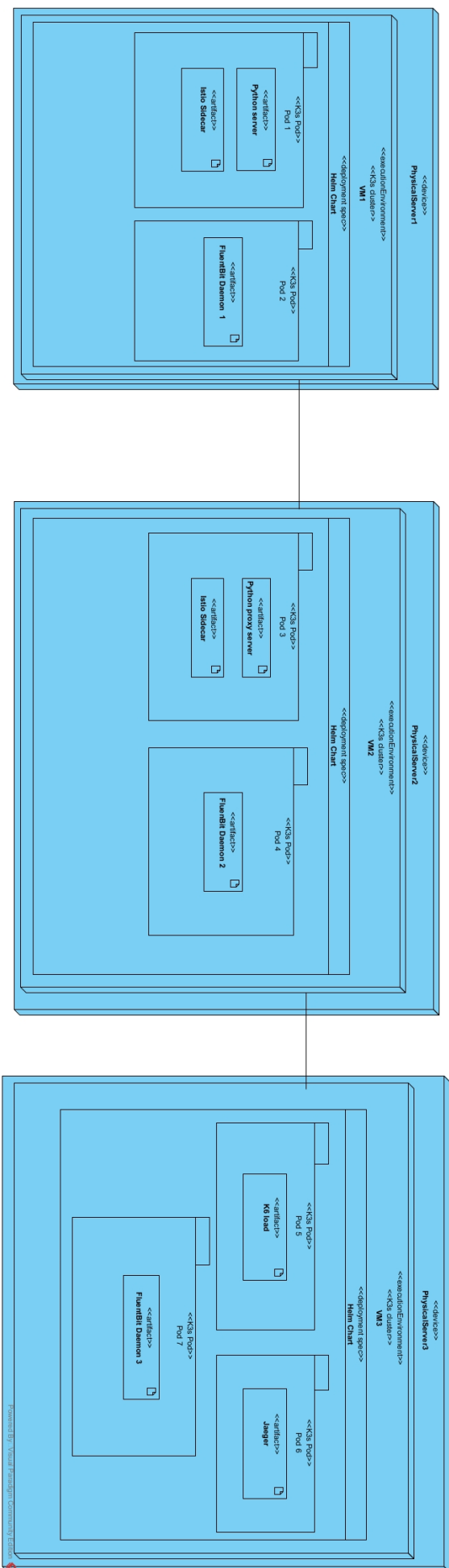


Рисунок 3.1 – Диаграмма развертывания версии с Isito

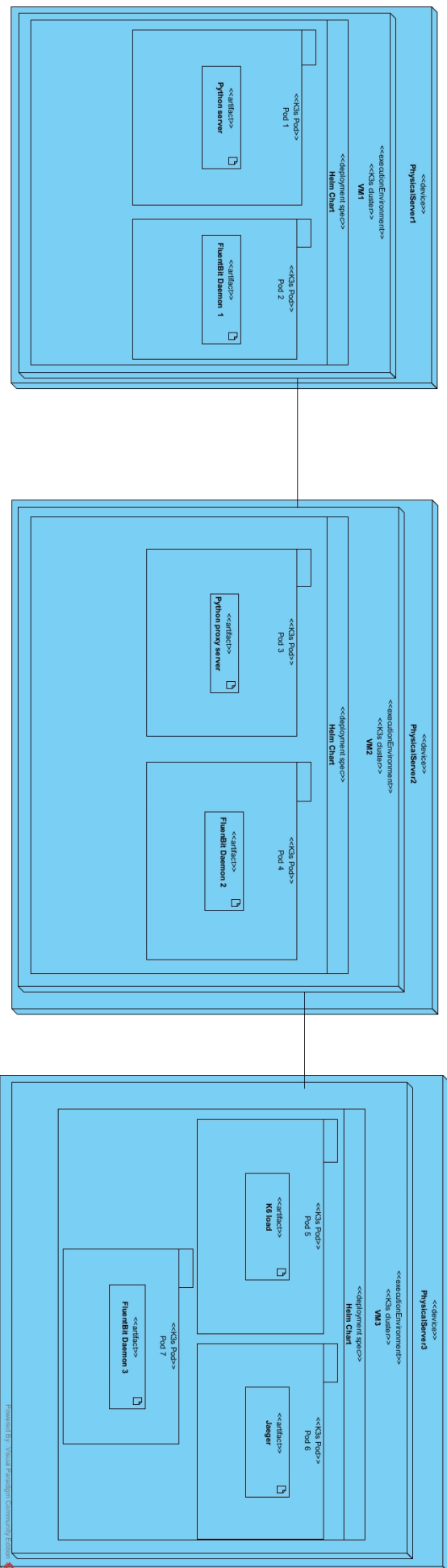


Рисунок 3.2 – Диаграмма развертывания версии без Isito



