

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РФ
Федеральное государственное бюджетное образовательное учреждение высшего
образования
**ПЕНЗЕНСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ПОЛИТЕХНИЧЕСКИЙ ИНСТИТУТ**

Факультет
«Вычислительная техника»
(наименование)

Кафедра
«Вычислительная техника»
(наименование)

Направление подготовки

09.04.01 Информатика и вычислительная техника
(код и наименование)

Профиль

Прикладной искусственный интеллект
(наименование)

МАГИСТЕРСКАЯ ДИССЕРТАЦИЯ

на тему:

**МОДЕЛИ, АЛГОРИТМЫ И ПРОГРАММНЫЕ
СРЕДСТВА ОПЕРАТИВНОГО ОБНАРУЖЕНИЯ
СОБЫТИЙ В БОЛЬШИХ ПОТОКАХ ДАННЫХ В
РАМКАХ МИКРОСЕРВИСНОЙ АРХИТЕКТУРЫ**

Студент

 23.06.2025 Ершов Даниил Вадимович
(подпись, дата) (ФИО полностью)

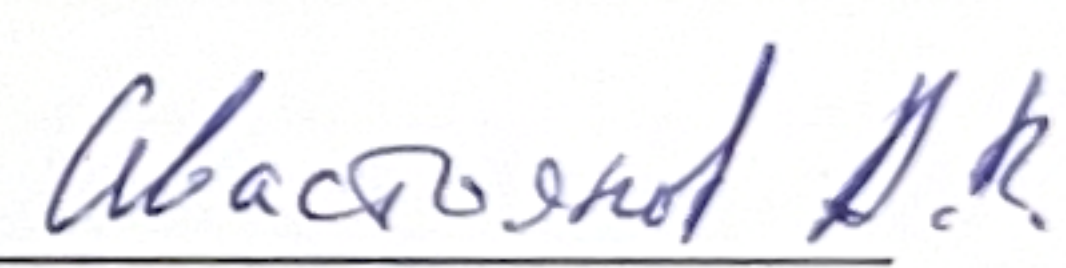
Руководитель

 23.06.2025 Шибанов С.В.
(подпись, дата) (фамилия, инициалы)

Нормоконтролёр

 23.06.2025 Бычков А.С.
(должность, место работы) (подпись, дата) (фамилия, инициалы)

Рецензент

наз. Управляющий ИТ  
(должность, место работы) (подпись, дата) (фамилия, инициалы)

Работа допущена к защите

Заведующий кафедрой

 Митрохин М.А.
(подпись) (фамилия, инициалы)

Работа защищена с отметкой

хорошо (протокол заседания ГЭК от 25.06.25 № 14)

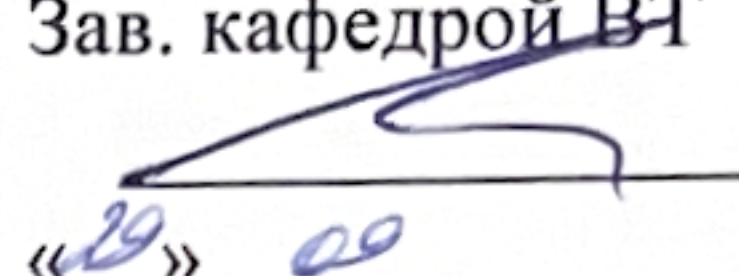
Секретарь ГЭК

 Гурин Е.И.
(подпись) (фамилия, инициалы)

Пенза 2025

УТВЕРЖДАЮ

Зав. кафедрой ВТ



М.А. Митрохин

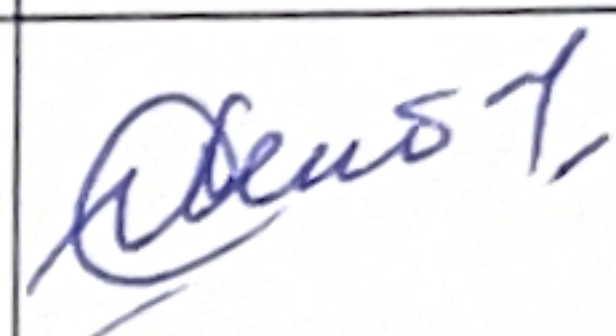
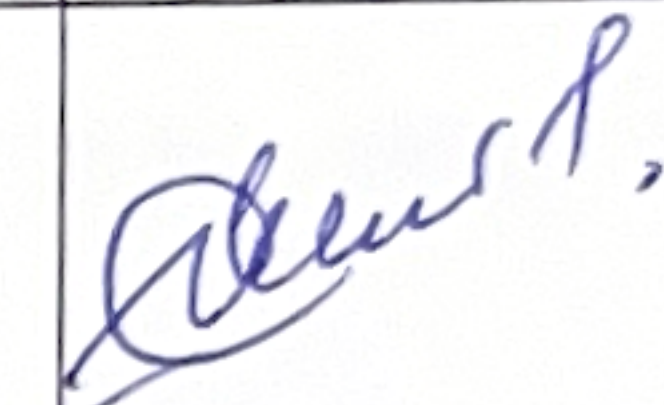
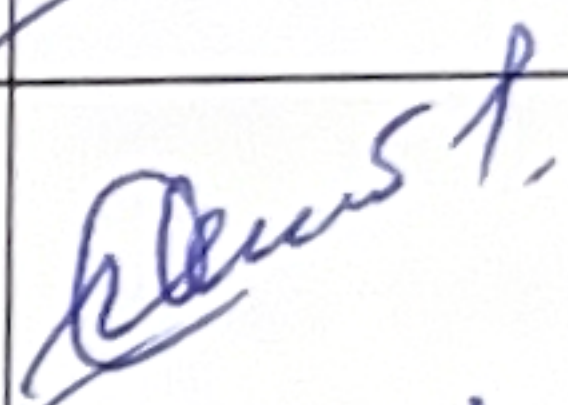
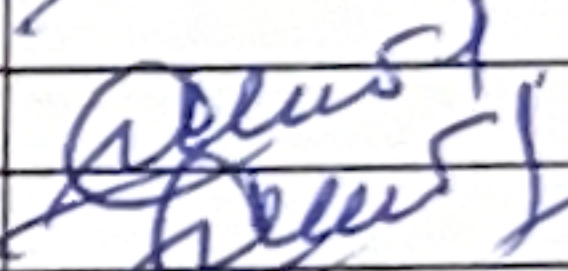
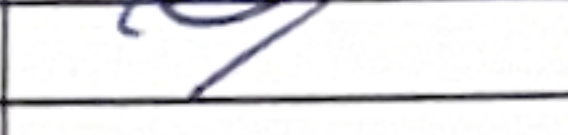
2025 г.

ЗАДАНИЕ

на выполнение магистерской работы

1. Студент гр. 23ВВИм1 факультета ВТ направление 09.04.01
Ершов Даниил Вадимович
(фамилия, имя, отчество)
2. Руководитель ВКР к.т.н., доцент кафедры МО и ПЭВМ ПГУ
Шибанов Сергей Владимирович
(Должность и место работы)
(фамилия, имя, отчество)
3. Время выполнения магистерской работы с _____ по _____
4. Тема магистерской работы Модели, алгоритмы и программные средства оперативного обнаружения событий в больших потоках данных в рамках микросервисной архитектуры
Тема утверждена приказом ПГУ № _____ от _____
5. Задание на магистерскую работу: предмет и назначение разработки, функциональные, программные, технические требования и т.п.
 - 5.1. Разрабатываемое ПО предназначено для оперативного обнаружения событий в больших потоках данных.
 - 5.2. Требования к функциональности: способность микросервиса оперировать потоком гетерогенных событий по мере его движения посредством непрерывных вычислений для реагирования без необходимости явного сбора или сохранения этих событий; способность микросервиса анализировать входной поток простых событий с целью извлечения из него сложных событий; обогащение событий метаданными; маршрутизация событий, наличие механизмов отказоустойчивости системы
 - 5.3 Требования к отслеживанию ПО: используются метрики и события, поступающие от компонентов системы и хранящиеся в Prometheus. Логгирование осуществляется на базе Elasticsearch.
 - 5.4 Требования к ПО: Java, SpringBoot, Kafka, ksqlDB, Prometheus, Grafana, ElasticSearch, Kibana
 - 5.5 Требования к аппаратному обеспечению: Операционная система Linux или Windows (версии 7 и выше), с поддержкой Docker
6. Содержание пояснительной записки магистерской работы: перечень подлежащих разработке, расчетов, обоснований, описаний
 - 6.1 Задача создания программных средств оперативного обнаружения событий в больших потоках данных в рамках микросервисной архитектуры
 - 6.2 Теоретическое представление потока и событий, а также формализация моделей и алгоритмов оперативного обнаружения событий в больших потоках данных
 - 6.3 Программные средства оперативного обнаружения событий в больших потоках данных в рамках микросервисной архитектуры.

7. Календарный график выполнения магистерской работы

Наименование этапов работы	Объем работы	Срок выполнения	Подпись руководителя
1. Задача создания программных средств оперативного обнаружения событий в больших потоках данных в рамках микросервисной архитектуры	20 %	25.11.2023	
2. Теоретическое представление потока и событий, а также формализация моделей и алгоритмов оперативного обнаружения событий в больших потоках данных	35 %	25.07.2024	
3. Программные средства оперативного обнаружения событий в больших потоках данных в рамках микросервисной архитектуры	35 %	25.03.2025	
4. Оформление ПЗ	10 %	30.05.2025	
5. Норм контроль		31.05.2025	

Дата выдачи задания

«29» 09 2023

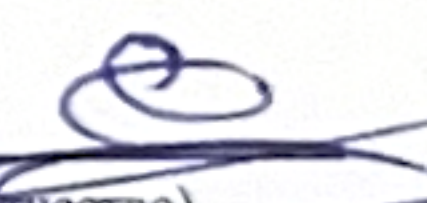
Руководитель магистерской работы

 Шибанов С.В.
(фамилия, имя, отчество)

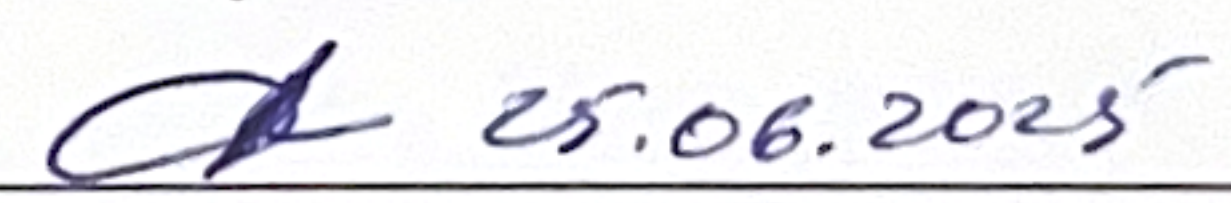
Задание к исполнению принял

«29» 09 2023

Студент

Ершов Д.В. 
(фамилия, имя, отчество)

Магистерскую работу к защите допустить

Декан ФВТ  25.06.2025 / Л.Р. Фионова
(Подпись, дата)

РЕФЕРАТ

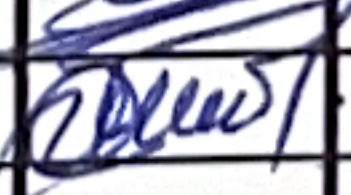
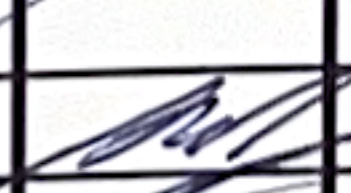
Пояснительная записка содержит 83 листа, 12 рисунков, 4 таблицы, 40 использованных источников, 2 приложения.

ПОТОКОВАЯ ОБРАБОТКА, СЕР, РАСПРЕДЕЛЕННЫЕ СИСТЕМЫ, ТОПОЛОГИЯ ПРОЦЕССА, APACHE KAFKA, KSQLDB, JAVA, SPRINGBOOT

Объектом разработки является программное средство оперативного обнаружения событий в больших потоках данных.

Цель работы – создание подходящего архитектурного/технологического решения для программного средства оперативного обнаружения событий в больших потоках данных и его проверка путем реализации одного из функциональных сценариев обработки событий.

Результаты работы – в ходе выполнения выпускной квалификационной работы было разработано и протестировано архитектурное/технологическое решение по созданию программных средств оперативного обнаружения событий в больших потоках данных.

					ПГУ 1.090401.05.001 ПЗ				
Изм.	Лист	№ докум.	Подпись	Дата	<i>Модели, алгоритмы и программные средства оперативного обнаружения событий в больших потоках данных в рамках микросервисной архитектуры</i> <i>Пояснительная записка</i>	Лит.	Лист	Листов	
Разраб.		Еришов Д.В.							
Провер.		Шибанов С.В.							
Н. Контр.		Бычков А.С.				гр. 23ВВИм1			
Утв.		Митрохин М.А							

СОДЕРЖАНИЕ

ПЕРЕЧЕНЬ СОКРАЩЕНИЙ И ОБОЗНАЧЕНИЙ	7
ВВЕДЕНИЕ	8
1 Задача оперативного обнаружения событий в больших потоках данных	10
1.1 Понятия и классы событий	10
1.2 Способы обнаружения и обработки событий	11
1.3 Постановка задачи	13
2 Модель и алгоритмы оперативного обнаружения событий в больших потоках данных.....	15
2.1 Теоретическое представление потока событий и его характеристик	15
2.2 Модели представления обработки потоков	20
2.3 Алгоритмы обнаружения события в потоках	23
3 Разработка системы оперативного обнаружения событий в больших потоках данных.....	29
3.1 Требования и архитектура системы	29
3.1.1 Нефункциональные требования к программным средствам.....	29
3.1.2 Функциональные требования к программным средствам.....	30
3.1.3 Разработка архитектуры программных средств	33
3.2 Сравнение и выбор средств разработки	35
3.2.1 Шина данных.....	35
3.2.2 Фреймворк потоковой обработки	38
3.2.3 Язык и каркас системы.....	43
3.3 Разработка программных средств	45
4 Тестирование программных средств.....	48
4.1 Представление модели данных тестирования	48
4.2 Сценарий обработки событий.....	49
4.3 Результаты тестирования	53
ЗАКЛЮЧЕНИЕ	58
СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ	60

ПРИЛОЖЕНИЕ А	64
ПРИЛОЖЕНИЕ Б.....	76

ПЕРЕЧЕНЬ СОКРАЩЕНИЙ И ОБОЗНАЧЕНИЙ

БД – база данных

ПО – Программное обеспечение

СУБД – средства управления базами данных

ASP – Analytical Stream Processing

CEP – Complex Event Processing

CQL – Continuous Query Languages

DAG – Directed Acyclic Graph

DSM – Data Stream Model

DSMS – Data Stream Management System

EDA – Event-Driven Architecture

IoT – Internet of Things

NFA – Nondeterministic Finite Automaton

SRM – Stream-Relational Model

ВВЕДЕНИЕ

Современные информационные системы сталкиваются с беспрецедентными объёмами и скоростью поступления данных. Цифровизация бизнеса, повсеместное внедрение интернета вещей (IoT) [1], развитие киберфизических систем и распределённых вычислений привели к тому, что данные поступают непрерывным потоком от разнородных источников: промышленных сенсоров, финансовых транзакционных систем, пользовательских интерфейсов и облачных платформ. В условиях, когда время реакции на происходящие события критично, обработка данных в реальном времени становится неотъемлемым компонентом интеллектуальных сервисов. Текущие традиционные методы анализа, основанные на пакетной обработке, не обеспечивают необходимую скорость или масштабируемость. Такие подходы предполагают предварительное накопление информации, что делает невозможным своевременное реагирование на события, происходящие в окружающей среде. Переход к потоковой модели даёт возможность обрабатывать информацию в процессе получения, без необходимости её предварительного накопления – тем самым устраняя задержки, характерные для пакетной обработки.

Особый интерес в данной области представляет концепция оперативного обнаружения событий – способность системы не просто обрабатывать поток данных, а идентифицировать в нём значимые ситуации (события) по заданным признакам, в том числе составные, агрегированные или контекстно зависимые. Одно из направлений вышеупомянутой концепции известно как комплексная обработка событий (СЕР) [2] и нацелено на выявление закономерностей, аномалий и критических состояний в непрерывных потоках информации. Современные СЕР системы решают задачи выявления паттернов событий в условиях ограниченного времени, высокой изменчивости данных и распределённой архитектуры. Ключевым фактором при этом выступает поддержка алгоритмов, обеспечивающих контекстуальное обогащение входной информации.

Основными задачами данного исследования будут являться обоснование и разработка алгоритмических моделей для оперативного обнаружения событий в больших потоках данных. Акцент работы сделан на практико-ориентированную реализацию системы в рамках микросервисной архитектуры с использованием средств потоковой обработки, обеспечивающих высокую доступность, устойчивость и масштабируемость решения.

1 Задача оперативного обнаружения событий в больших потоках данных

Данный раздел будет обозревать основные понятия потока, события и их методов обнаружения в упомянутых потоках.

Понятия и классы событий

В условиях обработки непрерывного потока информации под событием, как правило, понимается минимальная логически завершённая единица данных, отражающая наступление определённого факта или состояния в системе на заданном временном интервале. Событие может описывать как конкретное измерение с физического устройства, так и программное действие – например, перевод пользователя в другой статус или завершение бизнес-операции.

Для корректной организации логики обработки и последующей интерпретации событий в потоке требуется предварительная типизация входных сущностей. В прикладных и теоретических источниках различают несколько оснований для классификации, представленных в «таблице 1».

Таблица 1 – Классификация событий по критериям

Критерий классификации	Класс	Описание
По сложности	Простые	Элементарные события, неразложимые на более простые.
	Сложные	Формируются из совокупности простых событий, имеют внутреннюю структуру.
	Агрегированные	Сложное событие. Результат обобщения (сумма, среднее, максимум и др.) ряда простых событий.

Продолжение таблицы 1

Критерий классификации	Класс	Описание
	Составные	Сложное событие. Возникают при выполнении логически или временно связанных паттернов событий.
По структурности	Стандартизированные	Имеют фиксированный формат, удобны для обработки и хранения.
	Неструктурированные	Представлены в произвольной форме, требуют нормализации перед анализом.
По регулярности появления	Запланированные	Предусмотрены логикой системы, как правило, повторяются с заданной частотой.
	Незапланированные	Возникают вне сценариев, часто имеют аварийную или исключительную природу.
По регистрации	Зарегистрированные	Фиксируются в логах, базах данных или системах мониторинга.
	Незарегистрированные	Не попадают в системные журналы или теряются до момента фиксации.

|Способы обнаружения и обработки событий

Глобально существует два подхода обработки событий, пакетный и потоковый [3]. При пакетной обработке данные накапливаются в течение

некоторого периода, после чего анализируются разом. Такой подход часто используется в традиционной отчётности, где допустимы задержки, а приоритет отдаётся стабильной загрузке системы. Но при обработке событий, имеющих значение в момент появления, такой способ быстро теряет актуальность. Задержки, присущие пакетной схеме, мешают своевременному реагированию и делают систему негибкой. Поточковая обработка, напротив, анализирует отдельные события сразу, без необходимости ожидать заполнения накопителя. Это особенно важно в сферах, где высока плотность сигналов: финансовый сектор, системы наблюдения, автоматизация производственных процессов. В таких сценариях скорость обработки имеет приоритет над глобальной точностью.

Чтобы такая модель функционировала эффективно, в системах потоковой обработки применяются несколько базовых приёмов. К основным из них относятся:

- разбиение потока на временные промежутки с помощью окон,
- объединение данных из разных каналов для уточнения контекста,
- работа по сохранению состояния потока,
- устойчивость системы при изменениях интенсивности входного потока.

