

**Министерство науки и высшего образования Российской Федерации  
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ  
УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ  
НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО  
ITMO University**

**ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА  
GRADUATION THESIS**

**Программный модуль для мониторинга ресурсов и ошибок в информационных  
системах**

**Обучающийся / Student** Солопов Артём Сергеевич

**Факультет/институт/кластер/ Faculty/Institute/Cluster** факультет программной инженерии и компьютерной техники

**Группа/Group** P34222

**Направление подготовки/ Subject area** 09.03.04 Программная инженерия

**Образовательная программа / Educational program** Нейротехнологии и программирование 2021

**Язык реализации ОП / Language of the educational program** Русский

**Квалификация/ Degree level** Бакалавр

**Руководитель ВКР/ Thesis supervisor** Ткаченко Данил Михайлович, Университет ИТМО, факультет информационных технологий и программирования, преподаватель (квалификационная категория "преподаватель практики")

Обучающийся/Student

Документ подписан	
Солопов Артём Сергеевич	
22.05.2025	

(эл. подпись/ signature)

Солопов Артём  
Сергеевич

(Фамилия И.О./ name  
and surname)

Руководитель ВКР/  
Thesis supervisor

Документ подписан	
Ткаченко Данил Михайлович	
22.05.2025	

(эл. подпись/ signature)

Ткаченко Данил  
Михайлович

(Фамилия И.О./ name  
and surname)

**Министерство науки и высшего образования Российской Федерации  
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ  
УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ  
НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО  
ITMO University**

**ЗАДАНИЕ НА ВЫПУСКНУЮ КВАЛИФИКАЦИОННУЮ РАБОТУ /  
OBJECTIVES FOR A GRADUATION THESIS**

**Обучающийся / Student** Солопов Артём Сергеевич

**Факультет/институт/кластер/ Faculty/Institute/Cluster** факультет программной инженерии и компьютерной техники

**Группа/Group** P34222

**Направление подготовки/ Subject area** 09.03.04 Программная инженерия

**Образовательная программа / Educational program** Нейротехнологии и программирование 2021

**Язык реализации ОП / Language of the educational program** Русский

**Квалификация/ Degree level** Бакалавр

**Тема ВКР/ Thesis topic** Программный модуль для мониторинга ресурсов и ошибок в информационных системах

**Руководитель ВКР/ Thesis supervisor** Ткаченко Данил Михайлович, Университет ИТМО, факультет информационных технологий и программирования, преподаватель (квалификационная категория "преподаватель практики")

**Характеристика темы ВКР / Description of thesis subject (topic)**

**Тема в области фундаментальных исследований / Subject of fundamental research:** нет / not

**Тема в области прикладных исследований / Subject of applied research:** да / yes

**Основные вопросы, подлежащие разработке / Key issues to be analyzed**

**Цель:**

Сократить время реагирования на простои в информационных системах

**Техническое задание:**

Разработать архитектуру системы, включающую следующие компоненты:

- 1) API Gateway (на базе Nginx) для маршрутизации внешних запросов.
- 2) Auth Service для проверки JWT и работы с рефреш-токенами, с хранением данных в PostgreSQL и кэшированием в Redis.
- 3) Collector Service для получения логов от клиентских приложений и их публикации в брокер сообщений (Apache Kafka).
- 4) Rule Engine для проверки логов по заданным правилам (с использованием операторов eq, gt, repeat\_over и др.) и формирования уведомлений.
- 5) Alert-агенты (Telegram, Email, Discord) для отправки уведомлений и записи результатов в базу данных.
- 6) Реализовать REST-API сервис на языке Go, для работы клиентского веб-интерфейса

7) Реализовать клиентский веб-интерфейс, позволяющий пользователям получать Access-токен для SDK, создавать и редактировать правила, просматривать логи и управлять настройками системы.

Задачи:

- Выполнить обзор и анализ существующих решений для мониторинга логов, метрик и обработки ошибок в распределённых микросервисных системах .
- Исследовать архитектурные паттерны микросервисов и событийно ориентированных систем.
- Спроектировать и описать модель данных и API-контракты для приёма логов, управления правилами и отправки уведомлений.
- Разработать клиентский SDK для интеграции приложений с платформой Aletheia .
- Реализовать Auth Service на Go с поддержкой JWT и refresh токенов, хранением данных в PostgreSQL и Redis.
- Создать Collector Service на Go, принимающий HTTP запросы от SDK и публикующий события в Kafka-топики.
- Реализовать Rule Engine: парсинг и хранение правил, последовательную и вложенную проверку условий (операторы eq, contains, repeat\_over и др.) и публикацию задач уведомления.
- Разработать три Alert агента (Telegram, Email, Discord), читающих из Kafka и отправляющих уведомления, логируя результаты в базу данных.
- Настроить API Gateway на Nginx для маршрутизации, балансировки и аутентификации запросов.
- Спроектировать и реализовать веб интерфейс для управления токенами, правилами, просмотром логов и метрик.

Рекомендуемые материалы и пособия:

Сэм Ньюман. «Создание микросервисов». – М.: ДМК Пресс, 2016.

Нархид Н., Шапира Г. , Палино Т. «Apache Kafka. Поточковая обработка и анализ данных». – СПб.: Питер, 2020.

PostgreSQL. Профессиональный SQL : учеб. пособие / Е. П. Моргунов; под ред. Е. В. Рогова. — М.: ДМК Пресс, 2025. — 444 с.

И.В. Ананченко, Т.В. Зудилова, С.Е. Иванов КОНТЕЙНЕРИЗАТОР ПРИЛОЖЕНИЙ DOCKER — УСТАНОВКА, НАСТРОЙКА, ОСНОВ УПРАВЛЕНИЯ ПРИЛОЖЕНИЯМИ В СРЕДАХ С ПОДДЕРЖКОЙ КОНТЕЙНЕРИЗАЦИИ

Redis in Action - Josiah L. Carlson

Администрирование сервера NGINX - Димитрий Айвалиотис

Building Event-Driven Microservices - Адам Бельмар

**Форма представления материалов ВКР / Format(s) of thesis materials:**

Текст ВКР , приложение с программным кодом и методическими рекомендациями.

Дата выдачи задания / Assignment issued on: 06.02.2025

Срок представления готовой ВКР / Deadline for final edition of the thesis 25.05.2025

**СОГЛАСОВАНО / AGREED:**

Руководитель ВКР/  
Thesis supervisor

Документ подписан	
Ткаченко Данил Михайлович	
23.04.2025	

(эл. подпись)

Ткаченко Данил  
Михайлович

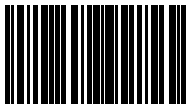
Задание принял к  
исполнению/ Objectives  
assumed BY

Документ подписан	
Солопов Артём Сергеевич	
27.04.2025	

(эл. подпись)

Солопов Артём  
Сергеевич

Руководитель ОП/ Head  
of educational program

Документ подписан	
Лисицына Любовь Сергеевна	
29.04.2025	

(эл. подпись)

Лисицына  
Любовь  
Сергеевна

**Министерство науки и высшего образования Российской Федерации  
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ  
УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ  
НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО  
ITMO University**

**АННОТАЦИЯ  
ВЫПУСКНОЙ КВАЛИФИКАЦИОННОЙ РАБОТЫ  
SUMMARY OF A GRADUATION THESIS**

**Обучающийся / Student** Солопов Артём Сергеевич

**Факультет/институт/кластер/ Faculty/Institute/Cluster** факультет программной инженерии и компьютерной техники

**Группа/Group** P34222

**Направление подготовки/ Subject area** 09.03.04 Программная инженерия

**Образовательная программа / Educational program** Нейротехнологии и программирование 2021

**Язык реализации ОП / Language of the educational program** Русский

**Квалификация/ Degree level** Бакалавр

**Тема ВКР/ Thesis topic** Программный модуль для мониторинга ресурсов и ошибок в информационных системах

**Руководитель ВКР/ Thesis supervisor** Ткаченко Данил Михайлович, Университет ИТМО, факультет информационных технологий и программирования, преподаватель (квалификационная категория "преподаватель практики")

**ХАРАКТЕРИСТИКА ВЫПУСКНОЙ КВАЛИФИКАЦИОННОЙ РАБОТЫ  
DESCRIPTION OF THE GRADUATION THESIS**

**Цель исследования / Research goal**

Создание программного модуля, который сократит время реагирования на простои в информационных системах

**Задачи, решаемые в ВКР / Research tasks**

1) Выполнить обзор и анализ существующих решений для мониторинга логов, метрик и обработки ошибок в распределённых микросервисных системах. 2) Исследовать архитектурные паттерны микросервисов и событийно ориентированных систем. 3) Спроектировать и описать модель данных и API-контракты для приёма логов, управления правилами и отправки уведомлений. 4) Разработать клиентский SDK для интеграции приложений с платформой Aletheia. 5) Реализовать Auth Service на Go с поддержкой JWT и refresh токенов, хранением данных в PostgreSQL и Redis. 6) Создать Collector Service на Go, принимающий HTTP запросы от SDK и публикующий события в Kafka-топики. 7) Реализовать Rule Engine: парсинг и хранение правил, последовательную и вложенную проверку условий (операторы eq, contains, repeat\_over и др.) и публикацию задач уведомления. 8) Разработать три Alert агента (Telegram, Email, Discord), читающих из Kafka и отправляющих уведомления, логируя результаты в базу данных. 9) Настроить API Gateway на Nginx для маршрутизации, балансировки и аутентификации запросов. 10) Спроектировать и реализовать веб интерфейс для управления токенами, правилами, просмотром логов и метрик 11) Опубликовать решение на GitHub репозитории

## Краткая характеристика полученных результатов / Short summary of results/findings

В ходе работы выполнен полный цикл разработки и валидации платформы Aletheia - rule-based решения для централизованного сбора логов, мониторинга метрик и мгновенной доставки оповещений в микросервисных системах. Проведён сравнительный анализ пяти популярных инструментов (Sentry, Datadog, Prometheus + Grafana, Elastic Stack, PagerDuty), выявивший нишу для open-source-решения с гибкими правилами алертинга и поддержкой как ошибок, так и инфраструктурных метрик. Спроектирована событийно-ориентированная архитектура на базе Kafka и набора микросервисов (Auth Service, Collector Service, Rule Engine, три Alert-агента), а также реализованы клиентский SDK и веб-интерфейс управления. Функциональные модули написаны на Go. Безопасность обеспечивается короткоживущими JWT и refresh-токенами (PostgreSQL + Redis). В нагрузочных испытаниях стенд обрабатывает около 800 событий/с при медианной задержке около 0,1 с от приёма события до отправки алерта, а lag потребителей Kafka остаётся < 30 сообщений. Репозиторий размещён на GitHub. Тем самым доказана практическая пригодность Aletheia для снижения времени реакции инженеров на простои.

Обучающийся/Student

Документ подписан	
Солопов Артём Сергеевич	
22.05.2025	

(эл. подпись/ signature)

Солопов Артём  
Сергеевич

(Фамилия И.О./ name  
and surname)

Руководитель ВКР/  
Thesis supervisor

Документ подписан	
Ткаченко Данил Михайлович	
22.05.2025	

(эл. подпись/ signature)

Ткаченко Данил  
Михайлович

(Фамилия И.О./ name  
and surname)

## СОДЕРЖАНИЕ

<b>СПИСОК СОКРАЩЕНИЙ И УСЛОВНЫХ ОБОЗНАЧЕНИЙ</b>	<b>8</b>
<b>ТЕРМИНЫ И ОПРЕДЕЛЕНИЯ</b>	<b>10</b>
<b>ВВЕДЕНИЕ</b>	<b>11</b>
<b>1 ТЕОРЕТИЧЕСКИЙ ОБЗОР</b>	<b>12</b>
1.1 Актуальность разработки	12
1.2 Обзор аналогов	12
1.2.1 Sentry: Специализированная платформа для отслеживания ошибок	12
1.2.2 Datadog: Полнофункциональная облачная платформа наблюдения	13
1.2.3 Prometheus + Grafana: Открытое решение для мониторинга	14
1.2.4 Elastic Stack (ELK): Решение для анализа логов	15
1.2.5 PagerDuty: Платформа управления инцидентами	16
1.2.6 Сравнительная таблица характеристик	17
1.2.7 Выводы по результатам анализа	18
1.3 Обзор технологий реализации	19
1.3.1 Nginx	19
1.3.3 Kafka (Apache Kafka)	20
1.3.4 TimescaleDB	21
1.3.5 Redis	21
1.3.6 PostgreSQL	22
1.3.7 Go (Golang)	22
1.3.8 React	23
<b>2 ПРОЕКТИРОВАНИЕ</b>	<b>31</b>
2.1 Архитектурная схема системы	31
2.2 Диаграмма бизнес-процесса(BPMN)	34
2.3 Диаграммы вариантов использования	36
<b>3 РЕАЛИЗАЦИЯ</b>	<b>38</b>
3.1 Разработка	38
3.1.1 Collector Service	38
3.1.2 Rule Engine	40
3.1.3 Alert-агенты	43
3.1.4 Auth Service	45
3.1.5 UI	48
3.2 Тестирование	51
3.3 Экспериментальное доказательство цели работы	52
3.4 Вывод	54
<b>ЗАКЛЮЧЕНИЕ</b>	<b>56</b>
<b>СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ</b>	<b>58</b>
<b>ПРИЛОЖЕНИЕ</b>	<b>60</b>

## СПИСОК СОКРАЩЕНИЙ И УСЛОВНЫХ ОБОЗНАЧЕНИЙ

**SDK (Software Development Kit)** - комплект инструментов и библиотек для разработки приложений

**API (Application Programming Interface)** - программный интерфейс приложения, обеспечивающий взаимодействие между компонентами ПО

**UI (User Interface)** - пользовательский интерфейс

**CRUD (Create, Read, Update, Delete)** - базовые операции над данными в информационных системах

**MQ (Message Queue)** - система очередей сообщений, используемая для обмена данными между процессами или сервисами

**Kafka (Apache Kafka)** - система распределённых очередей сообщений и потоковой обработки данных

**DevOps** - набор методологий и инструментов, направленных на взаимодействие команд разработки (Dev) и эксплуатации (Ops)

**CI/CD (Continuous Integration / Continuous Delivery)** - практика непрерывной интеграции и доставки программных продуктов

**REST (Representational State Transfer)** - стиль архитектуры веб-сервисов, использующий HTTP-протокол

**MongoDB** - документо-ориентированная NoSQL база данных

**TimescaleDB** - расширение для PostgreSQL, оптимизированное для хранения временных рядов

**CLI (Command Line Interface)** - интерфейс командной строки

**JWT (JSON Web Token)** - способ передачи подтверждённой информации о пользователе в виде компактного токена

**BPMN (Business Process Model and Notation)** - нотация для графического описания бизнес-процессов

**Docker** - платформа контейнеризации приложений

**Kubernetes (K8s)** - система оркестрации контейнеров



**Nginx** - веб-сервер и реверс-прокси, используемый для маршрутизации входящих запросов

**Redis** - высокопроизводительная система кэширования данных в оперативной памяти (in-memory key-value store)

**PostgreSQL** - реляционная СУБД, обеспечивающая хранение и управление структурированными данными

## ТЕРМИНЫ И ОПРЕДЕЛЕНИЯ

**API Gateway** - компонент системы, реализованный с помощью Nginx, который принимает входящие запросы, осуществляет аутентификацию (через Auth Service) и маршрутизирует их к соответствующим микросервисам.