28

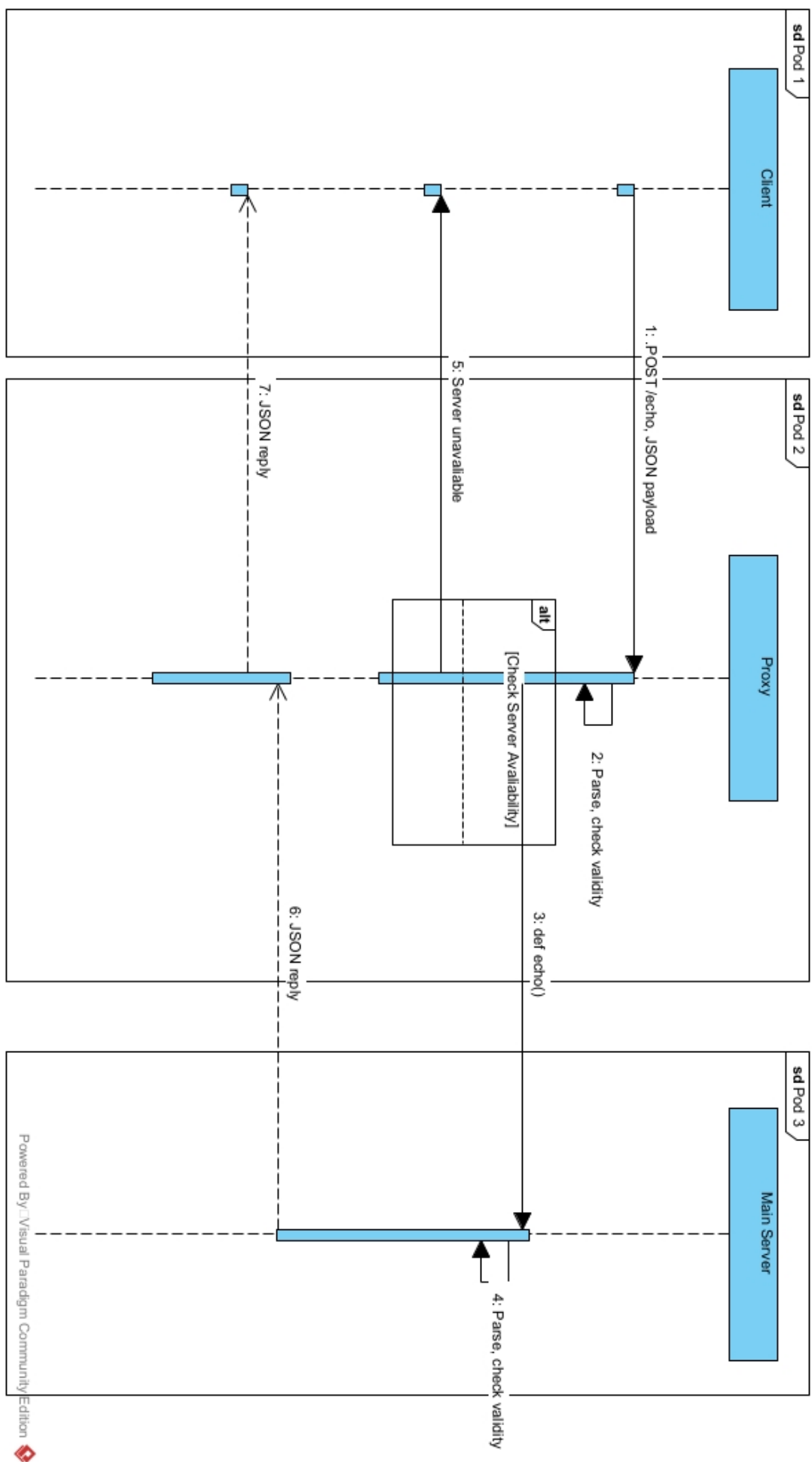


Рисунок 3.4 – Диаграмма последовательности версии без Istio

4. Реализация и сравнительное тестирование системы.

