РЕФЕРАТ

Расчетно-пояснительная записка 59 с., 7 рис., 8 источн., 1 прил.

АУТЕНТИФИКАЦИЯ, АВТОРИЗАЦИЯ, МИКРОСЕРВИСЫ, ИНФРАСТРУКТУРНЫЕ СЕРВИСЫ, KUBERNETES, КЛАСТЕР, SIDECAR, OIDC, OAUTH2.0, REST API,

Цель работы: реализация программно-алгоритмического комплекса по авторизации инфраструктурных сервисов.

СОДЕРЖАНИЕ

ПЕРЕЧЕНЬ СОКРАЩЕНИЙ И ОБОЗНАЧЕНИЙ

В настоящей расчетно-пояснительной записке к выпускной квалификационной работе применяют следующие сокращения и обозначения:

k8s	Kubernetes — программное обеспечение для оркестрирования контейнеризированных приложений — автоматизации их развертывания, масштабирования и координации в условиях кластера
Sidecar	Паттерн сайдкар контейнера, при котором в одной сущности существует прокси-контейнер, расширяя возможности основного контейнера
idP	identity provider — ключевая точка авторизации, запросы аутентификации проходят через нее, и в ней же выписываются OIDC токены

ВВЕДЕНИЕ

Последнее время все чаще можно услышать об утечке чувствительных персональных данных пользователей или компаний, занимающихся в таких сферах как финансы, здравоохранение, коммерция. Отчасти это происходит из-за того, что компания и ее сотрудники небрежно следят за соблюдением информационной безопасности.

Информационная безопасность играет критически важную роль в распределенных системах с микросервисной архитектурой, поскольку такие системы часто состоят из множества независимых, взаимодействующих компонентов. Микросервисы обрабатывают и хранят большое количество данных, включая личную информацию пользователей и конфиденциальные бизнес-данные. Компрометация доступов к внутренним информационным системам и данным представляет собой финансовые, репутационные и правовые риски для компаний

В таких системах есть необходимость в эффективных и безопасных механизмах аутентификации и авторизации действий, доступных одному микросервису по отношению к другому, чтобы ограничить действия злоумышленника, получившего внутренний доступ. Особенно это касается инфраструктурных микросервисов — там, где доступ к данным совершается наиболее часто.

В данной работе будет реализована система авторизации инфраструктурных сервисов в системах с микросервисной архитектурой, чтобы исключить один из возможных этапов утечки чувствительных данных — несогласованный доступ как внутреннего сотрудника, так и злоумышленника извне систем. При этом работа авторизация не должна оказывать существенного влияния на работу системы, так как может быть внедрена в высоконагруженные системы.

Цель выпускной квалификационной работы — реализация программно-алгоритмического комплекса для авторизации запросов в инфраструктурные сервисы.

Задачи выпускной квалификационной работы:

1) провести обзор существующих решений аутентификации и авторизации в микросервисной архитектуре;

2) рассмотреть основные протоколы аутентификации, применимые в микросервисной архитектуре;

3) разработать и описать ключевые алгоритмы работы программно-алгоритмического комплекса авторизации инфраструктурных микросервисов;

4) разработать программное обеспечение, реализующее аутентификацию и авторизацию инфраструктурных микросервисов;

5) провести исследование влияния работы авторизации на выполнения запросов между инфраструктурными сервисами.

1 Аналитический раздел

1.1 Микросервисная архитектура

Микросервисы – это архитектурный и организационный подход к разработке программного обеспечения, при котором программное обеспечение состоит из небольших не зависимых сервисов, взаимодействующих через четко определенные интерфейсы API, обычно основанные на протоколах HTTP или gRPC. Эти сервисы принадлежат небольшим автономным командам. Архитектуры микросервисов упрощают масштабирование и ускоряют разработку приложений, позволяя внедрять инновации и ускоряя вывод новых функций на рынок.

В монолитных архитектурах все процессы тесно связаны и работают как единая служба. Это означает, что если один процесс приложения испытывает всплеск спроса, необходимо масштабировать всю архитектуру. Добавление или улучшение функций монолитного приложения усложняется по мере роста базы кода. Эта сложность ограничивает экспериментирование и затрудняет реализацию новых идей. Монолитные архитектуры повышают риск доступности приложений, поскольку множество зависимых и тесно связанных процессов увеличивают влияние сбоя одного процесса.

В архитектуре микросервисов приложение строится как независимые компоненты, которые запускают каждый процесс приложения как службу. Эти сервисы взаимодействуют через четко определенный интерфейс с использованием облегченных API. Службы созданы для бизнес-возможностей, и каждая служба выполняет одну функцию. Поскольку они запускаются независимо, каждую службу можно обновлять, развертывать и масштабировать в соответствии со спросом на определенные функции приложения. [1]

На рисунке изображено отличие в монолитном и микросервисных подходах 1.1.

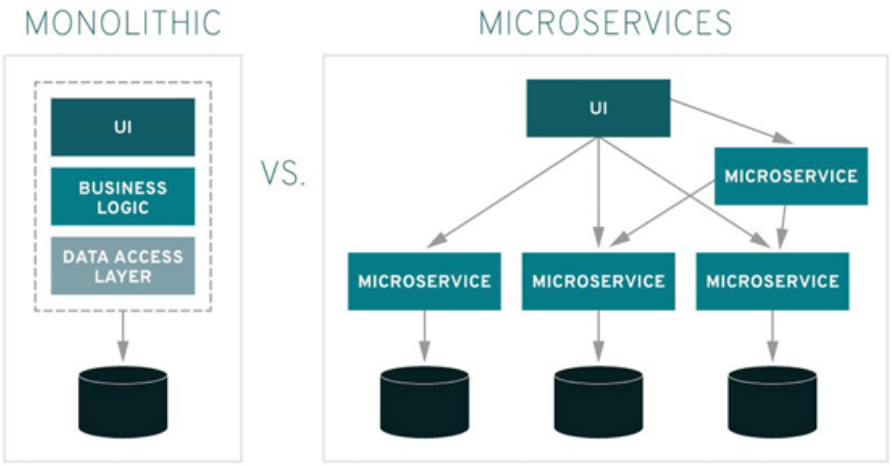

Рисунок 1.1 – Монолитная и микросервисная архитектура

Безопасность в микросервисных системах представляет собой важный аспект, обусловленный характером архитектуры, которая состоит из множества взаимодействующих компонентов. Каждый микросервис, функционирующий как автономная единица, взаимодействует с другими сервисами через API, что создает множество потенциальных точек уязвимости. Каждая точка входа является возможностью для несанкционированного доступа злоумышленников, стремящихся воспользоваться недостатками в системе. Компрометация одного микросервиса может иметь цепной эффект, влияя на другие компоненты системы. Кроме того, микросервисы часто обрабатывают конфиденциальную информацию, что обуславливает необходимость реализации шифрования как в состоянии покоя, так и в процессе передачи данных. [2]

1.2 Идентификация, аутентификация и авторизация

Идентификация представляет собой процесс, в ходе которого система определяет пользователя или объект, предоставляя уникальный идентификатор, такой как логин или номер учетной записи. Этот этап является первым шагом в установлении личности, позволяя системе распознать, с кем она взаимодействует, и подготовить соответствующий механизм контроля доступа.

Аутентификация представляет собой процесс проверки подлинности идентифицированного пользователя или объекта. На этом этапе система требует от пользователя подтверждения своей личности, что может осуществляться через различные методы, например ввод пароля. Прохождение аутентификация обеспечивает уверенность в том, что пользователь действительно является тем, за кого себя выдает.

Авторизация заключается в предоставлении пользователю системы прав доступа к определенным ресурсам или операциям на основании его роли, прав или политик безопасности. Этот процесс определяет, какие действия пользователь может выполнять и к каким ресурсам он имеет доступ, что позволяет минимизировать риски, связанные с несанкционированным доступом.

В рамках этих процессов важную роль играет концепция управления идентификацией и доступом (Identity and Access Management, IAM), которая включает в себя стратегии, технологии и практики, направленные на эффективное управление идентификацией пользователей и их правами доступа в организации. IAM обеспечивает централизованное управление учетными записями пользователей, автоматизацию процессов аутентификации и авторизации. [3]

В условия микросервисной архитектуры, когда система состоит из множества независимых компонентов, каждый из которых может обрабатывать запросы от пользователей и других сервисов, есть необходимость в надежной аутентификации. В микросервисах аутентификация в основном реализуется через токены, такие как JSON Web Tokens (JWT), которые обеспечивают безопасный способ передачи информации, шифруя содержимое, о роли и правах одного микросервиса по отношению к другому.

Аутентификация также должна быть интегрирована с механизмами управления доступом, что позволяет не только удостовериться в личности пользователя, совершающего запрос, но и определить его права на доступ к определенным ресурсам и операциям.

Аутентификация в микросервисах должна быть масштабируема, справляясь с ростом количества микросервисов в системе, при этом не теряя в уровне безопасности. [4]

1.3 Основные понятия Kubernetes

Kubernetes — открытый механизм оркестрации контейнеров для автоматизации развертывания, масштабирования и управления контейнеризированными приложениями [5].

Работающий кластер Kubernetes включает в себя агента, запущенного на нодах (kubelet) и компоненты мастера (APIs, scheduler, etc), поверх решения с распределённым хранилищем. Пример схемы представлен на рисунке **??**.