**Collector Service** - микросервис, обрабатывающий входящие логи/события от клиентских приложений и распределяющий их в очереди сообщений (Kafka).

**Rule Engine** - программный модуль, в котором описываются правила обработки событий (условия, операторы AND/OR, временные ограничения и т. д.), определяющие логику реакции системы.

**Alert-агент** - сервис, отвечающий за доставку уведомлений в различные каналы (Email, Telegram, Discord и др.) по сигналу от Rule Engine.

**Refresh-токен (Refresh-token)** - компонент механизма аутентификации, позволяющий безопасно продлевать короткоживущий access-токен без повторного ввода учётных данных.

**Access-токен (Access-token)** - короткоживущий токен, передаваемый при запросах к защищённым ресурсам; содержит базовую информацию о пользователе и время жизни.

**X-User-Id** - дополнительный заголовок, добавляемый Auth Service/Nginx при валидации JWT и передаваемый в Collector Service для определения владельца логов.

**Клиентское SDK** - набор инструментов и библиотек, предназначенных для интеграции внешних приложений с платформой Aletheia, позволяющий отправлять логи и настраивать уведомления через Public API.

## ВВЕДЕНИЕ

В микросервисных системах, особенно при их интенсивном росте и усложнении, мониторинг состояния приложения, сбор логов и мгновенная реакция на сбои становятся критически важными. Своевременное оповещение ответственных лиц о неполадках позволяет быстро устранять ошибки и сохранять высокий уровень доступности сервисов.

Настоящая выпускная квалификационная работа посвящена разработке платформы **Aletheia**, решающей задачу комплексного сбора логов, гибкой настройки правил и автоматической рассылки уведомлений. В ходе работы будут описаны ключевые компоненты Aletheia (Клиентское SDK, Collector Service, Rule Engine, Alert-агенты), а также механизм авторизации и маршрутизации, включающий Auth Service, работающий в связке с PostgreSQL и Redis, и реверс-прокси Nginx.

**Цель** данной работы - создать систему, сокращающую время простоя и упрощающую интеграцию мониторинга и автоматических оповещений в распределенных приложениях. Для достижения этой цели необходимо:

- 1) Проанализировать основные технологии сбора логов, мониторинга и аутентификации, применяемые в современных микро серверных архитектурах;
- 2) Написать клиентские SDK для разных языков программирования;
- 3) Рассмотреть инфраструктурные компоненты (Kafka, MongoDB, TimescaleDB, Redis, PostgreSQL) и подход к маршрутизации запросов (Nginx);
- 4) Описать авторизационный модуль (Auth Service), использующий паттерн короткоживущих access-токенов и рефреш-токенов;
- 5) Разработать модель взаимодействия Rule Engine и Alert-агентов для реагирования на критические события;
- 6) Провести тестирование и оценить эффективность разработанного решения.

# 1 ТЕОРЕТИЧЕСКИЙ ОБЗОР

## 1.1 Актуальность разработки

С переходом на микросервисную архитектуру растёт сложность взаимодействия отдельных модулей, а также повышаются требования к устойчивости системы. Если в монолитном приложении достаточно отследить один процесс, то в микросервисном окружении приходится контролировать множество сервисов, развернутых в разных контейнерах или даже на разных серверах.

Ключевой проблемой становится своевременное обнаружение сбоев и избыточных нагрузок. Традиционные методы ручной проверки логов уже не справляются с объёмами данных. Кроме того, в случае возникновения критической ошибки важно мгновенно уведомить ответственных разработчиков или операторов. Платформа **Aletheia** отвечает на эти вызовы, предлагая комплексный подход к сбору логов, их обработке (Rule Engine) и рассылке уведомлений через различные каналы.

Дополнительную актуальность придаёт вопрос безопасности и контроля доступа: прежде чем логи будут попадать в центральную систему, необходима надёжная аутентификация и авторизация. В Aletheia реализован Auth Service, использующий паттерн рефреш-токена. Благодаря этому можно безопасно управлять пользовательскими сессиями и исключать ситуации, когда скомпрометированный токен даёт неограниченный доступ к сервисам.

## 1.2 Обзор аналогов

### 1.2.1 Sentry: Специализированная платформа для отслеживания ошибок

#### 1.2.1.1 Архитектура и основные функции

Sentry использует централизованную архитектуру с моделью предоставления услуги по модели SaaS (Software-as-a-Service). Интеграция с приложениями осуществляется через REST API и официальные SDK (более 30 поддерживаемых языков). Платформа автоматически группирует ошибки

на основе стека вызовов, анализирует их частоту и формирует детальные диагностические отчёты, упрощая локализацию проблем.

#### 1.2.1.2. Преимущества

- 1) **Глубокая интеграция с CI/CD:** при возникновении критических ошибок система может автоматически создавать задачи (issues) в Jira или GitHub;
- 2) **Source Maps-анализ:** деобфускация минифицированного JavaScript-кода, что облегчает отладку фронтенд-приложений;
- 3) **Performance Monitoring:** отслеживание времени отклика API-эндпоинтов и метрик производительности для определения “узких мест” в приложении.

#### 1.2.1.3. Недостатки

- 1) **Ограниченный мониторинг ресурсов:** платформа ориентирована на ошибки, метрики CPU/RAM и их аналитику поддерживаются в минимальном объёме;
- 2) **Жёсткая модель тарификации:** в бесплатном тарифном плане ограничено число обрабатываемых событий;
- 3) **Слабая кастомизация алертов:** базовые правила триггеров, без возможности тонкой логики или комбинирования сложных условий;
- 4) **Пример конфигурации алерта в Sentry через SDK**

```
sentry_sdk.init(  
    dsn="https://examplePublicKey@o0.ingest.sentry.io/0",  
    traces_sample_rate=1.0,  
    integrations=[RedisIntegration()  
]).
```

### 1.2.2 Datadog: Полнофункциональная облачная платформа наблюдения

#### 1.2.2.1 Архитектурный подход и основные функции

Datadog собирает метрики, логи и трассировки, сводя их в единый дашборд. Система работает через агентов, устанавливаемых на сервера и

контейнеры, а также предлагает более 600 готовых интеграций с популярными сервисами (базы данных, брокеры сообщений и т.д.). Все данные анализируются в облачной платформе Datadog.

#### 1.2.2.2 Преимущества

- 1) **Unified Observability**: корреляция данных APM (Application Performance Monitoring), инфраструктурных метрик и логов для единой “панели здоровья” системы;
- 2) **AI-Driven Anomaly Detection**: прогнозирование аномалий на основе алгоритмов машинного обучения;
- 3) **Синтетический мониторинг**: проверка доступности сервисов из различных географических локаций (более 20 точек).

#### 1.2.2.3 Недостатки

- 1) **Высокий порог входа**: требуется время на изучение инструментов Datadog и их правильную конфигурацию;
- 2) **Отсутствие on-premise-решения**: система доступна только по SaaS-модели, что может быть неприемлемо для ряда организаций;
- 3) **Стоимость масштабирования**: при увеличении количества метрик или хостов стоимость возрастает экспоненциально.

### 1.2.3 Prometheus + Grafana: Открытое решение для мониторинга

#### 1.2.3.1 Архитектурный подход

Prometheus реализует “pull-модель” сбора метрик, при которой он периодически запрашивает (pull) данные у экспортёров (exporters). Собранные показатели хранятся во встроенной базе данных TSDB. Grafana используется как надстройка для визуализации метрик и создания дашбордов, а также имеет базовый функционал алертинга.

#### 1.2.3.2 Сильные стороны

- 1) **Гибкость конфигурации**: имеется множество стандартных exporters для популярных систем (базы данных, веб-серверы и т.д.), а также возможность создания собственных;

- 2) **Многооблачная поддержка:** Prometheus можно настроить для сбора метрик в гибридных инфраструктурах;
- 3) **Активное сообщество:** большое число разработчиков и пользователей обеспечивает регулярные обновления и плагины.

#### 1.2.3.3 Слабые места

- 1) **Отсутствие обработки ошибок:** система ориентирована на метрики, ошибки и их стеки не анализируются в глубину;
- 2) **Сложность настройки сложных правил:** требуется ручное прописывание алертов на языке выражений PromQL;
- 3) **Проблемы масштабирования:** типовая инсталляция Prometheus — это одиночный узел (single node), что усложняет сохранность данных и распределённый сбор при больших объёмах.

### 1.2.4 Elastic Stack (ELK): Решение для анализа логов

#### 1.2.4.1. Функциональные характеристики

Elastic Stack объединяет следующие компоненты:

- 1) **Elasticsearch** - распределённая поисковая и аналитическая система, ориентированная на полнотекстовый поиск и быструю индексацию данных.
- 2) **Logstash** - инструмент для сбора, фильтрации и преобразования логов;
- 3) **Kibana** - визуализационная надстройка, позволяющая строить графики и дашборды, а также выполнять поисковые запросы к данным в Elasticsearch.

#### 1.2.4.2 Преимущества

- 1) **Полнотекстовый поиск:** мощные возможности поиска по неструктурированным логам (match, wildcard, фразовый поиск и др.);
- 2) **Горизонтальное масштабирование:** Elasticsearch можно разворачивать в виде кластера, распределяя индексы по нескольким узлам;
- 3) **Data Enrichment:** при помощи фильтров Logstash можно добавлять к логам геоданные, метки временной зоны и т.д.

### 1.2.4.3 Недостатки

- 1) **Слабая система алертинга:** встроенный модуль Watcher относительно прост, и для более сложных сценариев требуется дополнительная разработка или внешние сервисы;
- 2) **Высокие аппаратные требования:** Elasticsearch требует достаточного объёма оперативной памяти и дисковой подсистемы для хранения индексов;
- 3) **Нет встроенного мониторинга метрик:** фокус сделан на логах, сбор метрик CPU/RAM придётся реализовывать интеграциями с другими системами (например, Metricbeat).

## 1.2.5. PagerDuty: Платформа управления инцидентами

### 1.2.5.1. Специфика реализации

**PagerDuty** специализируется на оркестрации инцидентов: интегрируется с системами мониторинга и логирования (в том числе с Datadog, Prometheus или ELK), получая от них уведомления о сбоях. Далее платформа формирует эскалационные цепочки, перенаправляет инциденты ответственным специалистам и ведёт учёт действия команды.

### 1.2.5.2. Сильные стороны

- 1) **Автоматическая эскалация:** каскадное оповещение команды по заранее заданному расписанию или ротации дежурных;
- 2) **Postmortem-аналитика:** ведётся журнал RCA (Root Cause Analysis), формируются отчёты об инцидентах, позволяя выявлять системные проблемы;
- 3) **Интеграция с ITSM:** двусторонняя синхронизация задач с ServiceNow или Jira, что упрощает совместную работу с сервис-деском.

### 1.2.5.3. Ограничения

- 1) **Нет собственного мониторинга:** платформа получает события извне, не собирая логи или метрики самостоятельно;
- 2) **Узкая специализация:** основное назначение — маршрутизация оповещений и последующая аналитика работы команды;



- 3) **Ценовая политика:** стоимость лицензии может стать чрезмерной для малых команд.

#### **1.2.6 Сравнительная таблица характеристик**

Таблица 1 - Сравнительная таблица сервисов

Параметр	Aletheia	Sentry	Datadog	Prometheus + Grafana	Elastic Stack	PagerDuty
Тип решения	Open-Source	SaaS	SaaS	Open-Source	Open-Source	SaaS
Мониторинг ошибок	Да	Да	Ограничено	Нет	Да	Нет
Метрики ресурсов	Да	Нет	Да	Да	Нет	Нет
Каналы уведомлений	Telegram, Discord, Email	Email, Slack (10+ интеграций)	Webhook	Webhook	Базовые интеграции	15+ интеграций
Правила алертинга	Гибкие (rule-based)	Шаблоны	AI, расширенное	Статическое (PromQL)	Watcher (ограничено)	Каскадная эскалация (посредники)
Управление инфраструктурой	CLI-утилита aletheia-paas	Отсутствует	Отсутствует	Ручное	Ручное	Отсутствует
Локализация данных	On-Premise	Облако	Облако	On-Premise	On-Premise	Облако
Стоимость	Бесплатно (OSS)	От \$26/мес	От \$15/хост	Бесплатно	Бесплатно	От \$21/пользователь

### 1.2.7 Выводы по результатам анализа

- 1) **Sentry** — ориентирован в первую очередь на обработку ошибок, отличаясь высокой степенью детализации стектрейсов и интеграцией с CI/CD. Однако мониторинг системных метрик в базовом функционале ограничен;
- 2) **Datadog** — облачная платформа, предоставляющая единое окно для логов, метрик и трассировок. Расширенные возможности аномалий (AI) и синтетический мониторинг делают её универсальной, но стоимость может быстро расти при масштабировании;
- 3) **Prometheus + Grafana** — классическое open-source решение для метрик, с возможностью базового алертинга и построения дашбордов. Не подходит для детального анализа ошибок;
- 4) **Elastic Stack** сфокусирован на сборе и анализе логов с мощным поиском и горизонтальной масштабируемостью, но имеет слабую встроенную систему алертинга и не покрывает метрики “из коробки”;
- 5) **PagerDuty** решает задачи оркестрации инцидентов и эскалации, но не обладает собственными средствами сбора метрик или логов, полностью полагаясь на интеграции со сторонними системами.

**Aletheia** выделяется среди рассмотренных решений сочетанием мониторинга ошибок, метрик и гибких сценариев алертинга в рамках единой open-source экосистемы.

### 1.3 Обзор технологий реализации

#### 1.3.1 Nginx

##### Описание

Nginx — высокопроизводительный веб-сервер и реверс-прокси, широко применяемый для маршрутизации HTTP-трафика и балансировки нагрузки.