Аннотация. В данном разделе представлено комплексное описание реализации и тестирования системы. Детально 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

Аннотация. Архитектура представлена в виде двух параллельных кластеров виртуальных машин, управляемых посредством k3s. Каждый кластер состоит из трёх виртуальных машин, одна из которых является главной. Микросервисами выступают Flask веб серверы, написанные на Python. Разворачивание компонентов системы осуществляется через Helm-чарты с использованием YAML-конфигураций, что упрощает масштабирование и обновление. На ноде не содержащей микросервисов, развернут инструмент k6, симулирующий нагрузку и моделирующий реальные сценарии эксплуатации системы. Система трассировки Jaeger также работает на ноде без микросервисов. Для обеспечения контроля и анализа работы компонентов на каждой ноде развернут сборщики трейсов fluentbit в виде daemon set, передающих данные в систему Jaeger.

В данной главе описана структура реализованного программного обеспечения, в основе которого лежит микросервисная архитектура, развернутая на кластере виртуальных машин с использованием k3s. Представлено алгоритмическое описание python сервера псевдокодом (4.1). Написан шаблон Python кода сервера (4.2), реализована версии с Istio, написана библиотека с реализацией паттерна Circuit Breaker. Для реализации веб-сервисов использован Flask, запущенный через Gunicorn для повышения производительности, что позволяет эффективно обрабатывать входящие запросы. Все компоненты упакованы в Docker-образы

и опубликованы на Docker Hub. Приведён листинг Docker-файла сервера (4.3), а так же листинг Docker-файла прокси-сервера (4.4).

Развертывание компонентов системы осуществляется посредством Helm-чартов, где все параметры заданы в файле values.yaml (4.5). Такой подход обеспечивает гибкую конфигурацию и автоматизацию обновлений, позволяя централизованно управлять параметрами развертывания в кластере.

4.2 Основные сценарии использования реализованной библиотеки

Аннотация. Разработана переиспользуемая библиотека с функционалом *circuit breaker* для повышения устойчивости распределенных приложений. Обоснованы принципы повторного использования кода, позволяющие интегрировать библиотеку в различные системы. Описаны методы настройки через Helm-чарты для проведения экспериментов с разным числом микросервисов. Показан способ добавления и изменения отслеживаемых трейсов для анализа работоспособности системы. Объяснены методы построения наглядных графиков задержек в проекте.

4.3 Результаты тестирования

Аннотация. Разработана методика тестирования ключевых сценариев эксплуатации системы, направленная на оценку её производительности и отказоустойчивости. Проведены нагрузочные тесты, предусматривающие постепенное увеличение числа виртуальных пользователей и количества запросов в секунду, что имитирует пиковые периоды активности. Проведены тест с увеличением размера запроса, при котором фиксированное число одновременных обращений обрабатывает всё более объёмные данные. Результаты тестирования представлены в виде таблиц и графиков, позволяющих наглядно оценить зависимость производительности от параметров нагрузки.

4.4 Выводы

Аннотация. В данном разделе подводятся итоги разработки и исследования решения, направленного на обеспечение отказоустойчивости распределённых систем с использованием механизма *circuit breaker*. Суммируются достигнутые практические результаты, включающие создание программного обеспечения, библиотеки на Python, разработку методик тестирования и формирование набора тестовых примеров для оценки задержек в работе паттерна. Анализируется степень эффективности промышленного решения и библиотеки. Оцениваются перспективы интеграции разработанного ПО в комплексные информационные системы, а также направления для дальнейших исследований.

