УДК 004.75
МРНТИ 28.23.15

https://doi.org/10.55452/1998-6688-2026-23-2-276-289

[1]*Кемелбеков Н.К.,
докторант, ORCID ID: 0009-0008-6784-3842,
*e-mail: kemellbekov@gmail.com
[1]Рахимова Д.Р.,
PhD, ассоциированный профессор, ORCID ID: 0000-0003-1427-198X,
e-mail: diana.rakhimova@kaznu.edu.kz
[2]Adali E.,
профессор-эмеритус, ORCID ID: 0000-0002-1561-8255,
e-mail: adali@itu.edu.tr

[1]Казахский национальный университет им. аль-Фараби, г. Алматы, Казахстан
[2]Istanbul Technical University, г. Стамбул, Турция

SERVICE MESH: АРХИТЕКТУРНЫЕ ПРИНЦИПЫ, ПРОИЗВОДИТЕЛЬНОСТЬ И ПРАКТИКА ПРИМЕНЕНИЯ В МИКРОСЕРВИСНЫХ СИСТЕМАХ

Аннотация

В работе рассматривается технология Service Mesh как выделенный инфраструктурный слой для управления межсервисным взаимодействием в микросервисных системах. Цель исследования – проанализировать архитектурные принципы построения сетевого слоя, оценить накладные расходы на производительность и показать практические подходы к эксплуатации в Kubernetes-кластерах. Описывается модель разделения на control plane и data plane, использование паттерна sidecar для прозрачной маршрутизации трафика, реализации mTLS и автоматического сбора телеметрии. На основе обзора литературы и открытой документации выполняется сравнительный анализ трех ключевых реализаций – Envoy, Istio и Linkerd – по функциональности, сложности внедрения и ресурсоемкости. Отдельно приводится практический пример конфигурации Istio с использованием манифестов Gateway, VirtualService и DestinationRule для управления HTTPS-трафиком и интеграции со стеком наблюдаемости Prometheus – Grafana – Jaeger – Kiali. Показано, что Service Mesh добавляет умеренные накладные расходы по задержке и потреблению ресурсов, однако обеспечивает унифицированное управление безопасностью и трафиком между микросервисами. Полученные результаты позволяют сформулировать практические рекомендации по целесообразности применения Service Mesh в корпоративных микросервисных платформах в зависимости от масштаба системы и требований к безопасности.

Ключевые слова: Service Mesh, микросервисы, Kubernetes, Istio, Envoy, Linkerd, mTLS, наблюдаемость, распределенные системы.

Введение

Переход от монолитных архитектур к микросервисным системам стал доминирующим трендом в разработке облачных приложений. По данным CNCF (Cloud Native Computing Foundation) Annual Survey 2023, Kubernetes используется в production у значительной доли организаций (для группы конечных потребителей – 66% в production, еще 18% оценивают), что косвенно отражает масштаб распространения облачно-нативных практик и микросервисных архитектур [1]. Однако увеличение числа сервисов приводит к экспоненциальному росту сложности межсервисного взаимодействия: система из N сервисов может иметь до O(N^2) потенциальных соединений.

Традиционные подходы к управлению коммуникацией требуют встраивания логики безопасности, наблюдаемости и устойчивости непосредственно в прикладной код каждого сервиса. Это приводит к дублированию кода, несогласованности реализаций и сложности обновления политик. Service Mesh решает эту проблему путем вынесения сетевой логики в отдельный инфраструктурный слой, реализуемый через паттерн sidecar proxy.

Цель и задачи исследования

Цель работы – провести комплексный анализ технологии Service Mesh, исследовать архитектурные решения, оценить характеристики и продемонстрировать практическое применение.

Задачи исследования: систематизировать архитектурные принципы и компоненты Service Mesh; выполнить сравнительный анализ реализаций Envoy, Istio и Linkerd; оценить основные накладные расходы; показать практический пример маршрутизации и мониторинга; обозначить основные ограничения и направления развития технологии.

Проблематика и мотивация

Вызовы микросервисной архитектуры

При масштабировании микросервисных систем возникают следующие критические проблемы. Сетевая сложность: в системе из 100 сервисов существует до 10 000 потенциальных соединений, каждое из которых требует настройки маршрутизации, балансировки и обработки отказов. Безопасность: обеспечение mutual TLS между всеми сервисами предполагает управление большим количеством сертификатов и их ротацией. Наблюдаемость: прослеживание распределенных транзакций через десятки сервисов без единой системы трассировки затруднительно. Устойчивость: внедрение паттернов circuit breaker, retry и timeout в каждом сервисе приводит к дублированию инфраструктурного кода. Гетерогенность: использование различных языков программирования требует поддержания нескольких клиентских библиотек и усложняет унификацию.

Обоснование Service Mesh

Service Mesh предоставляет унифицированный слой, который решает эти задачи за счет разделения ответственности (бизнес-логика остается в приложении, сетевые аспекты выносятся в mesh), языковой независимости (вся функциональность реализуется на уровне прокси), централизованного управления политиками через control plane и высокой степени автоматизации (service discovery, выпуск и ротация сертификатов, распространение конфигурации).

Сокращения и термины

CNCF – Cloud Native Computing Foundation.

CRD – Custom Resource Definition (пользовательские ресурсы Kubernetes).

mTLS – mutual TLS (взаимная TLS-аутентификация сервисов).

xDS – семейство API Envoy для доставки конфигурации (CDS/EDS/LDS/RDS и др.).

RPS – requests per second (запросов в секунду).

p95/p99 – 95/99 перцентиль задержки.

WASM – WebAssembly (механизм расширения прокси фильтрами).

SPIFFE – Secure Production Identity Framework For Everyone (идентичности workloads).

Материалы и методы

Концепция Service Mesh сформировалась в середине 2010-х годов как ответ на рост сложности микросервисных систем. Первые решения, такие как Linkerd (2016) и Envoy (2016), заложили основу современных mesh-платформ. Общие подходы к проектированию микросервисных систем и их эволюции подробно разобраны в работе Ньюмана [2].