##### Ключевые функции

- 1) Маршрутизация запросов: на основе URI и HTTP-заголовков перенаправляет входящие соединения на нужные бэкенд-сервисы.
- 2) Балансировка нагрузки: распределяет трафик между несколькими экземплярами приложений, повышая отказоустойчивость.

- 3) Кэширование: хранит ответы на часто запрашиваемые ресурсы, снижая нагрузку на сервера.
- 4) TLS/SSL-терминация: обеспечивает безопасное соединение HTTPS, разгружая бэкенды от шифрования.

### **Преимущества**

- 1) Обработка десятков тысяч одновременных соединений с минимальными затратами ресурсов.
- 2) Гибкая конфигурация через декларативные файлы, поддержка динамического перезагрузки настроек.
- 3) Широкая экосистема модулей (rate-limiting, gzip, geoIP и др.).

### **1.3.3 Kafka (Apache Kafka)**

#### **Описание**

Apache Kafka — распределённая платформа обмена сообщениями и потоковой обработки данных, ориентированная на публикацию-подписку (pub/sub) и обработку журналов событий (“commit log”).

#### **Ключевые функции**

- 1) Топики и партиционирование: обеспечивает горизонтальное масштабирование и упорядоченную доставку сообщений.
- 2) Consumer Group: позволяет множеству потребителей совместно обрабатывать поток, распределяя нагрузку.
- 3) Хранение сообщений: отказоустойчивое хранение в журнале, возможен произвольный ретроспективный “реплей”.
- 4) Высокая пропускная способность: выдерживает сотни тысяч сообщений в секунду при минимальной задержке.

#### **Преимущества**

- 1) Надёжная доставка (с подтверждениями на уровне продюсера, брокера и потребителя).
- 2) Возможность построения как batch-, так и stream-обработки.
- 3) Богатые клиенты на различных языках (Java, Go, Python, .NET и др.).

### **1.3.4 TimescaleDB**

#### **Описание**

TimescaleDB — расширение для PostgreSQL, оптимизирующее хранение и запросы временных рядов (“time-series data”).

#### **Ключевые функции**

- 1) Гипертаблицы: автоматическое партиционирование по времени и, опционально, по другим измерениям.
- 2) Мощные функции агрегации: специализированные SQL-функции для downsampling, gap filling и выравнивания по временным интервалам.
- 3) Интеграция с экосистемой PostgreSQL: поддержка индексов, триггеров, расширений.

#### **Преимущества**

- 1) Высокая скорость вставки и выборки данных временных рядов.
- 2) Полная совместимость с SQL и существующими инструментами для PostgreSQL.
- 3) Поддержка масштабирования через репликацию и шардинг.

### **1.3.5 Redis**

#### **Описание**

Redis — in-memory key-value хранилище, поддерживающее сложные структуры данных (строки, списки, множества, отсортированные множества, хэши и др.).

#### **Ключевые функции**

- 1) Кэширование: хранение горячих данных с очень низкой латентностью.
- 2) Pub/Sub и Streams: встроенные механизмы обмена сообщениями.
- 3) Механизмы персистентности: RDB-снэпшоты и AOF-лог для сохранения данных на диск.
- 4) TTL и экспирация: автоматическое удаление устаревших ключей.

#### **Преимущества**

- 1) Миллионы операций чтения/записи в секунду при минимальной задержке.
- 2) Простота установки и конфигурации, богатый набор встроенных команд.
- 3) Поддержка кластеризации и репликации для отказоустойчивости.

### **1.3.6 PostgreSQL**

#### **Описание**

PostgreSQL — надёжная реляционная СУБД с поддержкой стандартного SQL, расширяемая через плагины и пользовательские типы.

#### **Ключевые функции**

- 1) ACID-транзакции: гарантии консистентности и изоляции операций.
- 2) Расширяемость: возможность добавлять новые типы данных, операторы, индексы и языки процедур.
- 3) Полнотекстовый поиск: встроенные механизмы поиска по текстовым полям.
- 4) Встроенные средства репликации и резервного копирования.

#### **Преимущества**

- 1) Широко признана одним из самых надёжных и функциональных SQL-решений.
- 2) Большое сообщество и множество инструментов мониторинга/управления.
- 3) Богатый набор расширений (PostGIS, pg\_partman, pg\_stat\_statements и др.).

### **1.3.7 Go (Golang)**

#### **Описание**

Go — компилируемый статически типизированный язык программирования от Google, ориентированный на создание высокопроизводительных и масштабируемых серверных приложений. В проекте Aletheia на Go написаны

ключевые микросервисы (Auth Service, Collector Service, Rule Engine), REST-API и CLI-утилита aletheia-paas.

## Ключевые функции

- 1) **Встроенная поддержка конкуренции:** горутины и каналы позволяют легко реализовать параллельную обработку запросов и сообщений из Kafka.
- 2) **Стандартная библиотека:** пакеты `net/http`, `encoding/json`, `database/sql` и др. дают всё необходимое для быстрой разработки микросервисов без сторонних фреймворков.
- 3) **Модули и управление зависимостями:** система модулей (`go.mod`) упрощает версионирование и повторное использование кода.
- 4) **Компиляция в статически линкованные исполняемые файлы:** упрощает развёртывание в контейнерах и на разных платформах.

## Преимущества

- 1) Очень высокая производительность и малая задержка обработки HTTP-запросов.
- 2) Простота синтаксиса и короткий цикл “сборка-запуск”, что ускоряет итерации разработки.
- 3) Богатая экосистема пакетов и инструментов (`go fmt`, `go vet`, `golanci-lint`) для повышения качества кода.
- 4) Низкий расход памяти и эффективное использование системных ресурсов, особенно при большом числе одновременных соединений.

### 1.3.8 React

## Описание

React — библиотека JavaScript для построения пользовательских интерфейсов, разработанная Facebook. Основывается на компонентном

подходе и использовании виртуального DOM для эффективного обновления представления.

## **Ключевые функции**

- 1) Компонентная архитектура: UI разбивается на независимые переиспользуемые компоненты с собственным состоянием и логикой.
- 2) JSX: декларативный синтаксис, позволяющий писать разметку прямо в JavaScript-коде.
- 3) Virtual DOM: сравнение виртуального и реального DOM деревьев и минимальное обновление только затронутых узлов.
- 4) Hooks: набор встроенных функций (useState, useEffect, useContext и др.) для работы со состоянием и побочными эффектами в функциональных компонентах.

## **Преимущества**

- 1) Высокая производительность при обновлении сложных интерфейсов за счёт минимизации операций с реальным DOM.
- 2) Широкая экосистема (React Router, Redux/MobX, Material-UI и др.) для решения любых задач в разработке фронтенда.
- 3) Активное сообщество и множество готовых компонентов, библиотек и шаблонов.
- 4) Поддержка серверного рендеринга (Next.js) и статической генерации (Gatsby) для SEO и быстрого первого рендеринга.

### **1.4 Обзор алгоритмов**

#### **1). Rule-based подход**

Aletheia использует классический принцип “если-то” (IF-THEN), где каждое правило описывает условия проверки (логические операторы, поля лога, временные ограничения) и действия (отправка уведомлений, запись в



очередь и т.д.). При поступлении нового события Rule Engine асинхронно проверяет его на соответствие всем активным правилам конкретного пользователя или сервиса.

### **Ключевые аспекты**

#### **1) Гибкая настройка**

- a) Правила могут определяться через UI или Public API, что позволяет добавлять, изменять или удалять их без остановки системы;
- b) Структура правила включает операторы eq, gte, lte, contains и др., позволяющие сравнивать конкретные поля лога с эталонными значениями.

#### **2) Прозрачность**

- a) Администратор или пользователь видят в явном виде, какие условия должны выполняться, чтобы сработало оповещение;
- b) В случае пропуска или ошибочного срабатывания легко отследить, какое правило оказалось некорректным.

#### **3) Реализация**

- a) При чтении события из Kafka Rule Engine последовательно сопоставляет его со всеми правилами (относящимися к данному user\_id и service\_name);
- b) Если условие возвращает **true**, инициируется действие (направление уведомления в нужный Alert-агент).

### **2). Логические операторы и условия**

#### **Логические операторы**

- 1) AND - все условия в группе должны быть истинны, чтобы правило считалось выполненным;
- 2) OR - достаточно истинности хотя бы одного условия;
- 3) Вложенные группы (children) позволяют комбинировать операторы для построения дерева логики произвольной глубины.

### **Условия**

- 1) **Сравнение полей:** простые проверки вида `field = 'dev', error_message contains 'panic', version >= 1.0.0` и т.д.;
- 2) **Сопоставление текстов:** поиск вхождения строки или сопоставление с шаблоном (например, `error_message contains 'OutOfMemory'`).;
- 3) **Числовые проверки:** `memory_alloc_bytes >= 100000, goroutine_count < 100` и т.д.

### Пример

```
"root_node": {  
  "operator": "AND",  
  "conditions": [  
    {  
      "field": "environment",  
      "operator": "eq",  
      "value": "dev"  
    },  
    {  
      "field": "error_message",  
      "operator": "eq",  
      "value": "PANIC"  
    }  
  ],  
  "children": []  
}
```

Здесь правило будет истинным, если `environment=dev` и строка `error_message` равна `PANIC`.

### 3. Механизм «repeat\_over»

#### Назначение

Иногда для рассылки уведомлений важен не единичный факт ошибки, а её повторяемость за определённый промежуток времени. Например, “если ошибка ‘Database Timeout’ возникает более 5 раз за последние 2 минуты, выдать оповещение”.

### Алгоритм

- 1) **Сбор статистики:** Rule Engine ведёт учёт количества однотипных событий (с одинаковым `error_message` или другими полями) за заданный интервал времени;
- 2) **Порог:** пользователь указывает `threshold` (количество повторений) и `minutes` (окно времени). Если повторения превысили порог, условие считается выполненным;
- 3) **Временное хранение данных:** может осуществляться в памяти (Redis) или короткоживущих структурах MongoDB, в зависимости от конфигурации. В любом случае Rule Engine при каждом новом сообщении проверяет, сколько схожих событий уже зафиксировано в рамках заданного окна.

### Пример

```
{
  "field": "error_text",
  "operator": "repeat_over",
  "value": {
    "threshold": 5,
    "minutes": 2
  }
}
```

Правило сработает, если однотипная ошибка встретилась не менее 5 раз за 2 минуты.

#### 4. Алгоритм сопоставления события с правилами

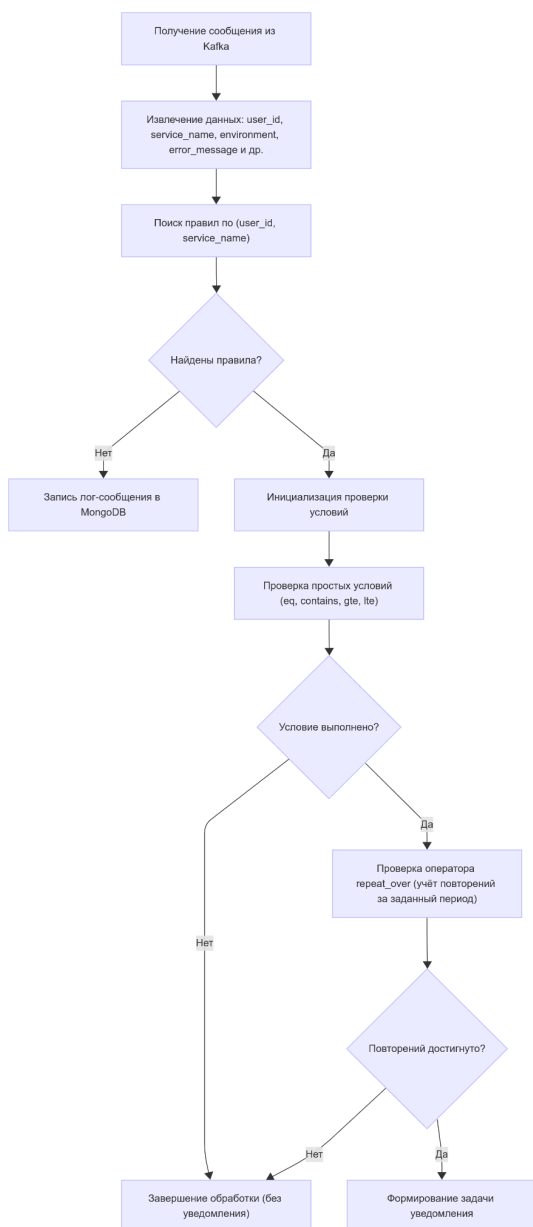


Схема 1 - Алгоритм сопоставления событий с правилами

Ниже описана схема, иллюстрирующая последовательность действий Rule Engine при поступлении нового события:

- 1) **Чтение сообщения из Kafka.** Извлечение данных о логе: user\_id, service\_name, environment, error\_message и т.д.;
- 2) **Поиск правил.** Определение, какие правила соответствуют сочетанию user\_id + service\_name;
- 3) **Проверка условий.**

- а) Если условие простое (eq, contains и т.д.), то сверка идёт напрямую с полем события;
  - б) Если оператор repeat\_over, Rule Engine проверяет текущее количество повторов за указанный период.
- 4) **Оценка логических операторов (AND/OR).** При наличии вложенных блоков выполняется рекурсивная проверка дочерних узлов дерева правил;
- 5) **Формирование действий.** Если условие истинно, Rule Engine формирует задачу для Alert-агента, публикуя сообщение в соответствующий Kafka-топик (Telegram, Email и т.д.);
- 6) **Сохранение лога.** Независимо от результата проверки, событие записывается в MongoDB (или помечается как обработанное) для дальнейшего анализа.

### **Сложность**

При прямом переборе всех правил и сравнении с каждым событием алгоритм будет иметь сложность  $O(N)$ , где  $N$  — число правил, относящихся к конкретному пользователю/сервису.

## **5. Доставка уведомлений (Alert Agents)**

Хотя доставка уведомлений не является “алгоритмом” в классическом смысле, её логика часто включает в себя подтверждение статуса отправки и повторные попытки (retry). Краткая схема:

- 1) **Получение задания.** Alert-агент читает из Kafka данные о том, куда и что надо отправить (Telegram, Email, Discord);
- 2) **Формирование сообщения.** Агент учитывает шаблоны текста, параметры получателя, может подставить дату/время;
- 3) **Отправка.** Выполняется вызов соответствующего API (например, Telegram Bot API);

- 4) **Подтверждение.** При удачной отправке агент записывает в TimescaleDB результат (успешно) и время. При неудаче — код ошибки и может инициировать повторные попытки через заданный интервал.