Поток данных не имеет естественных границ. Чтобы система могла выполнять анализ, необходимо логически выделить участки, с которыми удобно работать. Для этого применяются так называемые окна. Окно в данном контексте – это ограниченный фрагмент потока, выделенный по времени либо по количеству поступивших событий.

Разновидности окон зависят от поставленных задач. В простейшем случае поток делится на равные интервалы: например, по одной минуте или по ста сообщениям. Возможны и более гибкие схемы – с перекрытием, смещением или автоматическим закрытием окна по признаку завершённой активности. Последний подход полезен при анализе действий пользователя, где важна не временная отсечка, а логически завершённая сессия.

Часто события поступают из нескольких источников. Чтобы связать их между собой, применяются операции объединения. Это может быть, например, обогащение события дополнительной информацией из справочных данных, или синхронизация нескольких потоков по временной метке. Такие операции позволяют обогатить контекст и уточнить смысл происходящего.

Для анализа последовательностей событий система должна учитывать предыдущее состояние. Поддержка состояния означает, что между обработкой отдельных сообщений сохраняется нужная информация: количество повторов, накопленные значения или флаги выполнения условий. Состояние особенно важно при работе с шаблонами поведения, когда значение имеет не только факт события, но и его положение в ряду.

В условиях высокой нагрузки поток может превышать возможности обработки. В таких случаях важно сохранить стабильность: либо временно приостанавливая приём данных, либо распределяя нагрузку внутри системы. Отсутствие данной функции недопустимо в критически важных сценариях.

Таким образом, оконная обработка, объединение потоков, работа с состоянием и адаптация к переменной интенсивности – это основные приёмы, позволяющие системам эффективно работать с поступающими событиями в реальном времени.

|Постановка задачи

Простая фиксация отдельных событий в потоке не даёт достаточной информации для принятия решений. В большинстве реальных сценариев требуется не только зафиксировать факт, но и понять, с чем он связан: что происходило до него, как он соотносится с другими сигналами, в каком контексте возник. Это означает, что обработка должна быть контекстной и учитывать прошлые состояния. Обычное событие – это лишь точка во времени. Но в задачах мониторинга, аналитики и управления значимы именно закономерности: последовательности, повторения, отклонения от ожидаемого поведения. Такие шаблоны могут состоять из нескольких событий, и без поддержки состояния их невозможно отследить. Например, одиночный вход

пользователя в систему не интересен, но серия из нескольких входов с разных устройств за короткое время – уже потенциальная угроза. Чтобы её зафиксировать, системе нужно запоминать предыдущие входы и сравнивать их с новыми. Другой важный момент – использование метаданных. Само по себе событие может не нести всей необходимой информации. Но если оно сопровождается дополнительными данными – кто инициировал, из какого источника, при каких условиях – появляется возможность фильтровать, группировать и интерпретировать поток более точно. Например, два одинаковых события могут значить разное, если одно пришло от доверенного узла, а другое – с незарегистрированного устройства. Поэтому метаданные становятся важной частью модели. Также нужно учитывать, что события в реальном мире не происходят изолированно. Между ними часто существуют временные зависимости: одно событие должно происходить в течение заданного окна после другого, или наоборот – не должно происходить, если другое уже случилось. Эти зависимости можно описывать через условия во времени, сессионные границы и логические связки, но без чёткой модели они не будут работать устойчиво.

Таким образом, перед системой обнаружения стоит комплексная задача: выделять из потока целевые комбинации событий, отфильтровывать лишнее, учитывать сопутствующие данные и время, при этом делая все это последовательно, с опорой на текущий и прошлый контекст. Иными словами, система должна не просто реагировать на входящие сигналы, а распознавать в них значимые структуры – устойчивые или редкие, в зависимости от сценария. В рамках настоящей работы предполагается сосредоточиться именно на вышеупомянутой функциональности обнаружения событий, что с свою очередь будет определять модель и алгоритмы реализации. Решение будет строиться на принципах гибкой настройки правил и их декларативном способе задания, чего не хватает большинства системам. Формальное описание модели, а также используемые алгоритмы и принципы переходов между состояниями, будут рассмотрены в следующем разделе

2 Модель и алгоритмы оперативного обнаружения событий в больших потоках данных

Данный раздел обзора модели и алгоритмы обнаружения событий в потоке данных, а также теоретическое представление этого потока.

|Теоретическое представление потока событий и его характеристик

Определение и характеристики потока данных. Под потоком данных в общем случае понимается потенциально бесконечная последовательность событий, непрерывно поступающих во времени. Иными словами, поток – это неструктурированный набор данных, а упорядоченная по времени прибытия или по явной временной метке последовательность элементов, которая может рассматриваться как поступающая в реальном времени [4]. Каждый элемент потока имеет определенную структуру, обычно это набор атрибутов, среди которых обязательно присутствует значение события и временная метка события. Временная метка обычно отражает время возникновения события и может быть назначена либо на стороне источника данных, либо при поступлении в систему обработки. Кроме того, события часто имеют ключ идентификации, например, идентификатор источника или объекта, что позволяет разделить поток на логически независимые подпотоки по этому ключу для параллельной обработки или агрегирования по группам. Формально поток данных можно представить как функцию времени, отображающую каждое событие на момент времени его возникновения. Поток – это множество пар (время, значение), поступающих в хронологическом порядке событийного времени. Важно подчеркнуть, что в отличие от статических наборов данных, поток неограничен и не может быть полностью сохранен целиком во внешней памяти. Система не контролирует порядок поступления событий, они могут прибывать в произвольной последовательности, обусловленной внешними процессами, из этого следует, что перестроить порядок извне не получится. Эти фундаментальные свойства отличают потоковые данные от традиционных баз данных. Формально, поток событий можно определить следующим образом по формуле (1.1):

$$S = e_1, e_2, \dots, e_n \quad (1.1)$$

где S – поток (набор) последовательно поступающих событий;

e_i – событие в потоке, упорядоченное по возрастанию времени возникновения;

n – общее число событий в потоке (может быть произвольно большим или неограниченным).

Структура события [6]. Каждое событие в потоке имеет внутреннюю структуру, часто задаваемую схемой набора полей. Обязательным атрибутом является временная метка события, отражающая время события – момент, когда соответствующее событие фактически произошло или было зафиксировано в источнике данных. Помимо этого, может присутствовать идентификатор, например, ID сенсора, пользователя или сессии, который позволяет группировать события по источникам или объектам. Значимая нагрузка события несется остальными полями, например, показания датчиков, содержимое сообщения и т. д. Таким образом, событие можно абстрактно представить как кортеж вида: ключ, значение, временная метка. Современные модели потоковой обработки явно учитывают эти компоненты: например, в модели Google Dataflow (Apache Beam) [5] каждое событие представлено как четырехэлементный кортеж, включающий ключ, значение, метку времени события и принадлежность к определенному окну. Наличие временной метки в каждом элементе позволяет трактовать сам поток как функцию от времени – можно сказать, что поток задает некоторое отображение из времени события в данные событий, которые произошли в эти моменты времени.

Понятие времени [6]. Одной из ключевых особенностей потоковых данных является наличие двух типов времени для каждого события. Первый – это уже упомянутое время события, то есть метка времени, когда событие реально произошло в источнике. Второй – время обработки, то есть фактическое время (системные часы), когда событие было получено и

обработано потоковой системой. Эти два временных домена зачастую не совпадают: между временем события и временем его обработки почти всегда существует рассинхронизация. Например, события могут некоторое время накапливаться у источника или задерживаться в сети, поэтому к моменту обработки система может «опаздывать» относительно реального времени событий на секунды, минуты или даже часы. Логическое время в системе потоковой обработки обычно отождествляется с временем события: именно этот временной порядок (хронология происхождения событий) логично использовать для семантически корректного анализа потоковых данных. Важное следствие, если задача анализа чувствительна к тому, когда произошло событие, то мы должны оперировать именно временем события, а не временем его поступления в систему обработки. Таким образом, время события служит основой для логического, событийного времени обработки, тогда как время обработки отражает лишь прогресс исполнения системы во внешнем физическом времени.

Упорядоченность событий и проблема внепорядковых данных [6]. В идеальном случае поступающие события в точности сохраняют хронологический порядок: каждое следующее имеет большую временную метку, чем предыдущее. Однако реальные потоки данных редко бывают строго упорядочены по времени события, особенно в распределенных системах. Внешние источники могут генерировать события неравномерно, с задержками; сеть передачи данных вносит непредсказуемые задержки и переупорядочивания; некоторые события могут приходить существенно позже из-за временной недоступности источника. Например, если датчик или пользовательское устройство было временно офлайн, накопленные за этот период события поступят в поток с опозданием, их временные метки будут указывать на прошедшее время, тогда как время обработки наступит значительно позже. Такие внепорядковые события приводят к тому, что последовательность, упорядоченная по времени поступления, больше не совпадает с последовательностью по времени происхождения. Это создает

проблемы для алгоритмов обработки, особенно при вычислении агрегатов или обнаружении шаблонов: например, событие, опоздавшее в поток, может «попасть» не в ту временную группу или быть вообще проигнорировано при наивном подходе. Порядок событий, понятие относительное – если использовать время обработки, то порядок определяется моментом поступления, но при использовании времени события порядок должен соответствовать хронологии реального мира. Поскольку события приходят вне порядка, системе требуются специальные механизмы для восстановления или учета хронологии. Одним из распространенных подходов является использование водяных меток (watermarks) – логических маркеров прогресса времени события. Водяная метка с меткой времени t объявляет, что, по мнению системы, все или по крайней мере большинство событий с временем события меньше или равно t уже поступили. Таким образом, получая $\text{watermark}=t$, система «двигает» вперед свое логическое время до t и может, например, завершить окна или вычисления, охватывающие период до t . Это позволяет обрабатывать запаздывающие события корректно: в идеале, если водяные метки настроены правильно, каждая закрытая временная область уже содержит все соответствующие ей события, даже если они приходили не по порядку. Конечно, на практике невозможно гарантировать строгую границу, после которой никаких опоздавших событий не поступит, всегда остается вероятность очень «поздних» событий. Поэтому фреймворки потоковой обработки часто вводят политику допуска опоздавших событий и стратегии их обработки, например, перерасчет с учетом опоздавших, игнорирование событий старше определенного порога, и пр. В целом, проблема упорядочения решается сочетанием логического событийного времени и механизмов, отслеживающих прогресс этого времени в потоке. Формально, допустимое временное отклонения между событиями можно представить следующим образом по формуле (1.2):

$$\max_{i,j} |e_i.ts - e_j.ts| < W, \quad (1.2)$$

где $t(e_i), t(e_j)$ – метки времени событий e_i и e_j ;

W – максимально допустимая ширина окна.

Оконное представление потоковых данных [6]. Поскольку поток является бесконечной последовательностью, операции агрегации и шаблонного обнаружения требуют разбиения потока на конечные фрагменты. В теории и на практике это достигается с помощью окон – ограниченных временных интервалов или фрагментов по количеству событий, внутри которых производится вычисление. Оконные механизмы задают рамки для разделения непрерывного потока на дискретные части. Например, вместо попытки посчитать событие за весь бесконечный поток, вычисляют агрегат за какой-то промежуток времени или же за какое-то количество событий, такие ограничения и есть окно. Окон бывают временными, которые определяются длительностью или границами времени, и счетными, которые определяются фиксированным числом событий. К распространенным видам временных окон относятся:

- фиксированные (интервальные) окна – неперекрывающиеся интервалы фиксированной длины, например окна размером 1 минута, следующие друг за другом,
- скользящие окна – интервалы фиксированной длины с заданным шагом сдвига, что приводит к частичному перекрытию окон (например, окно длиной 10 минут с обновлением каждые 5 минут),
- сессионные окна – интервалы, определяемые периодами активности, разделенными паузами: окно продолжает расширяться, пока события идут без большого перерыва, и завершается, когда обнаружен промежуток тишины больше заданного порога.

Формально, определение окна можно представить следующим образом по формуле (1.3):

$$W_i = \{e \in S \mid t_s \leq t(e) < t_s + \Delta\}, \quad (1.3)$$

где W_i – множество событий, попавших в i -е окно;

$t(e)$ – временная метка события e ;

t_s – начало текущего окна;

Δ – длина окна по времени.

Выбор типа окна и его параметров позволяет задать нужные временные рамки для группировки событий. Таким образом, окно напрямую связано с понятием времени: обычно границы окон определяются в домене времени события (если система поддерживает событийное время), либо в домене времени обработки. Нужно отметить, что использование времени события для окон предпочтительнее, когда важна привязка именно к моментам происхождения событий, так окна действительно отражают натуральные временные промежутки, соответствующие событиям. При использовании окон по времени обработки возможно искажение результата, если поток внепорядковый, т. е. окна по группируют данные по времени их прибытия в систему, что может смешивать старые и новые события в одном окне, не соответствующем реальному интервалу возникновения событий. Поэтому современные системы стремятся реализовать именно оконные вычисления по времени событий. В любом случае, концепция окон является ключевой для разбиения бесконечного потока на последовательности, в пределах которых можно применять классические алгоритмы, агрегаты и обнаружение паттернов.

|Модели представления обработки потоков

В исследованиях и реализации потоковых систем сформировалось несколько моделей представления и обработки потоковых данных. Одна из базовых – модель потока данных (DSM) [7], в рамках которой поток рассматривается как первоклассный объект, отличающийся от статичных реляционных таблиц. В такой модели поток данных трактуется как бесконечная

последовательность кортежей, зачастую задаваемых некоторой схемой, где каждый кортеж имеет, помимо прочего, временную метку. Запросы в этой модели выполняются непрерывно: вместо единоразового вычисления над фиксированным набором, запрос подписывается на поток и обновляет результат по мере поступления новых данных. В традиционных СУБД запрос является временно активным и после выполнения завершается, а данные хранятся постоянно. В потоковой системе (DSMS) напротив, запрос действует постоянно, а данные обрабатываются и далее могут не храниться. В DSMS многие понятия являются противоположными СУБД. В СУБД данные хранятся постоянно, а запросы завершаются после выполнения. В DSMS запрос постоянен, а данные текучи. DSMS разделяют поток на окна и вычисляют заданный запрос на каждом новом окне, порождая бесконечный поток результатов. Такая парадигма требует нового подхода к оптимизации и алгоритмам. Ограничения по памяти и времени вынуждают использовать инкрементальные вычисления, скользящие структуры данных, например, скользящие счетчики, фильтры, стохастические алгоритмы, и допускать приближенные результаты там, где точный подсчет в реальном времени может опускаться. Тем не менее, DSM заложила фундамент для ряда систем потоковой обработки и сформировала понятийный аппарат: потоки, непрерывные запросы, окна, временные метки, и т. д.

Другая важная концепция – поточно-реляционная модель (SRM) [7]. Она нацелена на объединение идей реляционной модели данных с потребностями потоковой обработки. Основная идея здесь заключается в том, что поток можно рассматривать как динамическое отношение или непрерывно меняющуюся таблицу, содержание которого определяется событиями за определенный недавний период времени. Для этого вводится оператор оконного преобразования, который преобразует поток в отношение путем выделения окон, где каждое окно порождает ограниченное подмножество событий, которое можно трактовать как временную таблицу. После этого к полученному динамическому отношению применяются обычные реляционные

операции (SELECT, JOIN, агрегаты и т. д.), как если бы мы работали с классической таблицей. Результат таких операций может быть снова представлен в виде потока, например, путем вывода новой последовательности событий-результатов. В SRM возможны смешанные запросы, которые одновременно оперируют как непрерывными потоками, так и статическими отношениями. Подход SRM лег в основу языков непрерывных запросов (CQL), наиболее известным из которых является CQL, разработанный в проекте Stanford STREAM [8] – он формально определил соответствие между потоками и отношениями через три типа операторов: IStream, DStream, RStream для преобразования вставок/удалений в потоке и поддержки динамических окон. Проще говоря, оконная спецификация задает правило, как из бесконечного потока получить текущее отношение ограниченной длительности, а далее на уровне этого отношения используется знакомый аппарат реляционной алгебры. Поточно-реляционная модель удобна тем, что позволяет переиспользовать принципы SQL и наработки реляционных СУБД для потоковых данных, обеспечивая формальную семантику для запросов на потоки. В академической литературе данный подход описан, например, в работах Арвинда Арасу [9], где потоки трактуются как неупорядоченные отношения, текущие во времени, а окна используются для материализации этих отношений на заданных интервалах времени.

После рассмотрения небольшой теории о потоках данных – от структуры событий и роли временных меток до оконной организации и различных моделей представления можно перейти к методам обнаружения сложных событий. Зная, что поток – это бесконечная последовательность помеченных временем событий, а система способна отслеживать состояние и шаблоны во времени, следующий раздел будет посвящен вопросу формализации механизмов сравнения событий, отслеживания состояний и распознавания шаблонов. Именно такие механизмы необходимы, чтобы решать прикладные задачи обнаружения событий в описанных потоках, формально задавать

искомые шаблоны, удерживать контекст между связными конструкциями и в реальном времени выявлять эти сложные конструкции по заданным условиям.

Алгоритмы обнаружения события в потоках

Парадигма CEP была не случайно упомянута в начале из-за своих алгоритмических характеристик, удовлетворяющих условиям поставленной задачи. CEP обеспечивает обработку потоков данных с микрораздержкой – операции выполняются сразу при поступлении события, без необходимости накопления больших данных в хранилище. По сравнению с аналитической моделью обработки (ASP) [10], где расчёт идёт через аналитические функции в SQL-стиле, и особенно с классической пакетной обработкой, CEP выигрывает в скорости обнаружения закономерностей и реакции на них [10].

В CEP системах для обработки потоков данных определены специальные операторы для описания шаблонов сложных событий [10]. Основные из них:

- оператор последовательности (SEQ),
- дизъюнкции (OR),
- отрицания (NOT),
- оператор временного окна (WINDOW),
- оператор повторения (COUNT).

Ниже формально определяется семантика этих операторов.

Оператор последовательности (SEQ). Задаёт упорядоченное следование двух событий определённых типов. Семантически шаблон обнаруживает совпадение, когда сначала происходит событие, соответствующее шаблону типа A , а затем событие, соответствующее шаблону типа B . Формально множество совпадений для шаблона последовательности двух примитивных типов событий T_1 и T_2 может быть задано через следующую формулу (2.1):

$$T_1; T_2 = \{(e_1, e_2) \mid e_1 \in T_1, e_2 \in T_2, e_1.t < e_2.t\}, \quad (2.1)$$

где T_1, T_2 – множества событий типов A и B соответственно;

e_1, e_2 – отдельные события;

$e_i.t$ – временная метка события e_i .

Таким образом, $SEQ(T_1, T_2)$ требует строгого хронологического порядка появления событий типов T_1 и T_2 .

Оператор дизъюнкции (OR). Определяет альтернативу: шаблон считается выполненным, если происходит событие, соответствующее либо шаблону A , либо шаблону B . В случае двух примитивных типов событий T_1 и T_2 результатом шаблона $T_1 \vee T_2$ является объединение множеств событий этих типов, заданное через следующую формулу (2.2):

$$T_1 \vee T_2 = \{e \mid e \in T_1 \text{ или } e \in T_2\}, \quad (2.2)$$

где T_1, T_2 – множества событий;

e – событие, принадлежащее хотя бы одному из множеств.

Другими словами, дизъюнкция генерирует совпадение, как только поступает событие любого из указанных типов.