Исследование [3] рассматривает Service Mesh как эволюцию паттерна Service Discovery с добавлением механизмов управления трафиком и политиками. Burns и Oppenheimer [4] описывают sidecar pattern как базовый паттерн для контейнерных распределенных систем.

Работа показывает, что использование sidecar proxy обычно увеличивает задержку на 1–5 мс и требует 50–200 МБ памяти на инстанс в зависимости от конфигурации [5]. Исследования безопасности [6] подчеркивают, что автоматизация управления mTLS-сертификатами позволяет практически исключить инциденты, связанные с истекшими сертификатами. Дополнительные аспекты производительности и трассировки систем рассматриваются в [7].

Контекст перехода к микросервисам детально рассмотрен в работе Ричардсона [8], систематизировавшего паттерны межсервисного взаимодействия: API Gateway, Circuit Breaker, Saga и Event Sourcing. Автор показывает, что по мере роста числа сервисов управление межсервисной коммуникацией становится самостоятельной инженерной задачей, требующей инфраструктурного решения. Эмпирическое сравнение монолитной и микросервисной архитектур в облачных средах представлено в работе Вилламизара и соавторов [9]: при правильной декомпозиции микросервисы обеспечивают лучшую масштабируемость и отказоустойчивость, однако существенно усложняют сетевое взаимодействие и управление согласованностью данных. Контейнерные технологии и их эволюция в сторону облачно-нативных платформ систематизированы в обзоре Паля и соавторов [10], анализирующем основные инфраструктурные паттерны, включая оркестрацию, сервисный реестр и конфигурационное управление.

С точки зрения информационной безопасности концепция нулевого доверия (Zero Trust), формализованная в стандарте NIST SP 800-207 [11], предполагает верификацию каждого сетевого запроса вне зависимости от его источника и предшествующей сессии. Service Mesh является одним из ключевых практических инструментов реализации Zero Trust в среде Kubernetes: автоматическое взаимное TLS-шифрование и декларативные политики авторизации обеспечивают непрерывную аутентификацию на уровне сетевого взаимодействия без изменения кода приложений. Вопросы надежной эксплуатации крупных распределенных систем обобщены в книге по инженерии надежности сайтов Бейера и соавторов [12]: описанные принципы управления изменениями, мониторинга и управления инцидентами непосредственно применимы к операционной модели Service Mesh в production-среде. Практику работы с Kubernetes-кластерами в контексте безопасной и наблюдаемой эксплуатации дополнительно рассматривает Лукша [13].

Фундаментальные принципы распределенной трассировки, лежащие в основе подсистем наблюдаемости Service Mesh, были заложены в пионерской работе Зигельмана и соавторов о системе Dapper [14], разработанной в Google. Предложенная модель дерева span и механизм сквозного распространения контекста трассировки вошли в основу современных открытых стандартов, в первую очередь OpenTelemetry [15] и форматов B3/W3C TraceContext. Применение технологии eBPF для повышения производительности сетевых операций в Kubernetes описано Калаверой и Фонтаной [16]: перенос части сетевой логики в пространство ядра позволяет сократить накладные расходы по сравнению с классическим паттерном sidecar и является основой таких проектов, как Cilium Service Mesh [17]. Таким образом, анализ предметной области демонстрирует, что Service Mesh находится на пересечении нескольких активно развивающихся направлений: безопасности распределенных систем, наблюдаемости, контейнерной оркестрации и сетевых технологий.

Архитектурная модель Service Mesh

Архитектура Service Mesh опирается на прозрачность (приложения не знают о наличии mesh, весь трафик перехватывается автоматически), модульность (маршрутизация, безопасность и наблюдаемость разделены на отдельные компоненты), декларативную конфигурацию (YAML/CRD), расширяемость (плагины и фильтры) и отказоустойчивость (data plane продолжает обслуживать трафик даже при кратковременной недоступности control plane).

Control plane отвечает за управление и конфигурацию mesh. В его состав входят Istiod (в Istio – объединяющий функции маршрутизации, безопасности и валидации конфигурации), подсистема Service Discovery, интегрированная с Kubernetes API, Certificate Authority для выпуска и ротации X.509 сертификатов, а также компоненты, распространяющие политики через xDS (CDS, EDS, LDS, RDS).

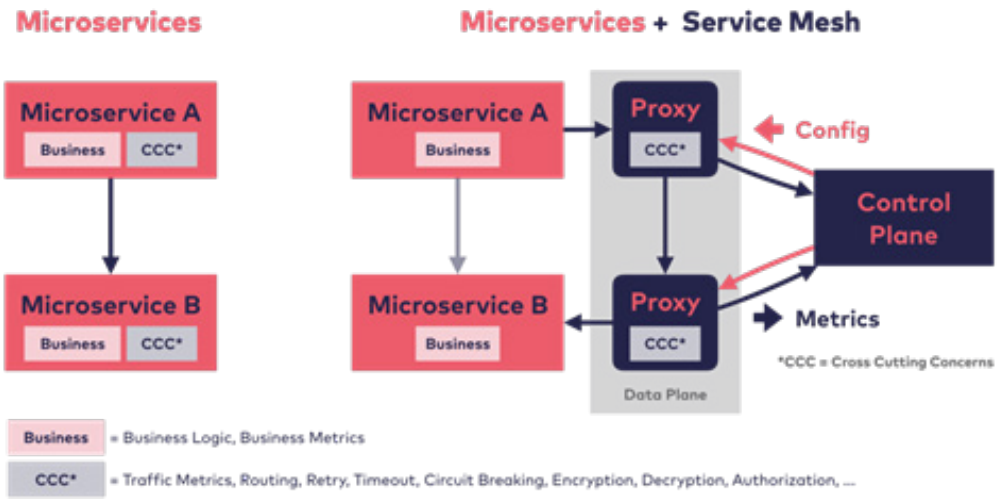

Рисунок 1 – Сравнение сетевого взаимодействия в микросервисной архитектуре:
(а) традиционный подход с сетевой логикой в приложениях; (б) архитектура с Service Mesh
и выделенным инфраструктурным слоем

Data plane состоит из sidecar-прокси, которые обрабатывают весь входящий и исходящий трафик сервисов. Они обеспечивают проксирование L4/L7-трафика (TCP, HTTP/1.1, HTTP/2, gRPC), TLS-терминацию, балансировку нагрузки, health checking, применение circuit breaker, генерацию телеметрии и авторизацию запросов.