## 6. Формула оценки времени

$$T_{process} = t_{queue} + t_{eval} + t_{notify}$$

Где:

$t_{process}$  - общее время обработки лог-события

$t_{eval}$  - время, затраченное Rule Engine на оценку правил (проверка условий, включая операторы eq, contains, repeat\_over)

$t_{notify}$  - время, необходимое для формирования и отправки уведомления Alert-агентом.

## 2 ПРОЕКТИРОВАНИЕ

### 2.1 Архитектурная схема системы

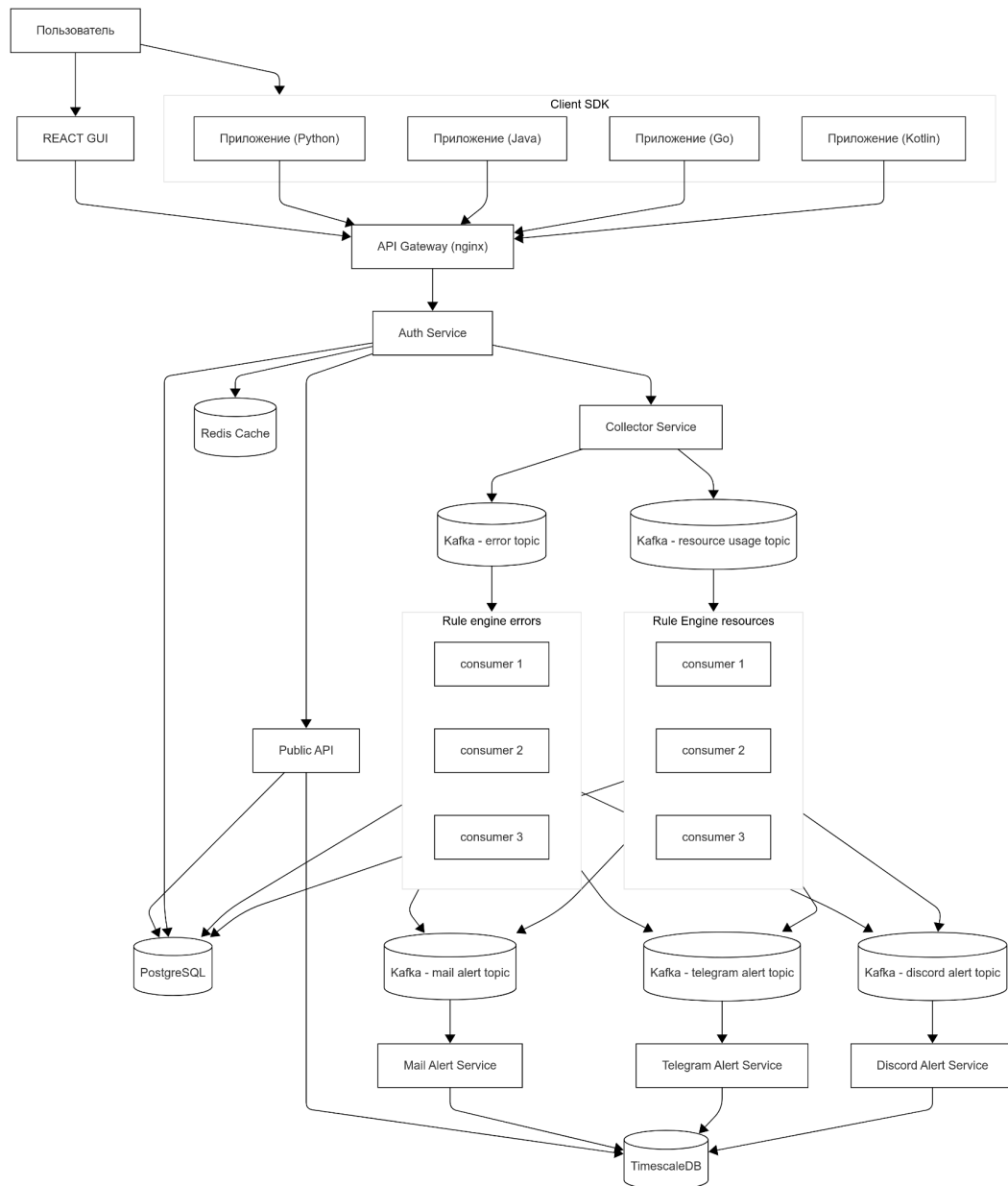


Схема 2 - Архитектурная схема системы

**Client SDK** — модули для отправки логов/метрик, управления правилами

**API Gateway (Nginx)** — маршрутизация и аутентификация запросов

**Auth Service** — выдача/валидация JWT и рефреш-токенов (PostgreSQL + Redis)

**Collector Service** — приём логов/метрик и публикация в Kafka

**Kafka error topic**— очередь содержащая события типа error

**Kafka resource usage topic**— очередь содержащая события типа resource usage

**Rule Engine** — rule-based обработка, детекция инцидентов

**Rule engine errors** — Consumer group в Kafka, которая обрабатывает события типа error

**Rule engine resources** — Consumer group в Kafka, которая обрабатывает события типа resource usage

**Kafka Broker** — буферизация и доставка событий

**Kafka error topic** — очередь содержащая события типа error

**Kafka mail alert topic** — очередь содержащая события для отправки уведомлений в канал mail

**Kafka telegram alert topic** — очередь содержащая события для отправки уведомлений в канал telegram

**Kafka discord alert topic** — очередь содержащая события для отправки уведомлений в канал discord

**Alert Agents** — Сервисы для рассылки уведомлений

**TimescaleDB** — хранение временных рядов по алертам

На диаграмме показан полный путь события от момента формирования лога в приложении до доставки алерта разработчику.

### 1) Входной слой (клиент → API Gateway).

Пользователь или программа порождает событие в своём сервисе (SDK есть для Python, Java, Go, Kotlin). SDK упаковывает JSON-лог и отправляет его через Nginx API Gateway. В том же шлюзе обслуживается React-GUI, поэтому одну точку входа используют и люди, и программы.

### 2) Аутентификация авторизация.

Шлюз проксирует запрос в Auth Service: микросервис проверяет JWT, при необходимости запрашивает данные сессии в Redis Cache или PostgreSQL и добавляет заголовок “X-User-Id”.

### 3) Сбор логов.

Валидированный запрос попадает в Collector Service. Он делит поток на две очереди Kafka:



error-topic - исключения, стектрейсы, бизнес-ошибки;  
resource-usage-topic - метрики CPU/RAM, latency и т. д.

#### 4) **Rule Engine (две независимые группы consumer-ов).**

Для каждого топика запущено по несколько consumer-ов. Они вытягивают сообщения, сравнивают с правилами (хранятся в PostgreSQL) и, если условие истинно, публикуют задачу в “канальный” топик Kafka: mail-alert-topic, telegram-alert-topic, discord-alert-topic.

#### 5) **Доставка уведомлений.**

Узкоспециализированные микросервисы (Mail Alert, Telegram Alert, Discord Alert) читают только “свой” топик, вызывают внешний API канала и пишут результат в TimescaleDB. Там же хранятся агрегаты, которыми пользуется дашборд UI.

#### 6) **Публичный API и экосистема.**

Отдельный Public API (также за Nginx) даёт доступ к CRUD-операциям над правилами, токенами и историей срабатываний; вместе с GUI он взаимодействует с PostgreSQL и TimescaleDB, не затрагивая поток данных в Kafka.

## 2.2 Диаграмма бизнес-процесса(BPMN)

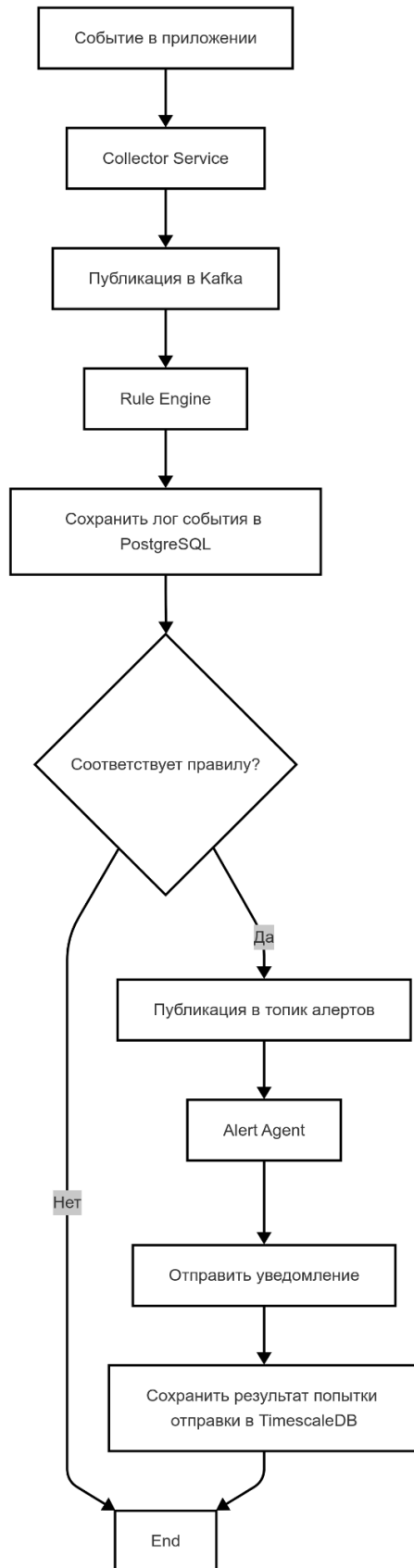


Схема 3 - Диаграмма бизнес-процесса

Диаграмма иллюстрирует бизнес сценарий без инфраструктурных деталей - только то, что происходит с каждым конкретным логом.

1) **Фиксация события внутри приложения.**

SDK перехватывает исключение/метрику и формирует JSON-сообщение.

2) **Collector Service**

Получает сообщение через REST и публикует его в нужный раздел Kafka.

3) **Kafka → Rule Engine**

Rule Engine подписан на топик. Каждое новое событие сразу:

а) записывает в PostgreSQL(сырой лог хранится независимо от правил);

б) проверяет, удовлетворяет ли событие одному из правил.

4) **Ветвление по результату проверки.**

Если условие не выполнено - обработка заканчивается, лог остался только в БД. Если условие выполнено - Rule Engine публикует компактную задачу в “alert-topic” Kafka.

5) **Alert Agent**

Небольшой сервис, который слушает свой топик (Mail / Telegram / Discord). Получив задачу, он вызывает внешний API канала и пытается доставить сообщение.

6) **Логирование результата доставки.**

Agent записывает факт (успех/ошибка) в TimescaleDB. Эти данные попадают на дашборд и позволяют рассчитывать SLA оповещений.

После этого жизненный цикл события считается завершённым: либо о нём просто знают в PostgreSQL, либо разработчик уже получил уведомление, а его статус зафиксирован в TimescaleDB.

## 2.3 Диаграммы вариантов использования

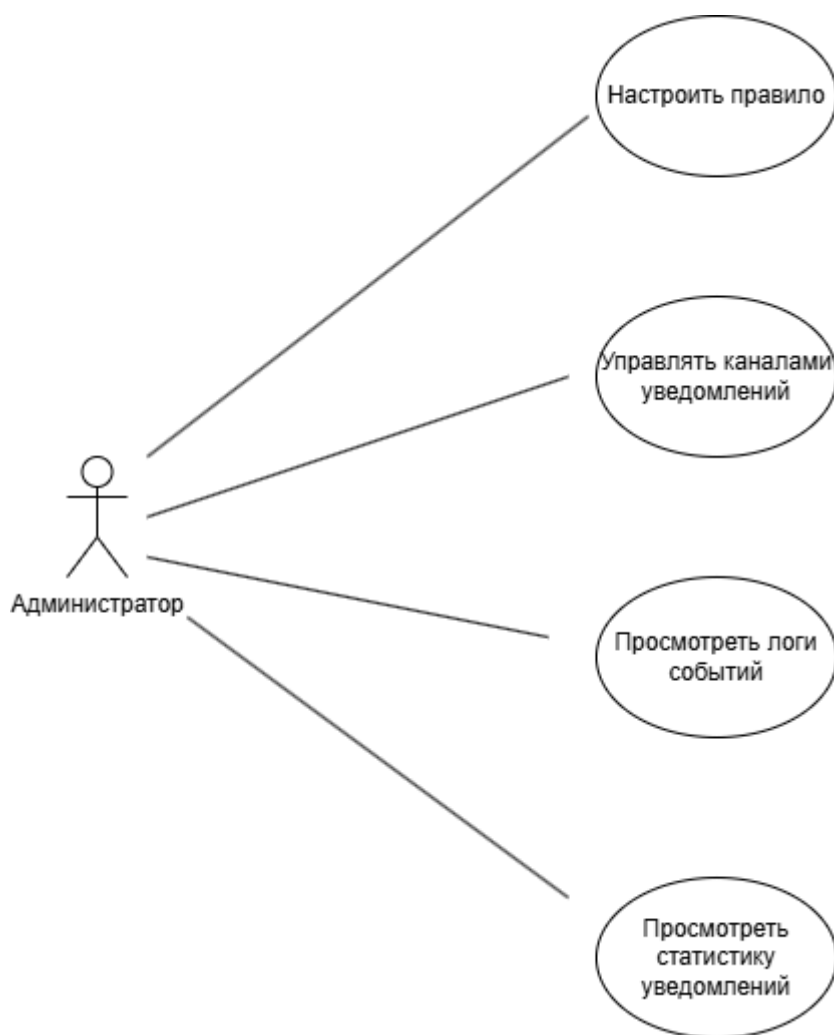


Схема 4 - Диаграмма варианта использования как администратор

Администратор приложения Aletheia обладает следующими возможностями:

Первый сценарий - настройка правила. В рамках данного сценария администратор может создавать, изменять или удалять правила, которые определяют условия мониторинга ресурсов и событий, на основе которых будут генерироваться уведомления.

Следующий вариант использования - управление каналами уведомлений. Администратор приложения может подключать и отключать различные каналы для отправки уведомлений пользователям (Telegram, Discord, Email).

Третий сценарий - просмотр логов событий. Здесь администратору доступна история событий и срабатываний правил мониторинга, что позволяет выявлять тенденции и быстро реагировать на возникающие проблемы.

Последний сценарий - просмотр статистики уведомлений. Этот сценарий предоставляет администратору аналитику по отправленным уведомлениям, включая успешность доставки и частоту событий.