Листинг 4.1 – Листинг псевдокода скрипта работы сервера

```
// 1. Импорт библиотек и модулей
Импортировать модули:
- os, time, Flask, jsonify, opentelemetry

// 2. Настройка OTLPСборщика- и ресурсов сервиса
Установить:
    otlp_endpoint ← значение из переменной окружения "OTLP_COLLECTOR_ENDPOINT"
Создать ресурс:
    resource ← { "service.name": "flask-server" }
Инициализировать провайдер трассировки:
    trace.set_tracer_provider(TracerProvider(resource = resource))
// 3. Инициализация экспортера спанов
Создать экспортера:

    otlp_exporter ← OTLPSpanExporter(endpoint = otlp_endpoint, insecure = True)
Добавить обработчик спанов:
    provider.add_span_processor(BatchSpanProcessor(otlp_exporter))

// 4. Инициализация Flask приложения и инструментирование трассировки
Создать Flask приложение:
    app ← Flask(__name__)
Настроить приложение для автоматического отслеживания:
    FlaskInstrumentor().instrument_app(app)
    RequestsInstrumentor().instrument()
    tracer ← trace.get_tracerимя( текущего модуля)

// 5. Определение маршрута для обработки POSTзапросов- по адресу '/echo'
Определить маршрут '/echo' с методом POST:
    ФУНКЦИЯ echo():
        // Запуск корневого спана для обработки запроса
        С ОТКРЫТЫМ спаном "echo_request_server":
            start_time ← текущее время

            // Спан для разбора запроса
            С ОТКРЫТЫМ спаном "parse_request_server":
                data ← получить JSON из запроса
                Если data существует, то:
                    message ← значение поля "message" из data
                Иначе:
                    message ← пустая строка
                Установить атрибут "message.length" равным длине message

            // Спан для обработки сообщения
            С ОТКРЫТЫМ спаном "process_message_server":
                Вывести в консоль
                сообщение: "[Server] Received message from client: " + message
                Установить атрибут "message.logged" равным True

            // Спан для формирования ответа
            С ОТКРЫТЫМ спаном "construct_response_server":
                response_data ← словарь с:
                    "message": "Echo: " + message
                    "timestamp": текущее время в( виде целого числа)
                Установить атрибут "response.size" равным размеру response_data
                response ← преобразовать response_data в JSON

            total_duration ← текущее время - start_time
            Установить атрибут
            "total.duration" в корневом спане равным total_duration
            Вернуть response клиенту
// 6. Запуск Flask сервера при прямом выполнении скрипта
app.run(host = "0.0.0.0", port = 5001, debug = False)
```

Листинг 4.2 – Листинг из файла server.py

```
# Импорт библиотек и модулей: Flask - для создания вебприложения-,
# opentelemetry - для реализации распределённого трассирования.
import os, time, flask, opentelemetry
# Настройка точки сбора OTLP и ресурсов сервиса:
# Получаем endpoint сборщика из переменной окружения с( запасным значением),
# создаём ресурс с атрибутом "service.name" для идентификации сервиса,
# устанавливаем провайдер трассировки с заданным ресурсом.
otlp_endpoint = os.getenv("OTLP_COLLECTOR_ENDPOINT", "collector-endpoint:4317")
resource = Resource(attributes={"service.name": "flask-server"})
trace.set_tracer_provider(TracerProvider(resource=resource))
# Инициализация экспортера спанов и добавление процессора:
# Создаём экспортера для отправки спанов через gRPC к OTLP коллектора,
# добавляем процессор спанов для пакетной отправки данных.
otlp_exporter = OTLPSpanExporter(endpoint=otlp_endpoint, insecure=True)
provider.add_span_processor(BatchSpanProcessor(otlp_exporter))
# Инициализация Flask приложения и инструментация для трассировки.
# Конфигурация приложения для автоматического отслеживания запросов.
app = Flask(__name__)
FlaskInstrumentor().instrument_app(app)
RequestsInstrumentor().instrument()
tracer = trace.get_tracer(__name__)
# Определение маршрута для обработки POST-запросов по адресу '/echo':
@app.route('/echo', methods=['POST'])
def echo():
    # Начало корневого спана для всего процесса обработки запроса echo.

    # Этот спан объединяет в себе все внутренние спаны, отражая общий цикл обработки.
    with tracer.start_as_current_span("echo_request_server") as span:
        start_time = time.time()
        # Спан для разбора входящего JSON-пайлоада:
        # Извлекаем данные из запроса и получаем сообщение.

        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))
            # Спан для обработки сообщения:
            # Логируем полученное сообщение и фиксируем факт его обработки.

            with tracer.start_as_current_span("process_message_server") as process_span:
                print(f"[Server] Received message from client: {message}")
                # Отмечаем, что сообщение было залогировано.
                process_span.set_attribute("message.logged", True)
                # Спан для формирования ответа:

                # Создаём словарь с ответом, включающий эхосообщение- и временную метку,
                # фиксируем размер сформированного ответа и преобразуем словарь в JSON.

                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)

                # Добавляем общую длительность обработки запроса в атрибуты корневой спан.
                span.set_attribute("total.duration", time.time() - start_time)
                # Возвращаем ответ клиенту.
                return response

    # При запуске скрипта напрямую, запускается Flask сервер на указанном хосте и порте.
if __name__ == "__main__":
    app.run(host="0.0.0.0", port=5001, debug=False)
```