Паттерн Sidecar

Sidecar proxy разворачивается в одном Pod с приложением и перехватывает трафик с помощью iptables redirect или eBPF. В новых подходах (ambient mesh) часть функций переносится на уровень узла, что позволяет уменьшить количество sidecar-контейнеров.

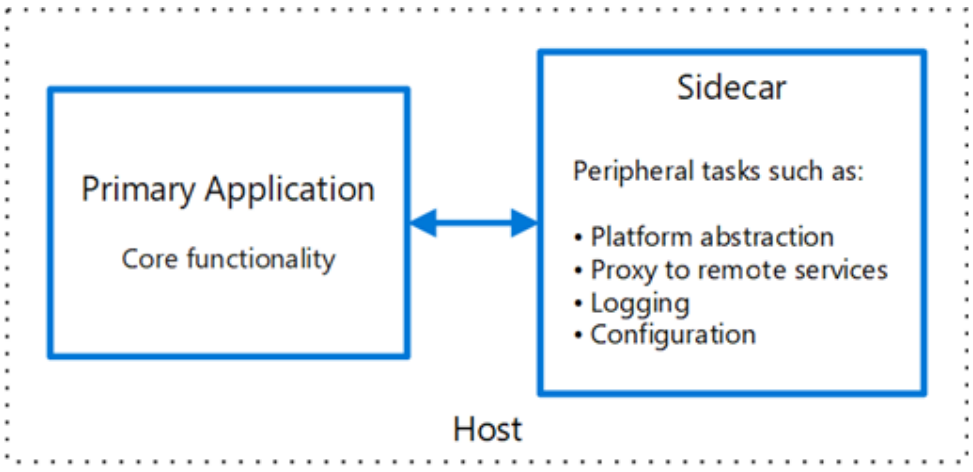

Рисунок 2 – Паттерн Sidecar: развертывание прокси в одном Pod с приложением

К преимуществам sidecar-паттерна относятся изоляция сетевой логики, возможность независимого обновления прокси и единая модель конфигурации для всех языков. Недостатки связаны с дополнительным потреблением CPU и памяти, увеличением задержки и усложнением отладки из-за дополнительного сетевого перехода.

Отказоустойчивость control plane (Istiod) и эксплуатационные риски

Control plane (Istiod) отвечает за распространение конфигураций и политик в data plane. При недоступности control plane уже сконфигурированные прокси продолжают обслуживать трафик, однако распространение новых конфигураций, политик безопасности и обновлений маршрутизации становится невозможным до восстановления control plane. Поэтому в production-средах критично обеспечить высокую доступность Istiod: несколько реплик, устойчивое размещение по узлам/зонам и при необходимости использование нескольких control plane в multi-cluster моделях, что ограничивает область влияния отказа одной инстанции control plane.

Дополнительно следует учитывать устойчивость канала доставки конфигураций (xDS) и механизмы защиты от недоступности control plane (например, TTL/поведение при срыве доставки динамической конфигурации).

Функциональные компоненты Service Mesh

Интеллектуальная маршрутизация трафика

Service Mesh предоставляет расширенные возможности маршрутизации. Content-based routing использует HTTP-заголовки, путь URI, query-параметры и HTTP-метод для принятия решений. Traffic splitting позволяет гибко распределять трафик между версиями сервиса: canary deployment с постепенным увеличением доли новой версии, A/B-тестирование и схемы blue-green.

Для повышения устойчивости применяются retry и timeout, а также circuit breaker, ограничивающий обращения к нестабильным backend-сервисам, и bulkhead-подходы, ограничивающие количество одновременных соединений.

Безопасность

Service Mesh автоматизирует реализацию mutual TLS: сервисы получают X.509 сертификаты при старте, сертификаты регулярно ротируются, а идентификация строится на основе SPIFFE ID [18]. Поддерживаются режимы STRICT (обязательный mTLS), PERMISSIVE (одновременная поддержка mTLS и незашифрованного трафика) и DISABLE.

Политики авторизации используют Service Identity, RBAC/ABAC-модели и проверку JWT-токенов, что позволяет задавать детальные правила доступа на уровне сервисов и отдельных маршрутов.

Наблюдаемость

Автоматический сбор метрик L4 и L7 покрывает основные Golden Signals (latency, traffic, errors, saturation) и позволяет строить RED- и USE-профили сервисов. Распределенная трассировка реализуется за счет генерации span на каждом hop и передачи trace ID (например, в формате B3-заголовков) с последующей визуализацией в системах вроде Jaeger или других решений на базе OpenTelemetry [15]. Структурированные access logs с trace ID интегрируются с системами логирования (ELK, Loki), обеспечивая сквозную диагностику.

Результаты и обсуждение

Сравнительный анализ реализаций Envoy, Istio и Linkerd

Envoy Proxy

Envoy – высокопроизводительный L4/L7-прокси, разработанный Lyft и переданный в CNCF [19]. Он реализован на C++ с асинхронной event-driven архитектурой, поддерживает HTTP/1.1, HTTP/2, gRPC, WebSocket и TCP, расширяется фильтрами (Lua, WASM) и настраивается через xDS API. Envoy обеспечивает высокую пропускную способность, добавляет 0,5–2 мс к задержке на высоких перцентилях и требует порядка десятков мегабайт памяти в базовой конфигурации.