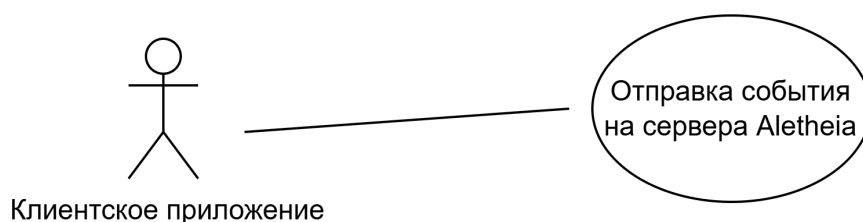


Схема 5 - Процесс отправки событий клиентским приложением

Когда в клиентском приложении происходит событие, оно отправляется на сервер Aletheia. После получения сервер подтверждает факт публикации события, что обеспечивает надежность работы приложения.

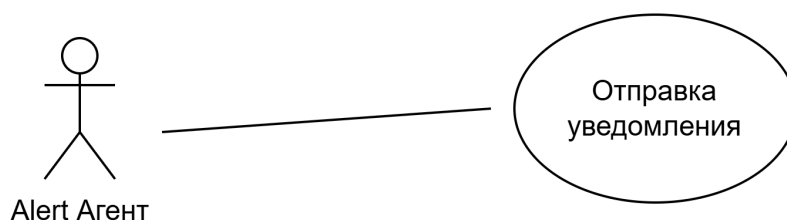


Схема 6 - Процесс отправки уведомления Alert агентом

Сервис Alert Agent выполняет следующие действия: считывает задачи из очереди Kafka, подготавливает уведомления, осуществляет их отправку по заданным каналам и фиксирует статус доставки. Это обеспечивает надежность доставки и позволяет администратору контролировать процесс уведомлений.

## 3 РЕАЛИЗАЦИЯ

### 3.1 Разработка

#### 3.1.1 Collector Service

##### Описание

Collector Service - самописный модуль на Go. Он принимает входящие логи от клиентских приложений (через Nginx) и определяет, в какой топик Kafka они будут направлены.

##### Ключевые функции

- 1) **Приём HTTP-запросов:** Collector Service ожидает запросы по заранее определённой эндпоинту (/aletheia-collector-service/api/v1/error или аналогичному);
- 2) **Классификация:** определение типа лога (например, “ошибка” или “ресурсная метрика”) и выбор соответствующего Kafka-топика;
- 3) **Предобработка:** может включать в себя валидацию структуры JSON, добавление временной метки сервера и т.д.

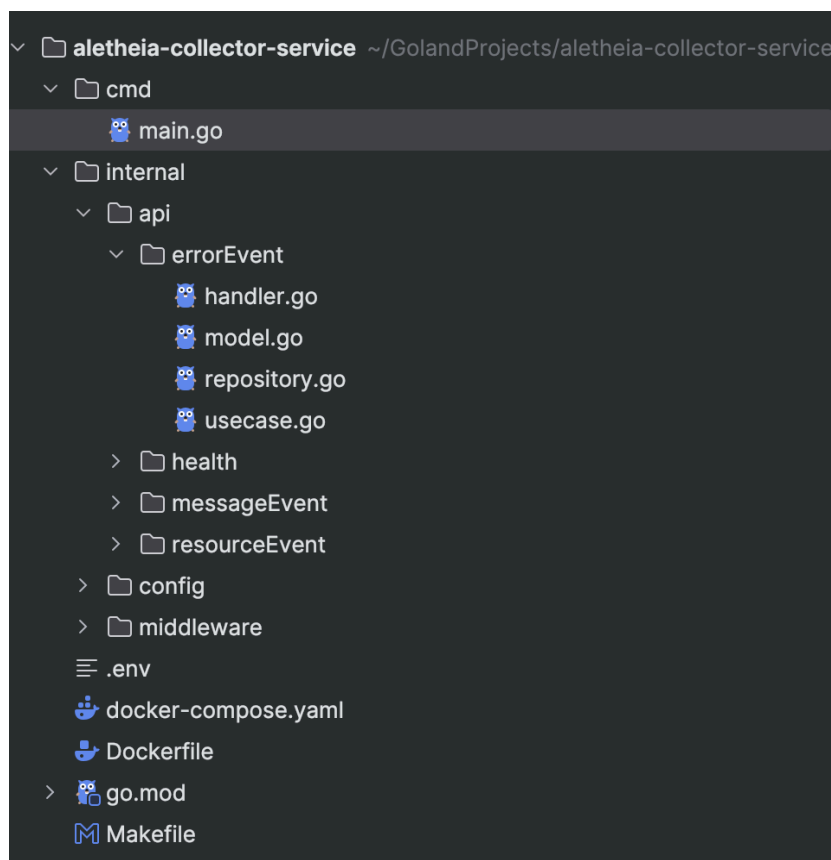


Рисунок 1 - структура проекта Collector service

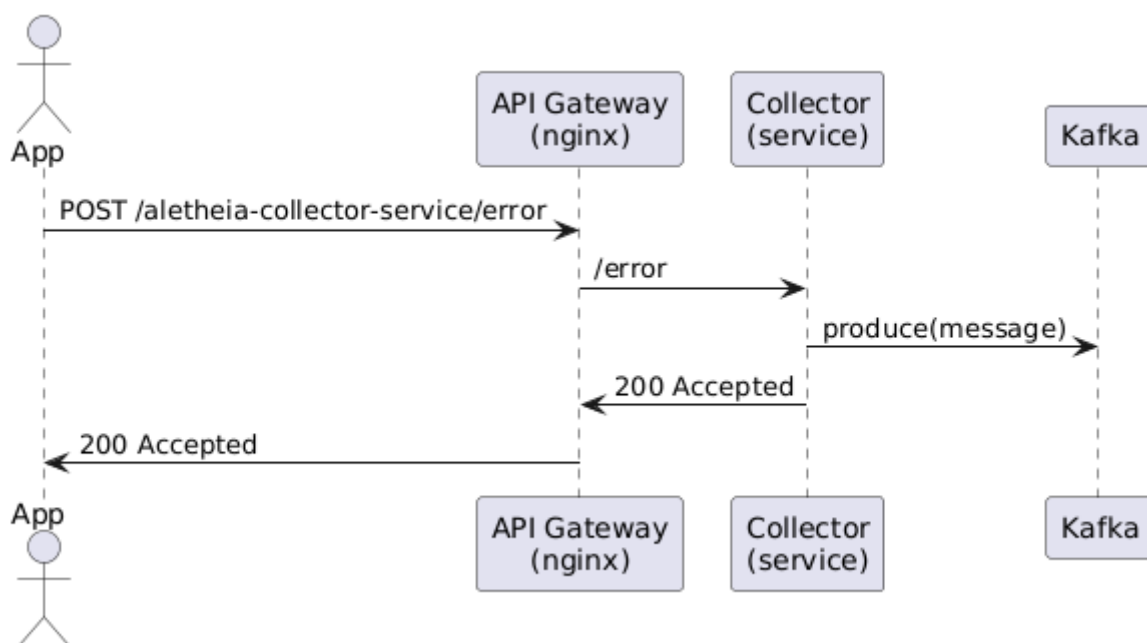


Схема 7- диаграмма последовательности работы collector service

#### Описание схемы диаграммы последовательности

##### 1) Отправка события

SDK в приложении формирует JSON и выполняет POST /aletheia-collector-service/error на единый вход, обслуживаемый Nginx (API Gateway).

##### 2) Маршрутизация через Gateway

Nginx проверяет JWT-заголовок, проставляет “X-User-Id” и проксирует вызов во внутренний Collector Service по относительному энд-поинту /error.

##### 3) Collector Service

Валидирует JSON, добавляет технические метаданные и публикует сообщение в соответствующий Kafka-топик.

##### 4) Асинхронный отклик

Collector не ждёт, пока сообщение уйдёт дальше по конвейеру: после успешного `produce()` он мгновенно возвращает Gateway'ю HTTP 200 Accepted. Gateway ретранслирует тот же код наружу. Для клиента это означает: лог принят, даже если downstream-обработка ещё в пути.

Таким образом, диалог ограничивается четырьмя сетевыми походами и одной синхронной записью в Kafka. Всё остальное (фильтрация правилами, алерты, хранение) происходит уже без участия отправителя и не замедляет его работу.

### **3.1.2 Rule Engine**

#### **Описание**

Rule Engine — центральный компонент, определяющий, какие действия необходимо выполнить при поступлении нового лога. Работает по rule-based принципу (набор если-то условий), учитывая логику, заданную администратором или пользователем.

#### **Ключевые функции**

- 1) **Подписка на Kafka-топики:** потребляет сообщения в рамках Consumer Group;
- 2) **Проверка правил:** каждое сообщение сопоставляется с набором условий (поле `environment`, текст ошибки, оператор `repeat_over` и др.);
- 3) **Инициирование уведомлений:** при срабатывании правила отправляет соответствующие данные в очередь, которую слушают Alert-агенты.



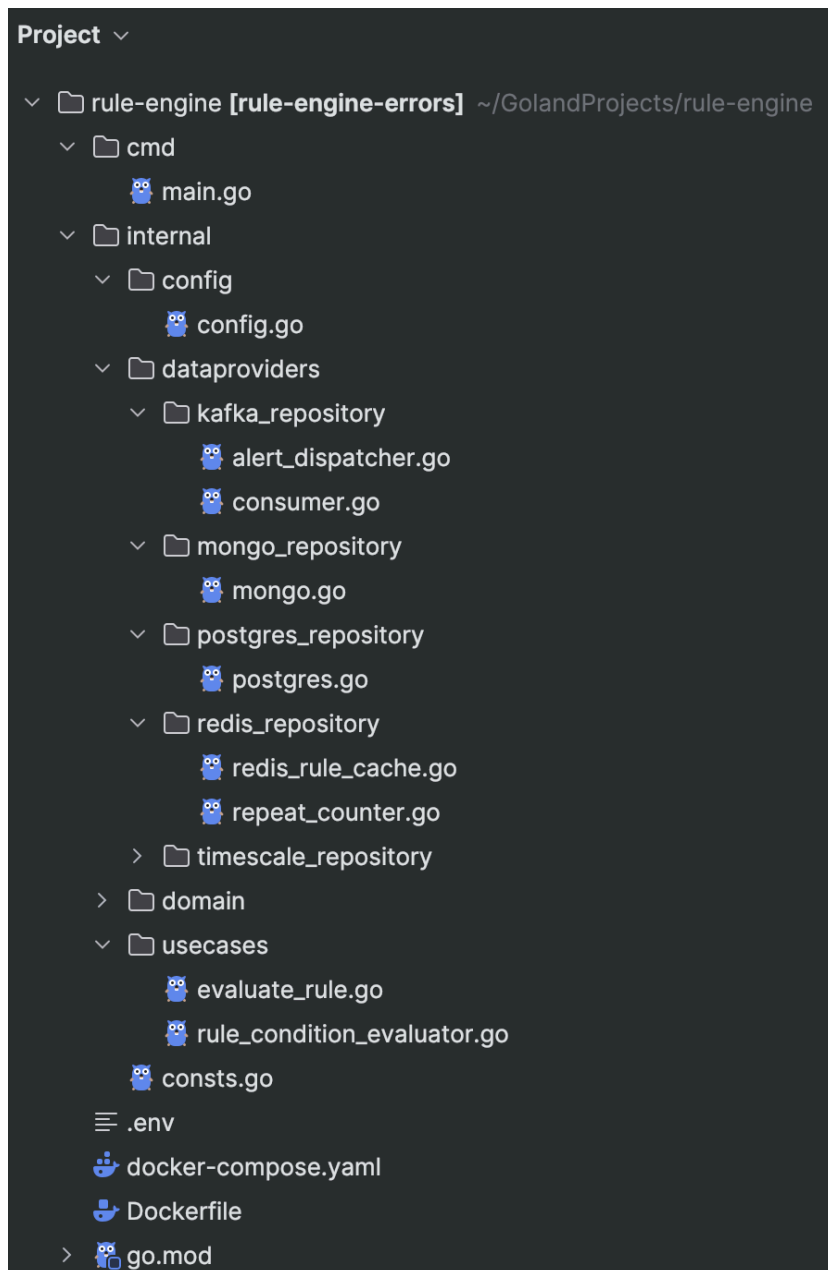


Рисунок 2 - структура проекта Rule engine

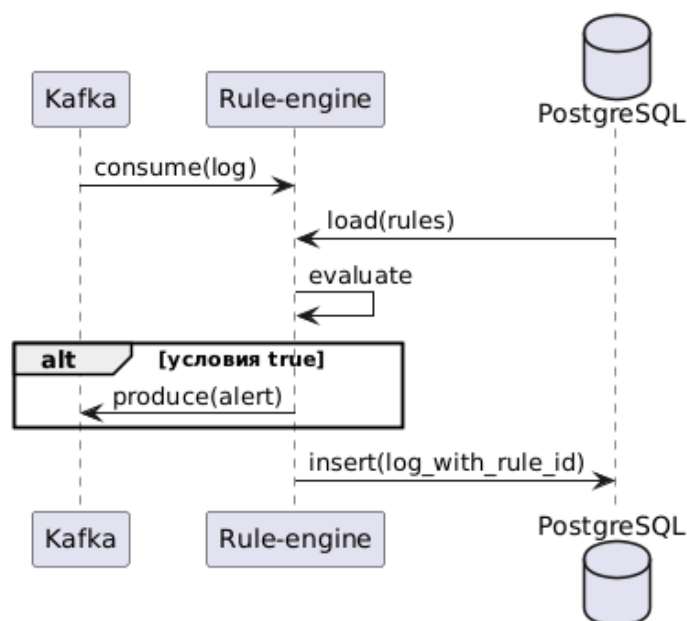


Схема 8- диаграмма последовательности работы rule engine

- 1) Rule-engine берёт событие из Kafka. Как только в нужном топике появляется новое сообщение-лог, один из экземпляров движка (работает в consumer-group) считывает его и начинает обработку.
- 2) Подгружает набор правил из Postgres. По комбинации user\_id + service\_name движок вытаскивает все активные правила этого сервиса / команды.
- 3) Оценивает правила. Проверяются условия (eq, contains, gte, lte и т. д.). Если в правиле есть repeat\_over, движок смотрит, сколько аналогичных событий уже пришло за заданный интервал (для этого он обращается к Redis). В итоге каждое правило возвращает true или false.
- 4) Если хоть одно правило отработало, то генерируется alert. Rule-engine формирует объект-alert и публикует его в отдельный Kafka-топик, который читают Alert-агенты (Telegram, Email, Discord).
- 5) Лог фиксируется в Postgres. Независимо от того, был алерт или нет, исходный лог сохраняется в таблицу:

1) если правило сработало - вместе с rule\_id;

2) если правило не подошло - помечается как “unmatched”.