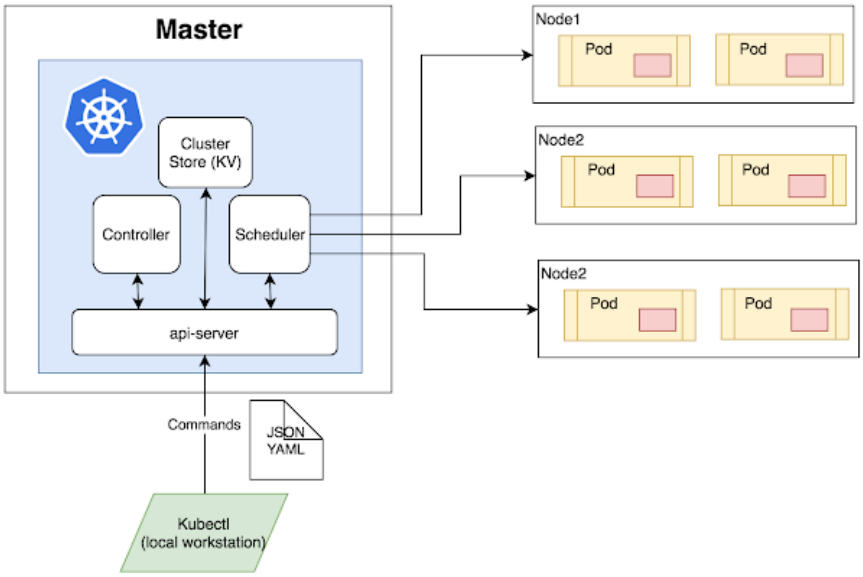

Рисунок 1.2 – Схема инфраструктуры Kubernetes

Узел (node) — это отдельная физическая или виртуальная машина, на которой развёрнуты и выполняются контейнеры приложений. Каждый узел в кластере содержит сервисы для запуска приложений в контейнерах (например Docker), а также компоненты, предназначенные для централизованного управления узлом.

Под (pod) — базовая единица для запуска и управления приложениями: один или несколько контейнеров, которым гарантирован запуск на одном узле, обеспечивается разделение ресурсов и межпроцессное взаимодействие и предоставляется уникальный в пределах кластера IP-адрес. Последнее позволяет приложениям, развёрнутым на поде, использовать фиксированные и предопределённые номера портов без риска конфликта. Поды могут напрямую управляться с использованием API Kubernetes или управление ими может быть передано контроллеру [5].

Том (volume) — общий ресурс хранения для совместного использования из контейнеров, развёрнутых в пределах одного пода.

Все объекты управления (узлы, поды, контейнеры) в Kubernetes помечаются метками (label), селекторы меток (label selector) — это запросы, которые позволяют получить ссылку на объекты, соответствующие какой-то из меток. Метки и селекторы — это главный механизм Kubernetes, который позволяет выбрать, какой из объектов следует использовать для запрашиваемой операции [5].

Сервисом в Kubernetes называют совокупность логически связанных

наборов подов и политик доступа к ним. Например, сервис может соответствовать одному из уровней программного обеспечения, разработанного в соответствии с принципами многоуровневой архитектуры программного обеспечения. Набор подов, соответствующий сервису, получается в результате выполнения селектора соответствующей метки.

Kubernetes обеспечивает функции обнаружения сервисов и маршрутизации по запросу. В частности, система умеет переназначать необходимые для обращения к сервису IP-адрес и доменное имя сервиса различным подам, входящим в его состав. При этом обеспечивается балансировка нагрузки в стиле Round robin DNS между подами, чьи метки соответствуют сервису, а также корректная работа в том случае, если один из узлов кластера вышел из строя и размещённые на нём поды автоматически были перемещены на другие узлы. По умолчанию сервис доступен внутри управляемого Kubernetes кластера — например, поды бэкенда группируются для обеспечения балансировки нагрузки и в таком виде предоставляются фронтенду. Также кластер может быть настроен и для предоставления доступа к входящим в его состав подам извне как к единому фронтенду [5].

Контроллер (controller) — это процесс, который управляет состоянием кластера, пытаясь привести его от фактического состояния к желаемому; он делает это, оперируя набором подов, определяемых с помощью селекторов меток и являющихся частью определения контроллера. Выполнение контроллеров обеспечивается компонентом Kubernetes Controller Manager. Один из типов контроллеров, самый известный — это контроллер репликации (Replication Controller), который обеспечивает масштабирование, запуская указанное количество копий пода в кластере. Он также обеспечивает запуск новых экземпляров пода в том случае, если узел, на котором работает управляемый этим контроллером под, выходит из строя. Другие контроллеры, входящие в основную систему Kubernetes, включают в себя «DaemonSet Controller», который обеспечивает запуск пода на каждой машине (или подмножеством машин), и «Job Controller» для запуска подов, которые выполняются до завершения, например, как часть пакетного задания [5].

Операторы (operators) — специализированный вид программного обеспечения Kubernetes, предназначенный для включения в кластер сервисов, сохраняющих своё состояние между выполнениями (stateful), таких как СУБД,

системы мониторинга или кэширования. Назначение операторов — предоставить возможность управления stateful-приложениями в кластере Kubernetes прозрачным способом и скрыть подробности их настроек от основного процесса управления кластером Kubernetes [5].

Токен Service Account (SA) — объект Kubernetes, предоставляющий идентичность для внутрикластерных процессов (например, подам). Управляется Kubernetes API, связан с RBAC для контроля доступа, токены SA верифицируются API-сервером [5].

Сертификат Certificate Authority (CA) — корневой центр сертификации, выпускающий TLS-сертификаты для компонентов кластера (apiserver, etcd, kubelet). Приватный ключ CA хранится защищенно (часто на control-plane нодах), сертификаты компонентов подписываются CA, обеспечивая криптографическую идентичность. Предустановка root CA-сертификата в компонентах гарантирует цепочку доверия [5].

Сертификат Kubernetes API JWKs — публичные ключи в формате JWKS, используемые для верификации JWT-токенов Service Accounts. Ключи генерируются и публикуются API-сервером (/openid/v1/jwks), подлинность JWKS гарантируется TLS (с валидацией по CA кластера). Динамическое обновление ключей контролируется apiserver [5].

Sidecar-контейнер — это контейнер, который должен быть запущен рядом с основным контейнером внутри пода. Этот паттерн нужен для расширения и улучшения функциональности основного приложения без внесения в него изменений. Sidecar-контейнеры могут использоваться для выполнения вспомогательных функций для main-контейнера: логирования, мониторинга, аутентификации, фоновых процессов или тестирования функциональности приложений [6].

Sidecar-контейнеры размещаются в той же внутренней сети, что и main-контейнер. Основное приложение может обращаться к sidecar-контейнеру, используя localhost с указанием порта, на котором работает sidecar-контейнер. Порты main- и sidecar-контейнера должны быть разными [6].

1.4 Обзор существующих решений

1.4.1 Service Mesh

Service Mesh — слой инфраструктуры (data plane + control plane), управляющий сетевым взаимодействием между сервисами в кластере Kubernetes, обеспечивает mTLS, балансировку, наблюдаемость и базовую авторизацию. Пример инфраструктуры приведен на рисунке **??**. Внедряется через sidecar-прокси (Envoy в Istio). Istio - это открытая платформа сервисной сетки, предоставляющая единый слой управления сетевыми взаимодействиями между микросервисами. Она включает управляющую плоскость (Control Plane) и плоскость данных (Data Plane), которые работают совместно для достижения функциональности сетки [7].

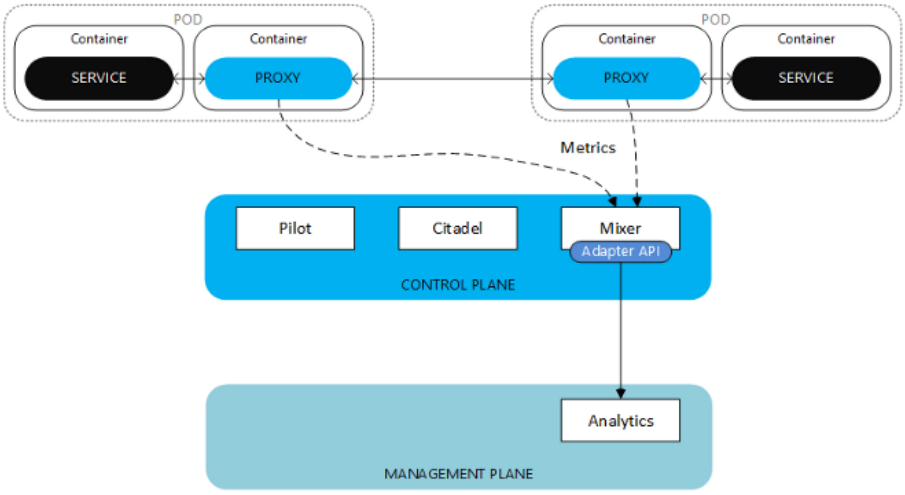

Рисунок 1.3 – Инфраструктура Service Mesh

Аутентификаця базируется на автоматически генерируемых X.509 сертификатах, привязанных к Kubernetes ServiceAccounts. Control plane (Istiod) выступает в роли корневого CA, выпуская короткоживущие сертификаты для sidecar [8].

Авторизация в Istio реализуется через AuthorizationPolicy, позволяющий задавать правила на основе идентификатора источника и http-атрибутов [7].

К недостаткам можно отнести:

1) авторизация L7 работает исключительно для HTTP/gRPC, что делает её неприменимой для нативных протоколов PostgreSQL, Kafka, Redis и других инфраструктурных сервисов, использующие свои собственные протоколы;

2) отсутствует централизованное управление политиками для разнородных систем — настройка доступа к БД и брокерам требует отдельных механизмов;

3) верификация сложных политик требует обращений к control plane [7].

1.4.2 SPIFFE/SPIRE

SPIFFE (Secure Production Identity Framework For Everyone) – стандарт для унифицированной идентификации рабочих нагрузок в гетерогенных средах. SPIRE (SPIFFE Runtime Environment) – эталонная реализация, функционирующая как доверенная система выпуска идентификаторов. Решение агностично к платформе оркестрации, поддерживая Kubernetes, виртуальные машины и bare-metal. Архитектура включает сервер SPIRE (управление идентичностью) и агентов (выпуск SVID на нодах). Сертификаты и токены SVID (X.509/JWT) ограничены временем жизни не более часа, совместимы со стандартом OIDC. Проверка подписи SVID осуществляется без обращения к центральным серверам [9].

Рабочая нагрузка для аутентификации регистрируется в SPIRE-агенте через плагин Workload Attestor (например, на основе Kubernetes ServiceAccount Token). После валидации выпускается SVID. Авторизация при этом не является частью SPIFFE и требует интеграции с внешними PDP (точка политики решения), например Open Policy Agent [10].