Istio

Istio – полнофункциональный Service Mesh, использующий Envoy в качестве data plane [20]. В его состав входят Istiod как control plane, Envoy sidecar и ingress/egress gateways.

Отличительными особенностями являются широкий набор функций для безопасности, маршрутизации и наблюдаемости, развитая экосистема и поддержка мультикластерных сценариев. Основные недостатки связаны со сложностью настройки, повышенными ресурсными требованиями и необходимостью наличия подготовленной команды эксплуатации.

Linkerd

Linkerd – легковесный Service Mesh, ориентированный на простоту и безопасность [21]. Его прокси написан на Rust, что обеспечивает низкое потребление ресурсов и малую задержку. Linkerd предлагает автоматический mTLS, минималистичный control plane и поддержку multi-cluster. В обмен на это реализовано меньше расширенных функций по сравнению с Istio, а возможности по кастомизации и расширяемости более ограничены.

3.1.4 Сравнительная таблица реализаций

Таблица 1 – Сравнение реализаций Service Mesh

Параметр	Envoy	Istio	Linkerd
Язык реализации	C++	Go + Envoy	Rust
Потребление памяти	50–100 МБ	100–200 МБ	10–20 МБ
Latency overhead	0,5–2 мс	1–5 мс	< 1 мс
Сложность настройки	Средняя	Высокая	Низкая
Расширяемость	WASM, Lua	WASM	Ограничена
Зрелость	Высокая	Высокая	Средняя

Источники накладных расходов и оптимизация производительности

Основные накладные расходы связаны с задержкой и ресурсами. Использование sidecar-прокси добавляет дополнительный сетевой hop и накладные расходы на TLS: время установки соединения (TLS handshake), обработку запросов в прокси и дополнительный round-trip. В совокупности это дае т увеличение p95 latency на 1–5 мс в типичных сценариях [5]. По ресурсам sidecar потребляет часть CPU и памяти (десятки-сотни mCPU и десятки-сотни мегабайт), а также несколько процентов пропускной способности сети за счет шифрования.

Накладные расходы Service Mesh по задержке зависят от параметров нагрузки и конфигурации прокси: размера запросов/ответов, числа клиентских соединений, целевого RPS, протокола (HTTP/1.1, HTTP/2, gRPC), числа worker threads у прокси и набора включе нных функций (в частности, фильтры телеметрии – метрики/логирование/трейсинг – заметно влияют на tail latency).

В официальных измерениях Istio приведен ориентир по ресурсам sidecar при 1000 HTTP rps и payload 1 KB [25]; эти же параметры удобно использовать как “baseline” для сопоставимых оценок overhead в практических кластерах.

При сравнительных тестах mTLS в Kubernetes-кластере в литературе фиксируются заметные изменения tail latency при росте нагрузки.

Для минимизации задержек используются connection pooling и HTTP/2, resumption TLS-сессий, локализационно-осознанная маршрутизация (предпочтение локальных endpoints) и отключение неиспользуемых фильтров. Потребление ресурсов снижается настройкой лимитов CPU и памяти, масштабированием control plane, selective injection (включение sidecar только там, где действительно нужен mesh), а также использованием ambient mesh [22] и eBPF-подходов [16], где часть логики переносится на уровень ядра и узла.

Практическая реализация в Istio

В Istio маршрутизация настраивается через набор CRD: Gateway, VirtualService и DestinationRule, которые концептуально согласуются с развивающимся стандартом

Kubernetes Gateway API [23]. Параллельно с собственными CRD Istio (Gateway/VirtualService/DestinationRule) индустрия развивает стандарт Kubernetes Gateway API как vendor-neutral модель маршрутизации. Istio поддерживает Gateway API и декларирует намерение сделать его API “по умолчанию” для traffic management в будущем [24], при этом собственные Istio-API сохраняются для существующих инсталляций и сценариев, где нужны расширенные возможности [24]. Рассмотрим конфигурацию для домена https://keycloak-faceless.epayment.kz.

Gateway определяет точку входа трафика в mesh и управляет TLS-терминацией: указываются selector для привязки к Ingress Gateway, список servers с портами, протоколами и набором hosts, а также параметры tls, ссылающиеся на секрет с сертификатами.

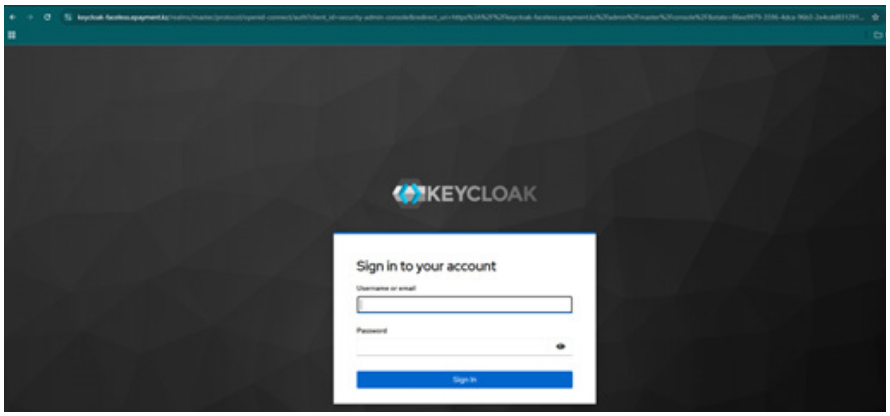

Рисунок 3 – Целевой хост с HTTPS endpoint

```
Name:         keycloak-gateway
Namespace:    istio-system
Labels:       <none>
Annotations:  <none>
API Version:  networking.istio.io/v1
Kind:         Gateway
Metadata:
  Creation Timestamp:  2025-10-20T07:03:49Z
  Generation:          1
  Resource Version:    29033127
  UID:                 2f275683-d70d-4dd8-95ab-626175a376a7
Spec:
  Selector:
    Istio:  ingressgateway
  Servers:
    Hosts:
      keycloak-faceless.epayment.kz
    Port:
      Name:      https
      Number:    443
      Protocol:  HTTPS
    Tls:
      Credential Name:  epayment-wildcard-tls
      Mode:             SIMPLE
Events:                 <none>
```