Оператор отрицания (NOT). Накладывает условие отсутствия определённого события в заданном контексте. Отрицание обычно используется внутри последовательностей, так называемая негативная последовательность. Например, шаблон вида $SEQ(A, \neg B, C)$ требует, чтобы после события типа A до наступления события типа C не произошло ни одного события типа B . Формально для шаблона с отрицанием $\neg(T_2)$ между T_1 и T_3 условие можно записать следующим образом через формулу (2.3):

$$\{(e_1, e_3) \mid e_1 \in T_1, e_3 \in T_3, e_1.t < e_3.t, \nexists f: (e_1.t < f.t < e_3.t \wedge f \in T_2)\}, (2.3)$$

где T_1, T_3 – множества событий типов A и C ;

T_2 – множество запрещённых событий типа B ;

f – промежуточное событие;

$\nexists$ – отрицание существования события f .

Здесь требование $\nexists f:; (e_1.t < f.t < e_3.t \wedge f \in T_2)$ означает, что в интервале времени между событиями e_1 и e_3 не возникло ни одного события типа T_2 . Таким образом, оператор *NOT* сам по себе не порождает выходного события, а лишь накладывает ограничение на отсутствие нежелательных событий в последовательности. Нужно отметить, что употребление отрицания обычно определено только в контексте последовательностей (например, между двумя «положительными» событиями либо с особыми оговорками в начале или конце шаблона), чтобы сохранялась событийная интерпретация результата.

Оператор повторения (*COUNT*). Позволяет задать шаблон, в котором некоторое событие может происходить несколько раз подряд. Частным случаем является оператор итерирования, который допускает неограниченное число появлений события определённого типа. Более строгий вариант – указание точного или минимального числа повторений. Формально, если T – тип события, то шаблон $COUNT(m, T)$ (итерация T ровно m раз) порождает множество всех последовательностей из m событий типа T с возрастающими метками времени по формуле (2.4):

$$T^m = \{ (e_1, e_2, \dots, e_m) \mid \forall i, e_i \in T, e_1.t < e_2.t < \dots < e_m.t \}, \quad (2.4)$$

где T – тип повторяющегося события;

m – количество повторений;

e_i – событие;

$e_i.t$ – временная метка события e_i .

Иными словами, совпадение по шаблону $COUNT(m, T)$ фиксируется, когда в потоке за определённый промежуток произошло ровно m событий типа T в хронологическом порядке. При необходимости задать диапазон повторений (например, от m до n появлений), такой шаблон можно определить аналогично через объединение последовательностей длиной от m до n .

Для реализации сопоставления шаблонов событий в СЕР системах широко применяется модель конечного автомата [11]. Чаще всего используется

недетерминированный конечный автомат (NFA), состояние которого отражает прогресс в распознавании шаблона, то есть соответствует частично совпавшей последовательности событий – префиксу шаблона. Формально NFA для заданного шаблона определяется через формулу (2.5):

$$(Q, \Sigma, \delta, q_0, F), \quad (2.5)$$

где Q – множество состояний;

Σ – алфавит входных символов (в текущем случае – типы поступающих событий);

$q_0 \in Q$ – начальное состояние;

$F \subset Q$ – множество принимающих (финальных) состояний, соответствующих полному обнаружению шаблона.

Переходы δ задаются в соответствии со структурой шаблона: из каждого состояния $q \in Q$ для каждого следующего ожидаемого типа события T определяется переход Tq' в некоторое следующее состояние q' , представляющее расширенный префикс шаблона. Формально переходы можно представить следующим образом по формуле (2.6):

$$\sigma_{t+1} = \delta(\sigma_t, e_{t+1}), \quad (2.6)$$

где σ_t – текущее состояние автомата;

e_{t+1} – следующее поступившее событие;

δ – функция перехода.

Начальное состояние q_0 соответствует пустому префиксу (ещё ни одно событие шаблона не сопоставлено). При поступлении события, подходящего под первый компонент шаблона, автомат переходит из q_0 в состояние q_1 , фиксируя начало нового частичного совпадения. Каждый дальнейший переход переводит частичное совпадение на следующую ступень, пока не будет

достигнуто некоторое принимающее состояние $q_f \in F$, сигнализирующее, что последовательность событий полностью удовлетворила шаблону. В этот момент накопленная последовательность событий выдаётся как обнаруженное сложное событие. Формально, последовательность переходов можно представить следующим образом по формуле (2.7):

$$q_0 \rightarrow q_1 \rightarrow q_2 \rightarrow q_3, \quad (2.7)$$

где q_0 – начальное состояние;

q_1, q_2 – промежуточные состояния;

q_3 – принимающее (финальное) состояние.

Модель NFA также учитывает отрицательные и итеративные условия шаблона. Например, для шаблона с отрицанием, как в примере $SEQ(A, \neg B, C)$, после приёма события типа A автомат переходит в промежуточное состояние, в котором ожидает либо событие C для завершения шаблона, либо событие B (запрещённого типа) – появление B приведёт к отмене данного частичного совпадения и выход из промежуточного состояния не произойдёт. Ограничение окна времени реализуется за счёт отслеживания метки времени начала каждого частичного совпадения: если автомат не достигает финального состояния для некоторой активной последовательности событий в пределах заданного интервала W , то соответствующее частичное совпадение принудительно сбрасывается. Таким образом, недетерминированный конечный автомат задаёт операционную семантику обнаружения шаблона, последовательно переходя по входным событиям через состояния согласно структуре шаблона и накапливая частичные совпадения до тех пор, пока не будет распознано полное совпадение.

Модели и алгоритмы, представленные в это разделе, полностью покрывают функциональность паттернов, задекларированных в рамках задачи. Это позволяет заложить вышеупомянутые конструкции в основу системы,

разрабатываемой в рамках данной работы. Реализации ПО будет посвящена следующий раздел.

3 Разработка системы оперативного обнаружения событий в больших потоках данных

Данный раздел обозревает основные этапы разработки системы по оперативному обнаружению событий в больших потоках данных.

|Требования и архитектура системы

|Нефункциональные требования к программным средствам

После формализации целей проектирования, изложенных в разделе 1, необходимо определить ключевые нефункциональные характеристики, предъявляемые к разрабатываемому ПО для оперативного обнаружения событий в потоках данных. Эти характеристики не связаны напрямую с функциональностью системы, однако определяют её эксплуатационные качества, устойчивость и применимость в условиях промышленного использования.

Надёжность. Система должна демонстрировать устойчивое функционирование при различных отклонениях от штатного режима работы. Под надёжностью в данном контексте понимается способность сохранять корректную работу при возникновении аппаратных сбоев, ошибочных или неконсистентных входных данных, временной недоступности отдельных компонентов, а также при ошибках, обусловленных внешними факторами. Обнаружение событий не должно прерываться или приводить к потере данных в случае локальных отказов.

Масштабируемость. Архитектура системы должна позволять пропорциональное увеличение производительности при росте объёмов входящих потоков, количества параллельных вычислений или усложнении моделей обработки. Учитывая распределённый характер потока данных, масштабируемость должна поддерживаться как на уровне горизонтального масштабирования вычислительных узлов, так и при необходимости их замены или расширения без остановки основного сервиса. При этом необходимо обеспечить согласованность обработки и сохранение временной упорядоченности событий.

Сопровождаемость. Поддержка жизненного цикла системы предполагает наличие инструментальных и организационных решений, упрощающих её модификацию, настройку и мониторинг. Разработка компонентов должна следовать принципам модульности, изоляции и повторного использования. Под сопровождаемостью также понимается возможность оперативного отслеживания состояния потоков, диагностики узлов и обновления логики обнаружения событий без вмешательства в основную инфраструктуру. Предпочтение отдаётся декларативным способам конфигурации, в том числе через SQL-подобные описания трансформаций.

Указанные нефункциональные требования определяют набор технических ограничений и предпочтений, которые прямо повлияют на выбор архитектурных решений и технологического инструментария.

|Функциональные требования к программным средствам

После формализации целей разработки, изложенных в разделе 1, можно выделить следующие функциональные требования:

- способность системы управлять бесконечным потоком гетерогенных событий посредством непрерывных вычислений для реагирования в процессе поступления этих событий,
- способность системы анализировать поток событий, с извлечением из него шаблонов и верхнеуровневого представления сложных событий (Detecting complex events),
- обогащение событий метаданными (Enriching events),
- маршрутизация событий (Routing events),
- управление отказоустойчивостью системы (Configure fault tolerance),
- надежность и доступность хранения событий (Configure fault tolerance),
- настройка гарантий записи и обработки событий (Configure event delivery guarantee),
- настройка консистентности хранения событий,

- возможность управлять пропускной способностью системы (Control the throughput),
- возможность отслеживания метрик приложения с целью предотвращения внештатных ситуаций (Providing metrics),
- организация допустимой задержки по обработке входных событий,
- ведение истории действий системы (Logging data).

Более наглядно требования представлены на диаграмме вариантов использования рисунка 1.

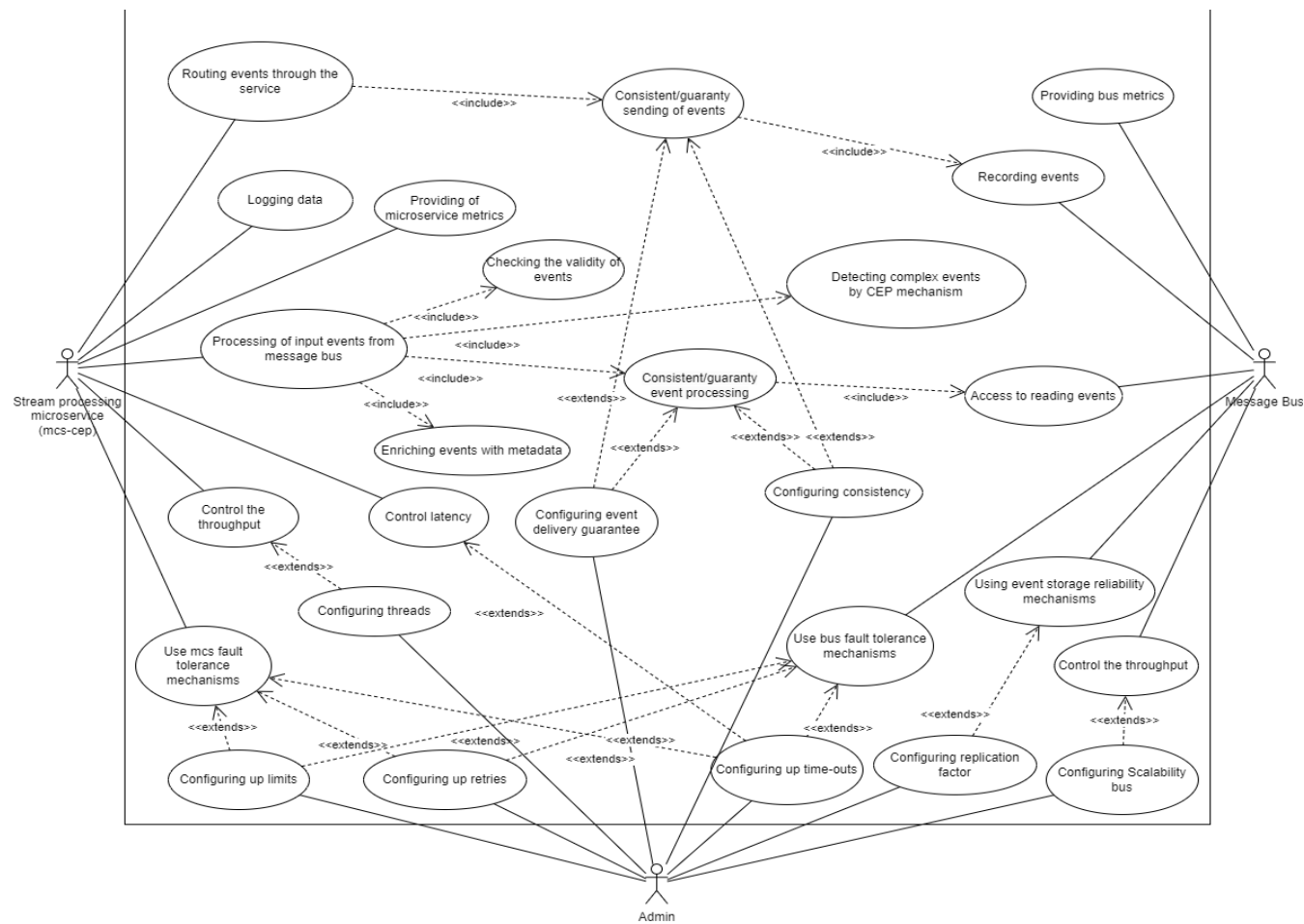


Рисунок 1 – Диаграмма вариантов использования системы потоковой обработки событий

По выполнения вышеупомянутых требований в диаграмму введено 3 типа акторов: система (микросервис) потоковой обработки событий, шина маршрутизации событий и администратор. Администратор может управлять настройкой всех ключевых параметров до старта приложения. Шина событий представлена в упрощенном виде со способностью записывать и давать доступ на чтение.

Разработка архитектуры программных средств

Событийная природа системы и удержание высокой нагрузки, вследствие обработки большого количества данных, а также указанные требования в ранее приведенном разделе требуют создание архитектуры со свойствами распределенности [12]. Выбор был сделан в пользу базового микросервисного подхода [13] и его паттернов [14, 15], использующего событийно-ориентированное взаимодействие (EDA) [16] на основе очереди сообщений [17].

EDA, как уже было сказано, подходит системе по сущности происходящего процесса – обработке событий [18]. Архитектура, управляемая событиями это парадигма архитектуры программного обеспечения, способствующая созданию и обработке событий. Свойство событий заключается в их непредсказуемом характере и отсутствии явного связывания с потребителями этих событий. События возникают самостоятельно и не зависят от наличия потребителей. Данная архитектура способствует практически полному разделению сервисов и независимому общению между ними. Независимость сервисов позволяет им свободно масштабироваться, убирает критические точки отказа системы и позволяет вести свободную разработку.

Верхнеуровневая архитектура сервиса на данный момент представлена диаграммой уровня C1 методологии C4 [19] на рисунке 2.

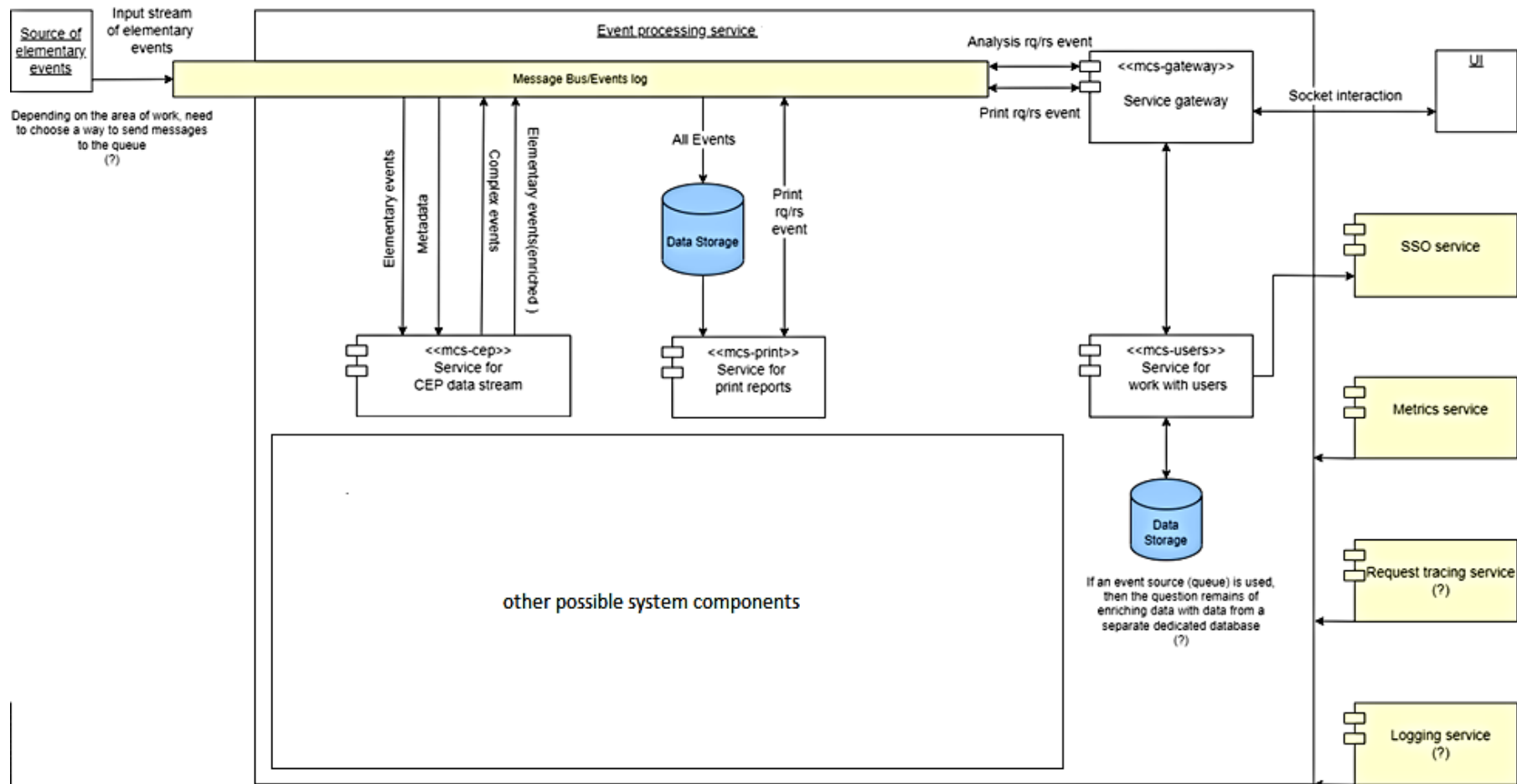


Рисунок 2 – Верхнеуровневая диаграмма компонентов С2 сервиса потоковой обработки событий

Доставка событий в сервис производится посредством центральной очереди – шины (message bus), которая занимается сбором, хранением, обработкой и маршрутизацией событий на основе заранее predetermined логики.

Микросервис потоковой обработки событий (mcs-ser) является ядром системы. Микросервис SER обрабатывает простые события в реальном времени, обогащает их метаданными [20], группирует/агрегирует/соединяет в абстракции более высокого уровня и передает их в шину для последующей маршрутизации. После прохождения по шине данных все события складываются в хранилище. В системе присутствует сервис доступа к этому хранилищу (внешнее UI-решение) и сервис генерации отчетов (mcs-print) для возможного анализа событий data-инженером.

Центральной точкой входа в приложение является сервис Gateway, который является маршрутизатором и собирает все точки доступа к разрозненным сервисам в одном месте. Gateway построен по стандартной REST [21] архитектуре. Для взаимодействия с клиентскими приложениями, конфигурации сервиса и управлением активных правил имеется пользовательский сервис (mcs-users).

|Сравнение и выбор средств разработки

|Шина данных

Выбор платформы для организации канала передачи событий напрямую зависит от её способности справляться с переменным объёмом трафика. При этом важны не только технические показатели, но и устойчивость функционирования системы в целом при растущей нагрузке. Дополнительным условием является свободное распространение кодовой базы, в перспективе это позволяет модифицировать архитектуру под специфические условия. Тем не менее, число решений, которые подходят под вышеупомянутые условия, весьма ограничено.

Apache Kafka [22]. Платформа, развиваемая под эгидой Apache Software Foundation. Архитектура Kafka построена вокруг идеи распределённого

журнала событий, где ключевыми сущностями выступают топики и партиции, тобеспечивающие масштабируемость и упорядоченность хранения данных. Брокеры Kafka взаимодействуют напрямую с клиентами, сохраняя сообщения на файловой системе и осуществляя их репликацию между узлами. Поддержка отказоустойчивости реализуется за счёт синхронной и асинхронной репликации. Ввиду своей зрелости и широкого сообщества, Kafka имеет богатую экосистему (Kafka Connect, Schema Registry, ksqlDB и др.) и является де-факто стандартом в современных потоковых системах.