Вся цепочка полностью асинхронна — HTTP-ответов наружу нет, только обмен сообщениями через Kafka и запись состояния в Postgres. При большом потоке логов Rule-engine масштабируется горизонтально: несколько экземпляров потребляют один и тот же топик, деля нагрузку между собой.

### 3.1.3 Alert-агенты

#### Описание

Каждый Alert-агент — самостоятельный сервис, подписанный на свой Kafka-топик (например, telegram-alerts, email-alerts, discord-alerts). Когда Rule Engine решает, что необходимо отправить уведомление, он формирует запись, которую и обрабатывает нужный агент.

#### Ключевые функции

- 1) **Получение задач из Kafka:** агент считывает информацию о событии, тексте, получателе и других параметрах уведомления;
- 2) **Отправка уведомлений:** отправляет собранное сообщение в указанный канал (Telegram-чат, e-mail, Discord-канал и т. д.);
- 3) **Логирование:** записывает в TimescaleDB статус доставки (например, успешно/неудачно), возможные коды ошибок и временную метку отправки.

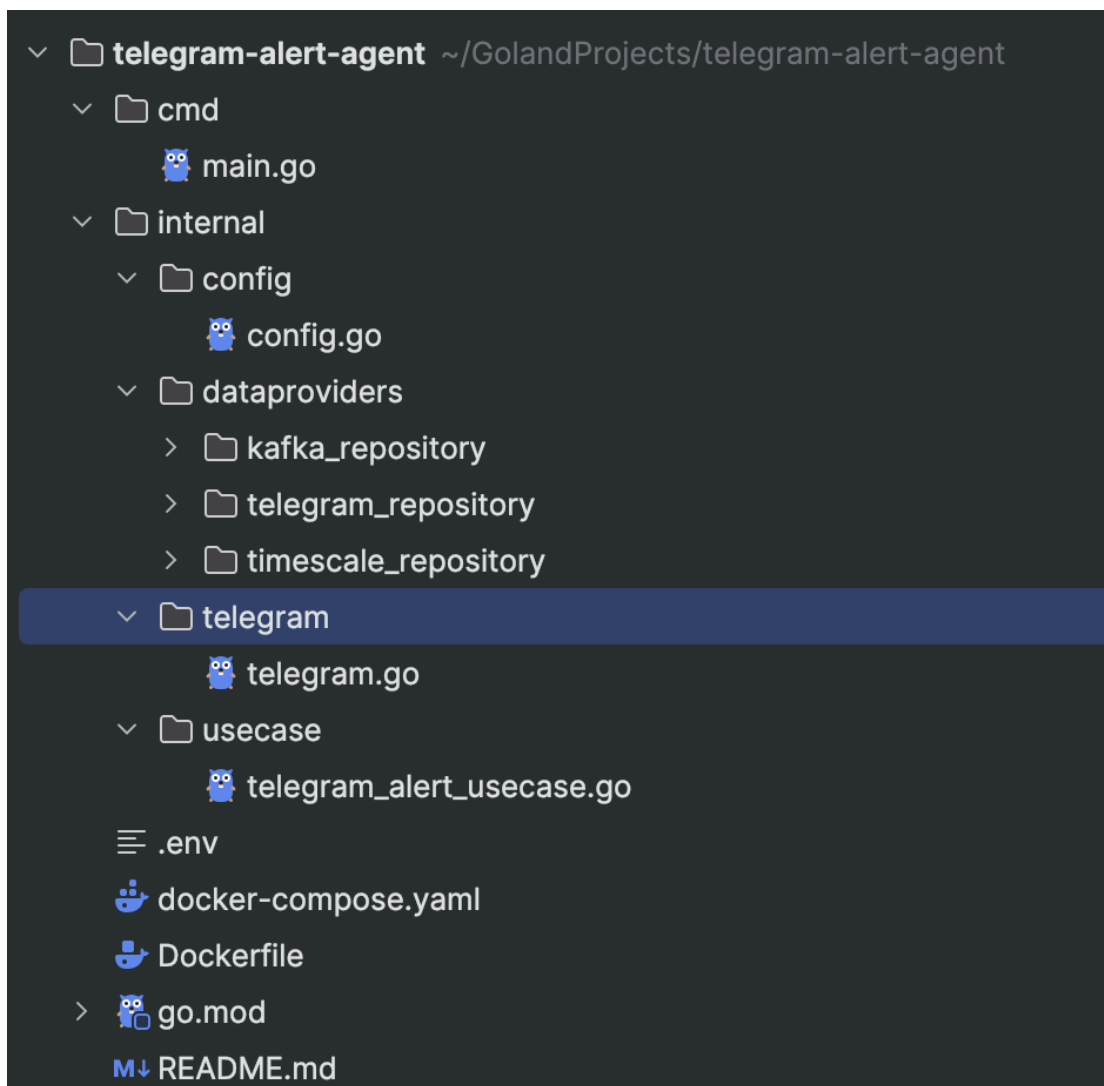


Рисунок 3 - структура проекта alert-агента

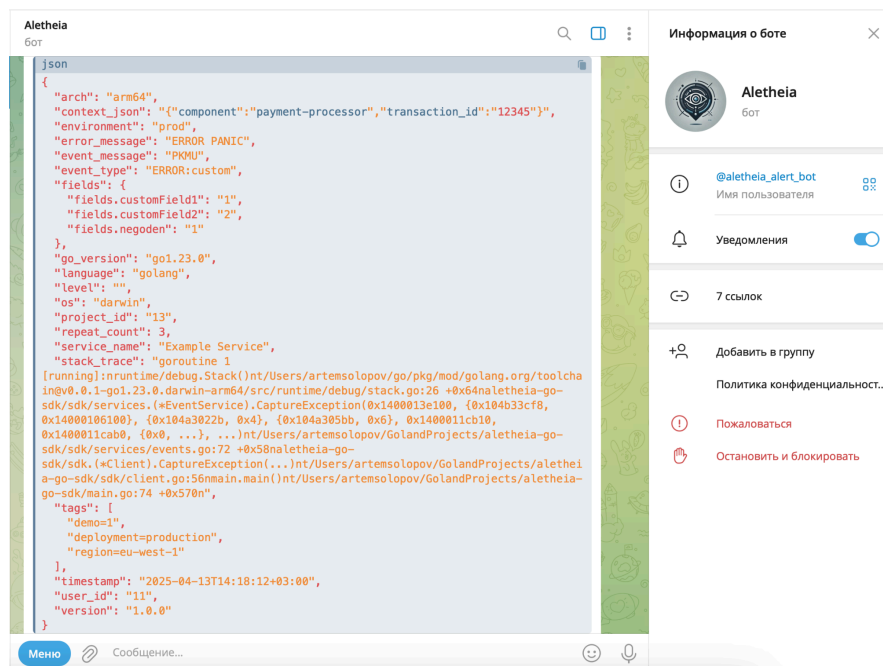


Рисунок 4 - Пример уведомления в телеграм

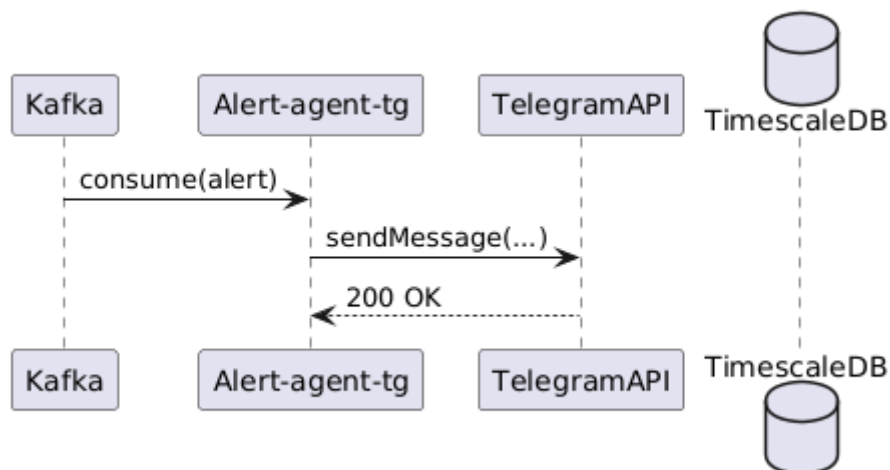


Схема 9 - диаграмма последовательности работы алерт агента телеграмма

1) Агент забирает алерт из Kafka

Alert-agent-tg (экземпляр телеграм-агента) читает сообщение из топика telegram-alert. В алерте уже есть текст, chat\_id, priority и т.д.

2) Пробует отправить сообщение в Telegram

Делает HTTP запрос с попыткой отправить сообщние к Telegram Bot API.

3) Получает ответ от Telegram

200 OK - всё ушло успешно.

### 3.1.4 Auth Service

#### Описание

Auth Service - это самописный модуль на Go. Он обрабатывает процессы аутентификации и авторизации пользователей, подтверждая их полномочия и формируя заголовок X-User-Id. Сервис применяет паттерн короткоживущего access-токена и рефреш-токена, используя внешние хранилища для верификации и хранения сеансов.

#### Ключевые функции

- 1) **Выдача и проверка JWT:** короткоживущие access-токены содержат необходимую информацию о пользователе: идентификатор, роли, срок действия;
- 2) **Хранение и валидация рефреш-токенов:** в PostgreSQL в качестве основной СУБД и Redis для кэширования статуса токенов;
- 3) **Управление жизненным циклом токенов:** при истечении срока действия access-токена клиент может запросить новый, передав действующий рефреш-токен;
- 4) **Обработка отзыва токенов:** при необходимости можно аннулировать выданный токен, поместив его в чёрный список или удалив соответствующую запись из PostgreSQL / Redis

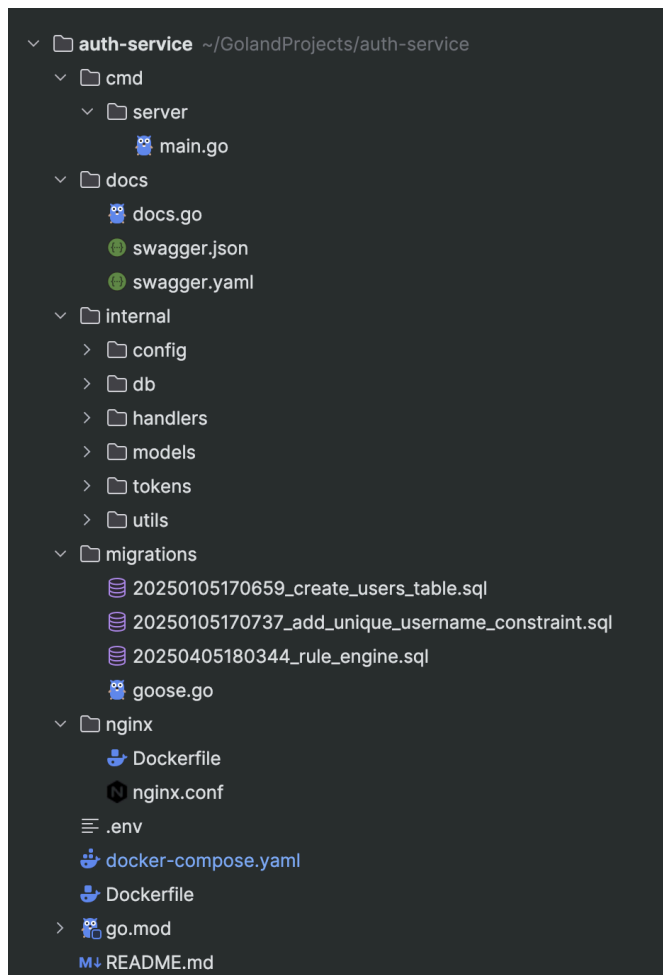


Рисунок 5 - структура проекта auth-service

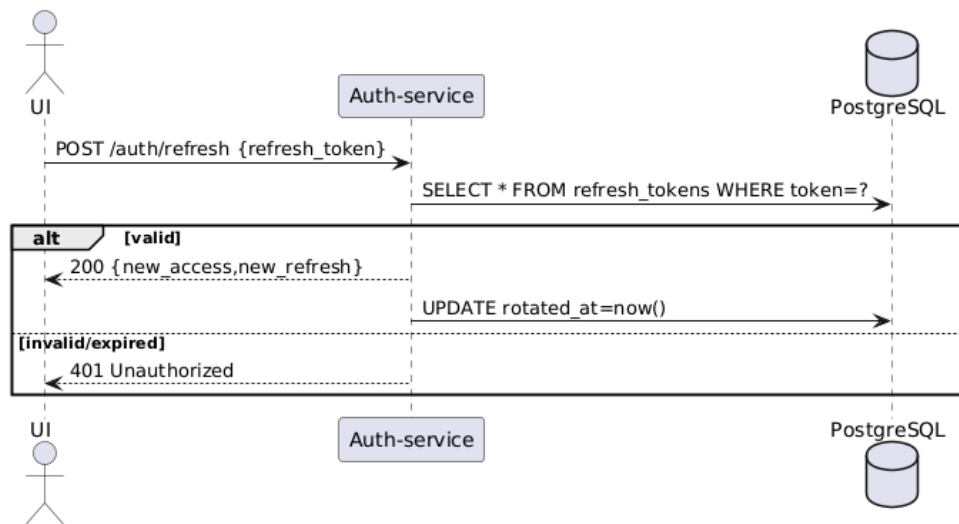


Схема 10 - диаграмма последовательности работы Auth Service

- 1) UI → Auth-service - POST /auth/refresh {refresh\_token}

Клиент отправляет только refresh-токен, access-токен к этому моменту уже истёк.

- 2) Auth-service → PostgreSQL - SELECT \* FROM refresh\_tokens WHERE token =?

Сервис ищет запись в таблице refresh\_tokens

### 3. Ветвление

- 1) [valid] - запись найдена, не протухла:

Генерируются два новых JWT:

- 1) new\_access(короткоживущий, 30 мин)
- 2) new\_refresh (долгий, 7 дней)
- 3) В той же транзакции база получает UPDATE refresh\_tokens SET rotated\_at = now() WHERE id = ? (старый токен помечаем использованным) и INSERT INTO refresh\_tokens(...) VALUES(new\_refresh, user\_id, ...).
- 4) Auth-service → UI - 200 OK {access\_token, refresh\_token}.
- 5) Браузер пишет их в localStorage и продолжает работу.