Рисунок 4 – Манифест Gateway с конфигурацией TLS

VirtualService описывает правила маршрутизации к backend-сервису, используя его FQDN, например keycloak.keycloak.svc.cluster.local. В VirtualService задаются условия сопоставления запросов (URI, заголовки, методы), правила распределения трафика между версиями, а также timeout и retry.

```
Name:         keycloak-vs
Namespace:    keycloak
Labels:       <none>
Annotations:  <none>
API Version:  networking.istio.io/v1
Kind:         VirtualService
Metadata:
  Creation Timestamp:  2025-10-20T06:55:01Z
  Generation:          4
  Resource Version:    29062762
  UID:                 f3fc0675-363d-47c3-9e1e-9df9da799032
Spec:
  Gateways:
    keycloak-gateway
  Hosts:
    keycloak-faceless.epayment.kz
  Http:
    Route:
      Destination:
        Host:  keycloak.keycloak.svc.cluster.local
        Port:
          Number:  8080
Events:            <none>
```

Рисунок 5 – Манифест VirtualService с правилами маршрутизации

DestinationRule задает политики взаимодействия с конкретным сервисом: стратегию балансировки нагрузки (например, LEAST_CONN), параметры connection pool, пороги circuit breaker и TLS-настройки для upstream-соединений. Subset-ы позволяют задавать отдельные политики для разных версий сервиса, выделяемых по labels.

```
Name:         keycloak-dr
Namespace:    keycloak
Labels:       <none>
Annotations:  <none>
API Version:  networking.istio.io/v1
Kind:         DestinationRule
Metadata:
  Creation Timestamp:  2025-10-27T10:05:43Z
  Generation:          1
  Resource Version:    33238027
  UID:                 2a2e5083-f18a-469f-9350-b626bd0cf154
Spec:
  Host:  keycloak.keycloak.svc.cluster.local
  Traffic Policy:
    Load Balancer:
      Simple:  LEAST_CONN
Events:        <none>
```

Рисунок 6 – Манифест DestinationRule с конфигурацией балансировки

Экосистема наблюдаемости

Istio интегрируется с распространенным Cloud Native-стеком наблюдаемости и предоставляет единое представление о состоянии mesh.

Prometheus собирает метрики от Envoy sidecar и компонентов control plane с заданным интервалом, формируя базу для мониторинга количества запросов, задержек, ошибок и использования ресурсов. На основе этих данных в Grafana строятся дашборды для mesh в целом, отдельных сервисов и workloads, а также отдельный дашборд для контроля состояния Istiod и xDS-потоков.

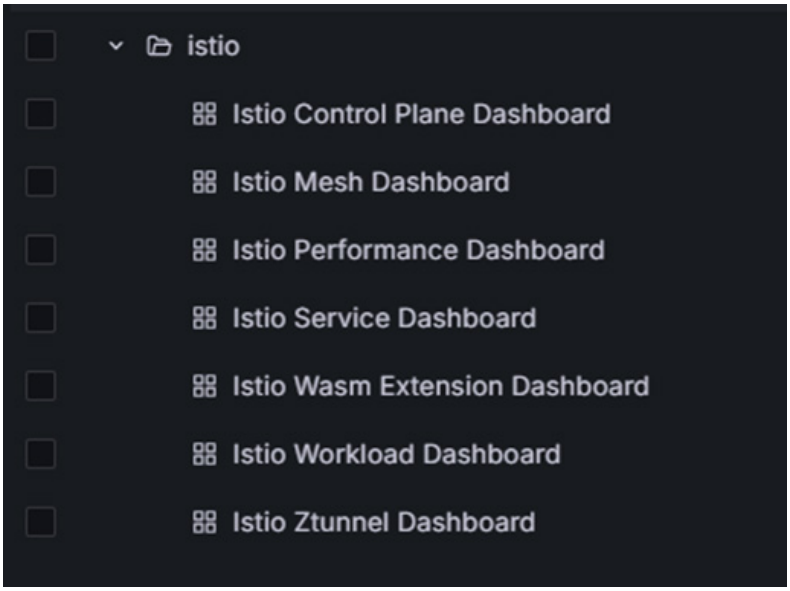

Рисунок 7 – Список дашбордов Grafana для Istio

Рисунок 8 – Istio Control Plane Dashboard: наблюдение
за состоянием control plane

Jaeger реализует распределенную трассировку: каждый запрос, проходящий через mesh, может быть представлен в виде цепочки span с временными метками и метаданными. Это позволяет выявлять узкие места и проблемные сервисы.