Apache Pulsar [23]. Представляет собой альтернативную систему потоковой передачи данных, также ориентированную на масштабируемые распределённые сценарии. Устройство Pulsar построено на разнесении логики маршрутизации и слоя хранения. Последний реализуется через BookKeeper — отдельную систему долговременного размещения сообщений. Архитектурное разнесение компонентов управления и хранения предоставляет определённую свободу конфигурации, однако требует дополнительных затрат при организации среды исполнения. Вместе с тем, платформа допускает сопряжение с рядом внешних инструментов потоковой аналитики, включая Flink и Presto. Предусмотрена поддержка механизмов многоканальной подписки и репликации, что позволяет адаптировать систему под разнообразные сценарии обработки.

RabbitMQ [24]. Система обмена сообщениями, изначально построенная на модели очередей и реализующая протокол AMQP. В отличие от Kafka и Pulsar, RabbitMQ ориентирована преимущественно на низкоуровневый обмен сообщениями между сервисами с минимальной задержкой. Архитектура очередей предоставляет гибкость в маршрутизации, однако масштабируемость ограничена: рост нагрузки требует горизонтального масштабирования через кластеры или шардирование. Кроме того, внутренняя реализация на языке Erlang усложняет поддержку и расширение системы в контексте современных DevOps-практик.

Исходя из краткого описания, информации в документации по всем трем технологиям на данный момент, а также на основании технического сравнения компании Confluent [25] можно сделать некоторые выводы.

Масштабируемость. Apache Kafka демонстрирует высокую линейную масштабируемость за счёт партиционирования топиков и независимого увеличения числа брокеров. Pulsar также масштабируется эффективно, но требует интеграции с BookKeeper, что усложняет развёртывание. RabbitMQ, в силу архитектурных ограничений, слабо масштабируется в условиях высокой нагрузки и не предназначен для интенсивной обработки потоков.

Производительность. При последовательной передаче больших объёмов данных Kafka, как правило, сохраняет стабильную скорость отклика. Этому способствует возможность предварительной упаковки сообщений и уменьшения их размера - за счёт этого нагрузка на сеть становится менее выраженной. Pulsar демонстрирует схожие характеристики, но может уступать в задержках при сложной топологии хранения. RabbitMQ показывает хорошие результаты в сценариях с небольшими объёмами и высокой плотностью соединений, однако теряет эффективность при потоковой нагрузке.

Надежность. Все три системы поддерживают различные уровни репликации. Kafka и Pulsar предоставляют расширенные механизмы управления отказами и обеспечивают семантику доставки exactly-once (при соответствующей конфигурации). В случае RabbitMQ возможность долговременного хранения предусмотрена, однако при возникновении сбоев в подсистемах записи поведение оказывается менее предсказуемым. При значительном объёме сообщений эта особенность становится особенно заметной.

Зрелость и поддержка сообщества. Apache Kafka обладает наиболее зрелой экосистемой и широкой поддержкой в индустрии. Pulsar активно развивается, но пока уступает Kafka в распространённости и зрелости документации. RabbitMQ остаётся надёжным решением для классических очередей, однако не соответствует требованиям стриминговой аналитики.

С учётом изложенных факторов, Apache Kafka представляет собой наиболее сбалансированное решение для задач потоковой обработки и обнаружения событий в реальном времени. Система давно используется в прикладных условиях и зарекомендовала себя в сложных конфигурациях. В некоторых вариантах применения, где структура хранения требует большей гибкости, допустимо использование Pulsar. Однако такая необходимость встречается реже и обычно связана с особенностями архитектуры окружения. RabbitMQ более уместен в сервисных взаимодействиях с низкой задержкой, но не подходит для высоконагруженных стриминговых решений.

|Фреймворк потоковой обработки

В современных системах реального времени важно правильно выбрать фреймворк потоковой обработки данных для своевременного обнаружения событий. Различные платформы отличаются по возможностям, производительности и требованиям к инфраструктуре, поэтому их сравнительный анализ позволит обоснованно определить оптимальное решение для реализации упомянутых требований.

Для объективного сравнения были определены ключевые критерии: модель обработки данных (непрерывный поток событий или микро-пакетная обработка), уровень задержки (latency) и пропускной способности, гарантии доставки сообщений (семантика at-least-once или exactly-once), поддержка сохранения состояния и оконных операций, интеграция с источниками/приемниками данных, а также сложность внедрения и использования (включая требуемую инфраструктуру и доступные языки разработки).

В данном разделе будут рассмотрены пять самых популярных open-source инструментов для потоковой обработки: KafkaStreams [26], ksqlDB [27], Apache Flink [28], Apache Storm [29] и Spark Streaming [30]. Технологии Kafka Streams и ksqlDB относятся к экосистеме Apache Kafka и работают только с потоками сообщений Kafka. Apache Flink, Apache Storm и Spark Streaming представляют собой автономные распределенные фреймворки со своими собственными

механизмами выполнения. Подход к обработке данных у этих решений различается. Kafka Streams и ksqlDB реализуют модель непрерывного потокового вычисления, обрабатывая каждое событие по мере поступления (event-at-a-time) с минимальной задержкой (порядка миллисекунд). Аналогично, Flink и Storm также работают с непрерывным потоком (истинная стриминговая обработка), тогда как Spark Structured Streaming применяет принцип микро-пакетной обработки: он разбивает входной поток на небольшие пакеты (например, с интервалом ~1 секунда) для микро-пакетной обработки. Этот микро-пакетный подход упрощает обеспечение отказоустойчивости, но вносит дополнительную задержку – латентность у Spark оказывается на порядок выше по сравнению с фреймворками, работающими с каждым событием напрямую. Некоторые системы могут поддерживать оба режима: например, Flink допускает как потоковую, так и пакетную обработку, а Storm через высокоуровневый API Trident способен выполнять микробатч-обработку для реализации транзакционных семантик. Инструменты Kafka (Kafka Streams и ksqlDB), напротив, строго ориентированы на непрерывный поток без батчирования, что соответствует требованию минимальной задержки.

Возможности по управлению состоянием и поддержке сложных событий (например, агрегатов и окон во временных промежутках) также влияют на пригодность фреймворка для детекции событий. Kafka Streams предлагает встроенные механизмы сохранения состояния (в виде локальных хранилищ состояния, State Stores, с сохранением на диск через RocksDB [31]) и позволяет задавать оконные операции (скользящие, сессионные и т. д.) для обработки событий за заданные интервалы времени. ksqlDB, используя тот же механизм внутри, предоставляя высокоуровневый SQL-интерфейс для определения агрегатов и оконных запросов, что упрощает реализацию типовых паттернов обнаружения без написания кода на общем языке программирования. Apache Flink славится расширенной поддержкой состояния: он способен сохранять большой объём промежуточных данных (ключево-значное состояние, табличное состояние) с периодическим сохранением снапшотов на постоянное

хранилище, что позволяет реализовывать сложные сценарии обработки (вплоть до сложных CEP-паттернов) с учётом времени события (event time) и механизмом вотермарок для корректной работы с опаздывающими сообщениями.

Spark Streaming тоже поддерживает сохранение состояния и окна: в модели есть механизм поддержания состояния между микропакетами (например, для агрегирования событий по ключу в серии), а также различные типы окон по времени (скользящие, сессионные окна). Apache Storm изначально разрабатывался как статическая (stateless) система, не сохраняющая состояние между событиями, поэтому реализация оконных вычислений или длительных агрегатов в Storm требует либо ручного обращения к внешним хранилищам, либо использования Trident API, который добавляет надстройку с поддержкой состояния. Таким образом, по критерию поддержки состояния и окон наиболее богатыми возможностями обладают Flink, Kafka Streams/ksqlDB и Spark, тогда как Storm без дополнительных надстроек уступает в гибкости и удобстве при реализации сложных сценариев обработки.

Различаются платформы и по интеграции с внешними источниками данных, а также по требованиям к инфраструктуре. Flink, Spark и Storm изначально спроектированы как универсальные движки обработки, поэтому для них разработано множество коннекторов к разным системам: очередям сообщений, файловым хранилищам, базам данных и т. п. Это делает их весьма гибкими в составе сложных конвейеров данных, но подразумевает разворачивание отдельного кластера вычислений. Например, Flink требует запуск собственного кластера (процесс-менеджер JobManager и набор TaskManagers) либо интеграции с менеджером ресурсов (Kubernetes [32]); Spark Streaming выполняется на кластере Apache Spark (YARN [33] или Kubernetes), а Storm разворачивает топологии на своем кластере с управляющим узлом Nimbus и рабочими узлами Supervisors. KafkaStreams же не является автономной платформой, а поставляется как библиотека, встраиваемая в приложение: потоковая задача запускается как часть обычного JVM-сервиса,

используя инфраструктуру Kafka для обмена сообщениями и хранения состояния. Это значительно упрощает эксплуатацию, если система уже использует Kafka: нет необходимости поддерживать отдельный кластер потоковой обработки, поскольку вычисления выполняются на стороне клиентских приложений. ksqlDB тоже не требует тяжелого кластера Big Data – он запускается как сервис (один или несколько узлов ksqlDB) поверх Kafka, потребляя сообщения из топиков и записывая результаты обратно в Kafka. По сути, ksqlDB-серверы масштабируются горизонтально подобно группе потребителей Kafka. Однако и Kafka Streams, и ksqlDB видят только данные, проходящие через брокера Kafka. Это ограничение не создаёт проблем только в том случае, когда Kafka является ядром архитектуры передачи и хранения данных.

Если рассматривать вопрос удобства разработки, то различия между инструментами во многом связаны с их ориентиром на разные категории пользователей. Kafka Streams преимущественно рассчитан на разработчиков, использующих JVM подобные языки [34], предоставляя доступ к функциональному API и специализированным конструкциям для описания потоковых преобразований. ksqlDB ориентирован на аналитиков и специалистов по данным, предлагая декларативный подход на основе SQL-запросов, что позволяет работать с потоками без необходимости писать код. Flink и Spark предлагают несколько уровней API (нижнего уровня и высокоуровневые DSL) на языках Java/Scala, а также поддерживают язык Python (PyFlink, PySpark) и собственные SQL-диалекты (Flink SQL, Spark SQL) для описания вычислительных конвейеров, что расширяет круг пользователей. Storm изначально реализован на Java, но благодаря использованию протокола Apache Thrift позволяет создавать болты и spouts практически на любом языке (существуют многоязычные интерфейсы для Python, Ruby, и др., хотя на практике чаще всего используются Java или Python).

Ниже в «таблице 2» сведены основные характеристики рассматриваемых решений.

Таблица 2 – Сводная таблица характеристик технологий

Решение	Kafka Streams	ksqlDB	Apache Flink	Apache Storm	Spark Streaming
Подход к обработке	Непрерывный поток; низкая задержка	Непрерывный поток, низкая задержка	Потоковая и пакетная; задержка зависит от режима	Потоковая, возможна микробатчевая обработка	Микробатчевая обработка; высокая задержка
Гарантия доставки	Exactly-once	Exactly-once	Exactly-once	At-least-once	Exactly-once
Состояние и окна	Встроенные хранилища, различные окна	SQL-окна, агрегаты	Масштабируемое состояние, гибкие окна	Ограничено;	Состояние между батчами, временные окна
Источник и данных	Только Kafka	Только Kafka	Kafka, файловые системы, БД и др.	Kafka, HTTP и др.	Kafka, файлы, сокет и др.

Развёртывание и язык	Библиотека в Java/Scala приложениях	Сервис поверх Kafka, SQL-запросы	Кластер; Java/Scala, Python, SQL	Кластер; Java, Python и др.	Кластер Spark; Scala, Java, Python, SQL
----------------------	-------------------------------------	----------------------------------	----------------------------------	-----------------------------	---

По итогам сравнительного анализа было принято решение в пользу использования ksqlDB как основной платформы потоковой обработки. Данный инструмент обеспечивает необходимую семантику доставки exactly-once, сохраняет устойчивость при высокой нагрузке и позволяет строить полноценную логику обнаружения событий без необходимости вмешательства в исходный код. Ключевым аргументом в его пользу стала декларативная модель конфигурации потоков: формирование сети обработки возможно через SQL-подобные запросы, без написания пользовательской логики. Это особенно важно в контексте частых изменений или масштабирования сценариев обработки. Кроме того, ksqlDB допускает горизонтальное масштабирование на уровне потоков и таблиц, что делает его более гибким решением по сравнению с Kafka Streams. При этом обе технологии органично встроены в архитектуру Apache Kafka, которая служит в данной системе в качестве шины передачи событий.

Язык и каркас системы

Построение системы потоковой обработки, в основе которой лежит Apache Kafka, предполагает выбор технологического стека, максимально согласованного с особенностями функционирования данной платформы. Поскольку Kafka и большая часть сопутствующих инструментов разработаны на основе JVM, выбор языка программирования был обусловлен необходимостью непосредственной и устойчивой интеграции с её внутренними механизмами. Наиболее очевидным и обоснованным решением в этом контексте стал язык Java, сочетающий зрелость экосистемы, широкую поддержку и наличие богатого набора библиотек для взаимодействия с Kafka. Кроме того, опыт автора разработки на Java позволяет избежать затрат на

адаптацию к новым языковым конструкциям и сосредоточить внимание на логике обработки событий.

Отдельное значение приобретает выбор архитектурного каркаса, так как структура системы должна оставаться слабо связанной и пригодной для последующего сопровождения. В рамках JVM-экосистемы наибольшее распространение получил фреймворк Spring Boot, позволяющий организовать модульную архитектуру с опорой на принципы инверсии управления и внедрения зависимостей. Эти принципы способствуют отделению компонентов и снижению когезии между ними, что положительно сказывается на сопровождаемости и устойчивости системы. Дополнительным фактором в пользу Spring Boot выступает поддержка аспектно-ориентированного программирования. Данный подход позволяет выносить повторяющиеся технические аспекты (например, логирование, обработку исключений, аудит) в отдельные слои, не затрагивая бизнес-логику. Такая организация обеспечивает чистоту кода и облегчает его тестирование.

В рамках реализации текущей архитектуры особое внимание уделяется упрощению взаимодействия с брокером сообщений без необходимости прямого обращения к его низкоуровневому API. С этой целью используется модуль spring-kafka, обеспечивающий слой абстракции, за счёт которого можно задать логику потребления и публикации событий через декларативные конструкции. Механизмы конфигурации, основанные на аннотациях, позволяют описывать основные компоненты системы в явной форме, при этом большая часть служебных операций, таких как сериализация, управление жизненным циклом соединений и распределение нагрузки между экземплярами потребителей, выполняется автоматически.

За счёт применения такого подхода удаётся избежать прямой зависимости от внутренних реализаций Kafka, сохраняя гибкость и контроль над ключевыми параметрами работы. стек на базе Java и Spring Boot предоставляет необходимый баланс между контролируемостью, читаемостью кода и способностью к масштабированию, что особенно важно при проектировании

микросервисной архитектуры, работающей в условиях переменной нагрузки и высокой доступности.

|Разработка программных средств

Для реализации взаимодействия с системой обработки событий был спроектирован REST-совместимый API, представляющий собой обёртку над функциональностью ksqlDB. Архитектура сервиса строится на основе фреймворка Spring Boot, где каждый из представленных эндпоинтов реализует отдельную часть логики управления объектами и выполнения запросов в ksqlDB.

Сервис разделён на два логических уровня: слой определения объектов и слой исполнения запросов. Первый отвечает за создание и модификацию логических сущностей в ksqlDB, таких как потоки и таблицы. Второй предоставляет интерфейс для выполнения pull/push-запросов [35] к объектам первого слоя, валидации этих запросов, управления активными сессиями. Каждый из методов работает с ksqlDB через внутренний SQL-коннектор, преобразуя HTTP-запросы в соответствующие SQL-подобные операции. Такие действия позволяют централизованно управлять схемой потоковой обработки.

Push-запросы реализованы на базе механизма `StreamingResponseBody` [36], что обеспечивает потоковую передачу данных через `Server-Sent Events` [37]. Данный режим позволяет подписаться на непрерывный вывод результата запроса и использовать сервис как мост между системами-источниками и потребителями событий.

Точка доступа по проверке запросов играет ключевую роль в предварительной валидации. Он предотвращает выполнение некорректных инструкций и минимизирует ошибки на этапе компоновки схемы.

Общая архитектура API спроектирована с учётом расширяемости и соответствия требованиям к управлению поточными вычислениями в условиях задекларированной архитектуры. Описанный интерфейс обеспечивает программное создание потоковых конвейеров и формирует основу для

конфигурируемой системы без необходимости ручного вмешательства в кодовую базу для настройки паттернов обработки потока событий.

Описание API представлено в «таблице 3» и «таблице 4».

Таблица 3 – Описание API слоя определения объектов

Метод	Точка доступа	Назначение
POST	/definition/stream	Создание нового потока (STREAM)
Метод	Точка доступа	Назначение

Продолжение таблицы 2

Метод	Точка доступа	Назначение
POST	/definition/table	Создание новой таблицы (TABLE)
PUT	/definition/stream/{name}	Обновление параметров уже существующего потока
PUT	/definition/table/{name}	Изменение структуры таблицы по её имени
DELETE	/definition/stream/{name}	Удаление потока и связанных данных
DELETE	/definition/table/{name}	Удаление таблицы и связанных данных

Таблица 4 – Описание API слоя исполнения запросов

Метод	Точка доступа	Назначение
POST	/execution/query/pull	Выполнение одиночного SELECT-запроса к TABLE с возвратом результата
POST	/execution/query/push	Установка непрерывной подписки (EMIT CHANGES)

		через SSE
POST	/execution/query/validate	Проверка корректности SQL-запроса без его исполнения (parseOnly)
GET	/execution/query/active	Получение информации о текущих активных запросах (аналог SHOW QUERIES)
DELETE	/execution/query/{id}	Прекращение активного запроса по его идентификатору (аналог TERMINATE)

Кодовая база текущего API и всех его слоев представлена в приложении А. Для упрощения восприятия и сокращения объема кода некоторые участки были представлены не полностью.

Тестированию ПО будет посвящен следующий раздел.

4 Тестирование программных средств

Данный раздел будет обозревать тестирование разработанной системы, используя эфемерную модель событий и созданную для нее топологию процесса обработки, с целью показать результат и применимость выбранных решений на практике

Представление модели данных тестирования

Модель данных представлена на ER-диаграмме логического уровня рисунка 3:

- метаданные атомарного события (Atomic Event Metadata),
- метаданные комплексного события (Complex Event Metadata),
- атомарное событие (Atomic Event),
- комплексное событие (Complex Event).