В итоге можно сказать, что SPIFFE обеспечивает только аутентификацию, делегируя авторизацию внешним системам, чем становится сложнее управлять. Также отсутствует нативная поддержка авторизации для инфраструктурных сервисов, потребуются кастомные плагины [10]. Динамическое управление правами ограничено необходимостью синхронизации между SPIRE и PDP, что может приводить к рассинхронизации реальных прав и фактических.

1.4.3 Встроенные RBAC БД

Категория объединяет протокол-специфичные механизмы управления доступом, которые используются непосредственно в инфраструктурных сервисах. Реализации включают: ролевую модель PostgreSQL (GRANT/REVOKE), ACL Kafka (kafka-acls), ACL Redis, SQL-driven RBAC ClickHouse и пр. Каждый

сервис предоставляет свой уникальный API для управления политиками, не интегрированный с Kubernetes [11].

Проверка прав осуществляется на уровне обработки запроса без внешних вызовов. Также отсутствуют внешние зависимости, так как вся проверка выполняется непосредственно в сервисе. При этом изменение правил часто требует перезапуска сервиса. Аутентификация осуществляется через статические секреты, например, логин и пароль у PostgreSQL, SASL-ключи у Kafka. При этом имеются статические TLS, обеспечивающие взаимную аутентификацию (mutual TLS) [11].

Можно сделать вывод, что правами в таких системах управлять тяжело, так как они отличаются от сервиса к сервису и имеют свои собственные протоколы для обмена информации. Автоматическая ротация секретов также не предусмотрена, секреты хранятся статически.

1.4.4 OAuth2 Proxy с Open Policy Agent

Комбинированное решение, где OAuth2 Proxy выполняет аутентификацию запросов через стандартные OIDC-потоки, а Open Policy Agent (OPA) обеспечивает авторизацию на основе политик Rego. Архитектура предполагает развёртывание OAuth2 Proxy в качестве ингресс-шлюза перед защищаемыми сервисами. OPA функционирует как отдельный сервис или sidecar [11].

OAuth2 Proxy проверяет токены через IdP (Keycloak, Okta), поддерживаются разные flow авторизации, такие как Client Crenedials и Authorization Code. Для авторизации OAuth2 Proxy отправляет атрибуты запроса (JWT, метод, путь) в OPA, где затем оцениваются политики на языке Rego. В листинге 1.1 приведен пример проверки прав на основе политик.

Листинг 1.1 – Оценка политики на языке Rego

```
default allow = false
allow { input.jwt.claims.access[_] == "postgres:orders:SELECT" }
```

К преимуществам этого решения можно отнести:

1) Rego используется как унифицированный язык политик для всех сервисов;

2) поддерживает стандарты OIDC, OAuth 2.0, JWT;

3) политики OPA обновляются через API, что делает удобным управление прав;

Решение обладает следующими недостатками:

1) каждый запрос на авторизацию требует синхронного вызова к OPA, добавляя нагрузку;

2) к разным сервисам потребуется разработка специализированных под конкретный сервис адаптер между HTTP-запросом и кастомизированным протоколом сервиса;

3) автоматическая ротация секретов не поддерживается, управление ключами OAuth2-клиентов осуществляется вручную.

1.4.5 Сравнение

Для сравнения рассмотренных решений можно выделить следующие критерии:

— централизация — единая точка управления правами для всех типов инфраструктурных сервисов;

— локальная валидация — проверка подлинности без сетевых вызовов к внешним сервисам;

— ротация — регулярная автоматическая смена ключей/токенов через определенный промежуток;

— универсальность — возможность использовать решение в разных инфраструктурных сервисах, решение не должно быть разработано под какой-то исключительно один или два сервиса.

Таблица 1.1 – Сравнение существующих решений

Решение	**Централи-зация**	**Локальная валида-ция**	**Ротация**	**Универсаль-ность**
Service Mesh	нет	нет	да	нет
SPIFFE/SPIRE	нет	да	да	да
Встроенные RBAC БД	нет	да	нет	нет
OAuth2 Proxy с OPA	да	нет	нет	нет

Судя по результатам сравнения, ни одно из рассмотренных решений не удовлетворяет сразу всем критериям.

1.5 Протоколы аутентификации

В данном разделе будут рассмотрены протоколы аутентификации, структуры и сущности, которыми они оперируют, и их flow. Flow авторизации — это последовательность шагов, которые проходят объект авторизации, в данном случае микросервисы, для получения доступа к защищенным ресурсам. Все протоколы применяются как системах с пользователем и сервером, так и в распределенных системах с микросервисной архитектурой.

1.5.1 OAuth 1.0 и OAuth 2.0

OAuth (Open Authorization) — это открытый стандарт, разработанный для делегирования доступа к защищенным ресурсам без необходимости передачи учетных данных пользователя. Он стал основой для более совершенной версии — OAuth 2.0, выпущенной в 2012 году, описанный в стандарте RFC-6749 [12]. OAuth 2.0 расширяет возможности своего предшественника, предлагая более гибкую архитектуру и упрощенные механизмы авторизации, один из самых используемых в современных веб-приложениях и API. [13]

Протокол OAuth включает несколько ключевых сущностей:

1) ресурсный владелец (Resource Owner) — это пользователь, который предоставляет доступ к своим защищенным ресурсам;

2) клиент (Client) — приложение, которое запрашивает доступ к защищенному ресурсу от имени resource owner и с его разрешения;

3) сервер авторизации (Authorization Server) — сервер, который отвечает за аутентификацию ресурсного владельца и выдачу токенов доступа. Он управляет процессом авторизации и хранит учетные данные пользователей;

4) ресурсный сервер (Resource Server) — это сервер, который хранит защищенные ресурсы и принимает запросы на доступ к ним на основании токенов доступа, выданных авторизационным сервером. Сервис может быть одновременно как клиентом, так и сервером.

OAuth 2.0 основывается на концепции токенов доступа, например JWT, которые используются для авторизации запросов к защищенным ресурсам. Основными компонентами структуры OAuth являются:

1) токен доступа (Access Token) — это строка с зашифрованным содержанием, которая представляет собой разрешение на доступ к защищенным ресурсам. Токены доступа имеют ограниченный срок действия и могут быть отозваны;

2) токен обновления (Refresh Token) — это токен, который используется для получения нового токена доступа после истечения его срока действия. Токены обновления позволяют клиенту продолжать доступ к ресурсам без повторной аутентификации пользователя;

3) клиентский идентификатор (Client ID) и секрет клиента (Client Secret) — эти идентификаторы используются для аутентификации клиента на авторизационном сервере.

OAuth 2.0 поддерживает несколько потоков авторизации (authorization flows), каждый из которых предназначен для определенных сценариев использования. Из них можно выделить основные:

1) Authorization code — этот flow основан на редиректах. Клиент должен уметь взаимодействовать с user-agent (обычно браузером) и обеспечивать клиент-серверное взаимодействие;

2) Client Credentials — этот flow больше остальных подходит для микросервисов. Авторизация выполняется на основе client id и client secret, что по сути из себя представляет логин и пароль, по которым сервер авторизации выдает клиенту access token к запрашиваемому ресурсу. На рисунке **??** представлен данный flow;

3) Implicit — по логике похож на Authorization code, но с тем отличием, что после успешной авторизации resource owner вместо получения authorization-code и его обмена на access token, сервер авторизации сразу возвращает access token в query в redirect URI. [12]

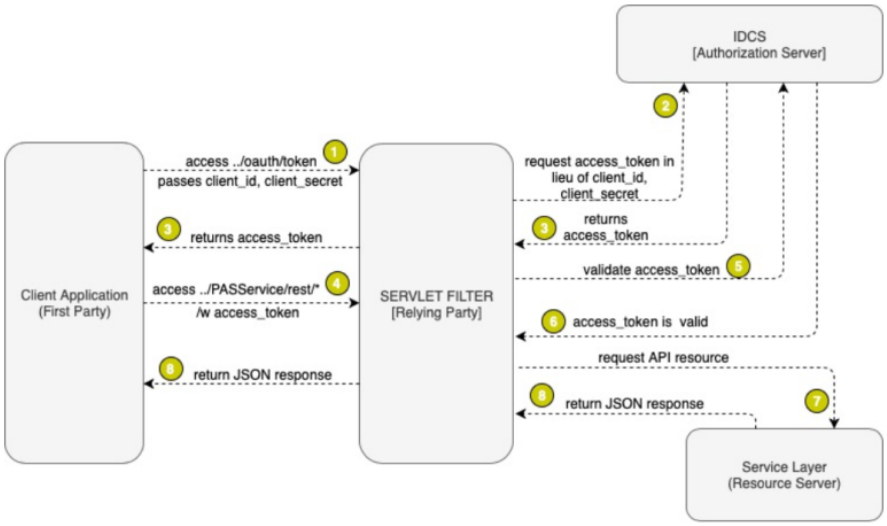

Рисунок 1.4 – OAuth2.0 Client Credentials flow авторизации

1.5.2 OpenID Connect

OpenID Connect (OIDC) представляет собой протокол аутентификации, построенный на основе OAuth 2.0, который позволяет клиентским приложениям проверять личность пользователей на основе аутентификации, выполненной авторизационным сервером. OAuth2.0 и OIDC часто используют вместе. [14]

Данный протокол содержит все те сущности и структуру, описанную в OAuth протоколе. Основные отличия данных протоколов:

1) ID Token — это токен, который содержит в зашифрованном виде информацию о пользователе, а также данные о сессии аутентификации. ID Token является основным элементом OpenID Connect и обычно кодируется в формате JWT (JSON Web Token). В OAuth2.0 токен не содержит информации о клиенте;

2) flow oidc позволяют клиентам получать как Access, так и ID Token;

3) в поле применения OIDC используется больше как протокол аутентификации, OAuth2.0 в связке с ним используется как протокол авторизации.