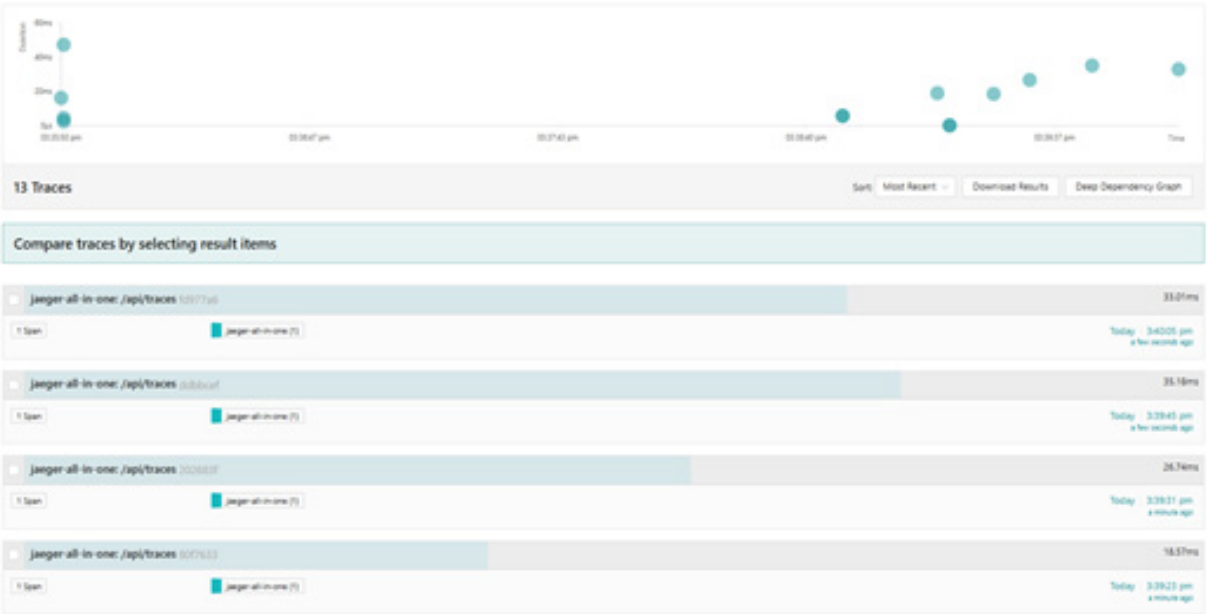

Рисунок 9 – Интерфейс Jaeger: визуализация распределенных трейсов

Kiali предоставляет графическую топологию mesh с отображением потоков трафика, метрики статуса сервисов, выполняет базовую валидацию конфигураций Istio и позволяет перейти от топологии к трассировкам и метрикам.

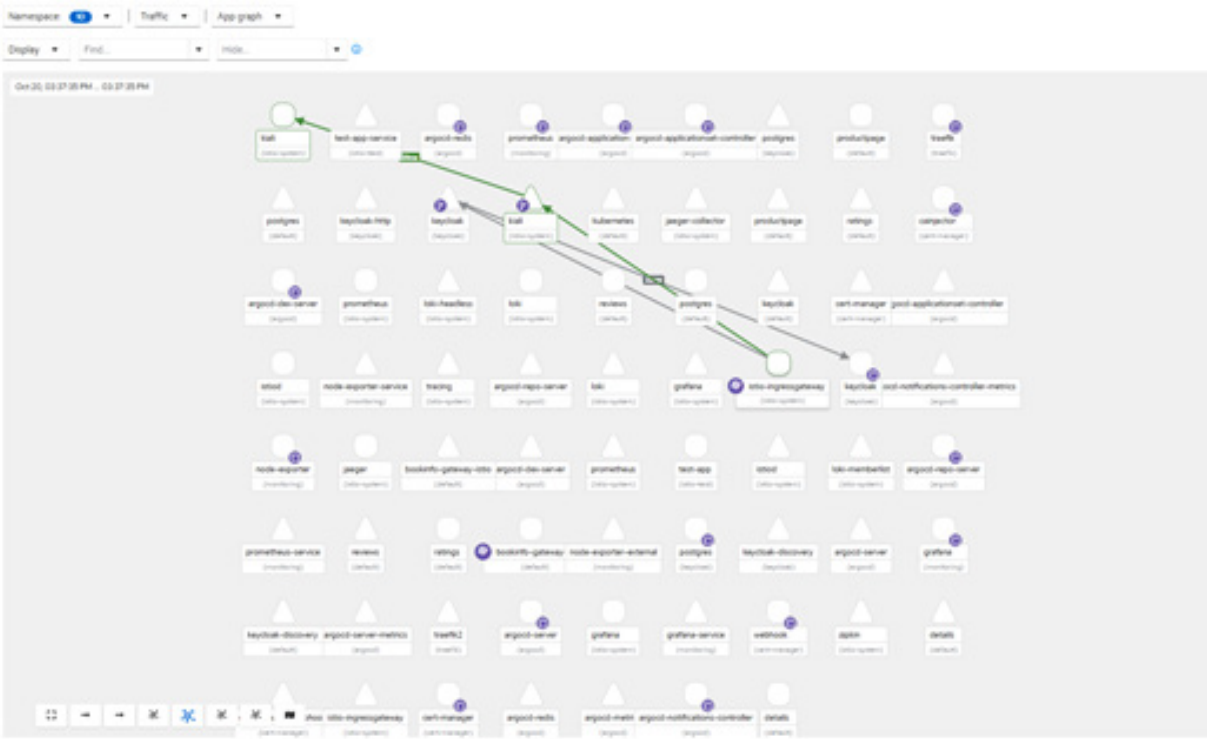

Рисунок 10 – Kiali: граф топологии mesh с метриками в реальном времени

Envoy sidecar генерируют метрики, трассировки и access logs, которые поступают в Prometheus, Jaeger и системы логирования, где связываются по trace ID и меткам Kubernetes. Это позволяет быстро переходить от алерта к конкретному проблемному запросу, оценивать загрузку сервисов и планировать ресурсы на основе фактических данных.

Заключение

Service Mesh представляет собой зрелый подход к управлению сложными микросервисными архитектурами. Он обеспечивает единообразную реализацию безопасности, маршрутизации, наблюдаемости и устойчивости на сетевом уровне, снимая с приложений необходимость содержать дублирующий инфраструктурный код.

Несмотря на накладные расходы по задержке и ресурсам, а также повышенную сложность эксплуатации, Service Mesh дает ощутимые преимущества для enterprise-систем с большим числом микросервисов и строгими требованиями к безопасности и наблюдаемости. Istio целесообразен в крупных корпоративных средах, Linkerd подходит для сценариев, где важны простота и минимальный overhead, Envoy может использоваться как базовый универсальный прокси и строительный блок для кастомных решений.

Практический пример конфигурации Istio показывает, что декларативная модель и интеграция с Cloud Native-стеком позволяют централизованно управлять трафиком и безопасностью, а также быстро диагностировать проблемы. Перспективные направления развития включают ambient mesh, использование eBPF и дальнейшую автоматизацию управления политиками и конфигурацией [22, 16].