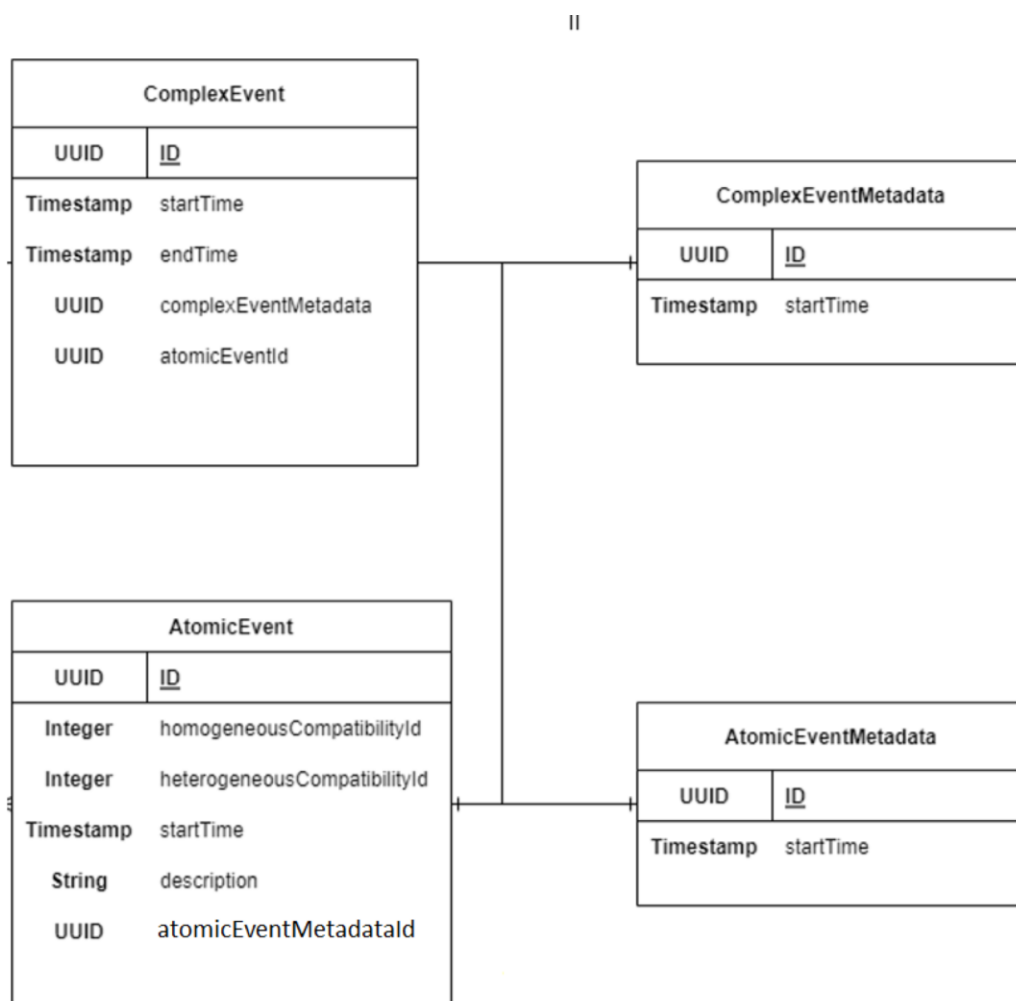


Рисунок 3 – Логическая модель данных сценария обработки событий

Представленная модель будет использоваться в качестве потока событий при реализации топологии обработки проверки архитектурных и технологических решений.

|Сценарий обработки событий

Сценарий обработки события будет представлен в виде топологии процесса. Топология процесса – это способ представления потока данных в виде ориентированного ациклического графа (DAG), что является одним из методов парадигмы программирования потоков данных [38]. Основным сценари¹ обработки событий будет включать в себя обогащение входящих событий Atomic Event. Дальнейшая обработка базируется на слиянии объектов Atomic Event в событие Complex Event.

На рисунках далее представлены графы, которые являются реализацией разрабатываемого процесса, где данные проходят через набор узлов обработчиков, а соединяющие их линии представляют собой потоки входных и выходных данных. Представленные графы являются частями общей топологии, но для удобства представления будут разделены на суб-топологии по типам обрабатываемых событий и производимым операциям.

Процесс обработки Atomic Event представляет из себя соединения событий из двух тем (логическая единица хранения сообщений) Apache Kafka, одна из которых хранит события необогащенного типа Atomic Event Raw (промежуточная сущность), а вторая метаданные для обогащения первого – Atomic Event Metadata. После успешного соединения событий происходит отправка получившегося результата в тему событий Atomic Event. Подробности представлены на рисунке 4.

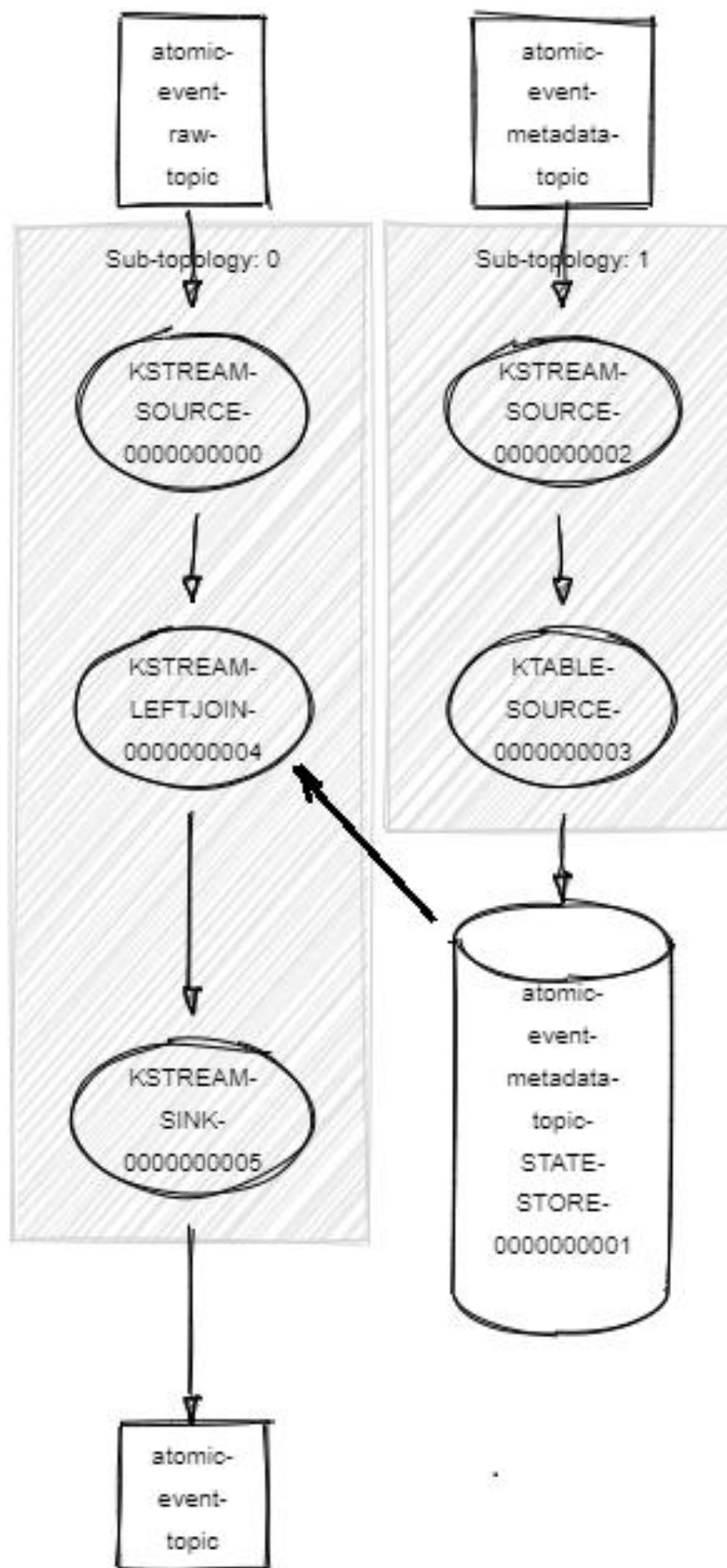


Рисунок 4 – Топология процесса обработки событий типа AtomicEvent

Топология процесса обработки события Complex Event включает в себя все предыдущие указанные топологии. Всего суб-топологий по обработке Complex Event будет три. Первая суб-топология аналогична обработке Atomic Event и, Вторая топология на рисунке 5 представляет из себя секционирование ключей для правильной работы. Третья суб-топология на рисунке 6 объединяет события Atomic Event за счет алгоритма временных окон.

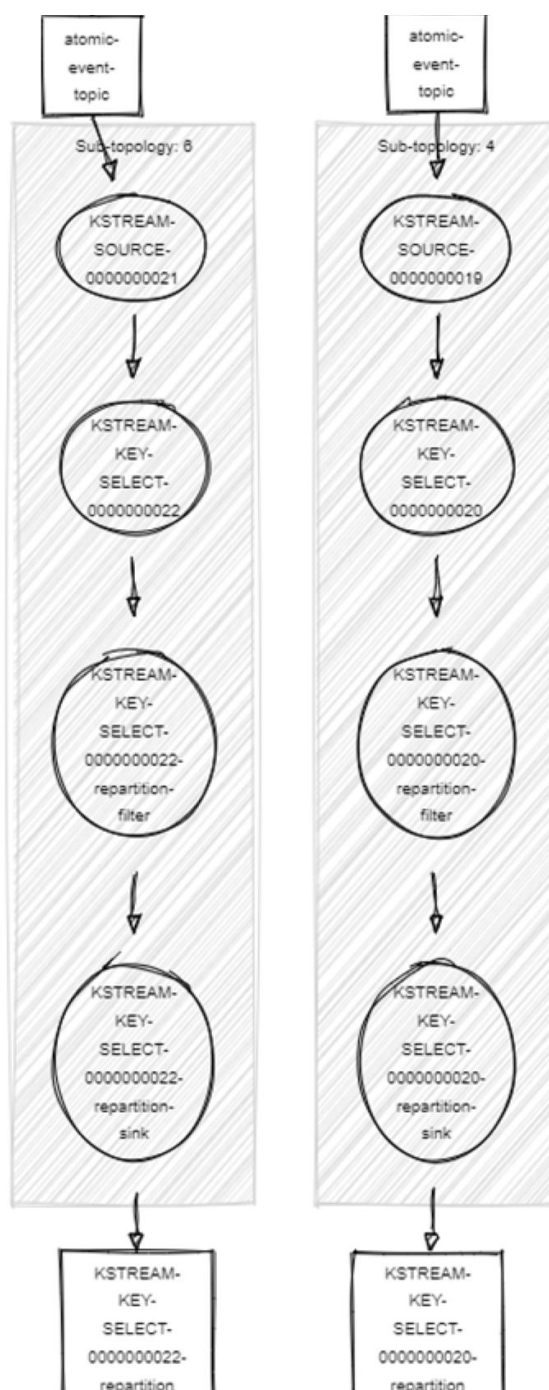


Рисунок 5 – Суб-топология процесса обработки Complex Event
(секционирование событий Atomic Event)

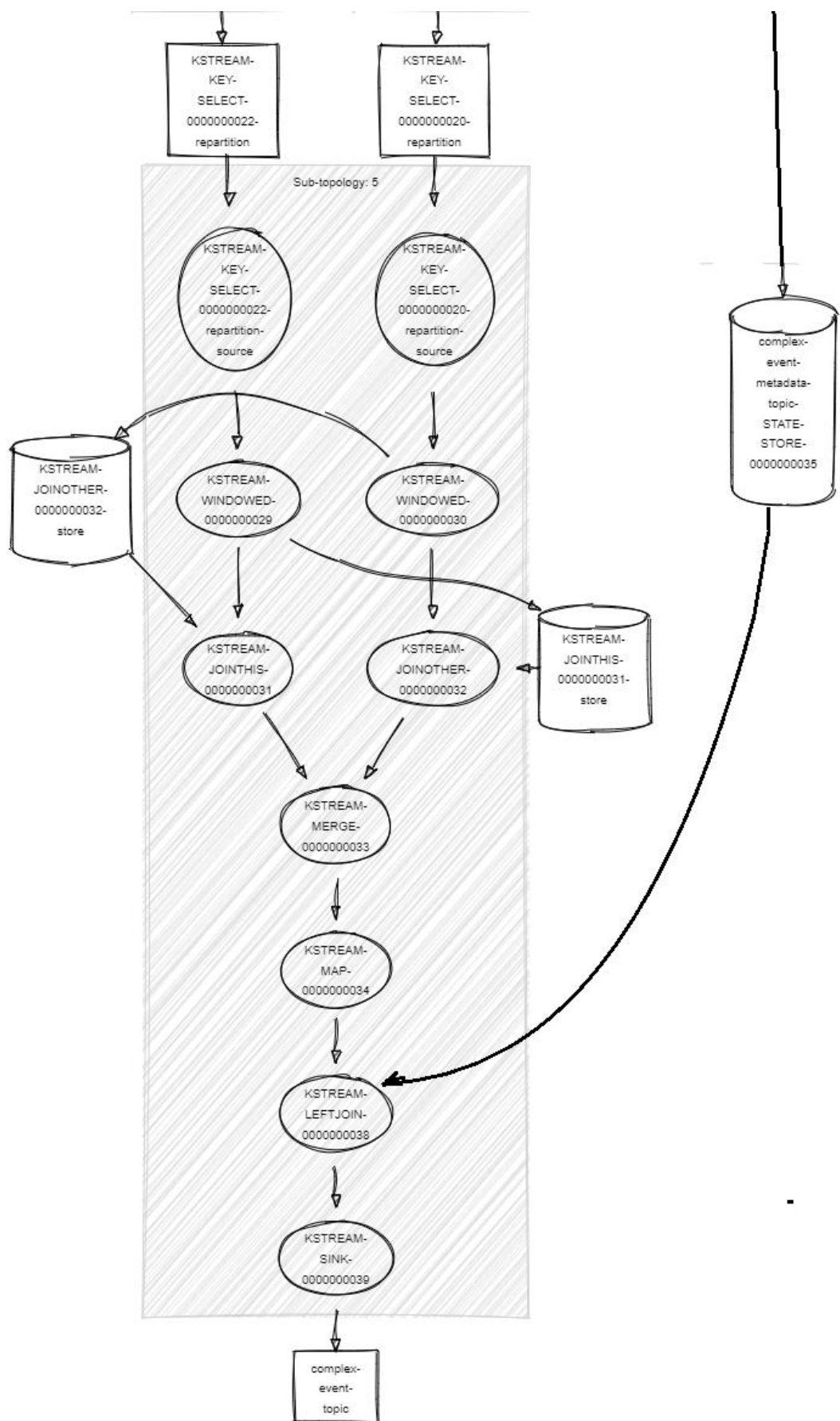


Рисунок 6 – Суб-топология процесса обработки Complex Event (соединение событий Atomic Event и последующее обогащение Complex)

Результаты тестирования

Результаты теста по обработке сценария первой суб-топологии представлены ниже. Пакет отправленных необогащенных событий Atomic Event Raw в брокер представлен на рисунке 7.

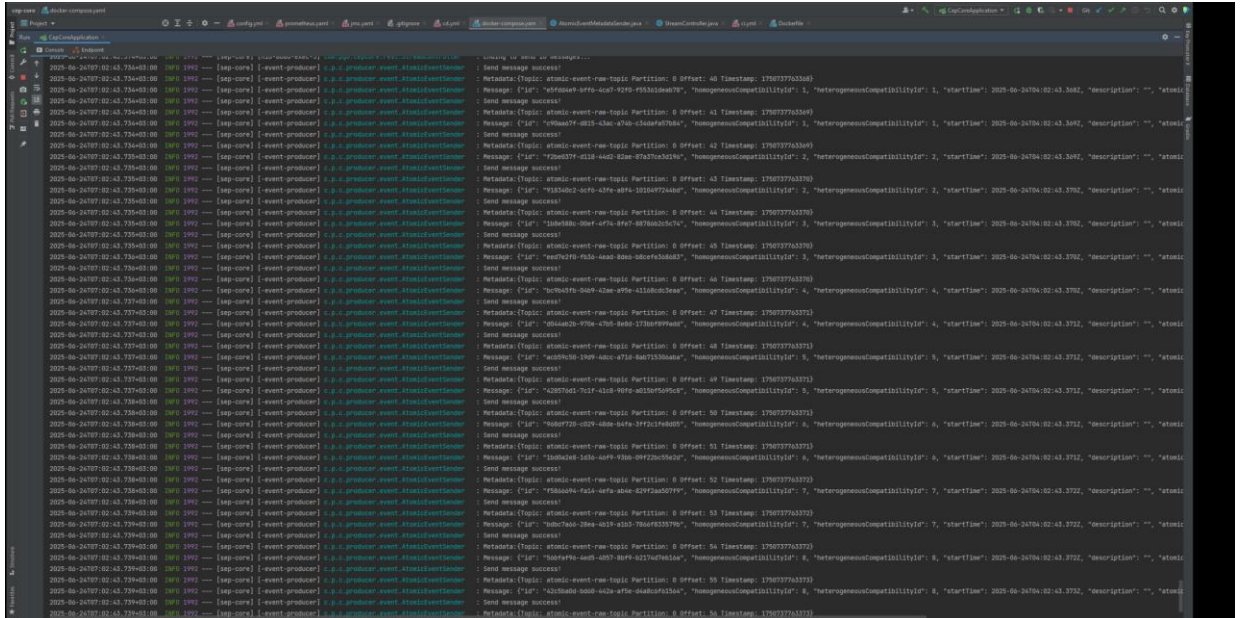


Рисунок 7 – Необогащенные события Atomic Event Raw

Отправленные в брокер следом метаданные Atomic Metadata Event для обогащения представлены на рисунке 8.

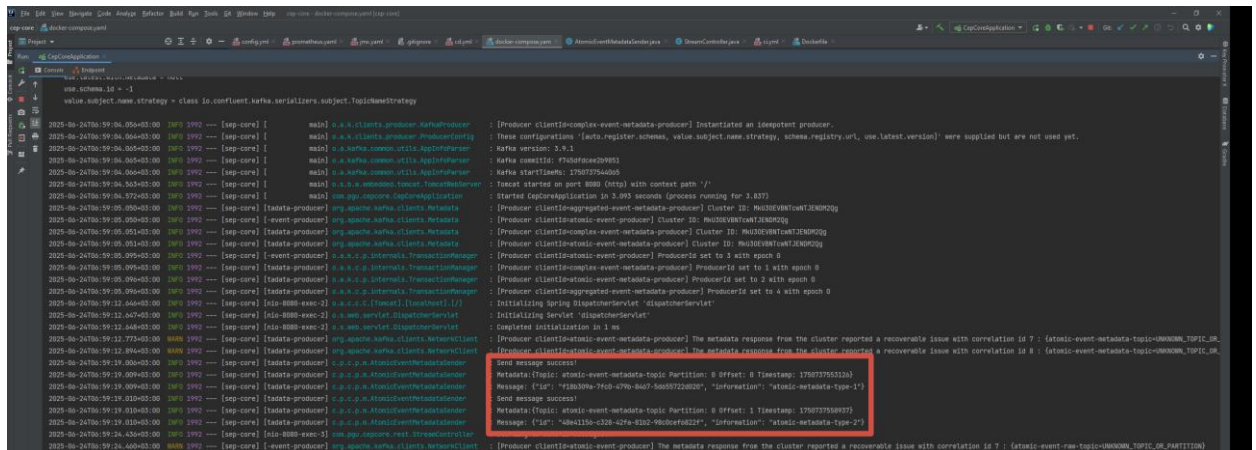


Рисунок 8 – Метаданные Atomic Metadata Event

После формирования потока в ksqlDB через скрипт, отправленный по реализованному API, были получены результаты, которые представлены на рисунке 9.