1.5.3 SAML

Протокол SAML (Security Assertion Markup Language) представляет собой стандарт, разработанный для обмена аутентификационной и авторизационной информацией между различными доменами безопасности.

Можно выделить основные сущности в данном протоколе:

1) Актор (Principal) — это пользователь или субъект, который пытается получить доступ к защищенным ресурсам. Актор инициирует процесс аутентификации;

2) Identity Provider (IdP) — это система, которая отвечает за аутентификацию акторов и выдачу утверждений (assertions). IdP проверяет личность пользователя и создает SAML-утверждения, которые содержат информацию о пользователе;

3) Service Provider (SP) — это система, которая предоставляет доступ к защищенным ресурсам. SP полагается на утверждения, выданные IdP, для принятия решения о том, предоставлять ли доступ пользователю;

4) SAML-утверждение (SAML Assertion) — это XML-документ, который содержит информацию о пользователе и его аутентификации. Утверждения могут содержать данные о том, как и когда пользователь был аутентифицирован, а также атрибуты, которые описывают пользователя.

Также можно выделить основную структуру протокола SAML:

1) SAML-Запрос (SAML Request) — сообщение, отправляемое SP в IdP с запросом на аутентификацию пользователя. Запрос может быть представлен в виде HTTP-запроса, содержащего параметры, такие как идентификатор запроса и URL-адрес, на который должно быть отправлено ответное сообщение;

2) SAML-Ответ (SAML Response) — сообщение, отправляемое IdP обратно в SP после успешной аутентификации пользователя. Ответ содержит SAML-утверждение, которое подтверждает аутентификацию и может содержать атрибуты пользователя;

3) XML — все сообщения SAML формализуются в виде XML-документов, что обеспечивает структурированный формат для обмена данными.

SAML имеет следующий flow авторизации, представленный на рисунке **??**:

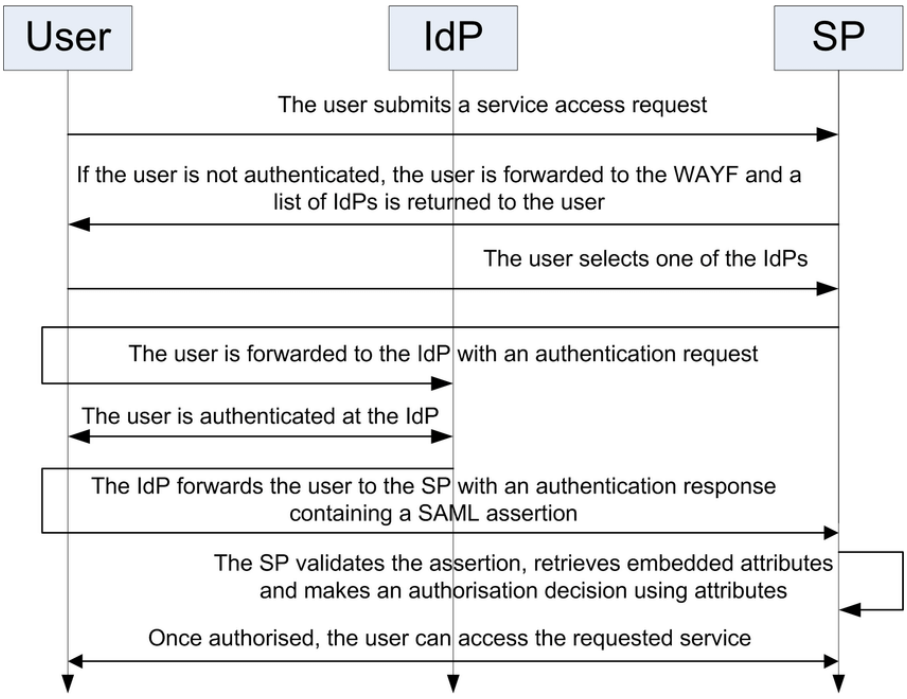

Рисунок 1.5 – SAML flow авторизации

1) инициация запроса на аутентификацию — актор пытается получить доступ к ресурсу на SP, который определяет, что актор не аутентифицирован, и перенаправляет его на IdP с SAML-запросом на аутентификацию. Запрос обычно кодируется в URL и передается через HTTP-редирект;

2) аутентификация пользователя — IdP получает SAML-запрос и инициирует процесс аутентификации. После успешной аутентификации IdP создает SAML-утверждение, которое содержит информацию о пользователе;

3) отправка SAML-ответа — IdP формирует SAML-ответ, который включает

в себя SAML-утверждение, и перенаправляет актера обратно к SP. Ответ может быть передан через HTTP POST или HTTP Redirect;

4) обработка SAML-ответа — SP получает SAML-ответ и проверяет его подлинность. Если SAML-утверждение действительно, SP выполняет авторизацию пользователя на основе атрибутов, содержащихся в утверждении (например, имя, адрес электронной почты, роли и права доступа);

5) доступ к ресурсу — после успешной авторизации SP предоставляет доступ к защищенным ресурсам актору. [15]

1.6 Постановка задачи

Формализованная постановка задачи в виде IDEF-0 диаграммы нулевого уровня представлена рисунке 1.6:

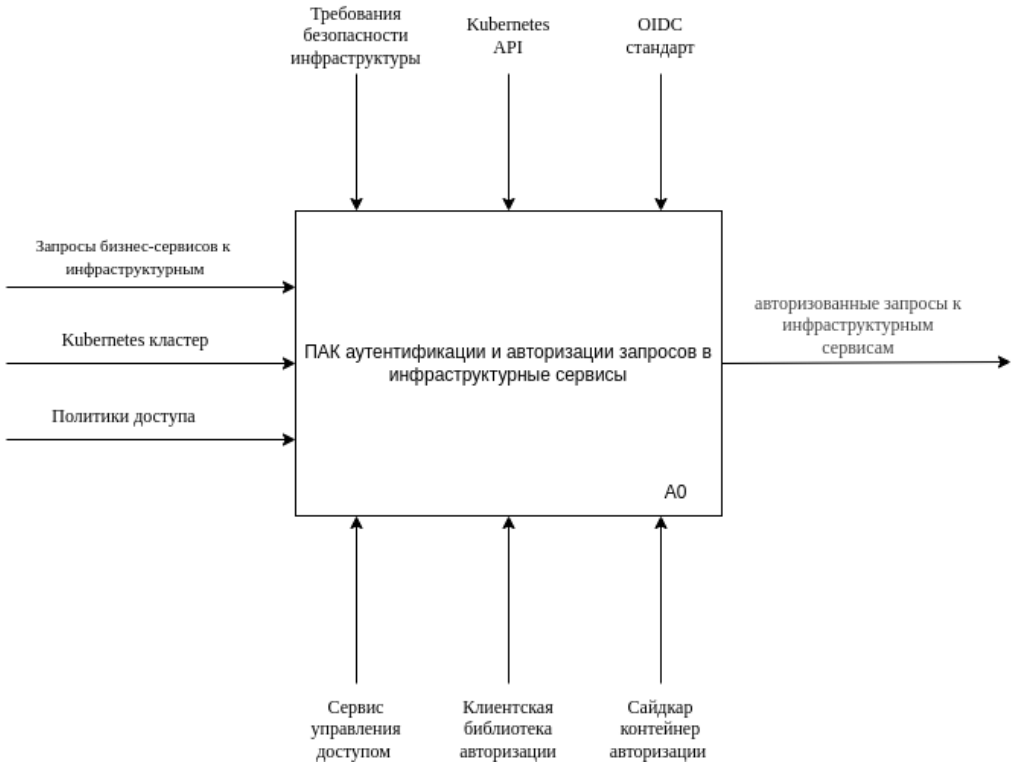

Рисунок 1.6 – Формализованная постановка задачи

Вывод

В данном разделе были рассмотрены основные понятия микросервисной архитектуре, инструменте и сущностях Kubernetes,паттерна Sidecar, рассмот-

рены существующие решения проблемы авторизации запросов сервисов, протоколы аутентификации, применимые к микросервисной архитектуры и которые могут использоваться для аутентификации инфраструктурных сервисов, а также формализована постановка задачи в виде IDEF-0 диаграммы нулевого уровня.

2 Конструкторский раздел

2.1 Метод авторизации HTTP запроса к инфраструктурному сервису

На рисунке 2.1 представлен формализованный в виде IDEF0 диаграммы метод авторизации HTTP запроса к инфраструктурному сервису

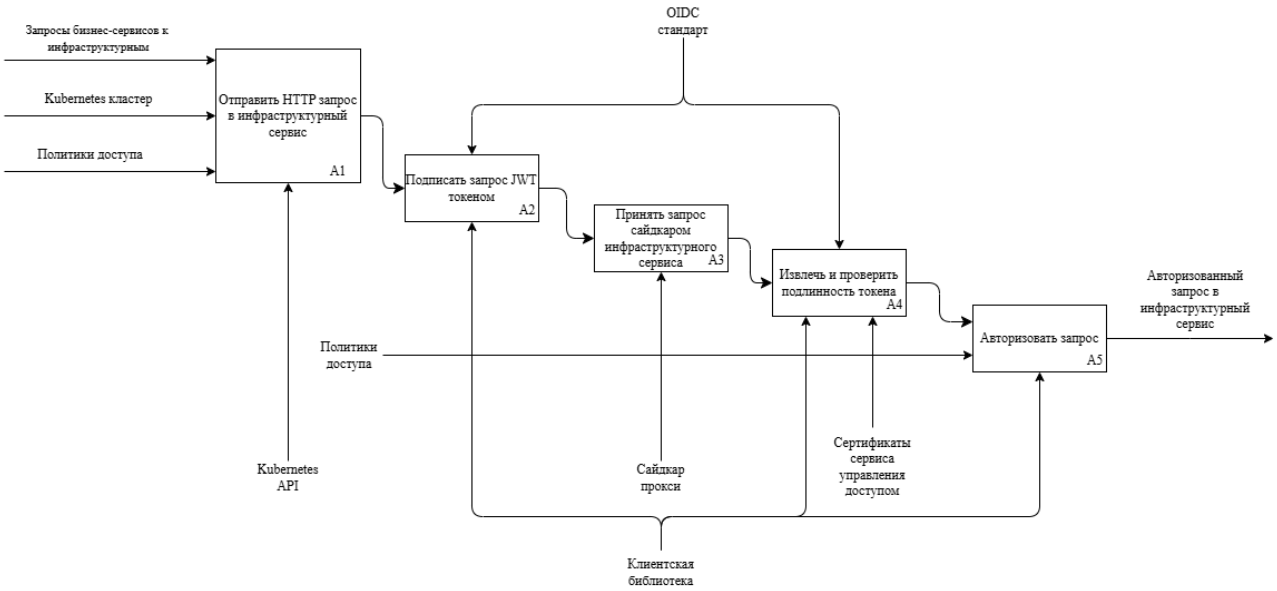

Рисунок 2.1 – Формализованный метод авторизации HTTP запроса к инфраструктурному сервису

2.2 Алгоритм аутентификации инфраструктурного сервиса

Для аутентификации сервиса-клиента, выполняющего подписанный запрос из сайдкар контейнера, необходим k8s токен пода, который хранится по пути с секретами. В обмен на этот k8s токен, IdP сервис выпускает токен аутентификации, который будет использоваться для подписи запроса от имени сервиса-клиента. На рисунке 2.2 представлен алгоритмы аутентификации сервиса-клиента.