Практический «порог входа» Service Mesh определяется балансом выгод (mTLS-by-default, единые политики, наблюдаемость, управление трафиком) и стоимости (ресурсы и сложность эксплуатации). В ориентировочных измерениях Istio один sidecar-proxy при 1000 rps и payload 1 KB потребляет порядка 0.20 vCPU и ~60 MB памяти (при 2 worker threads). Это дает грубую оценку:

Memory_overhead_GB ≈ Pods × 60 / 1024 (например, 500 pod → ~29.3 GB; 1000 pod → ~58.6 GB памяти только на прокси), плюс CPU-overhead.

Поэтому для небольших систем (условно: единицы сервисов, слабые требования к zero-trust и трассировке) часто достаточно ingress + NetworkPolicy + OpenTelemetry. Для средних систем (десятки сервисов) оправдан lightweight mesh (например, Linkerd) либо selective-injection. Для крупных enterprise-систем (сотни сервисов/тысячи pod) mesh оправдан при строгих требованиях к безопасности/наблюдаемости, но становится критично рассматривать оптимизации (селективная инъекция, scoping конфигураций, а также sidecarless/ambient-подходы, где узловой прокси (ztunnel) может иметь существенно меньший footprint).

Service Mesh рекомендуется к рассмотрению в организациях, эксплуатирующих десятки микросервисов и предъявляющих повышенные требования к безопасности и наблюдаемости, при условии готовности инвестировать в поэтапное внедрение и обучение инженерных команд.

Информация о финансировании. Исследование выполнено без внешней финансовой поддержки.

ЛИТЕРАТУРА

1 Cloud Native Computing Foundation. CNCF Annual Survey 2023 (2023). URL: https://www.cncf.io/reports/cncf-annual-survey-2023/ (accessed: 02.03.2026).

2 Newman, S. Building Microservices: Designing Fine-Grained Systems, 2nd ed. (Sebastopol: O'Reilly Media, 2021), 615 p.

3 Farkiani, B., Jain, R. Service Mesh: Architectures, Applications, and Implementations. arXiv preprint arXiv:2405.13333 (2024). URL: https://arxiv.org/abs/2405.13333 (accessed: 02.03. 2026).

4 Burns, B., Oppenheimer, D. Design Patterns for Container-based Distributed Systems. USENIX Workshop on Hot Topics in Cloud Computing (HotCloud '16) (2016). URL: https://www.usenix.org/node/196347 (accessed 02.03. 2026).

5 Zhu, X., She, G., Xue, B., et al. Dissecting Overheads of Service Mesh Sidecars. Proceedings of the ACM Symposium on Cloud Computing (SoCC '23), 142–157 (2023). https://doi.org/10.1145/3620678.3624652

6 Li, W., Lemieux, Y., Gao, J., Zhao, Z., and Han, Y. Service Mesh: Challenges, State of the Art, and Future Research Opportunities. Proceedings of the IEEE International Conference on Service-Oriented System Engineering (SOSE), 122–127 (2019). https://doi.org/10.1109/SOSE.2019.00026

7 Gregg, B. Systems Performance: Enterprise and the Cloud, 2nd ed. (Boston: Addison-Wesley Professional, 2020), 886 p.

8 Richardson, C. Microservices Patterns: With Examples in Java (Shelter Island: Manning Publications, 2018), 520 p.

9 Villamizar, M., Garcés, O., Castro, H., et al. Evaluating the Monolithic and the Microservice Architecture Pattern to Deploy Microservices on PaaS. Proceedings of the 12th Iberian Conference on Information Systems and Technologies (CISTI), 1–8 (2015). https://doi.org/10.1109/CISTI.2015.7170480

10 Pahl, C., Brogi, A., Soldani, J., and Jamshidi, P. Cloud Container Technologies: A State-of-the-Art Review. IEEE Transactions on Cloud Computing, 7 (3), 677–692 (2019). https://doi.org/10.1109/TCC.2017.2702586

11 Rose, S., Borchert, O., Mitchell, S., and Connelly, S. Zero Trust Architecture (NIST Special Publication 800-207) (Gaithersburg: National Institute of Standards and Technology, 2020), 50 p. https://doi.org/10.6028/NIST.SP.800-207

12 Beyer, B., Jones, C., Petoff, J., and Murphy, N. R. Site Reliability Engineering: How Google Runs Production Systems (Sebastopol: O'Reilly Media, 2016), 552 p.

13 Luksa, M. Kubernetes in Action, 2nd ed. (Shelter Island: Manning Publications, 2022), 856 p.

14 Sigelman, B. H., Barroso, L. A., Burrows, M., et al. Dapper, a Large-Scale Distributed Systems Tracing Infrastructure. Google Technical Report (2010). URL: https://research.google/pubs/pub36356/ (accessed: 02.03.2026).

15 OpenTelemetry Authors. OpenTelemetry: High-quality, portable telemetry to enable effective observability (2024). URL: https://opentelemetry.io/ (accessed: 02.03. 2026).

16 Calavera, D., Fontana, L. Linux Observability with BPF: Advanced Programming for Performance Analysis and Networking (Sebastopol: O'Reilly Media, 2019), 282 p.

17 Cilium Authors. Cilium Service Mesh (2024). URL: https://cilium.io/use-cases/service-mesh/ (accessed: 02.03.2026).

18 SPIFFE Authors. SPIFFE: Secure Production Identity Framework for Everyone (Specification) (2024). URL: https://spiffe.io/ (accessed: 02.03.2026).

19 Envoy Project Authors. Envoy Proxy: Modern HTTP Proxy and Communication Bus (Documentation) (2024). URL: https://www.envoyproxy.io/ (accessed: 02.03.2026).

20 Istio Authors. Istio: Connect, secure, control, and observe services (Documentation) (2024). URL: https://istio.io/ (accessed: 02.03.2026).