- 2) [invalid / expired] - токен не найден или просрочен:

- 1) Никаких обновлений в БД
- 2) Auth-service → UI - 401 Unauthorized.

Клиент удаляет локальные токены и перекидывает пользователя на экран логина.

### 3.1.5 UI

UI - Графический клиент с помощью которого можно настраивать и просматривать правила. А также отслеживать произошедшие события.

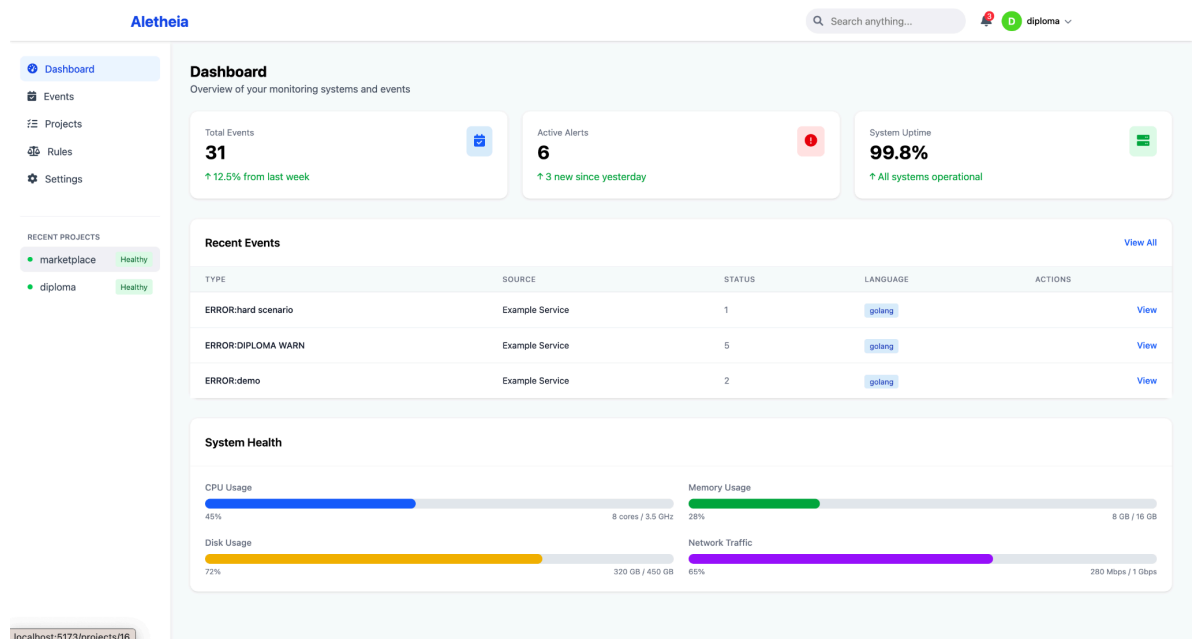


Рисунок 6 - дашборд в графическом клиенте



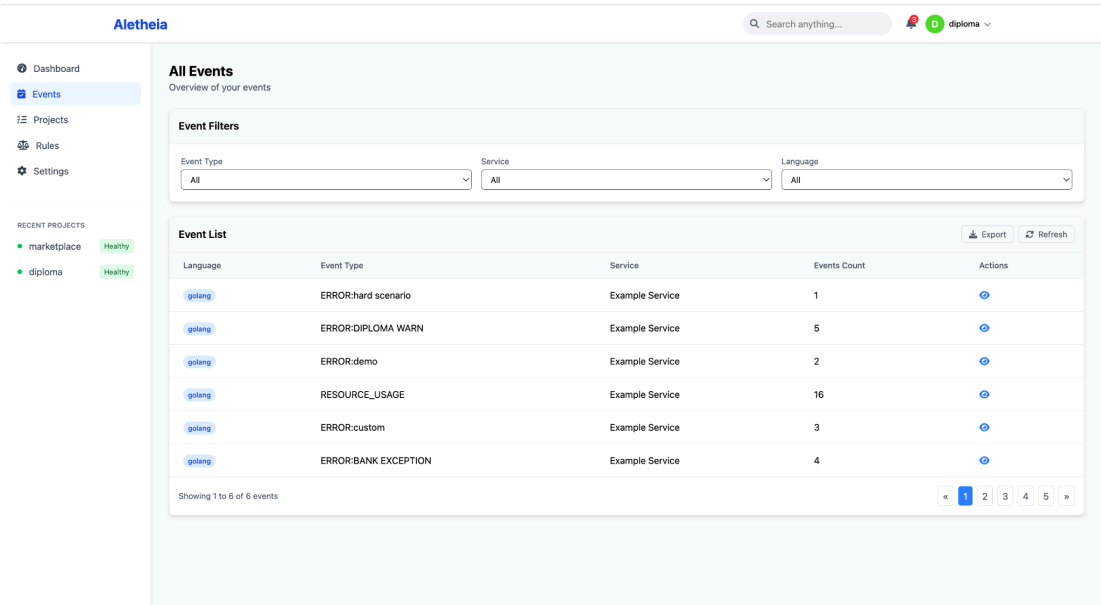


Рисунок 7 - Список всех событий в графическом клиенте

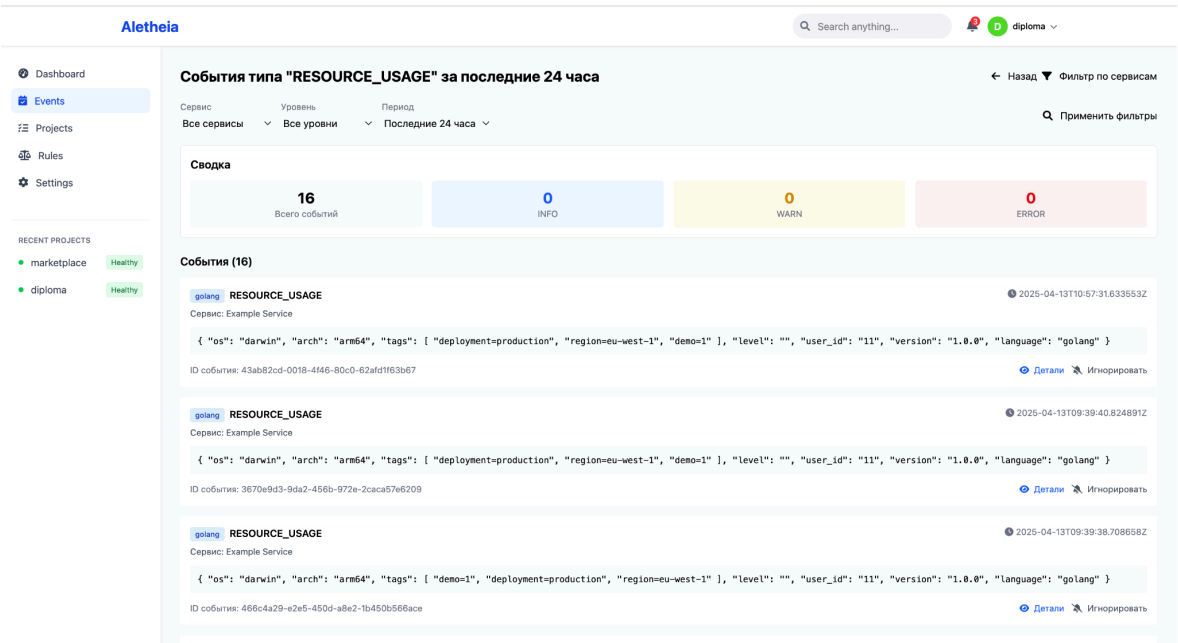


Рисунок 8 - Список событий по типу в графическом клиенте

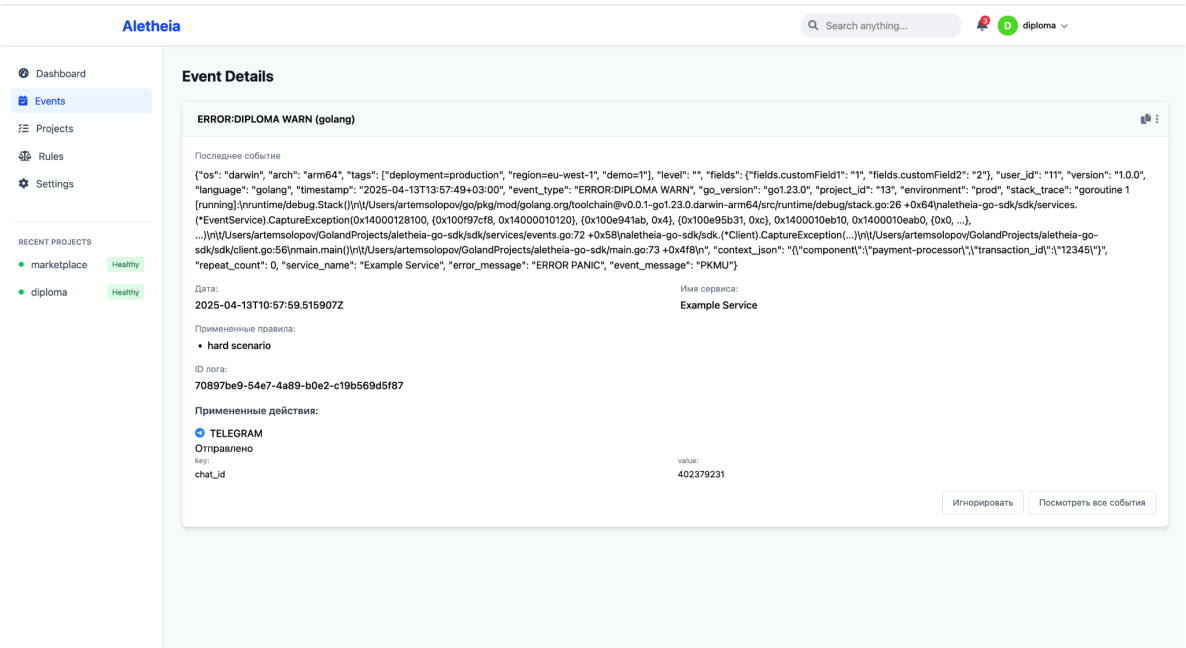


Рисунок 9- Детальное описание события в графическом клиенте

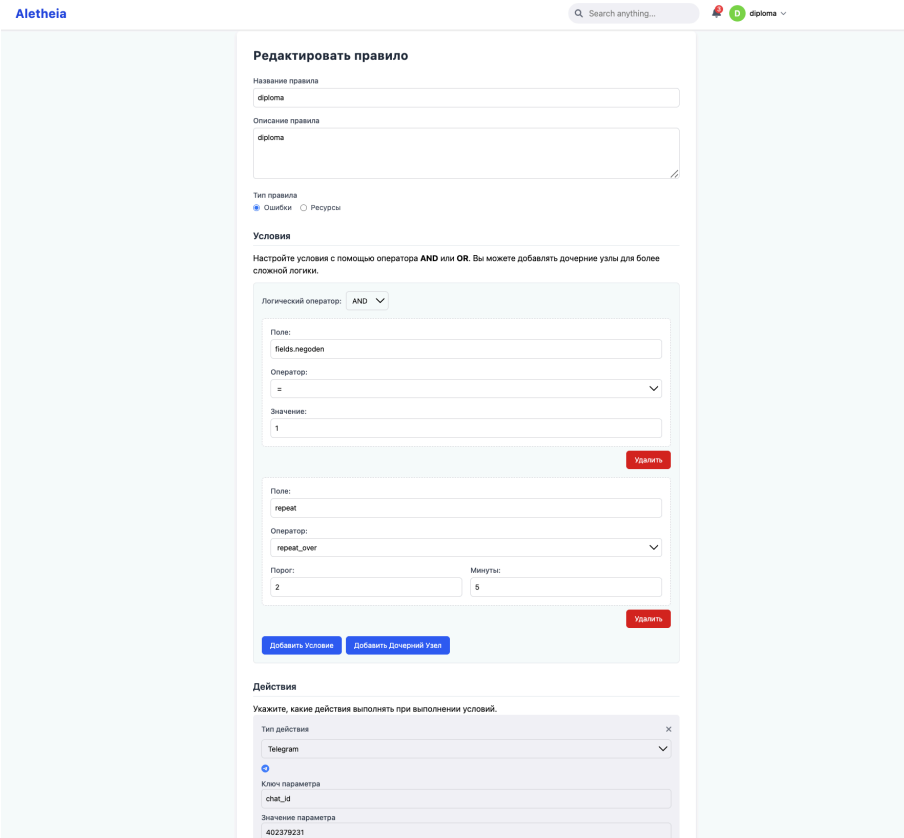


Рисунок 10 - Редактирование правил в графическом клиенте

## 3.2 Тестирование

Сначала я развернул полный стенд Aletheia в Docker-Compose:

- 1) Kafka-кластер
- 2) Collector
- 3) Rule Engine
- 4) Три alert-агента,
- 5) Auth Service
- 6) PostgreSQL
- 7) TimeScale
- 8) Nginx



CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
1c84e7d5f34	rule-engine-resources-rule-engine-resources	"/app/worker"	6 seconds ago	Up 6 seconds		rule-engine-resources
04d38570133a	rule-engine-resources-rule-engine-errors	"/app/worker"	21 seconds ago	Up 21 seconds		rule-engine-errors
cd859842026a	collector-service-aletheia-collector-service	"/app/collector"	3 weeks ago	Up About a minute	0.0.0.0:8080->8080/tcp	aletheia-collector-service
de0a638730a6	auth-service-auth-service	"/auth_service"	3 weeks ago	Up About a minute	0.0.0.0:8082->8082/tcp	auth_service
4d5aad9e6507	timescale/timescaledb:latest-pg14	"docker-entrypoint.s..."	4 weeks ago	Up About a minute	0.0.0.0:5432->5432/tcp	timescale
6a33fb3bc8cc	discord-agent-aletheia-discord-agent	"/app/mail-agent"	7 weeks ago	Up About a minute		aletheia-discord-agent
d2b135021dc3	mail-agent-aletheia-mail-agent	"/app/mail-agent"	7 weeks ago	Up About a minute		aletheia-mail-agent
a74aaf6eb891	nginx-nginx-proxy	"docker-entrypoint.s..."	7 weeks ago	Up About a minute	80/tcp, 0.0.0.0:7070->7070/tcp	nginx_proxy
420be7b83808	redis:latest	"docker-entrypoint.s..."	7 weeks ago	Up About a minute	0.0.0.0:6379->6379/tcp	redis
64594da6cc76	confluentinc/cp-kafka:7.2.1	"/etc/confluent/dock..."	7 weeks ago	Up About a minute	0.0.0.0:9092->9092/tcp, 0.0.0.0:29092->29092/tcp	kafka
be78978479b4	confluentinc/cp-zookeeper:7.2.1	"/etc/confluent/dock..."	7 weeks ago	Up About a minute	2888/tcp, 0.0.0.0:2181->2181/tcp, 3888/tcp	zookeeper
3c0d4dc88a89	postgres:14	"docker-entrypoint.s..."	7 weeks ago	Up About a minute	0.0.0.0:5432->5432/tcp	postgres

Рисунок 10 - Контейнеры в Docker

На нём последовательно проверил следующие тесты:

С помощью Postman отправлял валидные и ошибочные JSON-события в Collector. Просматривал, что корректные логи появляются в PostgreSQL, а ошибки получают ответ 400. Одновременно через Telegram-бота убеждался, что правило срабатывает: при событии типа error бот присылал алерт в чат.