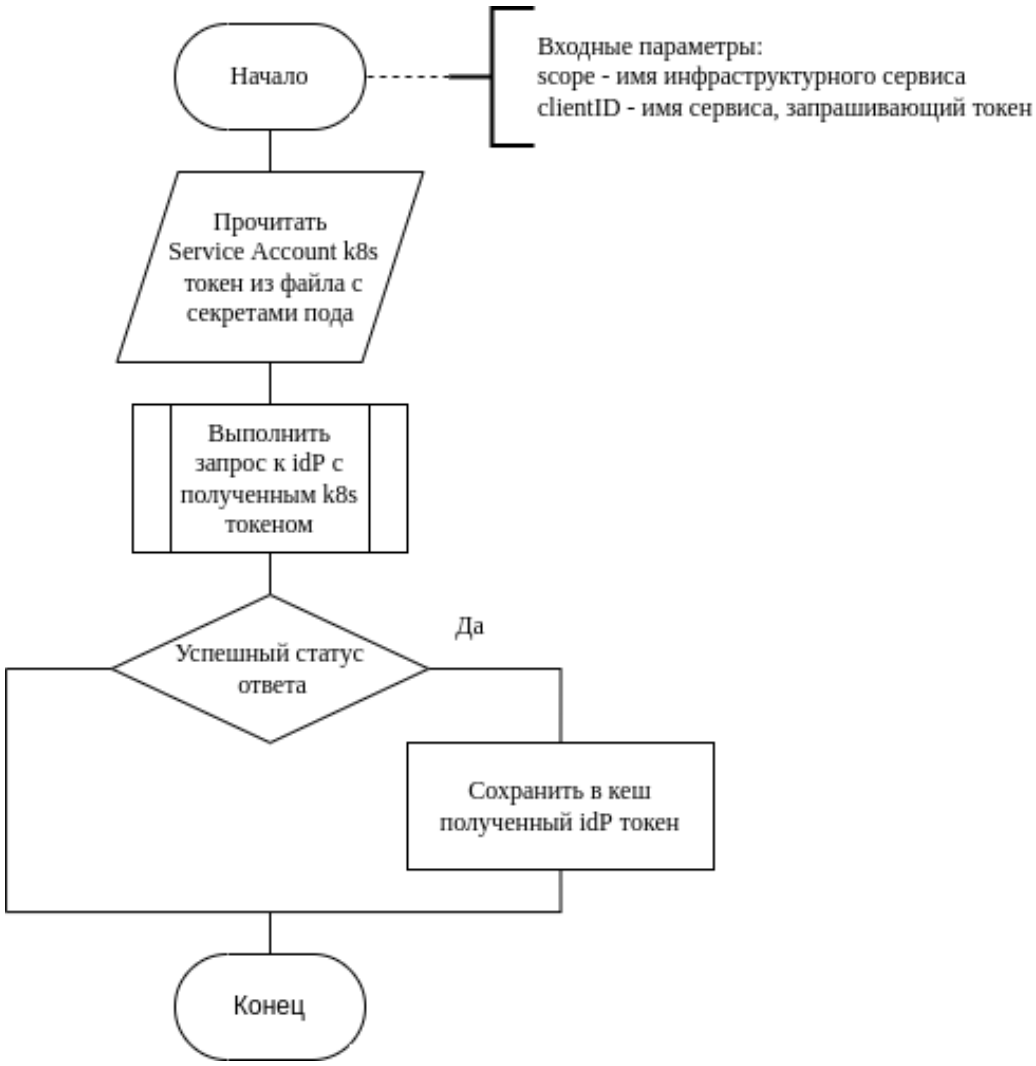

Рисунок 2.2 – Алгоритм аутентификации инфраструктурного сервиса

2.3 Алгоритм верификации k8s токена на стороне IdP

Для того, чтобы подтвердить подлинность k8s токена и личность, от имени которого он был выписан, IdP сервису нужен получить JWKs сертификаты от Kubernetes API, с помощью них расшифровать JWT токен и затем его проверить. Алгоритм верификации k8s токена представлен на рисунке 2.3.

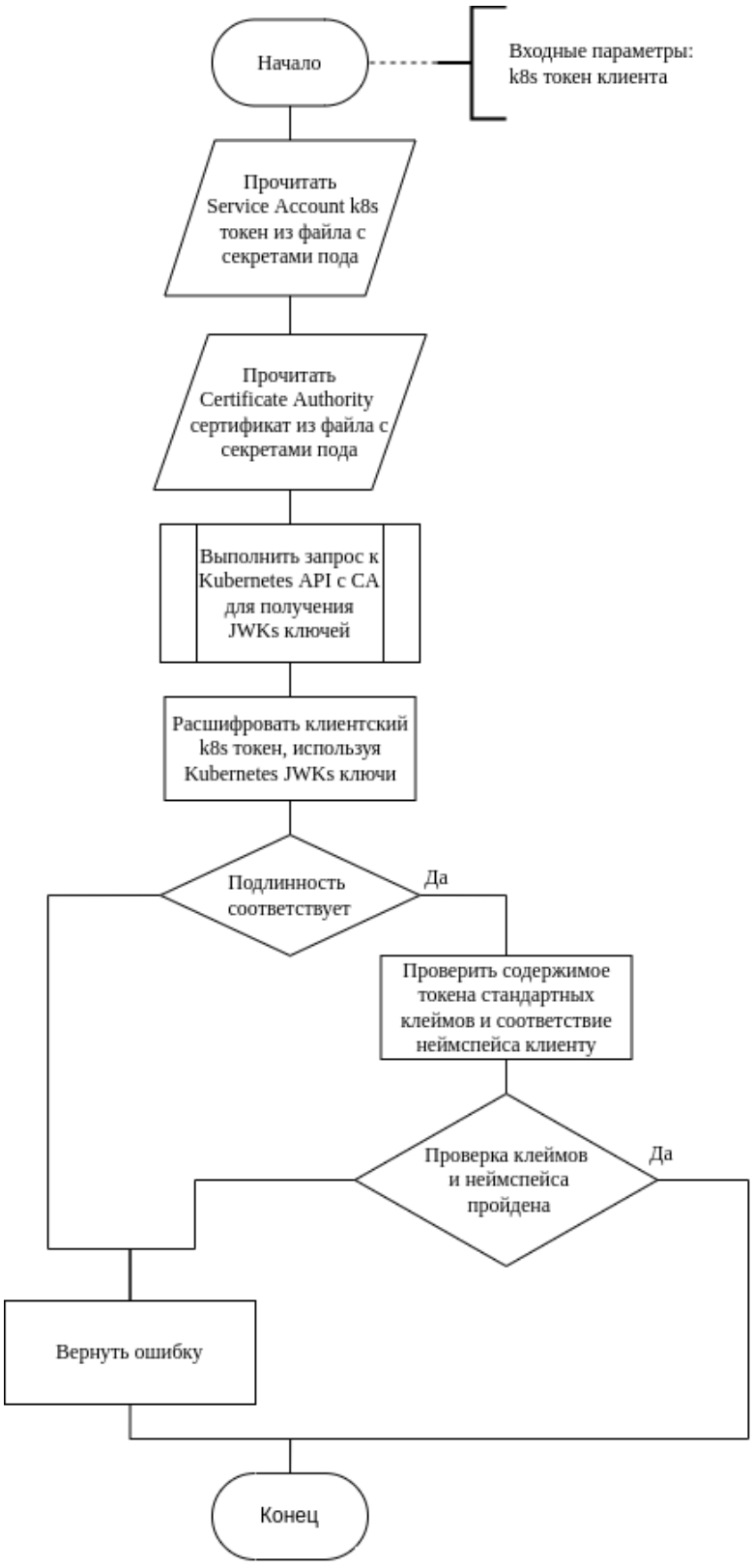

Рисунок 2.3 – Алгоритм верификации k8s токена

2.4 Алгоритм авторизации запроса на принимающей стороне

На принимающей стороне запроса необходимо сначала достать из заголовка сам токен, и проверить его подпись с помощью сертификатов IdP. Затем необходимо проанализировать сам запрос на предмет того, какие роли необходимы для его авторизации, и если эти роли содержатся в токене, только тогда можно пропустить запрос. Алгоритм авторизации представлен на рисунке 2.4.