21 Linkerd Authors. Linkerd: Ultralight, security-first service mesh for Kubernetes (Documentation) (2024). URL: https://linkerd.io/ (accessed: 02.03. 2026).

22 Howard, J., et al. Introducing Ambient Mesh. Istio Blog (2022). URL: https://istio.io/latest/blog/2022/introducing-ambient-mesh/ (accessed: 02.03.2026).

23 Kubernetes SIG Network. Kubernetes Gateway API (2024). URL: https://gateway-api.sigs.k8s.io/ (accessed: 02.03.2026).

24 Istio Authors. Kubernetes Gateway API (Istio Documentation) (2024). URL: https://istio.io/latest/docs/tasks/traffic-management/ingress/gateway-api/ (accessed: 02.03.2026).

25 Istio Authors. Performance and Scalability (2024). URL: https://istio.io/latest/docs/ops/deployment/performance-and-scalability/ (accessed: 02.03.2026).

[1]*Кемелбеков Н.К.,
докторант, ORCID ID: 0009-0008-6784-3842,
*e-mail: kemellbekov@gmail.com
[1]Рахимова Д.Р.,
қауымдастырылған профессор, PhD, ORCID ID: 0000-0003-1427-198X,
e-mail: diana.rakhimova@kaznu.edu.kz
[2]Адали Е.,
эмерит-профессор, ORCID ID: 0000-0002-1561-8255,
e-mail: adali@itu.edu.tr

[1]Әл-Фараби атындағы Қазақ ұлттық университеті, Алматы қ., Қазақстан
[2]Ыстамбұл техникалық университеті, Ыстамбұл қ., Түркия

SERVICE MESH: АРХИТЕКТУРАЛЫҚ ҚАҒИДАТТАРЫ, ӨНІМДІЛІГІ ЖӘНЕ МИКРОСЕРВИСТІК ЖҮЙЕЛЕРДЕ ҚОЛДАНУ ТӘЖІРИБЕСІ

Аңдатпа

Мақалада қызмет көрсету торы технологиясы микросервис жүйелеріндегі қызметаралық байланысты басқаруға арналған арнайы инфрақұрылымдық қабат ретінде қарастырылады. Зерттеудің мақсаты – желілік қабатты құрудың архитектуралық принциптерін талдау, өнімділік шығындарын бағалау және оны Kubernetes кластерлеріне енгізудің практикалық тәсілдерін көрсету. Мақалада басқару жазықтығы мен деректер жазықтығын бөлу моделі, мөлдір трафикті бағыттау үшін sidecar үлгісін пайдалану, mTLS енгізу және телеметрияны автоматты түрде жинау сипатталған. Әдебиеттерге шолу және ашық бастапқы кодты құжаттама негізінде функционалдық мүмкіндіктері, енгізу күрделілігі және ресурстарды тұтынуы бойынша үш негізгі іске асырудың – Envoy, Istio және Linkerd – салыстырмалы талдауы келтірілген. HTTPS трафигін басқару және Prometheus–Grafana–Jaeger–Kiali бақылау стегімен біріктіру үшін Gateway, VirtualService және DestinationRule манифесттерін пайдаланатын Istio конфигурациясының практикалық мысалы ұсынылған. Зерттеу нәтижелері Service Mesh технологиясының кідіріс пен ресурстарды тұтыну тұрғысынан орташа шығындарды арттырғанымен, микросервистер арасындағы қауіпсіздік пен трафикті орталықтандырылған түрде басқаруды қамтамасыз ететінін көрсетті. Алынған нәтижелер жүйенің масштабы мен қауіпсіздік талаптарына байланысты кәсіпорындық микросервис платформаларында Service Mesh технологиясын қолданудың орындылығы бойынша практикалық ұсыныстар жасауға мүмкіндік береді.

Түйін сөздер: Service Mesh, микросервистер, Kubernetes, Istio, Envoy, Linkerd, mTLS, бақылау, таралған жүйелер.

[1]*Kemelbekov N.K.,
doctoral student, ORCID ID: 0009-0008-6784-3842,
*e-mail: kemellbekov@gmail.com
[1]Rakhimova D.R.,
Associate Professor, PhD, ORCID ID: 0000-0003-1427-198X,
e-mail: diana.rakhimova@kaznu.edu.kz
[2]Adali E.,
Professor Emeritus, ORCID ID: 0000-0002-1561-8255,
e-mail: adali@itu.edu.tr

[1]Al-Farabi Kazakh National University, Almaty, Kazakhstan
[2]Istanbul Technical University, Istanbul, Türkiye

SERVICE MESH: ARCHITECTURAL PRINCIPLES, PERFORMANCE AND PRACTICAL USE IN MICROSERVICE SYSTEMS

Abstract

This paper examines service mesh technology as a dedicated infrastructure layer for managing interservice communication in microservice systems. The objective of the study is to analyze the architectural principles of network

layer construction, estimate performance overhead, and demonstrate practical approaches for operationalization in Kubernetes clusters. It describes a model for separating the control plane and data plane, using the sidecar pattern for transparent traffic routing, implementing mTLS, and automatically collecting telemetry. Based on a literature review and open source documentation, a comparative analysis of three key implementations – Envoy, Istio, and Linkerd – is provided based on functionality, implementation complexity, and resource consumption. A practical example of an Istio configuration is provided, using Gateway, VirtualService, and DestinationRule manifests to manage HTTPS traffic and integrate with the Prometheus – Grafana – Jaeger – Kiali observability stack. It was shown that Service Mesh adds moderate overhead in terms of latency and resource consumption, but provides unified management of security and traffic between microservices. The obtained results allow us to formulate practical recommendations on the feasibility of using Service Mesh in enterprise microservice platforms, depending on the system scale and security requirements.

Keywords: Service Mesh, microservices, Kubernetes, Istio, Envoy, Linkerd, mTLS, observability, distributed systems.

Received December 12, 2025; revised March 2, 15, 21, 2026; accepted May 12, 2026