ID	HOMOGENEOUSCOMPATIBILITYID	HETEROGENEOUSCOMPATIBILITYID	STARTTIME	DESCRIPTION	ATOMICEVENTMETADATAID	METADATAINFORMATION
e5fdd4e9-bff6-4ca71-92f0-f55361deab78	1	1	1750737763368		f18b309a-7fc0-479b-8467-5d655722d020	atomic-metadata-type-1
c90aa67f-d815-43ac-a74b-c34dafa57b84	1	1	1750737763369		48e41156-c328-42fa-81b2-98c0cef6822f	atomic-metadata-type-2
f2be037f-d118-44d2-82ae-87a37ce3d196	2	2	1750737763369		f18b309a-7fc0-479b-8467-5d655722d020	atomic-metadata-type-1
918340c2-6cf6-43fe-a8f4-1010497244bd	2	2	1750737763370		48e41156-c328-42fa-81b2-98c0cef6822f	atomic-metadata-type-2
1b8e588c-00ef-4f74-8fe7-88786b2c5c74	3	3	1750737763370		f18b309a-7fc0-479b-8467-5d655722d020	atomic-metadata-type-1
eed7e2f0-fb36-4ead-8de6-b8cefe368683	3	3	1750737763370		48e41156-c328-42fa-81b2-98c0cef6822f	atomic-metadata-type-2
bc9b45fb-04b9-42ae-a95e-41168cdc3eaa	4	4	1750737763370		f18b309a-7fc0-479b-8467-5d655722d020	atomic-metadata-type-1
d044ab2b-970e-47b5-8e8d-173bbf899add	4	4	1750737763371		48e41156-c328-42fa-81b2-98c0cef6822f	atomic-metadata-type-2
acb59c50-19d9-4dcc-a71d-8ab715306aba	5	5	1750737763371		f18b309a-7fc0-479b-8467-5d655722d020	atomic-metadata-type-1
428576d1-7c1f-41c8-90fd-a015bf5695c8	5	5	1750737763371		48e41156-c328-42fa-81b2-98c0cef6822f	atomic-metadata-type-2
968df720-c029-48de-b4fa-3ff2c1fe8d05	6	6	1750737763371		f18b309a-7fc0-479b-8467-5d655722d020	atomic-metadata-type-1
1bd0a2e8-1d36-46f9-93bb-09f22bc55e2d	6	6	1750737763371		48e41156-c328-42fa-81b2-98c0cef6822f	atomic-metadata-type-2
f5866694-fa14-4efa-ab4e-829f2aa507f9	7	7	1750737763372		f18b309a-7fc0-479b-8467-5d655722d020	atomic-metadata-type-1
bdbc7a66-28ea-4b19-a1b3-7866f833579b	7	7	1750737763372		48e41156-c328-42fa-81b2-98c0cef6822f	atomic-metadata-type-2
56bfaf96-4ed5-4057-8bf9-b2174d7eb16a	8	8	1750737763372		f18b309a-7fc0-479b-8467-5d655722d020	atomic-metadata-type-1
42c5ba0d-bd60-442a-af5e-d4a8c6f61564	8	8	1750737763373		48e41156-c328-42fa-81b2-98c0cef6822f	atomic-metadata-type-2
261d9068-1e88-49d3-a3eb-eea93c65486e	9	9	1750737763373		f18b309a-7fc0-479b-8467-5d655722d020	atomic-metadata-type-1
4b317240-ddcf-4e32-99d0-860d2912a0af	9	9	1750737763373		48e41156-c328-42fa-81b2-98c0cef6822f	atomic-metadata-type-2
43c37aa6-ab64-4e72-bf6b-d4705851cb04	10	10	1750737763373		f18b309a-7fc0-479b-8467-5d655722d020	atomic-metadata-type-1
848934d7-bfe1-484a-9196-5320bc348e98	10	10	1750737763374		48e41156-c328-42fa-81b2-98c0cef6822f	atomic-metadata-type-2

Рисунок 9 – Результат обогащения данными события Atomic Event

Как можно наблюдать, слияние двух объектов произошло успешно. Используемые скрипты представлен в листинга 1-3.

Листинг 1 – Скрипт создания сущности таблицы для метаданных Atomic Event.

```
CREATE TABLE atomic_metadata_table (
  id STRING PRIMARY KEY,
  information STRING
) WITH (
  KAFKA_TOPIC = 'atomic-event-metadata-topic',
  VALUE_FORMAT = 'AVRO',
  PARTITIONS = 1
);
```

Листинг 2 – Скрипт создания сущности потока для Atomic Event

```
CREATE STREAM atomic_raw_stream (
  id STRING KEY,
  homogeneousCompatibilityId INT,
  heterogeneousCompatibilityId INT,
  startTime BIGINT,
  description STRING,
  atomicEventMetadataId STRING
```



```

) WITH (
  KAFKA_TOPIC = 'atomic-event-raw-topic',
  VALUE_FORMAT = 'AVRO',
  PARTITIONS = 1
);

```

Листинг 3 – Скрипт объединения Atomic Event с метаданными.

```

CREATE STREAM atomic_event_stream WITH (
  KAFKA_TOPIC = 'atomic-event-topic',
  VALUE_FORMAT = 'AVRO',
  PARTITIONS = 1
) AS
SELECT
  r.id AS id,
  r.homogeneousCompatibilityId AS homogeneousCompatibilityId,
  r.heterogeneousCompatibilityId AS heterogeneousCompatibilityId,
  r.startTime AS startTime,
  r.description AS description,
  r.atomicEventMetadataId AS atomicEventMetadataId,
  m.information AS metadataInformation
FROM atomic_raw_stream r
LEFT JOIN atomic_metadata_table m
  ON r.atomicEventMetadataId = m.id
EMIT CHANGES;

```

Далее идет переход к субтопологии объединения Atomic Event по ключу гетерогенности в Complex Event.

Скрипт, позволяющий реализовать данную функциональность обработки, представлен в листинге 4.

Листинг 4 – Скрипт оконного объединения событий Atomic Event.

```

CREATE STREAM atomic_event_joined_windowed WITH (
  KAFKA_TOPIC = 'atomic-event-windowed-topic',
  VALUE_FORMAT = 'AVRO',
  PARTITIONS = 1
) AS
SELECT
  a.id AS id_a,
  b.id AS id_b,
  a.heterogeneousCompatibilityId AS compatibilityId,
  a.startTime AS start_a,
  b.startTime AS start_b,
  a.description AS description_a,
  b.description AS description_b
FROM atomic_raw_stream a
JOIN atomic_event_stream b
  WITHIN 10 SECONDS

```

```

ON a.heterogeneousCompatibilityId =
b.heterogeneousCompatibilityId
WHERE a.id <> b.id
EMIT CHANGES;

```

Результат работы представлен на рисунке 10.

COMPATIBILITYID	ID_A	ID_B	START_A	START_B	DESCRIPTION_A	DESCRIPTION_B
1	f525ce9f-1051-4f25-a378	f991eec5-0823-42c6-b377	1750739515637	1750739515637		
1	-7b2dcc051674	-e108aa3b041e				
1	f991eec5-0823-42c6-b377	f525ce9f-1051-4f25-a378	1750739515637	1750739515637		
1	-e108aa3b041e	-7b2dcc051674				
2	2ef6c95c-890c-4bfd-b4f3	43322095-4ee6-41b2-8263	1750739515637	1750739515637		
2	-aacda3060d09	-84dbc80e6495				
2	43322095-4ee6-41b2-8263	2ef6c95c-890c-4bfd-b4f3	1750739515637	1750739515637		
2	-84dbc80e6495	-aacda3060d09				
3	4e270791-ec73-4cc5-af06	42ac5d21-a63e-405e-a02e	1750739515638	1750739515637		
3	-e9baf3cd2169	-f6439fb688a3				
3	42ac5d21-a63e-405e-a02e	4e270791-ec73-4cc5-af06	1750739515637	1750739515638		
3	-f6439fb688a3	-e9baf3cd2169				
4	45e4297d-41a2-4013-b540	a2db741a-8753-47d6-b098	1750739515638	1750739515638		
4	-606027b85cb4	-743c381d6a5c				
4	a2db741a-8753-47d6-b098	45e4297d-41a2-4013-b540	1750739515638	1750739515638		
4	-743c381d6a5c	-606027b85cb4				
5	63a4d701-bcfc-4c86-824d	1dfb06f0-e8e5-4cbe-b76c	1750739515639	1750739515638		
5	-21d2ac775e11	-3ed3b5fa87c6				
5	1dfb06f0-e8e5-4cbe-b76c	63a4d701-bcfc-4c86-824d	1750739515638	1750739515639		
5	-3ed3b5fa87c6	-21d2ac775e11				
6	5f449ec5-2162-40ff-9357	d2b34266-2aa9-4d40-8db7	1750739515639	1750739515639		
6	-1a5f234eb416	-b733919027f8				
6	d2b34266-2aa9-4d40-8db7	5f449ec5-2162-40ff-9357	1750739515639	1750739515639		
6	-b733919027f8	-1a5f234eb416				
7	fd60b8d4-8718-42c0-beb5	74ef0307-c366-4ecc-a10e	1750739515640	1750739515639		
7	-28f05cf033e2	-deb90218f9af				
7	74ef0307-c366-4ecc-a10e	fd60b8d4-8718-42c0-beb5	1750739515639	1750739515640		
7	-deb90218f9af	-28f05cf033e2				
8	b4ddf454-545d-4389-a157	78487625-bb8d-457f-b6cb	1750739515640	1750739515640		
8	-93537c094d3f	-bb0a2a7bd128				
8	78487625-bb8d-457f-b6cb	b4ddf454-545d-4389-a157	1750739515640	1750739515640		
8	-bb0a2a7bd128	-93537c094d3f				
9	c3abd732-4cbc-4ef4-a385	70afca37-bffb-425c-a9c6	1750739515641	1750739515641		
9	-13d9e0c3e865	-c739f3ec6dbe				
9	70afca37-bffb-425c-a9c6	c3abd732-4cbc-4ef4-a385	1750739515641	1750739515641		
9	-c739f3ec6dbe	-13d9e0c3e865				
10	cbe64618-ad53-4ae9-89f8	3f241895-6fea-40bd-aca7	1750739515642	1750739515641		
10	-de8d7aa44f5b	-72ebf0573442				
10	3f241895-6fea-40bd-aca7	cbe64618-ad53-4ae9-89f8	1750739515641	1750739515642		
10	-72ebf0573442	-de8d7aa44f5b				

Рисунок 9 – Результат оконного объединения событий Atomic Event

Как можно заметить, события успешно объединились по ключу гетерогенности. Далее, на рисунках 10-12 представлена функциональность по отслеживанию метрик ПО.

The screenshot shows the Akno.io interface for managing Kafka topics. The 'Topics' section is active, displaying a table of topics. The table includes columns for Name, Count, Size, Last Record, Total, Factor, In Sync, and Consumer Groups. The topics listed are:

- atomic-event-metadata-topic: Count ≈ 2, Size 282 B, Last Record 21 minutes ago, Total 1, Factor 1, In Sync 1. Consumer Groups: confluent-ksql-default_query_CSAS_ATOMIC_EVENT_STREAM_5 (Lag 0).
- atomic-event-raw-topic: Count ≈ 40, Size 5.333 KB, Last Record 16 minutes ago, Total 1, Factor 1, In Sync 1. Consumer Groups: confluent-ksql-default_query_CSAS_ATOMIC_EVENT_JOINED_WINDOWED_7 (Lag 0) with 2 partitions.
- atomic-event-topic: Count ≈ 20, Size 2.561 KB, Last Record 16 minutes ago, Total 1, Factor 1, In Sync 1. Consumer Groups: confluent-ksql-default_query_CSAS_ATOMIC_EVENT_JOINED_WINDOWED_7 (Lag 0).
- atomic-event-windowed-topic: Count ≈ 20, Size 2.366 KB, Last Record 16 minutes ago, Total 1, Factor 1, In Sync 1. Consumer Groups: confluent-ksql-default_transient_transient_ATOMIC_EVENT_JOINED_WINDOWED_5108548441520082036_1750739502751 (Lag 0).
- default_ksql_processing_log: Count ≈ 0, Size 0 B, Last Record 16 minutes ago, Total 1, Factor 1, In Sync 1. Consumer Groups: none.

Рисунок 10 – Нагрузка на топики в байтах при тестировании

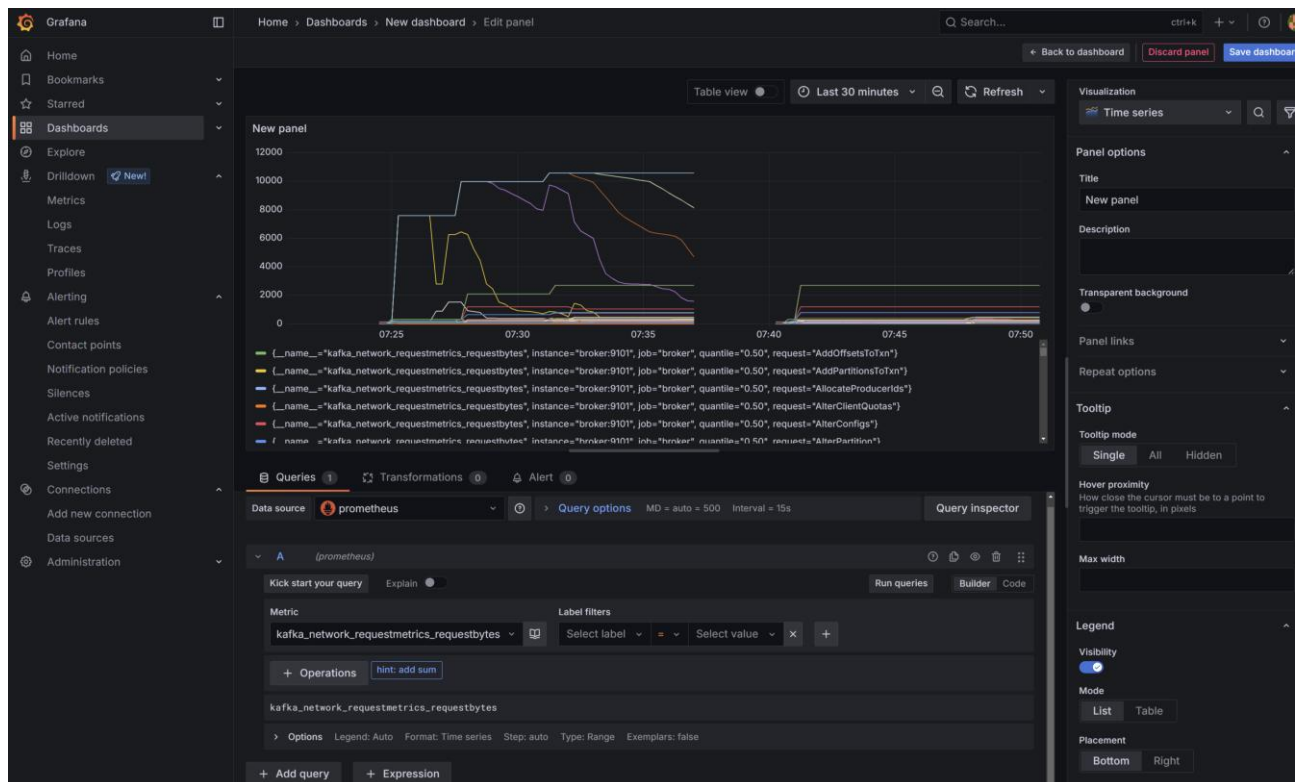


Рисунок 11 – Количество обработанных запросов брокером во время тестирования

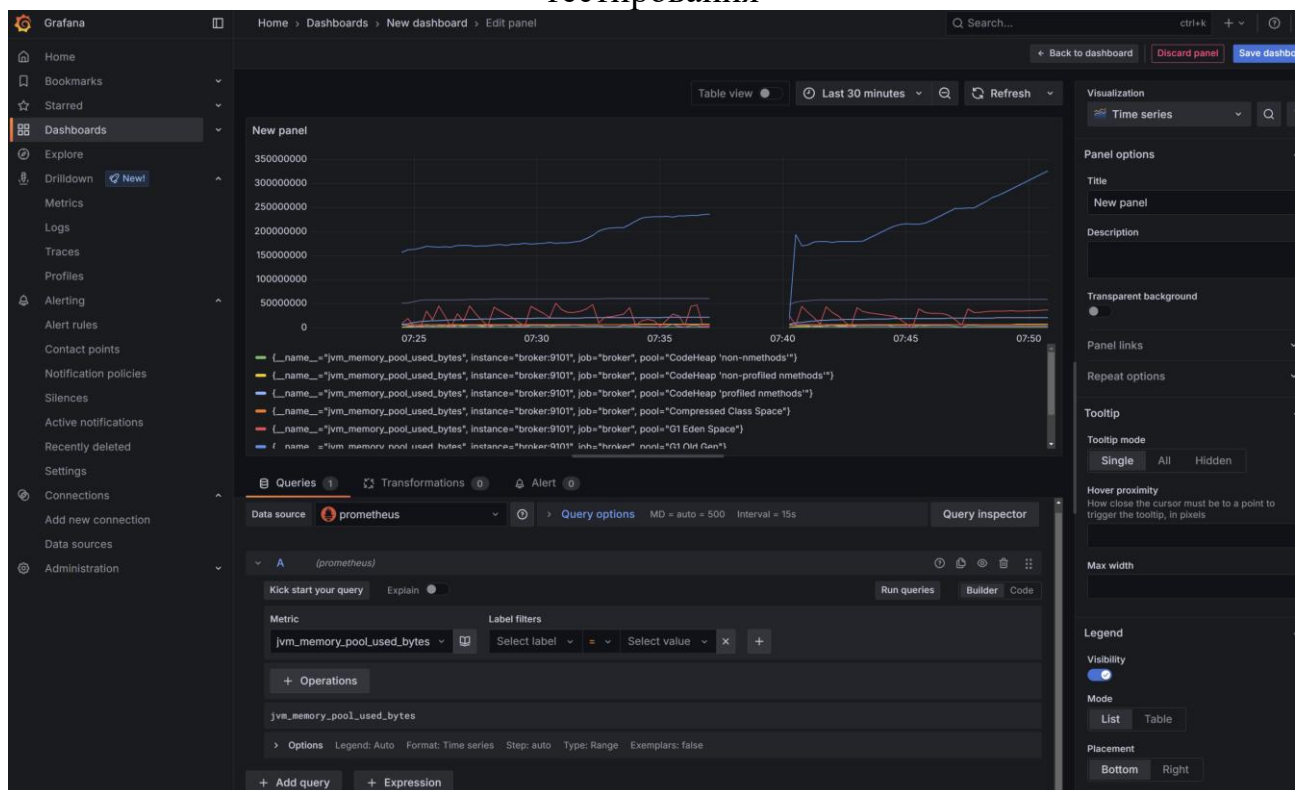


Рисунок 12 – Нагрузка JVM машины брокера

ЗАКЛЮЧЕНИЕ

В рамках настоящей работы была успешно решена задача проектирования и реализации системы оперативного обнаружения событий в больших потоках данных, соответствующей современным требованиям. Работа охватывала как теоретическую, так и практическую составляющие, обеспечивая целостность методологического и инженерного подхода.

На первом этапе были проанализированы ключевые концепции и подходы к обработке событий в потоке, включая формальные модели представления потоков, событий и оконных механизмов. На основе изученных источников была сформулирована модель событийной обработки, учитывающая семантику времени, состояние между событиями, оконные функции и шаблонное сопоставление. Предложенные в рамках раздела 2 формализации позволили задать чёткие условия для автоматического выявления сложных событий и были положены в основу проектных решений.

С практической стороны, был подобран и обоснован технологический стек, обеспечивающий реализацию всех перечисленных механизмов. В качестве шины передачи событий использована платформа Apache Kafka, а для организации логики потоковой обработки – ksqlDB, позволяющая на декларативном уровне формировать сеть обработки событий без необходимости модификации программного кода. Это решение обеспечило соответствие не только функциональным требованиям, но и критериям нефункциональным –общей отказоустойчивости. Благодаря встроенной поддержке семантики exactly-once, механизма replication и распределённого хранения состояния, система способна устойчиво функционировать в условиях высокой нагрузки и сетевых сбоев.