Листинг 4.3 – Листинг из файла Dockerfile сервера

```
# Используется базовый образ с Python версии 3.11.
FROM python:3.11
# Устанавливает рабочую директорию внутри контейнера, куда будут копироваться
WORKDIR /home/hpc/aleksandr
# Копирует файл зависимостей в рабочую директорию контейнера.
COPY requirements.txt .
# Выполняет установку зависимостей, перечисленных в requirements.txt.
RUN pip install --no-cache-dir -r requirements.txt
# Копирует файл server.py в рабочую директорию контейнера.
COPY server.py .
# Объявляет, что контейнер принимает входящие соединения на порту 5001.
EXPOSE 5001
# задает
# команду по умолчанию для запуска контейнера:
#   # запускается серверное приложение через Gunicorn с:
#   # 4 рабочими процессами,
#   # прослушиванием на порту 5001
CMD ["gunicorn", "-w", "4", "-k", "gthread", "-t", "30",
    "-b", "0.0.0.0:5001", "server:app"]
```

Листинг 4.4 – Листинг из файла Dockerfile прокси сервера

```
# Используется базовый образ с Python версии 3.11.
FROM python:3.11
# Устанавливает рабочую директорию внутри контейнера,
# куда будут копироваться файлы и выполняться приложение.
WORKDIR /home/hpc/aleksandr
# Копирует файл зависимостей в рабочую директорию контейнера.
COPY requirements.txt .
# Выполняет установку зависимостей, перечисленных в requirements.txt.
RUN pip install --no-cache-dir -r requirements.txt
# Копирует файлы в рабочую директорию контейнера.
COPY . .
# Объявляет, что контейнер принимает входящие соединения на порту 5001.
EXPOSE 8001
# задает
# команду по умолчанию для запуска контейнера:
#   # запускается серверное приложение через Gunicorn с:
#   # 4 рабочими процессами,
#   # прослушиванием на порту 5001
CMD ["gunicorn", "-w", "4", "-k", "gthread", "-t", "30",
    "-b", "0.0.0.0:8001", "client:app"]
```

Листинг 4.5 – Листинг из файла values.yaml

```
server:
  replicaCount: 1
  image:
    repository: "robocatt/flask-server-istio"
    tag: "latest"
    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: []
proxy:
  replicaCount: 1
  image:
    repository: "robocatt/flask-client-istio"
    tag: "latest"
    pullPolicy: IfNotPresent
  service:
    type: ClusterIP
    port: 8001
  affinity:
    nodeAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        nodeSelectorTerms:
          - matchExpressions:
              - key: worker
                operator: In
                values:
                  - "true"

  env:
    jaegerEndpoint: "jaeger-otlp-service:4317"
  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
  httpService:
    name: jaeger-http
    type: ClusterIP
    port: 4318
    targetPort: 4318
```

Заключение

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

Ожидаемыми 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 & 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. *Méndez S.* Edge Computing Systems with Kubernetes. — Packt Publishing, 2022.
10. *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>.
11. *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.

12. 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>.
13. Gatling: Load testing designed for DevOps and CI/CD. — URL: <https://gatling.io/> (дата обр. 25.03.2025) ; Accessed on 25 March 2025.
14. Locust: Locust Load Testing Tool. — URL: <https://locust.io/> (дата обр. 25.03.2025) ; Accessed on 25 March 2025.
15. 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>.
16. Grafana k6: Load testing for engineering teams. — URL: <https://k6.io/> (дата обр. 25.03.2025) ; Accessed on 25 March 2025.
17. *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>.
18. 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>.
19. Jaeger: open source, distributed tracing platform. — URL: <https://www.jaegertracing.io/> (дата обр. 26.03.2025) ; Accessed on 26 March 2025.
20. Zipkin. — URL: <https://zipkin.io/> (дата обр. 26.03.2025) ; Accessed on 25 March 2026.
21. *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>.
22. 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.