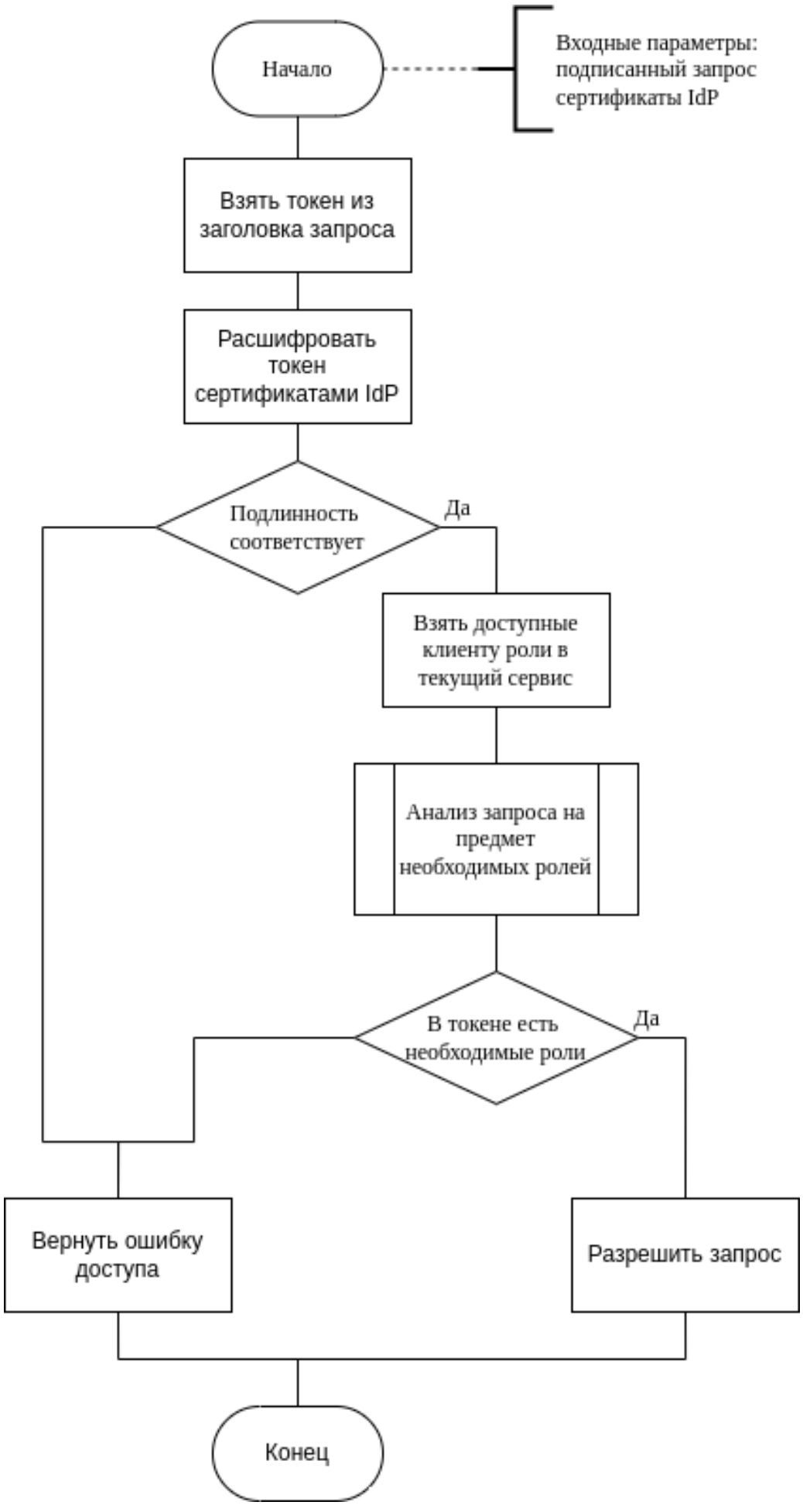

Рисунок 2.4 – Алгоритм авторизации запроса на принимающей стороне

Вывод

В данном разделе были спроектированы основные алгоритмы, необходимые для работы аутентификации.

3 Технологический раздел

Основные средства реализации:

1) k3d — утилита для поднятия k8s кластера локально, использует docker,
2) kubectl — утилита для ручного просмотра логов и состояния k8s кластера,
3) docker — инструмент для контейнеризации приложений. Используется в реализации для создания sidecar контейнеров,
4) ghcr.io — используется для загрузки docker образов в k8s кластер,
5) Golang — язык программирования, в основном использующийся для написания приложений в микросервисной архитектуре,
6) Prometheus — инструмент для сбора данных (метрик) сервисов,
7) Grafana — инструмент для визуализации запросов к метрикам, а также для составления дашбордов таких графиков.

3.1 Реализация IdP сервиса

IdP сервису необходимо реализовать REST API, чтобы в него могли ходить за выпуском токена. В соответствии с OIDC стандарт был реализован следующий контракт:

1) /realms/service2infra/.well-known/openid-configuration — обработчик запросов на получение OIDC конфигурации. Конфигурация должна содержать адреса и пути обработчиков запросов на выпуск токена и получения сертификатов, адрес issuer — где был выпущен токен, чтобы потом это проверить;
2) /realms/service2infra/protocol/openid-connect/token — обработчик запросов на выпуск и получение токена idP. Параметрами для запроса могут быть *"grant_type"* — способ проверки личности, в данном случае token-exchange, *"subject_token"* — сам токен для обмена, *"subject_token_type"* — тип токена для обмена, *"scope"* — в какую сущность выписывается токен;

3) /realms/service2infra/protocol/openid-connect/certs — обработчик запросов на получение сертификатов idP, возвращает публичный ключ сертификатов вместе с key-id.

В листинге 3.1 приведена реализация валидации k8s токена.

Листинг 3.1 – Валидация k8s токена

```
type Verifier struct {
    publicKey *rsa.PublicKey
}
func (v *Verifier) VerifyWithClient(k8sToken string) (string,
   jwt.Claims, error) {
    var claims privateClaims
    token, err := jwt.ParseWithClaims(k8sToken, &claims,
       func(token *jwt.Token) (interface{}, error) {
        if _, ok := token.Method.(*jwt.SigningMethodRSA); !ok {
            return nil, fmt.Errorf("unexpected method: %v",
               token.Header["alg"])
        }
        return v.publicKey, nil
    })
    if err != nil {
        return "", nil, fmt.Errorf("parsing jwt: %v", err)
    }

    if !token.Valid {
        return "", claims, fmt.Errorf("token cannot be converted
           to known one, which means it is invalid")
    }

    podName := claims.Kubernetes.Pod.Name
    namespace := claims.Kubernetes.Namespace

    if podName == "" || namespace == "" {
        return "", claims, fmt.Errorf("invalid k8s token claims
           (pod: %s, namespace: %s)", podName, namespace)
    } else if !strings.HasPrefix(podName+"-", namespace) {
        return "", claims, fmt.Errorf("pod name and namespace
           must both start with service name (pod: %s,
           namespace: %s)", podName, namespace)
    }

    return claims.Kubernetes.Namespace, claims, nil
}
```

В листинге 3.2 приведены клеймы, которые будут дальше зашифрованы

в токен.

Листинг 3.2 – Клеймы выпускаемых токенов

```
type Claims struct {
    Exp      time.Time `json:"exp"`
    Iat      time.Time `json:"iat"`
    Iss      string    `json:"iss"`
    Sub      string    `json:"sub"`
    Aud      string    `json:"aud"`
    Scope    string    `json:"scope"`
    Roles    []string  `json:"roles"`
    ClientID string    `json:"clientID"`
}
```

Сервис должен хранить свое состояние. В листинге 3.3 представлена таблица, в которой хранятся права.

Листинг 3.3 – Таблица для хранения прав

```
CREATE SCHEMA IF NOT EXISTS service2infra;

CREATE TABLE service2infra."Permissions" (
    ClientName TEXT NOT NULL,
    ServerName TEXT NOT NULL,
    roles TEXT[] NOT NULL
);
```

3.2 Реализация клиентской библиотеки

Чтобы постоянно не выпускать токен на каждый запрос, было принято решение делать это фоново. В листинге 3.4 показана функция, запрашивающая токен в фоне.

Листинг 3.4 – Фоновое обновление токенов

```
func (ts *TokenSource) runScheduler(ctx context.Context) {
    planner := time.NewTimer(0)
    defer planner.Stop()

    for {
        select {
        case <-ts.closeCh:
            return
        case <-ts.refreshCh:
        case <-planner.C:
        }

        delay := func() (delay time.Duration) {
            tokenResp, err := ts.issuer.IssueToken(ctx, ts.scope)
            if err != nil {
                log.Printf("failed to issue token to %s scope:
                   %v", ts.scope, err)
                return ts.cfg.ErrTokenBackoff
            }

            accessToken := tokenResp.AccessToken
            ts.token.Store(&accessToken)

            expiry := tokenResp.ExpiresIn
            newDelay := calcDelay(time.Until(expiry))
            log.Printf("New token to %s scope has been issued,
               expiry: %s, until_next: %s", ts.scope, expiry,
               newDelay)
            return newDelay
        }()

        resetTimer(planner, delay)
    }
}
```

В листинге 3.5 приведена реализация проверка подлинности токена.

Листинг 3.5 – Проверка подлинности токен

```
func verifyToken(rawToken string, certs *jose.JSONWebKeySet)
   (*tokenClaims, error) {
    token, err := jwt.ParseSigned(rawToken)
    if err != nil {
        log.Printf("failed to parse token: %v", err)
        return nil, fmt.Errorf("failed to parse token: %w", err)
    }

    var claims tokenClaims
    for _, header := range token.Headers {
        keys := certs.Key(header.KeyID)
        if len(keys) == 0 {
            continue
        }

        for _, key := range keys {
            if err := token.Claims(key.Public(), &claims); err
               == nil {
                return &claims, nil
            }
        }
    }

    log.Printf("no certificate found to parse token. certs: %v,
       tokenHeaders: %v", certs, token.Headers)
    return nil, fmt.Errorf("no certificate found to parse token")
}
```

3.3 Диаграмма компонентов разработанного ПО

На рисунке 3.1 приведена диаграмма компонентов разработанного ПО. Как можно видеть, реализация подписи и проверки токенов запросов между сервисами вынесена в отдельный компонент так, чтобы сервисы не были напрямую зависимы от сервиса IdP.