Таким образом, в ходе работы была создана система, способная в реальном времени выявлять значимые события в потоке данных с использованием теоретически обоснованных моделей, устойчиво работать в распределённой среде и адаптироваться к росту нагрузки без потери производительности. Результаты могут быть применены как в прикладных

системах мониторинга, так и в более широких сценариях аналитики потоков информации.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Елена Шпрингер, IoT для умных часов и IIoT для умных станков: что такое интернет вещей и каким он бывает // Журнал VK Cloud об IT-бизнесе, технологиях и цифровой трансформации [Электронный ресурс]. URL: <https://mcs.mail.ru/blog/iot-dlya-umnyh-chasov-iiot-umnyh-stankov-internet-veshchej> (дата обращения: 11.04.2025).
2. Sharma Chakravarthy, Qingchun Jiang, Stream Data Processing: A Quality of Service Perspective - Modeling, Scheduling, Load Shedding, and Complex Event Processing // ResearchGate [Электронный ресурс]. URL: https://www.researchgate.net/publication/220696388_Stream_Data_Processing_A_Quality_of_Service_Perspective__Modeling_Scheduling_Load_Shedding_and_Complex_Event_Processing (дата обращения: 16.04.2025).
3. Фишер Джош, Ван Нин, Грокам стриминг. – СПб.: Питер, 2023. – 288 с.: ил. – (Серия «Библиотека программиста»).
4. Large Scale Data Engineering 2016, Lecture Block 5: Data Streams: Storm, Spark Streaming and Google DataFlow [Электронный ресурс]. URL: <https://homepages.cwi.nl/~boncz/lse/2016/streaming.shtml#:~:text=A%20data%20stream%20is%20a,transactions%20and%20click%20stream%20analysis> (дата обращения: 16.04.2025).
5. The Unified Apache Beam Model // beam.apache.org [Электронный ресурс] URL: <https://beam.apache.org/> (дата обращения: 26.04.2025).
6. Streaming 101: The world beyond batch [Электронный ресурс]. URL: <https://www.oreilly.com/radar/the-world-beyond-batch-streaming-101/> (дата обращения: 27.04.2025).
7. Brian Babcock, Shivnath Babu, Mayur Datar, Rajeev Motwani, Jennifer Widom. Models and Issues in Data Stream Systems // cs.princeton.edu [Электронный ресурс] URL: <https://www.cs.princeton.edu/courses/archive/spr04/cos598B/bib/BabcockBDMW.pdf#:~:text=data%20items%20may%20be%20relational,some%20fundamentally%20new%20research%20problems> (дата обращения: 28.04.2025).

8. The CQL Continuous Query Language // Stanford [Электронный ресурс] URL: http://infolab.stanford.edu/stream/demo/cql_primer.html (дата обращения: 1.05.2025).
9. Articles by Arvind Arasu // scispace.com [Электронный ресурс]. URL: <https://scispace.com/authors/arvind-arasu-2o1oqmwqam> (дата обращения: 4.05.2025).
10. Ariane Ziehn¹, Philipp M. Grulich¹, Steffen Zeuch¹, Volker Mark. Bridging the Gap: Complex Event Processing on Stream Processing Systems // TU Berlin [Электронный ресурс]. URL: <https://openproceedings.org/2024/conf/edbt/paper-114.pdf> (дата обращения: 4.05.2025).
11. Конечный автомат // Wikipedia [Электронный ресурс]. URL: https://ru.wikipedia.org/wiki/Конечный_автомат (дата обращения: 5.05.2025).
12. Dr. Martin Kleppmann, Distributed Systems, University of Cambridge Computer Science Tripos, Part IB Michaelmas term 2021/22.
13. Chris Richardson. What are microservices? // Microservices [Электронный ресурс]. URL: <https://microservices.io/index.html> (дата обращения: 6.05.2025).
14. Крис Ричардсон, Микросервисы, Паттерны разработки и рефакторинга. – СПб.: Питер, 2019. – 544 с.: ил. — (Серия «Библиотека программиста»).
15. Ньюмен С. Создание микросервисов. СПб.: Питер, 2016. — 304 с.: ил. - (Серия «Бестселлеры O'Reilly»).
16. Martin Fowler. What do you mean by «Event-Driven»? // MartinFowler [Электронный ресурс]. URL: <https://martinfowler.com/articles/201701-event-driven.html> (дата обращения: 1.05.2023).
17. Стопфорд, Бен Проектирование событийно-ориентированных систем: Концепции и шаблоны проектирования сервисов потоковой обработки данных с использованием Apache Kafka / Бен Стопфорд ; Пер. с англ. – 2-е изд., испр. Иркутск : ITSumma Press, 2019. – 175 с.

18. Martin Fowler, Focusing on Events // MartinFowler [Электронный ресурс]. URL: <https://martinfowler.com/eaDev/EventNarrative.html> (дата обращения: 1.05.2023).
19. C4 model. The C4 model for visualising software architecture // C4 [Электронный ресурс]. URL: <https://c4model.com/> (дата обращения: 1.05.2023)
20. Шибанов С.В. Сервис потоковой обработки событий и исполнения активных правил / Шибанов С.В., Шлепнёв Я.С. // Математическое моделирование и суперкомпьютерные технологии. / Труды XXI Международной конференции (Н. Новгород, 22–26 ноября 2021 г.) / Под ред. проф. Д.В. Баландина. – Нижний Новгород: Изд-во Нижегородского госуниверситета, 2021. – С. 403-407.
21. Mike Ralphson. OpenAPI Specification // GitHub [Электронный ресурс]. URL: <https://github.com/OAI/OpenAPI-Specification> (дата обращения: 2.05.2023).
22. Apache Kafka, Kafka Intro // kafka.apache.org [Электронный ресурс]. URL: <https://kafka.apache.org/intro/> (дата обращения: 5.05.2023).
23. Apache Pulsar. Pulsar Docs // pulsar.apache.org [Электронный ресурс]. URL: <https://pulsar.apache.org/docs/3.0.x/> (дата обращения: 5.05.2023)
24. RabbitMQ. RabbitMQ Docs // pulsar.apache.org [Электронный ресурс]. URL: <https://www.rabbitmq.com/> (дата обращения: 5.05.2023).
25. Alok Nikhil, Vinoth Chair. Benchmarking Apache Kafka, Apache Pulsar, and RabbitMQ: Which is the Fastest? // Confluent [Электронный ресурс]. URL: <https://www.confluent.io/blog/kafka-fastest-messaging-system/#throughput-test> (дата обращения: 5.05.2023).
26. Kafka Streams, Spark Docs // kafka.apache.org [Электронный ресурс]. URL: <https://kafka.apache.org/documentation/streams/> (дата обращения: 5.05.2023).
27. KSQLDB Docs // kafka.apache.org [Электронный ресурс]. URL: <https://kafka.apache.org/documentation/ksqldb/> (дата обращения: 5.05.2023).

28. Apache Flink, Flink Docs // flink.apache.org [Электронный ресурс]. URL: <https://flink.apache.org/> (дата обращения: 5.05.2023).
29. Apache Storm, Storm Docs // [storm.apache.org](https://storm.apache.org/releases/2.4.0/index.html) [Электронный ресурс]. URL: <https://storm.apache.org/releases/2.4.0/index.html> (дата обращения: 6.05.2023).
30. Apache Spark, Spark Docs // [spark.apache.org](https://spark.apache.org/docs/latest/) [Электронный ресурс]. URL: <https://spark.apache.org/docs/latest/> (дата обращения: 6.05.2023).
31. RocksDB, Docs // rocksdb.org [Электронный ресурс]. URL: <https://rocksdb.org/> (дата обращения: 9.05.2023).
32. Kubernetes, Docs // kubernetes.io [Электронный ресурс]. URL: <https://kubernetes.io/> (дата обращения: 9.05.2023).
33. Yarnpkg, Docs // yarnpkg.com [Электронный ресурс]. URL: <https://yarnpkg.com/> (дата обращения: 9.05.2023).
34. JVM // Wikipedia [Электронный ресурс]. URL: https://ru.wikipedia.org/wiki/Java_Virtual_Machine (дата обращения: 6.05.2023).
35. Queries in ksqlDB for Confluent Platform, Docs // docs.confluent.io [Электронный ресурс]. URL: <https://docs.confluent.io/platform/current/ksqldb/concepts/queries.html> (дата обращения: 9.05.2023).
36. StreamingResponseBody, Docs // [docs.spring.io](https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/web/servlet/mvc/method/annotation/StreamingResponseBody.html) [Электронный ресурс]. URL: <https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/web/servlet/mvc/method/annotation/StreamingResponseBody.html> (дата обращения: 9.05.2023).
37. Server sent events [Электронный ресурс]. URL: <https://learn.javascript.ru/server-sent-events> (дата обращения: 9.05.2023).
38. Causal Inference: DAG // [habr.com](https://habr.com/ru/articles/693532/) [Электронный ресурс]. URL: <https://habr.com/ru/articles/693532/> (дата обращения: 9.05.2023).

ПРИЛОЖЕНИЕ А

(обязательное)

Модели, алгоритмы и программные средства оперативного
обнаружения событий в больших потоках данных в рамках
микросервисной архитектуры
Кодовая база

Листинг А.1 – Файл DefinitionController.java

```
@RestController
@RequestMapping("/definition")
public class DefinitionController {

    private final DefinitionStatementBuilder statementBuilder;
    private final KsqlDbClient ksqlDbClient;

    @Autowired
    public DefinitionController(DefinitionStatementBuilder
statementBuilder, KsqlDbClient ksqlDbClient) {
        this.statementBuilder = statementBuilder;
        this.ksqlDbClient = ksqlDbClient;
    }

    @PostMapping("/stream")
    public ResponseEntity<?> createStream(@RequestBody Map<String,
Object> request) {
        String ksql =
statementBuilder.buildCreateStatement("STREAM", request);
        return ksqlDbClient.executeStatement(ksql);
    }

    @PostMapping("/table")
    public ResponseEntity<?> createTable(@RequestBody Map<String,
Object> request) {
        String ksql =
statementBuilder.buildCreateStatement("TABLE", request);
        return ksqlDbClient.executeStatement(ksql);
    }

    @PutMapping("/stream/{name}")
    public ResponseEntity<?> updateStream(@PathVariable String
name, @RequestBody Map<String, Object> request) {
        String ksql =
statementBuilder.buildUpdateStatement("STREAM", name, request);
        return ksqlDbClient.executeStatement(ksql);
    }

    @PutMapping("/table/{name}")
    public ResponseEntity<?> updateTable(@PathVariable String
name, @RequestBody Map<String, Object> request) {
        String ksql =
statementBuilder.buildUpdateStatement("TABLE", name, request);
        return ksqlDbClient.executeStatement(ksql);
    }

    @DeleteMapping("/stream/{name}")
    public ResponseEntity<?> deleteStream(@PathVariable String
name) {
```

```

        return ksqlDbClient.executeStatement("DROP STREAM IF
EXISTS " + name + ";");
    }

    @DeleteMapping("/table/{name}")
    public ResponseEntity<?> deleteTable(@PathVariable String
name) {
        return ksqlDbClient.executeStatement("DROP TABLE IF EXISTS
" + name + ";");
    }
}

```

Листинг A.2 – Файл DefinitionStatementBuilder.java

```

@Service
public class DefinitionStatementBuilder {

    public String buildCreateStatement(String type, Map<String,
Object> request) {
        String name = (String) request.get("name");
        String topic = (String) request.get("topic");
        String format = ((String) request.getDefault("format",
"JSON")).toUpperCase();
        List<Map<String, String>> schema = (List<Map<String,
String>>) request.get("schema");

        String fields = schema.stream()
            .map(field -> field.get("name") + " " +
field.get("type"))
            .collect(Collectors.joining(", "));

        return String.format(
            "CREATE %s %s (%s) WITH (KAFKA_TOPIC='%s',
VALUE_FORMAT='%s');",
            type.toUpperCase(),
            name,
            fields,
            topic,
            format
        );
    }

    public String buildUpdateStatement(String type, String name,
Map<String, Object> request) {
        String drop = String.format("DROP %s IF EXISTS %s;",
type.toUpperCase(), name);
        String create = buildCreateStatement(type, request);
        return drop + "\n" + create;
    }
}

```


Листинг А.3 – Файл ExecutionController.java

```
@RestController
@RequestMapping("/execution/query")
public class ExecutionController {

    private final ExecutionKsqlDbClient ksqlClient;
    private final ExecutionStatementBuilder statementBuilder;

    @Autowired
    public ExecutionController(ExecutionKsqlDbClient ksqlClient,
        ExecutionStatementBuilder statementBuilder) {
        this.ksqlClient = ksqlClient;
        this.statementBuilder = statementBuilder;
    }

    // Pull-запрос (TABLE)
    @PostMapping("/pull")
    public ResponseEntity<?> executePull(@RequestBody Map<String,
        String> request) {
        String sql = request.get("ksql");
        String normalized =
statementBuilder.preparePullQuery(sql);
        return ksqlClient.executePullQuery(normalized);
    }

    // Push-запрос (STREAM или TABLE)
    @PostMapping("/push")
    public ResponseEntity<?> executePush(@RequestBody Map<String,
        String> request) {
        String sql = request.get("ksql");
        try {
            String normalized =
statementBuilder.preparePushQuery(sql);
            // Пока без SSE – просто сообщение об успешной
регистрации запроса
            ksqlClient.executePushQuery(normalized); // можно
сохранить ссылку или логировать queryId
            return ResponseEntity.ok("Streaming query started");
        } catch (Exception e) {
            return ResponseEntity.badRequest().body("Error
starting push query: " + e.getMessage());
        }
    }

    // Валидация
    @PostMapping("/validate")
    public ResponseEntity<?> validate(@RequestBody Map<String,
        String> request) {
        String sql = request.get("ksql");
        String normalized = statementBuilder.validateSql(sql);
        return ksqlClient.validateStatement(normalized);
    }
}
```

```

    }

    // Список активных запросов
    @GetMapping("/active")
    public ResponseEntity<?> getActiveQueries() {
        return ksqlClient.listActiveQueries();
    }

    // Остановка запроса
    @DeleteMapping("/{id}")
    public ResponseEntity<?> terminateQuery(@PathVariable String
id) {
        return ksqlClient.terminateQuery(id);
    }
}

```

Листинг А.4 – Файл ExecutionStatementBuilder.java

```

@Service
public class ExecutionStatementBuilder {

    /**
     * Формирует pull-запрос к TABLE (одиночный SELECT без EMIT
    CHANGES).
     */
    public String preparePullQuery(String rawSql) {
        String sql = normalizeSql(rawSql);
        if (containsEmitChanges(sql)) {
            throw new IllegalArgumentException("Pull-запрос к
    TABLE не должен содержать 'EMIT CHANGES'");
        }
        return ensureSemicolon(sql);
    }

    /**
     * Формирует push-запрос к STREAM или TABLE (обязателен EMIT
    CHANGES).
     */
    public String preparePushQuery(String rawSql) {
        String sql = normalizeSql(rawSql);
        if (!containsEmitChanges(sql)) {
            sql = sql.replace(";", " EMIT CHANGES;");
        }
        return sql;
    }

    /**
     * Проверка и нормализация SQL для валидации.
     */
    public String validateSql(String rawSql) {
        return ensureSemicolon(normalizeSql(rawSql));
    }
}

```

```

/**
 * Генерация запроса для просмотра активных запросов.
 */
public String showActiveQueries() {
    return "SHOW QUERIES;";
}

/**
 * Генерация запроса для остановки активного запроса по ID.
 */
public String terminateQuery(String queryId) {
    return "TERMINATE " + queryId + ";";
}

private String ensureSemicolon(String sql) {
    sql = sql.trim();
    return sql.endsWith(";") ? sql : sql + ";";
}

private String normalizeSql(String sql) {
    return sql.trim().replaceAll(";", ";");
}

private boolean containsEmitChanges(String sql) {
    return sql.toUpperCase().contains("EMIT CHANGES");
}
}

```

Листинг A.5 – Файл KsqlDbClient.java

```

@Service
public class KsqlDbClient {

    private Client ksqlClient;

    @PostConstruct
    public void init() {
        this.ksqlClient = Client.create(
            ClientOptions.create()
                .setHost("localhost")
                .setPort(8088)
        );
    }

    public ResponseEntity<?> executeStatement(String ksql) {
        try {
            ExecuteStatementResult result =
                ksqlClient.executeStatement(ksql).get();
            return ResponseEntity.ok("Statement executed: " +
                result.queryId().orElse("N/A"));
        }
    }
}

```



```

        } catch (InterruptedException | ExecutionException e) {
            return ResponseEntity.badRequest().body("Error
executing statement: " + e.getMessage());
        }
    }

    public ResponseEntity<?> executePullQuery(String ksql) {
        try {
            BatchedQueryResult result =
ksqlClient.executeQuery(ksql);
            List<Row> rows = result.get();
            return ResponseEntity.ok(rows);
        } catch (InterruptedException | ExecutionException e) {
            return ResponseEntity.badRequest().body("Error
executing pull query: " + e.getMessage());
        }
    }

    public StreamedQueryResult executePushQuery(String ksql)
throws ExecutionException, InterruptedException {
        return ksqlClient.streamQuery(ksql).get();
    }

    public ResponseEntity<?> validateStatement(String ksql) {
        try {
            ksqlClient.executeStatement(ksql, options ->
options.setParseOnly(true)).get();
            return ResponseEntity.ok("Valid SQL");
        } catch (InterruptedException | ExecutionException e) {
            return ResponseEntity.badRequest().body("Invalid SQL:
" + e.getMessage());
        }
    }

    public ResponseEntity<?> listActiveQueries() {
        try {
            return
ResponseEntity.ok(ksqlClient.listQueries().get());
        } catch (InterruptedException | ExecutionException e) {
            return ResponseEntity.badRequest().body("Failed to
retrieve active queries: " + e.getMessage());
        }
    }

    public ResponseEntity<?> terminateQuery(String queryId) {
        String ksql = "TERMINATE " + queryId + ";";
        return executeStatement(ksql);
    }
}

```

Листинг А.6 – Файл docker-compose.yaml

```
version: "3.9"
name: cep

services:

    ##### BROKER #####

    broker:
        image: confluentinc/cp-kafka:latest
        container_name: "broker"
        hostname: "broker"
        ports:
            - "9092:9092"
            - "9101:9101"
        environment:
            KAFKA_NODE_ID: 1
            CLUSTER_ID: "MkU3OEVBNtCwNTJENDM2Qk"
            KAFKA_PROCESS_ROLES: "broker,controller"
            KAFKA_CONTROLLER_QUORUM_VOTERS: "1@broker:39092"

            KAFKA_CONTROLLER_LISTENER_NAMES: "INTERNAL_CONTROLLER"
            KAFKA_INTER_BROKER_LISTENER_NAME: "INTERNAL_BROKER"
            KAFKA_LISTENERS:
"EXTERNAL_BROKER://broker:9092,INTERNAL_BROKER://broker:29092,INTE
RNAL_CONTROLLER://broker:39092"
            KAFKA_ADVERTISED_LISTENERS:
"EXTERNAL_BROKER://localhost:9092,INTERNAL_BROKER://broker:29092"
            KAFKA_LISTENER_SECURITY_PROTOCOL_MAP:
"EXTERNAL_BROKER:PLAINTEXT,INTERNAL_BROKER:PLAINTEXT,INTERNAL_CONT
ROLLER:PLAINTEXT"

            KAFKA_DEFAULT_REPLICATION_FACTOR: 1 # (?)
            KAFKA_MIN_ISR: 1 # (?)
            KAFKA_AUTO_LEADER_REBALANCE_ENABLE: true
            KAFKA_UNCLEAN_LEADER_ELECTION_ENABLE: false

            KAFKA_AUTO_CREATE_TOPICS_ENABLE: true
            KAFKA_DELETE_TOPIC_ENABLE: true
            KAFKA_NUM_PARTITIONS: 1 # (?)
            KAFKA_OFFSETS_TOPIC_REPLICATION_FACTOR: 1

            KAFKA_TRANSACTION_STATE_LOG_MIN_ISR: 1 # ?
            KAFKA_TRANSACTION_STATE_LOG_REPLICATION_FACTOR: 1 # ?

            KAFKA_LOG_CLEANER_ENABLE: true
            KAFKA_LOG_CLEANUP_POLICY: "compact" # (?)

        EXTRA_ARGS:
            "_
javaagent:/usr/share/jmx_exporter/jmx_prometheus_javaagent-
```