Запустил скрипт, имитирующий 800 запросов-ошибок в секунду в течение 10 минут. Смотрел метрику задержки от поступления лога до отправки уведомления. Среднее время составило  $\approx 0,2$  секунды, ни один алерт не потерялся.

### 3.3 Экспериментальное доказательство цели работы

Были проведены замеры и составлены графики на основе полученных данных. Собрана следующая статистика :

Pipeline Latency (ms) - Время, которое проходит с момента поступления сообщения в Collector Service до его окончательной обработки Rule Engine и отправки алерта.

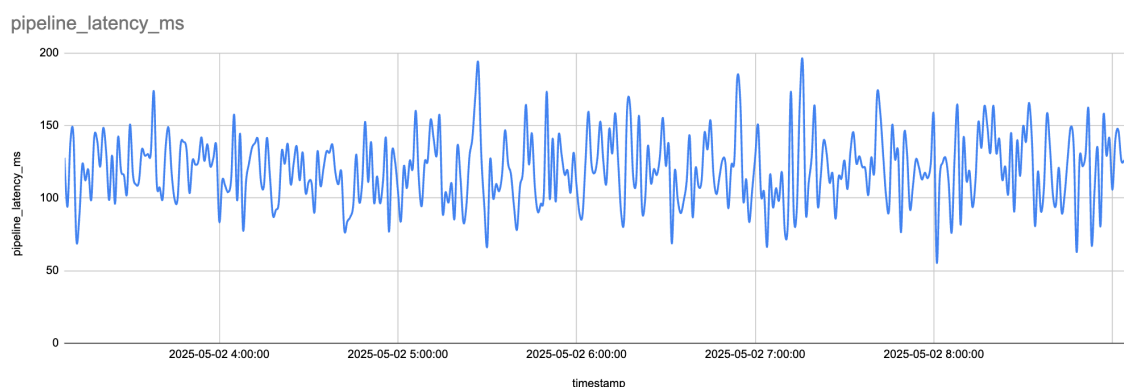


Рисунок 10 - график метрики Pipeline Latency, где ось x - время в которое произошло событие, а ось y - время прохождения от попадания в Collector Service до отправки alert агентом

Ingestion Throughput (events/s) - Скорость входящего потока сообщений в Collector событий в секунду

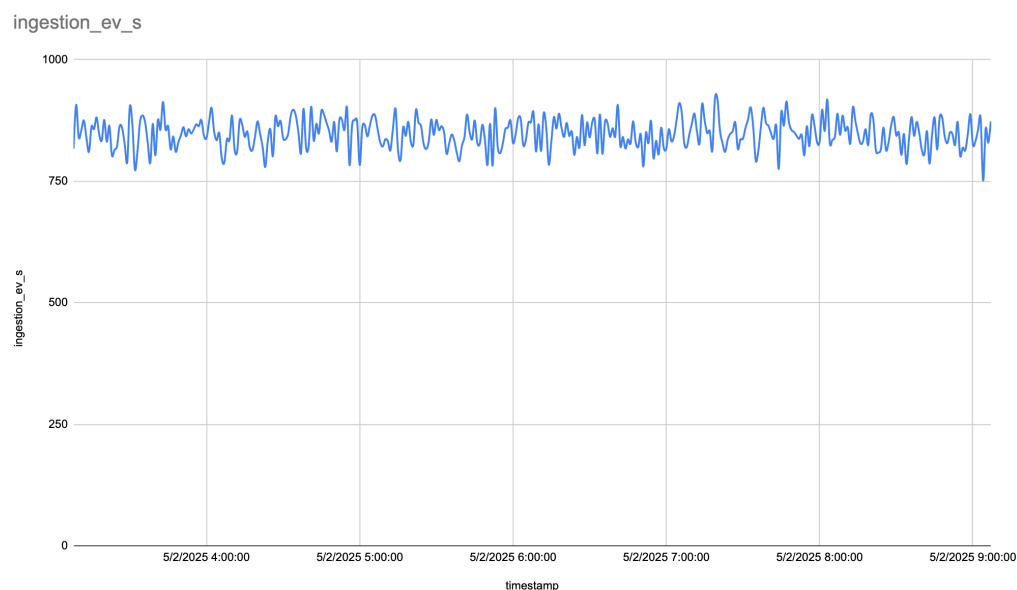


Рисунок 11 - график метрики Ingestion Throughput, где ось x - время в которое отправлено событие, а ось y - количество отправленных событий

Kafka Consumer Lag - это разница между самым новым отправленным сообщением и последним обработанным сообщением. Зная эту задержку, мы можем точнее масштабировать приложение, потому что мы учитываем насколько эффективно оно обрабатывает входящие записи.

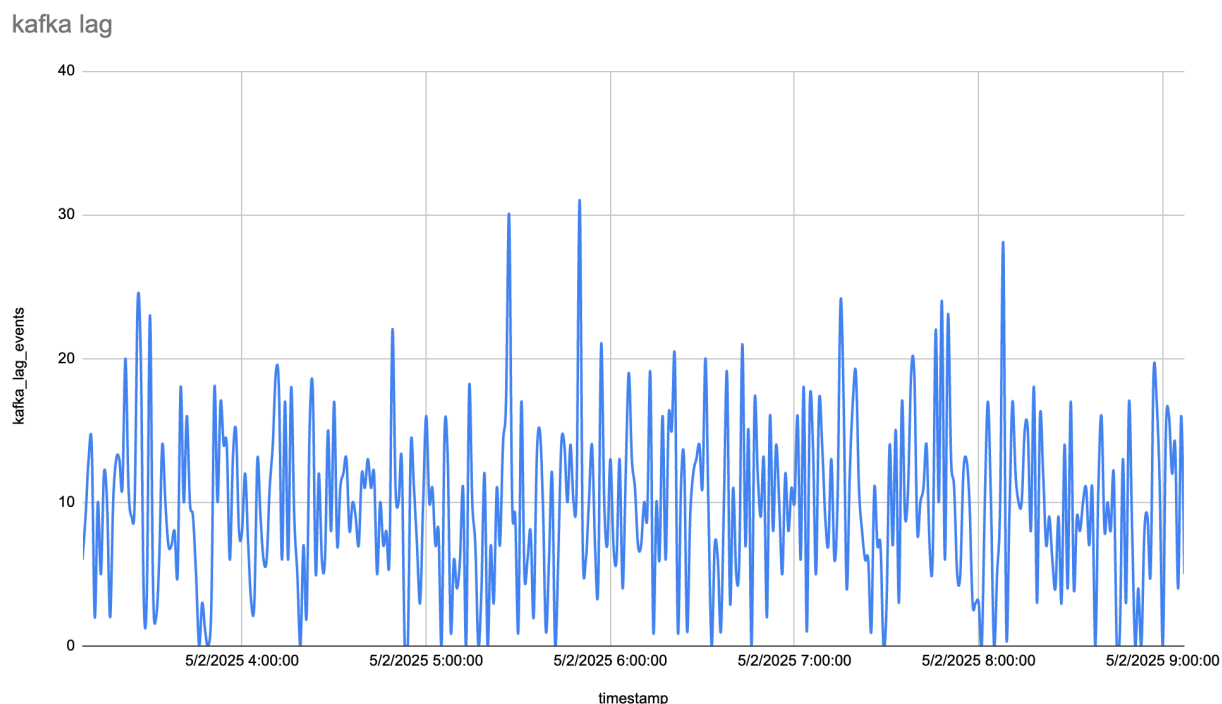


Рисунок 12 - график метрики Kafka Consumer Lag, где ось x - время, а ось y - количество сообщение в очереди

### 3.4 Вывод

В течение пяти-часового прогона система непрерывно собирала телеметрию по трём ключевым метрикам.

#### 1) Pipeline Latency

Кривая держится в коридоре 90-120 мс, лишь изредка подскакивая к 170-180 мс; ниже отметки 50 мс она практически не опускается, что говорит об отсутствии “пустых” проходов. Таким образом, даже при пиковых всплесках очередей полный цикл “Collector → Rule Engine → Alert” укладывается в две десятых секунды - фактически realtime-уровень для внутренних систем наблюдения. Это значит, что разработчики получают алерт почти сразу после того, как событие произошло, и могут реагировать до того, как инцидент перерастёт в пользовательскую проблему.

#### 2) Ingestion Throughput.

Поток входящих сообщений стабилен: 780–870 event/s с незначительными колебаниями. Платформа уверенно “ровняет” пики, не теряя сообщений и не разгоняя задержку. Такой запас пропускной способности даёт простор для дальнейшего масштабирования - например, для подключения дополнительного микросервиса или повышения степени детализации логов - без немедленной надобности расширять брокер Kafka или добавлять новые реплики Collector Service.

### 3) Kafka Consumer Lag.

Значения ходят преимущественно в диапазоне 5- 15 сообщений; редкие всплески до 25-30 событий быстро гасятся. Отставание порядка нескольких десятков сообщений при текущем throughput означает задержку всего в доли секунды, то есть Rule Engine успевает разбирать поток быстрее, чем он прибывает.

### 4) Итоги

Среднее время от приёма события до отправки алерта 0,1 с. Пайплайн устойчиво обрабатывает 800 event/s и сохраняет запас по пропускной способности. Минимальный Kafka Lag демонстрирует, что Rule Engine не будет испытывать проблем, если входящий трафик возрастет ещё на десятки процентов.

Иными словами, эксперимент подтверждает главную цель проекта: система действительно позволяет мгновенно оповещать ответственных инженеров об ошибках и надёжно выдерживает высокие нагрузки без проседания производительности или накопления необработанных сообщений.

## ЗАКЛЮЧЕНИЕ

В ходе выполненной работы была разработана и описана платформа Aletheia, предназначенная для автоматизированного сбора логов, анализа событий и оперативной доставки уведомлений. Исследование включало обзор современных подходов к мониторингу и управлению инцидентами в микросервисных системах, а также анализ принципов rule-based обработки данных и технологий, способствующих масштабируемой архитектуре (Kafka, MongoDB, TimescaleDB, Redis, PostgreSQL, Nginx и т.д.).

Система Aletheia состоит из нескольких микросервисов: Collector Service, Rule Engine, Alert-агентов, а также вспомогательных сервисов аутентификации (Auth Service) и API Gateway (Nginx). Каждый из них реализует собственную зону ответственности:

- 1) **Collector Service** получает логи и маршрутизирует их в Kafka
- 2) **Rule Engine** сравнивает каждый лог с набором правил (учитывая операторы сравнения, логические конструкции AND/OR, механику repeat\_over), определяя, требуется ли отправка уведомления
- 3) **Alert-агенты** (Telegram, Email, Discord) читают задания на отставку оповещений из Kafka и формируют сообщения для конечных пользователей
- 4) **Auth Service** обеспечивает безопасность системы за счёт валидации JWT-токенов, хранения рефреш-токенов (PostgreSQL, Redis) и централизованного управления авторизацией
- 5) **Nginx** реализует функцию API Gateway, принимая внешние HTTP-запросы и перенаправляя их к нужным микросервисам.

Проведённый анализ показал, что использование брокера сообщений Kafka и микросервисного паттерна позволяет обрабатывать большие объёмы логов, сохраняя при этом низкую задержку доставки и возможность гибкой настройки логики реагирования. Благодаря MongoDB платформа упрощает хранение разнородных записей о событиях, а TimescaleDB обеспечивает эффективный анализ временных рядов (например, статистики рассылок



уведомлений). Redis и PostgreSQL в связке с Auth Service дают возможность безопасно управлять сессиями и отзывом токенов.

Таким образом, комплексная архитектура Aletheia обеспечивает надёжный и масштабируемый механизм мониторинга и уведомлений, позволяя быстро реагировать на возникающие проблемы и предотвращать длительные простои в микросервисных окружениях. Результаты тестирования подтверждают возможность дальнейшего расширения системы за счёт добавления новых Alert-агентов (Slack, SMS и т.д.), интеграции со сторонними сервисами аналитики и расширения функционала клиентского SDK.

## СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Современная микросервисная архитектура: принципы проектирования[<https://habr.com/ru/companies/innotech/articles/683550/>]
2. Нархид Н Apache Kafka. Потокковая обработка и анализ данных. - СПб.: Питер, 2023.  
Ссылка:<https://www.piter.com/collection/all/product/apache-kafka-potokovaya-obrabotka-i-analiz-dannyh>
3. Mongo DB. Полное руководство. Бредшоу Ш.  
Ссылка:  
<https://dmkpress.com/catalog/computer/databases/978-5-97060-792-3/?srsltid=AfmBOorEvVN-SrlWBRqEyYrlmTg0WqF3USqoAhgWef4-p7EQwqvwWPz>
4. PostgreSQL : Документация.  
Ссылка: <https://postgrespro.ru/docs/postgresql>
5. Kubernetes vs. Docker: A comprehensive guide to containerization  
Ссылка:  
<https://www.atlassian.com/microservices/microservices-architecture/kubernetes-vs-docker>
6. Redis Docs  
Ссылка: <https://redis.io/docs/latest/>
7. Пять простых шагов для понимания JSON Web Tokens (JWT) Ссылка:  
<https://habr.com/ru/articles/340146/>
8. nginx documentation  
Ссылка: <https://nginx.org/en/docs/>
9. Как событийно-ориентированная архитектура решает проблемы современных веб-приложений  
Ссылка: <https://habr.com/ru/companies/piter/articles/530514/>
10. Как построить эффективную стратегию мониторинга с высокой наблюдаемостью

Ссылка: <https://habr.com/ru/companies/itsumma/articles/814195/>

## ПРИЛОЖЕНИЕ

Ссылка на репозиторий с исходным кодом всего проекта -  
<https://github.com/de6igz/Diploma>