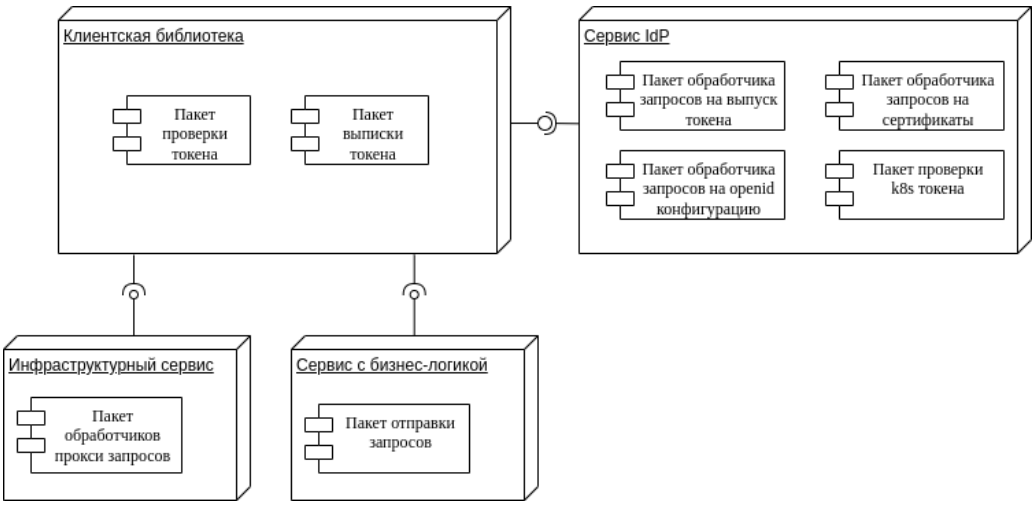

Рисунок 3.1 – Диаграмма компонентов разработанного ПО

3.4 Тестирование программного обеспечения

Для функционального тестирования были написаны unit-тесты. Пример unit теста, реализованного с использованием Arrange-Act-Assert паттерна, для проверки k8s токена приведен в листинге 3.6.

Листинг 3.6 – Тест для проверки k8s токена

```
func Test_Verify(t *testing.T) {
    // Arrange
    token := "ey..."
    jwk := JWK{
        N: `xItwc...`,
        E: "AQAB",
    }
    wantAud := jwt.ClaimStrings{
        "https://kubernetes.default.svc.cluster.local",
        "k3s",
    }
    publicKey, err := makeRSAPublicKey(jwk)
    if err != nil {
        t.Fatalf("failed to create public rsa key: %v", err)
    }
    verifier := &Verifier{
        publicKey: publicKey,
    }

    // Act
    gotClientID, gotClaims, gotErr :=
       verifier.VerifyWithClient(token)

    // Assert
    if gotErr != nil {
        t.Errorf("failed to verify token: %v", gotErr)
    } else if gotClientID != testClientID {
        t.Errorf("expected clientID: %s, got: %s", testClientID,
           gotClientID)
    }

    gotAud, gotAudErr := gotClaims.GetAudience()
    if gotAudErr != nil {
        t.Errorf("got unexpected aud err: %v", gotAudErr)
    }
    assert.EqualValues(t, wantAud, gotAud)
}
```

Всего было покрыто 77% кода функциональности ПО. На рисунке 3.2 представлено подробное покрытие.

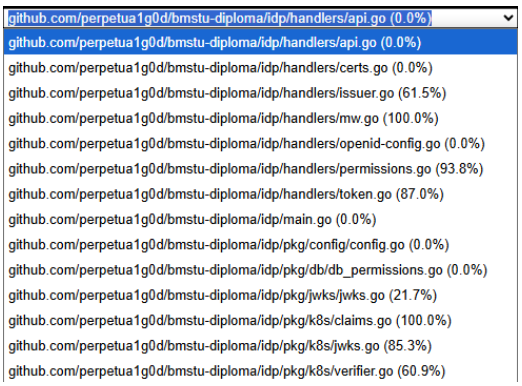

Рисунок 3.2 – Покрытие тестами функциональности

3.5 Интерфейс ПО

Для различного рода управления ПО был реализован интерфейс в виде административной панели. Пример интерфейса представлен на рисунке 3.3.

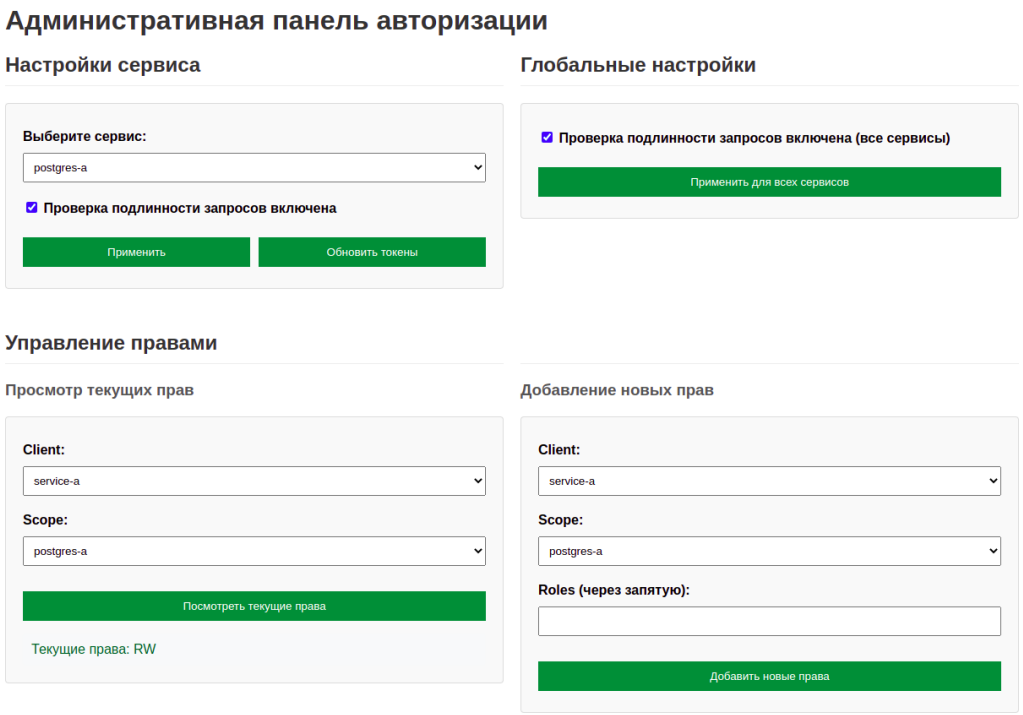

Рисунок 3.3 – Пример интерфейса разработанного ПО

Для наблюдения за состоянием системы была внедрен мониторинг ключевых показателей сервисов. Пример дашборда в Grafana с графиками, построенных по собранным данным, представлен на рисунке 3.4.

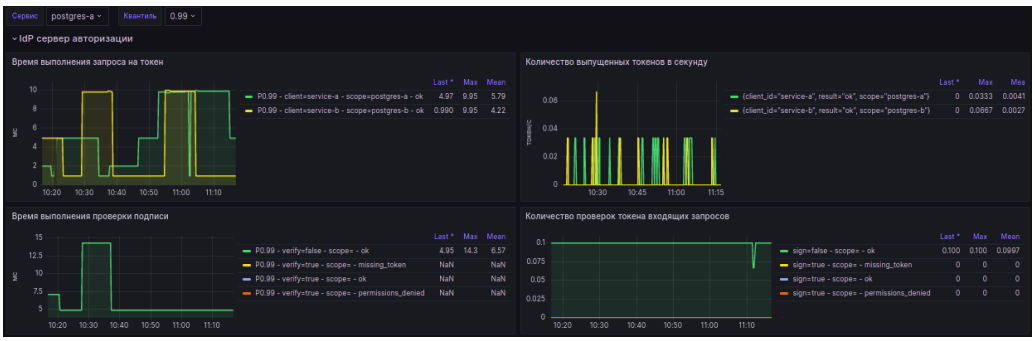

Рисунок 3.4 – Пример дашборда с графиками по собранным данным

Вывод

В данном разделе были описаны средства реализации программно-алгоритмического комплекса, приведена примеры реализации IdP сервиса и клиентской библиотеки, результаты тестирования, а также интерфейс ПО и системы мониторинга.

4 Исследовательский раздел

В исследовании будет проведено сравнение времени выполнения запроса с включенной и выключенной авторизацией от одного инфраструктурного сервиса к другому.

4.1 Описание проводимого исследования

На листингах 4.1–4.2 приведена реализация запроса к инфраструктурному сервису, а также запуск выполнения фиксированного количества запросов параллельно.