1.0.1.jar=9101:/etc/jmx/jmx.yaml"

```
volumes:
  - ./brokers/broker/log:/var/lib/kafka/log
  - ./brokers/broker/data:/var/lib/kafka/data
  - ./config/jmx/agents/jmx_prometheus_javaagent-
1.0.1.jar:/usr/share/jmx_exporter/jmx_prometheus_javaagent-
1.0.1.jar
  - ./config/jmx/jmx.yaml:/etc/jmx/jmx.yaml
networks:
  - cep-web
```

```
schema-registry:
  image: confluentinc/cp-schema-registry:latest
  container_name: "schema-registry"
  hostname: "schema-registry"
  ports:
    - "8081:8081"
  environment:
    SCHEMA_REGISTRY_HOST_NAME: schema-registry
    SCHEMA_REGISTRY_KAFKASTORE_BOOTSTRAP_SERVERS: "broker:29092"
    SCHEMA_REGISTRY_LISTENERS: http://0.0.0.0:8081
  depends_on:
    - broker
  networks:
    - cep-web
```

```
akhq:
  image: tchiotludo/akhq:latest
  ports:
    - "8082:8082"
  environment:
    AKHQ_CONFIGURATION: |
      micronaut:
        server:
          port: 8082
      akhq:
        connections:
          infrastructure:
            properties:
              bootstrap.servers: "broker:29092"
  depends_on:
    - broker
  networks:
    - cep-web
```

KSQL

```
ksqldb-server:
  image: confluentinc/cp-ksqldb-server:7.8.0
  hostname: "ksqldb-server"
  container_name: "ksqldb-server"
  ports:
```



```

    - "8088:8088"
environment:
  KSQL_CONFIG_DIR: "/etc/ksql"
  KSQL_BOOTSTRAP_SERVERS: "broker:29092"
  KSQL_LISTENERS: "http://0.0.0.0:8088"
  KSQL_KSQL_SCHEMA_REGISTRY_URL: "http://schema-registry:8081"
  KSQL_PRODUCER_INTERCEPTOR_CLASSES:
"io.confluent.monitoring.clients.interceptor.MonitoringProducerInt
erceptor"
  KSQL_CONSUMER_INTERCEPTOR_CLASSES:
"io.confluent.monitoring.clients.interceptor.MonitoringConsumerInt
erceptor"
  KSQL_KSQL_CONNECT_URL: "http://connect:8083"
  KSQL_KSQL_LOGGING_PROCESSING_TOPIC_REPLICATION_FACTOR: 1
  KSQL_KSQL_LOGGING_PROCESSING_TOPIC_AUTO_CREATE: 'true'
  KSQL_KSQL_LOGGING_PROCESSING_STREAM_AUTO_CREATE: 'true'
depends_on:
  - broker
networks:
  - cep-web

ksqldb-cli:
  image: confluentinc/cp-ksqldb-cli:7.8.0
  container_name: "ksqldb-cli"
  entrypoint: /bin/sh
  tty: true
  depends_on:
    - ksqldb-server
  networks:
    - cep-web

##### METRICS #####

prometheus:
  image: prom/prometheus:latest
  container_name: "prometheus"
  hostname: "prometheus"
  ports:
    - "9090:9090"
  volumes:
    -
./config/prometheus/prometheus.yml:/etc/prometheus/prometheus.yml
  depends_on:
    - broker
  networks:
    - cep-web

grafana:
  image: grafana/grafana:latest
  container_name: "grafana"
  hostname: "grafana"
  ports:
    - "3000:3000"

```

```

environment:
  GF_SECURITY_ADMIN_USER: admin
  GF_SECURITY_ADMIN_PASSWORD: admin
volumes:
  - grafana_data:/var/lib/grafana
depends_on:
  - prometheus
networks:
  - cep-web

##### LOGS #####

elasticsearch:
  image: elasticsearch:7.16.1
  container_name: "elasticsearch"
  hostname: "elasticsearch"
  ports:
    - "9200:9200"
    - "9300:9300"
  environment:
    ES_JAVA_OPTS: "-Xmx512m -Xms512m"
    ELASTIC_USERNAME: "elastic"
    ELASTIC_PASSWORD: "elastic"
    discovery.type: single-node
  volumes:
    -
    ./config/elasticsearch/config.yml:/usr/share/elasticsearch/config/
    elasticsearch.yml:ro
    -
    ./docker_volumes/elasticsearch/data:/usr/share/elasticsearch/data
  networks:
    - cep-web

logstash:
  image: logstash:7.16.2
  container_name: "logstash"
  hostname: "logstash"
  ports:
    - "5046:5046"
  environment:
    LS_JAVA_OPTS: "-Xmx512m -Xms512m"
  volumes:
    -
    ./config/logstash/pipelines.yml:/usr/share/logstash/config/pipeline
    es.yml:ro
    -
    ./config/logstash/pipelines:/usr/share/logstash/config/pipelines:r
    o
    - ./log:/usr/share/logstash/input
  depends_on:
    - elasticsearch
  networks:
    - cep-web

```

```

kibana:
  image: kibana:7.16.1
  container_name: "kibana"
  hostname: "kibana"
  ports:
    - "5601:5601"
  volumes:
    -
./config/kibana/config.yml:/usr/share/kibana/config/kibana.yml:ro
  depends_on:
    - elasticsearch
  networks:
    - cep-web

##### APP #####

volumes:
  grafana_data:

networks:
  # Название сети
  cep-web:
    driver: bridge

```


ПРИЛОЖЕНИЕ Б

(обязательное)

Модели, алгоритмы и программные средства оперативного
обнаружения событий в больших потоках данных в рамках
микросервисной архитектуры

Презентация



Модели, алгоритмы и программные средства оперативного обнаружения событий в больших потоках данных в рамках микросервисной архитектуры

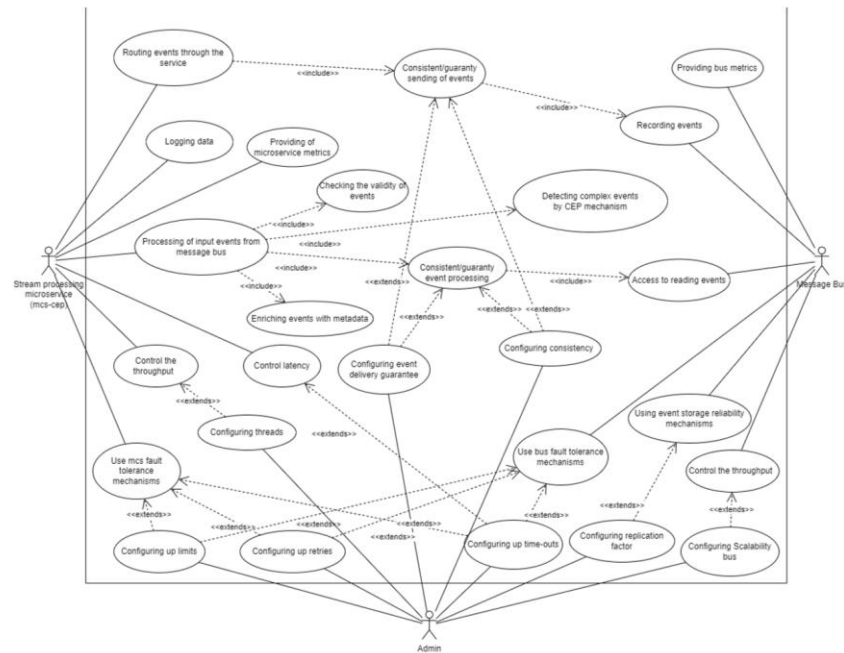
Студент: группа 23ВВИм1
Ершов Даниил Вадимович
Руководитель: к.т.н., доцент
Шибанов Сергей Владимирович

Пенза, 2025

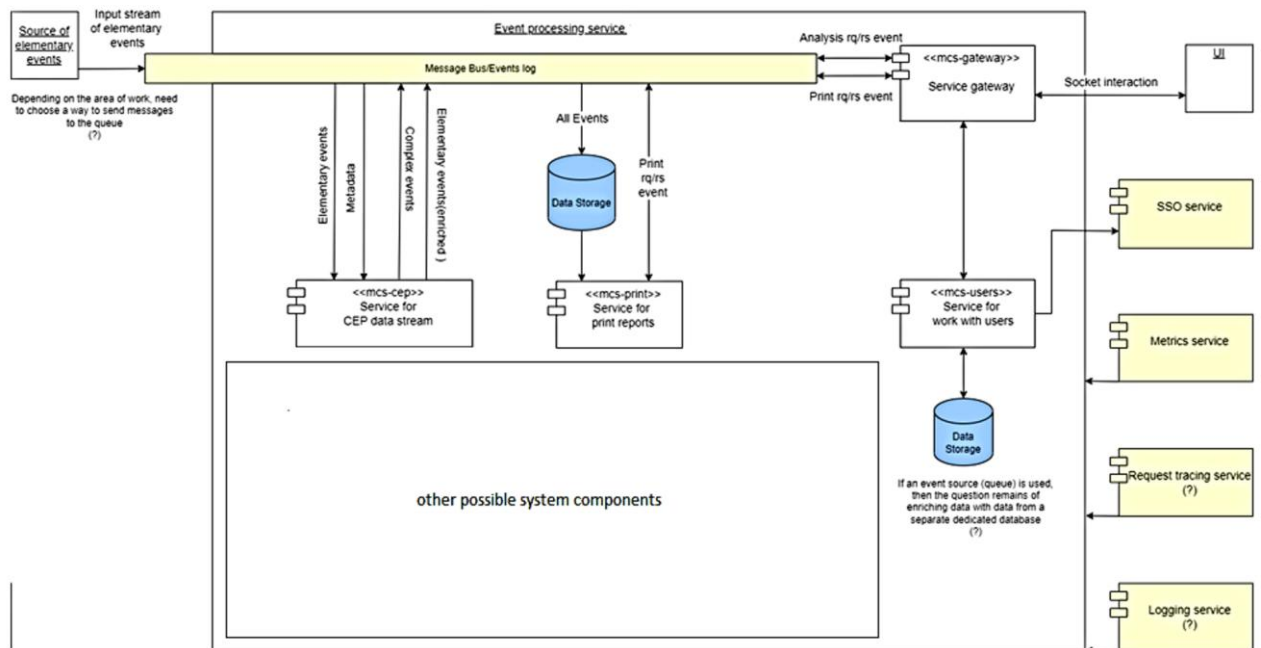
Задачи разработки

- изучить предметную область;
- провести анализ моделей и алгоритмов;
- провести анализ требований;
- спроектировать архитектуру ПО;
- выбрать подходящие технологии для реализации шины сообщений
- реализовать ПО;
- протестировать на сценарии обработки событий;

Функциональные требования



Архитектура ПО оперативного обнаружения событий в больших потоках данных



Выбор технологий

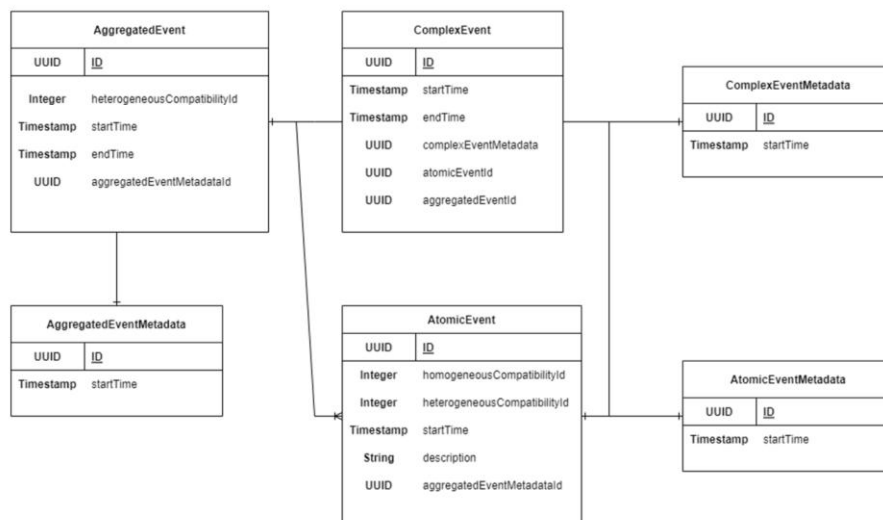
Инструменты потоковой обработки

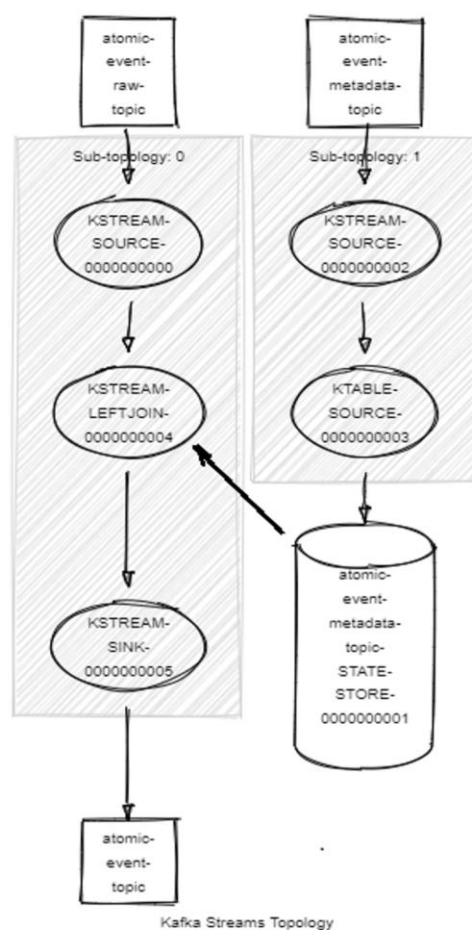
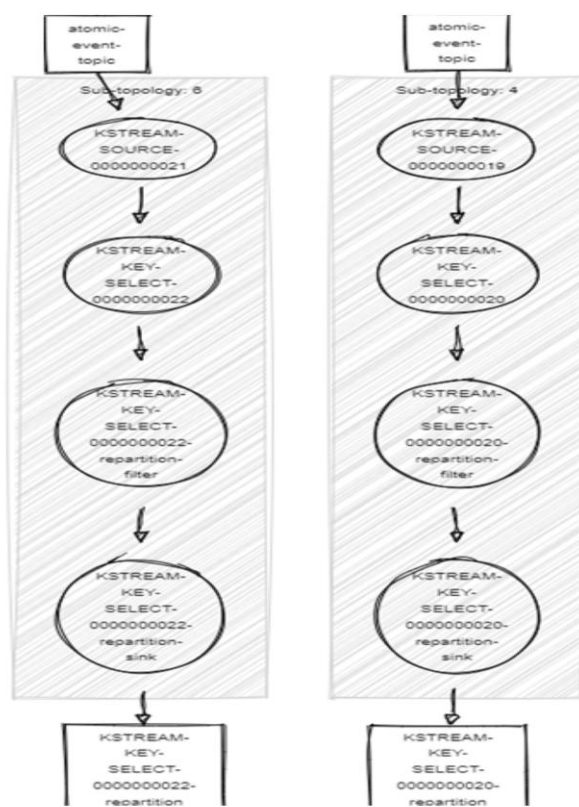


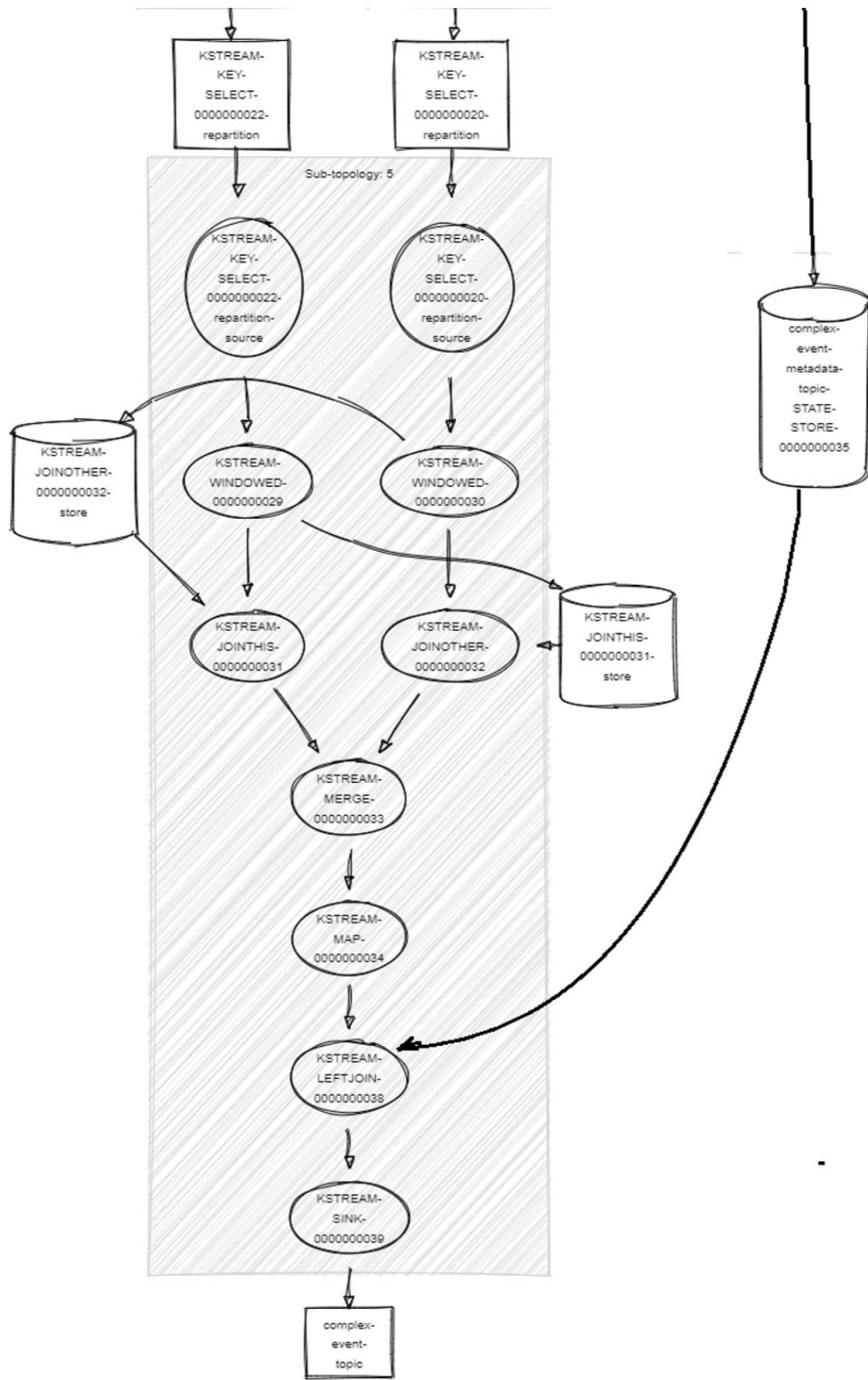
Очередь сообщений



Сценарий обработки событий







Kafka Streams Topology

Итог

- изучена предметная область;
- проведен анализ требований, предъявляемых к приложению;
- спроектирована гибкая архитектура;
- выбраны и обоснованы подходящие технологии для реализации шины сообщений
- архитектура была проверена на практике реализации;
- провести анализ качества кода;

Направления дальнейших исследований и разработок

- развитие метамодели с целью единого представления структуры и процессов управления метаданными сервиса потоковой обработки событий;
- Добавление Lambda/Kappa/Delta взаимодействия архитектур;
- Реализация машинного обучения

Спасибо за внимание!