Листинг 4.1 – Реализация запроса к PostgreSQL сервису

```
func sendBenchmarkQuery(cfg *config.Config, authClient
   *auth_client.AuthClient) {
    target :=
       fmt.Sprintf("http://%s.%s.svc.cluster.local:8080%s",
        cfg.InitTarget,
        cfg.InitTarget,
        cfg.ServiceEndpoint,
    )

    reqBody, _ := json.Marshal(map[string]interface{}{
        "sql":    `INSERT INTO log (message) VALUES ($1)`,
        "params": []interface{}{fmt.Sprintf("Write from %s, ts:
           %s", cfg.Namespace, time.Now())},
    })

    req, err := http.NewRequest("POST", target,
       bytes.NewBuffer(reqBody))
    if err != nil {
        log.Fatalf("failed to create post request: %v", err)
        return
    }

    if cfg.SignAuthEnabled {
        token, err := authClient.Token(cfg.InitTarget)
        if err != nil {
            log.Fatalf("failed to issue token in auth client on
               scope %s: %v", cfg.InitTarget, err)
            return
        }
```

```
        req.Header.Set("X-I2I-Token", token)
    }
    req.Header.Set("Content-Type", "application/json")

    client := &http.Client{Timeout: 4 * time.Second}
    resp, err := client.Do(req)

    errMsg := handlers.RespErr{}
    var respBytes []byte
    if resp != nil && resp.Body != nil {
        respBytes, _ = io.ReadAll(resp.Body)
        _ = json.Unmarshal(respBytes, &errMsg)
    }

    if err != nil {
        log.Fatalf("Initial query failed: %v; errMsg: %s", err,
           errMsg.Error)
        return
    }
    defer resp.Body.Close()
}
```

Листинг 4.2 – Реализация запуска выполнения параллельных запросов

```
func runBenchmarks(cfg *config.Config, authClient
   *auth_client.AuthClient) {
    file, err := os.Create(benchmarksResultsFile)
    if err != nil {
        log.Fatalf("Cannot create results file: %v", err)
    }
    defer file.Close()
    writer := csv.NewWriter(file)
    defer writer.Flush()
    writer.Write([]string{"requests", "time_ms", "operation",
       "sign_enabled", "sign_disabled"})

    requestCount := []int64{100, 250, 500, 750, 1000}
    rerunCount := 10
    for _, reqCount := range requestCount {
        var avgTime float64 = 0
        for _ = range rerunCount {
            wg := &sync.WaitGroup{}
            wg.Add(int(reqCount))
```

```
            start := time.Now()
            for i := 0; i < int(reqCount); i++ {
                go func() {
                    defer wg.Done()
                    sendBenchmarkQuery(cfg, authClient)
                }()
            }
            wg.Wait()

            duration := time.Since(start).Milliseconds()
            avgTime += float64(duration)
        }

        avgTime = avgTime / float64(rerunCount*int(reqCount))
        log.Printf("finished %d requests, avg: %f", reqCount,
           avgTime)
        writer.Write([]string{
            strconv.FormatInt(reqCount, 10),
            strconv.FormatFloat(avgTime, 'f', 2, 64),
            "write",
            fmt.Sprintf("%v", cfg.SignAuthEnabled),
            fmt.Sprintf("%v", cfg.VerifyAuthEnabled),
        })
    }
}
```

4.2 Технические характеристики устройства

Технические характеристики устройства, на котором проводилось исследование:

1) процессор Intel(R) Core(TM) i5-10210U CPU @ 1.60GHz 2.11 GHz,

2) оперативная память 8 ГБ,

3) операционная система Ubuntu 21.0.

Исследование проводилось на ноутбуке. Во время исследования ноутбук не был нагружен посторонними приложениями, которые не относятся к исследованию, а также ноутбук был подключен к сети питания.

4.3 Полученные результаты

Исследование проводилось при включенной и выключенной авторизации 100, 250, 500, 750, 1000 параллельных запросов из одного инфраструктурного сервиса к другому. Результаты для каждого количества запросов были усреднены путем запуска 10 раз.

Графики полученных усредненных результатов представлены на рисунке 4.1.

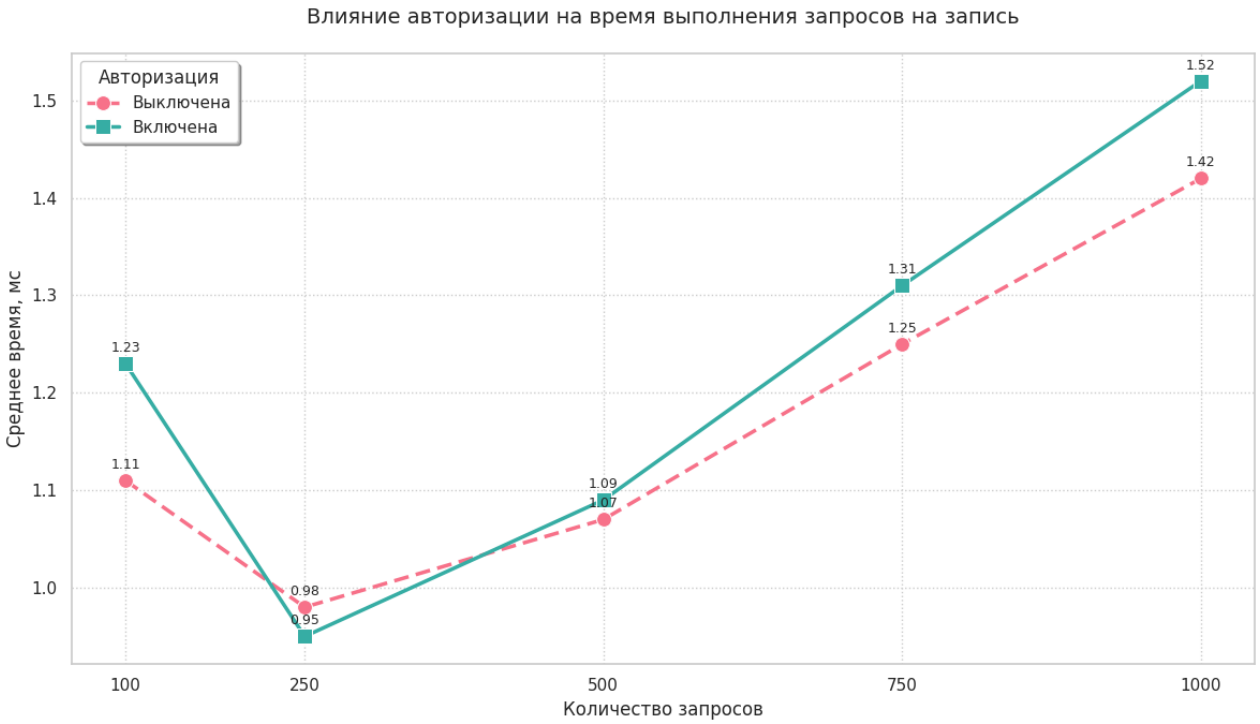

Рисунок 4.1 – Результаты исследования

Вывод

Как видно по графикам, время выполнения запросов с включенной авторизацией оказались в среднем на 10% дольше времени выполнения запросов с выключенной авторизацией. Это не окажет существенного влияния на работу системы из инфраструктурных сервисов.

ЗАКЛЮЧЕНИЕ

В рамках выпускной квалификационной работы был разработан программно-алгоритмический комплекс для авторизации запросов в инфраструктурные сервисы.

В ходе выполнения данной работы были решены следующие задачи:

1) провести обзор существующих подходов аутентификации и авторизации в микросервисной архитектуре;

2) рассмотреть основные протоколы аутентификации, применимые в микросервисной архитектуре;

3) разработать и описать ключевые алгоритмы работы программно-алгоритмического комплекса авторизации инфраструктурных микросервисов;

4) разработать программное обеспечение, реализующее аутентификацию и авторизацию инфраструктурных микросервисов;

5) провести исследование влияния работы авторизации на выполнения запросов между инфраструктурными сервисами.

Было проведено исследование времени выполнения запросов инфраструктурных сервисов как с включенной авторизацией, так и с выключенной. Исследование показало, что внедрение в систему инфраструктурных сервисов авторизации критического влияния на работу не оказало.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. *Кравченко Д. А.* Микросервисная архитектура // Интерактивная наука. — 2022. — № 4. — С. 69.
2. *Mateus-Coelho N.*, *Cruz-Cunha M.*, *Ferreira L. G.* Security in Microservices Architectures // Procedia Computer Science. — 2021. — Т. 181. — С. 1225—1236. — ISSN 1877-0509. — DOI: https://doi.org/10.1016/j.procs.2021.01.320.
3. *Epping M.*, *Morowczynski M.* Authentication and Authorization // IDPro Body of Knowledge. — 2021. — Сент. — Т. 1. — DOI: 10.55621/idpro.78.
4. *Shaikh S.*, *Mane S.* Authentic techniques of authentication in microservices // International Journal of Current Advanced Research. — 2017. — Апр. — Т. 6. — С. 3342—3345. — DOI: 10.24327/ijcar.2017.3345.0267.
5. Kubernetes Documentation [Электронный доступ]. — Режим доступа URL: https://kubernetes.io/docs/home/ (дата обращения: 05.06.2025).
6. *Catherine Meadows S. H.*, *Bloom G.* Sidecar-based Path-aware Security for Microservices // ACM Symposium on Access Control Models and Technologies. — 2023. — Июнь. — DOI: 10.1145/3589608.3594742.
7. Introduction to Service Mesh Technologies [Электронный доступ]. — Режим доступа URL: https://www.wallarm.com/cloud-native-products-101/istio-vs-linkerd-service-mesh-technologies (дата обращения: 05.06.2025).
8. *Barr A. B.*, *Lavi O.*, *Naor Y.* Technical Report: Performance Comparison of Service Mesh Frameworks: the MTLS Test Case. — 2024. — Нояб. — DOI: 10.48550/arXiv.2411.02267.
9. SPIFFE Overview [Электронный доступ]. — Режим доступа URL: https://spiffe.io/docs/latest/spiffe-about/overview/ (дата обращения: 05.06.2025).
10. SPIRE Concepts [Электронный доступ]. — Режим доступа URL: https://spiffe.io/docs/latest/spire-about/spire-concepts/ (дата обращения: 05.06.2025).

11. *Rahaman MS Tisha SN S. E.* Access Control Design Practice and Solutions in Cloud-Native Architecture: A Systematic Mapping Study // Sensors (Basel). — 2023. — Март. — DOI: 10.3390/s23073413.

12. The OAuth 2.0 Authorization Framework [Электронный доступ]. — Режим доступа URL: https://datatracker.ietf.org/doc/html/rfc6749 (дата обращения: 05.06.2025).

13. *Khedrane A.*, *Salmi H.*, *Abdelhamid B.* Securing Web APIs with OAuth 2.0 // Bernaoui Abdelhamid. — 2014. — Май.

14. *Li W.*, *Mitchell C.* User Access Privacy in OAuth 2.0 and OpenID Connect //. — 09.2020. — С. 664—6732. — DOI: 10.1109/EuroSPW51379.2020.00095.

15. *Ferdous M. S.* User-controlled Identity Management Systems using mobile devices : дис. ... канд. / Ferdous Md. Sadek. — 06.2015. — DOI: 10.13140/RG.2.1.3905.3287.

ПРИЛОЖЕНИЕ А

Презентация